From Termination to Cost
(in Object-Oriented Languages)

Elvira Albert

Complutense University of Madrid (Spain)

11th International Workshop on Termination

Edinburgh, 14 July, 2010

Outline of the Talk

1. Simple Imperative Bytecode Programs

Outline of the Talk

1. Simple Imperative Bytecode Programs
    - transform into rule-based form by means of CFG
    - abstract interpretation based size analysis
    - find ranking functions for each loop

Outline of the Talk

1. Simple Imperative Bytecode Programs
    - transform into rule-based form by means of CFG
    - abstract interpretation based size analysis
    - find ranking functions for each loop
2. From Termination to Cost

Outline of the Talk

1. Simple Imperative Bytecode Programs
    - transform into rule-based form by means of CFG
    - abstract interpretation based size analysis
    - find ranking functions for each loop
2. From Termination to Cost
    - generating recurrence equations from abstract rules
    - use ranking functions as UB on # iterations
    - the COSTA system: asymptotic bounds, verification, etc.

Outline of the Talk

1. Simple Imperative Bytecode Programs
    - transform into rule-based form by means of CFG
    - abstract interpretation based size analysis
    - find ranking functions for each loop
2. From Termination to Cost
    - generating recurrence equations from abstract rules
    - use ranking functions as UB on # iterations
    - the COSTA system: asymptotic bounds, verification, etc.
3. Field-Sensitive Analysis

Outline of the Talk

1. Simple Imperative Bytecode Programs
    - transform into rule-based form by means of CFG
    - abstract interpretation based size analysis
    - find ranking functions for each loop
2. From Termination to Cost
    - generating recurrence equations from abstract rules
    - use ranking functions as UB on # iterations
    - the COSTA system: asymptotic bounds, verification, etc.
3. Field-Sensitive Analysis
    - shared mutable data
        - shared (i.e., aliases are allowed)
        - mutable (i.e., can be modified multiple times)

Outline of the Talk

1. Simple Imperative Bytecode Programs
    - transform into rule-based form by means of CFG
    - abstract interpretation based size analysis
    - find ranking functions for each loop
2. From Termination to Cost
    - generating recurrence equations from abstract rules
    - use ranking functions as UB on # iterations
    - the COSTA system: asymptotic bounds, verification, etc.
3. Field-Sensitive Analysis
    - shared mutable data
        - shared (i.e., aliases are allowed)
        - mutable (i.e., can be modified multiple times)
4. Tool demo

PART 1: TERMINATION OF SIMPLE BYTECODE PROGRAMS

- Simple imperative stack-based bytecode language
- Ignoring global memory (heap)
- Without OO features: virtual invocations, etc.

PART 1: TERMINATION OF SIMPLE BYTECODE PROGRAMS

- Simple imperative stack-based bytecode language
- Ignoring global memory (heap)
- Without OO features: virtual invocations, etc.

- Why bytecode ?

PART 1: TERMINATION OF SIMPLE BYTECODE PROGRAMS

- Simple imperative stack-based bytecode language
- Ignoring global memory (heap)
- Without OO features: virtual invocations, etc.

- Why bytecode ?
  - common in Java to have access to bytecode but not to source
  - even more in commercial software and in mobile code
  - kind of *normal form* for Java programs (similar to .net)

PART 1: TERMINATION OF SIMPLE BYTECODE PROGRAMS

- Simple imperative stack-based bytecode language
- Ignoring global memory (heap)
- Without OO features: virtual invocations, etc.

- Why bytecode ?
  - common in Java to have access to bytecode but not to source
  - even more in commercial software and in mobile code
  - kind of *normal form* for Java programs (similar to .net)
- What are the challenges?

PART 1: TERMINATION OF SIMPLE BYTECODE PROGRAMS

- Simple imperative stack-based bytecode language
- Ignoring global memory (heap)
- Without OO features: virtual invocations, etc.

- Why bytecode ?
  - common in Java to have access to bytecode but not to source
  - even more in commercial software and in mobile code
  - kind of *normal form* for Java programs (similar to .net)
- What are the challenges?
  - loops originate from different sources, such as conditional and unconditional jumps, method calls, or even exceptions

PART 1: TERMINATION OF SIMPLE BYTECODE PROGRAMS

- Simple imperative stack-based bytecode language
- Ignoring global memory (heap)
- Without OO features: virtual invocations, etc.

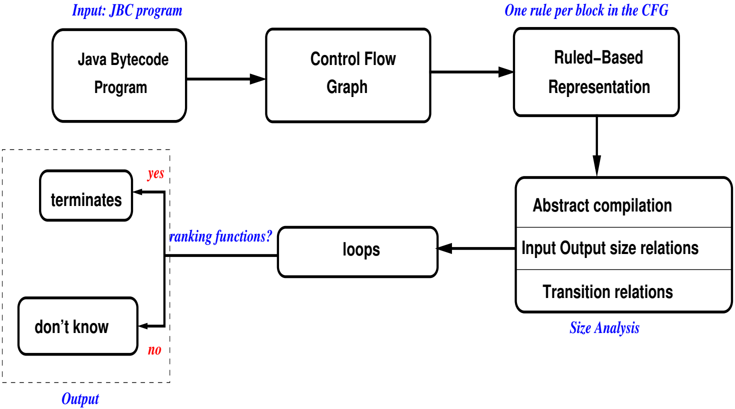
- Why bytecode ?
  - common in Java to have access to bytecode but not to source
  - even more in commercial software and in mobile code
  - kind of *normal form* for Java programs (similar to .net)
- What are the challenges?
  - loops originate from different sources, such as conditional and unconditional jumps, method calls, or even exceptions
  - size measures must consider primitive types, user defined objects, and arrays;

PART 1: TERMINATION OF SIMPLE BYTECODE PROGRAMS

- Simple imperative stack-based bytecode language
- Ignoring global memory (heap)
- Without OO features: virtual invocations, etc.

- Why bytecode ?
  - common in Java to have access to bytecode but not to source
  - even more in commercial software and in mobile code
  - kind of *normal form* for Java programs (similar to .net)
- What are the challenges?
  - loops originate from different sources, such as conditional and unconditional jumps, method calls, or even exceptions
  - size measures must consider primitive types, user defined objects, and arrays;
  - tracking data is more difficult, as data can be stored in variables, operand stack elements or heap locations.

Termination Analysis Components



Input: JBC program

Java Bytecode Program

Control Flow Graph

One rule per block in the CFG

Ruled–Based Representation

Abstract compilation

Input Output size relations

Transition relations

Size Analysis

loops

ranking functions?

yes

terminates

no

don't know

Output

Java Program and its translation to Java Bytecode

```
static void sort(int a[]) {
  for (int i=a.length-2; i≥0; i--) {
    int j=i+1;
    int v=a[i];
    while ( j<a.length && a[j]<v) {
      a[j-1]=a[j];
      j++;
    }
    a[j-1]=v;
  }
}
```

Java Program and its translation to Java Bytecode

```
static void sort(int a[]) {
  for (int i=a.length-2; i≥0; i--) {
    int j=i+1;
    int v=a[i];
    while ( j<a.length && a[j]<v) {
      a[j-1]=a[j];
      j++;
    }
    a[j-1]=v;
  }
}
```

```
 0: aload_0
 1: arraylength
 2: iconst_2
 3: isub
 4: istore_1
 5: iload_1
 6: iflt 56
 9: iload_1
10: iconst_1
11: iadd
12: istore_2
13: aload_0
14: iload_1
15: iaload
16: istore_3
17: iload_2
18: aload_0
19: arraylength
20: if_icmpge 44
23: aload_0
24: iload_2
25: iaload
```

```
26: iload_3
27: if_icmpge 44
30: aload_0
31: iload_2
32: iconst_1
33: isub
34: aload_0
35: iload_2
36: iaload
37: iastore
38: iinc 2, 1
41: goto 17
44: aload_0
45: iload_2
46: iconst_1
47: isub
48: iload_3
49: iastore
50: iinc 1, -1
53: goto 5
56: return
```

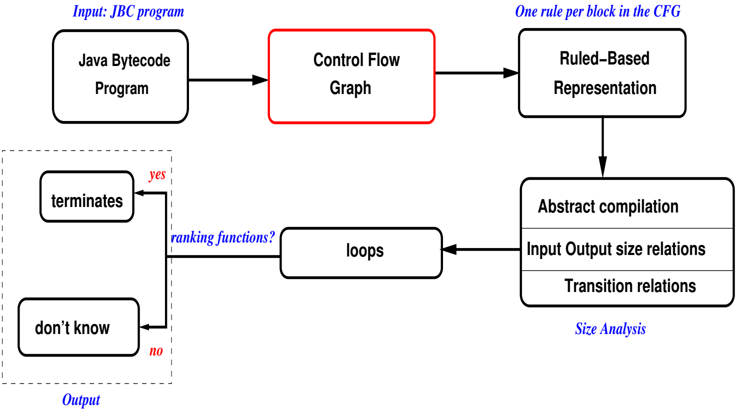
Java Program and its translation to Java Bytecode

```
static void sort(int a[]) {
  for (int i=a.length-2; i≥0; i--) {
    int j=i+1;
    int v=a[i];
    while ( j<a.length && a[j]<v) {
      a[j-1]=a[j];
      j++;
    }
    a[j-1]=v;
  }
}
```

```
 0: aload_0
 1: arraylength
 2: iconst_2
 3: isub
 4: istore_1
 5: iload_1
 6: iflt 56
 9: iload_1
10: iconst_1
11: iadd
12: istore_2
13: aload_0
14: iload_1
15: iaload
16: istore_3
17: iload_2
18: aload_0
19: arraylength
20: if_icmpge 44
23: aload_0
24: iload_2
25: iaload
```

```
26: iload_3
27: if_icmpge 44
30: aload_0
31: iload_2
32: iconst_1
33: isub
34: aload_0
35: iload_2
36: iaload
37: iastore
38: iinc 2, 1
41: goto 17
44: aload_0
45: iload_2
46: iconst_1
47: isub
48: iload_3
49: iastore
50: iinc 1, -1
53: goto 5
56: return
```

Termination Analysis Components



Input: JBC program

Java Bytecode Program

Control Flow Graph

One rule per block in the CFG

Ruled–Based Representation

Abstract compilation

Input Output size relations

Transition relations

Size Analysis

loops

ranking functions?

