# 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

# 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

# Program termination: Overview

## Turing 1949

Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.

"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")

# Program termination: Overview

## Turing 1949

Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.

"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")

# Program termination: Overview

## Turing 1949

Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.

"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

# Program termination: Overview

## Turing 1949

Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer giving a further definite assertion to be verified. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.

"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)

```
while x > 0 do
    x = x - 1
done
```

# Program termination: Challenges

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

# 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

# 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)**

```
y = x
while not y.isEmpty() do
    x.pop()
done
```

# Program termination: Our approach

Program

# Program termination: Our approach

$\boxed{\text{Program}} \longrightarrow \boxed{\text{Termination Graph}}$

# Program termination: Our approach

$\boxed{\text{Program}} \longrightarrow \boxed{\text{Termination Graph}} \longrightarrow \boxed{\text{Simple Program}}$

# Program termination: Our approach

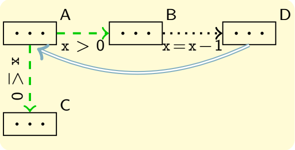
JAVA ⟶ Termination Graph ⟶ Simple Program

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

# Program termination: Our approach

JAVA —1→ Termination Graph → Simple Program

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

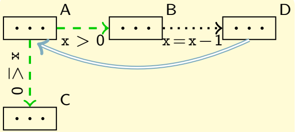



1. Symbolic evaluation & abstraction

# Program termination: Our approach

JAVA ➊ ⟶ Termination Graph ➋ ⟶ intTRS

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




