Automated Termination Proofs for Java Bytecode with Cyclic Data

M. Brockschmidt, R. Musiol, C. Otto, J. Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

WST 2012, Obergurgl
Termination Analysis for Imperative Programs
Synthesis of Linear Ranking Functions

(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...
Termination Analysis for Imperative Programs

- Synthesis of Linear Ranking Functions
  \textit{(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), \ldots}

- Terminator
  Termination Analysis by Abstraction & Model Checking
  \textit{(Cook, Podelski, Rybalchenko et al., since 05)}
Termination Analysis for Imperative Programs

- Synthesis of Linear Ranking Functions
  \textit{(Colon & Sipma, 01), (Podelski & Rybalchenko, 04), ...}

- Terminator
  Termination Analysis by Abstraction & Model Checking
  \textit{(Cook, Podelski, Rybalchenko et al., since 05)}

- Julia & COSTA
  Termination Analysis of \texttt{JAVA Bytecode} (JBC)
  Fixed abstraction, via Constraint Logic Programs
  \textit{(Spoto, Mesnard, Payet, 10)}
  \textit{(Albert, Arenas, Codish, Genaim, Puebla, Zanardini, 08)}
Rewriting-based approach: Structure

- Programming languages hard ⇐ Simpler representation needed
Rewriting-based approach: Structure

- Programming languages *hard* $\rightsquigarrow$ Simpler representation needed
- Termination Graphs: Simple, all information

Diagram:

- Haskell
- Prolog
- Java
- C

Termination Graph
Rewriting-based approach: Structure

- Programming languages hard $\leadsto$ Simpler representation needed
- Termination Graphs: Simple, all information
- Term Rewrite Systems (TRSs) generated from Termination Graph

Diagram:

- Haskell
- Prolog
- Java
- C
- Termination Graph
- TRS
Rewriting-based approach: Structure

- Programming languages *hard* \(\sim\) Simpler representation needed
- Termination Graphs: Simple, all information
- Term Rewrite Systems (TRSs) generated from Termination Graph
- Prove TRS termination using existing provers

Diagram:

```
Haskell -> Termination Graph -> TRS

Prolog -> Termination Graph

Java -> Termination Graph

C -> Termination Graph
```
Rewriting-based approach: Advantages

Handling of user-defined data structures:

```java
public class List {
    int value;
    List next;
}
```
Rewriting-based approach: Advantages

Handling of user-defined data structures:

- **Other techniques:**
  - Fixed abstraction to **number**
  - List [2, 4, 6] abstracted to **length 3**
Rewriting-based approach: Advantages

Handling of user-defined data structures:

- **Other techniques:**
  - Fixed abstraction to **number**
  - List [2, 4, 6] abstracted to **length 3**
- **Our technique:**
  - Abstraction to **terms**
  - List [2, 4, 6] becomes
    List(2, List(4, List(6, null)))

```java
class List {
    int value;
    List next;
}
```
Rewriting-based approach: Advantages

Handling of user-defined data structures:

- **Other techniques:**
  - **Fixed** abstraction to **number**
  - List [2, 4, 6] abstracted to **length 3**

- **Our technique:**
  - Abstraction to **terms**
  - List [2, 4, 6] becomes
    - List(2, List(4, List(6, null)))

- **TRS techniques** search for suitable orders automatically
  ⇒ Complex orders for user-defined data structures possible
Handling of user-defined cyclic data structures:

- **Our technique:**
  Abstraction to **terms** impossible
Rewriting-based approach: Challenges

Handling of user-defined cyclic data structures:

- **Our technique:**
  - Abstraction to **terms** impossible
  - List [2, 4, 6, 2, 4, 6, …] is abstracted to free variable
  - Suitable order cannot be found

```java
public class List {
    int value;
    List next;
}
```
Rewriting-based approach: Challenges

Handling of user-defined *cyclic* data structures:

- **Our technique:**
  Abstraction to **terms** impossible
- List [2, 4, 6, 2, 4, 6, ...] is abstracted to free variable
  - Suitable order cannot be found

- **Solution:**
  1. Find suitable measures on Termination Graph level
  2. Encode (numeric) measures into TRS
1. Introduction

2. Marking traversal algorithms

3. Definite Cyclicity

4. Conclusion
visit the example

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }}}
visit the example

class L {
    int v;   List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }}}

1 Store first v
2 Continue if obj. unvisited
3 Change v
4 Go to next element
visit the example

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element
class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element
visit the example

class L {
    int v; List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        } 
    }
}

1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element
visit the example

class L {
    int v;    List n;
static void visit(L x) {
    int e = x.v;
while (x.v == e) {
    x.v = e + 1;
    x = x.n;
}}

1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element
visit the example

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element
visit the example

class L {
    int v;   List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

1. Store first v
2. Continue if obj. unvisited
3. Change v
4. Go to next element
class L {
    int v; List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

00: aload_0 #load x
01: getfield v #get v from x
04: istore_1 #store to e
05: aload_0 #load x
06: getfield v #get v from x
09: iload_1 #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0 #load x
14: iload_1 #load e
15: iconst_1 #load 1
16: iadd #add e and 1
17: putfield v #store to x.v
20: aload_0 #load x
21: getfield n #get n from x
24: astore_0 #store to x
25: goto 5 #store to x
28: return
Abstract Java virtual machine states

class L {
    int v; List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

00: aload_0  #load x
01: getfield v  #get v from x
04: istore_1  #store to e
05: aload_0  #load x
06: getfield v  #get v from x
09: iload_1  #load e
10: if_icmpne 28  #jump if x.v != e
13: aload_0  #load x
14: iload_1  #load e
15: iconst_1  #load 1
16: iadd  #add e and 1
17: putfield v  #store to x.v
20: aload_0  #load x
21: getfield n  #get n from x
24: astore_0  #store to x
25: goto 5
28: return
Abstract JAVA virtual machine states

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction

00: aload_0    #load x
01: getfield v  #get v from x
04: istore_1    #store to e
05: aload_0    #load x
06: getfield v  #get v from x
09: iload_1     #load e
10: if_icmpne 28 #jump if x.v != e
13: aload_0    #load x
14: iload_1     #load e
15:  iconst_1    #load 1
16:  iadd        #add e and 1
17:  putfield v  #store to x.v
20:  aload_0    #load x
21:  getfield n  #get n from x
24:  astore_0    #store to x
25:  goto 5
28:  return
Abstract **JAV**A virtual machine states

class L {
    int v;  List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack
class L {
    int v; List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack

Heap information:
Abstract Java virtual machine states

class L {
    int v; List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack

Heap information:
- At $o_1$ is L object or null
Abstract Java virtual machine states

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack

Heap information:
- At $o_1$ is L object or null
- At $i_1$ is unknown integer

<table>
<thead>
<tr>
<th>05</th>
<th>x: $o_1$, e: $i_1$</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$: L(?)</td>
<td>$i_1$: Z</td>
<td>$o_1$</td>
</tr>
</tbody>
</table>
Abstract Java virtual machine states

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack

Heap information:
- At $o_1$ is L object or null
- At $i_1$ is unknown integer
- Known L object: $o_2 : L(v = i_2, n = o_3)$
Abstract Java virtual machine states

class L {
    int v; List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack

Heap information:
- At $o_1$ is L object or null
- At $i_1$ is unknown integer
- Known L object: $o_2 : L(v = i_2, n = o_3)$

Only explicit sharing
Abstract JAVA virtual machine states

class L {
    int v; List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack

Heap information:
- At $o_1$ is L object or null
- At $i_1$ is unknown integer
- Known L object: $o_2 : \text{L}(v = i_2, n = o_3)$

Heap annotations: Only explicit sharing
- Reference might be cyclic: $o_1\checkmark$
Abstract **Java** virtual machine states