yes

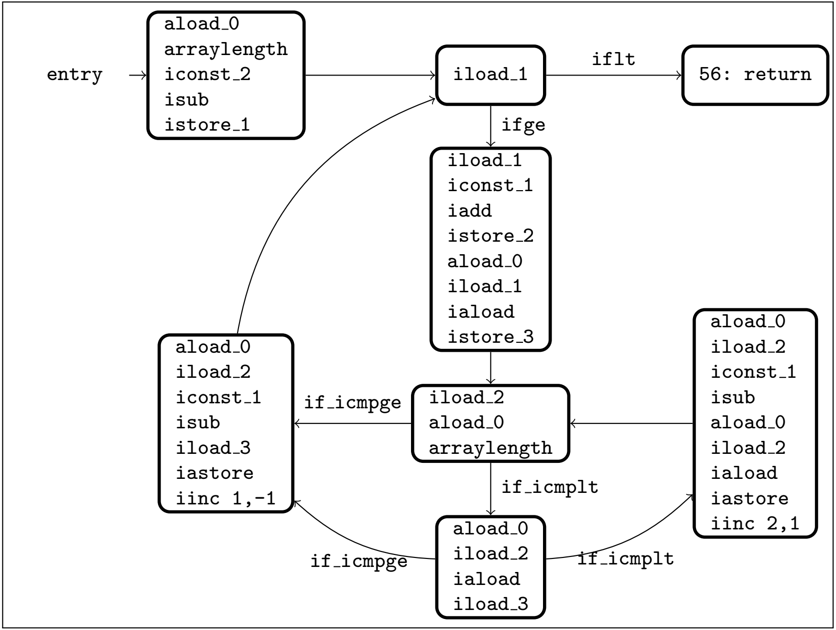
terminates

no

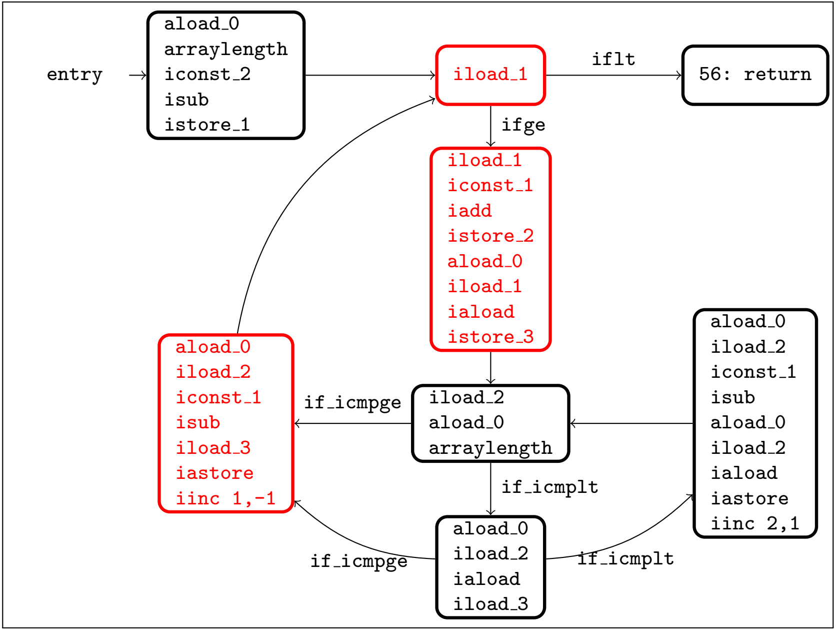
don't know

Output

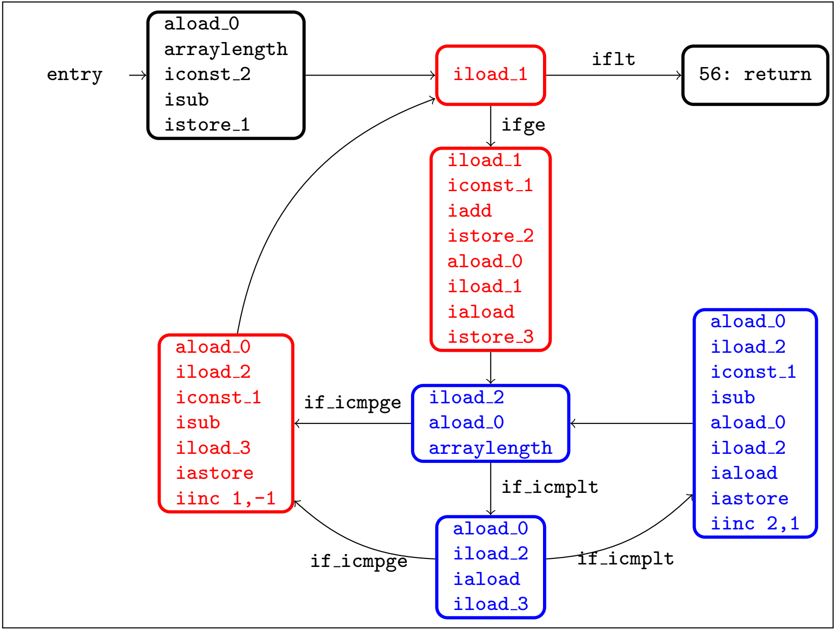
Control Flow Graph - Loops Extraction



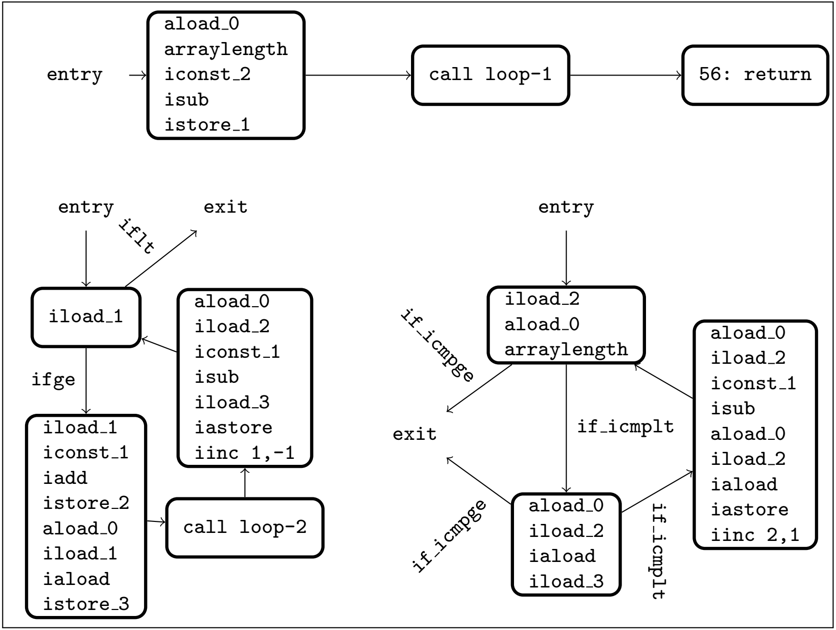
Control Flow Graph - Loops Extraction



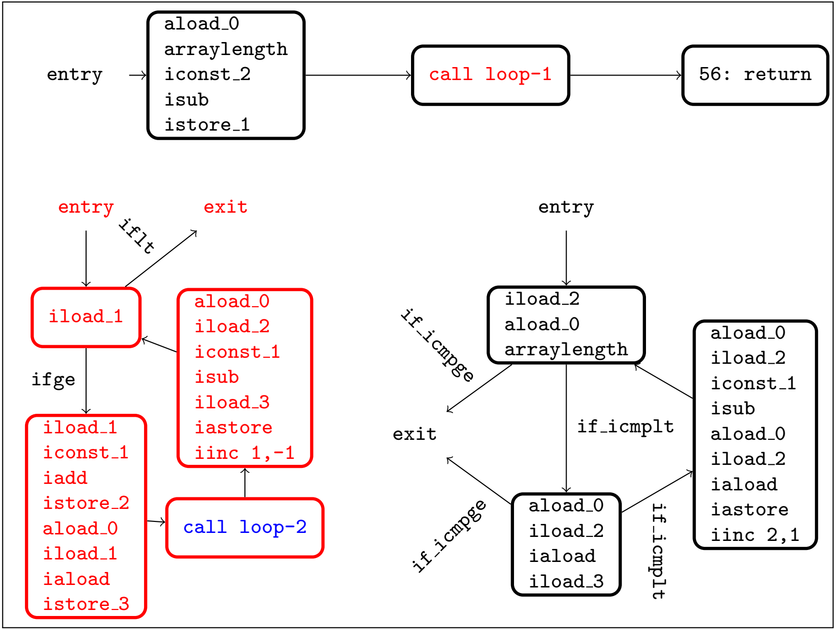
Control Flow Graph - Loops Extraction



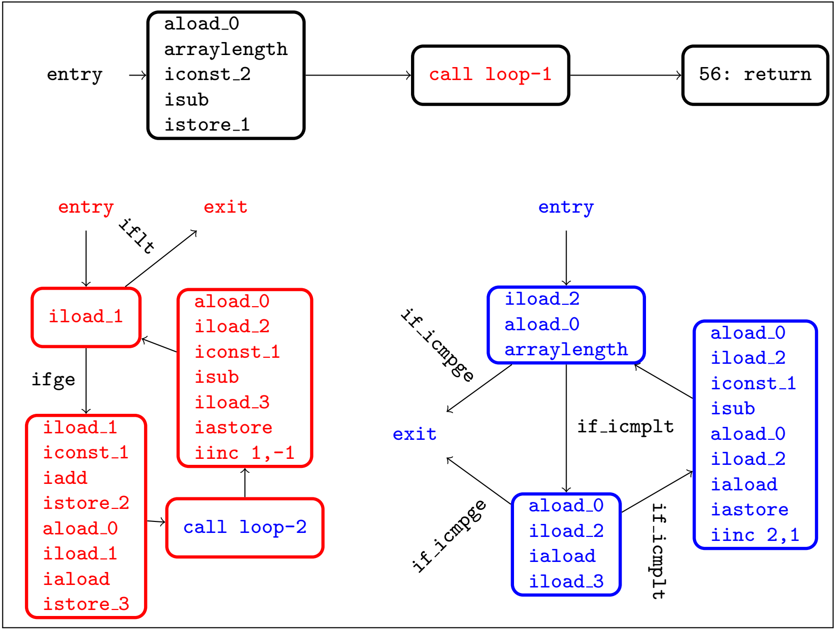
Control Flow Graph - Loops Extraction



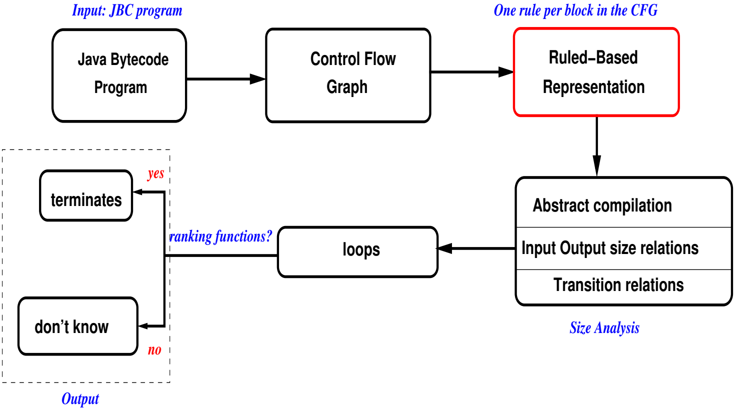
Control Flow Graph - Loops Extraction



Control Flow Graph - Loops Extraction



Termination Analysis Components



Input: JBC program

Java Bytecode Program

Control Flow Graph

One rule per block in the CFG

Ruled–Based Representation

Abstract compilation

Input Output size relations

Transition relations

Size Analysis

loops

ranking functions?

yes

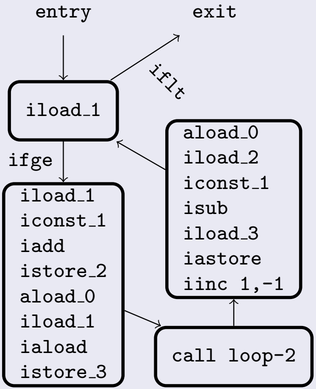
terminates

no

don't know

Output

Rule-based Representation - Cont.



entry

exit

iflt

iload_1

ifge

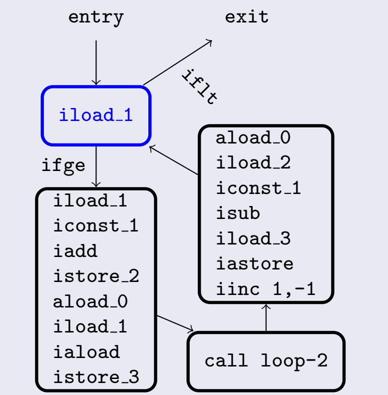
iload_1
iconst_1
iadd
istore_2
aload_0
iload_1
iaload
istore_3

aload_0
iload_2
iconst_1
isub
iload_3
iastore
iinc 1,-1

call loop-2

```
for (int i=a.length-2;i≥0;i--) {
  int j=i+1;
  int v=a[i];
  .....
  a[j-1]=v;
}
```

