Automatic Inference of Resource Consumption Bounds

Elvira Albert

Complutense University of Madrid (Spain)

The 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning

10-15 March, Mérida, Venezuela

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on a given input data static

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on a given input data *without actually executing* P

The aim of *static* COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data ***without actually executing*** P

automatic
static

The aim of ~~~~ COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data ***without actually executing*** P

automatic
static

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data *without actually executing* P

automatic
static

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data ***without actually executing*** P

- Upper Bounds (***worst case***)
- Lower Bounds (***best case***)

automatic

static

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ any input data *without actually executing* P

- Upper Bounds (*worst case*)
- Lower Bounds (*best case*)
- Non-Asymptotic: $P(x) = 2 + 3{\cdot}x + 2{\cdot}x^2$
- Asymptotic: $P(x) = O(x^2)$

WHAT IS COST ANALYSIS?

automatic

static

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data *without actually executing* P

automatic

static

The aim of ~~COST~~ COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ any input data *without actually executing* P

- Execution steps
- Visits to specific program points
- Memory (possibly with garbage collection)

automatic *static*

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data *without actually executing* P

- Execution steps
- Visits to specific program points
- Memory (possibly with garbage collection)

noncumulative

automatic

static

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data *without actually executing* P

- Execution steps
- Visits to specific program points
- Memory (possibly with garbage collection)

  noncumulative
- Time? Energy?

automatic
static

The aim of COST ANALYSIS is to bound the resource consumption (aka cost) of executing a given program P on ~~a given~~ *any* input data *without actually executing* P

- Execution steps
- Visits to specific program points
- Memory (possibly with garbage collection)
  noncumulative
- ~~Time? Energy?~~
  platform dependent

State of the Art

- Work on automatic cost analysis dates back to 1975, with the seminal work of Wegbreit
- His system was able to compute:
  - interesting results, but for
  - restricted class of functional programs

State of the Art

- Work on automatic cost analysis dates back to 1975, with the seminal work of Wegbreit
- His system was able to compute:
  - interesting results, but for
  - restricted class of functional programs
- Seminal work on abstract interpretation [Cousot & Cousot'77] mentions performance analysis as application

State of the Art

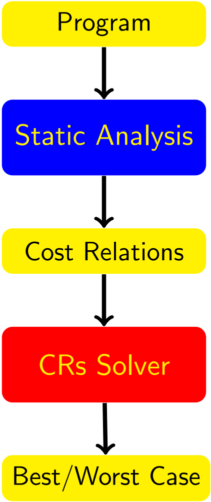
- Work on automatic cost analysis dates back to 1975, with the seminal work of Wegbreit
- His system was able to compute:
  - interesting results, but for
  - restricted class of functional programs
- Seminal work on abstract interpretation [Cousot & Cousot'77] mentions performance analysis as application
- Since then, a number of analyses and systems have been built which extend the capabilities of cost analysis:
  - functional programs [Le Metayer'88, Rosendahl'89, Wadler'88, Sands'95, Benzinger'04, ..., Hofmann'10, ...]
  - logic programs [Debray and Lin'93,..., Navas et al.'07,...]
  - imperative programs [Adachi et al.'79, Albert et al.'07, Gulwani'09, ...]

State of the Art

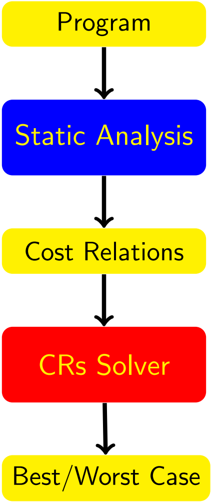
- Work on automatic cost analysis dates back to 1975, with the seminal work of Wegbreit
- His system was able to compute:
  - interesting results, but for
  - restricted class of functional programs
- Seminal work on abstract interpretation [Cousot & Cousot'77] mentions performance analysis as application
- Since then, a number of analyses and systems have been built which extend the capabilities of cost analysis:
  - functional programs [Le Metayer'88, Rosendahl'89, Wadler'88, Sands'95, Benzinger'04, ..., Hofmann'10, ...]
  - logic programs [Debray and Lin'93,..., Navas et al.'07,...]
  - imperative programs [Adachi et al.'79, Albert et al.'07, Gulwani'09, ...]

ESOP'07

A Classical Approach to Cost Analysis

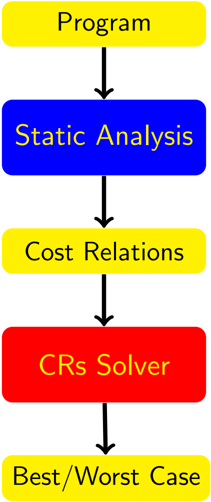


A Classical Approach to Cost Analysis



A classical approach [Wegbreit'75] to cost analysis consists of:

1. expressing the cost of a program part in terms of other program parts, thus obtaining *recurrence relations*
2. solving the relations by obtaining a *closed-form* for the cost in terms of the input arguments



A classical approach [Wegbreit'75] to cost analysis consists of:

1. expressing the cost of a program part in terms of other program parts, thus obtaining *recurrence relations*
2. solving the relations by obtaining a *closed-form* for the cost in terms of the input arguments

The current situation is that

- Most work has concentrated on the 1^{st} phase
- difficulties of the 2^{nd} phase have been overseen
- usage of cost analysis requires both!
- COSTA we address both phases.

What do we expect from cost analysis?

```
static void sort(int a[]) {

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

q     int j=i+1;
      int value=a[i];

      while ( j<a.length && a[j]<value) {
p        a[j-1]=a[j];
         j++;
      }

      a[j-1]=value;
   }

}
```

What do we expect from cost analysis?

```
static void sort(int a[]) {

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

q      int j=i+1;
       int value=a[i];

       while ( j<a.length && a[j]<value) {
p         a[j-1]=a[j];
          j++;
       }

       a[j-1]=value;
   }

}
```

What do we expect from cost analysis?

```
static void sort(int a[]) {
   for (int i=a.length-2; i>=0; i--) {

q     int j=i+1;
      int value=a[i];

      while ( j<a.length && a[j]<value) {
p        a[j-1]=a[j];
         j++;
      }

      a[j-1]=value;
   }
}
```

WHAT DO WE EXPECT FROM COST ANALYSIS?

```
static void sort(int a[]) {

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

q      int j=i+1;
       int value=a[i];

       while ( j<a.length && a[j]<value) {
p         a[j-1]=a[j];
          j++;
       }

       a[j-1]=value;
   }

}
```

What do we expect from cost analysis?

```
static void sort(int a[]) {

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

      int j=i+1;
      int value=a[i];

      while ( j<a.length && a[j]<value) {
         a[j-1]=a[j];
         j++;
      }

      a[j-1]=value;
   }

}
```

q

p

What do we expect from cost analysis?

```
static void sort(int a[]) {

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

q     int j=i+1;
      int value=a[i];

      while ( j<a.length && a[j]<value) {
p        a[j-1]=a[j];
         j++;
      }

      a[j-1]=value;
   }

}
```

What do we expect from cost analysis?

```
static void sort(int a[]) {
  for (int i=a.length-2; i>
q     int j=i+1;
      int value=a[i];

      while ( j<a.length &&
p        a[j-1]=a[j];
         j++;
      }

      a[j-1]=value;
  }

}
```

Worst-Case (UB)

$$sort(a) = q * (a-1) + p * (a-1)^2$$

What do we expect from cost analysis?

```
static void sort(int a[]) {

   for (int i=a.length-2; i>

q     int j=i+1;
      int value=a[i];

      while ( j<a.length &&
p        a[j-1]=a[j];
         j++;
      }

      a[j-1]=value;
   }

}
```

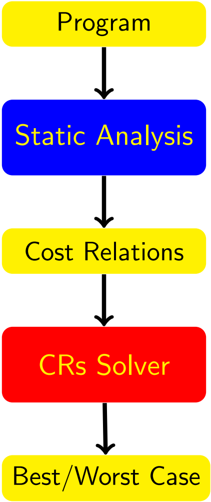
Worst-Case (UB)

$sort(a) = q * (a-1) + p * (a-1)^2$

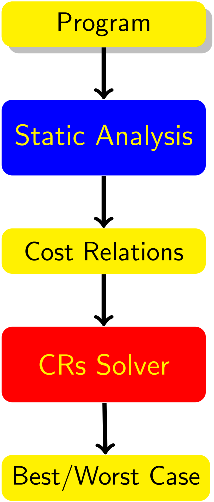
Best-Case (LB)

$sort(a) = q * (a-1)$

First phase: Generation of Cost Relations

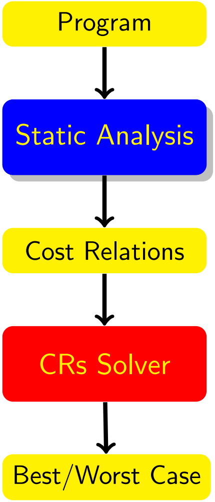


First phase: Generation of Cost Relations



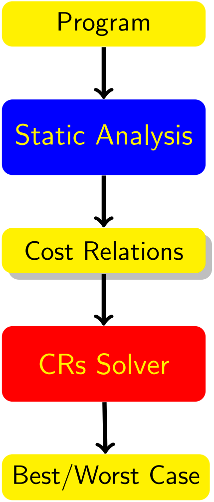

```
static void sort(int a[]) {
   for (int i=a.length-2; i>=0; i--) {
q      int j=i+1;
       int value=a[i];
       while ( j<a.length && a[j]<value ) {
p         a[j-1]=a[j];
          j++;
       }
       a[j-1]=value;
   }
}
```

First phase: Generation of Cost Relations




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

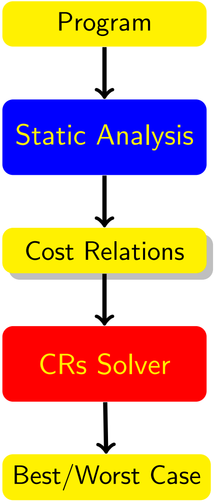
First phase: Generation of Cost Relations