$$f_A(x) \to f_A(x-1) \; [\![\, x > 0 \,]\!]$$

$$f_A(x) \to f_C(x) \; [\![\, x \leq 0 \,]\!]$$

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

# intTRS termination: Our approach

intTRS

$\mathsf{g}(\mathsf{List}(n), i) \to \mathsf{g}(n, i-1)$
$[\![\, i > 0 \,]\!]$

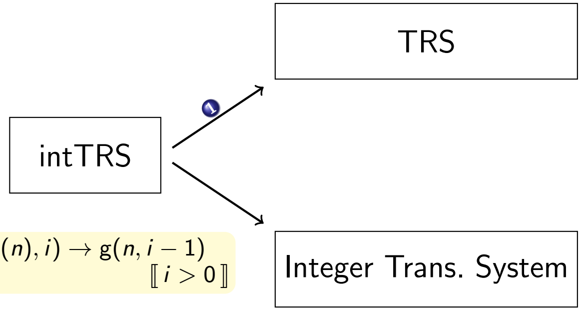
# intTRS termination: Our approach

$$\text{intTRS} \rightarrow \text{TRS}$$

$$\text{intTRS} \rightarrow \text{Integer Trans. System}$$

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

# intTRS termination: Our approach

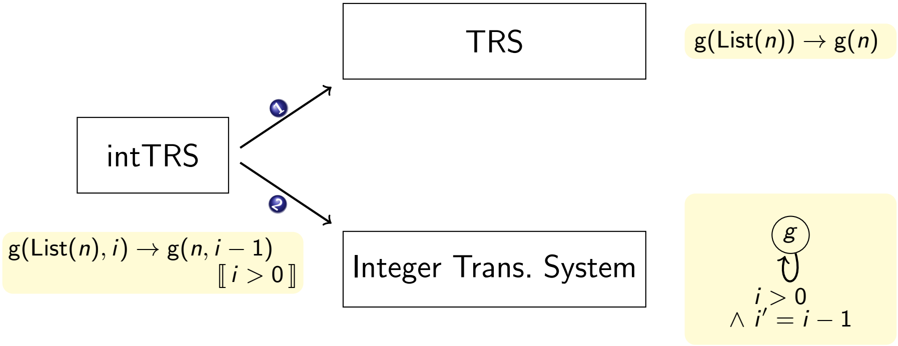



$\mathrm{g}(\mathrm{List}(n)) \to \mathrm{g}(n)$

$\mathrm{g}(\mathrm{List}(n), i) \to \mathrm{g}(n, i-1)$ $[\![\, i > 0 \,]\!]$

1. Restrict to terms

# intTRS termination: Our approach

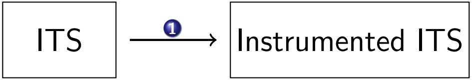



1. Restrict to terms
2. Restrict to integers (and/or replacing terms by their "sizes")

# Integer Transition System termination: Our approach

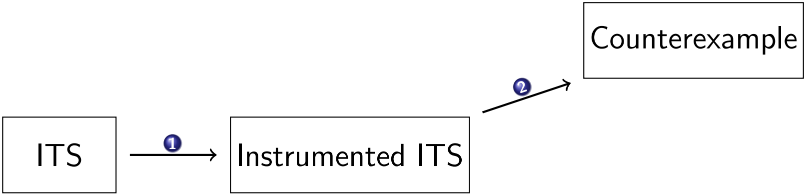
ITS

# Integer Transition System termination: Our approach



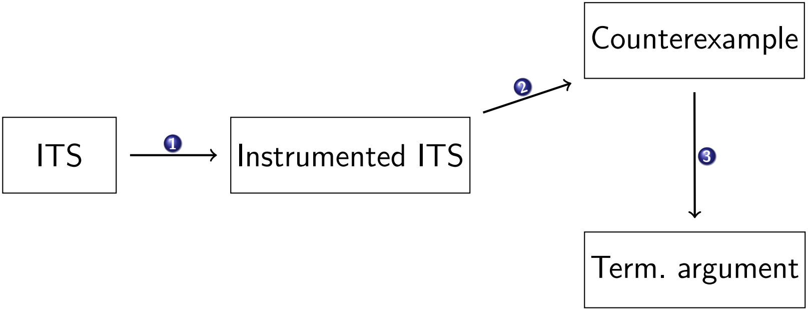

1. Instrumentation (check initially empty termination argument)

# Integer Transition System termination: Our approach



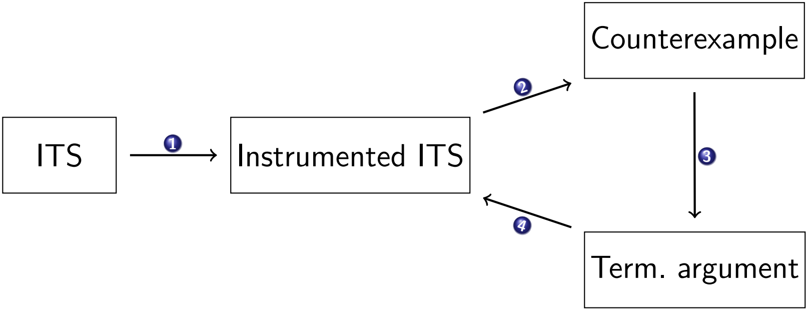

1. Instrumentation (check initially empty termination argument)
2. Check termination argument

# 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)

# Related Work

- 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 (AProVE)
- 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

# From Java to intTRSs: Challenges

Terms cannot fully represent the heap

# From JAVA to intTRSs: Challenges

Terms cannot fully represent the heap:

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

# From JAVA to intTRSs: Challenges

Terms cannot fully represent the heap:

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

Solutions:

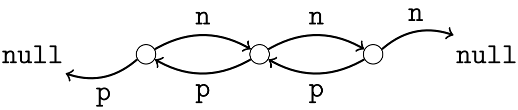
1. Overapproximate
2. Handle in symbolic evaluation
3. Post-process: Make distances/cycles explicit via counters

