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

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
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**
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 → Ruled-Based Representation

Abstract compilation
Input Output size relations
Transition relations

Yes

terminates

no

don’t know

Size Analysis

One rule per block in the CFG

Elvira Albert
From Termination to Cost
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;
    }
}
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: icall
3: isub
4: istore_1
5: iload_1
6: iflt 56
9: iload_1
10: icall
11: iadd
12: istore_2
13: iload_0
14: iload_1
15: iaload
16: istore_3
17: iload_2
18: iload_0
19: arraylength
20: if_icmpge 44
23: iload_0
24: iload_2
25: iastore
26: iload_3
27: if_icmpge 44
30: iload_0
31: iload_2
32: icall
33: isub
34: iload_0
35: iload_2
36: iastore
37: iastore
38: iinc 2, 1
41: goto 17
44: iload_0
45: iload_2
46: icall
47: isub
48: iload_3
49: iastore
50: iinc 1, -1
53: goto 5
56: return
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;
    }
}
Termination Analysis Components

Input: JBC program

Java Bytecode Program → Control Flow Graph → Ruled-Based Representation → Abstract compilation

- Input Output size relations
- loops
- Transition relations
- Size Analysis

Output

- terminates
- don’t know

Elvira Albert
From Termination to Cost
Control Flow Graph - Loops Extraction

entry →

aload_0
arraylength
iconst_2
isub
istore_1

iload_1
iflt
56: return

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

if_icmpge

iload_2
aload_0
arraylength

if_icmplt

aload_0
iload_2
iconst_1
isub
aload_0
iload_2
iaload
iastore
iinc 2,1

if_icmpge

iload_0
iload_2
iaload
iload_3

if_icmplt

Elvira Albert
From Termination to Cost
Control Flow Graph - Loops Extraction

entry →
| aload_0
| arraylength
| iconst_2
| isub
| istore_1

→ call loop-1 →
56: return

entry

entry

iiflt

| iload_1

| iload_2
| iconst_1
| isub
| iload_3
| iastore
| iinc 1,-1

→ call loop-2

exit

exit

entry

entry

if_icmpge

| iload_2
| arraylength

→

| iload_2
| iconst_1
| isub
| iload_0
| iload_2
| iload_1
| iaload
| iastore
| iinc 2,1

if_icmpge

if_icmpge

if_icmpge

if_icmplt

Elvira Albert

From Termination to Cost
Control Flow Graph - Loops Extraction

entry → aload_0
arraylength
iconst_2
isub
istore_1
→ call loop-1
→ 56: return

entry
→ iload_1
ifge
→ iload_1
iconst_1
iadd
istore_2
aload_0
iload_2
iaload
istore_3
→ call loop-2
→ exit

entry
→ iload_2
aload_0
arraylength
→ if_icmpge
→ if_icmplt
→ iload_2
aload_0
→ if_icmpge
→ if_icmplt
→ iload_2
iaload
→ if_icmpge
→ if_icmplt
→ iinc 2,1
→ exit

Elvira Albert
From Termination to Cost
Termination Analysis Components

Input: JBC program

- Java Bytecode Program
- Control Flow Graph
- Ruled-Based Representation

One rule per block in the CFG

abstract compilation

- Input: JBC program
- loops
- ranking functions?

Output

- Output size relations
- Transition relations

Elvira Albert

From Termination to Cost
for (int i=a.length-2;i≥0;i--) {
    int j=i+1;
    int v=a[i];
    ...... 
    a[j-1]=v;
}
for (int i=a.length-2; i>=0; i--) {
    int j=i+1;
    int v=a[i];
    ....
    a[j-1]=v;
}
for (int i=a.length-2;i≥0;i--) {
    int j=i+1;
    int v=a[i];
    ....
    a[j-1]=v;
}

\[
\begin{align*}
\mathcal{B}_1(\langle a, i \rangle, \langle \rangle) & := \text{iload}(i, s_0), \\
B_1^c(\langle a, i, s_0 \rangle, \langle \rangle) & := \text{guard}(s_0 < 0), \\
B_1^c(\langle a, i, s_0 \rangle, \langle \rangle) & := \text{guard}(s_0 \geq 0)
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}_2(\langle a, i \rangle, \langle \rangle) & := \text{iload}(i, s_0), \\
& \hspace{1cm} \text{iconst}(1, s_1), \\
& \hspace{1cm} \text{isub}(s_1, s_2, s_1), \\
& \hspace{1cm} \text{iload}(v, s_2), \\
& \hspace{1cm} \text{iastore}(s_0, s_1, s_2), \\
& \hspace{1cm} \text{iinc}(i, -1), \\
\mathcal{B}_1(\langle a, i \rangle, \langle \rangle) & := \mathcal{B}_3(\langle a, i, j, v \rangle, \langle \rangle).
\end{align*}
\]
Nice features of rule-based representation

rule-based program

set of procedures defined by one or more rules:

\[ p(\langle x \rangle, \langle y \rangle) := g, b_1, \ldots, b_n \]
Nice features of rule-based representation

rule-based program

set of procedures defined by one or more rules:
\[ p(\langle x \rangle, \langle y \rangle) := g, b_1, \ldots, b_n \]

- **Loops** are extracted in separate procedures:
  \[ B_3(\langle a, i, j, v \rangle, \langle \rangle) := C_1(\langle a, j, v \rangle, \langle j \rangle), B_4(\langle a, i, j, v \rangle, \langle \rangle). \]
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:
  \[ B_3(\langle a, i, j, v \rangle, \langle \rangle) := C_1(\langle a, j, v \rangle, \langle j \rangle), B_4(\langle a, i, j, v \rangle, \langle \rangle). \]

- 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:
\[ p(⟨\bar{x}, \bar{y}⟩) := g, b_1, \ldots, b_n \]

- **Loops** are extracted in separate procedures:
  \[ B_3(⟨a, i, j, v⟩, ⟨⟩) := C_1(⟨a, j, v⟩, ⟨j⟩), B_4(⟨a, i, j, v⟩, ⟨⟩). \]
- 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

\[ B_2(⟨a, i⟩, ⟨⟩) := iload(i, s_0), iconst(1, s_1), \ldots \]
rule-based program

set of procedures defined by one or more rules:
\[ p(⟨\bar{x}⟩, ⟨\bar{y}⟩) := g, b_1, \ldots, b_n \]

- **Loops** are extracted in separate procedures:
  \[ B_3(⟨a, i, j, v⟩, ⟨⟩) := C_1(⟨a, j, v⟩, ⟨j⟩), B_4(⟨a, i, j, v⟩, ⟨⟩). \]
- 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
  \[ B_2(⟨a, i⟩, ⟨⟩) := iload(i, s_0), iconst(1, s_1), \ldots \]
- **Guards** are the only form of conditional
  \[ B_1^c(⟨a, i, s_0⟩, ⟨⟩) := guard(s_0 \geq 0), B_2(⟨a, i⟩, ⟨⟩). \]
Nice features of rule-based representation

**rule-based program**

set of *procedures* defined by one or more rules:
\[ p(⟨\bar{x}, \bar{y}⟩) := g, b_1, \ldots, b_n \]

- **Loops** are extracted in separate procedures:
  \[ B_3(⟨a, i, j, v⟩, ⟨⟩) := C_1(⟨a, j, v⟩, ⟨j⟩), B_4(⟨a, i, j, v⟩, ⟨⟩). \]