Rule-based Representation - Cont.



```
iload_1
```

```
iload_1
iconst_1
iadd
istore_2
aload_0
iload_1
iaload
istore_3
```

```
call loop-2
```

```
aload_0
iload_2
iconst_1
isub
iload_3
iastore
iinc 1,-1
```

```
for (int i=a.length-2;i≥0;i--) {
  int j=i+1;
  int v=a[i];
  .....
  a[j-1]=v;
}
```

$$\mathcal{B}_1(\langle a,i\rangle,\langle\rangle) := iload(i,s_0),\ \mathcal{B}_1^c(\langle a,i,s_0\rangle,\langle\rangle).$$

$$\mathcal{B}_1^c(\langle a,i,s_0\rangle,\langle\rangle) := guard(s_0 < 0).$$

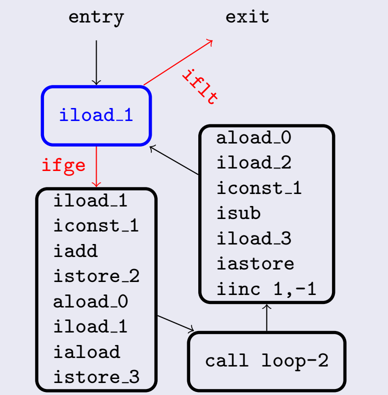
$$\mathcal{B}_1^c(\langle a,i,s_0\rangle,\langle\rangle) := guard(s_0 \geq 0),\ \mathcal{B}_2(\langle a,i\rangle,\langle\rangle).$$

$$\mathcal{B}_2(\langle a,i\rangle,\langle\rangle) := iload(i,s_0),\ iconst(1,s_1),\ iadd(s_0,s_1,s_0),\ istore(s_0,j),\ aload(a,s_0),\ iload(i,s_1),\ iaload(s_0,s_1,s_0),\ istore(s_0,v),\ \mathcal{B}_3(\langle a,i,j,v\rangle,\langle\rangle).$$

$$\mathcal{B}_3(\langle a,i,j,v\rangle,\langle\rangle) := \mathcal{C}_1(\langle a,j,v\rangle,\langle j\rangle),\ \mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle).$$

$$\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) := aload(a,s_0),\ iload(j,s_1),\ iconst(1,s_2),\ isub(s1,s_2,s_1),\ iload(v,s_2),\ iastore(s_0,s_1,s_2),\ iinc(i,-1),\ \mathcal{B}_1(\langle a,i\rangle,\langle\rangle).$$

Rule-based Representation - Cont.



entry

exit

iflt

ifge

```
iload_1
```

```
iload_1
iconst_1
iadd
istore_2
aload_0
iload_1
iaload
istore_3
```

```
aload_0
iload_2
iconst_1
isub
iload_3
iastore
iinc 1,-1
```

```
call loop-2
```

```
for (int i=a.length-2;i≥0;i--) {
  int j=i+1;
  int v=a[i];
  .....
  a[j-1]=v;
}
```

$\mathcal{B}_1(\langle a,i\rangle,\langle\rangle) :=$
$\quad iload(i,s_0),$
$\quad \mathcal{B}_1^c(\langle a,i,s_0\rangle,\langle\rangle).$

$\mathcal{B}_1^c(\langle a,i,s_0\rangle,\langle\rangle) :=$
$\quad guard(s_0 < 0).$

$\mathcal{B}_1^c(\langle a,i,s_0\rangle,\langle\rangle) :=$
$\quad guard(s_0 \geq 0)$
$\quad \mathcal{B}_2(\langle a,i\rangle,\langle\rangle).$

$\mathcal{B}_2(\langle a,i\rangle,\langle\rangle) :=$
$\quad iload(i,s_0),$
$\quad iconst(1,s_1),$
$\quad iadd(s_0,s_1,s_0),$
$\quad istore(s_0,j),$
$\quad aload(a,s_0),$
$\quad iload(i,s_1),$
$\quad iaload(s_0,s_1,s_0),$
$\quad istore(s_0,v),$
$\quad \mathcal{B}_3(\langle a,i,j,v\rangle,\langle\rangle).$

$\mathcal{B}_3(\langle a,i,j,v\rangle,\langle\rangle) :=$
$\quad \mathcal{C}_1(\langle a,j,v\rangle,\langle j\rangle),$
$\quad \mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle).$

$\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) :=$
$\quad aload(a,s_0),$
$\quad iload(j,s_1),$
$\quad iconst(1,s_2),$
$\quad isub(s1,s_2,s_1),$
$\quad iload(v,s_2),$
$\quad iastore(s_0,s_1,s_2),$
$\quad iinc(i,-1),$
$\quad \mathcal{B}_1(\langle a,i\rangle,\langle\rangle).$

Nice features of rule-based representation

rule-based program

set of *procedures* defined by one or more rules:
$p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \; := g, b_1, \ldots, b_n$

Nice features of rule-based representation

rule-based program

set of *procedures* defined by one or more rules:
$p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \ := g, b_1, \ldots, b_n$

- Loops are extracted in separate procedures:
  $\mathcal{B}_3(\langle a, i, j, v \rangle, \langle \rangle) := \mathcal{C}_1(\langle a, j, v \rangle, \langle j \rangle), \mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle).$

Nice features of rule-based representation

rule-based program

set of *procedures* defined by one or more rules:
$p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \ := g, b_1, \ldots, b_n$

- Loops are extracted in separate procedures:
  $\mathcal{B}_3(\langle a, i, j, v \rangle, \langle \rangle) := \mathcal{C}_1(\langle a, j, v \rangle, \langle j \rangle), \mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle).$
- All iterative constructs (loops) fit in the same setting:
  - recursive calls
  - iterative loops (conditional and unconditional jumps)

Nice features of rule-based representation

rule-based program

set of *procedures* defined by one or more rules:
$p(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \ := g, b_1, \ldots, b_n$

- Loops are extracted in separate procedures:
  $\mathcal{B}_3(\langle a, i, j, v\rangle, \langle\rangle) := \mathcal{C}_1(\langle a, j, v\rangle, \langle j\rangle), \mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle).$
- All iterative constructs (loops) fit in the same setting:
  - recursive calls
  - iterative loops (conditional and unconditional jumps)
- All kinds of variables are just arguments:
  - local variables
  - stack elements

  $\mathcal{B}_2(\langle \mathtt{a}, \mathtt{i}\rangle, \langle\rangle) := \mathtt{iload}(\mathtt{i}, \mathtt{s}_0), \mathtt{iconst}(1, \mathtt{s}_1), \ldots$

Nice features of rule-based representation

rule-based program

set of *procedures* defined by one or more rules:
$p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \ := g, b_1, \ldots, b_n$

- Loops are extracted in separate procedures:
  $\mathcal{B}_3(\langle a, i, j, v \rangle, \langle\rangle) := \mathcal{C}_1(\langle a, j, v \rangle, \langle j \rangle), \mathcal{B}_4(\langle a, i, j, v \rangle, \langle\rangle).$
- All iterative constructs (loops) fit in the same setting:
  - recursive calls
  - iterative loops (conditional and unconditional jumps)
- All kinds of variables are just arguments:
  - local variables
  - stack elements

  $\mathcal{B}_2(\langle \mathtt{a}, \mathtt{i} \rangle, \langle\rangle) := \mathtt{iload}(\mathtt{i}, \mathtt{s}_0), \mathtt{iconst}(1, \mathtt{s}_1), \ldots$
- Guards are the only form of conditional
  $\mathcal{B}_1^c(\langle a, i, s_0 \rangle, \langle\rangle) := guard(s_0 \geq 0), \mathcal{B}_2(\langle a, i \rangle, \langle\rangle).$

Nice features of rule-based representation

rule-based program

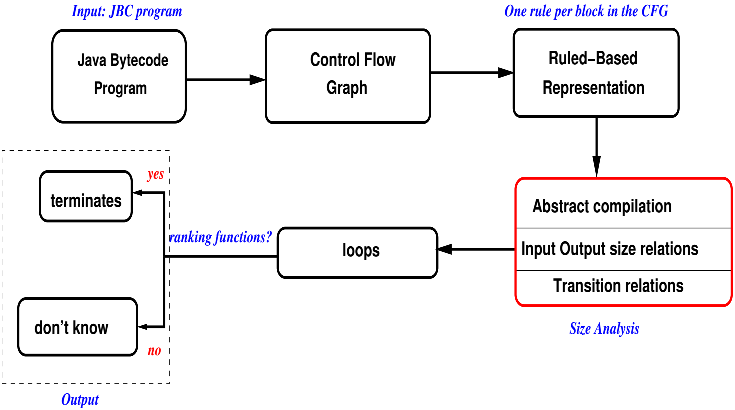
set of *procedures* defined by one or more rules:

$p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \ := g, b_1, \ldots, b_n$

- Loops are extracted in separate procedures:
  $\mathcal{B}_3(\langle a, i, j, v \rangle, \langle \rangle) := \mathcal{C}_1(\langle a, j, v \rangle, \langle j \rangle), \mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle).$
- All iterative constructs (loops) fit in the same setting:
  - recursive calls
  - iterative loops (conditional and unconditional jumps)
- All kinds of variables are just arguments:
  - local variables
  - stack elements

  $\mathcal{B}_2(\langle \mathtt{a}, \mathtt{i} \rangle, \langle \rangle) := \mathtt{iload}(\mathtt{i}, \mathtt{s}_0), \mathtt{iconst}(1, \mathtt{s}_1), \ldots$
- Guards are the only form of conditional
  $\mathcal{B}_1^c(\langle a, i, s_0 \rangle, \langle \rangle) := guard(s_0 \geq 0), \mathcal{B}_2(\langle a, i \rangle, \langle \rangle).$
- Rules may have multiple output parameters

Termination Analysis Components



Input: JBC program

Java Bytecode Program

Control Flow Graph

One rule per block in the CFG

Ruled–Based Representation

Abstract compilation

Input Output size relations

Transition relations

Size Analysis

loops

ranking functions?

terminates

yes

don't know

no

Output

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

$$
\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}
$$

$$
\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a,
\end{array}
$$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\le}s_1$

$$
\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) :=\\
\quad aload(a, s_0),\\
\quad iload(j, s_1),\\
\quad iconst(1, s_2),\\
\quad isub(s1, s_2, s_1),\\
\quad iload(v, s_2),\\
\quad iastore(s_0, s_1, s_2),\\
\quad iinc(i, -1),\\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}
$$

$$
\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) :=\\
\quad s_0 = a,\\
\quad s_1 = j,
\end{array}
$$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1,
\end{array}$$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

$$\begin{array}{l} \mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\ \quad aload(a, s_0), \\ \quad iload(j, s_1), \\ \quad iconst(1, s_2), \\ \quad isub(s1, s_2, s_1), \\ \quad iload(v, s_2), \\ \quad iastore(s_0, s_1, s_2), \\ \quad iinc(i, -1), \\ \quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle). \end{array}$$

$$\begin{array}{l} \mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\ \quad s_0 = a, \\ \quad s_1 = j, \\ \quad s_2 = 1, \\ \quad s_1' = s_1 - s_2, \end{array}$$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v,
\end{array}$$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

$$\boxed{\begin{array}{l}\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\ \quad aload(a, s_0), \\ \quad iload(j, s_1), \\ \quad iconst(1, s_2), \\ \quad isub(s1, s_2, s_1), \\ \quad iload(v, s_2), \\ \quad iastore(s_0, s_1, s_2), \\ \quad iinc(i, -1), \\ \quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).\end{array}}$$

$$\boxed{\begin{array}{l}\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\ \quad s_0 = a, \\ \quad s_1 = j, \\ \quad s_2 = 1, \\ \quad s_1' = s_1 - s_2, \\ \quad s_2' = v, \\ \quad true,\end{array}}$$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

$$\boxed{\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}}
\qquad
\boxed{\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v, \\
\quad true, \\
\quad i' = i - 1, \\
\\
\end{array}}$$

Size Relations Analysis

Norms:

- The size of an integer variable is its value
- The size of an array is its length
- The size of an object is the longest path-length reachable from that object (*unknown for cyclic structures*)

Abstract compilation:

- $iadd(a, b, c)$ is abstracted to $c{=}b{+}a$
- $guard(icmple(s_0, s_1))$ is abstracted to $s_0{\leq}s_1$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v, \\
\quad true, \\
\quad i' = i - 1, \\
\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{array}$$

Size Relations Analysis - Cont.

