

APPLICATIONS OF STATIC SLICING IN COST ANALYSIS OF JAVA BYTECODE

Samir Genaim

CLIP, UNIVERSIDAD POLITÉCNICA DE MADRID
DSIC, UNIVERSIDAD COMPLUTENSE DE MADRID

Joint work with E. Albert, P. Arenas, G. Puebla, and D. Zanardini

What is the aim of Cost Analysis?

- Two important key features of a program are:
 - correctness
 - efficiency, i.e., the cost of program execution in terms of:
 - time
 - memory
 - billable events
- *Cost Analysis* is the automatic study of *program efficiency*.
- Cost analysis has been studied for:
 - Logic Programs
 - Functional Programs
 - Imperative Programs
- And recently we have developed a cost analysis of Java bytecode:
 - For mobile code, we do not have access to source code
 - We can use cost analysis to accept/reject mobile code

Cost analysis of Java bytecode generates a **Cost Equation System** (CES) that defines the cost of the program as a function of its input (abstract) values.

Cost analysis of Java bytecode generates a **Cost Equation System** (CES) that defines the cost of the program as a function of its input (abstract) values.

```
int sum(int,int) {
  0:  iconst_0      18:  iload_2
  1:  istore_2     19:  iload_3
  2:  iconst_1     20:  iload  4
  3:  istore_3     22:  imul
  4:  iload_3      23:  iadd
  5:  iload_0      24:  istore_2
  6:  if_icmpgt 37  25:  iinc 4, 1
  9:  iload_3      28:  goto 12
 12:  istore 4      31:  iinc 3, 1
 14:  iload 4       34:  goto 4
 15:  iload_1      37:  iload_2
 15:  if_icmpgt 31  38:  ireturn
}
```

Cost analysis of Java bytecode generates a **Cost Equation System** (CES) that defines the cost of the program as a function of its input (abstract) values.

```
int sum(int,int) {
0:  iconst_0      18:  iload_2
1:  istore_2      19:  iload_3
2:  iconst_1      20:  iload 4
3:  istore_3      22:  imul
4:  iload_3        23:  iadd
5:  iload_0        24:  istore_2
6:  if_icmpgt 37  25:  iinc 4, 1
9:  iload_3        28:  goto 12
10: istore 4       31:  iinc 3, 1
12: iload 4        34:  goto 4
14: iload_1        37:  iload_2
15: if_icmpgt 31  38:  ireturn
}
```

$sum(m, n) = sum_0(m, n, res, i, j)$	$\{ res=0, i=0, j=0 \}$
$sum_0(m, n, res, i, j) = 6 + sum_1(m, n, res', i', j)$	$\{ res'=0, i'=1 \}$
$sum_1(m, n, res, i, j) = 3 + sum_1^C(m, n, res, i, j, s_0, s_1)$	$\{ s_0=i, s_1=m \}$
$sum_1^C(m, n, res, i, j, s_0, s_1) = sum_2(m, n, res, i, j)$	$\{ s_0 \leq s_1 \}$
$sum_1^C(m, n, res, i, j, s_0, s_1) = 0$	$\{ s_0 > s_1 \}$
$sum_2(m, n, res, i, j) = 4 + sum_3(m, n, res, i, j')$	
$\quad + sum_1(m, n, res', i', j'')$	$\{ j'=i, i'=i+1 \}$
$sum_3(m, n, res, i, j) = 3 + sum_3^C(m, n, res, i, j, s_0, s_1)$	$\{ s_0=j, s_1=n \}$
$sum_3^C(m, n, res, i, j, s_0, s_1) = sum_4(m, n, res, i, j)$	$\{ s_0 \leq s_1 \}$
$sum_3^C(m, n, res, i, j, s_0, s_1) = 0$	$\{ s_0 > s_1 \}$
$sum_4(m, n, res, i, j) = 10 + sum_3(m, n, res, i, j')$	$\{ j'=j+1 \}$

Cost analysis of Java bytecode generates a **Cost Equation System** (CES) that defines the cost of the program as a function of its input (abstract) values.

```
int sum(int,int) {
  0:  iconst_0      18:  iload_2
  1:  istore_2     19:  iload_3
  2:  iconst_1     20:  iload  4
  3:  istore_3     22:  imul
  4:  iload_3     23:  iadd
  5:  iload_0     24:  istore_2
  6:  if_icmpgt 37 25:  iinc 4, 1
  9:  iload_3     28:  goto 12
 10:  istore 4     31:  iinc 3, 1
 12:  iload 4     34:  goto 4
 14:  iload_1     37:  iload_2
 15:  if_icmpgt 31 38:  ireturn
}
```

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n, \text{res}, i, j) && \{ \text{res}=0, i=0, j=0 \} \\ \text{sum}_0(m, n, \text{res}, i, j) &= 6 + \text{sum}_1(m, n, \text{res}', i', j) && \{ \text{res}'=0, i'=1 \} \end{aligned}$$
$$\begin{aligned} \text{sum}_1(m, n, \text{res}, i, j) &= 3 + \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=i, s_1=m \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_2(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_2(m, n, \text{res}, i, j) &= 4 + \text{sum}_3(m, n, \text{res}, i, j') \\ &\quad + \text{sum}_1(m, n, \text{res}', i', j'') && \{ j'=i, i'=i+1 \} \end{aligned}$$
$$\begin{aligned} \text{sum}_3(m, n, \text{res}, i, j) &= 3 + \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=j, s_1=n \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_4(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_4(m, n, \text{res}, i, j) &= 10 + \text{sum}_3(m, n, \text{res}, i, j') && \{ j'=j+1 \} \end{aligned}$$

The CES is not useful unless we solve it and obtain a *closed form* upper bound. E.g $\text{sum}(m, n) = \mathbf{O}(m * n)$.