# 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; }}
```

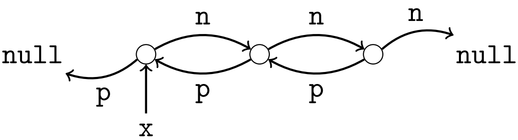
# length: the example

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



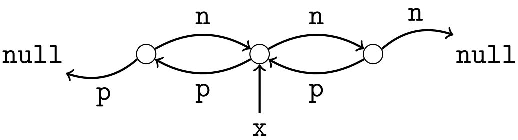
# length: the example

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



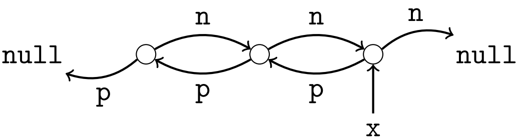
# length: the example

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



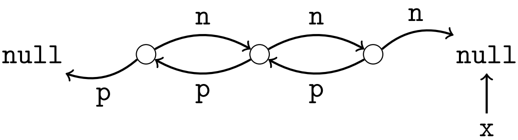
# length: the example

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



# 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; }}
```

```
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; }}
```

```
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; }}
```

```
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
```

**Stack frame:**

- Next program instruction

| 00 | x : $o_1$ | $\varepsilon$ |
|---|---|---|

$o_1$ : L(?)    $o_1 \circlearrowleft_{\{p,n\}}$

# 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; }}
```

```
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
```

| | | |
|---|---|---|
| 00 | $\texttt{x}: o_1$ | $\varepsilon$ |
| $o_1$: L(?) | $o_1 \circlearrowleft_{\{p,n\}}$ | |

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

# 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; }}
```

```
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
```

$00 \mid \mathtt{x} : o_1 \mid \varepsilon$

$o_1 : \mathrm{L}(?) \qquad o_1 \circlearrowleft_{\{\mathrm{p},\mathrm{n}\}}$

**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; }}
```

```
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
```

| | | |
|---|---|---|
| 00 | $\mathtt{x}: o_1$ | $\varepsilon$ |
| $o_1{:}\,\mathtt{L}(?)$ | $o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$ | |

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

**Heap information:**

- $o_1$ is L object or null

# 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; }}
```

```
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
```

$00 \mid \mathtt{x}: o_1 \mid \varepsilon$

$o_1 : \mathtt{L}(?) \qquad o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

**Heap information:**

- $o_1$ is L object or null
- Known L object: $o_2 : \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4)$

# 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; }}
```

```
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
```

$00 \mid \mathtt{x} : o_1 \mid \varepsilon$

$o_1 : \mathrm{L}(?) \qquad o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

**Heap information:**

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

# 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; }}
```

```
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
```

$00 \mid \mathtt{x} : o_1 \mid \varepsilon$

$o_1 : \mathrm{L}(?) \qquad o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

**Heap information:**

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

**Only explicit sharing**

# 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; }}
```

```
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
```

| | | |
|---|---|---|
| 00 | $\mathtt{x}:o_1$ | $\varepsilon$ |
| $o_1 : \mathrm{L}(?)$ | $o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$ | |

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

**Heap information:**

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

**Heap predicates:** **Only explicit sharing**

- Two references may be equal: $o_1 =^? o_2$

# 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; }}
```

```
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
```

$00 \mid \mathtt{x}: o_1 \mid \varepsilon$

$o_1 : \mathrm{L}(?) \qquad o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

**Heap information:**

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

**Heap predicates:** **Only explicit sharing**

- Two references may be equal: $o_1 =^? o_2$
- Two references may share: $o_1 \searrow\!\swarrow o_2$

# 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; }}
```

```
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
```

$00 \mid \mathtt{x} : o_1 \mid \varepsilon$

$o_1 : \mathtt{L}(?) \quad o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

**Stack frame:**

- Next program instruction
- Local variables
- Operand stack

**Heap information:**

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

**Heap predicates:** **Only explicit sharing**

- Two references may be equal: $o_1 =^? o_2$
- Two references may share: $o_1 \searrow\!\!\swarrow o_2$
- Reference might have cycles containing all fields $F$: $o_1 \circlearrowleft_F$