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

$$\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}$$

First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\[6pt]
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\[6pt]
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$

Extracting loops

Loops are extracted from the CFG of the program and cost relations are generated loop by loop

```
nt a[]) {
ngth-2; i>=0; i--) {

i];
ength && a[j]<value ) {
];
j++;
        }
        a[j-1]=value;
    }
}
```

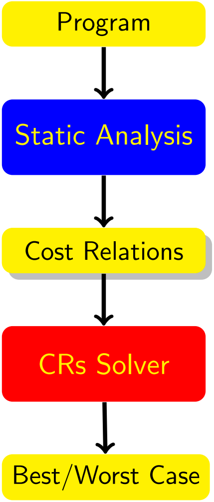
Cost Relations

CRs Solver

Best/Worst Case

$sort(a) = B(a,i)$	$\{i{=}a{-}2,\ a{\geq}0\}$
$B(a,i) = 0$	$\{i{<}0\}$
$B(a,i) = q + C(a,j) + B(a,i')$	$\{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$
$C(a,j) = 0$	$\{j{\geq}a\}$
$C(a,j) = 0$	$\{j{<}a\}$
$C(a,j) = p + C(a,j')$	$\{j{<}a,\ j'{=}j{+}1\}$

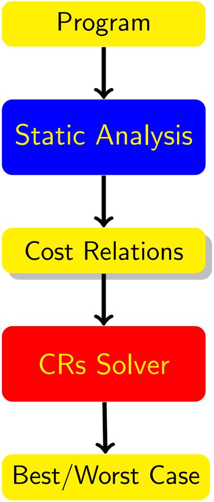
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
 & \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
 & \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

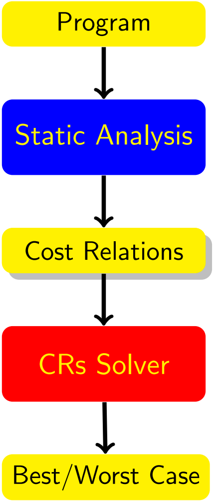
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$

First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

Size analysis

Global size analysis is performed in order to infer how the sizes of data change along execution

Cost Relations

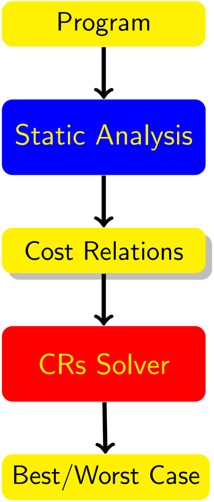
CRs Solver

Best/Worst Case

```
nt a[]) {
ngth-2; i>=0; i--) {

i];
ength && a[j]<value ) {
];
j++;
        }
        a[j-1]=value;
    }
}
```

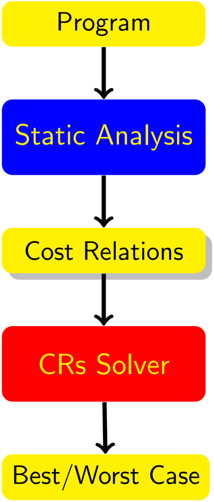
$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$

First phase: Generation of Cost Relations

Program

Static Analysis

Cost Relations

CRs Solver

Best/Worst Case

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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

Modular definition

Cost relations define the cost of executing a program fragment in terms of the cost of executing other fragments

```
nt a[]) {
ngth-2; i>=0; i--) {
i];
ength && a[j]<value ) {
];
j++;
      }
      a[j-1]=value;
   }
}
```

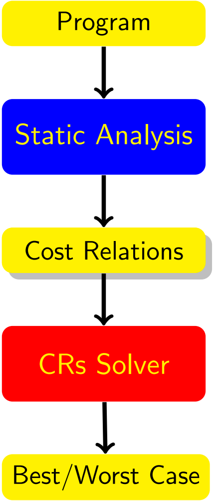
Cost Relations

CRs Solver

Best/Worst Case

$sort(a) = B(a,i)$	$\{i{=}a{-}2,\ a{\geq}0\}$
$B(a,i) = 0$	$\{i{<}0\}$
$B(a,i) = q + C(a,j) + B(a,i')$	$\{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$
$C(a,j) = 0$	$\{j{\geq}a\}$
$C(a,j) = 0$	$\{j{<}a\}$
$C(a,j) = p + C(a,j')$	$\{j{<}a,\ j'{=}j{+}1\}$

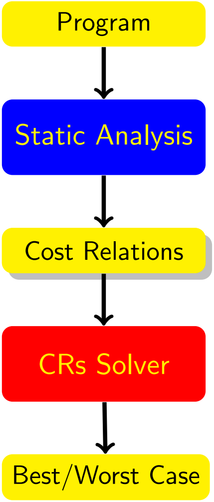
First phase: Generation of Cost Relations




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

$sort(a) = B(a,i)$	$\{i=a-2,\ a\geq 0\}$
$B(a,i) = 0$	$\{i<0\}$
$B(a,i) = q + C(a,j) + B(a,i')$	$\{i\geq 0,\ j=i+1,\ i'=i-1\}$
$C(a,j) = 0$	$\{j\geq a\}$
$C(a,j) = 0$	$\{j<a\}$
$C(a,j) = p + C(a,j')$	$\{j<a,\ j'=j+1\}$

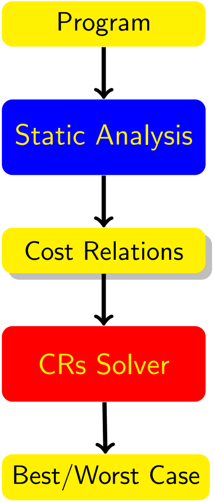
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i = a-2,\ a \geq 0\} \\
B(a,i) = 0 & \{i < 0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i \geq 0,\ j = i+1,\ i' = i-1\} \\
C(a,j) = 0 & \{j \geq a\} \\
C(a,j) = 0 & \{j < a\} \\
C(a,j) = p + C(a,j') & \{j < a,\ j' = j+1\}
\end{array}
$$

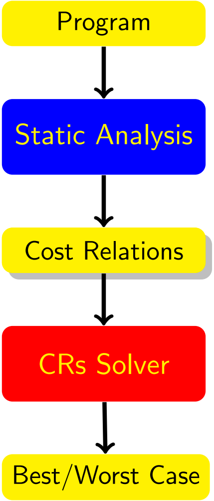
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
 & \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$

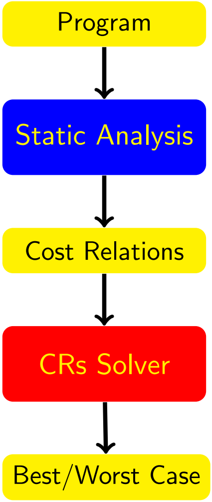
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
 & \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
 & \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

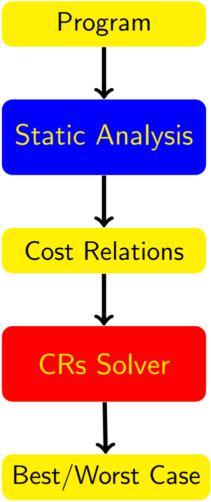
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
\\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
\\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$

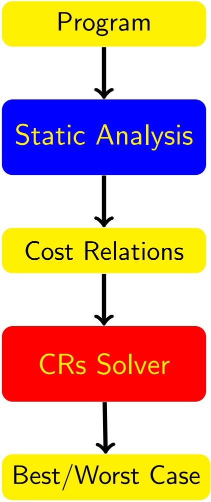
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
 & \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

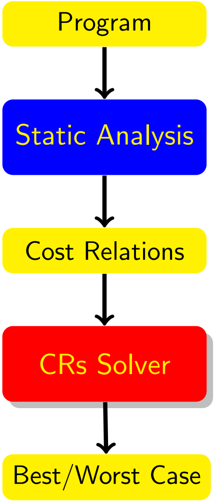
First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i = a-2,\ a \geq 0\} \\
B(a,i) = 0 & \{i < 0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i \geq 0,\ j = i+1,\ i' = i-1\} \\
 & \\
C(a,j) = 0 & \{j \geq a\} \\
C(a,j) = 0 & \{j < a\} \\
C(a,j) = p + C(a,j') & \{j < a,\ j' = j+1\}
\end{array}
$$

First phase: Generation of Cost Relations




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

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
 & \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
 & \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$

Summary: Generation of Cost Relations

Summary: Generation of Cost Relations

Program ⟶ **Static Analysis** ⟶ Cost Relations

- Apply class analysis to compute the reachable code

Summary: Generation of Cost Relations

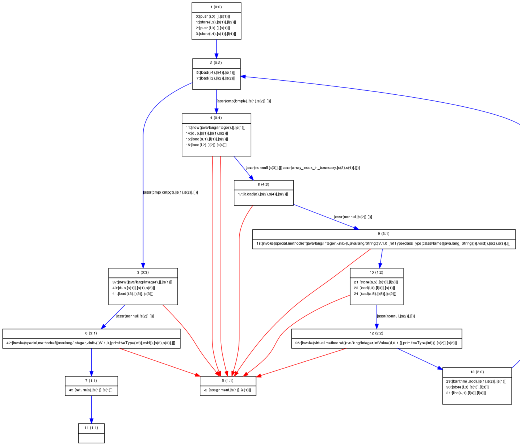
Program $\longrightarrow$ Static Analysis $\longrightarrow$ Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method

