Termination Analysis for Imperative Programs Operating on the Heap

Marc Brockschmidt

November 2013
Termination! What is it good for?
Termination! What is it good for?

1 Program: produces result
Termination! What is it good for?

1. **Program**: produces result

2. **Input handler**: system reacts

Variations of the same problem:
- Special case of the program can be interpreted as the input handler.
- Probabilistic version of the program.
Termination! What is it good for?

1. **Program**: produces result
2. **Input handler**: system reacts
3. **Mathematical proofs**: induction is valid
Termination! What is it good for?

1. Program: produces result
2. Input handler: system reacts
3. Mathematical proofs: induction is valid
4. Biological process: reaches stable state
Termination! What is it good for?

1. **Program**: produces result
2. **Input handler**: system reacts
3. **Mathematical proofs**: induction is valid
4. **Biological process**: reaches stable state

Variations of same problem:

- 2 special case of 1
- 3 can be interpreted as 1
- 4 probabilistic version of 1
“Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.”

1. Find rank function \( f \) ("quantity")
“Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.”

1. Find rank function $f$ (“quantity”)
2. Prove $f$ to have a lower bound (“vanishes when the machine stops”)

Turing 1949
“Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.”

1. Find rank function \( f \) (“quantity”)
2. Prove \( f \) to have a lower bound (“vanishes when the machine stops”)
3. Prove \( f \) to decrease over time
“Finally the checker has to verify that the process comes to an end. [...] This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.”

1. Find rank function $f$ (“quantity”)
2. Prove $f$ to have a lower bound (“vanishes when the machine stops”)
3. Prove $f$ to decrease over time

Example (Termination can be simple)

```plaintext
while x > 0 do
    x = x - 1
done
```
Real programs have

- **Sharing**: Changing variable $x$ influences $y$
Program termination: Challenges

Real programs have

- **Sharing**: Changing variable \( x \) influences \( y \)
- **User-defined data types**: Data has unknown shape

Example (Termination not always simple)

\[
y = x
\]

while not \( y.\text{isEmpty()} \)

\[ \text{do}
\]

\[ x.\text{pop()}
\]

\[ \text{done} \]
Program termination: Challenges

Real programs have

- **Sharing**: Changing variable $x$ influences $y$
- **User-defined data types**: Data has unknown shape
- **Dynamic dispatch**: Executed code chosen only at runtime

Example (Termination not always simple)

```plaintext
y = x
while not y.isEmpty():
    x.pop()
```
Real programs have

- **Sharing**: Changing variable `x` influences `y`
- **User-defined data types**: Data has unknown shape
- **Dynamic dispatch**: Executed code chosen only at runtime

Example (Termination not always simple)

```plaintext
y = x
while not y.isEmpty() do
  x.pop()
done
```
Program termination: Our approach

Program
Program termination: Our approach

Program → Termination Graph
Program termination: Our approach

Program → Termination Graph → Simple Program
Program termination: Our approach

```
while (x > 0)
    x = x - 1;
```
Program termination: Our approach

while (x > 0)
  x = x - 1;

Symbolic evaluation & abstraction
Program termination: Our approach

JAVa → Termination Graph → intTRS

1. Symbolic evaluation & abstraction
2. Translate graph edges to rules, data to terms

while (x > 0)
    x = x - 1;

\( f_A(x) \rightarrow f_A(x - 1) \quad \square x > 0 \)
\( f_A(x) \rightarrow f_C(x) \quad \square x \leq 0 \)
intTRS termination: Our approach

\[
g(\text{List}(n), i) \rightarrow g(n, i - 1) \quad \[ i > 0 \]
\]
intTRS termination: Our approach

\( g(\text{List}(n), i) \to g(n, i - 1) \quad [i > 0] \)
intTRS termination: Our approach

\[ g(\text{List}(n), i) \rightarrow g(n, i - 1) \quad [i > 0] \]

Restrict to terms

Restrict to integers (and/or replacing terms by their “sizes”)
intTRS termination: Our approach

1. Restrict to terms
2. Restrict to integers (and/or replacing terms by their “sizes”)

$$g(\text{List}(n), i) \rightarrow g(n, i - 1) \quad [i > 0]$$

$$g(\text{List}(n)) \rightarrow g(n)$$
Integer Transition System termination: Our approach
Instrumentation (check initially empty termination argument)
Integer Transition System termination: Our approach

1. Instrumentation (check initially empty termination argument)
2. Check termination argument
Integer Transition System termination: Our approach

1. Instrumentation (check initially empty termination argument)
2. Check termination argument
3. Synthesise better termination argument
Integer Transition System termination: Our approach

1. Instrumentation (check initially empty termination argument)
2. Check termination argument
3. Synthesise better termination argument
4. Simplification & Instrumentation (check better termination argument)
Symbolic execution in program analysis:

- Abstract Interpretation
- Termination Graphs for Haskell, Prolog (AProVE)
Related Work

- Symbolic execution in program analysis:
  - *Abstract Interpretation*
  - *Termination Graphs* for **Haskell, Prolog (APoVE)**

- Termination with heap:
  - *Path-length* (**COSTA, Julia**)
  - *Separation Logic* (**Mutant, Thor, Cyclist**)
Related Work

- Symbolic execution in program analysis:
  - Abstract Interpretation
  - Termination Graphs for Haskell, Prolog (AProVE)

- Termination with heap:
  - Path-length (COSTA, Julia)
  - Separation Logic (Mutant, Thor, Cyclist)