```
00: iconst_0
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
```

| $00 \mid \mathtt{x} : o_1 \mid \varepsilon$ |
|---|
| $o_1 : \mathtt{L}(?) \quad o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$ |

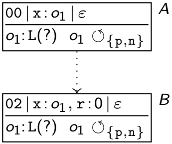
$A$

## State $A$:

- $\mathtt{x}$ some list, might contain cycles using $\mathtt{p}$ and $\mathtt{n}$

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

```
00: iconst_0
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
```




State *A*:

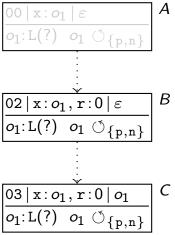
- x some list, might contain cycles using p and n

State *B*:

- Initialized variable r to 0

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

```
00: iconst_0
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
```




State *A*:

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

State *B*:

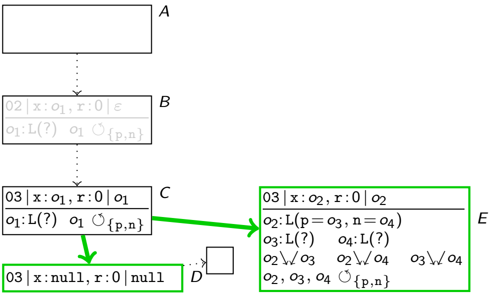
- Initialized variable r to 0

State *C*:

- x ($o_1$) null? We do not know!

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

```
00: iconst_0
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
```




State *A*:

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

State *B*:

- Initialized variable r to 0

States *C*, *D*, *E*:

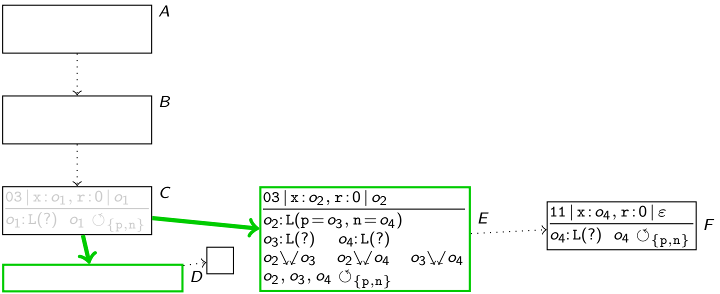
- x ($o_1$) null? We do not know!

⇒ Refinement

- In *D*: $o_1$ is null (↷ program ends)
- In *E*: $o_1$ replaced by $o_2$, which exists and has fields:
  - Field values can share (↷ add ⩛)
  - Field values can be cyclic again (↷ add ↺)

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

```
00: iconst_0
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
```

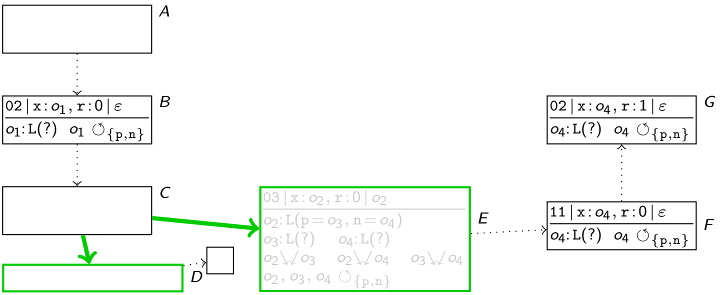



State $F$:

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

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

```
00: iconst_0
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
```




State *F*:

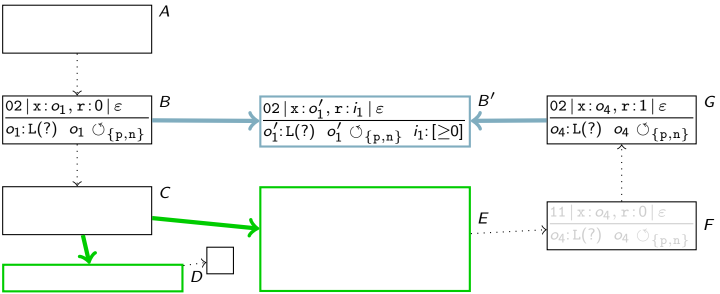
- Stored `x.n` to `x` (allowing for GC)

State *G*:

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

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