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;
        }
    }
}

<table>
<thead>
<tr>
<th>05</th>
<th>x: o₁, e: i₁</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₁: L(?)</td>
<td>i₁: Z</td>
<td>o₁○</td>
</tr>
</tbody>
</table>

**Stack frame:**
- Next program instruction
- Local variables
- Operand stack

**Heap information:**
- At o₁ is L object or null
- At i₁ is unknown integer
- Known L object: o₂ : L(v = i₂, n = o₃)

**Heap annotations:** **Only explicit sharing**
- Reference might be cyclic: o₁○
- Two references may be equal: o₁ =? o₂
Abstract JAVA virtual machine states

class L {
    int v;    List n;
    static void visit(L x) {
        int e = x.v;
        while (x.v == e) {
            x.v = e + 1;
            x = x.n;  }
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack

Heap information:
- At $o_1$ is L object or null
- At $i_1$ is unknown integer
- Known L object: $o_2 : L(v = i_2, n = o_3)$

Heap annotations: Only explicit sharing
- Reference might be cyclic: $o_1 \ominus$
- Two references may be equal: $o_1 =? o_2$
- Two references may share: $o_1 \forall o_2$
State A:

- x some (possibly cyclic) list
- e some integer

static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
State B:
- **Evaluation** between A and B
- Need field of \( o_1 \)

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
States $B$, $C$, $D$:

- **Evaluation** between $A$ and $B$
- Need field of $o_1$ ⇒ **Refinement**:
  - In $C$: $o_1$ is null
  - In $D$: $o_1$ renamed to $o_2$, pointing to L-object with successor $o_3$:
    - $o_3$ possibly cyclic
    - $o_3$ possibly equal to $o_2$ and may reach $o_2$
States $B$, $C$, $D$:

- **Evaluation** between $A$ and $B$
- **Need field of** $o_1$ $\Rightarrow$ **Refinement**:
  - In $C$: $o_1$ is null (program crashes)
  - In $D$: $o_1$ renamed to $o_2$, pointing to L-object with successor $o_3$:
    - $o_3$ possibly cyclic
    - $o_3$ possibly equal to $o_2$ and may reach $o_2$

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
States $E$, $F$:

- Need to read field of $o_2$ ⇒ Refinement
  - In $E$: $o_2 \neq o_3$
  - In $F$: $o_2 = o_3$

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
State G:
- Evaluation: Read v, loaded e
- Need to decide $i_1 \neq i_2$

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
States $G$, $I$, $H$:

- Evaluation: Read $v$, loaded $e$
- Need to decide $i_1 \neq i_2$ ⇒ Refinement:
  - In $I$: $i_1 = i_2$
  - In $H$: $i_1 \neq i_2$

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
States $G$, $I$, $H$:  
- Evaluation: Read $v$, loaded $e$  
- Need to decide $i_1 \neq i_2 \Rightarrow$ Refinement:  
  - In $I$: $i_1 = i_2$ (program ends)  
  - In $H$: $i_1 \neq i_2$

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
States \( G, I, H \):
- Evaluation: Read \( v \), loaded \( e \)
- Need to decide \( i_1 \neq i_2 \) ⇒ Refinement:
  - \( i_1 = i_2 \) (program ends)
  - \( i_1 \neq i_2 \)
- State \( J \) reached by evaluation

```
static void visit(L x) {
  int e = x.v;
  while (x.v == e) {
    x.v = e + 1;
    x = x.n;
  }
}
```
States $G$, $I$, $H$:

- Evaluation: Read $v$, loaded $e$
- Need to decide $i_1 \neq i_2 \Rightarrow$ Refinement:
  - In $I$: $i_1 = i_2$ (program ends)
  - In $H$: $i_1 \neq i_2$
- State $J$ reached by evaluation, represented by (instance of) $A$

static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
States $K$, $L$: Analogous for one-element list

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
- All leaves program ends ⇒ Graph finished
- How can we prove termination?

static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
• All leaves program ends ⇒ Graph finished
• How can we prove termination?
• Only consider SCCs

static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
- All leaves program ends \(\Rightarrow\) Graph finished
- How can we prove termination?
- Only consider SCCs

High-level argument: Number of unvisited elements strictly decreasing

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
All leaves program ends ⇒ Graph finished

How can we prove termination?