- Combining termination arguments:
  - Lexicographic (PolyRank, Rank, T2)
  - Dependency Pair Framework (AProVE, TTT2, Mu-Term, CiME, Matchbox, KITTeL)
  - Transition Invariants (Terminator, ARMC, CProver, TRex, T2, HSF, Acabar)
Overview

1 Introduction

2 Termination of Java
   - Symbolic states
   - Constructing Termination Graphs
   - Generating intTRSSs from Termination Graphs

3 Termination of Integer Transition Systems
   - Termination by iterative strengthening
   - Termination by iterative simplification
   - Cooperative termination proving

4 Conclusion
Terms cannot fully represent the heap
From **Java** to intTRRs: Challenges

Terms cannot fully represent the heap:

1. Side-effects via *sharing*
2. No representation for *cyclic* structures
3. No measure of *distances*
Terms cannot fully represent the heap:

1. Side-effects via *sharing*
2. No representation for *cyclic* structures
3. No measure of *distances*

Solutions:

1. Overapproximate
2. Handle in symbolic evaluation
3. Post-process: Make distances/cycles explicit via counters
length: the example

class L {
  L p, n;
  static int length(L x) {
    int r = 0;
    while (x != null) {
      x = x.n;
      r++;
    }
    return r;
  }
}
class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}
class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}
length: the example

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}
class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}
length: the example

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}
length: the example

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

Stack frame:
- Next program instruction

00: iconst_0  #load 0
01: istore_1  #store to r
02: aload_0  #load x
03: ifnull 17  #jump if x null
06: aload_0  #load x
07: getfield n  #get n from x
10: astore_0  #store to x
11: iinc 1, 1  #increment r
14: goto 2
17: iload_1  #load r
18: ireturn  #return r
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

<table>
<thead>
<tr>
<th>Stack frame</th>
<th>00</th>
<th>x: o₁</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o₁: L(?)</td>
<td>o₁ ⊆ {p,n}</td>
<td></td>
</tr>
</tbody>
</table>

00: iconst 0    #load 0
01: istore 1    #store to r
02: aload 0     #load x
03: ifnull 17   #jump if x null
06: aload 0     #load x
07: getfield n  #get n from x
10: astore 0     #store to x
11: iinc 1, 1   #increment r
14: goto 2
17: iload 1     #load r
18: ireturn     #return r
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

Heap information:
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

Heap information:
- $o_1$ is L object or null

```
00 | x: o_1 | ε
o_1:L(?) o_1 ∩ {p,n}

00: iconst_0  #load 0
01: istore_1  #store to r
02: aload_0  #load x
03: ifnull 17  #jump if x null
06: aload_0  #load x
07: getfield n  #get n from x
10: astore_0  #store to x
11: iinc 1, 1  #increment r
14: goto 2
17: iload_1  #load r
18: ireturn  #return r
```
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

Heap information:
- $o_1$ is L object or null
- Known L object: $o_2 : \text{L}(p = o_3, n = o_4)$
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

Heap information:
- $o_1$ is L object or null
- Known L object: $o_2 : L(p = o_3, n = o_4)$
- Symbolic integers: $i_1 : \mathbb{Z} \quad i_2 : [>0]$
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

Heap information:
- $o_1$ is L object or null
- Known L object: $o_2 : L(p = o_3, n = o_4)$
- Symbolic integers: $i_1 : \mathbb{Z}$, $i_2 : [> 0]$
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

Heap information:
- $o_1$ is L object or null
- Known L object: $o_2 : L(p = o_3, n = o_4)$
- Symbolic integers: $i_1 : \mathbb{Z}$ $i_2 : [> 0]$

Heap predicates: Only explicit sharing
- Two references may be equal: $o_1 =? o_2$
class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

Stack frame:
- Next program instruction
- Local variables
- Operand stack
- Heap information:
  - $o_1$ is L object or null
  - Known L object: $o_2 : L(p = o_3, n = o_4)$
  - Symbolic integers: $i_1 : \mathbb{Z}$ $i_2 : [> 0]$

Heap predicates: Only explicit sharing
- Two references may be equal: $o_1 = ? o_2$
- Two references may share: $o_1 \sqcup o_2$
The abstract domain: symbolic states

class L {
    L p, n;
    static int length(L x) {
        int r = 0;
        while (x != null) {
            x = x.n;
            r++;
        }
        return r;
    }
}

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

Heap information:
- $o_1$ is L object or null
- Known L object: $o_2 : L(p = o_3, n = o_4)$
- Symbolic integers: $i_1 : \mathbb{Z}$ $i_2 : [>0]$

Heap predicates: Only explicit sharing
- Two references may be equal: $o_1 = ? o_2$
- Two references may share: $o_1 \sqsubseteq o_2$
- Reference might have cycles containing all fields $F$: $o_1 \sqcup F$
State A:

- x some list, might contain cycles using p and n

```java
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
```
State A:
- x some list, might contain cycles using p and n

State B:
- Initialized variable r to 0

```java
int length(List<Integer> x) {
    int r = 0;
    while (x != null) {
        x = x.next;
        r++;
    }
    return r;
}
```
State A:
- x some list, might contain cycles using p and n

State B:
- Initialized variable r to 0

State C:
- x (o₁) null? We do not know!

```java
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
```
State A:
- \( x \) some list, might contain cycles using \( p \) and \( n \)

State B:
- Initialized variable \( r \) to 0

States C, D, E:
- \( x \) (\( o_1 \)) null? We do not know!