- We can apply existing techniques to reason on the termination of the abstract program (CLP, TRS, ...)

Size Relations Analysis - Cont.

- We can apply existing techniques to reason on the termination of the abstract program (CLP, TRS, ...)
- Reduce the termination of the abstract program to the termination (well-foundness) of the transition relation

Size Relations Analysis - Cont.

- We can apply existing techniques to reason on the termination of the abstract program (CLP, TRS, ...)
- Reduce the termination of the abstract program to the termination (well-foundness) of the transition relation

$$\begin{array}{lcll} \mathcal{B}_1(\langle a, i\rangle) & \longrightarrow & \mathcal{B}_1(\langle a', i'\rangle) & \{a' = a, i >= 0, i' = i - 1\} \\ \mathcal{C}_1(\langle a, j, v\rangle & \longrightarrow & \mathcal{C}_1(\langle a', j', v'\rangle) & \{a' = a, j' = j + 1, j < a\} \end{array}$$

Size Relations Analysis - Cont.

- We can apply existing techniques to reason on the termination of the abstract program (CLP, TRS, ...)
- Reduce the termination of the abstract program to the termination (well-foundness) of the transition relation

$$\begin{array}{lcll} \mathcal{B}_1(\langle a,i\rangle) & \longrightarrow & \mathcal{B}_1(\langle a',i'\rangle) & \{a'=a, i>=0, i'=i-1\} \\ \mathcal{C}_1(\langle a,j,v\rangle & \longrightarrow & \mathcal{C}_1(\langle a',j',v'\rangle) & \{a'=a, j'=j+1, j<a\} \end{array}$$

- We can easily find ranking functions:

```
for (int i=a.length-2; i≥0; i--) {
   ...

   while ( j<a.length && a[j]<v) {
    ...
    j++;
}
```

Size Relations Analysis - Cont.

- We can apply existing techniques to reason on the termination of the abstract program (CLP, TRS, ...)
- Reduce the termination of the abstract program to the termination (well-foundness) of the transition relation

$$
\begin{array}{lcll}
\mathcal{B}_1(\langle a, i\rangle) & \longrightarrow & \mathcal{B}_1(\langle a', i'\rangle) & \{a' = a, i >= 0, i' = i - 1\} \\
\mathcal{C}_1(\langle a, j, v\rangle & \longrightarrow & \mathcal{C}_1(\langle a', j', v'\rangle) & \{a' = a, j' = j + 1, j < a\}
\end{array}
$$

- We can easily find ranking functions:

```
//@decreasing f_B1(a,i) = i
for (int i=a.length-2; i≥0; i--) {
   ...

   while ( j<a.length && a[j]<v) {
    ...
    j++;
}
```

Size Relations Analysis - Cont.

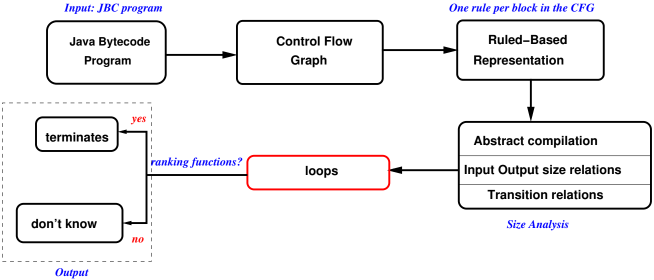
- We can apply existing techniques to reason on the termination of the abstract program (CLP, TRS, ...)
- Reduce the termination of the abstract program to the termination (well-foundness) of the transition relation

$$\begin{array}{lcll} \mathcal{B}_1(\langle a,i\rangle) & \longrightarrow & \mathcal{B}_1(\langle a',i'\rangle) & \{a'=a, i>=0, i'=i-1\} \\ \mathcal{C}_1(\langle a,j,v\rangle & \longrightarrow & \mathcal{C}_1(\langle a',j',v'\rangle) & \{a'=a, j'=j+1, j<a\} \end{array}$$

- We can easily find ranking functions:

```
//@decreasing $\mathbf{f}_{\mathcal{B}_1}(\mathbf{a},\mathbf{i}) = \mathbf{i}$
for (int i=a.length-2; i≥0; i--) {
   ...
   //@decreasing $\mathbf{f}_{\mathcal{C}_1}(\mathbf{a},\mathbf{j},\mathbf{v}) = \mathbf{a} - \mathbf{j}$
   while ( j<a.length && a[j]<v) {
    ...
    j++;
}
```

Proving Termination - Cont.



Loops

$$\begin{array}{lll} \mathcal{B}_1(\langle a, i\rangle) \longrightarrow & \mathcal{B}_1(\langle a', i'\rangle) & \{a' = a, i >= 0, i' = i - 1\} \\ \mathcal{C}_1(\langle a, j, v\rangle \longrightarrow & \mathcal{C}_1(\langle a', j', v'\rangle) & \{a' = a, j' = j + 1, j < a\} \end{array}$$

Soundness

Theorem (Soundness)

Let P be a JBC program and $C_{\mathcal{A}}$ the transition relations computed from P. If there exists a non-terminating trace in P then there exists a non-terminating derivation in $C_{\mathcal{A}}$.

Proof.

- By construction: the rule-based program captures all possible non-terminating traces in the original program.
- By correctness of size analysis: given a trace in the rule-based program, there exists an equivalent one in the transition relations.
- Termination of transition relations entails termination in the original JBC program.

Conclusions (Part 1)

- simple imperative programs automatically transformed into rule-based representation [ESOP'07]

Conclusions (Part 1)

- simple imperative programs automatically transformed into rule-based representation [ESOP'07]
- transition relations can be obtained from rules and techniques for proving termination can be adapted [FMOODS'08]

Conclusions (Part 1)

- simple imperative programs automatically transformed into rule-based representation [ESOP'07]
- transition relations can be obtained from rules and techniques for proving termination can be adapted [FMOODS'08]
  - COSTA: COSt and Termination Analyzer for Java bytecode [2007-2010]

Conclusions (Part 1)

- simple imperative programs automatically transformed into rule-based representation [ESOP'07]
- transition relations can be obtained from rules and techniques for proving termination can be adapted [FMOODS'08]
    - COSTA: COSt and Termination Analyzer for Java bytecode [2007-2010]
    - Julia: abstraction of data structures using path-length

Conclusions (Part 1)

- simple imperative programs automatically transformed into rule-based representation [ESOP'07]
- transition relations can be obtained from rules and techniques for proving termination can be adapted [FMOODS'08]
  - COSTA: COSt and Termination Analyzer for Java bytecode [2007-2010]
  - Julia: abstraction of data structures using path-length
  - AProVE: recent work proposes finer abstractions of data structures into terms

Conclusions (Part 1)

- simple imperative programs automatically transformed into rule-based representation [ESOP'07]
- transition relations can be obtained from rules and techniques for proving termination can be adapted [FMOODS'08]
  - COSTA: COSt and Termination Analyzer for Java bytecode [2007-2010]
  - Julia: abstraction of data structures using path-length
  - AProVE: recent work proposes finer abstractions of data structures into terms
- COSTA can infer more than termination: complexity and resource usage (cost)

PART 2: FROM TERMINATION TO COST

Introduction to cost analysis (Wegbreit'75)

static cost analysis

bound the cost of executing program P on any input data $\overline{x}$ without having to actually *run* $P(\overline{x})$

Introduction to cost analysis (Wegbreit'75)

static cost analysis

bound the cost of executing program P on any input data $\overline{x}$ without having to actually *run* $P(\overline{x})$

- reasoning about execution cost is difficult and error-prone
- cost analysis, or resource analysis or complexity analysis should be automatic

Introduction to cost analysis (Wegbreit'75)

static cost analysis

bound the cost of executing program P on any input data $\overline{x}$ without having to actually *run* $P(\overline{x})$

- reasoning about execution cost is difficult and error-prone
- cost analysis, or resource analysis or complexity analysis should be automatic
- applications:
  - performance debugging and validation
  - resource bound certification
  - program synthesis and optimization
  - scheduling distributed execution

Introduction to cost analysis (Wegbreit'75)

- cost model: specify the resource of interest
    - number of executed instructions
    - memory consumption
    - number of calls to method

Introduction to cost analysis (Wegbreit'75)

- cost model: specify the resource of interest
  - number of executed instructions
  - memory consumption
  - number of calls to method
- two phases:
  1. produce a system of recursive equations which capture the cost of the program in terms of the size of its input data.

$$\begin{array}{ll} \mathcal{C}_1(a,j,v) = 3 & \{j >= a\} \\ \mathcal{C}_1(a,j,v) = 8 & \{j < a\} \\ \mathcal{C}_1(a,j,v) = 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\} \end{array}$$

Introduction to cost analysis (Wegbreit'75)

- cost model: specify the resource of interest
  - number of executed instructions
  - memory consumption
  - number of calls to method
- two phases:
  1. produce a system of recursive equations which capture the cost of the program in terms of the size of its input data.

$$\begin{array}{ll} \mathcal{C}_1(a,j,v) = 3 & \{j >= a\} \\ \mathcal{C}_1(a,j,v) = 8 & \{j < a\} \\ \mathcal{C}_1(a,j,v) = 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\} \end{array}$$

  2. compute *closed-forms* for them, i.e., cost expressions which are not in recursive form

$$\mathcal{C}_1(a,j,v) = 8 + 17 * nat(a-j)$$

From Termination to Cost

from cost to termination

if the cost model assigns a non-zero cost to all instructions, finding an upper bound implies termination

From Termination to Cost

from cost to termination

if the cost model assigns a non-zero cost to all instructions, finding an upper bound implies termination

from termination to cost

techniques used in termination analysis are very useful in cost analysis

From Termination to Cost

from cost to termination

if the cost model assigns a non-zero cost to all instructions, finding an upper bound implies termination

from termination to cost

techniques used in termination analysis are very useful in cost analysis

- phase 1: abstract programs are instrumental to build to recurrence relations
- phase 2: ranking functions can be used to (upper bound) bound the number of iterations

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{aligned}
&\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
&\quad aload(a, s_0), \\
&\quad iload(j, s_1), \\
&\quad iconst(1, s_2), \\
&\quad isub(s1, s_2, s_1), \\
&\quad iload(v, s_2), \\
&\quad iastore(s_0, s_1, s_2), \\
&\quad iinc(i, -1), \\
&\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{aligned}$$

$$\begin{aligned}
&\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
&\quad s_0 = a, \\
&\quad s_1 = j, \\
&\quad s_2 = 1, \\
&\quad s_1' = s_1 - s_2, \\
&\quad s_2' = v, \\
&\quad true, \\
&\quad i' = i - 1, \\
&\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{aligned}$$

$$\mathcal{B}_4(a, i, j, v) =$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{array}{l}\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) := \\ \quad aload(a,s_0), \\ \quad iload(j,s_1), \\ \quad iconst(1,s_2), \\ \quad isub(s1,s_2,s_1), \\ \quad iload(v,s_2), \\ \quad iastore(s_0,s_1,s_2), \\ \quad iinc(i,-1), \\ \quad \mathcal{B}_1(\langle a,i\rangle,\langle\rangle).\end{array}$$

$$\begin{array}{l}\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) := \\ \quad s_0 = a, \\ \quad s_1 = j, \\ \quad s_2 = 1, \\ \quad s_1' = s_1 - s_2, \\ \quad s_2' = v, \\ \quad true, \\ \quad i' = i - 1, \\ \quad \mathcal{B}_1(\langle a,i'\rangle,\langle\rangle).\end{array}$$

