Counting Affine Calculator and Applications

Sven Verdoolaege

Team ALCHEMY, INRIA Saclay, France
Sven.Verdoolaege@inria.fr

April 3, 2011

Outline

Outline

Introduction

- What is `iscc`?
  - ⇒ interactive interface to the `barvinok` counting library
  - ⇒ also provides interface to the `CLooG` code generation library and to some operations of the `isl` integer set library
  - ⇒ inspired by Omega Calculator from the Omega Project

Introduction

- What is `iscc`?
  - ⇒ interactive interface to the `barvinok` counting library
  - ⇒ also provides interface to the `CLooG` code generation library and to some operations of the `isl` integer set library
  - ⇒ inspired by Omega Calculator from the Omega Project
- Where to get `iscc`?
  - ⇒ currently distributed as part of the `barvinok` distribution
  - ⇒ available from `http://freshmeat.net/projects/barvinok/`

Introduction

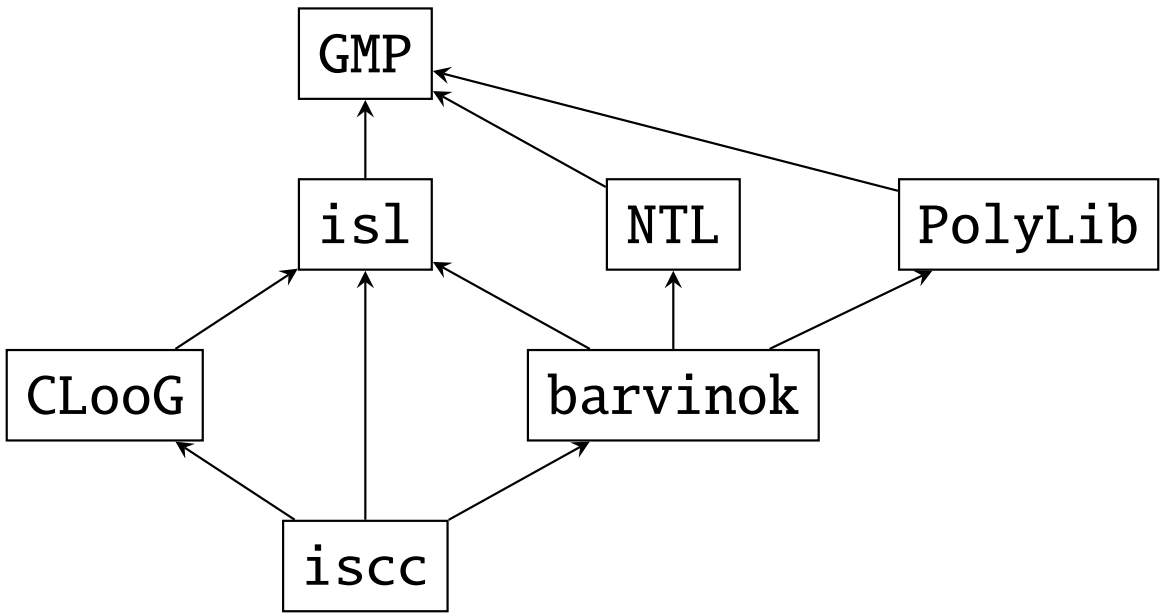
- What is `iscc`?
  - ⇒ interactive interface to the `barvinok` counting library
  - ⇒ also provides interface to the `CLooG` code generation library and to some operations of the `isl` integer set library
  - ⇒ inspired by Omega Calculator from the Omega Project
- Where to get `iscc`?
  - ⇒ currently distributed as part of the `barvinok` distribution
  - ⇒ available from `http://freshmeat.net/projects/barvinok/`
- How to run `iscc`?
  - ⇒ optionally obtain `CLooG` from `http://www.cloog.org/`
  - ⇒ 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