⇒ Refinement
  - In D: \( o_1 \) is null (\( \bowtie \) program ends)
  - In E: \( o_1 \) replaced by \( o_2 \), which exists and has fields:
    - Field values can share (\( \bowtie \) add \( \bowtie \))
    - Field values can be cyclic again (\( \bowtie \) add \( \bowtie \))

```java
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
```
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
State F:
- Stored `x.n` to `x` (allowing for GC)

State G:
- Incremented `r`, back to position 02 (as B)

```java
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
```
State F:

- Stored `x.n` to `x` (allowing for GC)

States G, B':

- Incremented `r`, back to position 02 (as B)

⇒ Generalization: “Merge” states B, G

```java
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
```
State $F$:
- Stored $x.n$ to $x$ (allowing for GC)

States $G$, $B'$:
- Incremented $r$, back to position 02 (as $B$)

$\Rightarrow$ Generalization: “Merge” states $B$, $G$

States $C'$, $G'$:
- Repetition of $C$, $G$

```java
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
```
State $F$:
- Stored $x.n$ to $x$ (allowing for GC)

States $G$, $B'$:
- Incremented $r$, back to position 02 (as $B$)

$\Rightarrow$ Generalization: “Merge” states $B$, $G$

States $C'$, $G'$:
- Repetition of $C$, $G$

```java
int length(L x) {
    int r = 0;
    while (x != null) {
        x = x.n; r++;
    }
    return r;
}
```
Orientation: Term Rewriting

- Generalized Functional Programming

Rules

\[
app(\text{Cons}(x, xs), ys) \rightarrow \text{Cons}(x, app(xs, ys)) \quad (1)
\]

\[
app(\text{Nil}, ys) \rightarrow ys \quad (2)
\]

Rewriting of term \(t\) with rule \(l \rightarrow r\):

1. Find subterm \(s\) of \(t\)
2. Find variable instantiation \(\sigma\) with \(\sigma(l) = s\)
3. Result \(t'\) is \(t\) with \(s\) replaced by \(\sigma(r)\)

\[
app(\text{Cons}(1, \text{Nil}), \text{Cons}(2, \text{Nil})) \quad \text{with (1), } x = 1, xs = \text{Nil}, ys = \text{Cons}(2, \text{Nil}) \rightarrow \text{Cons}(1, \text{app(\text{Nil}, Cons}(2, \text{Nil})) \quad \text{with (2), } ys = \text{Cons}(2, \text{Nil}) \rightarrow \text{Cons}(1, \text{Cons}(2, \text{Nil}))
\]
Generalized Functional Programming

Rules $\mathcal{R}$ define rewrite relation:

$$
\text{app}(\text{Cons}(x, xs), ys) \rightarrow \text{Cons}(x, \text{app}(xs, ys)) \quad (1)
$$

$$
\text{app}(\text{Nil}, ys) \rightarrow ys \quad (2)
$$

Rewriting of term $t$ with rule $l \rightarrow r$:

1. Find subterm $s$ of $t$
2. Find variable instantiation $\sigma$ with $\sigma(l) = s$
3. Result $t'$ is $t$ with $s$ replaced by $\sigma(r)$
Orientation: Term Rewriting

- Generalized Functional Programming
- Rules $\mathcal{R}$ define rewrite relation:
  \[
  \text{app}(\text{Cons}(x, xs), ys) \rightarrow \text{Cons}(x, \text{app}(xs, ys)) \quad (1)
  \]
  \[
  \text{app}(\text{Nil}, ys) \rightarrow ys \quad (2)
  \]
- Rewriting of term $t$ with rule $l \rightarrow r$:
  1. Find subterm $s$ of $t$
  2. Find variable instantiation $\sigma$ with $\sigma(l) = s$
  3. Result $t'$ is $t$ with $s$ replaced by $\sigma(r)$

\[
\text{app}(\text{Cons}(1, \text{Nil}), \text{Cons}(2, \text{Nil})) \quad \rightarrow \quad \text{Cons}(1, \text{app}(\text{Nil}, \text{Cons}(2, \text{Nil}))) \quad \rightarrow \quad \text{Cons}(1, \text{Cons}(2, \text{Nil}))
\]

with (1), $x = 1$, $xs = \text{Nil}$, $ys = \text{Cons}(2, \text{Nil})$
Transforming values to terms

\[ 03 \mid x: o_2, r: 0 \mid o_2 \]

\[ o_2: L(p = o_3, n = o_4) \]

\[ o_3: L(?) \quad o_4: L(?) \]

\[ o_2 \downarrow o_3 \quad o_2 \downarrow o_4 \quad o_3 \downarrow o_4 \]

\[ o_2, o_3, o_4 \circlearrowright \{p, n\} \]

\( E \)
Transforming values to terms

\[
\begin{align*}
03 & | x : o_2, r : 0 | o_2 \\
o_2 &: L(p = o_3, n = o_4) \\
o_3 &: L(?) \\
o_4 &: L(?) \\
o_2 \setminus o_3 & \quad o_2 \setminus o_4 \quad o_3 \setminus o_4 \\
o_2, o_3, o_4 & \circlearrowleft \{p, n\}
\end{align*}
\]