$$\begin{array}{l}\mathcal{B}_4(a,i,j,v) = \\ \quad 1+\end{array}$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v, \\
\quad true, \\
\quad i' = i - 1, \\
\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(a, i, j, v) = \\
\quad 1+ \\
\quad 1+
\end{array}$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$
\begin{aligned}
&\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
&\quad aload(a, s_0), \\
&\quad iload(j, s_1), \\
&\quad iconst(1, s_2), \\
&\quad isub(s1, s_2, s_1), \\
&\quad iload(v, s_2), \\
&\quad iastore(s_0, s_1, s_2), \\
&\quad iinc(i, -1), \\
&\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
&\quad s_0 = a, \\
&\quad s_1 = j, \\
&\quad s_2 = 1, \\
&\quad s_1' = s_1 - s_2, \\
&\quad s_2' = v, \\
&\quad true, \\
&\quad i' = i - 1, \\
&\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{B}_4(a, i, j, v) = \\
&\quad 1+ \\
&\quad 1+ \\
&\quad 1+
\end{aligned}
$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$
\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}
$$

$$
\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v, \\
\quad true, \\
\quad i' = i - 1, \\
\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{array}
$$

$$
\begin{array}{l}
\mathcal{B}_4(a, i, j, v) = \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+
\end{array}
$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{array}{l}\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) := \\ \quad aload(a,s_0), \\ \quad iload(j,s_1), \\ \quad iconst(1,s_2), \\ \quad isub(s1,s_2,s_1), \\ \quad iload(v,s_2), \\ \quad iastore(s_0,s_1,s_2), \\ \quad iinc(i,-1), \\ \quad \mathcal{B}_1(\langle a,i\rangle,\langle\rangle).\end{array}$$

$$\begin{array}{l}\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) := \\ \quad s_0 = a, \\ \quad s_1 = j, \\ \quad s_2 = 1, \\ \quad s_1' = s_1 - s_2, \\ \quad s_2' = v, \\ \quad true, \\ \quad i' = i - 1, \\ \quad \mathcal{B}_1(\langle a,i'\rangle,\langle\rangle).\end{array}$$

$$\begin{array}{l}\mathcal{B}_4(a,i,j,v) = \\ \quad 1+ \\ \quad 1+ \\ \quad 1+ \\ \quad 1+ \\ \quad 1+\end{array}$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$
\begin{aligned}
&\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
&\quad aload(a, s_0), \\
&\quad iload(j, s_1), \\
&\quad iconst(1, s_2), \\
&\quad isub(s1, s_2, s_1), \\
&\quad iload(v, s_2), \\
&\quad iastore(s_0, s_1, s_2), \\
&\quad iinc(i, -1), \\
&\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
&\quad s_0 = a, \\
&\quad s_1 = j, \\
&\quad s_2 = 1, \\
&\quad s_1' = s_1 - s_2, \\
&\quad s_2' = v, \\
&\quad true, \\
&\quad i' = i - 1, \\
&\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{B}_4(a, i, j, v) = \\
&\quad 1+ \\
&\quad 1+ \\
&\quad 1+ \\
&\quad 1+ \\
&\quad 1+ \\
&\quad 1+
\end{aligned}
$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v, \\
\quad true, \\
\quad i' = i - 1, \\
\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(a, i, j, v) = \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+
\end{array}$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad aload(a, s_0), \\
\quad iload(j, s_1), \\
\quad iconst(1, s_2), \\
\quad isub(s1, s_2, s_1), \\
\quad iload(v, s_2), \\
\quad iastore(s_0, s_1, s_2), \\
\quad iinc(i, -1), \\
\quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v, \\
\quad true, \\
\quad i' = i - 1, \\
\quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(a, i, j, v) = \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad \mathcal{B}_1(a, i'),
\end{array}$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{array}{l}
\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) := \\
\quad aload(a,s_0), \\
\quad iload(j,s_1), \\
\quad iconst(1,s_2), \\
\quad isub(s1,s_2,s_1), \\
\quad iload(v,s_2), \\
\quad iastore(s_0,s_1,s_2), \\
\quad iinc(i,-1), \\
\quad \mathcal{B}_1(\langle a,i\rangle,\langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(\langle a,i,j,v\rangle,\langle\rangle) := \\
\quad s_0 = a, \\
\quad s_1 = j, \\
\quad s_2 = 1, \\
\quad s_1' = s_1 - s_2, \\
\quad s_2' = v, \\
\quad true, \\
\quad i' = i - 1, \\
\quad \mathcal{B}_1(\langle a,i'\rangle,\langle\rangle).
\end{array}$$

$$\begin{array}{l}
\mathcal{B}_4(a,i,j,v) = \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad 1+ \\
\quad \mathcal{B}_1(a,i'), \\
\quad \{i' = i - 1\}
\end{array}$$

Phase 1: Generation of recurrence equations

The abstract compilation obtained by the termination module is used to generate the recurrence relations

$$\begin{array}{l} \mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\ \quad aload(a, s_0), \\ \quad iload(j, s_1), \\ \quad iconst(1, s_2), \\ \quad isub(s1, s_2, s_1), \\ \quad iload(v, s_2), \\ \quad iastore(s_0, s_1, s_2), \\ \quad iinc(i, -1), \\ \quad \mathcal{B}_1(\langle a, i\rangle, \langle\rangle). \end{array}$$

$$\begin{array}{l} \mathcal{B}_4(\langle a, i, j, v\rangle, \langle\rangle) := \\ \quad s_0 = a, \\ \quad s_1 = j, \\ \quad s_2 = 1, \\ \quad s_1' = s_1 - s_2, \\ \quad s_2' = v, \\ \quad true, \\ \quad i' = i - 1, \\ \quad \mathcal{B}_1(\langle a, i'\rangle, \langle\rangle). \end{array}$$

$$\begin{array}{l} \mathcal{B}_4(a, i, j, v) = \\ \quad 1+ \\ \quad 1+ \\ \quad 1+ \\ \quad 1+ \\ \quad 1+ \\ \quad 1+ \\ \quad 1+ \\ \quad \mathcal{B}_1(a, i'), \\ \quad \{i' = i - 1\} \end{array}$$

cost equation systems

Given a rule $\mathrm{p}(\langle\bar{\mathrm{x}}\rangle, \langle\bar{\mathrm{y}}\rangle) := \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$ and φ_r its corresponding size relations. The cost equation is: $p(\bar{x}) = \Sigma_{i=1}^{n} M(\mathrm{b}_\mathrm{i}), \varphi_r$

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$
\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a, i) && \{i = a - 2\} \\
\mathcal{B}_1(a, i) &= 2 && \{i < 0\} \\
\mathcal{B}_1(a, i) &= 18 + \mathcal{C}_1(a, j, v) + \mathcal{B}_1(a, i') && \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a, j, v) &= 3 && \{j \geq a\} \\
\mathcal{C}_1(a, j, v) &= 8 && \{j < a\} \\
\mathcal{C}_1(a, j, v) &= 17 + \mathcal{C}_1(a, j', v) && \{j < a, j' = j + 1\}
\end{aligned}
$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a-2\} \\
\mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i-1, j = i+1\} \\
\mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\}
\end{array}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

The cost of sort depends on *a*, the length of *a*

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$
\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a - 2\} \\
\mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j + 1\}
\end{array}
$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

It costs 6 units that correspond to the initialization of *i*. Initialization reflected in constraints!

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a,i) && \{i = a - 2\}\\
\mathcal{B}_1(a,i) &= 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i \geq 0, i' = i - 1, j = i + 1\}\\
\mathcal{C}_1(a,j,v) &= 3 && \{j \geq a\}\\
\mathcal{C}_1(a,j,v) &= 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j + 1\}
\end{aligned}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

Plus the cost of the for loop $\mathcal{B}_1$

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$
\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a - 2\} \\
\mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j + 1\}
\end{array}
$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

The first case when we do not enter the loop

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a,i) && \{i = a - 2\}\\
\mathcal{B}_1(a,i) &= 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i \geq 0, i' = i - 1, j = i + 1\}\\
\mathcal{C}_1(a,j,v) &= 3 && \{j \geq a\}\\
\mathcal{C}_1(a,j,v) &= 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j + 1\}
\end{aligned}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

This costs 2 which corresponds to the comparison

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a,i) && \{i = a-2\}\\
\mathcal{B}_1(a,i) &= 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i \geq 0, i' = i-1, j = i+1\}\\
\mathcal{C}_1(a,j,v) &= 3 && \{j \geq a\}\\
\mathcal{C}_1(a,j,v) &= 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

The second case when we enter the loop

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$
\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a,i) && \{i = a-2\}\\
\mathcal{B}_1(a,i) &= 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i \geq 0, i' = i-1, j = i+1\}\\
\mathcal{C}_1(a,j,v) &= 3 && \{j \geq a\}\\
\mathcal{C}_1(a,j,v) &= 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}
$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

We add 18 units which correspond to comparison, instructions before and after the while

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{array}{rll} sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a - 2\} \\ \mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\ \mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\ \mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\ \mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\ \mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j + 1\} \end{array}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

The cost of executing the while. The constraint $j = i + 1$ is the initial value of j

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a,i) && \{i = a - 2\}\\
\mathcal{B}_1(a,i) &= 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i \geq 0, i' = i - 1, j = i + 1\}\\
\mathcal{C}_1(a,j,v) &= 3 && \{j \geq a\}\\
\mathcal{C}_1(a,j,v) &= 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j + 1\}
\end{aligned}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

and the cost of executing the for loop again after decreasing *i*

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a - 2\} \\
\mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j + 1\}
\end{array}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

the first equation captures the case where we do not enter the loop because the first condition does not hold

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a - 2\} \\
\mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j + 1\}
\end{array}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

the cost is 3 which corresponds to the comparison

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$
\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a - 2\} \\
\mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j + 1\}
\end{array}
$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

the second equation when the first condition holds and the second does not. The condition $a[i] > v$ has been lost by the size abstraction!

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{array}{rll} \mathit{sort}(a) = & 6+\mathcal{B}_1(a,i) & \{i=a-2\} \\ \mathcal{B}_1(a,i) = & 2 & \{i<0\} \\ \mathcal{B}_1(a,i) = & 18+\mathcal{C}_1(a,j,v)+\mathcal{B}_1(a,i') & \{i\geq 0, i'=i-1, j=i+1\} \\ \mathcal{C}_1(a,j,v) = & 3 & \{j\geq a\} \\ \mathcal{C}_1(a,j,v) = & 8 & \{j<a\} \\ \mathcal{C}_1(a,j,v) = & 17+\mathcal{C}_1(a,j',v) & \{j<a, j'=j+1\} \end{array}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

The cost is 8 which corresponds to the two comparisons

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$
\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a, i) & \{i = a - 2\} \\
\mathcal{B}_1(a, i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a, i) = & 18 + \mathcal{C}_1(a, j, v) + \mathcal{B}_1(a, i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a, j, v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a, j, v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a, j, v) = & 17 + \mathcal{C}_1(a, j', v) & \{j < a, j' = j + 1\}
\end{array}
$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

The third equation is when both conditions hold. It is not mutually recursive with the second one!

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{array}{rll}
sort(a) = & 6 + \mathcal{B}_1(a,i) & \{i = a - 2\} \\
\mathcal{B}_1(a,i) = & 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a,j,v) = & 3 & \{j \geq a\} \\
\mathcal{C}_1(a,j,v) = & 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j + 1\}
\end{array}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

It costs 17 which is the cost of the comparisons plus the body of the while

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a,i) && \{i = a-2\}\\
\mathcal{B}_1(a,i) &= 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i \geq 0, i' = i-1, j = i+1\}\\
\mathcal{C}_1(a,j,v) &= 3 && \{j \geq a\}\\
\mathcal{C}_1(a,j,v) &= 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

plus executing the while again after incrementing j

Phase 1: Generation of recurrence equations

The result of generating the cost equations for all program rules is:

$$
\begin{aligned}
sort(a) &= 6 + \mathcal{B}_1(a, i) && \{i = a - 2\} \\
\mathcal{B}_1(a, i) &= 2 && \{i < 0\} \\
\mathcal{B}_1(a, i) &= 18 + \mathcal{C}_1(a, j, v) + \mathcal{B}_1(a, i') && \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{C}_1(a, j, v) &= 3 && \{j \geq a\} \\
\mathcal{C}_1(a, j, v) &= 8 && \{j < a\} \\
\mathcal{C}_1(a, j, v) &= 17 + \mathcal{C}_1(a, j', v) && \{j < a, j' = j + 1\}
\end{aligned}
$$

```
static void sort(int a[]) {
 for (int i=a.length-2; i≥0; i--) {
  ...
  while ( j<a.length && a[j]<v) {
   ...
   j++
  }
  ...
 }
```

Partial Evaluation: equations are converted into directly recursive form by applying the well-known technique of partial evaluation

Phase 2: Solving the cost equations

Solving the equations with CAS

Though syntactically similar to standard recurrence relations, their additional features make them not solvable using CAS (like MAPLE, MAXIMA,...)