`isl`: manipulates parametric affine sets and relations

`barvinok`: counts elements in parametric affine sets and relations

`CLooG`: generates code to scan elements in parametric affine sets

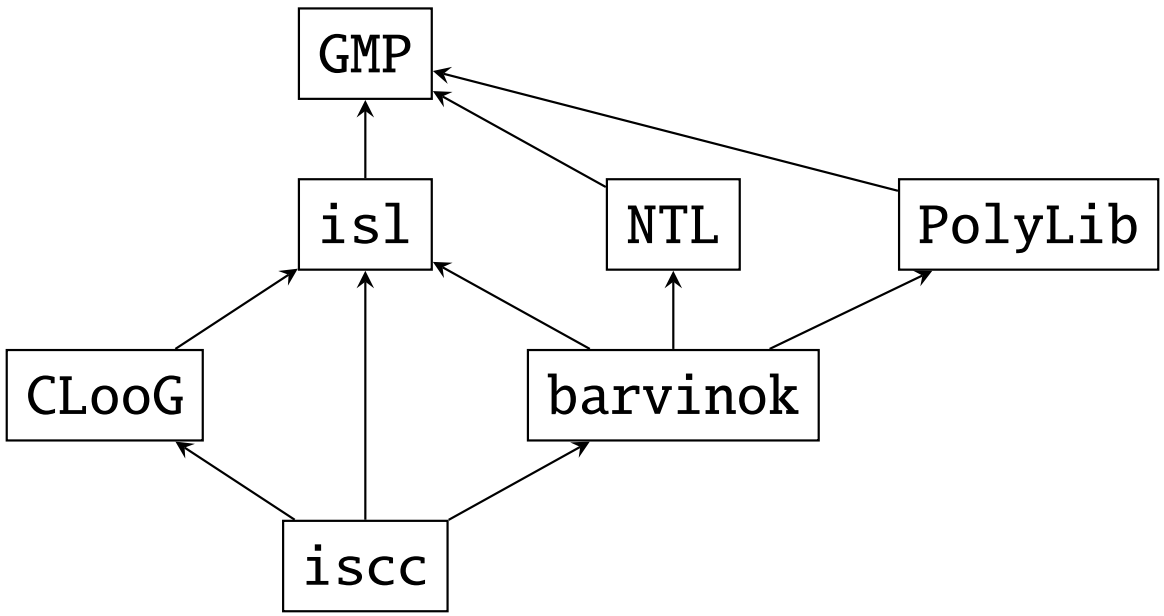


Interaction with Libraries

`isl`: manipulates parametric affine sets and relations

`barvinok`: counts elements in parametric affine sets and relations

`CLooG`: generates code to scan elements in parametric affine sets



Future work:

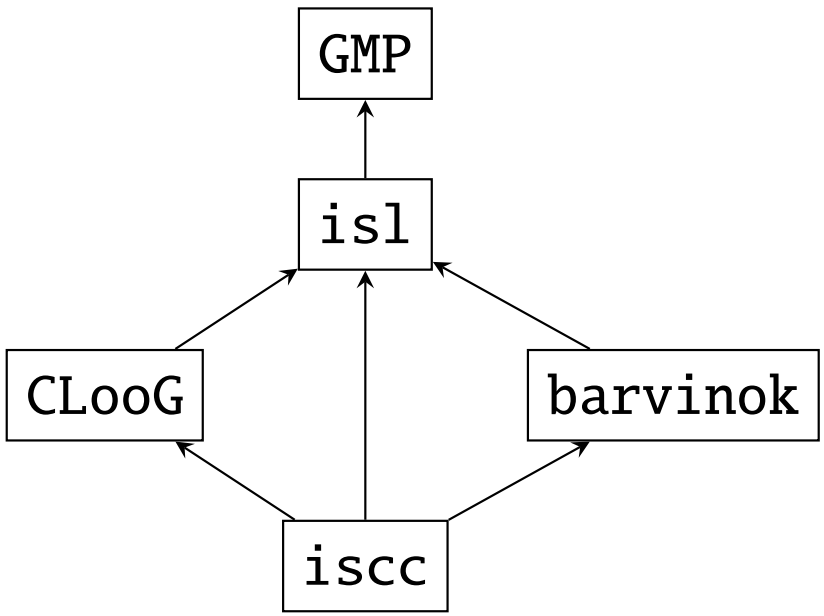
- remove dependence on `PolyLib` and `NTL`

Interaction with Libraries

`isl`: manipulates parametric affine sets and relations

`barvinok`: counts elements in parametric affine sets and relations

`CLooG`: generates code to scan elements in parametric affine sets



Future work:

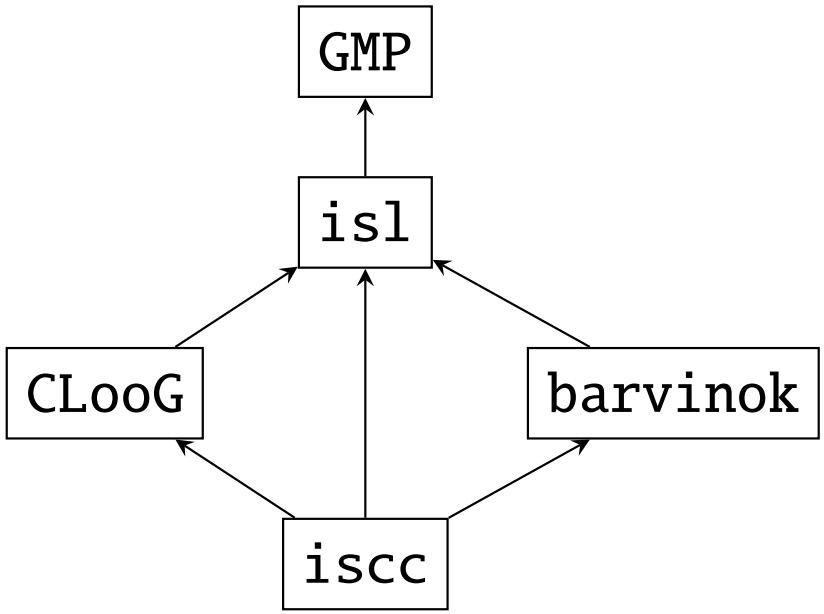
- remove dependence on `PolyLib` and `NTL`

Interaction with Libraries

`isl`: manipulates parametric affine sets and relations

`barvinok`: counts elements in parametric affine sets and relations

`CLooG`: generates code to scan elements in parametric affine sets



Future work:

- remove dependence on `PolyLib` and `NTL`
- merge `barvinok` into `isl`

Outline

Representation

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]);
```

Representation

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]);
```

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Representation

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]);
```

parameters

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Representation

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]);
```

parameters

(optional) name of space

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Representation

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]);
```

parameters (optional) name of space disjunction

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Representation

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]);
```

parameters (optional) name of space disjunction

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Read accesses:

```
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
```

Representation

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]);
```

parameters (optional) name of space disjunction

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Read accesses: intersect domain of map on the left with set on the right

```
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
```

Representation

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]);
```

parameters

(optional) name of space

disjunction

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Read accesses:

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

```
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
```

Write accesses:

```
W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;
```

Representation

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]);
```

parameters

(optional) name of space

disjunction

Iteration domains:

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
```

Read accesses:

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

```
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
```

Write accesses:

```
W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;
```

Schedule:

```
S := { S1[i] -> [0,i]; S2[i] -> [1,i] };
```

Dataflow Analysis

```
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]);
```

Dataflow Analysis

```
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]);
```

- individual pair of accesses

```
A1:=[N] -> { S1[i] -> t[i] : 0 <= i < N };
A2:=[N] -> { S2[i] -> t[N-i-1] : 0 <= i < N };
```

Dataflow Analysis

```
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]);
```

- individual pair of accesses

  ```
  A1:=[N] -> { S1[i] -> t[i] : 0 <= i < N };
  A2:=[N] -> { S2[i] -> t[N-i-1] : 0 <= i < N };
  ```

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

Dataflow Analysis

```
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]);
```

- individual pair of accesses

  ```
  A1:=[N] -> { S1[i] -> t[i] : 0 <= i < N };
  A2:=[N] -> { S2[i] -> t[N-i-1] : 0 <= i < N };
  ```

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

  Last write: `lexmax R;`

Dataflow Analysis

```
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]);
```

- individual pair of accesses

  ```
  A1:=[N] -> { S1[i] -> t[i] : 0 <= i < N };
  A2:=[N] -> { S2[i] -> t[N-i-1] : 0 <= i < N };
  ```

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

  Last write: `lexmax R;`

- globally

  ```
  D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
  R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
  W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;
  S := { S1[i] -> [0,i]; S2[i] -> [1,i] };
  last W before R under S;
  ```

Code Generation

```
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]);
```

- Original schedule

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
S := { S1[i] -> [0,i]; S2[i] -> [1,i] };
codegen (S * D);
```

Code Generation

```
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]);
```

- Original schedule

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
S := { S1[i] -> [0,i]; S2[i] -> [1,i] };
codegen (S * D);
```

- Alternative schedule

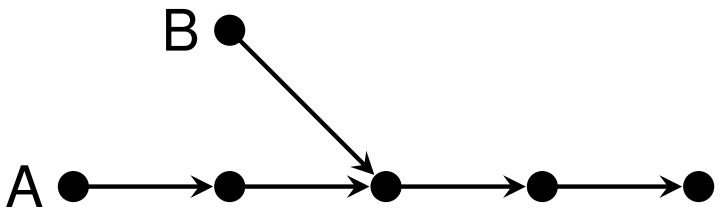
```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
S2 := [N] -> { S1[i] -> [i,0]; S2[i] -> [N-i-1,1] };
codegen (S2 * D);
```