Summary: Generation of Cost Relations

Prog … Relations

- Ap … e
- Co



Summary: Generation of Cost Relations

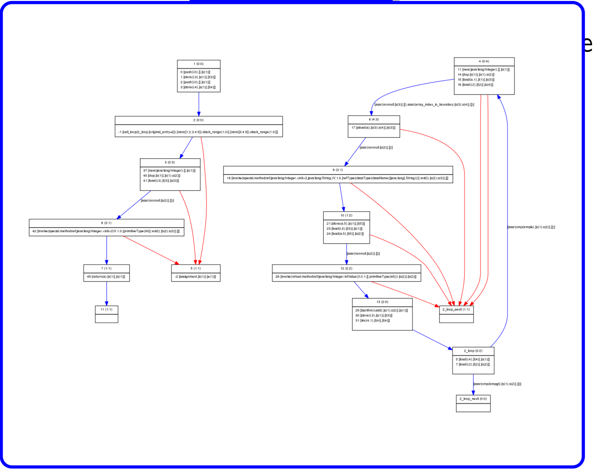
Program $\longrightarrow$ **Static Analysis** $\longrightarrow$ Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops

Summary: Generation of Cost Relations

Program → Static Analysis → Cost Relations

- Ap … e
- Co
- Ide



Summary: Generation of Cost Relations

Program $\longrightarrow$ **Static Analysis** $\longrightarrow$ Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops

Summary: Generation of Cost Relations

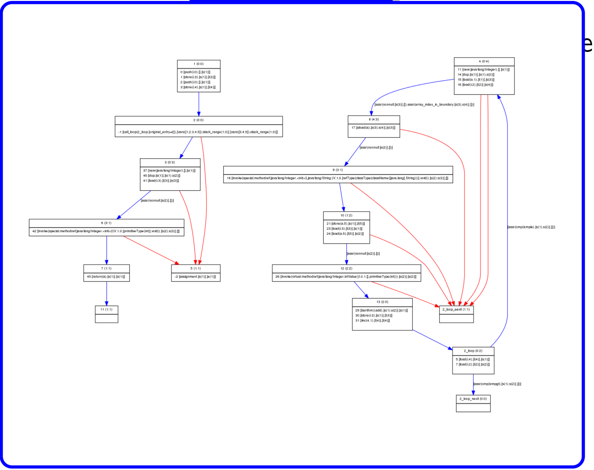
Program → **Static Analysis** → Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds

Summary: Generation of Cost Relations

Program → Static Analysis → Cost Relations

- Ap
- Co
- Ide
- Eli



e

Summary: Generation of Cost Relations

Program → **Static Analysis** → Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost

Summary: Generation of Cost Relations

Program ⟶ **Static Analysis** ⟶ Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost
- Cyclicity analysis: identify cyclic data structures

Summary: Generation of Cost Relations

Program → **Static Analysis** → Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost
- Cyclicity analysis: identify cyclic data structures

```
while ( x != null ) {
  x = x.next;
}
```

Summary: Generation of Cost Relations

Program $\longrightarrow$ Static Analysis $\longrightarrow$ Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost
- Cyclicity analysis: identify cyclic data structures

Summary: Generation of Cost Relations

Program → **Static Analysis** → Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost
- Cyclicity analysis: identify cyclic data structures
- Size analysis: infers relations between variables

Summary: Generation of Cost Relations

Program → Static Analysis → Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost
- Cyclicity analysis: identify cyclic data structures
- Size analysis: infers relations between variables
- Generate cost relations

Summary: Generation of Cost Relations

Program ⟶ **Static Analysis** ⟶ Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost
- Cyclicity analysis: identify cyclic data structures
- Size analysis: infers relations between variables
- Generate cost relations

```
C(a,j)=0                {j>=a}
C(a,j)=0                {j<a}
C(a,j)=p+C(a,j-1)       {j<a,j'=j+1}
```

Summary: Generation of Cost Relations

Program → Static Analysis → Cost Relations

- Apply class analysis to compute the reachable code
- Construct a CFG for each method
- Identify and separate (for/while/do) loops
- Eliminate dead code: nullity, array bounds
- Slicing: eliminates variables that is irrelevant to cost
- Cyclicity analysis: identify cyclic data structures
- Size analysis: infers relations between variables
- Generate cost relations
- References: ESOP'07, TCS'12

Second phase: Solving CRs

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

Second phase: Solving CRs

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

- Why not using directly Computer Algebra Systems?

Second phase: Solving CRs

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

CAS can obtain an exact closed-form solution for:

$$P(0) = 0$$
$$P(n) = E + P(n-1) + \cdots + P(n-1)$$

deterministic, 1 base-case, 1 recursive case, 1 argument

Second phase: Solving CRs

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

- Why not using directly Computer Algebra Systems?
- CRs are not deterministic

Second phase: Solving CRs

$$sort(a) = B(a,i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a,i) = 0 \qquad \{i{<}0\}$$
$$B(a,i) = q + C(a,j) + B(a,i') \qquad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$$

$$C(a,j) = 0 \qquad \{j{\geq}a\}$$
$$C(a,j) = 0 \qquad \{j{<}a\}$$
$$C(a,j) = p + C(a,j') \qquad \{j{<}a,\ j'{=}j{+}1\}$$

- Wh
- CRs

Several possible runs for $C(10,1)$

$C(10,1) \rightarrow C(10,2)$
$C(10,1) \rightarrow C(10,2) \rightarrow C(10,3)$
$C(10,1) \rightarrow C(10,2) \rightarrow C(10,3) \rightarrow C(10,4)$

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$

- Why not using directly Computer Algebra Systems?
- CRs are not deterministic
- CRs have multiple arguments

Second phase: Solving CRs

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i = a-2,\ a \geq 0\} \\
B(a,i) = 0 & \{i < 0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i \geq 0,\ j = i+1,\ i' = i-1\} \\
C(a,j) = 0 & \{j \geq a\} \\
C(a,j) = 0 & \{j < a\} \\
C(a,j) = p + C(a,j') & \{j < a,\ j' = j+1\}
\end{array}
$$

- Why not using directly Computer Algebra Systems?
- CRs are not deterministic
- CRs have multiple arguments
- CRs have multiple (not mutually exclusive) equations

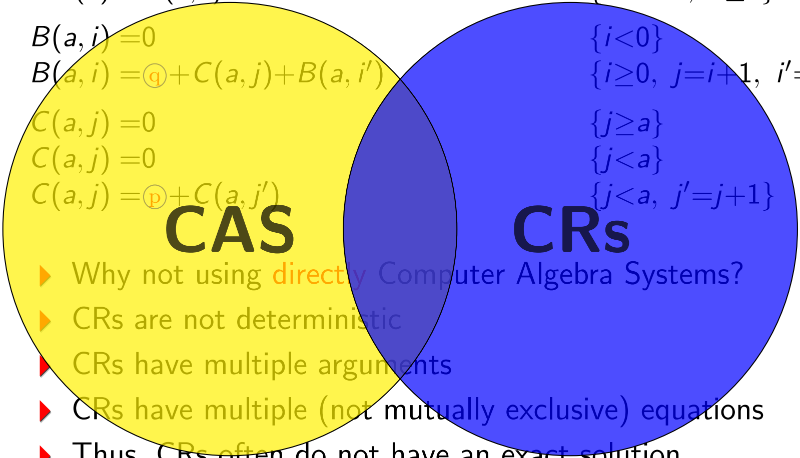
Second phase: Solving CRs

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

- Why not using directly Computer Algebra Systems?
- CRs are not deterministic
- CRs have multiple arguments
- CRs have multiple (not mutually exclusive) equations
- Thus, CRs often do not have an exact solution

Second phase: Solving CRs

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i=a-2,\ a\geq 0\} \\
B(a,i) = 0 & \{i<0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i\geq 0,\ j=i+1,\ i'=i-1\} \\
C(a,j) = 0 & \{j\geq a\} \\
C(a,j) = 0 & \{j<a\} \\
C(a,j) = p + C(a,j') & \{j<a,\ j'=j+1\}
\end{array}
$$



- Why not using directly Computer Algebra Systems?
- CRs are not deterministic
- CRs have multiple arguments
- CRs have multiple (not mutually exclusive) equations
- Thus, CRs often do not have an exact solution

A Practical Upper Bound Solver (PUBS)

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

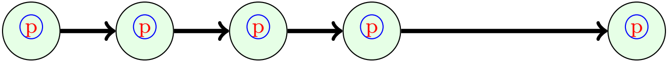
A Practical Upper Bound Solver (PUBS)

$$sort(a) = B(a,i) \quad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a,i) = 0 \quad \{i{<}0\}$$
$$B(a,i) = q + C(a,j) + B(a,i') \quad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$$

$$C(a,j) = 0 \quad \{j{\geq}a\}$$
$$C(a,j) = 0 \quad \{j{<}a\}$$
$$C(a,j) = p + C(a,j') \quad \{j{<}a,\ j'{=}j{+}1\}$$

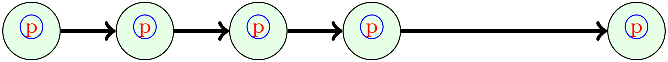
- An evaluation for $C(a_0, j_0)$ looks like:



A Practical Upper Bound Solver (PUBS)

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

- An evaluation for $C(a_0, j_0)$ looks like:



- How many p has this chain ?

A Practical Upper Bound Solver (PUBS)

$$sort(a) = B(a,i) \qquad \{i=a-2,\ a\geq 0\}$$

$$B(a,i) = 0 \qquad \{i<0\}$$

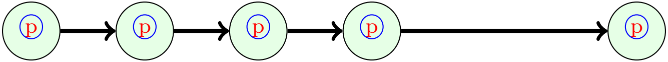
$$B(a,i) = q + C(a,j) + B(a,i') \qquad \{i\geq 0,\ j=i+1,\ i'=i-1\}$$

$$C(a,j) = 0 \qquad \{j\geq a\}$$

$$C(a,j) = 0 \qquad \{j<a\}$$

$$C(a,j) = p + C(a,j') \qquad \{\underbrace{j<a,\ j'=j+1}_{\varphi}\}$$

- An evaluation for $C(a_0, j_0)$ looks like:



- How many p has this chain ?