Phase 2: Solving the cost equations

Solving the equations with CAS

Though syntactically similar to standard recurrence relations, their additional features make them not solvable using CAS (like MAPLE, MAXIMA,...)

- Additional features:
  - Multiple arguments $\mathcal{C}_1(a, j, v) = 3 \quad \{j >= a\}$
  - Inexact size relations $\mathcal{C}_1(a, j, v) = 8 \quad \{j < a\}$
  - Non-deterministic $\mathcal{C}_1(a, j, v) = 17 + \mathcal{C}_1(a, j', v)$

$$\{j < a, j' = j + 1\}$$

Phase 2: Solving the cost equations

Solving the equations with CAS

Though syntactically similar to standard recurrence relations, their additional features make them not solvable using CAS (like MAPLE, MAXIMA,...)

- Additional features:
  - Multiple arguments $\mathcal{C}_1(a, j, v) = 3 \quad \{j >= a\}$
  - Inexact size relations $\mathcal{C}_1(a, j, v) = 8 \quad \{j < a\}$
  - Non-deterministic $\mathcal{C}_1(a, j, v) = 17 + \mathcal{C}_1(a, j', v)$ $\{j < a, j' = j + 1\}$
- A precise solution often does not exist:

Phase 2: Solving the cost equations

Solving the equations with CAS

Though syntactically similar to standard recurrence relations, their additional features make them not solvable using CAS (like MAPLE, MAXIMA,...)

- Additional features:
  - Multiple arguments $\mathcal{C}_1(a, j, v) = 3 \quad \{j >= a\}$
  - Inexact size relations $\mathcal{C}_1(a, j, v) = 8 \quad \{j < a\}$
  - Non-deterministic $\mathcal{C}_1(a, j, v) = 17 + \mathcal{C}_1(a, j', v) \quad \{j < a, j' = j + 1\}$
- A precise solution often does not exist:
  - upper-bounds on the worst case cost
  - lower-bounds on the best case cost

Phase 2: Solving the cost equations (upper bounds)

basic idea for upper bound

Given a loop (or cost relation C), its upper bound can be computed as $\boxed{\textbf{UB} = \textbf{\# iter}*\textbf{max\_cost} + \textbf{max\_base}}$:

Phase 2: Solving the cost equations (upper bounds)

basic idea for upper bound

Given a loop (or cost relation C), its upper bound can be computed as $\boxed{\textbf{UB} = \textbf{\# iter}*\textbf{max\_cost} + \textbf{max\_base}}$:

- **# iter**: bound the number of iterations in loop (or chain of recursive calls in the relation)

Phase 2: Solving the cost equations (upper bounds)

basic idea for upper bound

Given a loop (or cost relation C), its upper bound can be computed as $\boxed{\textbf{UB} = \textbf{\# iter} * \textbf{max\_cost} + \textbf{max\_base}}$:

- **# iter**: bound the number of iterations in loop (or chain of recursive calls in the relation)
    - a ranking function $f(\bar{x})$ for C guarantees that the length of any chain of recursive calls to C cannot exceed $f(\bar{v})$

Phase 2: Solving the cost equations (upper bounds)

basic idea for upper bound

Given a loop (or cost relation C), its upper bound can be computed as $\boxed{\textbf{UB} = \textbf{\# iter}*\textbf{max\_cost} + \textbf{max\_base}}$:

- **# iter**: bound the number of iterations in loop (or chain of recursive calls in the relation)
  - a ranking function $f(\bar{x})$ for C guarantees that the length of any chain of recursive calls to C cannot exceed $f(\bar{v})$
- **max_cost/max_base**: bound the maximal cost of expression

Phase 2: Solving the cost equations (upper bounds)

basic idea for upper bound

Given a loop (or cost relation C), its upper bound can be computed as $\boxed{\textbf{UB} = \textbf{\# iter} * \textbf{max\_cost} + \textbf{max\_base}}$:

- **# iter**: bound the number of iterations in loop (or chain of recursive calls in the relation)
    - a ranking function $f(\bar{x})$ for C guarantees that the length of any chain of recursive calls to C cannot exceed $f(\bar{v})$
- **max_cost/max_base**: bound the maximal cost of expression

upper bound

We then look at the shape of equations $C(\bar{x}) = expr + C(\bar{y}) + \ldots + C(\bar{w})$:

Phase 2: Solving the cost equations (upper bounds)

basic idea for upper bound

Given a loop (or cost relation C), its upper bound can be computed as $\boxed{\textbf{UB} = \textbf{\# iter}*\textbf{max\_cost} + \textbf{max\_base}}$:

- **# iter**: bound the number of iterations in loop (or chain of recursive calls in the relation)
  - a ranking function $f(\bar{x})$ for C guarantees that the length of any chain of recursive calls to C cannot exceed $f(\bar{v})$
- **max_cost/max_base**: bound the maximal cost of expression

upper bound

We then look at the shape of equations $C(\bar{x}) = expr + C(\bar{y}) + \ldots + C(\bar{w})$:

- one recursive call: $f(\bar{x})*\textbf{max\_cost(expr)}$

Phase 2: Solving the cost equations (upper bounds)

basic idea for upper bound

Given a loop (or cost relation C), its upper bound can be computed as $\boxed{\textbf{UB} = \textbf{\# iter} * \textbf{max\_cost} + \textbf{max\_base}}$:

- **# iter**: bound the number of iterations in loop (or chain of recursive calls in the relation)
    - a ranking function $f(\bar{x})$ for C guarantees that the length of any chain of recursive calls to C cannot exceed $f(\bar{v})$
- **max_cost/max_base**: bound the maximal cost of expression

upper bound

We then look at the shape of equations $C(\bar{x}) = expr + C(\bar{y}) + \ldots + C(\bar{w})$:

- one recursive call: $f(\bar{x}) * \textbf{max\_cost(expr)}$
- n recursive calls: $n^{f(\bar{x})} * \textbf{max\_cost(expr)}$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \# \textbf{ iter*max\_cost} + \textbf{max\_base}}$$

$$\begin{array}{ll}
\mathcal{C}_1(a,j,v) = 3 & \{j >= a\} \\
\mathcal{C}_1(a,j,v) = 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\}
\end{array}$$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \textcolor{red}{\#\ \textbf{iter}}\textbf{*max\_cost+max\_base}}$$

$$\begin{array}{ll}
\mathcal{C}_1(a,j,v) = 3 & \{j >= a\} \\
\mathcal{C}_1(a,j,v) = 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\}
\end{array}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j)$$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \#\ \textbf{iter*max\_cost+max\_base}}$$

$$\begin{array}{ll} \mathcal{C}_1(a,j,v) = 3 & \{j >= a\} \\ \mathcal{C}_1(a,j,v) = 8 & \{j < a\} \\ \mathcal{C}_1(a,j,v) = 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\} \end{array}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17$$

Phase 2: Solving the cost equations (example)

UB = # iter*max_cost+max_base

$$
\begin{array}{ll}
\mathcal{C}_1(a,j,v) = 3 & \{j >= a\} \\
\mathcal{C}_1(a,j,v) = 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) = 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\}
\end{array}
$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \textbf{\# iter*max\_cost+max\_base}}$$

$$\begin{aligned}
\mathcal{C}_1(a,j,v) &= 3 & \{j >= a\}\\
\mathcal{C}_1(a,j,v) &= 8 & \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\}
\end{aligned}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

$$\begin{aligned}
\mathcal{B}_1(a,i) &= 2 & \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}$$

Phase 2: Solving the cost equations (example)

UB = # iter*max_cost+max_base

$$\begin{aligned}
\mathcal{C}_1(a,j,v) =& 3 & \{j >= a\} \\
\mathcal{C}_1(a,j,v) =& 8 & \{j < a\} \\
\mathcal{C}_1(a,j,v) =& 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\}
\end{aligned}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

$$\begin{aligned}
\mathcal{B}_1(a,i) =& 2 & \{i < 0\} \\
\mathcal{B}_1(a,i) =& 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}$$

$$\mathcal{B}_1(a,i) = nat(i+1)$$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \textbf{\# iter*}\textbf{max\_cost}+\textbf{max\_base}}$$

$$
\begin{aligned}
\mathcal{C}_1(a,j,v) =& 3 && \{j >= a\}\\
\mathcal{C}_1(a,j,v) =& 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) =& 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}
$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

$$
\begin{aligned}
\mathcal{B}_1(a,i) =& 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) =& 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}
$$

$$\mathcal{B}_1(a,i) = nat(i+1) * \textbf{max\_cost}[18 + \mathcal{C}_1(a,j,v)]$$

Phase 2: Solving the cost equations (example)

UB = # iter*max_cost+max_base

$$
\begin{aligned}
\mathcal{C}_1(a,j,v) =& 3 && \{j >= a\}\\
\mathcal{C}_1(a,j,v) =& 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) =& 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}
$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

$$
\begin{aligned}
\mathcal{B}_1(a,i) =& 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) =& 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}
$$

$$\mathcal{B}_1(a,i) = nat(i+1) * \textbf{max\_cost}[18 + nat(a-j) * 17 + 8]$$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \#\ \textbf{iter}{*}\textbf{max\_cost}{+}\textbf{max\_base}}$$

$$\begin{aligned}
\mathcal{C}_1(a,j,v) &= 3 && \{j >= a\}\\
\mathcal{C}_1(a,j,v) &= 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) &= 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

$$\begin{aligned}
\mathcal{B}_1(a,i) &= 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) &= 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}$$

$$\mathcal{B}_1(a,i) = nat(i+1) * (18 + \textbf{max\_cost}[nat(a-j)] * 17 + 8)$$

Phase 2: Solving the cost equations (example)

UB = # iter*max_cost+max_base

$$
\begin{aligned}
\mathcal{C}_1(a,j,v) = & 3 && \{j >= a\} \\
\mathcal{C}_1(a,j,v) = & 8 && \{j < a\} \\
\mathcal{C}_1(a,j,v) = & 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}
$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

$$
\begin{aligned}
\mathcal{B}_1(a,i) = & 2 && \{i < 0\} \\
\mathcal{B}_1(a,i) = & 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}
$$

$$\mathcal{B}_1(a,i) = nat(i+1) * (18 + nat(a-1) * 17 + 8)$$

Phase 2: Solving the cost equations (example)

UB = # iter*max_cost+max_base

$$\begin{aligned}
\mathcal{C}_1(a,j,v) =& 3 && \{j >= a\}\\
\mathcal{C}_1(a,j,v) =& 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) =& 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j)*17+8$$

$$\begin{aligned}
\mathcal{B}_1(a,i) =& 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) =& 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}$$

$$\mathcal{B}_1(a,i) = nat(i+1)*(nat(a-1)*17+26)$$

Phase 2: Solving the cost equations (example)