Only consider SCCs

High-level argument: Number of unvisited elements strictly decreasing

... Let’s drag that down to our level!

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
Q: What is an “unvisited element”, formally?

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
Q: What is an “unvisited element”, formally?
A: One with $L.v = i_1 = e$
Q: What is an “unvisited element”, formally?

A: One with $L.v = i_1 = e$

- Automatically finding this relation:
  - Identify constant $c$ in SCC

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
Q: What is an “unvisited element”, formally?
A: One with \( L.v = i_1 = e \)
   - Automatically finding this relation:
     1. Identify constant \( c \) in SCC
     2. Search property \( M = C.f \bowtie C \) checked on all cycles

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
Q: What is an “unvisited element”, formally?

A: One with $L.v = i_1 = e$

- Automatically finding this relation:
  1. Identify constant $c$ in SCC
  2. Search property $M = C.f \triangleleft c$ checked on all cycles
- Track number of objects where $C.f \triangleleft c$ holds ($\#_M$)

```java
static void visit(L x) {
    int e = x.v;
    while (x.v == e) {
        x.v = e + 1;
        x = x.n;
    }
}
```
Property $M = C.f \bowtie c$ (here: $c = i_1$). When does $\#_M$ change?
Property $M = C.f \otimes c$ (here: $c = i_1$). When does $\#_M$ change?

- $C.f$ written (old value $u$, new value $w$):
Property $M = C.f \otimes c$ (here: $c = i_1$). When does $\#M$ change?

- **C.f written** (old value $u$, new value $w$):
  - $u \otimes c \land \neg w \otimes c$ tautology $\Rightarrow \#M$ decremented by 1
Property \( M = C.f \otimes c \) (here: \( c = i_1 \)). When does \( \#_M \) change?

- \( C.f \) written (old value \( u \), new value \( w \)):
  - \( u \otimes c \land \neg w \otimes c \) tautology \( \Rightarrow \#_M \) decremented by 1
  - \( u \otimes c \leftrightarrow w \otimes c \) tautology \( \Rightarrow \#_M \) unchanged
Property \( M = C.f \times c \) (here: \( c = i_1 \)). When does \( \#_M \) change?

- \( C.f \) written (old value \( u \), new value \( w \)):
  - \( u \times c \land \neg w \times c \) tautology ⇒ \( \#_M \) decremented by 1
  - \( u \times c \leftrightarrow w \times c \) tautology ⇒ \( \#_M \) unchanged
  - Otherwise: \( \#_M \) incremented by 1.
Property $M = C.f \Join c$ (here: $c = i_1$). When does $\#_M$ change?

- **$C.f$ written** (old value $u$, new value $w$):
  - $u \Join c \land \neg w \Join c$ tautology $\Rightarrow \#_M$ decremented by 1
  - $u \Join c \leftrightarrow w \Join c$ tautology $\Rightarrow \#_M$ unchanged
  - Otherwise: $\#_M$ incremented by 1.

In example: $l \rightarrow j$: $i_1$ old, $i_3$ new

$\Rightarrow i_1 = i_1 \land \neg i_3 = i_1$
Property $M = C.f \otimes c$ (here: $c = i_1$). When does $\#M$ change?

- **C.f written** (old value $u$, new value $w$):
  - $u \otimes c \land \neg w \otimes c$ tautology $\Rightarrow \#M$ decremented by 1
  - $u \otimes c \leftrightarrow w \otimes c$ tautology $\Rightarrow \#M$ unchanged
  - Otherwise: $\#M$ incremented by 1.