- Known integers transformed to themselves
Transforming values to terms

| 03 | x: o₂, r: 0 | o₂ |

\[
o₂: L(p = o₃, n = o₄)
\]

\[
o₃: L(?) \quad o₄: L(?)
\]

\[
o₂ \cup o₃ \quad o₂ \cup o₄ \quad o₃ \cup o₄
\]

\[
o₂, o₃, o₄ \circ \{p, n\}
\]

- Known integers transformed to themselves
- Unknown values transformed to variables
Transforming values to terms

03 | x: o₂, r: 0 | o₂

| o₂: L(p = o₃, n = o₄) |
| o₃: L(?) |
| o₄: L(?) |
| o₂ ∨ o₃ |
| o₂ ∨ o₄ |
| o₃ ∨ o₄ |
| o₂, o₃, o₄ ⊲ {p, n} |

- Known integers transformed to themselves
- Unknown values transformed to variables
- Data structures transformed to nested constructor terms:
  - Class Cl with n fields ⊲ symbol Cl of arity n
Transforming values to terms

Known integers transformed to themselves
Unknown values transformed to variables
Data structures transformed to nested constructor terms:
Class $C_1$ with $n$ fields $\sim$ symbol $C_1$ of arity $n$

Encoding cycles: Special symbol $\bigcirc$ for repetition

- $o_5: L(p = \text{null}, n = o_6)$
- $o_6: L(p = o_5, n = o_7)$
- $o_7: L(p = o_6, n = \text{null})$

Encoding of $o_5$: $L(\text{null}, L(\bigcirc, L(\bigcirc, \text{null})))$
Encoding of $o_6$: $L(L(\text{null}, \bigcirc), L(\bigcirc, \text{null}))$
Transforming states to terms

03 | x: o2, r: 0 | o2
---|---|---
o2: L(p = o3, n = o4)
o3: L(?)  o4: L(?)
o2 \(\lor\) o3  o2 \(\lor\) o4  o3 \(\lor\) o4
o2, o3, o4 \(\circ\) \{p, n\}

\[ f_E(\text{L}(o3, o4), 0, \text{L}(o3, o4)) \]

- **State s term encoding:**
  - Root symbol (≡ program position) \( f_s \)
  - All local variables, stack entries as arguments
Transforming edges to rules

\[
E: \begin{array}{l}
03 \mid x: o_2, r: 0 \mid o_2 \\
o_2: L(p = o_3, n = o_4) \\
o_3: L(?) \\
o_4: L(?) \\
o_2 \lor o_3 \lor o_2 \lor o_4 \lor o_4 \\
o_2, o_3, o_4 \circlearrowleft \{ p, n \}
\end{array}
\]

\[
F: \begin{array}{l}
11 \mid x: o_4, r: 0 \mid \varepsilon \\
o_4 : L(?) \\
o_4 \circlearrowleft \{ p, n \}
\end{array}
\]

- **State s term encoding:**
  - Root symbol (≡ program position) \( f_s \)
  - All local variables, stack entries as arguments

- **Evaluation edges:** Encode states, put in →

\[
f_E(\underbrace{L(o_3, o_4)}_{o_2}, 0, \underbrace{L(o_3, o_4)}_{o_2}) \rightarrow f_F(o_4, 0)
\]
Transforming edges to rules

\[03 | x: o_2, r: 0 | o_2\]
\[o_2: L(p = o_3, n = o_4)\]
\[o_3: L(?)\]
\[o_4: L(?)\]
\[o_2 \lor o_3\]
\[o_2 \lor o_4\]
\[o_3 \lor o_4\]
\[o_2, o_3, o_4 \Rightarrow \{p, n\}\]

\[11 | x: o_4, r: 0 | \varepsilon\]
\[o_4 : L(?)\]
\[o_4 \Rightarrow \{p, n\}\]

- **State term encoding:**
  - Root symbol (≡ program position) \(f_s\)
  - All local variables, stack entries as arguments

- **Evaluation edges:** Encode states, put in →
  - **Problem:** Cycle encoding changes \(\rightsquigarrow\) free var on rhs

\[f_E(L(o_3, o_4), 0, L(o_3, o_4)) \rightarrow f_F(o_4', 0)\]
Transforming edges to rules

State $s$ term encoding:
- Root symbol ($\equiv$ program position) $f_s$
- All local variables, stack entries as arguments

Evaluation edges: Encode states, put in $\rightarrow$
- **Problem**: Cycle encoding changes $\leadsto$ free var on rhs
- **Solution**: Filter: Only encode non-cyclic parts!

\[
f_E(\underbrace{L(\begin{array}{c} o_2 \\ o_4 \end{array})}_0, \underbrace{L(\begin{array}{c} o_2 \\ o_4 \end{array})}_0) \rightarrow f_F(o_4, 0)\]
Transforming edges to rules

State $s$ term encoding:
- Root symbol (≡ program position) $f_s$
- All local variables, stack entries as arguments

Evaluation edges: Encode states, put in $\rightarrow$
- **Problem**: Cycle encoding changes $\bowtie$ free var on rhs
- **Solution**: Filter: Only encode non-cyclic parts!

Refinement edges: Encode target state twice, relabel

$$f_C(L(o_4), 0, L(o_4)) \rightarrow f_E(L(o_4), 0, L(o_4))$$
Transforming edges to rules

State $s$ term encoding:
- Root symbol ($\equiv$ program position) $f_s$
- All local variables, stack entries as arguments

Evaluation edges: Encode states, put in $\rightarrow$
- **Problem**: Cycle encoding changes $\bowtie$ free var on rhs
- **Solution**: Filter: Only encode non-cyclic parts!