Cost analysis of Java bytecode generates a **Cost Equation System** (CES) that defines the cost of the program as a function of its input (abstract) values.

```
int sum(int, int) {
  0:  iconst_0      18:  iload_2
  1:  istore_2     19:  iload_3
  2:  iconst_1     20:  iload_4
  3:  istore_3     22:  imul
  4:  iload_3      23:  iadd
  5:  iload_0      24:  istore_2
  6:  if_icmpgt 37  25:  iinc 4, 1
  9:  iload_3      28:  goto 12
 10:  istore_4     31:  iinc 3, 1
 12:  iload_4      34:  goto 4
 14:  iload_1      37:  iload_2
 15:  if_icmpgt 31  38:  ireturn
}
```

$sum(m, n) = sum_0(m, n, res, i, j)$	$\{ res=0, i=0, j=0 \}$
$sum_0(m, n, res, i, j) = 6 + sum_1(m, n, res', i', j)$	$\{ res'=0, i'=1 \}$
$sum_1(m, n, res, i, j) = 3 + sum_1^c(m, n, res, i, j, s_0, s_1)$	$\{ s_0=i, s_1=m \}$
$sum_1^c(m, n, res, i, j, s_0, s_1) = sum_2(m, n, res, i, j)$	$\{ s_0 \leq s_1 \}$
$sum_1^c(m, n, res, i, j, s_0, s_1) = 0$	$\{ s_0 > s_1 \}$
$sum_2(m, n, res, i, j) = 4 + sum_3(m, n, res, i, j')$ $+ sum_1(m, n, res', i', j')$	$\{ j'=i, i'=i+1 \}$
$sum_3(m, n, res, i, j) = 3 + sum_3^c(m, n, res, i, j, s_0, s_1)$	$\{ s_0=j, s_1=n \}$
$sum_3^c(m, n, res, i, j, s_0, s_1) = sum_4(m, n, res, i, j)$	$\{ s_0 \leq s_1 \}$
$sum_3^c(m, n, res, i, j, s_0, s_1) = 0$	$\{ s_0 > s_1 \}$
$sum_4(m, n, res, i, j) = 10 + sum_3(m, n, res, i, j')$	$\{ j'=j+1 \}$

Solving such equations can be done by *computer algebra systems* such as Mathematica and Maple.

Cost analysis of Java bytecode generates a **Cost Equation System** (CES) that defines the cost of the program as a function of its input (abstract) values.

<pre>int sum(int,int) { 0: iconst_0 18: iload_2 1: istore_2 19: iload_3 2: iconst_1 20: iload_4 3: istore_3 22: imul 4: iload_3 23: iadd 5: iload_0 24: istore_2 6: if_icmpgt 37 25: iinc 4, 1 9: iload_3 28: goto 12 10: istore 4 31: iinc 3, 1 12: iload 4 34: goto 4 14: iload_1 37: iload_2 15: if_icmpgt 31 38: ireturn }</pre>	<pre>sum(m, n)=sum₀(m, n, res, i, j) sum₀(m, n, res, i, j)=6 + sum₁(m, n, res', i', j) sum₁(m, n, res, i, j)=3 + sum₁^c(m, n, res, i, j, s₀, s₁) sum₁^c(m, n, res, i, j, s₀, s₁)=sum₂(m, n, res, i, j) sum₁^c(m, n, res, i, j, s₀, s₁)=0 sum₂(m, n, res, i, j)=4 + sum₃(m, n, res, i, j') + sum₁(m, n, res', i', j'') sum₃(m, n, res, i, j)=3 + sum₃^c(m, n, res, i, j, s₀, s₁) sum₃^c(m, n, res, i, j, s₀, s₁)=sum₄(m, n, res, i, j) sum₃^c(m, n, res, i, j, s₀, s₁)=0 sum₄(m, n, res, i, j)=10 + sum₃(m, n, res, i, j')</pre>	<pre>{ res=0, i=0, j=0 } { res'=0, i'=1 } { s₀=i, s₁=m } { s₀≤s₁ } { s₀>s₁ } { j'=i, i'=i + 1 } { s₀=j, s₁=n } { s₀≤s₁ } { s₀>s₁ } { j'=j + 1 }</pre>
--	---	--

But these equations are not even a valid input for Mathematica or Maple, therefore a series of transformation should be applied first.

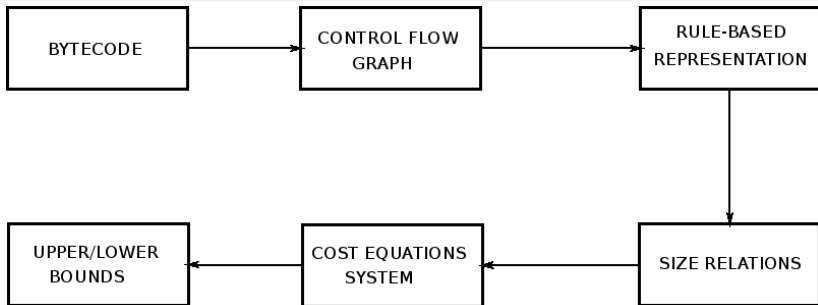
Cost analysis of Java bytecode generates a **Cost Equation System** (CES) that defines the cost of the program as a function of its input (abstract) values.

```
int sum(int,int) {
  0:  iconst_0      18:  iload_2
  1:  istore_2     19:  iload_3
  2:  iconst_1     20:  iload  4
  3:  istore_3     22:  imul
  4:  iload_3      23:  iadd
  5:  iload_0      24:  istore_2
  6:  if_icmpgt 37  25:  iinc 4, 1
  9:  iload_3      28:  goto 12
 10:  istore 4      31:  iinc 3, 1
 12:  iload 4       34:  goto 4
 14:  iload_1      37:  iload_2
 15:  if_icmpgt 31  38:  ireturn
}
```