Outline

Transitive Closures

Given a directed graph (represented as an affine map)

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

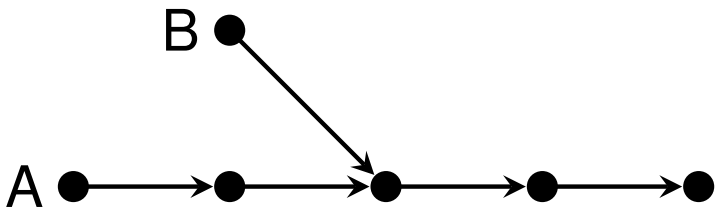


What are the paths in the graph (transitive closure of map)?

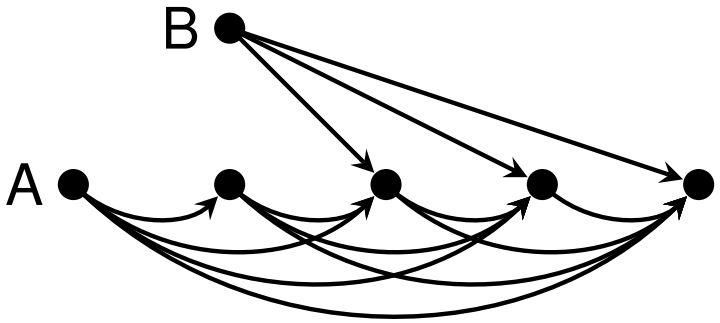
Transitive Closures

Given a directed graph (represented as an affine map)

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



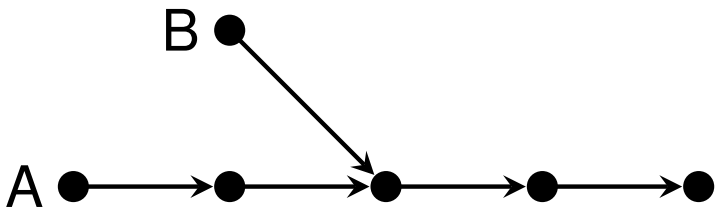
What are the paths in the graph (transitive closure of map)? $\Rightarrow$ `M^+;`



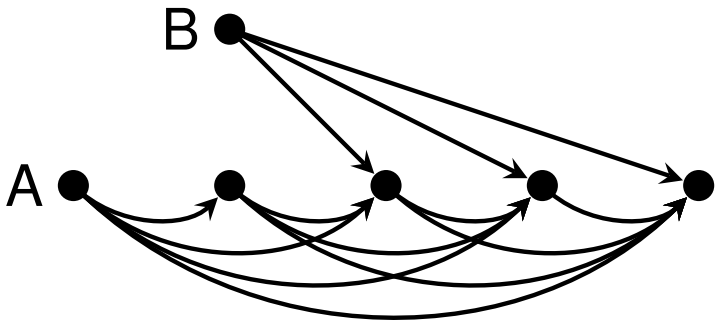
Transitive Closures

Given a directed graph (represented as an affine map)

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



What are the paths in the graph (transitive closure of map)? $\Rightarrow$ `M^+;`



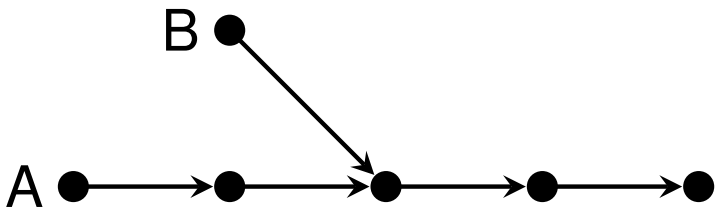
Result:

```
({ 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)
```

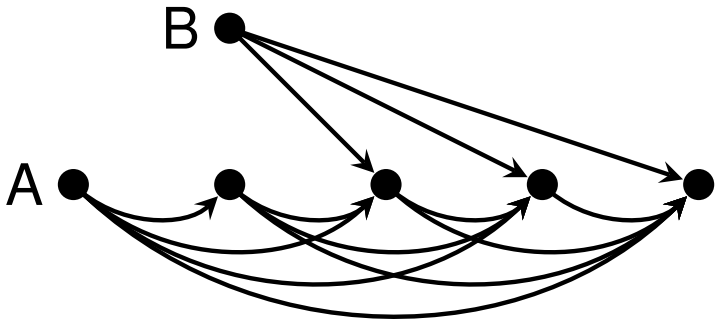
Transitive Closures

Given a directed graph (represented as an affine map)