```
00: iconst_0
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
```




State *F*:

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

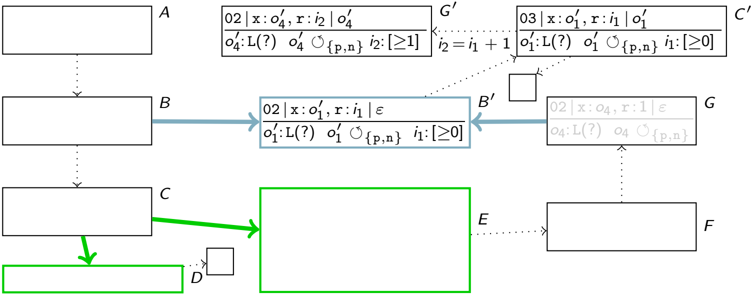
States *G*, *B′*:

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

⇒ Generalization: "Merge" states *B*, *G*

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

```
00: iconst_0
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
```




State $F$:

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

States $G$, $B'$:

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

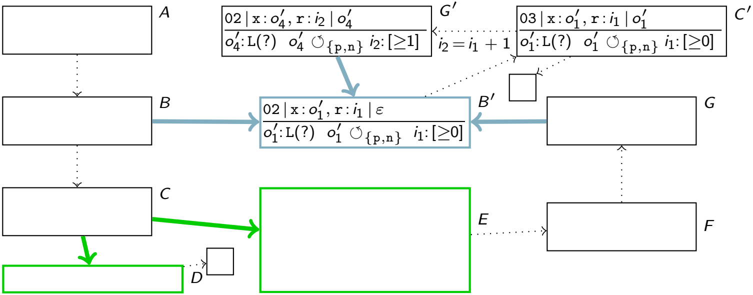
⇒ Generalization: “Merge” states $B$, $G$

States $C'$, $G'$:

- Repetition of $C$, $G$

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