A Practical Upper Bound Solver (PUBS)

Ranking function

We are seeking a (ranking) function $\hat{f}$ that maps the program states to integers, such that

$$\forall i,j,j'.\ \varphi \models \hat{f}(i,j) - \hat{f}(i,j') \geq 1 \ \wedge\ \hat{f}(i,j) \geq 0$$

There are automatic techniques for synthesizing such functions [Sohn and Van Gelder 1991, Podelski and Rybalchenko 2004]

so … $0\}$

$B($ …

$B($ … $,\ i'{=}i{-}1\}$

$C($ …

$C($ …

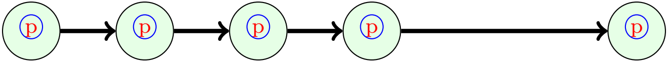
$C($ … $\}$

- A …

- How many p has this chain ?

$$
\begin{array}{ll}
sort(a) = B(a,i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \{i{<}0\} \\
B(a,i) = q + C(a,j) + B(a,i') & \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

- An evaluation for $C(a_0, j_0)$ looks like:



- How many p has this chain ?

$$\hat{f}(a_0, j_0) = \mathtt{nat}(a_0 - j_0)$$

A Practical Upper Bound Solver (PUBS)

$$
\begin{array}{ll}
sort(a) = B(a, i) & \{i{=}a{-}2,\ a{\geq}0\} \\
B(a,i) = 0 & \ldots 0\} \\
B(a,i) = q + \ldots & \ldots{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\} \\
C(a,j) = 0 & \{j{\geq}a\} \\
C(a,j) = 0 & \{j{<}a\} \\
C(a,j) = p + C(a,j') & \{j{<}a,\ j'{=}j{+}1\}
\end{array}
$$

Worst-Case

$$C(a_0, j_0) = p * \mathtt{nat}(a_0 - j_0)$$

- An evaluation for $C(a_0, j_0)$ looks like:

$$p \rightarrow p \rightarrow p \rightarrow p \longrightarrow p$$

- How many p has this chain ?

$$\hat{f}(a_0, j_0) = \mathtt{nat}(a_0 - j_0)$$

$sort(a) = B(a, i)$ $\{i = a-2,\ a \geq 0\}$

$B(a, i) = 0$ $\ldots 0\}$

$B(a, i) = q + \ldots$ $\ldots \geq 0,\ j = i+1,\ i' = i-1\}$

$C(a, j) = 0$ $\{j \geq a\}$

$C(a, j) = 0$ $\{j < a\}$

$C(a, j) = p + C(a, j') + C(a, j')$ $\{j < a,\ j' = j+1\}$

Worst-Case

$$C(a_0, j_0) = p * 2^{\mathtt{nat}(a_0 - j_0)}$$

- An evaluation for $C(a_0, j_0)$ looks like:

$$p \rightarrow p \rightarrow p \rightarrow p \longrightarrow p$$

- How many p has this chain ?

$$\hat{f}(a_0, j_0) = \mathtt{nat}(a_0 - j_0)$$

A Practical Upper Bound Solver (PUBS)

$$sort(a) = B(a, i) \qquad \{i = a - 2,\ a \geq 0\}$$

$$B(a, i) = 0 \qquad \{i < 0\}$$

$$B(a, i) = q + C(a, j) + B(a, i') \qquad \{i \geq 0,\ j = i + 1,\ i' = i - 1\}$$

A Practical Upper Bound Solver (PUBS)

$$sort(a) = B(a,i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a,i) = 0 \qquad \{i{<}0\}$$

$$B(a,i) = q + C(a,j) + B(a,i') \qquad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$$

A Practical Upper Bound Solver (PUBS)

$$sort(a) = B(a,i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a,i) = 0 \qquad \{i{<}0\}$$

$$B(a,i) = q + p * \mathtt{nat}(a{-}j) + B(a,i') \qquad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$$

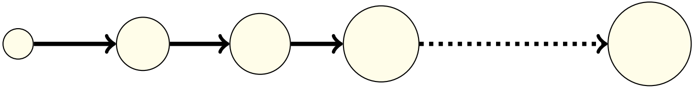
A Practical Upper Bound Solver (PUBS)

$$sort(a) = B(a,i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a,i) = 0 \qquad \{i{<}0\}$$

$$B(a,i) = q + p * \mathtt{nat}(a{-}j) + B(a,i') \qquad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$$

- An evaluation for $B(a_0, i_0)$ looks like:



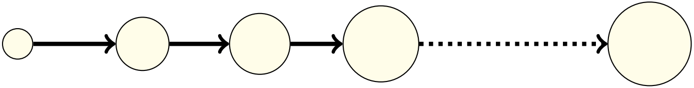
A Practical Upper Bound Solver (PUBS)

$$\mathit{sort}(a) = B(a,i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a,i) = 0 \qquad \{i{<}0\}$$

$$B(a,i) = q + p * \mathtt{nat}(a{-}j) + B(a,i') \qquad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$$

- An evaluation for $B(a_0, i_0)$ looks like:



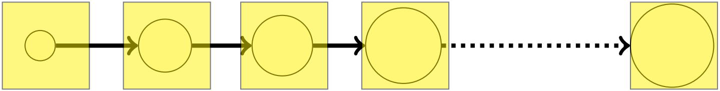
- There are at most $\mathtt{nat}(i_0 + 1)$ circles (*ranking function*)

$$sort(a) = B(a,i) \qquad \{i = a-2,\ a \geq 0\}$$

$$B(a,i) = 0 \qquad \{i < 0\}$$

$$B(a,i) = q + p * \mathtt{nat}(a-j) + B(a,i') \qquad \{i \geq 0,\ j = i+1,\ i' = i-1\}$$

- An evaluation for $B(a_0, i_0)$ looks like:



- There are at most $\mathtt{nat}(i_0 + 1)$ circles (*ranking function*)

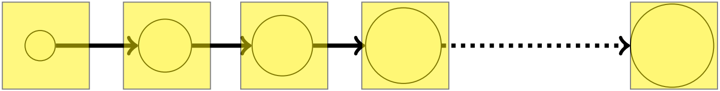
A Practical Upper Bound Solver (PUBS)

$sort(a) = B(a,i) \qquad \{i=a-2,\ a\geq 0\}$

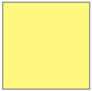
$B(a,i) = 0 \qquad \{i<0\}$

$B(a,i) = q + p * \mathtt{nat}(a-j) + B(a,i') \qquad \{i\geq 0,\ j=i+1,\ i'=i-1\}$

- An evaluation for $B(a_0, i_0)$ looks like:



- There are at most $\mathtt{nat}(i_0+1)$ circles (*ranking function*)

- Worst-case is  $*\ \mathtt{nat}(i_0+1)$

$sort(a) = B(a, i) \quad \{i = a - 2,\ a \geq 0\}$

$B(a, i) = 0 \quad \{i < 0\}$

$B(a, i) = q + p * \mathtt{nat}(a - j) + B(a, i') \quad \underbrace{\{i \geq 0,\ j = i + 1,\ i' = i - 1\}}_{\varphi}$

Inferring the box (or maximizing expression)

- What is the maximum value that $q + p * \mathtt{nat}(a - j)$ can take in terms of the initial values $\langle a_0, i_0 \rangle$?

$$sort(a) = B(a, i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a, i) = 0 \qquad \{i{<}0\}$$

$$B(a, i) = q + p * \mathtt{nat}(a{-}j) + B(a, i') \qquad \underbrace{\{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}}_{\varphi}$$

Inferring the box (or maximizing expression)

- What is the maximum value that $q + p * \mathtt{nat}(a - j)$ can take in terms of the initial values $\langle a_0, i_0 \rangle$?
- It is $q + p * \mathtt{nat}(a_0 - 1)$

A Practical Upper Bound Solver (PUBS)

$sort(a) = B(a, i) \quad \{i{=}a{-}2,\ a{\geq}0\}$

$B(a, i) = 0 \quad \{i{<}0\}$

$B(a, i) = q + p * \mathtt{nat}(a{-}j) + B(a, i') \quad \underbrace{\{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}}_{\varphi}$

Inferring the box (or maximizing expression)

- What is the maximum value that $q + p * \mathtt{nat}(a - j)$ can take in terms of the initial values $\langle a_0, i_0 \rangle$?
- It is $q + p * \mathtt{nat}(a_0 - 1)$
- Infer an invariant $\langle B(a_0, i_0) \rightsquigarrow B(a, i), \Psi \rangle$

A Practical Upper Bound Solver (PUBS)

$sort(a) = B(a, i)$ $\{i = a - 2,\ a \geq 0\}$

$B(a, i) = 0$ $\{i < 0\}$

$B(a, i) = q + p * \mathtt{nat}(a - j) + B(a, i')$ $\{\underbrace{i \geq 0,\ j = i + 1,\ i' = i - 1}_{\varphi}\}$

Inferring the box (or maximizing expression)

- What is the maximum value that $q + p * \mathtt{nat}(a - j)$ can take in terms of the initial values $\langle a_0, i_0 \rangle$?
- It is $q + p * \mathtt{nat}(a_0 - 1)$
- Infer an invariant $\langle B(a_0, i_0) \rightsquigarrow B(a, i), \Psi \rangle$
- Use (parametric) integer programming to maximize $a - j$ w.r.t $\Psi \wedge \varphi$ and the parameters $\langle a_0, i_0 \rangle$

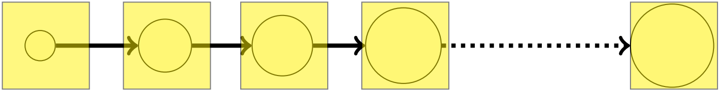
A Practical Upper Bound Solver (PUBS)

$$sort(a) = B(a, i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$$

$$B(a, i) = 0 \qquad \{i{<}0\}$$

$$B(a, i) = q + p * \texttt{nat}(a{-}j) + B(a, i') \qquad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$$

- An evaluation for $B(a_0, i_0)$ looks like:



- There are at most $\texttt{nat}(i_0 + 1)$ circles (*ranking function*)