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



What are the paths in the graph (transitive closure of map)? $\Rightarrow$ `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);
```

Outline

Maximal Number of Live Memory elements

```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```

Maximal Number of Live Memory elements

```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```

```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->
D := dom S;
```

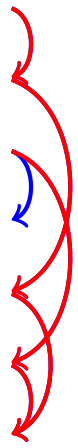
Maximal Number of Live Memory elements

```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```

```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->
D := dom S;
Dep := (last W before R under S)[0];
```

Maximal Number of Live Memory elements

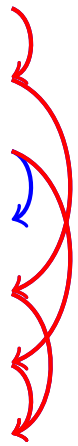
```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```



```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->
D := dom S;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
```

Maximal Number of Live Memory elements

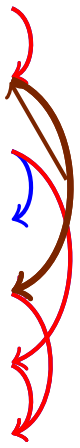
```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```



```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->[
D := dom S;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
```

Maximal Number of Live Memory elements

```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```



```
domain_map { S2[] -> S5[] };

{ [S2[] -> S5[]] -> S2[] }
```

```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->
D := dom S;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
```

Maximal Number of Live Memory elements

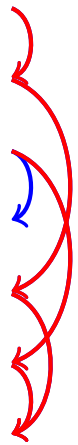
```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```

```
domain_map { S2[] -> S5[] };

{ [S2[] -> S5[]] -> S2[] }
```

```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->
D := dom S;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
```

Maximal Number of Live Memory elements

```
S1: A = 0;
S2: B = A;
S3: A = 1;
S4: C = A;
S5: C = B;
S6: B = A;
S7: f(B,C);
```



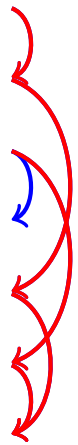
```
domain_map { S2[] -> S5[] };

{ [S2[] -> S5[]] -> S2[] }
```

```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->
D := dom S;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
Before_Read := range_map(LR) . LGE;
N_Live := card ((After_Write * Before_Read)^-1);
```

number of image elements

Maximal Number of Live Memory elements

```
S1: A = 0;        0
S2: B = A;        1
S3: A = 1;        1
S4: C = A;        2
S5: C = B;        2
S6: B = A;        2
S7: f(B,C);       2
```



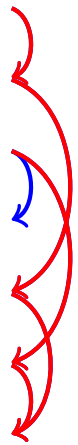
```
domain_map { S2[] -> S5[] };

{ [S2[] -> S5[]] -> S2[] }
```

```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->[
D := dom S;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
Before_Read := range_map(LR) . LGE;
N_Live := card ((After_Write * Before_Read)^-1);
```

number of image elements

Maximal Number of Live Memory elements

```
S1: A = 0;        0
S2: B = A;        1
S3: A = 1;        1
S4: C = A;        2
S5: C = B;        2
S6: B = A;        2
S7: f(B,C);       2
```



```
domain_map { S2[] -> S5[] };

{ [S2[] -> S5[]] -> S2[] }
```

```
W:={S1[]->A[];S2[]->B[];S3[]->A[];S4[]->C[];S5[]->C[];S6[]->B[]};
R:={S2[]->A[];S4[]->A[];S5[]->B[];S6[]->A[];S7[]->B[];S7[]->C[]};
S:={S1[]->[1];S2[]->[2];S3[]->[3];S4[]->[4];S5[]->[5];S6[]->[6];S7[]->
D := dom S;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
Before_Read := range_map(LR) . LGE;
N_Live := card ((After_Write * Before_Read)^-1);
ub N_Live;
```

number of image elements

Maximal Number of Live Memory elements

```
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]);
```

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;
S := { S1[i] -> [0,i]; S2[i] -> [1,i] } * D;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
Before_Read := range_map(LR) . LGE;
N_Live := card ((After_Write * Before_Read)^-1);
ub N_Live;
```

Maximal Number of Live Memory elements

```
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]);
```

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;
S := { S1[i] -> [0,i]; S2[i] -> [1,i] } * D;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
Before_Read := range_map(LR) . LGE;
N_Live := card ((After_Write * Before_Read)^-1);
ub N_Live;
```

Result:

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

live2,live3

Maximal Number of Live Memory elements

```
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]);
```

```
D := [N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N };
R := [N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] } * D;
W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;
S := { S1[i] -> [0,i]; S2[i] -> [1,i] } * D;
Dep := (last W before R under S)[0];
LR := (lexmax (Dep . S)) . S^-1;
LLT := S << S; LGE := S >>= S;
After_Write := domain_map(LR) . LLT;
Before_Read := range_map(LR) . LGE;
N_Live := card ((After_Write * Before_Read)^-1);
ub N_Live;
```

Result:

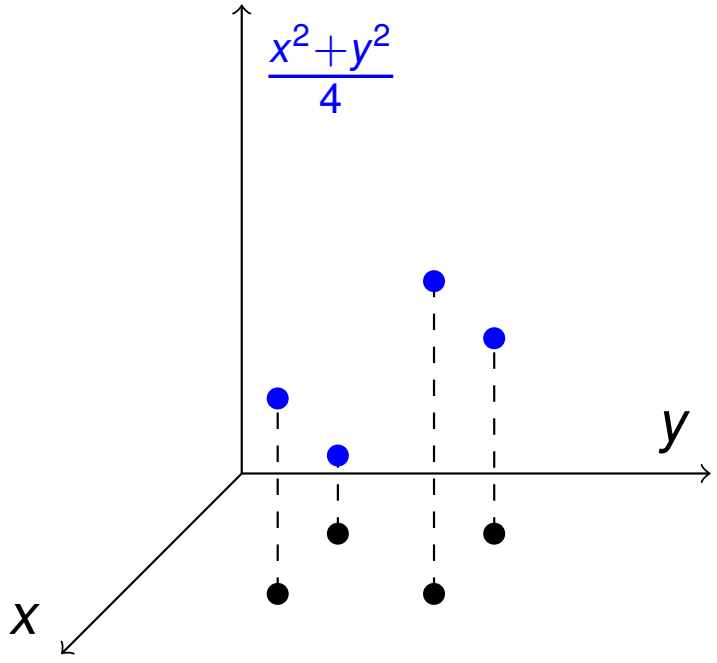
```
([N] -> { max(N) : N >= 2; max(N) : N = 1 }, True)
```