```
00: iconst_0
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
```




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*

States *C′*, *G′*:

- Repetition of *C*, *G*

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

# Orientation: Term Rewriting

- Generalized Functional Programming

# Orientation: Term Rewriting

- Generalized Functional Programming
- Rules $\mathcal{R}$ define rewrite relation:

$$\mathsf{app}(\mathsf{Cons}(x, xs), ys) \to \mathsf{Cons}(x, \mathsf{app}(xs, ys)) \tag{1}$$

$$\mathsf{app}(\mathsf{Nil}, ys) \to ys \tag{2}$$

- Rewriting of term $t$ with rule $l \to r$:
  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:

$$\mathsf{app}(\mathsf{Cons}(x, xs), ys) \to \mathsf{Cons}(x, \mathsf{app}(xs, ys)) \quad (1)$$
$$\mathsf{app}(\mathsf{Nil}, ys) \to ys \quad (2)$$

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

$$\begin{aligned}
&\underline{\mathsf{app}(\mathsf{Cons}(1, \mathsf{Nil}), \mathsf{Cons}(2, \mathsf{Nil}))} && \text{with } (1), x = 1, xs = \mathsf{Nil}, \\
& && \qquad ys = \mathsf{Cons}(2, \mathsf{Nil}) \\
\to\ & \mathsf{Cons}(1, \underline{\mathsf{app}(\mathsf{Nil}, \mathsf{Cons}(2, \mathsf{Nil}))}) && \text{with } (2), ys = \mathsf{Cons}(2, \mathsf{Nil}) \\
\to\ & \mathsf{Cons}(1, \mathsf{Cons}(2, \mathsf{Nil}))
\end{aligned}$$

# Transforming values to terms

$$\frac{03 \mid \mathtt{x} : o_2, \mathtt{r} : 0 \mid o_2}{\begin{array}{l} o_2 \colon \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3 \colon \mathtt{L}(?) \qquad o_4 \colon \mathtt{L}(?) \\ o_2 \curlyvee o_3 \qquad o_2 \curlyvee o_4 \qquad o_3 \curlyvee o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \quad E$$

# Transforming values to terms

$$\frac{\texttt{03} \mid \texttt{x}: o_2, \texttt{r}: 0 \mid o_2}{\begin{array}{l} o_2: \texttt{L}(\texttt{p} = o_3, \texttt{n} = o_4) \\ o_3: \texttt{L}(?) \quad o_4: \texttt{L}(?) \\ o_2 \searrow\!\swarrow o_3 \quad o_2 \searrow\!\swarrow o_4 \quad o_3 \searrow\!\swarrow o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\texttt{p},\texttt{n}\}} \end{array}} \quad E$$

- Known integers transformed to themselves

0

# Transforming values to terms

$$\frac{03 \mid \mathtt{x}: o_2, \mathtt{r}: 0 \mid o_2}{\begin{array}{l} o_2\!: \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\ o_3\!: \mathtt{L}(?) \qquad o_4\!: \mathtt{L}(?) \\ o_2 \searrow\!\!\swarrow o_3 \qquad o_2 \searrow\!\!\swarrow o_4 \qquad o_3 \searrow\!\!\swarrow o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \quad E$$

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

$o_3, o_4 \quad 0$

# Transforming values to terms

$$\begin{array}{l}
\texttt{03} \mid \texttt{x}: o_2, \texttt{r}: 0 \mid o_2 \\
\hline
o_2{:}\, \texttt{L}(\texttt{p} = o_3, \texttt{n} = o_4) \\
o_3{:}\, \texttt{L}(?) \qquad o_4{:}\, \texttt{L}(?) \\
o_2 \searrow\!\swarrow o_3 \qquad o_2 \searrow\!\swarrow o_4 \qquad o_3 \searrow\!\swarrow o_4 \\
o_2, o_3, o_4 \circlearrowleft_{\{\texttt{p},\texttt{n}\}}
\end{array} \quad E$$

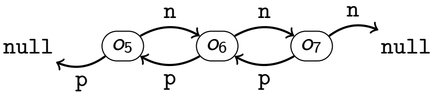
- Known integers transformed to themselves
- Unknown values transformed to variables
- Data structures transformed to nested constructor terms:
  Class `Cl` with $n$ fields $\curvearrowright$ symbol Cl of arity $n$

$$\overbrace{\mathsf{L}(o_3, o_4)}^{o_2}\ 0$$

$$\frac{03 \mid \mathtt{x}: o_2, \mathtt{r}: 0 \mid o_2}{\begin{array}{l} o_2{:}\,\mathtt{L}(\mathtt{p}=o_3, \mathtt{n}=o_4) \\ o_3{:}\,\mathtt{L}(?) \quad o_4{:}\,\mathtt{L}(?) \\ o_2 \searrow\!\swarrow o_3 \quad o_2 \searrow\!\swarrow o_4 \quad o_3 \searrow\!\swarrow o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \end{array}} \quad E$$

- Known integers transformed to themselves
- Unknown values transformed to variables
- Data structures transformed to nested constructor terms: Class `Cl` with $n$ fields $\curvearrowright$ symbol Cl of arity $n$
- Encoding cycles: Special symbol $\bigcirc$ for repetition

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




Encoding of $o_5$: $\mathsf{L}(\mathsf{null}, \mathsf{L}(\bigcirc, \mathsf{L}(\bigcirc, \mathsf{null})))$
Encoding of $o_6$: $\mathsf{L}(\mathsf{L}(\mathsf{null}, \bigcirc), \mathsf{L}(\bigcirc, \mathsf{null}))$

# Transforming states to terms

$$\begin{array}{l}
03 \mid \mathtt{x} : o_2, \mathtt{r} : 0 \mid o_2 \\
\hline
o_2 : \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4) \\
o_3 : \mathtt{L}(?) \quad o_4 : \mathtt{L}(?) \\
o_2 \searrow\!\!\swarrow o_3 \quad o_2 \searrow\!\!\swarrow o_4 \quad o_3 \searrow\!\!\swarrow o_4 \\
o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}
\end{array} \quad E$$

- State $s$ term encoding:
  - Root symbol ($\equiv$ program position) $\mathsf{f}_s$
  - All local variables, stack entries as arguments

$$\mathsf{f}_E(\overbrace{\mathsf{L}(o_3, o_4)}^{o_2}, 0, \overbrace{\mathsf{L}(o_3, o_4)}^{o_2})$$

# Transforming edges to rules

$$\begin{array}{|l|}\hline 03 \mid \mathtt{x}: o_2, \mathtt{r}: 0 \mid o_2 \\ \hline o_2{:}\,\mathtt{L}(\mathtt{p}=o_3, \mathtt{n}=o_4) \\ o_3{:}\,\mathtt{L}(?) \quad o_4{:}\,\mathtt{L}(?) \\ o_2 \searrow\!\!\swarrow o_3 \quad o_2 \searrow\!\!\swarrow o_4 \quad o_3 \searrow\!\!\swarrow o_4 \\ o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \\ \hline \end{array}\ E \cdots\cdots\rightarrow \begin{array}{|l|}\hline 11 \mid \mathtt{x}: o_4, \mathtt{r}: 0 \mid \varepsilon \\ \hline o_4 : \mathtt{L}(?) \\ o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \\ \hline \end{array}\ F$$

- State $s$ term encoding:
  - Root symbol ($\equiv$ program position) $\mathsf{f}_s$
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$

$$\mathsf{f}_E(\overbrace{\mathsf{L}(o_3, o_4)}^{o_2}, 0, \overbrace{\mathsf{L}(o_3, o_4)}^{o_2}) \quad \rightarrow \quad \mathsf{f}_F(o_4\ , 0)$$

# Transforming edges to rules

$\texttt{03} \mid \texttt{x}: o_2, \texttt{r}: 0 \mid o_2$

$o_2: \texttt{L}(\texttt{p} = o_3, \texttt{n} = o_4)$

$o_3: \texttt{L}(?) \quad o_4: \texttt{L}(?)$

$o_2 \searrow\!\!\swarrow o_3 \quad o_2 \searrow\!\!\swarrow o_4 \quad o_3 \searrow\!\!\swarrow o_4$

$o_2, o_3, o_4 \circlearrowleft_{\{\texttt{p},\texttt{n}\}}$

$E$

$\texttt{11} \mid \texttt{x}: o_4, \texttt{r}: 0 \mid \varepsilon$

$o_4 : \texttt{L}(?)$

$o_4 \circlearrowleft_{\{\texttt{p},\texttt{n}\}}$

$F$