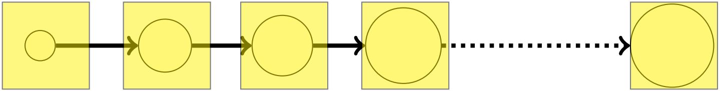
- $B(a_0, i_0) = [q + p * \texttt{nat}(a_0 - 1)] * \texttt{nat}(i_0 + 1)$

$sort(a) = B(a, i) \qquad \{i{=}a{-}2,\ a{\geq}0\}$

$B(a, i) = 0 \qquad \{i{<}0\}$

$B(a, i) = q + p * \mathtt{nat}(a{-}j) + B(a, i') \qquad \{i{\geq}0,\ j{=}i{+}1,\ i'{=}i{-}1\}$

- An evaluation for $B(a_0, i_0)$ looks like:



- There are at most $\mathtt{nat}(i_0 + 1)$ circles (*ranking function*)

- $B(a_0, i_0) = [q + p * \mathtt{nat}(a_0{-}1)] * \mathtt{nat}(i_0{+}1)$

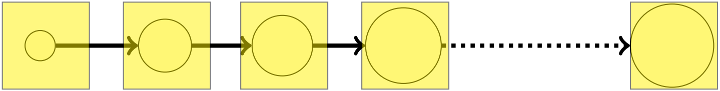
A Practical Upper Bound Solver (PUBS)

$$sort(a) = [q + p * \mathtt{nat}(a-1)] * \mathtt{nat}(i+1) \qquad \{i=a-2,\ a\geq 0\}$$

$$B(a,i) = 0 \qquad \{i<0\}$$

$$B(a,i) = q + p * \mathtt{nat}(a-j) + B(a,i') \qquad \{i\geq 0,\ j=i+1,\ i'=i-1\}$$

- An evaluation for $B(a_0, i_0)$ looks like:



- There are at most $\mathtt{nat}(i_0+1)$ circles (*ranking function*)

- $B(a_0, i_0) = [q + p * \mathtt{nat}(a_0-1)] * \mathtt{nat}(i_0+1)$

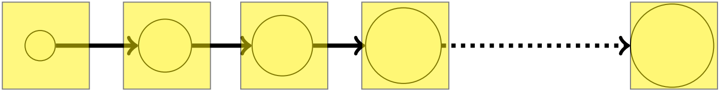
A Practical Upper Bound Solver (PUBS)

$$sort(a) = [q + p * \mathtt{nat}(a-1)] * \mathtt{nat}(i+1) \qquad \{i = a-2,\ a \geq 0\}$$

$$B(a,i) = 0 \qquad \{i < 0\}$$

$$B(a,i) = q + p * \mathtt{nat}(a-j) + B(a,i') \qquad \{i \geq 0,\ j = i+1,\ i' = i-1\}$$

- An evaluation for $B(a_0, i_0)$ looks like:



- There are at most $\mathtt{nat}(i_0 + 1)$ circles (*ranking function*)

- $B(a_0, i_0) = [q + p * \mathtt{nat}(a_0 - 1)] * \mathtt{nat}(i_0 + 1)$

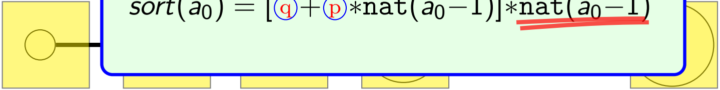
A Practical Upper Bound Solver (PUBS)

$sort(a) = [q + p * \mathtt{nat}(a-1)] * \mathtt{nat}(i+1) \qquad \{i=a-2,\ a\geq 0\}$

$B(a,i) = 0 \qquad \{i<0\}$

$B(a,i) = q + p * \mathtt{nat}(a-j) + B(a,i') \qquad \{i\geq 0,\ j=i+1,\ i'=i-1\}$

- An evalu



Worst-case UB for *sort*

$sort(a_0) = [q + p * \mathtt{nat}(a_0-1)] * \mathtt{nat}(a_0-1)$

- There are at most $\mathtt{nat}(i_0+1)$ circles (*ranking function*)

- $B(a_0, i_0) = [q + p * \mathtt{nat}(a_0-1)] * \mathtt{nat}(i_0+1)$

Summary: Generation of Cost Relations

- Cost relations: common target of cost analyzers for any programming language (widely applicable)

Summary: Generation of Cost Relations

- Cost relations: common target of cost analyzers for any programming language (widely applicable)
- Transformation of relations into direct recursion form (using the technique of partial evaluation)

Summary: Generation of Cost Relations

- Cost relations: common target of cost analyzers for any programming language (widely applicable)
- Transformation of relations into direct recursion form (using the technique of partial evaluation)
- Bound the maximum number of iterations for all recursive relations (ranking functions)

Summary: Generation of Cost Relations

- Cost relations: common target of cost analyzers for any programming language (widely applicable)
- Transformation of relations into direct recursion form (using the technique of partial evaluation)
- Bound the maximum number of iterations for all recursive relations (ranking functions)
- Maximize the cost of iterations (relying on invariants and PIP)

Summary: Generation of Cost Relations

- Cost relations: common target of cost analyzers for any programming language (widely applicable)
- Transformation of relations into direct recursion form (using the technique of partial evaluation)
- Bound the maximum number of iterations for all recursive relations (ranking functions)
- Maximize the cost of iterations (relying on invariants and PIP)
- Upper bounds of polynomial, logarithmic, exponential complexity (any combitation of those elementary costs)

Summary: Generation of Cost Relations

- Cost relations: common target of cost analyzers for any programming language (widely applicable)
- Transformation of relations into direct recursion form (using the technique of partial evaluation)
- Bound the maximum number of iterations for all recursive relations (ranking functions)
- Maximize the cost of iterations (relying on invariants and PIP)
- Upper bounds of polynomial, logarithmic, exponential complexity (any combitation of those elementary costs)
- Combined with CAS to obtain more precise lower bounds

Summary: Generation of Cost Relations

- Cost relations: common target of cost analyzers for any programming language (widely applicable)
- Transformation of relations into direct recursion form (using the technique of partial evaluation)
- Bound the maximum number of iterations for all recursive relations (ranking functions)
- Maximize the cost of iterations (relying on invariants and PIP)
- Upper bounds of polynomial, logarithmic, exponential complexity (any combitation of those elementary costs)
- Combined with CAS to obtain more precise lower bounds
- References: SAS'08, JAR'11, VMCAI'11

Theorem (soundness)

- *Let $P(\bar{x})$ be a method,*
- *R the resource we are measuring,*
- *$UB(\bar{x})$ 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 $\boxed{UB(\bar{v}) \geq R(t)}$

Advanced Topics in Current Research

- Non-cumulative types of resources (memory)
- Certification of results
- Handling the shared memory (heap)
- Modularity, incrementality of analysis
- Concurrency in cost analysis
- Implementing a cost analyzer

Non-Cumulative Resources

- Total memory consumption (ignoring GC):
  - Without GC it is a cumulative resource.
  - General cost analyis framework directly applicable.

Non-Cumulative Resources

- Total memory consumption (ignoring GC):
  - Without GC it is a cumulative resource.
  - General cost analyis framework directly applicable.
- Peak of memory consumption:
  - With GC the memory consumption increases and decreases along the execution (maximum among all states).
  - GC is unpredictable $\Rightarrow$ We need to assume some characteristics of it.
    - "When" it is applied (scope-based vs. ideal)
    - "What" it is removed (reachability vs. liveness)

Non-Cumulative Resources

- Total memory consumption (ignoring GC):
  - Without GC it is a cumulative resource.
  - General cost analyis framework directly applicable.
- Peak of memory consumption:
  - With GC the memory consumption increases and decreases along the execution (maximum among all states).
  - GC is unpredictable $\Rightarrow$ We need to assume some characteristics of it.
    - "When" it is applied (scope-based vs. ideal)
    - "What" it is removed (reachability vs. liveness)
- Assumption: scope-based GC based on reachability
  - Scoped memory managers are common.
  - Useful for estimating consumption in systems with stack-allocation.

Memory Consumption with Scope-based GC

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Memory Consumption with Scope-based GC

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Memory Consumption with Scope-based GC

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Total

$\mathtt{print}^{ub}(\mathtt{t}) = 2^{\mathtt{t}} * \mathrm{s}(\mathrm{SB})$

Memory Consumption with Scope-based GC

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Total

$$\mathtt{print}^{ub}(\mathtt{t}) = 2^{\mathtt{t}} * \mathrm{s(SB)}$$

Peak (Scope-Based GC)