Refinement edges: Encode target state twice, relabel

Instantiation edges: Encode source state twice, relabel

$$f_G(o_4, 2) \rightarrow f_{B'}(o_4, 2)$$
The example TRS

```
00: iconst 1
01: istore 1
02: aload 0
03: ifnull 17
06: aload 0
07: getfield n
10: astore 0
11: iinc 1, 1
14: goto 2
17: iload 1
18: ireturn
```

Diagram of the example TRS
The example TRS

00: icnst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn

1 Only consider SCCs!
The example TRS

00: iconst_1
01: istore_1
02: aload_0
03: ifnonnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload 1
18: ireturn

1. Only consider SCCs!
2. Transform all edges as before, simplify:

\[
f_{B'}(L(o'_4), i_1) \rightarrow f_{B'}(o'_4, i_1 + 1)\]
AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA’10]
AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA’10]
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA’10]
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA’10]
  - on acyclic user-defined data structures (trees, DAGs, . . .) [RTA’10]
Implementation for full Java without reflection and multithreading

Correctness proof w.r.t. JINJA [VITA’10]

Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity

Termination analysis for algorithms

- on integers [RTA’10]
- on acyclic user-defined data structures (trees, DAGs, . . .) [RTA’10]
- using recursion [RTA’11]
AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA’10]
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA’10]
  - on acyclic user-defined data structures (trees, DAGs, . . . ) [RTA’10]
  - using recursion [RTA’11]
  - on cyclic data [CAV’12]
    - by measuring distances
    - by detecting (and ignoring) irrelevant cyclicity
    - by automatically finding and counting markers
- Non-termination analysis [FoVeOOS’11]
AProVE features for Java

- Implementation for full Java without reflection and multithreading
- Correctness proof w.r.t. JINJA [VITA’10]
- Built-in, implicit analyses for nullness, aliasing, sharing, cyclicity
- Termination analysis for algorithms
  - on integers [RTA’10]
  - on acyclic user-defined data structures (trees, DAGs, . . .) [RTA’10]
  - using recursion [RTA’11]
  - on cyclic data [CAV’12]
    - by measuring distances
    - by detecting (and ignoring) irrelevant cyclicity
    - by automatically finding and counting markers
- Non-termination analysis [FoVeOOS’11]
Experimental results

- Evaluated on collection of 441 programs from *Termination Problem Data Base*

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
<th>Fail</th>
<th>Run (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>289</td>
<td>125</td>
<td>27</td>
<td>16.6</td>
</tr>
<tr>
<td>Julia</td>
<td>205</td>
<td>79</td>
<td>157</td>
<td>6.5</td>
</tr>
<tr>
<td>COSTA</td>
<td>163</td>
<td>0</td>
<td>278</td>
<td>13.1</td>
</tr>
</tbody>
</table>
Experimental results

- Evaluated on collection of 441 programs from *Termination Problem Data Base*

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
<th>Fail</th>
<th>Run (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>289</td>
<td>125</td>
<td>27</td>
<td>16.6</td>
</tr>
<tr>
<td>Julia</td>
<td>205</td>
<td>79</td>
<td>157</td>
<td>6.5</td>
</tr>
<tr>
<td>COSTA</td>
<td>163</td>
<td>0</td>
<td>278</td>
<td>13.1</td>
</tr>
</tbody>
</table>

- Won Termination Competition 2012/2013

Open problems:

- Abstraction refinement
- Modular analysis

Specialized abstract domains:

- easy to automate
- very effective
Experimental results

- Evaluated on collection of 441 programs from *Termination Problem Data Base*

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
<th>Fail</th>
<th>Run (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>289</td>
<td>125</td>
<td>27</td>
<td>16.6</td>
</tr>
<tr>
<td>Julia</td>
<td>205</td>
<td>79</td>
<td>157</td>
<td>6.5</td>
</tr>
<tr>
<td>COSTA</td>
<td>163</td>
<td>0</td>
<td>278</td>
<td>13.1</td>
</tr>
</tbody>
</table>

- Won Termination Competition 2012/2013

- Open problems:
  - Abstraction refinement
  - Modular analysis
Experimental results

- Evaluated on collection of 441 programs from *Termination Problem Data Base*

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
<th>Fail</th>
<th>Run (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AProVE</td>
<td>289</td>
<td>125</td>
<td>27</td>
<td>16.6</td>
</tr>
<tr>
<td>Julia</td>
<td>205</td>
<td>79</td>
<td>157</td>
<td>6.5</td>
</tr>
<tr>
<td>COSTA</td>
<td>163</td>
<td>0</td>
<td>278</td>
<td>13.1</td>
</tr>
</tbody>
</table>

- Won Termination Competition 2012/2013
- Open problems:
  - Abstraction refinement
  - Modular analysis

**Specialized abstract domains:**
- easy to automate
- very effective
Example

y := 1;
while x > 0 do
  x := x − y;
  y := y + 1;
done

- Invariant \( y > 0 \) and rank function \( x \) prove termination
- How do we know that we need \( y > 0 \)? → \( x \) requires it
Termination Analysis: Invariants and Rank Functions

Example

```plaintext
y := 1;
while x > 0 do
    x := x − y;
y := y + 1;
done
```

- Invariant $y > 0$ and rank function $x$ prove termination.
- How do we know that we need $y > 0$? $\rightarrow$ $x$ requires it.
- How do we know that $x$ is a RF? $\rightarrow$ $y > 0$ proves it.
Termination by iterative strengthening: Idea