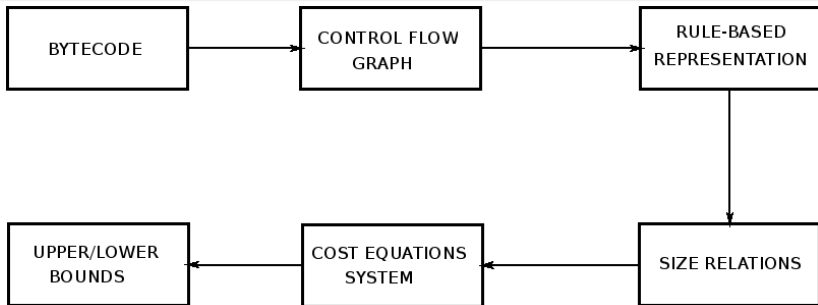
$sum(m, n) = sum_0(m, n, res, i, j)$	$\{ res=0, i=0, j=0 \}$
$sum_0(m, n, res, i, j) = 6 + sum_1(m, n, res', i', j)$	$\{ res'=0, i'=1 \}$
$sum_1(m, n, res, i, j) = 3 + sum_1^c(m, n, res, i, j, s_0, s_1)$	$\{ s_0=i, s_1=m \}$
$sum_1^c(m, n, res, i, j, s_0, s_1) = sum_2(m, n, res, i, j)$	$\{ s_0 \leq s_1 \}$
$sum_1^c(m, n, res, i, j, s_0, s_1) = 0$	$\{ s_0 > s_1 \}$
$sum_2(m, n, res, i, j) = 4 + sum_3(m, n, res, i, j')$ $+ sum_1(m, n, res', i', j'')$	$\{ j'=i, i'=i+1 \}$
$sum_3(m, n, res, i, j) = 3 + sum_3^c(m, n, res, i, j, s_0, s_1)$	$\{ s_0=j, s_1=n \}$
$sum_3^c(m, n, res, i, j, s_0, s_1) = sum_4(m, n, res, i, j)$	$\{ s_0 \leq s_1 \}$
$sum_3^c(m, n, res, i, j, s_0, s_1) = 0$	$\{ s_0 > s_1 \}$
$sum_4(m, n, res, i, j) = 10 + sum_3(m, n, res, i, j')$	$\{ j'=j+1 \}$

A crucial transformation is *removing all variables which are not relevant* to the cost.

Cost Analysis Components

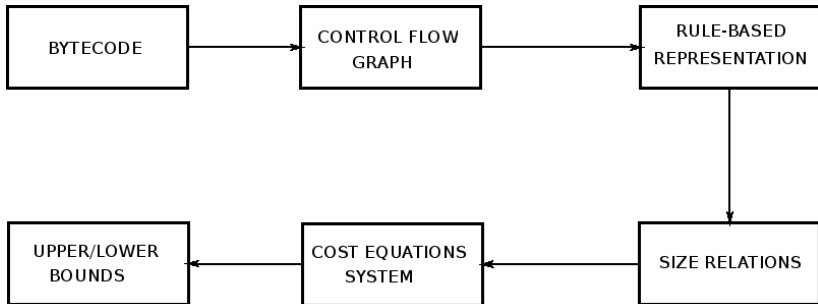


Cost Analysis Components



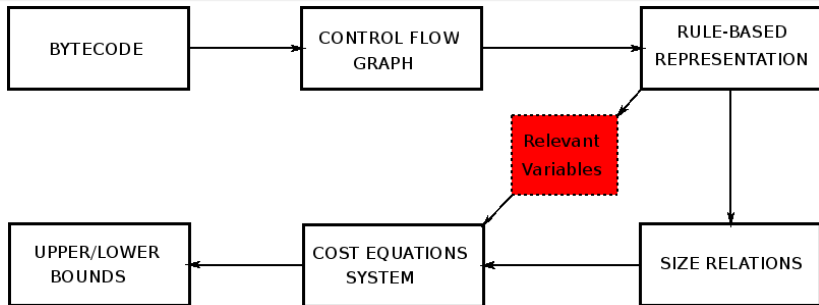
- Several sources for branching: conditions, virtual methods, exceptions.
- Unstructured control flow: use of goto.

Cost Analysis Components



- Several sources for branching: conditions, virtual methods, exceptions.
- Unstructured control flow: use of goto.
- Loops are in different forms
- JVM is a stack-based abstract machine.

Cost Analysis Components



- Several sources for branching: conditions, virtual methods, exceptions.
- Unstructured control flow: use of goto.
- Loops are in different forms
- JVM is a stack-based abstract machine.
- Identifying program variables that are relevant to the cost and eliminate the others from the cost equations.

Recovering the Structure of a Java Bytecode Program

Java

```
class a {  
    static int sum(int m, int n) {  
        int res=0;  
  
        for (int i=1; i<=m; i++) {  
            for (int j=i; j<=n; j++) {  
                res += i*j;  
            }  
        }  
  
        return res;  
    }  
}
```

Java Bytecode

```
int sum(int,int) {  
    0:  iconst_0           | 18:  iload_2  
    1:  istore_2          | 19:  iload_3  
    2:  iconst_1         | 20:  iload 4  
    3:  istore_3         | 22:  imul  
    4:  iload_3          | 23:  iadd  
    5:  iload_0          | 24:  istore_2  
    6:  if_icmpgt 37     | 25:  iinc 4, 1  
    9:  iload_3          | 28:  goto 12  
   10:  istore 4         | 31:  iinc 3, 1  
   12:  iload 4          | 34:  goto 4  
   14:  iload_1         | 37:  iload_2  
   15:  if_icmpgt 31    | 38:  ireturn  
  
}
```

Recovering the Structure of a Java Bytecode Program

Java

```
class a {  
    static int sum(int m, int n) {  
        int res=0;  
  
        for (int i=1; i<=m; i++) {  
            for (int j=i; j<=n; j++) {  
                res += i*j;  
            }  
        }  
  
        return res;  
    }  
}
```

Java Bytecode

```
int sum(int,int) {  
    0:  iconst_0      | 18:  iload_2  
    1:  istore_2     | 19:  iload_3  
    2:  iconst_1     | 20:  iload 4  
    3:  istore_3     | 22:  imul  
    4:  iload_3      | 23:  iadd  
    5:  iload_0      | 24:  istore_2  
    6:  if_icmpgt 37 | 25:  iinc 4, 1  
    9:  iload_3      | 28:  goto 12  
   10:  istore_4     | 31:  iinc 3, 1  
   12:  iload 4      | 34:  goto 4  
   14:  iload_1      | 37:  iload_2  
   15:  if_icmpgt 31 | 38:  ireturn  
  
}
```

Recovering the Structure of a Java Bytecode Program

Java

```
class a {  
    static int sum(int m, int n) {  
        int res=0;  
  
        for (int i=1; i<=m; i++) {  
            for (int j=i; j<=n; j++) {  
                res += i*j;  
            }  
        }  
  
        return res;  
    }  
}
```

Java Bytecode

```
int sum(int,int) {  
    0:  iconst_0           | 18:  iload_2  
    1:  istore_2          | 19:  iload_3  
    2:  iconst_1         | 20:  iload 4  
    3:  istore_3         | 22:  imul  
    4:  iload_3          | 23:  iadd  
    5:  iload_0          | 24:  istore_2  
    6:  if_icmpgt 37     | 25:  iinc 4, 1  
    9:  iload_3          | 28:  goto 12  
   10:  istore 4         | 31:  iinc 3, 1  
   12:  iload 4          | 34:  goto 4  
   14:  iload_1         | 37:  iload_2  
   15:  if_icmpgt 31    | 38:  ireturn  
  
}
```