$$\mathtt{print}^{ub}(\mathtt{t}) = \mathtt{t} * \mathrm{s(SB)}$$

Memory Consumption with Scope-based GC

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①   // new StringBuffer(...)
    if ( left != null ) left.print;
②   if ( right != null ) right.print;
  }
}
```

Total

$$\mathtt{print}^{ub}(\mathtt{t}) = 2^{\mathtt{t}} * s(SB)$$

Peak (Scope-Based GC)

$$\mathtt{print}^{ub}(\mathtt{t}) = \mathtt{t} * s(SB)$$

print()

$s(SB)$

l.print()

Memory Consumption with Scope-based GC

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Total

$$\texttt{print}^{ub}(\texttt{t}) = 2^{\texttt{t}} * \mathrm{s(SB)}$$

Peak (Scope-Based GC)

$$\texttt{print}^{ub}(\texttt{t}) = \texttt{t} * \mathrm{s(SB)}$$

$$\texttt{print()} \xrightarrow{s(SB)} \texttt{l.print()} \xrightarrow{s(SB)} \texttt{l.l.print()}$$

Memory Consumption with Scope-based GC

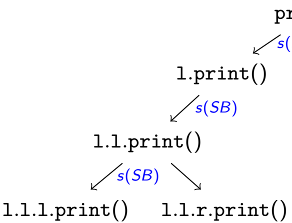
```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
 1  // new StringBuffer(...)
    if ( left != null ) left.print;
 2  if ( right != null ) right.print;
  }
}
```

Total

$$\mathtt{print}^{ub}(\mathtt{t}) = 2^{\mathtt{t}} * s(SB)$$

Peak (Scope-Based GC)

$$\mathtt{print}^{ub}(\mathtt{t}) = \mathtt{t} * s(SB)$$



Memory Consumption with Scope-based GC

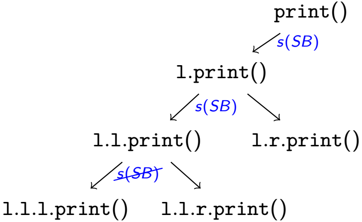
```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
 1  // new StringBuffer(...)
    if ( left != null ) left.print;
 2  if ( right != null ) right.print;
  }
}
```

Total

$$\mathtt{print}^{ub}(\mathtt{t}) = 2^{\mathtt{t}} * \mathrm{s}(\mathrm{SB})$$

Peak (Scope-Based GC)

$$\mathtt{print}^{ub}(\mathtt{t}) = \mathtt{t} * \mathrm{s}(\mathrm{SB})$$



Memory Consumption with Scope-based GC

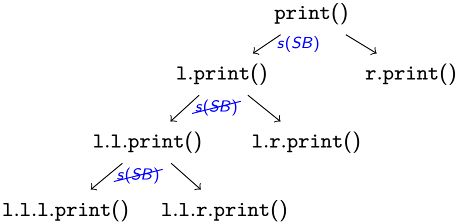
```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Total

$\texttt{print}^{ub}(\texttt{t}) = 2^{\texttt{t}}*\mathrm{s}(\mathrm{SB})$

Peak (Scope-Based GC)

$\texttt{print}^{ub}(\texttt{t}) = \texttt{t}*\mathrm{s}(\mathrm{SB})$



Memory Consumption with Scope-based GC

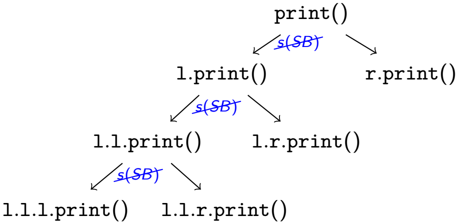
```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Total

$$\mathtt{print}^{ub}(\mathtt{t}) = 2^{\mathtt{t}} * s(SB)$$

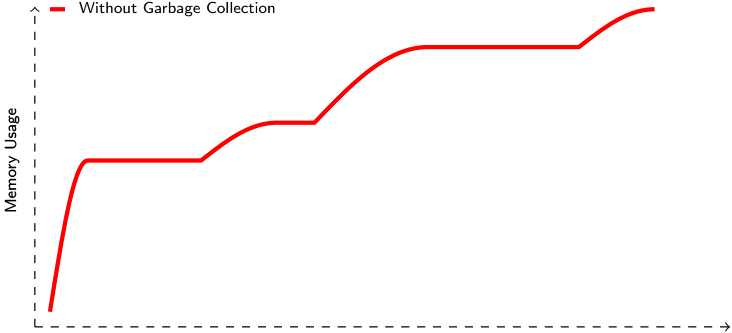
Peak (Scope-Based GC)

$$\mathtt{print}^{ub}(\mathtt{t}) = \mathtt{t} * s(SB)$$

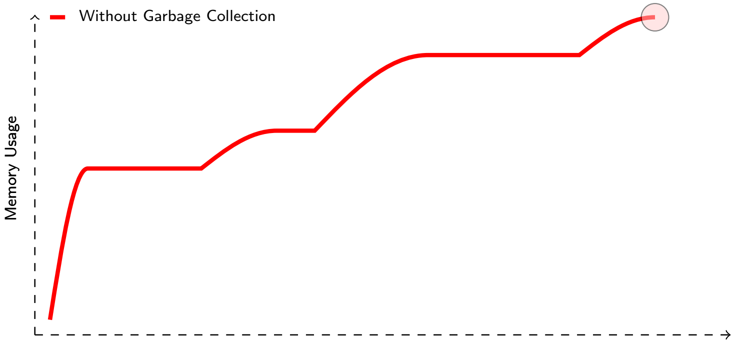


Cost Analysis - Peak Resource Consumptions

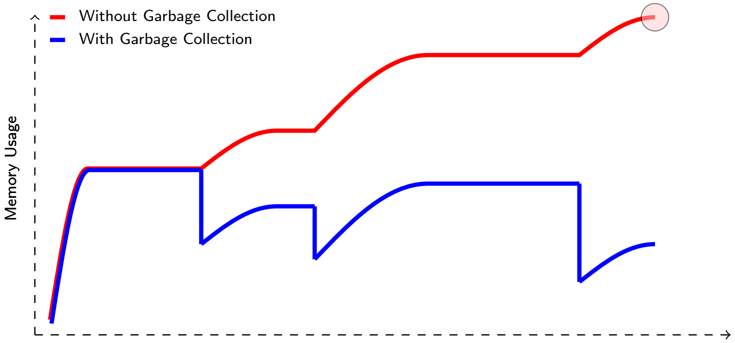
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



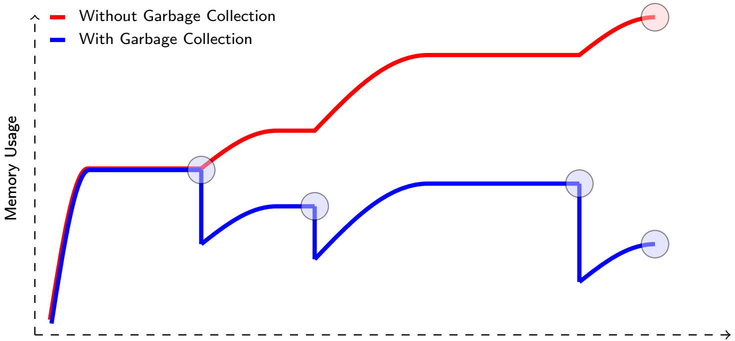
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



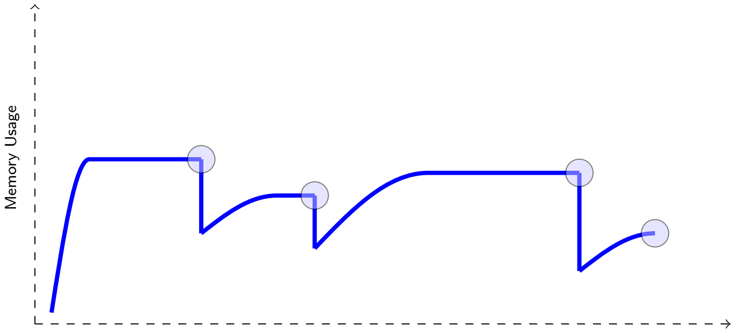
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



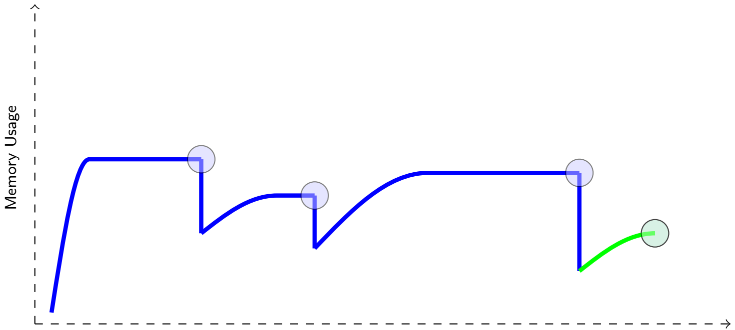
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



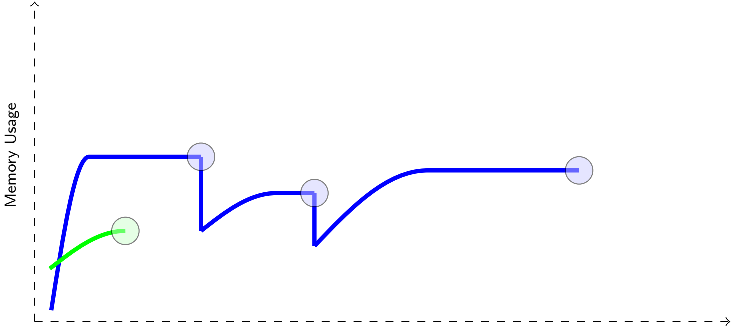
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



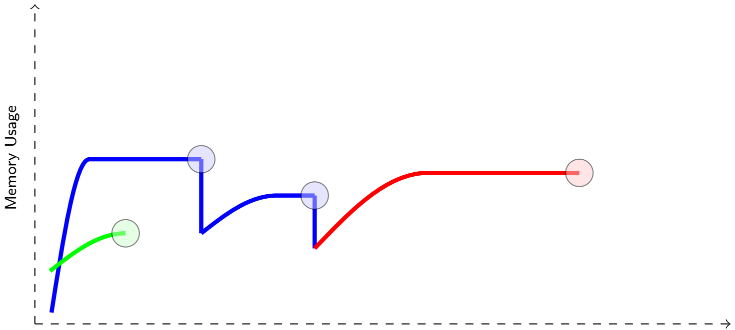
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



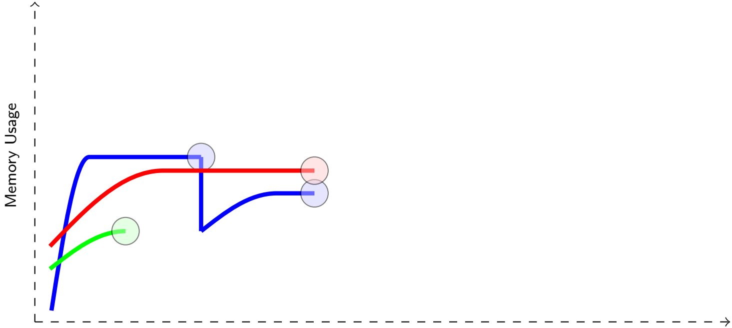
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



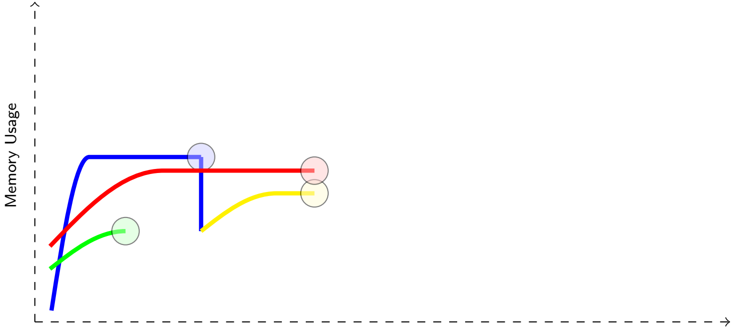
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS

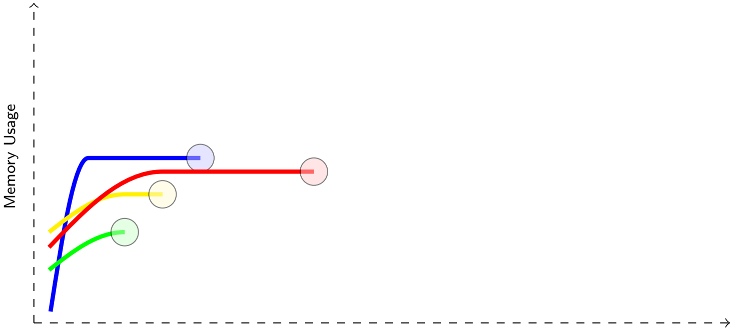


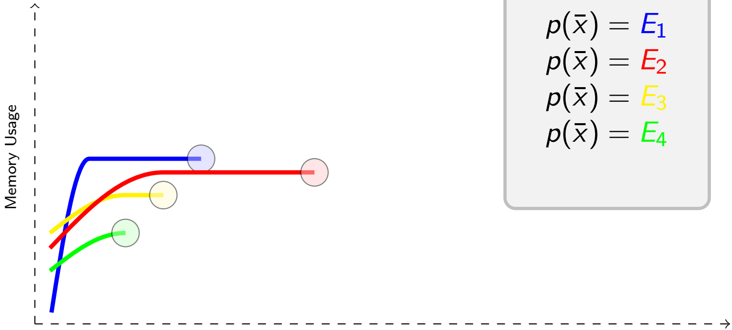
COST ANALYSIS - PEAK RESOURCE CONSUMPTIONS



Cost Analysis - Peak Resource Consumptions




Memory Usage



$$p(\bar{x}) = E_1$$
$$p(\bar{x}) = E_2$$
$$p(\bar{x}) = E_3$$
$$p(\bar{x}) = E_4$$

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$$\mathtt{print}(t) = \mathrm{s(SB)} \qquad \{t = 1\}$$
$$\mathtt{print}(t) = \mathrm{s(SB)} + 1\,\mathtt{print}(\mathrm{t}') + 2\,\mathtt{print}(\mathrm{t}'') \quad \{t>1, t>t', t>t''\}$$

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
1   // new StringBuffer(...)
    if ( left != null ) left.print;
2   if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$
$$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) + ①\texttt{print}(\mathrm{t}') + ②\texttt{print}(\mathrm{t}'') \quad \{t>1, t>t', t>t''\}$$

Upper Bound on Total: $\texttt{print}^{\mathrm{ub}}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$

$$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) + (1)\texttt{print}(\mathrm{t}') + (2)\texttt{print}(\mathrm{t}'') \quad \{t>1, t>t', t>t''\}$$

Upper Bound on Total: $\texttt{print}^{\mathrm{ub}}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

Peak (scope-based) equations:

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①   // new StringBuffer(...)
    if ( left != null ) left.print;
②   if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$

$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) + ①\texttt{print}(\mathrm{t}') + ②\texttt{print}(\mathrm{t}'') \quad \{t>1, t>t', t>t''\}$

Upper Bound on Total: $\texttt{print}^{\texttt{ub}}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

Peak (scope-based) equations:

$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
① // new StringBuffer(...)
    if ( left != null ) left.print;
② if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$$\mathtt{print}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$
$$\mathtt{print}(t) = \mathrm{s}(\mathrm{SB}) + ①\mathtt{print}(t') + ②\mathtt{print}(t'') \quad \{t>1, t>t', t>t''\}$$

Upper Bound on Total: $\mathtt{print}^{\mathrm{ub}}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

Peak (scope-based) equations:

$$\widehat{\mathtt{print}}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$
$$\widehat{\mathtt{print}}(t) = max($$

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$\mathtt{print}(t) = \mathrm{s}(\mathrm{SB}) \quad \{t = 1\}$

$\mathtt{print}(t) = \mathrm{s}(\mathrm{SB}) + ①\mathtt{print}(\mathrm{t}') + ②\mathtt{print}(\mathrm{t}'') \quad \{t>1, t>t', t>t''\}$

Upper Bound on Total: $\mathtt{print}^{\mathrm{ub}}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

Peak (scope-based) equations:

$$\hat{\mathtt{print}}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$

$$\hat{\mathtt{print}}(t) = max($$

$$\mathcal{A}(①, \mathcal{G}_s) + \hat{\mathtt{print}}(\mathrm{t}'),$$

Example (Scope-based GC)

Total Consumption Equations:

$$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$
$$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) + ①\texttt{print}(\mathrm{t}') + ②\texttt{print}(\mathrm{t}'') \quad \{t>1, t>t', t>t''\}$$

Upper Bound on Total: $\texttt{print}^{ub}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Peak (scope-based) equations:

$$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$
$$\hat{\texttt{print}}(t) = max(\mathcal{A}(①, \mathcal{G}_s) + \hat{\texttt{print}}(\mathrm{t}'), \mathcal{A}(②, \mathcal{G}_s) + \hat{\texttt{print}}(\mathrm{t}''))$$

Example (Scope-based GC)

Total Consumption Equations:

$$\texttt{print}(t) = \mathrm{s(SB)} \qquad \{t=1\}$$
$$\texttt{print}(t) = \mathrm{s(SB)} + (1)\texttt{print}(t') + (2)\texttt{print}(t'') \qquad \{t>1, t>t', t>t''\}$$

Upper Bound on Total: $\texttt{print}^{ub}(t) = 2^t * \mathrm{s(SB)}$

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
(1) // new StringBuffer(...)
    if ( left != null ) left.print;
(2) if ( right != null ) right.print;
  }
}
```