1. Safety: Provide samples (Counterexamples)
2. Rank tool: Find specific termination argument
3. Safety: Prove generality, or 1
Termination by iterative strengthening: Idea

1. Safety: Provide samples (Counterexamples)
2. Rank tool: Find specific termination argument
3. Safety: Prove generality, or 1
Termination by iterative strengthening

Find counterexample
then strengthen argument
Termination by iterative strengthening

Execution

Find counterexample then strengthen argument

Loop states
Termination by iterative strengthening

Loop states

Terminating states

Execution

Find counterexample
then strengthen argument
Termination by iterative strengthening

Find counterexample then strengthen argument

Execution

Terminating states
Termination by iterative strengthening

Find counterexample then strengthen argument
Termination by iterative strengthening: Worst case

1. Safety: Look at everything, then return old sample
2. Rank tool: Find too specific termination argument
3. Safety: Can’t prove generality, repeat 1
Termination by iterative strengthening: Worst case

1. Safety: Look at everything, then return old sample
2. Rank tool: Find **too** specific termination argument
3. Safety: Can’t prove generality, repeat 1

![Fine! Fine!](https://i.imgur.com/3Q5J.png)
Termination by iterative simplification

Loop trans.
Termination by iterative simplification

Execution

Loop trans.
Termination by iterative simplification

Find rank function for SCC
Termination by iterative simplification

Find rank function for SCC then remove transitions
Termination by iterative simplification

Find rank function for SCC then remove transitions

Execution
Termination by iterative simplification

Find rank function for SCC then remove transitions
Termination by cooperation

1. Safety: Provide samples (Counterexamples)
2. Rank tool: Find termination argument in context
3. Rank tool: Mark definitely terminating parts (simplify)
4. Safety: Prove generality for rest, or 1
Cooperation: High-level view

Safety

Termination
Cooperation: High-level view

Safety

Termination
Cooperation: High-level view

- Safety
- Termination
  - Terminating states
Cooperation: High-level view

- Safety
- Termination
- Terminating states
Cooperation: High-level view

Example (Source)

```plaintext
if k ≥ 1 then
i := 0;

ℓ1 while i < n do
j := 0;

ℓ2 while j ≤ i do
j := j + k;
    done
    j := j + k;
    done
fi
```
\( \tau_0 : \text{if}(k \geq 1); \quad i := 0; \)

\( \tau_1 : \text{if}(i < n); \quad j := 0; \)

\( \tau_2 : \text{if}(j > i); \quad i := i + 1; \)

\( \tau_3 : \text{if}(j \leq i); \quad j := j + k; \)
Intuition:

- **Safety subgraph**: original program
- **Termination subgraph**: instrumented copy
Cooperation

Intuition:

- **Safety subgraph**: original program
- **Termination subgraph**: instrumented copy

- **Ranking**: Simplify problem, “point out hard bits”
Cooperation

Intuition:

- **Safety subgraph**: original program
- **Termination subgraph**: instrumented copy

- **Ranking**: Simplify problem, “point out hard bits”
- **Safety**: Analyze whole program, “point out invariants”
Cooperation

Intuition:

- **Safety subgraph**: original program
- **Termination subgraph**: instrumented copy

- **Ranking**: Simplify problem, “point out hard bits”
- **Safety**: Analyze whole program, “point out invariants”

Approach:

- Analyze whole SCC, not counterexample slice
Intuition:
- **Safety subgraph**: original program
- **Termination subgraph**: instrumented copy

- **Ranking**: Simplify problem, “point out hard bits”
- **Safety**: Analyze whole program, “point out invariants”

Approach:
- Analyze whole SCC, not counterexample slice
- Remove transitions after proof
\(\tau_0: \text{if}(k \geq 1); \quad i := 0;\)

\(\tau_1: \text{if}(i < n); \quad j := 0;\)

\(\tau_2: \text{if}(j > i); \quad i := i + 1;\)

\(\tau_3: \text{if}(j \leq i); \quad j := j + k;\)

maybe take a snapshot

check decrease

maybe take a snapshot

check decrease
Cooperation: Simplification

Simplification

- Check decrease
  - $\text{\(\ell_t^1\)}$
    - If $i < n$, $j := 0$
  - If $j > i$, $i := i + 1$
  - If $j \leq i$, $j := j + k$

- Maybe take a snapshot
  - $\text{\(\ell_t^2\)}$
  - $\text{\(\ell_d^1\)}$
  - $\text{\(\ell_d^2\)}$

Find SCC $S$ in termination graph:
- $\text{\(\ell_t^1, \ell_d^1, \ell_t^2, \ell_d^2\)}$

Find $S$-orienting RF:
- $f_1(\ell_t^1(i, j, k, n) = n - i + 1$
- $f_1(\ell_d^1(i, j, k, n) = n - i + 1$
- $f_1(\ell_t^2(i, j, k, n) = n$
- $f_1(\ell_d^2(i, j, k, n) = n$

Delete decr./bounded

Clean up
Cooperation: Simplification

1. Find SCC $S$ in termination graph:
\[ \ell^t_1, \ell^d_1, \ell^t_2, \ell^d_2 \]

- Simplification
- Find SCC $S$ in termination graph:
  - $\ell^t_1, \ell^d_1, \ell^t_2, \ell^d_2$
- maybe take a snapshot
- maybe take a snapshot
- check decrease
- check decrease
- $\tau^t_1$: if ($i < n$); $j := 0$;
- $\tau^t_2$: if ($j \leq i$); $i := i + 1$;
- $\tau^t_3$: if ($j > i$); $j := j + k$;