bound is tight

live2,live3

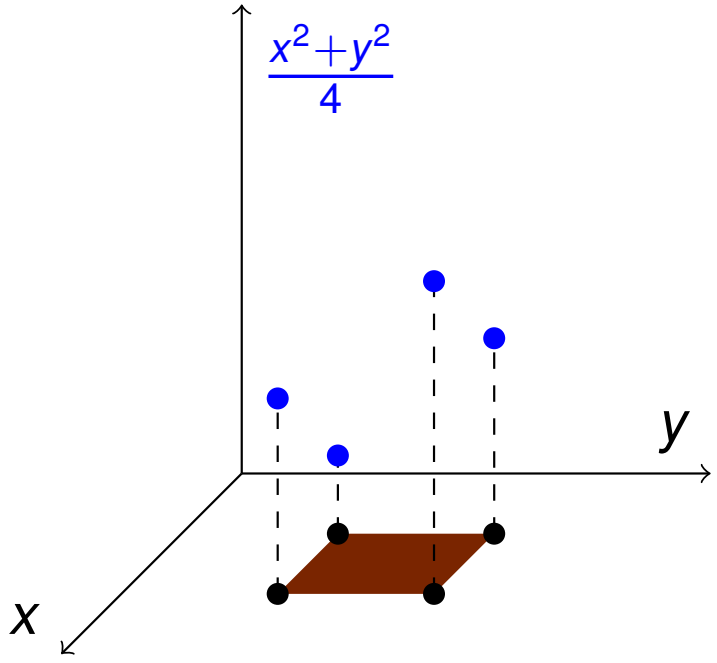
Outline

Weighted Counting



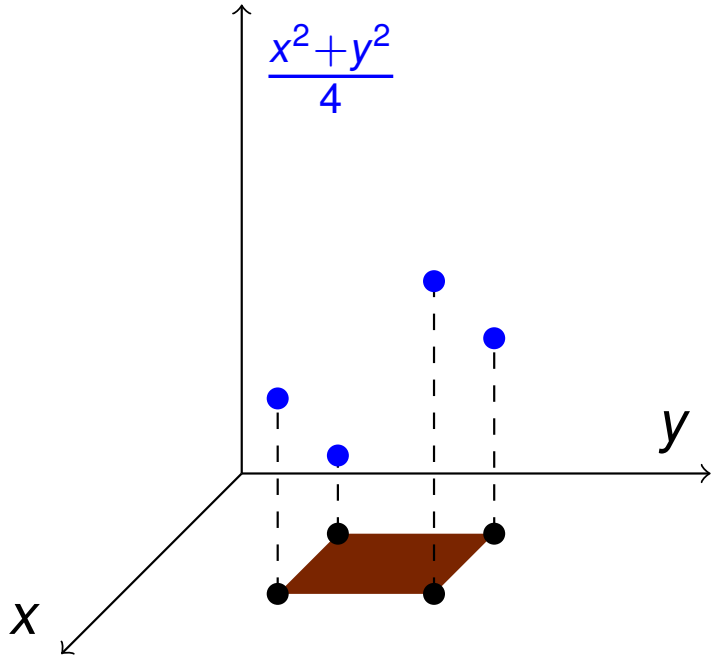

```
F := { [x,y] -> 1/4*x^2+1/4*y^2 : 1 <= x,y <= 2 };
```