UB = # iter*max_cost+max_base

$$\begin{aligned}
\mathcal{C}_1(a,j,v) =& 3 & \{j >= a\}\\
\mathcal{C}_1(a,j,v) =& 8 & \{j < a\}\\
\mathcal{C}_1(a,j,v) =& 17 + \mathcal{C}_1(a,j',v) & \{j < a, j' = j+1\}
\end{aligned}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j) * 17 + 8$$

$$\begin{aligned}
\mathcal{B}_1(a,i) =& 2 & \{i < 0\}\\
\mathcal{B}_1(a,i) =& 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') & \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}$$

$$\mathcal{B}_1(a,i) = nat(i+1) * (nat(a-1) * 17 + 26) + 2$$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \#\ \textbf{iter*max\_cost+max\_base}}$$

$$\begin{aligned}
\mathcal{C}_1(a,j,v) =& 3 && \{j >= a\}\\
\mathcal{C}_1(a,j,v) =& 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) =& 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j)*17+8$$

$$\begin{aligned}
\mathcal{B}_1(a,i) =& 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) =& 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}$$

$$\mathcal{B}_1(a,i) = nat(i+1)*(nat(a-1)*17+26)+2$$

$$sort(a) = 6 + \mathcal{B}_1(a,i) \quad \{i = a-2\}$$

Phase 2: Solving the cost equations (example)

$$\boxed{\textbf{UB} = \textbf{\# iter*max\_cost+max\_base}}$$

$$\begin{aligned}
\mathcal{C}_1(a,j,v) =& 3 && \{j >= a\}\\
\mathcal{C}_1(a,j,v) =& 8 && \{j < a\}\\
\mathcal{C}_1(a,j,v) =& 17 + \mathcal{C}_1(a,j',v) && \{j < a, j' = j+1\}
\end{aligned}$$

$$\mathcal{C}_1(a,j,v) = nat(a-j)*17+8$$

$$\begin{aligned}
\mathcal{B}_1(a,i) =& 2 && \{i < 0\}\\
\mathcal{B}_1(a,i) =& 18 + \mathcal{C}_1(a,j,v) + \mathcal{B}_1(a,i') && \{i >= 0, i' = i-1, j = i+1\}
\end{aligned}$$

$$\mathcal{B}_1(a,i) = nat(i+1)*(nat(a-1)*17+26)+2$$

$$sort(a) = 6 + \mathcal{B}_1(a,i) \quad \{i = a-2\}$$

$$sort(a) = 8 + nat(a-1)*(nat(a-1)*17+26)$$

Phase 2: Solving the cost equations (theorem)

Theorem (soundness)

- *Let* $P(\bar{x})$ *be a method,*
- M *a cost model,*
- $UB(\bar{x})$ *the upper bound computed from* P*.*

For any valid input $\bar{v}$*, if there exists a trace* t *from* $P(\bar{v})$*, then we ensure* $\boxed{UB(\bar{v}) \geq M(t)}$

Conclusions (Part 2)

- Techniques developed in termination useful in cost analysis:
  - Abstract rules used to generate cost equations
  - Ranking functions bound the number of iterations

Conclusions (Part 2)

- Techniques developed in termination useful in cost analysis:
    - Abstract rules used to generate cost equations
    - Ranking functions bound the number of iterations
- Powerful approach to cost analysis: logarithmic, polynomial, exponential bounds [ESOP'07]

Conclusions (Part 2)

- Techniques developed in termination useful in cost analysis:
  - Abstract rules used to generate cost equations
  - Ranking functions bound the number of iterations
- Powerful approach to cost analysis: logarithmic, polynomial, exponential bounds [ESOP'07]
- Enhanced equations which capture the behaviour of GC [ISMM'07, ISMM'09, ISMM'10]

Conclusions (Part 2)

- Techniques developed in termination useful in cost analysis:
  - Abstract rules used to generate cost equations
  - Ranking functions bound the number of iterations
- Powerful approach to cost analysis: logarithmic, polynomial, exponential bounds [ESOP'07]
- Enhanced equations which capture the behaviour of GC [ISMM'07, ISMM'09, ISMM'10]
- Solving cost relations requires powerful solvers: non-deterministic relations, multiple arguments, size constraints [SAS'08, JAR'10]

Conclusions (Part 2)

- Techniques developed in termination useful in cost analysis:
    - Abstract rules used to generate cost equations
    - Ranking functions bound the number of iterations
- Powerful approach to cost analysis: logarithmic, polynomial, exponential bounds [ESOP'07]
- Enhanced equations which capture the behaviour of GC [ISMM'07, ISMM'09, ISMM'10]
- Solving cost relations requires powerful solvers: non-deterministic relations, multiple arguments, size constraints [SAS'08, JAR'10]
- Comparing cost functions [FOPARA'09]

Conclusions (Part 2)

- Techniques developed in termination useful in cost analysis:
  - Abstract rules used to generate cost equations
  - Ranking functions bound the number of iterations
- Powerful approach to cost analysis: logarithmic, polynomial, exponential bounds [ESOP'07]
- Enhanced equations which capture the behaviour of GC [ISMM'07, ISMM'09, ISMM'10]
- Solving cost relations requires powerful solvers: non-deterministic relations, multiple arguments, size constraints [SAS'08, JAR'10]
- Comparing cost functions [FOPARA'09]
- Asymptotic bounds [APLAS'09]

PART 3: FIELD-SENSITIVE ANALYSIS

Shared Mutable Data Structures

- Reasoning about data stored in the *heap* is rather difficult:

Shared Mutable Data Structures

- Reasoning about data stored in the *heap* is rather difficult:
  while $(i<\boxed{n})\ \{i{+}{+};o.m();\}$ $\boxed{n\text{-}i}$ is a ranking function

Shared Mutable Data Structures

- Reasoning about data stored in the *heap* is rather difficult:
  while $(i<\boxed{n})\ \{i{+}{+};o.m();\}$ $\boxed{n\text{-}i}$ is a ranking function
  while $(i<\boxed{f.n})\ \{i{+}{+};o.m();\}$ $\boxed{f.n\text{-}i}$ ranking function?

Shared Mutable Data Structures

- Reasoning about data stored in the *heap* is rather difficult:
  while (i<$\boxed{n}$) {i++;o.m();} $\boxed{n\text{-}i}$ is a ranking function
  while (i<$\boxed{f.n}$) {i++;o.m();} $\boxed{f.n\text{-}i}$ ranking function?

- Static analysis of object fields (numeric or references) classified:
  - field-sensitive - approximate them
    precise but inefficient
  - field-insensitive - ignore them
    efficient but imprecise

Shared Mutable Data Structures

- Reasoning about data stored in the *heap* is rather difficult:
  $while\ (i<\boxed{n})\ \{i{+}{+};o.m();\}$ $\boxed{n\text{-}i}$ is a ranking function
  $while\ (i<\boxed{f.n})\ \{i{+}{+};o.m();\}$ $\boxed{f.n\text{-}i}$ ranking function?
- Static analysis of object fields (numeric or references) classified:
  - field-sensitive - approximate them
    precise but inefficient
  - field-insensitive - ignore them
    efficient but imprecise
- *Numeric fields* and *reference fields* are used all the time in real programs

Shared Mutable Data Structures

- Reasoning about data stored in the *heap* is rather difficult:
  while $(i<\boxed{n})\ \{i{+}{+};o.m();\}$ $\boxed{n-i}$ is a ranking function
  while $(i<\boxed{f.n})\ \{i{+}{+};o.m();\}$ $\boxed{f.n-i}$ ranking function?
- Static analysis of object fields (numeric or references) classified:
  - field-sensitive - approximate them
    precise but inefficient
  - field-insensitive - ignore them
    efficient but imprecise
- *Numeric fields* and *reference fields* are used all the time in real programs

Challenge:

develop techniques that have good balance between:

- accuracy of analysis,
- computational cost.

Field-sensitive analysis by field-insensitive analysis

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
    - the memory location does not change

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
    - the memory location does not change ⇒ AI-based static analysis to prove constancy of references

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
    - the memory location does not change ⇒ AI-based static analysis to prove constancy of references
    - write accesses are done through the same memory location

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
    - the memory location does not change ⇒ AI-based static analysis to prove constancy of references
    - write accesses are done through the same memory location ⇒ check after analysis

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
    - the memory location does not change ⇒ AI-based static analysis to prove constancy of references
    - write accesses are done through the same memory location ⇒ check after analysis

```
while ( x != null ) {
  for(; x.c<n; x.c++)
    value[x.c]++;
  x=x.next;
}
```

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
    - the memory location does not change ⇒ AI-based static analysis to prove constancy of references
    - write accesses are done through the same memory location ⇒ check after analysis

```
while ( x != null ) {
  for(; x.c<n; x.c++)
    value[x.c]++;
  x=x.next;
}
```

```
while (x.size > 0) {
  x.size++;
  y.size--;
}
```

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
    - the memory location does not change ⇒ AI-based static analysis to prove constancy of references
    - write accesses are done through the same memory location ⇒ check after analysis

```
while ( x != null ) {
  for(; x.c<n; x.c++)
    value[x.c]++;
  x=x.next;
}
```

```
while (x.size > 0) {
  x.size++;
  y.size--;
}
```

3. transform the code to replace local fields by variables

Field-sensitive analysis by field-insensitive analysis

1. analyze the behavior of scopes or program fragments
2. model only those fields which behave as local variables
   - the memory location does not change ⇒ AI-based static analysis to prove constancy of references
   - write accesses are done through the same memory location ⇒ check after analysis

```
while ( x != null ) {
  for(; x.c<n; x.c++)
    value[x.c]++;
  x=x.next;
}
```

```
while ( x != null ) {
  g=x.c ;
  for(; g<n; g++)
    value[g]++;
  x.c=g ;
  x=x.next;
}
```

3. transform the code to replace local fields by variables
4. infer information on the fields through associated ghost variables

Conclusions and Related Work

- COSTA has been the first termination analyzer for sequential Java Bytecode
    - It deals with Java libraries
    - It checks termination and computes upper bounds
    - It allows assertions on upper bounds (and thus termination)

Conclusions and Related Work

- COSTA has been the first termination analyzer for sequential Java Bytecode
    - It deals with Java libraries
    - It checks termination and computes upper bounds
    - It allows assertions on upper bounds (and thus termination)
- Julia had many components (nullity, class, path-length analyses) and recently has integrated a termination analyzer [Spoto et al., Toplas'10]

Conclusions and Related Work

- COSTA has been the first termination analyzer for sequential Java Bytecode
  - It deals with Java libraries
  - It checks termination and computes upper bounds
  - It allows assertions on upper bounds (and thus termination)
- Julia had many components (nullity, class, path-length analyses) and recently has integrated a termination analyzer [Spoto et al., Toplas'10]
- AProVe had many powerful termination techniques for TRS and now translates Java bytecode to TRS [Otto et al, RTA'10]

Conclusions and Related Work

- First analysis to support numeric [FM'09] and reference fields [SAS'10] in cost/termination analysis of OO bytecode.

Conclusions and Related Work

- First analysis to support numeric [FM'09] and reference fields [SAS'10] in cost/termination analysis of OO bytecode.
- Allows significant accuracy gains at a reasonable overhead.

Conclusions and Related Work

- First analysis to support numeric [FM'09] and reference fields [SAS'10] in cost/termination analysis of OO bytecode.
- Allows significant accuracy gains at a reasonable overhead.
- Existing approaches based on static analysis:
  - Track all possible updates of fields (inefficient), or
  - Abstract all field updates into a single element (inaccurate)

Conclusions and Related Work

- First analysis to support numeric [FM'09] and reference fields [SAS'10] in cost/termination analysis of OO bytecode.
- Allows significant accuracy gains at a reasonable overhead.
- Existing approaches based on static analysis:
  - Track all possible updates of fields (inefficient), or
  - Abstract all field updates into a single element (inaccurate)
- Related work on field-sensitive analysis:
  - The analysis for C programs in [Miné06] enriches the abstract domain to be field sensitive.
  - The notion of restricted variables [AikenFKT'03] is related to our analysis to prove constancy of references.
  - Also related are the notions of local reasoning [OHearnRY'01] and separation logic [Reynolds'02].

Systems and Contact

- COSTA Team (UCM+UPM): Puri Arenas, Samir Genaim, Miguel G-Zamalloa, Germán Puebla, Damiano Zanardini, etc.