- $\ell^t_1$
- $\ell^t_2$
- $\ell^d_1$
- $\ell^d_2$
Simplification

1. Find SCC $S$ in termination graph:
   $\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$

2. Find $S$-orienting RF:
   
   $f_{\ell_1^t}^1(i,j,k,n) = n - i + 1$

   $f_{\ell_1^d}^1(i,j,k,n) = n - i + 1$

   $f_{\ell_2^t}^1(i,j,k,n) = n - i$

   $f_{\ell_2^d}^1(i,j,k,n) = n - i$

Delete decr./bounded
Clean up
Cooperation: Simplification

Simplification

1. Find SCC $S$ in termination graph:
   $\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$

2. Find $S$-orienting RF:
   $f_{\ell_1^t}(i,j,k,n) = n - i + 1$
   $f_{\ell_1^d}(i,j,k,n) = n - i + 1$
   $f_{\ell_2^t}(i,j,k,n) = n - i$
   $f_{\ell_2^d}(i,j,k,n) = n - i$

3. Delete decre./bounded
Cooperation: Simplification

Simplification

1. Find SCC $S$ in termination graph:
   $\ell^t_1, \ell^d_1, \ell^t_2, \ell^d_2$

2. Find $S$-orienting RF:
   - $f_{\ell^t_1}(i,j,k,n) = n - i + 1$
   - $f_{\ell^d_1}(i,j,k,n) = n - i + 1$
   - $f_{\ell^t_2}(i,j,k,n) = n - i$
   - $f_{\ell^d_2}(i,j,k,n) = n - i$

3. Delete decr./bounded

\[ \tau^t_3: \text{if}(j \leq i); \]
\[ j := j + k; \]
\[ \tau^d_2: i := i + 1. \]
Cooperation: Simplification

Simplification

1. Find SCC $S$ in termination graph:
   $$\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$$

2. Find $S$-orienting RF:
   $$f_{\ell_1^t}^1(i,j,k,n) = n - i + 1$$
   $$f_{\ell_1^d}^1(i,j,k,n) = n - i + 1$$
   $$f_{\ell_2^t}^1(i,j,k,n) = n - i$$
   $$f_{\ell_2^d}^1(i,j,k,n) = n - i$$

3. Delete decr./bounded

4. Clean up

$$\tau_3^t: \text{if}(j \leq i); j := j + k;$$

maybe take a snapshot

check decrease
Cooperation

\[\ell_1:\text{if}(k \geq 1);
\quad i := 0;\]

\[\ell_2:\text{if}(j > i);
\quad i := i + 1;\]

\[\tau_0:\text{if}(i < n);
\quad j := 0;\]

\[\tau_1:\text{if}(j \leq i);
\quad j := j + k;\]

\[\tau_2:\text{if}(j \leq i);
\quad j := j + k;\]

maybe take a snapshot

check decrease
Cooperation: Invariants

\begin{align*}
\tau_0 &: \text{if}(k \geq 1); \\
i &: = 0;
\end{align*}

\begin{align*}
\tau_2 &: \text{if}(j > i); \\
i &: = i + 1;
\end{align*}

\begin{align*}
\tau_1 &: \text{if}(i < n); \\
j &: = 0;
\end{align*}

\begin{align*}
\tau_3 &: \text{if}(j \leq i); \\
j &: = j + k;
\end{align*}

Construction/Checking

1. No simplification possible

\begin{align*}
\tau_3^t &: \text{if}(j \leq i); \\
j &: = j + k;
\end{align*}

maybe take a snapshot

check decrease
Cooperation: Invariants

\[ \tau_0 : \text{if}(k \geq 1); ~ i := 0; \]

\[ \tau_1 : \text{if}(i < n); ~ j := 0; \]

\[ \tau_2 : \text{if}(j > i); ~ i := i + 1; \]

\[ \tau_3 : \text{if}(j \leq i); ~ j := j + k; \]

Construction/Checking

1. No simplification possible
2. Obtain counterexample:
   \[ \tau_0 \tau_1 (\text{snapshot}) \tau_3^t \]

maybe take a snapshot

check decrease
Cooperation: Invariants

\[ \tau_0 : \text{if}(k \geq 1); \]
\[ i := 0; \]

\[ \tau_1 : \text{if}(i < n); \]
\[ j := 0; \]

\[ \tau_2 : \text{if}(j > i); \]
\[ i := i + 1; \]

\[ \tau_3 : \text{if}(j \leq i); \]
\[ j := j + k; \]

Construction/Checking

1. No simplification possible
2. Obtain counterexample:
   \[ \tau_0 \tau_1 (\text{snapshot}) \tau_3^t \]
3. Compute RF:
   \[ f_{\ell_2^t}(i,j,k,n) = i - j \]

maybe take a snapshot
Cooperation: Invariants

\[ \tau_0: \text{if}(k \geq 1); \]
\[ i := 0; \]

\[ \tau_1: \text{if}(i < n); \]
\[ j := 0; \]

\[ \tau_2: \text{if}(j > i); \]
\[ i := i + 1; \]

\[ \tau_3: \text{if}(j \leq i); \]
\[ j := j + k; \]