- 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

  \[ B_2(⟨a, i⟩, ⟨⟩) := iload(i, s_0), iconst(1, s_1), \ldots \]

- **Guards** are the only form of conditional
  \[ B_1^c(⟨a, i, s_0⟩, ⟨⟩) := guard(s_0 \geq 0), B_2(⟨a, i⟩, ⟨⟩). \]

- Rules may have **multiple output** parameters
Termination Analysis Components

- **Input:** JBC program
- **Java Bytecode Program** → **Control Flow Graph** → **Ruled-Based Representation**
- **One rule per block in the CFG**
- **Abstract compilation**
- **Input Output size relations**
- **Transition relations**
- **Size Analysis**

- **Output:**
  - yes: terminates
  - no: don’t know
  - ranking functions?
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:
- \( \text{iadd}(a, b, c) \) is abstracted to \( c = b + a \)
- \( \text{guard}(\text{icmple}(s_0, s_1)) \) 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 \( s_0 \leq s_1 \)

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

\[
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) :=
\begin{align*}
&s_0 = a,
\end{align*}
\]
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 $s_0 \leq s_1$

\[
B_4(\langle a, i, j, v \rangle, \langle \rangle) :=
\begin{align*}
&aload(a, s_0), \\
&iload(j, s_1), \\
&iconst(1, s_2), \\
&isub(s_1, s_2, s_1), \\
&iload(v, s_2), \\
&iastore(s_0, s_1, s_2), \\
&iinc(i, -1), \\
&B_1(\langle a, i \rangle, \langle \rangle).
\end{align*}
\]

\[
B_4(\langle a, i, j, v \rangle, \langle \rangle) :=
\begin{align*}
&s_0 = a, \\
&s_1 = j,
\end{align*}
\]
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 $s_0 \leq s_1$

\[
B_4(\langle a, i, j, v \rangle, \langle \rangle) := \\
aload(a, s_0), \\
iload(j, s_1), \\
iconst(1, s_2), \\
isub(s_1, s_2, s_1), \\
iload(v, s_2), \\
ialstore(s_0, s_1, s_2), \\
iinc(i, -1), \\
B_1(\langle a, i \rangle, \langle \rangle).
\]

\[
B_4(\langle a, i, j, v \rangle, \langle \rangle) := \\
s_0 = a, \\
s_1 = j, \\
s_2 = 1,
\]
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 \( c = b + a \)
- \( \text{guard(icmple}(s_0, s_1)) \) is abstracted to \( s_0 \leq s_1 \)

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

\[
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) := \\
s_0 = a, \\
s_1 = j, \\
s_2 = 1, \\
s_1' = s_1 - s_2,
\]
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 $s_0 \leq s_1$

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) & := \\
& \text{aload}(a, s_0), \\
& \text{iload}(j, s_1), \\
& \text{iconst}(1, s_2), \\
& \text{isub}(s_1, s_2, s_1), \\
& \text{iload}(v, s_2), \\
& \text{iastore}(s_0, s_1, s_2), \\
& \text{iinc}(i, -1), \\
& \mathcal{B}_1(\langle a, i \rangle, \langle \rangle).
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) & := \\
& s_0 = a, \\
& s_1 = j, \\
& s_2 = 1, \\
& s_1' = s_1 - s_2, \\
& s_2' = v,
\end{align*}
\]
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 \( s_0 \leq s_1 \)

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) & := \\
& a\text{load}(a, s_0), \\
& i\text{load}(j, s_1), \\
& i\text{const}(1, s_2), \\
& i\text{sub}(s_1, s_2, s_1), \\
& i\text{load}(v, s_2), \\
& i\text{astore}(s_0, s_1, s_2), \\
& i\text{inc}(i, -1), \\
& \mathcal{B}_1(\langle a, i \rangle, \langle \rangle). \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) & := \\
& s_0 = a, \\
& s_1 = j, \\
& s_2 = 1, \\
& s'_1 = s_1 - s_2, \\
& s'_2 = v, \\
& true,
\end{align*}
\]
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 $s_0 \leq s_1$

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) & := \\
& \text{aload}(a, s_0), \\
& \text{iload}(j, s_1), \\
& \text{iconst}(1, s_2), \\
& \text{isub}(s_1, s_2, s_1), \\
& \text{iload}(v, s_2), \\
& \text{iastore}(s_0, s_1, s_2), \\
& \text{iinc}(i, -1), \\
& \mathcal{B}_1(\langle a, i \rangle, \langle \rangle).
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) & := \\
& s_0 = a, \\
& s_1 = j, \\
& s_2 = 1, \\
& s_1' = s_1 - s_2, \\
& s_2' = v, \\
& true, \\
& i' = i - 1,
\end{align*}
\]
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:
- \( \text{iadd}(a, b, c) \) is abstracted to \( c = b + a \)
- \( \text{guard}(\text{icmple}(s_0, s_1)) \) is abstracted to \( s_0 \leq s_1 \)

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) := & \\
\text{aload}(a, s_0), & \\
\text{iload}(j, s_1), & \\
\text{iconst}(1, s_2), & \\
\text{isub}(s_1, s_2, s_1), & \\
\text{iload}(v, s_2), & \\
\text{iastore}(s_0, s_1, s_2), & \\
\text{iinc}(i, -1), & \\
\mathcal{B}_1(\langle a, i \rangle, \langle \rangle). & \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) := & \\
\text{aload}(a, s_0), & \\
\text{iload}(j, s_1), & \\
\text{iconst}(1, s_2), & \\
\text{isub}(s_1, s_2, s_1), & \\
\text{iload}(v, s_2), & \\
\text{iastore}(s_0, s_1, s_2), & \\
\text{iinc}(i, -1), & \\
\text{true}, & \\
i' = i - 1, & \\
\mathcal{B}_1(\langle a, i' \rangle, \langle \rangle). & \\
\end{align*}
\]
We can apply existing techniques to reason on the termination of the abstract program (CLP, TRS, ...)

We can easily find ranking functions:

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

\[ B_1(\langle a, i \rangle) \rightarrow B_1(\langle a', i' \rangle) \quad \{ a' = a, i \geq 0, i' = i - 1 \} \]
\[ C_1(\langle a, j, v \rangle) \rightarrow C_1(\langle a', j', v' \rangle) \quad \{ a' = a, j' = j + 1, j < a \} \]
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

\[ B_1(\langle a, i \rangle) \rightarrow B_1(\langle a', i' \rangle) \quad \{ a' = a, i \geq 0, i' = i - 1 \} \]

\[ C_1(\langle a, j, v \rangle) \rightarrow C_1(\langle a', j', v' \rangle) \quad \{ a' = a, j' = j + 1, j < a \} \]

We can easily find ranking functions:

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

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

\[ B_1(\langle a, i \rangle) \rightarrow B_1(\langle a', i' \rangle) \quad \{ a' = a, i \geq 0, i' = i - 1 \} \]

\[ C_1(\langle a, j, v \rangle) \rightarrow C_1(\langle a', j', v' \rangle) \quad \{ a' = a, j' = j + 1, j < a \} \]

We can easily find ranking functions:

```java
//@decreasing f_{B_1}(a, i) = i
for (int i=a.length-2; i\geq 0; i--)
{
    ...

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

Elvira Albert
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

\[
\begin{align*}
B_1(\langle a, i \rangle) & \rightarrow B_1(\langle a', i' \rangle) \quad \{ a' = a, i \geq 0, i' = i - 1 \} \\
C_1(\langle a, j, v \rangle) & \rightarrow C_1(\langle a', j', v' \rangle) \quad \{ a' = a, j' = j + 1, j < a \}
\end{align*}
\]

We can easily find ranking functions:

```java
//@decreasing f_{B_1}(a, i) = i
for (int i=a.length-2; i\geq0; i--) {
    ...
    //@decreasing f_{C_1}(a, j, v) = a - j
    while ( j<a.length && a[j]<v) {
        ...
        j++;
    }
}
```
Proving Termination - Cont.

Loops

\[ B_1(\langle a, i \rangle) \to B_1(\langle a', i' \rangle) \quad \{a' = a, i >= 0, i' = i - 1\} \]
\[ C_1(\langle a, j, v \rangle) \to C_1(\langle a', j', v' \rangle) \quad \{a' = a, j' = j + 1, j < a\} \]
Theorem (Soundness)

Let $P$ be a JBC program and $C_A$ the transition relations computed from $P$. If there exists a non-terminating trace in $P$ then there exists a non-terminating derivation in $C_A$.

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.
simple imperative programs automatically transformed into rule-based representation [ESOP’07]
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]
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]
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
static cost analysis

bound the cost of executing program $P$ on any input data $\bar{x}$ without having to actually run $P(\bar{x})$
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
**cost model**: specify the resource of interest
- number of executed instructions
- memory consumption
- number of calls to method
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.

\[
\begin{align*}
C_1(a, j, v) &= 3 & \{j \geq a\} \\
C_1(a, j, v) &= 8 & \{j < a\} \\
C_1(a, j, v) &= 17 + C_1(a, j', v) & \{j < a, j' = j + 1\}
\end{align*}
\]
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.

\[
C_1(a, j, v) = \begin{cases} 
3 & \{j \geq a\} \\
8 & \{j < a\} \\
17 + C_1(a, j', v) & \{j < a, j' = j + 1\}
\end{cases}
\]

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

\[
C_1(a, j, v) = 8 + 17 \times \text{nat}(a - j)
\]
from cost to termination

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

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

d onto termination to cost

techniques used in termination analysis are very useful in cost analysis
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

\[
B_4(\langle a, i, j, v \rangle, \langle \rangle) := \\
\text{aload}(a, s_0), \\
\text{iload}(j, s_1), \\
\text{iconst}(1, s_2), \\
\text{isub}(s_1, s_2, s_1), \\
\text{iload}(v, s_2), \\
\text{iastore}(s_0, s_1, s_2), \\
\text{iinc}(i, -1), \\
B_1(\langle a, i \rangle, \langle \rangle).
\]

\[
B_4(\langle a, i, j, v \rangle, \langle \rangle) := \\
s_0 = a, \\
s_1 = j, \\
s_2 = 1, \\
s'_1 = s_1 - s_2, \\
s'_2 = v, \\
\text{true}, \\
i' = i - 1, \\
B_1(\langle a, i' \rangle, \langle \rangle).
\]

\[
B_4(a, i, j, v) =
\]
The abstract compilation obtained by the termination module is used to generate the recurrence relations

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

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

\begin{align*}
\mathcal{B}_4(a, i, j, v) & = \\
& 1+
\end{align*}
Phase 1: Generation of recurrence equations

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

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ \text{aload}(a, s_0), \]
\[ iload(j, s_1), \]
\[ iconst(1, s_2), \]
\[ isub(s_1, s_2, s_1), \]
\[ iload(v, s_2), \]
\[ iastore(s_0, s_1, s_2), \]
\[ iinc(i, -1), \]
\[ B_1(\langle a, i \rangle, \langle \rangle). \]

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ s_0 = a, \]
\[ s_1 = j, \]
\[ s_2 = 1, \]
\[ s'_1 = s_1 - s_2, \]
\[ s'_2 = v, \]
\[ true, \]
\[ i' = i - 1, \]
\[ B_1(\langle a, i' \rangle, \langle \rangle). \]

\[ B_4(a, i, j, v) = 1+ \]
\[ 1+ \]
Phase 1: Generation of recurrence equations

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

\[ B_4(⟨a, i, j, v⟩, ⟨⟩) := \]
\[aload(a, s_0), \]
\[iloads_1, \]
\[iastype(1, s_2), \]
\[isub(s_1, s_2, s_1), \]
\[iloads_2, \]
\[iastype(s_0, s_1, s_2), \]
\[iinc(i, −1), \]
\[B_1(⟨a, i⟩, ⟨⟩). \]

\[ B_4(⟨a, i, j, v⟩, ⟨⟩) := \]
\[s_0 = a, \]
\[s_1 = j, \]
\[s_2 = 1, \]
\[s'_1 = s_1 − s_2, \]
\[s'_2 = v, \]
\[true, \]
\[i' = i − 1, \]
\[B_1(⟨a, i'⟩, ⟨⟩). \]

\[ B_4(a, i, j, v) = \]
\[1+ \]
\[1+ \]
\[1+ \]
### Phase 1: Generation of recurrence equations

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

<table>
<thead>
<tr>
<th>$B_4(\langle a, i, j, v \rangle, \langle \rangle)$ :=</th>
<th>$B_4(\langle a, i, j, v \rangle, \langle \rangle)$ :=</th>
<th>$B_4(a, i, j, v)$ =</th>
</tr>
</thead>
<tbody>
<tr>
<td>$aload(a, s_0)$,</td>
<td>$s_0 = a$,</td>
<td>$1+$</td>
</tr>
<tr>
<td>$iload(j, s_1)$,</td>
<td>$s_1 = j$,</td>
<td>$1+$</td>
</tr>
<tr>
<td>$iconst(1, s_2)$,</td>
<td>$s_2 = 1$,</td>
<td>$1+$</td>
</tr>
<tr>
<td>$isub(s_1, s_2, s_1)$,</td>
<td>$s'_1 = s_1 - s_2$,</td>
<td>$1+$</td>
</tr>
<tr>
<td>$iload(v, s_2)$,</td>
<td>$s'_2 = v$,</td>
<td></td>
</tr>
<tr>
<td>$iastore(s_0, s_1, s_2)$,</td>
<td>$true$,</td>
<td></td>
</tr>
<tr>
<td>$iinc(i, -1)$,</td>
<td>$i' = i - 1$,</td>
<td></td>
</tr>
<tr>
<td>$B_1(\langle a, i \rangle, \langle \rangle)$.</td>
<td>$B_1(\langle a, i' \rangle, \langle \rangle)$.</td>
<td></td>
</tr>
</tbody>
</table>