Peak (scope-based) equations:

$$\hat{\texttt{print}}(t) = \mathrm{s(SB)} \qquad \{t=1\}$$
$$\hat{\texttt{print}}(t) = max(\mathrm{s(SB)} + \hat{\texttt{print}}(t'), \mathcal{A}((2), \mathcal{G}_s) + \hat{\texttt{print}}(t''))$$

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$$\mathtt{print}(t) = \mathrm{s(SB)} \qquad \{t = 1\}$$

$$\mathtt{print}(t) = \mathrm{s(SB)} + ①\mathtt{print}(t') + ②\mathtt{print}(t'') \qquad \{t>1, t>t', t>t''\}$$

Upper Bound on Total: $\mathtt{print}^{ub}(t) = 2^t * \mathrm{s(SB)}$

Peak (scope-based) equations:

$$\hat{\mathtt{print}}(t) = \mathrm{s(SB)} \qquad \{t = 1\}$$

$$\hat{\mathtt{print}}(t) = max(\mathrm{s(SB)} + \hat{\mathtt{print}}(t'), \mathrm{s(SB)} + 2^t * \cancel{\mathrm{s(SB)}} + \hat{\mathtt{print}}(t''))$$

Example (Scope-based GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Total Consumption Equations:

$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$

$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) + ①\texttt{print}(\mathrm{t}') + ②\texttt{print}(\mathrm{t}'') \quad \{t>1, t>t', t>t''\}$

Upper Bound on Total: $\texttt{print}^{\mathrm{ub}}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

Peak (scope-based) equations:

$$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$$

$$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) + \hat{\texttt{print}}(\mathrm{t}') \qquad \{t > 1, t > t', t > t''\}$$

$$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) + \hat{\texttt{print}}(\mathrm{t}'') \qquad \{t > 1, t > t', t > t''\}$$

EXAMPLE (SCOPE-BASED GC)

```
class Tree {
  int data;
  Tree left;
  Tree right;

  void print() {
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

TOTAL CONSUMPTION EQUATIONS:

$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$

$\texttt{print}(t) = \mathrm{s}(\mathrm{SB}) + ①\texttt{print}(\mathrm{t}') + ②\texttt{print}(\mathrm{t}'') \qquad \{t>1, t>t', t>t''\}$

UPPER BOUND ON TOTAL: $\texttt{print}^{\texttt{ub}}(t) = 2^t * \mathrm{s}(\mathrm{SB})$

UPPER BOUND ON PEAK:

$\hat{\texttt{print}}^{\texttt{ub}}(t) = t * \mathrm{s}(\mathrm{SB})$

PEAK (SCOPE-BASED) EQUATIONS:

$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) \qquad \{t = 1\}$

$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) + \hat{\texttt{print}}(\mathrm{t}') \qquad \{t > 1, t > t', t > t''\}$

$\hat{\texttt{print}}(t) = \mathrm{s}(\mathrm{SB}) + \hat{\texttt{print}}(\mathrm{t}'') \qquad \{t > 1, t > t', t > t''\}$

Example (Scope-based GC)

Total Consumption Equations:

```
cl
```

$,t > t''\}$

$s(SB)$

- What does it mean that *e* is an upper bound on the *peak* heap consumption?