Recovering the Structure of a Java Bytecode Program

Java

```
class a {
    static int sum(int m, int n) {
        int res=0;

        for (int i=1; i<=m; i++) {
            for (int j=i; j<=n; j++) {
                res += i*j;
            }
        }

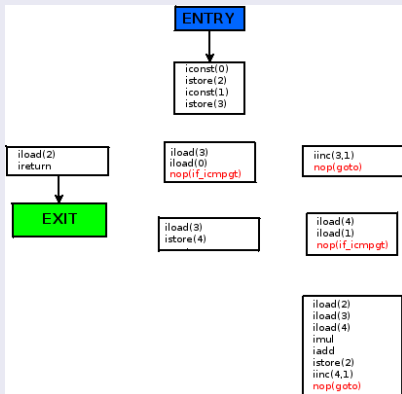
        return res;
    }
}
```

Java Bytecode

```
int sum(int,int) {
    0:  iconst_0
    1:  istore_2
    2:  iconst_1
    3:  istore_3
    4:  iload_3
    5:  iload_0
    6:  if_icmpgt 37
    9:  iload_3
    10: istore_4
    12: iload_4
    14: iload_1
    15: if_icmpgt 31
    18: iload_2
    19: iload_3
    20: iload_4
    22: imul
    23: iadd
    24: istore_2
    25: iinc 4, 1
    28: goto 12
    31: iinc 3, 1
    34: goto 4
    37: iload_2
    38: ireturn
}
```

Recovering the Structure of a Java Bytecode Program

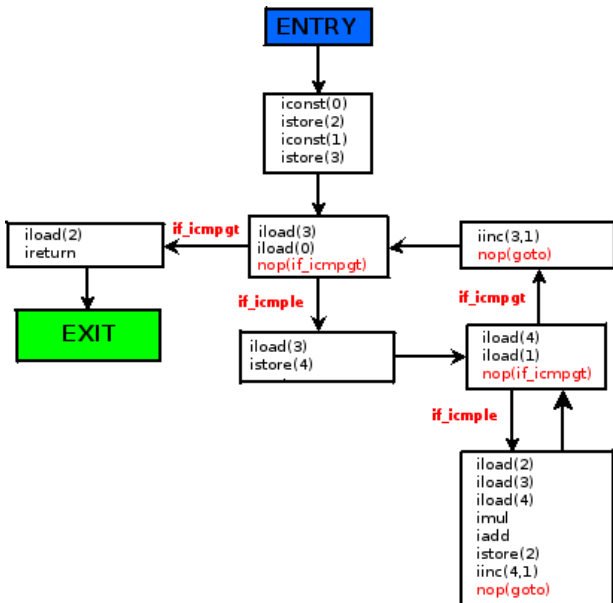
Control Flow Graph



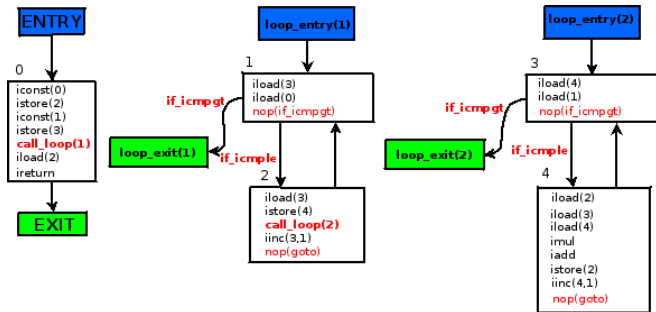
Java Bytecode

```
int sum(int,int) {  
  0:  iconst_0  
  1:  istore_2  
  2:  iconst_1  
  3:  istore_3  
  4:  iload_3  
  5:  iload_0  
  6:  if_icmpgt 37  
  9:  iload_3  
 10:  istore_4  
 12:  iload_4  
 14:  iload_1  
 15:  if_icmpgt 31  
 18:  iload_2  
 19:  iload_3  
 20:  iload_4  
 22:  imul  
 23:  iadd  
 24:  istore_2  
 25:  iinc 4, 1  
 28:  goto 12  
 31:  iinc 3, 1  
 34:  goto 4  
 37:  iload_2  
 38:  ireturn  
}
```


Recovering the Structure - CFG



Recovering the Structure - CFG + Loop Extraction

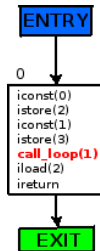


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle, \langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle, \langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

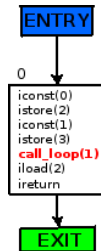


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle, \langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle, \langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

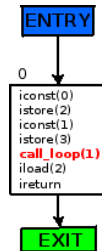


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle, \langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle, \langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

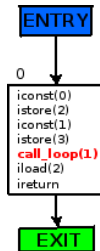


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle$ ,  $\langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle$ ,  $\langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle$ ,  $\langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle$ ,  $\langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

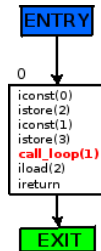


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle, \langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle, \langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

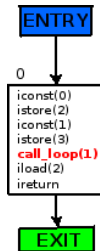


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle, \langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle, \langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

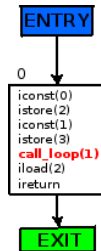


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle$ ,  $\langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle$ ,  $\langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle$ ,  $\langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle$ ,  $\langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

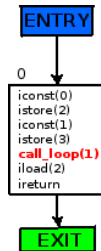


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle, \langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle, \langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```

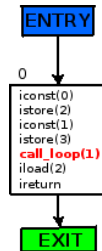


Recovering the Structure - Rule-Based Representation

Rule-Based Program

```
sum( $\langle m, n \rangle, \langle r \rangle$ ) :=  
  init_local_vars( $\langle res, i, j \rangle$ ),  
  sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ).
```

```
sum0( $\langle m, n, res, i, j \rangle, \langle r \rangle$ ) :=  
  iconst(0, s0),  
  istore(s0, res),  
  iconst(1, s0),  
  istore(s0, i),  
  sum1( $\langle m, n, res, i, j \rangle, \langle res, i, j \rangle$ ),  
  iload(res, s0),  
  ireturn(s0, r).
```