Phase 1: Generation of recurrence equations

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

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ \text{aload}(a, s_0), \]
\[ \text{iload}(j, s_1), \]
\[ \text{iconst}(1, s_2), \]
\[ \text{isub}(s_1, s_2, s_1), \]
\[ \text{iload}(v, s_2), \]
\[ \text{iastore}(s_0, s_1, s_2), \]
\[ \text{iinc}(i, -1), \]
\[ B_1(\langle a, i \rangle, \langle \rangle). \]

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ s_0 = a, \]
\[ s_1 = j, \]
\[ s_2 = 1, \]
\[ s'_1 = s_1 - s_2, \]
\[ s'_2 = v, \]
\[ \text{true}, \]
\[ i' = i - 1, \]
\[ B_1(\langle a, i' \rangle, \langle \rangle). \]

\[ B_4(a, i, j, v) = \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
Phase 1: Generation of recurrence equations

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

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ \text{aload}(a, s_0), \]
\[ \text{iload}(j, s_1), \]
\[ \text{iconst}(1, s_2), \]
\[ \text{isub}(s_1, s_2, s_1), \]
\[ \text{iload}(v, s_2), \]
\[ \text{iastore}(s_0, s_1, s_2), \]
\[ \text{iinc}(i, -1), \]
\[ B_1(\langle a, i \rangle, \langle \rangle). \]

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ s_0 = a, \]
\[ s_1 = j, \]
\[ s_2 = 1, \]
\[ s_1' = s_1 - s_2, \]
\[ s_2' = v, \]
\[ \text{true}, \]
\[ i' = i - 1, \]
\[ B_1(\langle a, i' \rangle, \langle \rangle). \]

\[ B_4(a, i, j, v) = \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
Phase 1: Generation of recurrence equations

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

\[ B_4(⟨a, i, j, v⟩, ⟨⟩) := \]
\[ \text{aload}(a, s_0), \]
\[ \text{iload}(j, s_1), \]
\[ \text{iconst}(1, s_2), \]
\[ \text{isub}(s_1, s_2, s_1), \]
\[ \text{iload}(v, s_2), \]
\[ \text{iastore}(s_0, s_1, s_2), \]
\[ \text{iinc}(i, −1), \]
\[ B_1(⟨a, i⟩, ⟨⟩). \]

\[ B_4(⟨a, i, j, v⟩, ⟨⟩) := \]
\[ s_0 = a, \]
\[ s_1 = j, \]
\[ s_2 = 1, \]
\[ s'_1 = s_1 − s_2, \]
\[ s'_2 = v, \]
\[ true, \]
\[ i' = i − 1, \]
\[ B_1(⟨a, i'⟩, ⟨⟩). \]

\[ B_4(a, i, j, v) = \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]

Phase 1: Generation of recurrence equations

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

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
- \text{aload}(a, s_0),
- \text{iload}(j, s_1),
- \text{iconst}(1, s_2),
- \text{isub}(s_1, s_2, s_1),
- \text{iload}(v, s_2),
- \text{iastore}(s_0, s_1, s_2),
- \text{iinc}(i, -1),
- \text{B}_1(\langle a, i \rangle, \langle \rangle). \]

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
- \( s_0 = a, \)
- \( s_1 = j, \)
- \( s_2 = 1, \)
- \( s'_1 = s_1 - s_2, \)
- \( s'_2 = v, \)
- \text{true,}
- \( i' = i - 1, \)
- \text{B}_1(\langle a, i' \rangle, \langle \rangle). \]

\[ B_4(a, i, j, v) = \]
- \( 1+ \)
- \( 1+ \)
- \( 1+ \)
- \( 1+ \)
- \( 1+ \)
- \( 1+ \)
- \text{B}_1(a, i'), \)
Phase 1: Generation of recurrence equations

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

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ \text{aload}(a, s_0), \]
\[ iload(j, s_1), \]
\[ iconst(1, s_2), \]
\[ isub(s_1, s_2, s_1), \]
\[ iload(v, s_2), \]
\[ iastore(s_0, s_1, s_2), \]
\[ iinc(i, -1), \]
\[ B_1(\langle a, i \rangle, \langle \rangle). \]

\[ B_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ s_0 = a, \]
\[ s_1 = j, \]
\[ s_2 = 1, \]
\[ s_1' = s_1 - s_2, \]
\[ s_2' = v, \]
\[ true, \]
\[ i' = i - 1, \]
\[ B_1(\langle a, i' \rangle, \langle \rangle). \]

\[ B_4(a, i, j, v) = \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ 1+ \]
\[ B_1(a, i'), \]
\[ \{i' = i - 1\} \]
Phase 1: Generation of recurrence equations

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

\[ \mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ \begin{align*}
    & \text{aload}(a, s_0), \\
    & \text{iload}(j, s_1), \\
    & \text{iconst}(1, s_2), \\
    & \text{isub}(s_1, s_2, s_1), \\
    & \text{iload}(v, s_2), \\
    & \text{iastore}(s_0, s_1, s_2), \\
    & \text{iinc}(i, -1), \\
    & \mathcal{B}_1(\langle a, i \rangle, \langle \rangle).
\end{align*} \]

\[ \mathcal{B}_4(\langle a, i, j, v \rangle, \langle \rangle) := \]
\[ \begin{align*}
    & s_0 = a, \\
    & s_1 = j, \\
    & s_2 = 1, \\
    & s_1' = s_1 - s_2, \\
    & s_2' = v, \\
    & \text{true}, \\
    & i' = i - 1, \\
    & \mathcal{B}_1(\langle a, i' \rangle, \langle \rangle).
\end{align*} \]

\[ \mathcal{B}_4(a, i, j, v) = \]
\[ \begin{align*}
    & 1+ \\
    & 1+ \\
    & 1+ \\
    & 1+ \\
    & 1+ \\
    & 1+ \\
    & 1+ \\
    & \mathcal{B}_1(a, i'), \\
    & \{i' = i - 1\}.
\end{align*} \]

cost equation systems

Given a rule \( p(\langle \bar{x} \rangle, \langle \bar{y} \rangle) := g, b_1, \ldots, b_n \) and \( \varphi_r \) its corresponding size relations. The cost equation is: \( p(\bar{x}) = \sum_{i=1}^{n} M(b_i), \varphi_r \)
The result of generating the cost equations for all program rules is:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{ i = a - 2 \} & \quad \{ i < 0 \} & \quad \{ i \geq 0, i' = i - 1, j = i + 1 \} \\
\{ j \geq a \} & \quad \{ j < a \} & \quad \{ j < a, j' = j + 1 \}
\end{align*}
\]