Construction/Checking

1. No simplification possible
2. Obtain counterexample:
   \[ \tau_0 \tau_1 (\text{snapshot}) \tau_3^t \]
3. Compute RF:
   \[ f_{\ell_2^t}(i, j, k, n) = i - j \]
4. Add to instrumentation

maybe take a snapshot
Cooperation: Invariants

\[ \tau_0 : \text{if}(k \geq 1); \]
\[ \quad i := 0; \]

\[ \tau_1 : \text{if}(i < n); \]
\[ \quad j := 0; \]

\[ \tau_2 : \text{if}(j > i); \]
\[ \quad i := i + 1; \]

\[ \tau_3 : \text{if}(j \leq i); \]
\[ \quad j := j + k; \]

\[ \rho_2 : \text{if}(c_{p2} \geq 1); \]
\[ \quad \text{if}(-(i - j > i - j)); \]

\[ \tau_3^t : \text{if}(j \leq i); \]
\[ \quad j := j + k; \]

maybe take a snapshot
Evaluated on 449 termination proving benchmarks
260 known terminating, 181 known non-terminating, 8 unknown
Sources: Windows drivers, Apache, PostgreSQL, ...
Cooperation: Evaluation

Evaluated on 449 termination proving benchmarks
260 known terminating, 181 known non-terminating, 8 unknown
Sources: Windows drivers, Apache, PostgreSQL, ...

<table>
<thead>
<tr>
<th>Source</th>
<th>Term (#)</th>
<th>Term (avg. s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperating-T2</td>
<td>245</td>
<td>3.42</td>
</tr>
<tr>
<td>AProVE</td>
<td>197</td>
<td>2.21</td>
</tr>
<tr>
<td>KITTeL</td>
<td>196</td>
<td>4.65</td>
</tr>
<tr>
<td>T2</td>
<td>189</td>
<td>5.15</td>
</tr>
<tr>
<td>AProVE+Interproc</td>
<td>185</td>
<td>1.53</td>
</tr>
<tr>
<td>Terminator</td>
<td>177</td>
<td>4.99</td>
</tr>
<tr>
<td>Size-Change/MCNP</td>
<td>156</td>
<td>17.50</td>
</tr>
<tr>
<td>ARMC</td>
<td>138</td>
<td>16.16</td>
</tr>
</tbody>
</table>
Cooperation: Evaluation

Evaluated on 449 termination proving benchmarks
260 known terminating, 181 known non-terminating, 8 unknown
Sources: Windows drivers, Apache, PostgreSQL, ...

<table>
<thead>
<tr>
<th>Cooperator</th>
<th>Term (#)</th>
<th>Term (avg. s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperating-T2</td>
<td>245</td>
<td>3.42</td>
</tr>
<tr>
<td>AProVE</td>
<td>197</td>
<td>2.21</td>
</tr>
<tr>
<td>KITTeL</td>
<td>196</td>
<td>4.65</td>
</tr>
<tr>
<td>T2</td>
<td>189</td>
<td>5.15</td>
</tr>
<tr>
<td>AProVE+Interproc</td>
<td>185</td>
<td>1.53</td>
</tr>
<tr>
<td>Terminator</td>
<td>177</td>
<td>4.99</td>
</tr>
<tr>
<td>Size-Change/MCNP</td>
<td>156</td>
<td>17.50</td>
</tr>
<tr>
<td>ARMC</td>
<td>138</td>
<td>16.16</td>
</tr>
</tbody>
</table>

Sources available: http://research.microsoft.com/en-us/projects/t2/
Program termination: Our approach

1. Symbolic evaluation
2. Translation + post-processing
Program termination: Our approach

1. Symbolic evaluation
2. Translation + post-processing
3. Restriction to terms
4. Restriction to integers & replacing terms by their “sizes”
Proving termination of \textsc{java}:

1. Translation from Termination Graph to intTRS
2. Post-processing Termination Graphs: Handle cycles, distances
3. Non-termination proofs on Termination Graphs
Program termination: Contributions of this thesis

- Proving termination of **Java**:
  1. Translation from Termination Graph to intTRS
  2. Post-processing Termination Graphs: Handle cycles, distances
  3. Non-termination proofs on Termination Graphs

- Proving termination of intTRSs:
  4. Simplification of automatically generated intTRSs
  5. Abstracting terms to their height
  6. Termination proofs by alternating TRS/ITS techniques
Program termination: Contributions of this thesis

- Proving termination of **Java**:  
  1. Translation from Termination Graph to intTRS  
  2. Post-processing Termination Graphs: Handle cycles, distances  
  3. Non-termination proofs on Termination Graphs

- Proving termination of intTRSs:  
  4. Simplification of automatically generated intTRSs  
  5. Abstracting terms to their height  
  6. Termination proofs by alternating TRS/ITS techniques

- Proving termination of Integer Transition Systems:  
  7. Cooperative termination proving  
  8. Alternating termination/non-termination proving
Program termination: Contributions of this thesis

- Proving termination of **Java**:
  1. Translation from Termination Graph to intTRS
  2. Post-processing Termination Graphs: Handle cycles, distances
  3. Non-termination proofs on Termination Graphs

- Proving termination of intTRSs:
  4. Simplification of automatically generated intTRSs
  5. Abstracting terms to their height
  6. Termination proofs by alternating TRS/ITS techniques

- Proving termination of Integer Transition Systems:
  7. Cooperative termination proving
  8. Alternating termination/non-termination proving

- Implementations, most powerful in their fields:
  a. **AProVE**: 1-6
  b. **T2**: 7-8