Weighted Counting




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

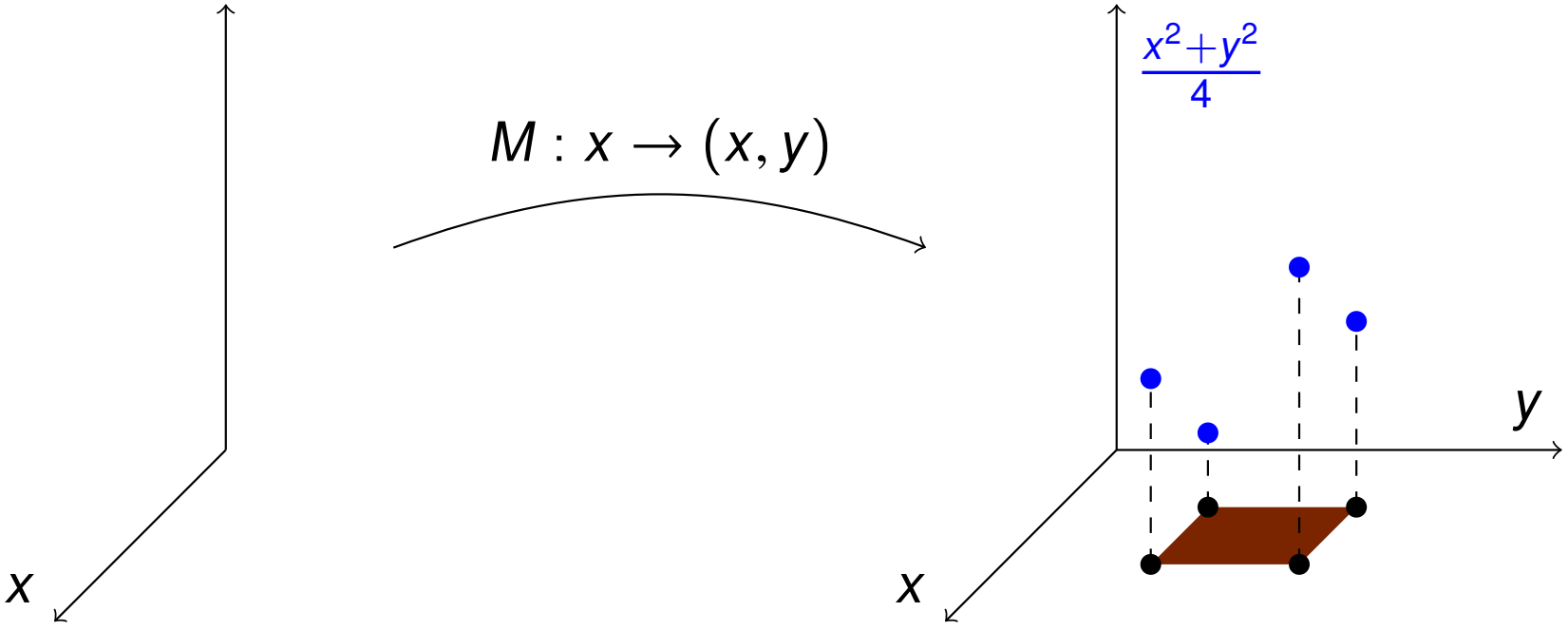
Weighted Counting




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

$\Rightarrow$ sum of `F` over points in `D`

Weighted Counting

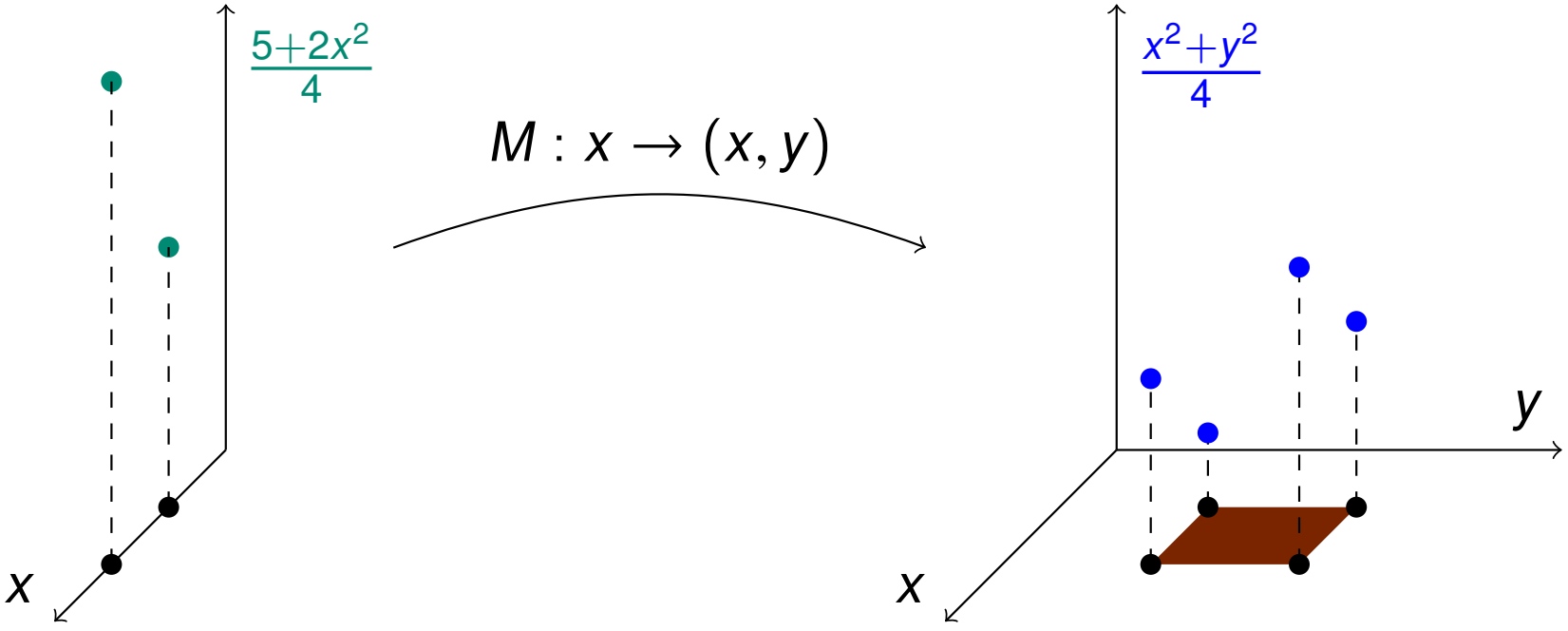



```
F := { [x,y] -> 1/4*x^2+1/4*y^2 : 1 <= x,y <= 2 };
D := dom F;
F(D);
```

$\Rightarrow$ 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

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
};
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 $\mathtt{m}$

$\mathrm{cap}_{\mathtt{m}}$ size of memory "captured" (not returned) by $\mathtt{m}$

$\mathrm{req}_{\mathtt{m}}$ total memory requirements of $\mathtt{m}$

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

Dynamic Memory Requirement Estimation [CFGV2006]

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

ret_m size of memory returned by m

cap_m size of memory "captured" (not returned) by m

req_m total memory requirements of m

$$\text{req}_m = \text{cap}_m + \max_{p \text{ called by } m} \text{req}_p$$

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

Dynamic Memory Requirement Estimation [CFGV2006]

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

ret_m size of memory returned by m

cap_m size of memory "captured" (not returned) by m

req_m total memory requirements of m

$$\mathrm{req}_m = \mathrm{cap}_m + \max_{p \text{ called by } m} \mathrm{req}_p$$

```
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) };
```

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 */
    }
}
```

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

`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`

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 */
    }
}
```

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

`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`

$\Rightarrow$ 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 */
    }
}
```

$$\mathrm{req_m} = \mathrm{cap_m} + \max_{\text{p called by m}} \mathrm{req_p}$$

```
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