```
    System.out.println("data="+data);
①  // new StringBuffer(...)
    if ( left != null ) left.print;
②  if ( right != null ) right.print;
  }
}
```

Upper Bound on Peak:

$\hat{\mathtt{print}}^{ub}(t) = t * \mathrm{s(SB)}$

Peak (scope-based) equations:

$\hat{\mathtt{print}}(t) = \mathrm{s(SB)} \qquad \{t = 1\}$

$\hat{\mathtt{print}}(t) = \mathrm{s(SB)} + \hat{\mathtt{print}}(\mathrm{t}') \qquad \{t > 1, t > t', t > t''\}$

$\hat{\mathtt{print}}(t) = \mathrm{s(SB)} + \hat{\mathtt{print}}(\mathrm{t}'') \qquad \{t > 1, t > t', t > t''\}$

Example (Scope-based GC)

Total Consumption Equations:

cl

$,t>t''\}$

s(SB)

- What does it mean that e is an upper bound on the *peak* heap consumption?

```
System.out.println("data="+data);
```

1 // new StringBuffer(...)

Upper Bound on Peak:

2

B)

1. If we set the memory limit to e; *and*
2. Assume that GC will always collect unreachable objects upon method's return; *then*
3. The program will execute without running out of memory.

}

P

pr

pr $> t''\}$

pr $> t''\}$

Verification of Resource Guarantees

- Resource guarantees allow being certain that programs will run within the indicated amount of resources, but who guarantees that the inferred upper bounds are correct?

Verification of Resource Guarantees

- Resource guarantees allow being certain that programs will run within the indicated amount of resources, but who guarantees that the inferred upper bounds are correct?
- The goal is to formally verify the correctness of the upper bounds inferred by COSTA

Verification of Resource Guarantees

- Resource guarantees allow being certain that programs will run within the indicated amount of resources, but who guarantees that the inferred upper bounds are correct?
- The goal is to formally verify the correctness of the upper bounds inferred by COSTA
- Possible approaches:
  - Perform full-blown verification of COSTA
  - Formally verify the results obtained by COSTA

Verification of Resource Guarantees

- Resource guarantees allow being certain that programs will run within the indicated amount of resources, but who guarantees that the inferred upper bounds are correct?
- The goal is to formally verify the correctness of the upper bounds inferred by COSTA
- Possible approaches:
  - Perform full-blown verification of COSTA
  - Formally verify the results obtained by COSTA
- Selected alternative: construct a validating tool which, after every run of COSTA, formally confirms results are correct
  - COSTA $\Rightarrow$ Generates the upper bounds
  - KeY $\Rightarrow$ Verifies the correctness of the upper bounds

COSTA and KeY

- COSTA: Inferring Upper Bounds
  - The basic components for inferring bounds for loops
    - Ranking functions,
    - Size relations and
    - Loop invariants

COSTA and KeY

- COSTA: Inferring Upper Bounds
  - The basic components for inferring bounds for loops
    - Ranking functions,
    - Size relations and
    - Loop invariants
- KeY: Verifying Upper Bounds
  - A Development Tool Supporting Formal Analysis of Software
  - Verification Backend: *automated interactive theorem prover*
  - Java Card Dynamic Logic, and Symbolic Execution
  - JML is used for specifications

COSTA and KeY

- COSTA: Inferring Upper Bounds
  - The basic components for inferring bounds for loops
    - Ranking functions,
    - Size relations and
    - Loop invariants
- KeY: Verifying Upper Bounds
  - A Development Tool Supporting Formal Analysis of Software
  - Verification Backend: *automated interactive theorem prover*
  - Java Card Dynamic Logic, and Symbolic Execution
  - JML is used for specifications
- COSTA +KeY:
  - COSTA outputs JML annotations on the Java source
  - KeY reads annotated Java source code to verify the correctness of all JML annotations, and generates a formal proof
  - The process is fully automatic

EXAMPLE

```
int f(int n) {
  int x=0;



  while (n>0) {
    x += n;
    n--;
  }
  return x;
}
```

- COSTA infers the Upper Bound $f(n) = 7 + 7 * \mathtt{nat}(n)$

Example

```
int f(int n) {
  int x=0;



  while (n>0) {
    x += n;
    n--;
  }
  return x;
}
```

- COSTA infers the Upper Bound $f(n) = 7 + 7 * \mathtt{nat}(n)$
  - Ranking function: $\mathtt{nat}(n)$

EXAMPLE

```
int f(int n) {
  int x=0;
  //@ decreases (1*n) >= 0 ? (1*n) : 0;


  while (n>0) {
    x += n;
    n--;
  }
  return x;
}
```

- COSTA infers the Upper Bound $f(n) = 7 + 7 * \mathtt{nat}(n)$
  - Ranking function: $\mathtt{nat}(n)$

Example

```
int f(int n) {
  int x=0;
  //@ decreases (1*n) >= 0 ? (1*n) : 0;


  while (n>0) {
    x += n;
    n--;
  }
  return x;
}
```

- COSTA infers the Upper Bound $f(n) = 7 + 7 * \mathtt{nat}(n)$
  - Ranking function: $\mathtt{nat}(n)$
  - Invariant: $\{$ n'-n$\geqslant$0 $\wedge$ x-x' $\geqslant 0 \wedge \cdots \}$

Example

```
int f(int n) {
  int x=0;
  //@ decreases  (1*n) >= 0 ? (1*n) : 0;
  //@ ghost int gh_n = n; ghost int gh_x = x;

  while (n>0) {
    x += n;
    n--;
  }
  return x;
}
```

- COSTA infers the Upper Bound $f(n) = 7 + 7 * \mathtt{nat}(n)$
  - Ranking function: $\mathtt{nat}(n)$
  - Invariant: { n'-n⩾0 ∧ x-x' ⩾ 0 ∧ ⋯ }

Example

```
int f(int n) {
  int x=0;
  //@ decreases (1*n) >= 0 ? (1*n) : 0;
  //@ ghost int gh_n = n; ghost int gh_x = x;
  //@ loop_invariant gh_n-n >= 0 && x-gh_x>=0 && ··· ;
  while (n>0) {
    x += n;
    n--;
  }
  return x;
}
```

- COSTA infers the Upper Bound $f(n) = 7 + 7 * \mathtt{nat}(n)$
  - Ranking function: $\mathtt{nat}(n)$
  - Invariant: $\{$ n'-n$\geqslant$0 $\wedge$ x-x' $\geqslant 0 \wedge \cdots \}$

Example

```
int f(int n) {
   int x=0;
   //@ decreases  (1*n) >= 0 ? (1*n) : 0;
   //@ ghost int gh_n = n; ghost int gh_x = x;
   //@ loop_invariant gh_n-n >= 0 && x-gh_x>=0 && ··· ;
   while (n>0) {
      x += n;
      n--;
   }
   return x;
}
```

- COSTA infers the Upper Bound $f(n) = 7 + 7 * \mathtt{nat}(n)$
  - Ranking function: $\mathtt{nat}(n)$
  - Invariant: $\{$ n'-n$\geqslant$0 $\wedge$ x-x' $\geqslant 0 \wedge \cdots \}$
- KeY is used to verify the annotated program

TOOL DEMO

Conclusions

- Cost Analysis
  - generate cost relations from real programs
  - solve different forms of recurrence relations

Conclusions

- Cost Analysis
  - generate cost relations from real programs
  - solve different forms of recurrence relations
- COSTA – COSt and Termination Analyzer for Java bytecode

Conclusions

- Cost Analysis
  - generate cost relations from real programs
  - solve different forms of recurrence relations
- COSTA – COSt and Termination Analyzer for Java bytecode
  - PUBS (the CRs solver) made the difference
    - based on program analysis techniques
    - widely applicable
    - can use CAS to improve precision

Conclusions

- Cost Analysis
  - generate cost relations from real programs
  - solve different forms of recurrence relations
- COSTA – COSt and Termination Analyzer for Java bytecode
  - PUBS (the CRs solver) made the difference
    - based on program analysis techniques
    - widely applicable
    - can use CAS to improve precision
  - non-cumulative resource: garbage collection, X10 task-level

Conclusions

- Cost Analysis
  - generate cost relations from real programs
  - solve different forms of recurrence relations
- COSTA – COSt and Termination Analyzer for Java bytecode
  - PUBS (the CRs solver) made the difference
    - based on program analysis techniques
    - widely applicable
    - can use CAS to improve precision
  - non-cumulative resource: garbage collection, X10 task-level
  - COSTABS – A recent extension for CONCURRENT OBJECTS, a concurrency model which is simpler than Java multithreading

Conclusions

- Cost Analysis
  - generate cost relations from real programs
  - solve different forms of recurrence relations
- COSTA – COSt and Termination Analyzer for Java bytecode
  - PUBS (the CRs solver) made the difference
    - based on program analysis techniques
    - widely applicable
    - can use CAS to improve precision
  - non-cumulative resource: garbage collection, X10 task-level
  - COSTABS – A recent extension for CONCURRENT OBJECTS, a concurrency model which is simpler than Java multithreading
  - Eclipse Plugin, web-interface, and command-line

Conclusions

- Cost Analysis
  - generate cost relations from real programs
  - solve different forms of recurrence relations
- COSTA – COSt and Termination Analyzer for Java bytecode
  - PUBS (the CRs solver) made the difference
    - based on program analysis techniques
    - widely applicable
    - can use CAS to improve precision
  - non-cumulative resource: garbage collection, X10 task-level
  - COSTABS – A recent extension for CONCURRENT OBJECTS, a concurrency model which is simpler than Java multithreading
  - Eclipse Plugin, web-interface, and command-line
- Future Directions
  - handle Java multithreaded programs
  - handle cyclic data structures
  - average case, and distribution

Credits - the COSTA team

http://costa.ls.fi.upm.es

Elvira Albert
Diego Alonso
Puri Arenas
Jesús Correas
Antonio Flores
Samir Genaim
Miguel Gómez-Zamalloa
Abu Naser Masud
Jose Miguel Rojas
Germán Puebla
Diana Ramírez
Guillermo Román
Damiano Zanardini