In example: $I \rightarrow J$: $i_1$ old, $i_3$ new

$$i_1 = i_2 \land i_3 = i_1 + 1 \rightarrow i_1 = i_1 \land \neg i_3 = i_1$$
Property $M = C.f \bowtie c$ (here: $c = i_1$). When does $\#_M$ change?

- $C.f$ written (old value $u$, new value $w$):
  - $u \bowtie c \land \neg w \bowtie c$ tautology $\Rightarrow$ $\#_M$ decremented by 1
  - $u \bowtie c \leftrightarrow w \bowtie c$ tautology $\Rightarrow$ $\#_M$ unchanged
  - Otherwise: $\#_M$ incremented by 1.

In example: $F \rightarrow K$: $i_1$ old, $i_4$ new

$\Rightarrow i_1 = i_2 \land i_4 = i_1 + 1 \rightarrow i_1 = i_1 \land \neg i_4 = i_1$
Property $M = C.f \vartriangleleft c$ (here: $c = i_1$). When does $#M$ change?

- **C.f written** (old value $u$, new value $w$):
  - $u \vartriangleleft c \land \neg w \vartriangleleft c$ tautology $\Rightarrow$ $#M$ decremented by 1
  - $u \vartriangleleft c \Leftrightarrow w \vartriangleleft c$ tautology $\Rightarrow$ $#M$ unchanged
  - Otherwise: $#M$ incremented by 1.

- New L object is created: Same for default value
- Add variable for counter to states, changes to edges
- Require counter > 0 at checks
Add variable for counter to states, changes to edges

Require counter > 0 at checks

Termination proof via TRS now trivial:

\[ f_A(\ldots, i_5) \Rightarrow f_I(\ldots, i_5) \quad | \quad i_5 > 0 \]
\[ f_I(\ldots, i_5) \Rightarrow f_J(\ldots, i_5 - 1) \]
\[ f_J(\ldots, i_6) \Rightarrow f_A(\ldots, i_6) \]
\[ f_A(\ldots, i_5) \Rightarrow f_F(\ldots, i_5) \]
\[ f_F(\ldots, i_5) \Rightarrow f_K(\ldots, i_5 - 1) \quad | \quad i_5 > 0 \]
\[ f_F(\ldots, i_7) \Rightarrow f_A(\ldots, i_7) \]
Overview

1. Introduction

2. Marking traversal algorithms

3. Definite Cyclicity

4. Conclusion
The iterate example

1. Keep first element in this
2. Iterate until reaching it again

```
class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}
```
The iterate example

00: aload_0  #load this
01: getfield n  #get n from this
04: astore_1  #store to x
05: aload_1  #load x
06: aload_0  #load this
07: if_acmpeq 18  #jump if x == this
10: aload_1  #load x
11: getfield n  #get n from x
14: astore_1  #store x
15: goto 05
18: return

class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}

1  Keep first element in this
2  Iterate until reaching it again
The iterate example

00: aload_0    #load this
01: getfield n #get n from this
04: astore_1   #store to x
05: aload_1    #load x
06: aload_0    #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1    #load x
11: getfield n #get n from x
14: astore_1   #store x
15: goto 05
18: return

class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}

1. Keep first element in this
2. Iterate until reaching it again
The iterate example

class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}

1. Keep first element in this
2. Iterate until reaching it again
The iterate example

00: aload_0      #load this
01: getfield n   #get n from this
04: astore_1     #store to x
05: aload_1      #load x
06: aload_0      #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1      #load x
11: getfield n   #get n from x
14: astore_1     #store x
15: goto 05
18: return

class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}

1. Keep first element in this
2. Iterate until reaching it again
The iterate example

```
class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}
```

1. Keep first element in this
2. Iterate until reaching it again
Definite Reachability

```java
class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}
```

1. Keep first element in this
2. Iterate until reaching it again

New annotation: Define reachability \( F \rightarrow ! \)

\[ o \xrightarrow{F}! o' \implies \]

All paths from \( o \) using fields from \( F \) reach \( o' \)
Definite Reachability

00: aload_0  #load this
01: getfield n  #get n from this
04: astore_1  #store to x
05: aload_1  #load x
06: aload_0  #load this
07: if_acmpeq 18  #jump if x == this
10: aload_1  #load x
11: getfield n  #get n from x
14: astore_1  #store x
15: goto 05
18: return

class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}

1. Keep first element in this
2. Iterate until reaching it again

New annotation: Definite reachability $\overset{F}{\rightarrow}$!