```java
static void sort(int a[]) {
    for (int i=a.length-2; i\geq0; 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:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{ i = a - 2 \} \\
\{ i < 0 \} \\
\{ i \geq 0, i' = i - 1, j = i + 1 \} \\
\{ j \geq a \} \\
\{ j < a \} \\
\{ j < a, j' = j + 1 \}
\end{align*}
\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq 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:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{ i = a - 2 \} \\
\{ i < 0 \} \\
\{ i \geq 0, i' = i - 1, j = i + 1 \} \\
\{ j \geq a \} \\
\{ j < a \} \\
\{ j < a, j' = j + 1 \}
\end{align*}
\]

```java
static void sort(int a[]) {
    for (int i=a.length-2; i\geq0; 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:

\[
\begin{align*}
sort(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{ i = a - 2 \} \\
\{ i < 0 \} \\
\{ i \geq 0, i' = i - 1, j = i + 1 \} \\
\{ j \geq a \} \\
\{ j < a \} \\
\{ j < a, j' = j + 1 \}
\end{align*}
\]

static void sort(int a[])
{
    for (int i=a.length-2; i\geq 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:

\[
\text{sort}(a) = 6 + B_1(a, i) \\
B_1(a, i) = 2 \\
B_1(a, i) = 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) = 3 \\
C_1(a, j, v) = 8 \\
C_1(a, j, v) = 17 + C_1(a, j', v)
\]

\[
\{i = a - 2\} \\
\{i < 0\} \\
\{i \geq 0, i' = i - 1, j = i + 1\} \\
\{j \geq a\} \\
\{j < a\} \\
\{j < a, j' = j + 1\}
\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq 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:

\[\text{sort}(a) = 6 + B_1(a, i)\]
\[B_1(a, i) = 2\]
\[B_1(a, i) = 18 + C_1(a, j, v) + B_1(a, i')\]
\[C_1(a, j, v) = 3\]
\[C_1(a, j, v) = 8\]
\[C_1(a, j, v) = 17 + C_1(a, j', v)\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq 0; i--) {
        ...
        while ( j\lt a.length & & a[j]\lt v) {
            ...
            j++
        }
    ...
}
Phase 1: Generation of recurrence equations

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

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{i &= a - 2\} \\
\{i < 0\} \\
\{i \geq 0, i' = i - 1, j = i + 1\} \\
\{j \geq a\} \\
\{j < a\} \\
\{j < a, j' = j + 1\}
\end{align*}
\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq0; 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:

\[ \text{sort}(a) = 6 + B_1(a, i) \]
\[ B_1(a, i) = \begin{cases} 2 & \text{if } i < 0 \\ 18 + C_1(a, j, v) + B_1(a, i') & \text{if } i \geq 0, i' = i - 1, j = i + 1 \end{cases} \]
\[ C_1(a, j, v) = \begin{cases} 3 & \text{if } j \geq a \\ 8 & \text{if } j < a \end{cases} \]
\[ C_1(a, j, v) = 17 + C_1(a, j', v) \]

\[ \text{static void sort(int a[]) { for (int i=a.length-2; i\geq0; i--) { \ldots while ( j<a.length && a[j]<v) { \ldots j++ \} \ldots } } } \]

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:

\[ \text{sort}(a) = 6 + B_1(a, i) \]
\[ B_1(a, i) = 2 \]
\[ B_1(a, i) = 18 + C_1(a, j, v) + B_1(a, i') \]
\[ C_1(a, j, v) = 3 \]
\[ C_1(a, j, v) = 8 \]
\[ C_1(a, j, v) = 17 + C_1(a, j', v) \]

\[ \{ i = a - 2 \} \]
\[ \{ i < 0 \} \]
\[ \{ i \geq 0, i' = i - 1, j = i + 1 \} \]
\[ \{ j \geq a \} \]
\[ \{ j < a \} \]
\[ \{ j < a, j' = j + 1 \} \]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq0; 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:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{i = a - 2\} & \quad \{i < 0\} & \quad \{i \geq 0, i' = i - 1, j = i + 1\} \\
\{j \geq a\} & \quad \{j < a\} & \quad \{j < a, j' = j + 1\}
\end{align*}
\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq0; 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:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\{i = a - 2\} \\
\{i < 0\} \\
\{i \geq 0, i' = i - 1, j = i + 1\} \\
\{j \geq a\} \\
\{j < a\} \\
\{j < a, j' = j + 1\}

static void sort(int a[]) {
    for (int i = a.length-2; i \geq 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:

$$
\text{sort}(a) = 6 + B_1(a, i) \\
B_1(a, i) = 2 \\
B_1(a, i) = 18 + C_1(a,j,v) + B_1(a,i') \quad \{i \geq 0, i' = i - 1, j = i + 1\} \\
C_1(a,j,v) = 3 \quad \{j \geq a\} \\
C_1(a,j,v) = 8 \quad \{j < a\} \\
C_1(a,j,v) = 17 + C_1(a,j',v) \quad \{j < a, j' = j + 1\}
$$

static void sort(int a[]) {
    for (int i=a.length-2; i\geq 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:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{ i = a - 2 \} \\
\{ i < 0 \} \\
\{ i \geq 0, i' = i - 1, j = i + 1 \} \\
\{ j \geq a \} \\
\{ j < a \} \\
\{ j < a, j' = j + 1 \}
\end{align*}
\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq0; 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:

\[ \text{sort}(a) = 6 + B_1(a, i) \]
\[ B_1(a, i) = 2 \]
\[ B_1(a, i) = 18 + C_1(a, j, v) + B_1(a, i') \]
\[ C_1(a, j, v) = 3 \]
\[ C_1(a, j, v) = 8 \]
\[ C_1(a, j, v) = 17 + C_1(a, j', v) \]

\{ i = a - 2 \}
\{ i < 0 \}
\{ i \geq 0, i' = i - 1, j = i + 1 \}
\{ j \geq a \}
\{ j < a \}
\{ j < a, j' = j + 1 \}

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:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[
\begin{align*}
\{ i = a - 2 \} \\
\{ i < 0 \} \\
\{ i \geq 0, i' = i - 1, j = i + 1 \} \\
\{ j \geq a \} \\
\{ j < a \} \\
\{ j < a, j' = j + 1 \}
\end{align*}
\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq 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:

\[ \text{sort}(a) = 6 + B_1(a, i) \]
\[ B_1(a, i) = 2 \]
\[ B_1(a, i) = 18 + C_1(a, j, v) + B_1(a, i') \]
\[ C_1(a, j, v) = 3 \]
\[ C_1(a, j, v) = 8 \]
\[ C_1(a, j, v) = 17 + C_1(a, j', v) \]

\[ \{ i = a - 2 \} \]
\[ \{ i < 0 \} \]
\[ \{ i \geq 0, i' = i - 1, j = i + 1 \} \]
\[ \{ j \geq a \} \]
\[ \{ j < a \} \]
\[ \{ j < a, j' = j + 1 \} \]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq 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:

\[
\begin{align*}
\text{sort}(a) &= 6 + B_1(a, i) \\
B_1(a, i) &= 2 \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \\
C_1(a, j, v) &= 3 \\
C_1(a, j, v) &= 8 \\
C_1(a, j, v) &= 17 + C_1(a, j', v)
\end{align*}
\]

\[\begin{align*}
\{i = a - 2\} \\
\{i < 0\} \\
\{i \geq 0, i' = i - 1, j = i + 1\} \\
\{j \geq a\} \\
\{j < a\} \\
\{j < a, j' = j + 1\}
\end{align*}\]

static void sort(int a[]) {
    for (int i=a.length-2; i\geq0; 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:

\[ \text{sort}(a) = 6 + \mathcal{B}_1(a, i) \]
\[ \mathcal{B}_1(a, i) = 2 \]
\[ \mathcal{B}_1(a, i) = 18 + \mathcal{C}_1(a, j, v) + \mathcal{B}_1(a, i') \]
\[ \mathcal{C}_1(a, j, v) = 3 \]
\[ \mathcal{C}_1(a, j, v) = 8 \]
\[ \mathcal{C}_1(a, j, v) = 17 + \mathcal{C}_1(a, j', v) \]

\[ \{ i = a - 2 \} \]
\[ \{ i < 0 \} \]
\[ \{ i \geq 0, i' = i - 1, j = i + 1 \} \]
\[ \{ j \geq a \} \]
\[ \{ j < a \} \]
\[ \{ j < a, j' = j + 1 \} \]

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

Partial Evaluation: equations are converted into directly recursive form by applying the well-known technique of partial evaluation.
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
- Inexact size relations
- Non-deterministic

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

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
    \[ C_1(a, j, v) = \begin{cases} 3 & \text{if } j \geq a \\ \emptyset & \text{otherwise} \end{cases} \]
  - Inexact size relations
    \[ C_1(a, j, v) = \begin{cases} 3 & \text{if } j < a \\ \emptyset & \text{otherwise} \end{cases} \]
  - Non-deterministic
    \[ C_1(a, j, v) = 17 + C_1(a, j', v) \]
      \[ \begin{cases} j < a & j' = j + 1 \end{cases} \]
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: $C_1(a, j, v) = 3 \quad \{j \geq a\}$
  - Inexact size relations: $C_1(a, j, v) = 8 \quad \{j < a\}$
  - Non-deterministic: $C_1(a, j, v) = 17 + C_1(a, j', v)\quad \{j < a, j' = j + 1\}$

- A precise solution often does not exist:
Solving the cost 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
    \[ C_1(a, j, v) = \begin{cases} 3 & \{j \geq a\} \\ \end{cases} \]
  - Inexact size relations
    \[ C_1(a, j, v) = \begin{cases} 8 & \{j < a\} \\ \end{cases} \]
  - Non-deterministic
    \[ C_1(a, j, v) = 17 + C_1(a, j', v) \quad \{j < a, j' = j + 1\} \]

- 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:

$$UB = \# \text{ iter} \times \text{max}_\text{cost} + \text{max}_\text{base}$$

- $\# \text{ iter}$: bound the number of iterations in loop (or chain of recursive calls in the relation)
- A ranking function $f(\bar{v})$ for $C$ guarantees that the length of any chain of recursive calls to $C$ cannot exceed $f(\bar{v})$
- $\text{max}_\text{cost}/\text{max}_\text{base}$: bound the maximal cost of expression upper bound

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

- One recursive call: $f(\bar{x}) \times \text{max}_\text{cost}(\text{expr})$
- $n$ recursive calls: $n \times f(\bar{x}) \times \text{max}_\text{cost}(\text{expr})$
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 $\text{UB} = \# \text{ iter} \times \text{max\_cost} + \text{max\_base}$:

- $\# \text{ iter}$: 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:

$$UB = \# \text{ iter} \times \max \text{ cost} + \max \text{ base}$$

- $\# \text{ iter}$: 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:

$$\text{UB} = \# \text{ iter} \times \text{max\_cost} + \text{max\_base}$$

- **$\# \text{ iter}$**: 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 $f(\bar{v})$
- **$\text{max\_cost}/\text{max\_base}$**: bound the maximal cost of expression
**Phase 2: Solving the cost equations (upper bounds)**

<table>
<thead>
<tr>
<th>Basic Idea for Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given a loop (or cost relation $C$), its upper bound can be computed as: $UB = #\ iter \times max_cost + max_base$</td>
</tr>
<tr>
<td><strong># iter</strong>: bound the number of iterations in loop (or chain of recursive calls in the relation)</td>
</tr>
<tr>
<td>a ranking function $f(\bar{x})$ for $C$ guarantees that the length of any chain of recursive calls to $C$ cannot exceed $f(\bar{v})$</td>
</tr>
<tr>
<td><strong>max_cost/max_base</strong>: bound the maximal cost of expression</td>
</tr>
</tbody>
</table>

**Upper Bound**

We then look at the shape of equations:

$C(\bar{x}) = expr + C(\bar{y}) + \ldots + C(\bar{w})$: 

---

Elvira Albert  
From Termination to 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

$$UB = \# \text{ iter} \cdot \text{max\_cost} + \text{max\_base}$$

- $\# \text{ iter}$: 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 $f(\bar{v})$
- $\text{max\_cost}/\text{max\_base}$: 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: $f(\bar{x}) \cdot \text{max\_cost}(expr)$
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

$$\text{UB} = \# \text{ iter} \times \text{max\_cost} + \text{max\_base}$$

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

**upper bound**

We then look at the shape of equations

$$C(\vec{x}) = expr + C(\vec{y}) + \ldots + C(\vec{w})$$

- One recursive call: $f(\vec{x}) \times \text{max\_cost}(expr)$
- $n$ recursive calls: $n^{f(\vec{x})} \times \text{max\_cost}(expr)$
Phase 2: Solving the cost equations (example)

\[
UB = \# \text{ iter} \times \text{max cost} + \text{max base}
\]

\[
\begin{align*}
C_1(a, j, v) &= 3 \quad \{j \geq a\} \\
C_1(a, j, v) &= 8 \quad \{j < a\} \\
C_1(a, j, v) &= 17 + C_1(a, j', v) \quad \{j < a, j' = j + 1\}
\end{align*}
\]
Phase 2: Solving the cost equations (example)

\[ UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base} \]

\[
\begin{align*}
C_1(a, j, v) &= 3 & \{ j \geq a \} \\
C_1(a, j, v) &= 8 & \{ j < a \} \\
C_1(a, j, v) &= 17 + C_1(a, j', v) & \{ j < a, j' = j + 1 \} \\
C_1(a, j, v) &= \text{nat}(a - j)
\end{align*}
\]
Phase 2: Solving the cost equations (example)

\[ UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base} \]

\[
C_1(a, j, v) = \begin{cases} 
3 & \{ j \geq a \} \\
8 & \{ j < a \} \\
17 + C_1(a, j', v) & \{ j < a, j' = j + 1 \}
\end{cases}
\]

\[ C_1(a, j, v) = \text{nat}(a - j) \times 17 \]
**UB** = \# iter*max\_cost+max\_base

\[ C_1(a, j, v) = \begin{cases} 3 & \{ j \geq a \} \\ 8 & \{ j < a \} \\ 17 + C_1(a, j', v) & \{ j < a, j' = j + 1 \} \end{cases} \]

\[ C_1(a, j, v) = nat(a - j) * 17 + 8 \]
Phase 2: Solving the cost equations (example)

\[ UB = \# \text{ iter} \cdot \text{max \_ cost} + \text{max \_ base} \]

\[ C_1(a, j, v) = 3 \quad \{ j \geq a \} \]
\[ C_1(a, j, v) = 8 \quad \{ j < a \} \]
\[ C_1(a, j, v) = 17 + C_1(a, j', v) \quad \{ j < a, j' = j + 1 \} \]
\[ C_1(a, j, v) = \text{nat}(a - j) \cdot 17 + 8 \]

\[ B_1(a, i) = 2 \quad \{ i < 0 \} \]