Recovering the Structure - Rule-Based Representation

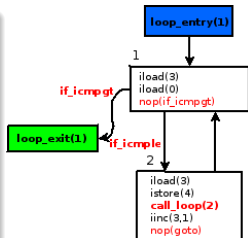
Rule-Based Program

```
sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  iload(m, s1),  
  nop(if_icmpgt(s0, s1)),  
  sum1c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, i, j⟩) :=  
  guard(if_icmple(s0, s1),  
  sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, i, j⟩) :=  
  guard(if_icmpge(s0, s1)).
```

```
sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  istore(s0, j),  
  sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩),  
  iinc(i, 1),  
  nop(goto),  
  sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```



Recovering the Structure - Rule-Based Representation

Rule-Based Program

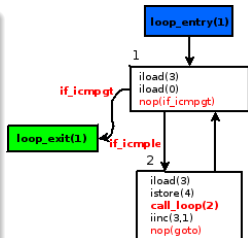
```
sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  iload(m, s1),  
  nop(if_icmpgt(s0, s1)),  
  sum1c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, i, j⟩) :=  
  guard(if_icmple(s0, s1)),
```

```
  sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, i, j⟩) :=  
  guard(if_icmpge(s0, s1)).
```

```
sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  istore(s0, j),  
  sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩),  
  iinc(i, 1),  
  nop(goto),  
  sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```



Recovering the Structure - Rule-Based Representation

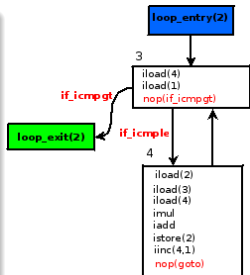
Rule-Based Program

```
sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩) :=  
  iload(j, s0),  
  iload(n, s1),  
  nop(if_icmpgt(s0, s1)),  
  sum3c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, j⟩).
```

```
sum3c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, j⟩) :=  
  guard(if_icmple(s0, s1)),  
  sum4(⟨m, n, res, i, j⟩, ⟨res, j⟩).
```

```
sum3c(⟨m, n, res, i, j, s0, s1⟩, ⟨res, j⟩) :=  
  guard(if_icmpge(s0, s1)).
```

```
sum4(⟨m, n, res, i, j⟩, ⟨res, j⟩) :=  
  iload(res, s0), iload(i, s1), iload(j, s2),  
  imul(s1, s2, s1), iadd(s1, s0, s0),  
  istore(s0, res),  
  iinc(j, 1),  
  nop(goto),  
  sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩).
```



Generating Cost equations System

- **COST MODEL:** The cost of a program $P \equiv$ is the number of bytecode instructions performed.

Generating Cost equations System

- **COST MODEL:** The cost of a program $P \equiv$ is the number of bytecode instructions performed.
- The cost of the rule:

$$\begin{aligned} \text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) := & \\ & \text{iload}(i, s_0), \\ & \text{istore}(s_0, j), \\ & \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ & \text{iinc}(i, 1), \\ & \text{nop}(\text{goto}), \\ & \text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle). \end{aligned}$$

can be defined as:

Generating Cost equations System

- **COST MODEL:** The cost of a program $P \equiv$ is the number of bytecode instructions performed.
- The cost of the rule:

$$\begin{aligned} \text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) := & \\ & \text{iload}(i, s_0), \\ & \text{istore}(s_0, j), \\ & \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ & \text{iinc}(i, 1), \\ & \text{nop}(\text{goto}), \\ & \text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle). \end{aligned}$$

can be defined as:

$$\text{sum}_2(m, n, \text{res}, i, j) =$$

Generating Cost equations System

- **COST MODEL:** The cost of a program $P \equiv$ is the number of bytecode instructions performed.
- The cost of the rule:

$$\begin{aligned} \text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) := & \\ & \text{iload}(i, s_0), \\ & \text{istore}(s_0, j), \\ & \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ & \text{iinc}(i, 1), \\ & \text{nop}(\text{goto}), \\ & \text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle). \end{aligned}$$

can be defined as:

$$\text{sum}_2(m, n, \text{res}, i, j) = 4$$

Generating Cost equations System

- **COST MODEL:** The cost of a program $P \equiv$ is the number of bytecode instructions performed.
- The cost of the rule:

```
sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  istore(s0, j),  
  sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩).  
  iinc(i, 1),  
  nop(goto),  
  sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

can be defined as:

$$sum_2(m, n, res, i, j) = 4 + sum_3(m', n', res', i', j') + sum_1(m'', n'', res'', i'', j'')$$

Generating Cost equations System

- **COST MODEL:** The cost of a program $P \equiv$ is the number of bytecode instructions performed.
- The cost of the rule:

$$\begin{aligned} \text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) := & \\ & \text{iload}(i, s_0), \\ & \text{istore}(s_0, j), \\ & \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ & \text{iinc}(i, 1), \\ & \text{nop}(\text{goto}), \\ & \text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle). \end{aligned}$$

can be defined as:

$$\text{sum}_2(m, n, \text{res}, i, j) = 4 + \text{sum}_3(m', n', \text{res}', i', j') + \text{sum}_1(m'', n'', \text{res}'', i'', j'') \quad | \Psi$$

- $\Psi = \{m' = m, m'' = m, i' = i, i'' = i + 1, \dots\}$.

Generating Cost equations System

- **COST MODEL:** The cost of a program $P \equiv$ is the number of bytecode instructions performed.
- The cost of the rule:

$$\begin{aligned} \text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) := & \\ & \text{iload}(i, s_0), \\ & \text{istore}(s_0, j), \\ & \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ & \text{iinc}(i, 1), \\ & \text{nop}(\text{goto}), \\ & \text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle). \end{aligned}$$

can be defined as:

$$\text{sum}_2(m, n, \text{res}, i, j) = 4 + \text{sum}_3(m', n', \text{res}', i', j') + \text{sum}_1(m'', n'', \text{res}'', i'', j'') \quad | \Psi$$

- $\Psi = \{m' = m, m'' = m, i' = i, i'' = i + 1, \dots\}$.
- Ψ is inferred using *size relations analysis*.

Generating Cost equations System - Cont.

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n, \text{res}, i, j) && \{ \text{res}=0, i=0, j=0 \} \\ \text{sum}_0(m, n, \text{res}, i, j) &= 6 + \text{sum}_1(m, n, \text{res}', i', j) && \{ \text{res}'=0, i'=1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_1(m, n, \text{res}, i, j) &= 3 + \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=i, s_1=m \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_2(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_2(m, n, \text{res}, i, j) &= 4 + \text{sum}_3(m, n, \text{res}, i, j') \\ &\quad + \text{sum}_1(m, n, \text{res}', i', j'') && \{ j'=i, i'=i+1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_3(m, n, \text{res}, i, j) &= 3 + \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=j, s_1=n \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_4(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_4(m, n, \text{res}, i, j) &= 10 + \text{sum}_3(m, n, \text{res}, i, j') && \{ j'=j+1 \} \end{aligned}$$

- We want a closed form solution, e.g. $\text{sum}(m, n) = O(m * n)$.
- The CES is not valid input for Computer Algebra Systems.
- Some transformations are required.

Generating Cost equations System - Cont.

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n, \text{res}, i, j) && \{ \text{res}=0, i=0, j=0 \} \\ \text{sum}_0(m, n, \text{res}, i, j) &= 6 + \text{sum}_1(m, n, \text{res}', i', j) && \{ \text{res}'=0, i'=1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_1(m, n, \text{res}, i, j) &= 3 + \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=i, s_1=m \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_2(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_2(m, n, \text{res}, i, j) &= 4 + \text{sum}_3(m, n, \text{res}, i, j') \\ &\quad + \text{sum}_1(m, n, \text{res}', i', j'') && \{ j'=i, i'=i+1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_3(m, n, \text{res}, i, j) &= 3 + \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=j, s_1=n \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_4(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_4(m, n, \text{res}, i, j) &= 10 + \text{sum}_3(m, n, \text{res}, i, j') && \{ j'=j+1 \} \end{aligned}$$

- We want a closed form solution, e.g. $\text{sum}(m, n) = O(m * n)$.
- The CES is not valid input for Computer Algebra Systems.
- Some transformations are required.
- Removing (temporal) stack variables.

Generating Cost equations System - Cont.

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n, \text{res}, i, j) && \{ \text{res}=0, i=0, j=0 \} \\ \text{sum}_0(m, n, \text{res}, i, j) &= 6 + \text{sum}_1(m, n, \text{res}', i', j) && \{ \text{res}'=0, i'=1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_1(m, n, \text{res}, i, j) &= 3 + \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=i, s_1=m \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_2(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_2(m, n, \text{res}, i, j) &= 4 + \text{sum}_3(m, n, \text{res}, i, j') \\ &\quad + \text{sum}_1(m, n, \text{res}', i', j'') && \{ j'=i, i'=i+1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_3(m, n, \text{res}, i, j) &= 3 + \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=j, s_1=n \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_4(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_4(m, n, \text{res}, i, j) &= 10 + \text{sum}_3(m, n, \text{res}, i, j') && \{ j'=j+1 \} \end{aligned}$$

- We want a closed form solution, e.g. $\text{sum}(m, n) = O(m * n)$.
- The CES is not valid input for Computer Algebra Systems.
- Some transformations are required.
- Removing (temporal) stack variables.
- Removing variables which are irrelevant to the cost.

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) := & \\ & \text{iload}(j, s_0), \\ & \text{iload}(n, s_1), \\ & \text{nop}(\text{if_icmpgt}(s_0, s_1)), \\ & \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle). \end{aligned}$$
$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmple}(s_0, s_1)), \\ & \text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmpgt}(s_0, s_1)). \end{aligned}$$

This step is done using simple dependencies analysis:

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{iload}(j, s_0), && \{s_0 \mapsto j\} \\ &\text{iload}(n, s_1), \\ &\text{nop}(\text{if_icmpgt}(s_0, s_1)), \\ &\text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle). \end{aligned}$$
$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{guard}(\text{if_icmple}(s_0, s_1)), \\ &\text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{guard}(\text{if_icmpgt}(s_0, s_1)). \end{aligned}$$

This step is done using simple dependencies analysis:

- Track the dependencies introduced by the bytecode instructions;

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) := & \\ & \text{iload}(j, s_0), \quad \{s_0 \mapsto j\} \\ & \text{iload}(n, s_1), \quad \{s_0 \mapsto j, s_1 \mapsto i\} \\ & \text{nop}(\text{if_icmpgt}(s_0, s_1)), \\ & \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle). \end{aligned}$$
$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmple}(s_0, s_1)), \\ & \text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmpgt}(s_0, s_1)). \end{aligned}$$

This step is done using simple dependencies analysis:

- Track the dependencies introduced by the bytecode instructions;

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) := & \\ & \text{iload}(j, s_0), \quad \{s_0 \mapsto j\} \\ & \text{iload}(n, s_1), \quad \{s_0 \mapsto j, s_1 \mapsto i\} \\ & \text{nop}(\text{if_icmpgt}(j, n)), \\ & \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle). \end{aligned}$$
$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmple}(s_0, s_1)), \\ & \text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmpgt}(s_0, s_1)). \end{aligned}$$

This step is done using simple dependencies analysis:

- Track the dependencies introduced by the bytecode instructions;
- Use them to transform stack elements to local variables;

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) := & \\ & \text{iload}(j, s_0), \quad \{s_0 \mapsto j\} \\ & \text{iload}(n, s_1), \quad \{s_0 \mapsto j, s_1 \mapsto i\} \\ & \text{nop}(\text{if_icmpgt}(j, n)), \\ & \text{sum}_3^c(\langle m, n, \text{res}, i, j, j, n \rangle, \langle \text{res}, j \rangle). \end{aligned}$$
$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmple}(s_0, s_1)), \\ & \text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j, s_0, s_1 \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmpgt}(s_0, s_1)). \end{aligned}$$

This step is done using simple dependencies analysis:

- Track the dependencies introduced by the bytecode instructions;
- Use them to transform stack elements to local variables;

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{iload}(j, s_0), && \{s_0 \mapsto j\} \\ &\text{iload}(n, s_1), && \{s_0 \mapsto j, s_1 \mapsto i\} \\ &\text{nop}(\text{if_icmpgt}(j, n)), \\ &\text{sum}_3^c(\langle m, n, \text{res}, i, j, j, n \rangle, \langle \text{res}, j \rangle). \end{aligned}$$
$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{guard}(\text{if_icmple}(j, n)), \\ &\text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{guard}(\text{if_icmpgt}(j, n)). \end{aligned}$$

This step is done using simple dependencies analysis:

- Track the dependencies introduced by the bytecode instructions;
- Use them to transform stack elements to local variables;
- Modify the target rule(s);

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{iload}(j, s_0), && \{s_0 \mapsto j\} \\ &\text{iload}(n, s_1), && \{s_0 \mapsto j, s_1 \mapsto i\} \\ &\text{nop}(\text{if_icmplt}(j, n)), \\ &\text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \end{aligned}$$

$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{guard}(\text{if_icmplt}(j, n)), \\ &\text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) &:= \\ &\text{guard}(\text{if_icmplt}(j, n)). \end{aligned}$$

This step is done using simple dependencies analysis:

- Track the dependencies introduced by the bytecode instructions;
- Use them to transform stack elements to local variables;
- Modify the target rule(s);
- Remove duplicated arguments

Removing (temporal) Stack Variables

$$\begin{aligned} \text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) := & \\ & \text{iload}(j, s_0), \quad \{s_0 \mapsto j\} \\ & \text{iload}(n, s_1), \quad \{s_0 \mapsto j, s_1 \mapsto i\} \\ & \text{nop}(\text{if_icmpgt}(j, n)), \\ & \text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \end{aligned}$$
$$\begin{aligned} \text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmple}(j, n)), \\ & \text{sum}_4(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle). \\ \text{sum}_3^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle) := & \\ & \text{guard}(\text{if_icmpgt}(j, n)). \end{aligned}$$

- Applying this transformation before *size analysis* improves the performance significantly, since the abstract states will have much less variables (and instructions).
- In practice we can eliminate all stack variables in the rule-based representation, except maybe those that correspond to a method return value.

Computing Relevant Variables (Relevant Slice)

- The cost is affected *only* by conditional instructions (**guards**)
 - loop exit conditions
 - recursion base-case conditions
 - conditional jump
 - etc.

Computing Relevant Variables (Relevant Slice)

- The cost is affected *only* by conditional instructions (**guards**)
 - loop exit conditions
 - recursion base-case conditions
 - conditional jump
 - etc.
- If a variable affects *directly* or *indirectly* the result of a guard, then it is relevant to the cost.

Computing Relevant Variables (Relevant Slice)

- The cost is affected *only* by conditional instructions (**guards**)
 - loop exit conditions
 - recursion base-case conditions
 - conditional jump
 - etc.
- If a variable affects *directly* or *indirectly* the result of a guard, then it is relevant to the cost.
- Computing the set of arguments (for each rule) that might affect the cost is a:
 - backward slicing problem
 - where the slicing criterion is *all* guards and their variables.

The slicing criterion is:

“All guards and their variables”

The slicing criterion is:

“All guards and their variables”

- This criterion requires a simple slicing algorithm.
- The slicing algorithm does not need to track *implicit dependencies* (e.g., which come from conditional updates) because all guards' variables are considered to be relevant.
- The slicing algorithm should track *direct* and *indirect* dependencies that come from assignment.