Systems and Contact

- COSTA Team (UCM+UPM): Puri Arenas, Samir Genaim, Miguel G-Zamalloa, Germán Puebla, Damiano Zanardini, etc.
- More information on the COSTA system can be found at `http://costa.ls.fi.upm.es`
  (or google "The COSTA System")

Systems and Contact

- COSTA Team (UCM+UPM): Puri Arenas, Samir Genaim, Miguel G-Zamalloa, Germán Puebla, Damiano Zanardini, etc.
- More information on the COSTA system can be found at `http://costa.ls.fi.upm.es`
  (or google "The COSTA System")
- COSTA will shortly be released under the General Public License.
- For information about the upcoming release and other issues, you may consider joining the list `costa-users @ listas.fi.upm.es`

Locality Conditions Numeric Fields

Sufficient conditions

1. The memory location where the field is stored does not change.
2. All write accesses done through the same reference (not aliases).

```
while (x.f.getSize() > 0)
   i+=y.getSize();
   x.f.setSize(x.f.getSize()-1);
```

Locality Conditions Numeric Fields

Sufficient conditions

1. The memory location where the field is stored does not change.
2. All write accesses done through the same reference (not aliases).

```
while (x.f.getSize() > 0)
   i+=y.getSize();
   x.f.setSize(x.f.getSize()-1);
```

```
if (k > 0)
then x=z else x=y;
x.f=10;
for(; i<x.f; i++)
  b[i]=x.b[i];
```

Locality Conditions Numeric Fields

Sufficient conditions

1. The memory location where the field is stored does not change.
2. All write accesses done through the same reference (not aliases).

```
while (x.f.getSize() > 0)
   i+=y.getSize();
   x.f.setSize(x.f.getSize()-1);
```

```
if (k > 0)
then x=z else x=y;
x.f=10;
for(; i<x.f; i++)
  b[i]=x.b[i];
```

```
while ( x != null ) {
  for(; x.c<n; x.c++)
    value[x.c]++;
x=x.next;}
```

Locality Conditions Numeric Fields

Sufficient conditions

1. The memory location where the field is stored does not change.
2. All write accesses done through the same reference (not aliases).

```
while (x.f.getSize() > 0)
   i+=y.getSize();
   x.f.setSize(x.f.getSize()-1);
```

```
if (k > 0)
then x=z else x=y;
x.f=10;
for(; i<x.f; i++)
  b[i]=x.b[i];
```

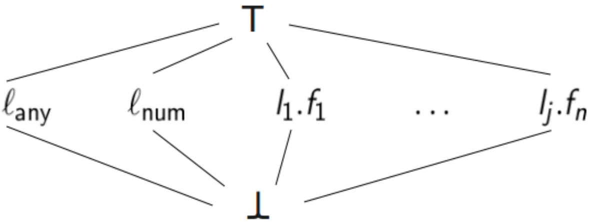
```
while ( x != null ) {
  for(; x.c<n; x.c++)
    value[x.c]++;
x=x.next;}
```

```
while (x.size > 0)
  {x.size++; y.size--;}
```

Cond 1: proving that memory location is constant

reference constancy analysis

- associate an access path to constant reference variables.
- given an entry $p(l_1, \ldots, l_n)$, an *access path* ℓ for a variable y at program point (k, j) is a syntactic construction:
  - ℓ_{any}. Variable y might point to any heap location at (k, j).
  - $l_i.f_1 \ldots .f_h$. Variable y always refers to the same heap location represented by $l_i.f_1 \ldots .f_h$ whenever (k, j) is reached.



Cond 2: write accesses done through the same reference

$R(S, f)/W(S, f)$

Given a scope S and a field signature f, the set of *read/write access paths* is the set of access path of variables y used for reading/writing f in S^*.

$S \equiv$ while (x.f.size > 0) $\{i{=}i{+}y.size;\ \ x.f.size{=}x.f.size{-}1;\}$

x.f.size$=\ell_1.f$ and y.size $= \ell_2$

- $R(S, size) = \{\ell_2, \ell_1.f\}$
- $W(S, size) = \{\ell_1.f\}$

proving condition 2

$W(S, f) = \emptyset$; or $W(S, f) = \{\ell\}$ and ℓ is of the form $l_j.f_1 \ldots f_n$.

Transformation of a Scope S and a Field f

Generate new unique variable names $\bar{v}$ for the local heap locations *ap* to be tracked in the scope S.

1. **Add arguments:** each head or call $p(\langle\bar{x}\rangle, \langle\bar{y}\rangle)$ such that $p \in S$ is converted to $p(\langle\bar{x}{\cdot}\bar{v}_r\rangle, \langle\bar{y}{\cdot}\bar{v}_w\rangle)$
   1. if $W(S, f) = \emptyset$ then $\bar{v}_r = \{v_{ap.f} \mid R(S, f)\}$
   2. if $W(S, f) = \{\ell\}$ then $\bar{v}_r = \{v_{ap.f}\}$
2. **Replicate field accesses:**
   1. each $y.f = x \in S$ produces assignment $v_{ap.f} = x$ if $AP(y) = ap \neq l_{any}$
   2. each $x = y.f \in S$ produces assignment $x = v_{ap.f}$ if $AP(y) = ap \neq l_{any}$
3. **Handle external calls:** external calls $q(\bar{x}, \bar{y}) \in S$ are transformed into $q(\langle\bar{x}{\cdot}\rho(\bar{v}'_r)\rangle), \langle\bar{y}{\cdot}\rho(\bar{v}'_w)\rangle))$

Instrumented Example

(1) $loop(\langle x, y, i\rangle, \langle r\rangle) :=$
$s_0{:=}x, s_0{:=}s_0.f,$
$getSize(\langle s_0\rangle, \langle s_0\rangle),$
$loop_c(\langle x, y, i, s_0\rangle, \langle r\rangle).$

(2) $loop_c(\langle x, y, i, s_0\rangle, \langle r\rangle) :=$
$\mathbf{s_0} \leq \mathbf{0}, s_0{:=}i, r{:=}s_0.$

(3) $loop_c(\langle x, y, i, s_0\rangle, \langle r\rangle) \ :=$
$\mathbf{s_0} > \mathbf{0}, s_0{:=}i, s_1{:=}y, getSize(\langle s_1\rangle, \langle s_1\rangle),$
$s_0{:=}s_0 + s_1, i{:=}s_0, s_0{:=}x, s_0{:=}s_0.f,$
$s_1{:=}s_0, getSize(\langle s_1\rangle, \langle s_1\rangle),$
$s_2{:=}1, s_1{:=}s_1 - s_2, setSize(\langle s_0, s_1\rangle, \langle\rangle),$
$loop(\langle this, x, y, i\rangle, \langle r\rangle).$

(4) $getSize(\langle this\rangle, \langle r\rangle) :=$
$s_0{:=}this, s_0{:=}s_0.size, r{:=}s_0.$

(5) $setSize(\langle this, n\rangle, \langle\rangle) :=$
$s_0{:=}this, s_1{:=}n, s_0.size{:=}s_1.$

Instrumented Example

$$
\begin{array}{ll}
(1) & loop(\langle x, y, i, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle) := \\
& \quad s_0{:=}x, s_0{:=}s_0.f, \\
& \quad getSize(\langle s_0, \underline{v_1}\rangle, \langle s_0\rangle), \\
& \quad loop_c(\langle x, y, i, s_0, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle). \\
(2) & loop_c(\langle x, y, i, s_0, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle) := \\
& \quad \mathbf{s_0} \leq \mathbf{0}, s_0{:=}i, r{:=}s_0. \\
(3) & loop_c(\langle x, y, i, s_0, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle) \ := \\
& \quad \mathbf{s_0} > \mathbf{0}, s_0{:=}i, s_1{:=}y, getSize(\langle s_1, *\rangle, \langle s_1\rangle), \\
& \quad s_0{:=}s_0 + s_1, i{:=}s_0, s_0{:=}x, s_0{:=}s_0.f, \\
& \quad s_1{:=}s_0, getSize(\langle s_1, \underline{v_1}\rangle, \langle s_1\rangle), \\
& \quad s_2{:=}1, s_1{:=}s_1 - s_2, setSize(\langle s_0, s_1\rangle, \langle \underline{v_1}\rangle), \\
& \quad loop(\langle this, x, y, i, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle). \\
(4) & getSize(\langle this, \underline{v_1}\rangle, \langle r\rangle) := \\
& \quad s_0{:=}this, s_0{:=}\underline{v_1}, r{:=}s_0. \\
(5) & setSize(\langle this, n\rangle, \langle \underline{v_1}\rangle) := \\
& \quad s_0{:=}this, s_1{:=}n, \underline{v_1}{:=}s_1.
\end{array}
$$

What is it different in reference fields?

- Replicating instructions is not a good idea:
  - assume an instruction like $y.ref{:=}x$ is followed by $v_{ref}{:=}x$.
  - replicating instructions makes $y.ref$ and v_{ref} alias,
  - therefore, the path-length relations of v_{ref} affected by those of $y.ref$
  - updates to $y.ref$ will force losing path-length information about v_{ref},
  - $\boxed{\text{replace } y.ref{:=}x \text{ by } v_{ref}{:=}x\text{, not replicate}}$
- The locality condition is not always appropriate:
  - clearly, when loops traverse data structures
  - we want to keep track of reference fields which are used as cursors for traversing them
  - reference fields which are part of the data structure itself, seldomly affect termination or cost
  - $\boxed{\text{require that the field signature is both read and written}}$ $(R(S,f)) = (W(S,f)) = \{\ell\}$

Iterator Example

```
class Iter implements Iterator {
  List state; List aux;
  boolean hasNext() {
    return (this.state != null); }
  Object next() {
    this.state = this.state.rest;
    return obj;}}
```

Sufficient Conditions

1. we access two reference fields within method `next`
2. field `state` is the cursor of the data structure
3. field `rest` is part of the data structure
4. we track (i.e., transform to local variable) only `state`

Iterator Example

```
class Iter implements Iterator {
  List state; List aux;
  boolean hasNext() {
    return (this.state != null); }
  Object next() {
    this.state = this.state.rest;
    return obj;}}
```

```
class Test {
  static void m(Iter x, Aux y, Aux z)
    while (x.hasNext()) x.next();
}}
```

Sufficient Conditions

1. we access two reference fields within method `next`
2. field `state` is the cursor of the data structure
3. field `rest` is part of the data structure
4. we track (i.e., transform to local variable) only `state`

termination of `while` loop can be proven

Polyvariant Transformation

```
static void m(Ref x, Ref y) {
  x.f++; y.f--; }
static void m1(Ref x) {
  while (x.f>0) m(x,x); }
static void m2(Ref x) {
  y = new Ref();
  while (x.f>0) m(x,y); }
```

1. considering *f* local in *m*2 is essential for proving the termination
2. however, making *f* local in all contexts is not sound,
3.

Polyvariant Transformation

```
static void m(Ref x, Ref y) {
  x.f++; y.f--; }
static void m1(Ref x) {
  while (x.f>0) m(x,x); }
static void m2(Ref x) {
  y = new Ref();
  while (x.f>0) m(x,y); }
```

1. considering f local in $m2$ is essential for proving the termination
2. however, making f local in all contexts is not sound,
3. in order to take full advantage of context-sensitivity, we do a *polyvariant* transformation which generates two versions for m

Polyvariant Transformation

```
static void m(Ref x, Ref y) {
  x.f++; y.f--; }
static void m1(Ref x) {
  while (x.f>0) m(x,x); }
static void m2(Ref x) {
  y = new Ref();
  while (x.f>0) m(x,y); }
```

```
static int m$1(Ref x, Ref y) {
  x.f++; v ++;
  y.f--; v ++;   return v; }
static void m$2(Ref x, Ref y) {
  x.f++; y.f--; }
```

1. considering f local in $m2$ is essential for proving the termination
2. however, making f local in all contexts is not sound,
3. in order to take full advantage of context-sensitivity, we do a *polyvariant* transformation which generates two versions for m