\[ B_1(a, i) = 18 + C_1(a, j, v) + B_1(a, i') \quad \{ i \geq 0, i' = i - 1, j = i + 1 \} \]
Phase 2: Solving the cost equations (example)

\[ UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base} \]

\[ C_1(a, j, v) = \begin{cases} 3 & \{ j \geq a \} \\ 8 & \{ j < a \} \end{cases} \]
\[ C_1(a, j, v) = 17 + C_1(a, j', v) & \{ j < a, j' = j + 1 \} \]
\[ C_1(a, j, v) = \text{nat}(a - j) \times 17 + 8 \]

\[ B_1(a, i) = \begin{cases} 2 & \{ i < 0 \} \\ 18 + C_1(a, j, v) + B_1(a, i') & \{ i \geq 0, i' = i - 1, j = i + 1 \} \end{cases} \]
\[ B_1(a, i) = \text{nat}(i + 1) \]
Phase 2: Solving the cost equations (example)

$$UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base}$$

\[ C_1(a, j, v) = \begin{cases} 3 & \{j \geq a\} \\ 8 & \{j < a\} \\ 17 + C_1(a, j', v) & \{j < a, j' = j + 1\} \end{cases} \]

\[ C_1(a, j, v) = \text{nat}(a - j) \times 17 + 8 \]

\[ B_1(a, i) = \begin{cases} 2 & \{i < 0\} \\ 18 + C_1(a, j, v) + B_1(a, i') & \{i \geq 0, i' = i - 1, j = i + 1\} \end{cases} \]

\[ B_1(a, i) = \text{nat}(i + 1) \times \text{max\_cost}[18 + C_1(a, j, v)] \]
Phase 2: Solving the cost equations (example)

\[
\begin{align*}
\text{UB} &= \# \text{ iter} \times \text{max\_cost} + \text{max\_base} \\
C_1(a, j, v) &= 3 \quad \{j \geq a\} \\
C_1(a, j, v) &= 8 \quad \{j < a\} \\
C_1(a, j, v) &= 17 + C_1(a, j', v) \quad \{j < a, j' = j + 1\} \\
\quad C_1(a, j, v) &= \text{nat}(a - j) \times 17 + 8 \\
B_1(a, i) &= 2 \quad \{i < 0\} \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') \quad \{i \geq 0, i' = i - 1, j = i + 1\} \\
\quad B_1(a, i) &= \text{nat}(i + 1) \times \text{max\_cost}[18 + \text{nat}(a - j) \times 17 + 8]
\end{align*}
\]
Phase 2: Solving the cost equations (example)

\[ UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base} \]

\[
C_1(a, j, v) = \begin{cases} 3 & \{j \geq a\} \\ 8 & \{j < a\} \end{cases}
\]

\[
C_1(a, j, v) = 17 + C_1(a, j', v) & \{j < a, j' = j + 1\} \\
C_1(a, j, v) = \text{nat}(a - j) \times 17 + 8
\]

\[
B_1(a, i) = \begin{cases} 2 & \{i < 0\} \\ 18 + C_1(a, j, v) + B_1(a, i') & \{i \geq 0, i' = i - 1, j = i + 1\} \end{cases}
\]

\[
B_1(a, i) = \text{nat}(i + 1) \times (18 + \text{max\_cost} \left[ \text{nat}(a - j) \right] \times 17 + 8)
\]
Phase 2: Solving the cost equations (example)

$$\text{UB} = \# \text{ iter} \times \text{max_cost} + \text{max_base}$$

\[
\begin{align*}
C_1(a, j, v) & = 3 & \{j \geq a\} \\
C_1(a, j, v) & = 8 & \{j < a\} \\
C_1(a, j, v) & = 17 + C_1(a, j', v) & \{j < a, j' = j + 1\} \\
C_1(a, j, v) & = \text{nat}(a - j) \times 17 + 8
\end{align*}
\]

\[
\begin{align*}
B_1(a, i) & = 2 & \{i < 0\} \\
B_1(a, i) & = 18 + C_1(a, j, v) + B_1(a, i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
B_1(a, i) & = \text{nat}(i + 1) \times (18 + \text{nat}(a - 1) \times 17 + 8)
\end{align*}
\]
Phase 2: Solving the cost equations (example)

$$UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base}$$

$$C_1(a, j, v) = 3 \quad \{j \geq a\}$$
$$C_1(a, j, v) = 8 \quad \{j < a\}$$
$$C_1(a, j, v) = 17 + C_1(a, j', v) \quad \{j < a, j' = j + 1\}$$

$$C_1(a, j, v) = \text{nat}(a - j) \times 17 + 8$$

$$B_1(a, i) = 2 \quad \{i < 0\}$$
$$B_1(a, i) = 18 + C_1(a, j, v) + B_1(a, i') \quad \{i \geq 0, i' = i - 1, j = i + 1\}$$

$$B_1(a, i) = \text{nat}(i + 1) \times (\text{nat}(a - 1) \times 17 + 26)$$
\[ UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base} \]

\[
\begin{align*}
C_1(a, j, v) &= 3 & \{j \geq a\} \\
C_1(a, j, v) &= 8 & \{j < a\} \\
C_1(a, j, v) &= 17 + C_1(a, j', v) & \{j < a, j' = j + 1\} \\
\end{align*}
\]

\[
C_1(a, j, v) = \text{nat}(a - j) \times 17 + 8
\]

\[
\begin{align*}
B_1(a, i) &= 2 & \{i < 0\} \\
B_1(a, i) &= 18 + C_1(a, j, v) + B_1(a, i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\end{align*}
\]

\[
B_1(a, i) = \text{nat}(i + 1) \times (\text{nat}(a - 1) \times 17 + 26) + 2
\]
Phase 2: Solving the cost equations (example)

$$UB = \# \text{ iter} \times \text{max\_cost} + \text{max\_base}$$

\[
C_1(a, j, v) = \begin{cases} 
3 & \{j \geq a\} \\
8 & \{j < a\} \\
17 + C_1(a, j', v) & \{j < a, j' = j + 1\} 
\end{cases}
\]

\[
C_1(a, j, v) = \text{nat}(a - j) \times 17 + 8
\]

\[
B_1(a, i) = \begin{cases} 
2 & \{i < 0\} \\
18 + C_1(a, j, v) + B_1(a, i') & \{i \geq 0, i' = i - 1, j = i + 1\} 
\end{cases}
\]

\[
B_1(a, i) = \text{nat}(i + 1) \times (\text{nat}(a - 1) \times 17 + 26) + 2
\]

\[
\text{sort}(a) = 6 + B_1(a, i) \quad \{i = a - 2\}
\]
Phase 2: Solving the cost equations (example)

\[ \text{UB} = \# \text{ iter} \times \text{max} \text{ cost} + \text{max} \text{ base} \]

\[
\begin{align*}
\mathcal{C}_1(a, j, v) &= 3 & \{j \geq a\} \\
\mathcal{C}_1(a, j, v) &= 8 & \{j < a\} \\
\mathcal{C}_1(a, j, v) &= 17 + \mathcal{C}_1(a, j', v) & \{j < a, j' = j + 1\} \\
\mathcal{C}_1(a, j, v) &= \text{nat}(a - j) \times 17 + 8
\end{align*}
\]

\[
\begin{align*}
\mathcal{B}_1(a, i) &= 2 & \{i < 0\} \\
\mathcal{B}_1(a, i) &= 18 + \mathcal{C}_1(a, j, v) + \mathcal{B}_1(a, i') & \{i \geq 0, i' = i - 1, j = i + 1\} \\
\mathcal{B}_1(a, i) &= \text{nat}(i + 1) \times (\text{nat}(a - 1) \times 17 + 26) + 2
\end{align*}
\]

\[
\begin{align*}
\text{sort}(a) &= 6 + \mathcal{B}_1(a, i) & \{i = a - 2\} \\
\text{sort}(a) &= 8 + \text{nat}(a - 1) \times (\text{nat}(a - 1) \times 17 + 26)
\end{align*}
\]
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 $UB(\bar{v}) \geq M(t)$.
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]

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]

Elvira Albert
From Termination to Cost
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]
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
Reasoning about data stored in the heap is rather difficult:
Reasoning about data stored in the heap is rather difficult:

\[
\text{while } (i < n) \{ i++; o.m(); \}\]

\[n - i\] is a ranking function.
Reasoning about data stored in the heap is rather difficult:

\[\text{while } (i < n) \{ i++; o.m(); \} \]
\[n - i\] is a ranking function

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

\[
\text{while } (i < n) \{ i++; o.m(); \} \quad n-i \text{ is a ranking function}
\]

\[
\text{while } (i < f.n) \{ i++; o.m(); \} \quad f.n-i \text{ ranking function?}
\]

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:

```c
while (i < n) {
    i++; o.m();
}
```

$n - i$ is a ranking function

```c
while (i < f.n) {
    i++; o.m();
}
```

$f.n - i$ 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
Shared Mutable Data Structures

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

  ```
  while (i < n) { i++; o.m(); }
  ```

  $n-i$ is a ranking function

  $$
  while (i < f.n) \{ i++; o.m(); \}
  $$

  $f.n-i$ 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

1. Analyze the behavior of scopes or program fragments.
2. Model only those fields which behave as local variables.
3. The memory location does not change.
4. AI-based static analysis to prove constancy of references.
5. Write accesses are done through the same memory location.
6. Check after analysis.
7. Transform the code to replace local fields by variables.
8. Infer information on the fields through associated ghost variables.
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*
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
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 $\Rightarrow$ 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 $\Rightarrow$ AI-based static analysis to prove constancy of references
   - write accesses are done through the same memory location $\Rightarrow$ 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

```java
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 \( \Rightarrow \) AI-based static analysis to prove constancy of references
   - Write accesses are done through the same memory location \( \Rightarrow \) check after analysis

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

```java
while ( x.size > 0 ) {
    x.size++;
    y.size--;
}
```
analyze the behavior of scopes or program fragments

2. model only those fields which behave as local variables
   - the memory location does not change $\Rightarrow$ AI-based static analysis to prove constancy of references
   - write accesses are done through the same memory location $\Rightarrow$ 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
**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)
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]
First analysis to support numeric [FM’09] and reference fields [SAS’10] in cost/termination analysis of OO bytecode.
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)
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].
COSTA Team (UCM+UPM): Puri Arenas, Samir Genaim, Miguel G-Zamalloa, Germán Puebla, Damiano Zanardini, etc.
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 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
**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).

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

```plaintext
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).

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

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

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

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

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

```plaintext
if (k > 0)
then x=z else x=y;

x.f=10;
for(; i<x.f; i++)
    b[i]=x.b[i];
```

```plaintext
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 \( \ell \) for a variable \( y \) at program point \((k, j)\) is a syntactic construction:
  - \( \ell_{\text{any}} \). 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 \textit{read/write access paths} is the set of access path of variables \( y \) used for reading/writing \( f \) in \( S^* \).

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

\[ x.f.size = \ell_1.f \text{ and } y.size = \ell_2 \]

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

proving condition 2

\[ W(S, f) = \emptyset; \text{ or } W(S, f) = \{ \ell \} \text{ and } \ell \text{ is of the form } l_j.f_1 \ldots f_n. \]
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 $q(\langle \bar{x} \cdot \rho(\bar{v}_r') \rangle, \langle \bar{y} \cdot \rho(\bar{v}_w') \rangle)$
(1) $\text{loop} (\langle x, y, i \rangle, \langle r \rangle) :=$

$\quad s_0 := x, s_0 := s_0 . f,$

$\quad \text{getSize}(\langle s_0 \rangle, \langle s_0 \rangle),$

$\quad \text{loop}_c (\langle x, y, i, s_0 \rangle, \langle r \rangle).$

(2) $\text{loop}_c (\langle x, y, i, s_0 \rangle, \langle r \rangle) :=$

$\quad s_0 \leq 0, s_0 := i, r := s_0.$

(3) $\text{loop}_c (\langle x, y, i, s_0 \rangle, \langle r \rangle) :=$

$\quad s_0 > 0, s_0 := i, s_1 := y, \text{getSize}(\langle s_1 \rangle, \langle s_1 \rangle),$

$\quad s_0 := s_0 + s_1, i := s_0, s_0 := x, s_0 := s_0 . f,$

$\quad s_1 := s_0, \text{getSize}(\langle s_1 \rangle, \langle s_1 \rangle),$

$\quad s_2 := 1, s_1 := s_1 - s_2, \text{setSize}(\langle s_0, s_1 \rangle, \langle \rangle),$

$\quad \text{loop}(\langle this, x, y, i \rangle, \langle r \rangle).$

(4) $\text{getSize}(\langle this \rangle, \langle r \rangle) :=$

$\quad s_0 := \text{this}, s_0 := s_0 . \text{size}, r := s_0.$

(5) $\text{setSize}(\langle this, n \rangle, \langle \rangle) :=$

$\quad s_0 := \text{this}, s_1 := n, s_0 . \text{size} := s_1.$
(1) \( \text{loop}(\langle x, y, i, v_1\rangle, \langle r, v_1\rangle) := \)
\[
\begin{align*}
s_0 & := x, s_0 := s_0 \cdot f, \\
g & := \text{getSize}(\langle s_0, v_1\rangle, \langle s_0\rangle), \\
\text{loop}_c(\langle x, y, i, s_0, v_1\rangle, \langle r, v_1\rangle).
\end{align*}
\]

(2) \( \text{loop}_c(\langle x, y, i, s_0, v_1\rangle, \langle r, v_1\rangle) := \)
\[
\begin{align*}
s_0 & \leq 0, s_0 := i, r := s_0.
\end{align*}
\]

(3) \( \text{loop}_c(\langle x, y, i, s_0, v_1\rangle, \langle r, v_1\rangle) := \)
\[
\begin{align*}
s_0 & > 0, s_0 := i, s_1 := y, g := \text{getSize}(\langle s_1, \ast\rangle, \langle s_1\rangle), \\
s_0 & := s_0 + s_1, i := s_0, s_0 := x, s_0 := s_0 \cdot f, \\
s_1 & := s_0, g := \text{getSize}(\langle s_1, v_1\rangle, \langle s_1\rangle), \\
s_2 & := 1, s_1 := s_1 - s_2, \text{setSize}(\langle s_0, s_1\rangle, \langle v_1\rangle), \\
\text{loop}(\langle this, x, y, i, v_1\rangle, \langle r, v_1\rangle).
\end{align*}
\]

(4) \( \text{getSize}(\langle this, v_1\rangle, \langle r\rangle) := \)
\[
\begin{align*}
s_0 & := \text{this}, s_0 := v_1, r := s_0.
\end{align*}
\]

(5) \( \text{setSize}(\langle this, n\rangle, \langle v_1\rangle) := \)
\[
\begin{align*}
s_0 & := \text{this}, s_1 := n, v_1 := s_1.
\end{align*}
\]
What is it different in reference fields?

- Replicating instructions is not a good idea:
  - assume an instruction like \( y_{\text{ref}} := x \) is followed by \( v_{\text{ref}} := x \).
  - replicating instructions makes \( y_{\text{ref}} \) and \( v_{\text{ref}} \) alias,
  - therefore, the path-length relations of \( v_{\text{ref}} \) affected by those of \( y_{\text{ref}} \)
  - updates to \( y_{\text{ref}} \) will force losing path-length information about \( v_{\text{ref}} \),
  - replace \( y_{\text{ref}} := x \) by \( v_{\text{ref}} := x \), 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 \( \text{cursors} \) for traversing them
  - reference fields which are part of the data structure itself, seldomly affect termination or cost
  - require that the field signature is both read and written

\[
(R(S, f)) = (W(S, f)) = \{\ell\}
\]
**Iterator Example**

```java
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

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

```java
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

```java
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. Elvira Albert From Termination to Cost
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 \emph{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 $m$