- $o \overset{F}{\rightarrow} o' \Rightarrow$ All paths from $o$ using fields from $F$ reach $o'$

- $=\exists, \forall, \O$ extending annotations:
  Allow (not enforce) sharing/shapes
Definite Reachability

00: aload_0  #load this
01: getfield n #get n from this
04: astore_1  #store to x
05: aload_1   #load x
06: aload_0   #load this
07: if_acmpeq 18 #jump if x == this
10: aload_1   #load x
11: getfield n #get n from x
14: astore_1  #store x
15: goto 05
18: return

class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}

1 Keep first element in this
2 Iterate until reaching it again

New annotation: Definite reachability $\vdash F$!

- $o \vdash F \Rightarrow o'$
  
  All paths from $o$ using fields from $F$ reach $o'$

- \textit{extending} annotations:
  
  $\vdash$, $\bigvee$, $\emptyset$

- \textit{Allow} (not enforce) sharing/shapes

- \textit{restricting} annotation:
  
  $\vdash F$ Enforce sharing
Definite Reachability

```java
class L {
    L n;
    void iterate() {
        L x = this.n;
        while (x != this) {
            x = x.n;
        }
    }
}
```

New annotation: Definite reachability $\overset{F}{\Rightarrow}$!

- $o \overset{F}{\Rightarrow} o'$ \Rightarrow
  All paths from $o$ using fields from $F$ reach $o'$

- $=\otimes, \sqcap, \circ$ extending annotations:
  * Allow (not enforce) sharing/shapes

- $\overset{F}{\Rightarrow}$ restricting annotation:
  * Enforce sharing

1. Keep first element in this
2. Iterate until reaching it again
State A:
- t some definitely cyclic list
- x second element in list

```java
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
```
**State A:**
- t some definitely cyclic list
- x second element in list

**State B:**
- First equals second element?

```java
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
State A:
- t some definitely cyclic list
- x second element in list

State B:
- First equals second element?

⇒ Refinement
- In C: References equal ((program ends)
- In D: References not equal

```
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
```
State $E$:
- Access to unknown object $o_2$

```java
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
```
States $E, F$:

- Access to unknown object $o_2$  
  $\Rightarrow$ Refinement

- Case $o_2 = \text{null}$ not possible (implies $o_2$ not reaching $o_1$)

void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: astore_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return

---

**State G:**
- **Same program position as A ⇒ Instantiate**
  - In A: this = o₁ \(\xrightarrow{n}o₂ = x\)
  - In G: this = o₁ \(\xrightarrow{n}o₃ \xrightarrow{n}o₄ = x\)

```java
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
```
00: aload_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return

States $G$, $H$:
- Same program position as $A \Rightarrow$ Instantiate
  - In $A$: $\text{this} = o_1 \xrightarrow{n} o_2 = x$
  - In $G$: $\text{this} = o_1 \xrightarrow{n} o_3 \xrightarrow{n} o_4 = x$
  $\Rightarrow$ In $H$: Abstract to $\text{this} = o_1 \{n\}! o_4 = x$

void iterate() {
  L x = this.n;
  while (x != this) {
    x = x.n;
  }
}
States $G, H$:

- Same program position as $A \Rightarrow$ Instantiate
  
  In $A$: $\text{this} = o_1 \xrightarrow{n} o_2 = x$
  
  In $G$: $\text{this} = o_1 \xrightarrow{n} o_3 \xrightarrow{n} o_4 = x$

  $\Rightarrow$ In $H$: Abstract to $\text{this} = o_1 \xrightarrow{\{n\}!} o_4 = x$

- Restart construction from more general state