```
sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  iload(m, s1),  
  nop(if_icmpgt(i, m)),  
  sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  guard(if_icmple(i, m)),  
  sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  guard(if_icmpge(i, m)).
```

```
sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  istore(s0, j)  
  sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩).  
  iinc(i, 1)  
  nop(goto),  
  sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

- We adapt standard backward slicing and information flow algorithms.

Computing Relevant Variables (Relevant Slice) - Cont.

```
sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  iload(m, s1),  
  nop(if_icmpgt(i, m)),  
  sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  guard(if_icmple(i, m)),  
  sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  guard(if_icmpge(i, m)).
```

```
sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  istore(s0, j)  
  sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩).  
  iinc(i, 1)  
  nop(goto),  
  sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

- We adapt standard backward slicing and information flow algorithms.
- The relevant variables for sum_3 are relevant for sum_2 .

Computing Relevant Variables (Relevant Slice) - Cont.

$$\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{iload}(m, s_1),$
 $\text{nop}(\text{if_icmpgt}(i, m)),$
 $\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmple}(i, m)),$
 $\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmptge}(i, m)).$

$$\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{istore}(s_0, j)$
 $\text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle).$
 $\text{iinc}(i, 1)$
 $\text{nop}(\text{goto}),$
 $\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

- We adapt standard backward slicing and information flow algorithms.
- The relevant variables for sum_3 are relevant for sum_2 .
- All guards are relevant.

Computing Relevant Variables (Relevant Slice) - Cont.

$$\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{iload}(m, s_1),$
 $\text{nop}(\text{if_icmpgt}(i, m)),$
 $\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmple}(i, m)),$
 $\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmpge}(i, m)).$

$$\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{istore}(s_0, j)$
 $\text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle).$
 $\text{iinc}(i, 1)$
 $\text{nop}(\text{goto}),$
 $\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

- We adapt standard backward slicing and information flow algorithms.
- The relevant variables for sum_3 are relevant for sum_2 .
- All guards are relevant.
- Fixpoint computation to propagate information backwards.

Computing Relevant Variables (Relevant Slice) - Cont.

```
sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  iload(m, s1),  
  nop(if_icmpgt(i, m)),  
  sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  guard(if_icmple(i, m)),  
  sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

```
sum1c(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  guard(if_icmpge(i, m)).
```

```
sum2(⟨m, n, res, i, j⟩, ⟨res, i, j⟩) :=  
  iload(i, s0),  
  istore(s0, j)  
  sum3(⟨m, n, res, i, j⟩, ⟨res, j⟩).  
  iinc(i, 1)  
  nop(goto),  
  sum1(⟨m, n, res, i, j⟩, ⟨res, i, j⟩).
```

- We adapt standard backward slicing and information flow algorithms.
- The relevant variables for sum_3 are relevant for sum_2 .
- All guards are relevant.
- Fixpoint computation to propagate information backwards.

Computing Relevant Variables (Relevant Slice) - Cont.

$$\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{iload}(m, s_1),$
 $\text{nop}(\text{if_icmpgt}(i, m)),$
 $\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmple}(i, m)),$
 $\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmpge}(i, m)).$

$$\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{istore}(s_0, j)$
 $\text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle).$
 $\text{iinc}(i, 1)$
 $\text{nop}(\text{goto}),$
 $\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

- We adapt standard backward slicing and information flow algorithms.
- The relevant variables for sum_3 are relevant for sum_2 .
- All guards are relevant.
- Fixpoint computation to propagate information backwards.
- m, n and i are relevant for sum_1 and its rules.