```java
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
```
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}

States $G, H$:
- Same program position as $A \Rightarrow$ Instantiate
  - In $A$: this = $o_1$ $\xrightarrow{n} o_2 = x$
  - In $G$: this = $o_1$ $\xrightarrow{n} o_3$ $\xrightarrow{n} o_4 = x$
  - $\Rightarrow$ In $H$: Abstract to this = $o_1$ $\{n}\!\Rightarrow o_4 = x$

- Restart construction from more general state

States $I, J, K, L$: As before
Proving termination with $R = o \xrightarrow{F} o'$:

1. Associate length $\ell_R$ with each $R$

```java
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
void iterate() {
  L x = this.n;
  while (x != this) {
    x = x.n;
  }
}

Proving termination with $R = o \xrightarrow{F}^{!} o'$:

1. Associate length $\ell_R$ with each $R$
2. In refinements:
   - $o = ? o'$ resolved to $o \neq o'$: $\ell_R > 0$

---

**Diagram Notes:**
- **A**: $05: t: o_1, x: o_2 \mid \epsilon$
  - $o_1: \text{L}(n = o_2) \quad o_2: \text{L}(?)$
  - $o_1, o_2 \triangleright o_1 = ? o_2$
  - $o_1 \land o_2 \quad o_2 \rightarrow o_1$
- **H**: $11: t: o_1, x: o_5 \mid o_5$
  - $o_1: \text{L}(?) \quad o_5: \text{L}(n = o_6)$
  - $o_1, o_5 \triangleright o_1 \quad o_5 \triangleright o_1$
  - $o_6 = ? o_1 \quad o_6 \triangleright o_1$
  - $o_6 \rightarrow o_5 \quad o_6 \rightarrow o_1$
- **I**: $07: t: o_1, x: o_4 \mid o_1, o_4$
  - $o_1: \text{L}(?) \quad o_1, o_4 \triangleright o_4 = ? o_1$
  - $o_1 \land o_4 \quad o_4 \rightarrow o_1$
- **J**: $07: t: o_1, x: o_4 \mid o_1, o_4$
  - $o_1: \text{L}(?) \quad o_1, o_4 \triangleright o_4 = ? o_1$
  - $o_1 \land o_4 \quad o_4 \rightarrow o_1$
- **K**: $11: t: o_1, x: o_4 \mid o_4$
  - $o_1: \text{L}(?) \quad o_4: \text{L}(?)$
  - $o_1, o_4 \triangleright o_1 \quad o_4 \rightarrow o_1$
  - $o_1 \land o_4 \quad o_4 \rightarrow o_1$
00: astore_0
01: getfield n
04: astore_1
05: aload_1
06: aload_0
07: if_acmpeq 18
10: aload_1
11: getfield n
14: astore_1
15: goto 05
18: return

Proving termination with $R = o \xrightarrow{F} ! o'$:

1. Associate length $\ell_R$ with each $R$
2. In refinements:
   - $o = ? o'$ resolved to $o \neq o'$: $\ell_R > 0$
   - $\tilde{o}$ new $F$-child of $o$: $\ell_{R'} = \ell_R - 1$ (for $R' = \tilde{o} \xrightarrow{F} ! o'$)

```
void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
```
Proving termination with $R = o \xrightarrow{F} o'$:

1. Associate length $\ell_R$ with each $R$
2. In refinements:
   - $o =? o'$ resolved to $o \neq o'$: $\ell_R > 0$
   - $\tilde{o}$ new $F$-child of $o$: $\ell_{R'} = \ell_R - 1$ (for $R' = \tilde{o} \xrightarrow{F} o'$)
3. Add variable for lengths to graphs (here: only done for $\ell_{o_4 \{n\} ! o_1}$)

void iterate() {
    L x = this.n;
    while (x != this) {
        x = x.n;
    }
}
Proving termination with $R = o \xrightarrow{F} o'$:

1. **Associate length $\ell_R$ with each $R$**

2. **In refinements:**
   - $o = ? o'$ resolved to $o \neq o'$: $\ell_R > 0$
   - \( ? \) new $F$-child of $o$: $\ell_{R'} = \ell_R - 1$ (for $R' = ? \xrightarrow{F} o'$)

3. **Add variable for lengths to graphs (here: only done for $\ell o_4 {n}!o_1$)**

Resulting TRS:

$$f(\ldots, \ell o_4 {n}!o_1) \rightarrow f(\ldots, \ell o_4 {n}!o_1 - 1) \mid \ell o_4 {n}!o_1 > 0$$
Automated Termination Proofs for Java Bytecode with Cyclic Data

- Implemented in AProVE for full single-threaded Java
Automated Termination Proofs for Java Bytecode with Cyclic Data

- Implemented in AProVE for full single-threaded Java
- Evaluated on collection of 387 programs (including the *Termination Problem Data Base*):

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>N</th>
<th>F</th>
<th>T</th>
<th>R</th>
<th>Y</th>
<th>N</th>
<th>F</th>
<th>T</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>267</td>
<td>81</td>
<td>11</td>
<td>28</td>
<td>9.5</td>
<td>51</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>15.8</td>
</tr>
<tr>
<td>AProVE ’11</td>
<td>225</td>
<td>81</td>
<td>45</td>
<td>36</td>
<td>11.4</td>
<td>23</td>
<td>0</td>
<td>29</td>
<td>8</td>
<td>18.3</td>
</tr>
<tr>
<td>Julia</td>
<td>191</td>
<td>22</td>
<td>174</td>
<td>0</td>
<td>4.7</td>
<td>32</td>
<td>0</td>
<td>28</td>
<td>0</td>
<td>8.2</td>
</tr>
<tr>
<td>COSTA</td>
<td>160</td>
<td>0</td>
<td>181</td>
<td>46</td>
<td>11.0</td>
<td>29</td>
<td>0</td>
<td>5</td>
<td>26</td>
<td>30.4</td>
</tr>
</tbody>
</table>

- all examples
- LinkedList + HashMap
Automated Termination Proofs for Java Bytecode with Cyclic Data

- Implemented in AProVE for full single-threaded Java
- Evaluated on collection of 387 programs (including the Termination Problem Data Base):

<table>
<thead>
<tr>
<th></th>
<th>Y</th>
<th>N</th>
<th>F</th>
<th>T</th>
<th>R</th>
<th></th>
<th>Y</th>
<th>N</th>
<th>F</th>
<th>T</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>267</td>
<td>81</td>
<td>11</td>
<td>28</td>
<td>9.5</td>
<td>AProVE</td>
<td>51</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>15.8</td>
</tr>
<tr>
<td>AProVE '11</td>
<td>225</td>
<td>81</td>
<td>45</td>
<td>36</td>
<td>11.4</td>
<td>23</td>
<td>0</td>
<td>29</td>
<td>8</td>
<td>18.3</td>
<td></td>
</tr>
<tr>
<td>Julia</td>
<td>191</td>
<td>22</td>
<td>174</td>
<td>0</td>
<td>4.7</td>
<td>32</td>
<td>0</td>
<td>28</td>
<td>0</td>
<td>8.2</td>
<td></td>
</tr>
<tr>
<td>COSTA</td>
<td>160</td>
<td>0</td>
<td>181</td>
<td>46</td>
<td>11.0</td>
<td>29</td>
<td>0</td>
<td>5</td>
<td>26</td>
<td>30.4</td>
<td></td>
</tr>
</tbody>
</table>

- Termination depending on cyclic data requires early abstraction

- all examples

- LinkedList + HashMap