Computing Relevant Variables (Relevant Slice) - Cont.

$$\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{iload}(m, s_1),$
 $\text{nop}(\text{if_icmpgt}(i, m)),$
 $\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmple}(i, m)),$
 $\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

$$\text{sum}_1^c(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{guard}(\text{if_icmpge}(i, m)).$

$$\text{sum}_2(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle) :=$$

$\text{iload}(i, s_0),$
 $\text{istore}(s_0, j)$
 $\text{sum}_3(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, j \rangle).$
 $\text{iinc}(i, 1)$
 $\text{nop}(\text{goto}),$
 $\text{sum}_1(\langle m, n, \text{res}, i, j \rangle, \langle \text{res}, i, j \rangle).$

- We adapt standard backward slicing and information flow algorithms.
- The relevant variables for sum_3 are relevant for sum_2 .
- All guards are relevant.
- Fixpoint computation to propagate information backwards.
- m, n and i are relevant for sum_1 and its rules.
- *It is possible to apply the slicing directly on the CES instead of on the rules.*

Computing Relevant Variables (Relevant Slice) - Cont.

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n, \text{res}, i, j) && \{ \text{res}=0, i=0, j=0 \} \\ \text{sum}_0(m, n, \text{res}, i, j) &= 6 + \text{sum}_1(m, n, \text{res}', i', j) && \{ \text{res}'=0, i'=1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_1(m, n, \text{res}, i, j) &= 3 + \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=i, s_1=m \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_2(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_2(m, n, \text{res}, i, j) &= 4 + \text{sum}_3(m, n, \text{res}, i, j') \\ &\quad + \text{sum}_1(m, n, \text{res}', i', j'') && \{ j'=i, i'=i+1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_3(m, n, \text{res}, i, j) &= 3 + \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=j, s_1=n \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_4(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_4(m, n, \text{res}, i, j) &= 10 + \text{sum}_3(m, n, \text{res}, i, j') && \{ j'=j+1 \} \end{aligned}$$

Computing Relevant Variables (Relevant Slice) - Cont.

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n, \text{res}, i, j) && \{ \text{res}=0, i=0, j=0 \} \\ \text{sum}_0(m, n, \text{res}, i, j) &= 6 + \text{sum}_1(m, n, \text{res}', i', j) && \{ \text{res}'=0, i'=1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_1(m, n, \text{res}, i, j) &= 3 + \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=i, s_1=m \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_2(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_2(m, n, \text{res}, i, j) &= 4 + \text{sum}_3(m, n, \text{res}, i, j') \\ &\quad + \text{sum}_1(m, n, \text{res}', i', j'') && \{ j'=i, i'=i+1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_3(m, n, \text{res}, i, j) &= 3 + \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=j, s_1=n \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_4(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_4(m, n, \text{res}, i, j) &= 10 + \text{sum}_3(m, n, \text{res}, i, j') && \{ j'=j+1 \} \end{aligned}$$

Computing Relevant Variables (Relevant Slice) - Cont.

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n, \text{res}, i, j) && \{ \text{res}=0, i=0, j=0 \} \\ \text{sum}_0(m, n, \text{res}, i, j) &= 6 + \text{sum}_1(m, n, \text{res}', i', j) && \{ \text{res}'=0, i'=1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_1(m, n, \text{res}, i, j) &= 3 + \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=i, s_1=m \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_2(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_1^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_2(m, n, \text{res}, i, j) &= 4 + \text{sum}_3(m, n, \text{res}, i, j') \\ &\quad + \text{sum}_1(m, n, \text{res}', i', j'') && \{ j'=i, i'=i+1 \} \end{aligned}$$

$$\begin{aligned} \text{sum}_3(m, n, \text{res}, i, j) &= 3 + \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) && \{ s_0=j, s_1=n \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= \text{sum}_4(m, n, \text{res}, i, j) && \{ s_0 \leq s_1 \} \\ \text{sum}_3^c(m, n, \text{res}, i, j, s_0, s_1) &= 0 && \{ s_0 > s_1 \} \\ \text{sum}_4(m, n, \text{res}, i, j) &= 10 + \text{sum}_3(m, n, \text{res}, i, j') && \{ j'=j+1 \} \end{aligned}$$

Computing Relevant Variables (Relevant Slice) - Cont.

$$\begin{aligned} \text{sum}(m, n) &= \text{sum}_0(m, n) && \{res=0, i=0, j=0\} \\ \text{sum}_0(m, n) &= 6 + \text{sum}_1(m, n, i') && \{res'=0, i'=1\} \end{aligned}$$

$$\begin{aligned} \text{sum}_1(m, n, i) &= 3 + \text{sum}_1^c(m, n, i) && \{s_0=i, s_1=m\} \\ \text{sum}_1^c(m, n, i) &= \text{sum}_2(m, n, i) && \{s_0 \leq s_1\} \\ \text{sum}_1^c(m, n, i) &= 0 && \{s_0 > s_1\} \\ \text{sum}_2(m, n, i) &= 4 + \text{sum}_3(n, j') && \\ &+ \text{sum}_1(m, n, i') && \{j'=i, i'=i+1\} \end{aligned}$$

$$\begin{aligned} \text{sum}_3(n, j) &= 3 + \text{sum}_3^c(n, j) && \{s_0=j, s_1=n\} \\ \text{sum}_3^c(n, j) &= \text{sum}_4(n, j) && \{s_0 \leq s_1\} \\ \text{sum}_3^c(n, j) &= 0 && \{s_0 > s_1\} \\ \text{sum}_4(n, j) &= 10 + \text{sum}_3(n, j') && \{j'=j+1\} \end{aligned}$$

- Practical cost analysis of Java bytecode requires:
 - Simple variables dependency analysis to eliminate stack variables.
 - Program slicing to eliminate parts which irrelevant to cost.
- Variables dependency analysis
 - can be applied directly on the cost equations; or
 - on the rule-based representation which improves significantly the performance of size analysis
- Slicing of cost equations
 - can be applied directly on the cost equations; or
 - on the rule-based representation which might (?) improve the performance of size analysis.