- 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 $\curvearrowright$ free var on rhs

$$f_E(\overbrace{L(o_3, o_4)}^{o_2}, 0, \overbrace{L(o_3, o_4)}^{o_2}) \quad \rightarrow \quad f_F(o_4{}', 0)$$

# Transforming edges to rules

$03 \mid \mathtt{x}: o_2, \mathtt{r}: 0 \mid o_2$

$o_2 : \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4)$

$o_3 : \mathtt{L}(?) \quad o_4 : \mathtt{L}(?)$

$o_2 \mathrel{\backslash\!\!/\!/} o_3 \quad o_2 \mathrel{\backslash\!\!/\!/} o_4 \quad o_3 \mathrel{\backslash\!\!/\!/} o_4$

$o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

$E$ $\dashrightarrow$ $F$

$11 \mid \mathtt{x}: o_4, \mathtt{r}: 0 \mid \varepsilon$

$o_4 : \mathtt{L}(?)$

$o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

- State $s$ term encoding:
  - Root symbol ($\equiv$ program position) $\mathsf{f}_s$
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
  - **Problem**: Cycle encoding changes $\curvearrowright$ free var on rhs
  - **Solution**: Filter: Only encode non-cyclic parts!

$$\mathsf{f}_E(\overbrace{\mathsf{L}(\quad o_4)}^{o_2}, 0, \overbrace{\mathsf{L}(\quad o_4)}^{o_2}) \quad \rightarrow \quad \mathsf{f}_F(o_4\ , 0)$$

# Transforming edges to rules

$C$

$03 \mid \mathtt{x}: o_1, \mathtt{r}: 0 \mid o_1$

$o_1 : \mathtt{L}(?)$

$o_1 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

$\longrightarrow$

$E$

$03 \mid \mathtt{x}: o_2, \mathtt{r}: 0 \mid o_2$

$o_2 : \mathtt{L}(\mathtt{p} = o_3, \mathtt{n} = o_4)$

$o_3 : \mathtt{L}(?) \quad o_4 : \mathtt{L}(?)$

$o_2 \curlyvee o_3 \quad o_2 \curlyvee o_4 \quad o_3 \curlyvee o_4$

$o_2, o_3, o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}}$

- State $s$ term encoding:
  - Root symbol ($\equiv$ program position) $\mathsf{f}_s$
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
  - **Problem**: Cycle encoding changes $\curvearrowright$ free var on rhs
  - **Solution**: Filter: Only encode non-cyclic parts!
- Refinement edges: Encode target state twice, relabel

$$\mathsf{f}_C(\mathsf{L}(o_4), 0, \mathsf{L}(o_4)) \rightarrow \mathsf{f}_E(\mathsf{L}(o_4), 0, \mathsf{L}(o_4))$$

# Transforming edges to rules

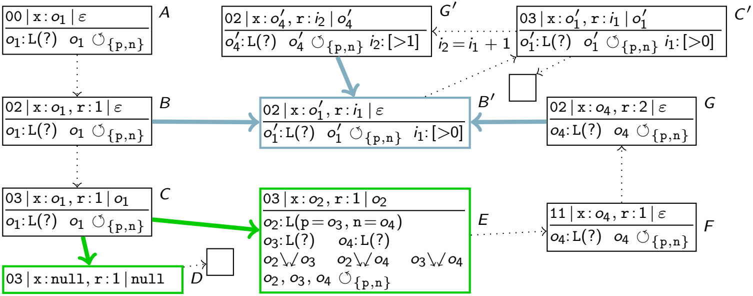
$$\begin{array}{|l|}\hline 02 \mid \mathtt{x}: o_4, \mathtt{r}: 1 \mid \varepsilon \\ \hline o_4 \;:\; \mathtt{L}(?) \\ o_4 \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \\ \hline \end{array} \quad G \xrightarrow{\qquad\qquad} \begin{array}{|l|}\hline 02 \mid \mathtt{x}: o_1', \mathtt{r}: i_1 \mid \varepsilon \\ \hline o_1'\!:\mathtt{L}(?) \quad o_1' \circlearrowleft_{\{\mathtt{p},\mathtt{n}\}} \quad i_1\!: [\geq 0] \\ \hline \end{array} \quad B'$$

- State $s$ term encoding:
  - Root symbol ($\equiv$ program position) $\mathsf{f}_s$
  - All local variables, stack entries as arguments
- Evaluation edges: Encode states, put in $\rightarrow$
  - **Problem**: Cycle encoding changes $\curvearrowright$ 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

$$\mathsf{f}_G(o_4, 2) \;\rightarrow\; \mathsf{f}_{B'}(o_4, 2)$$

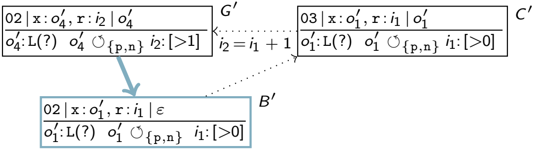
# The example TRS

```
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
```



# The example TRS

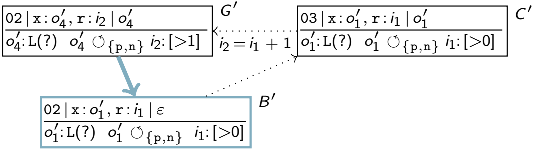
```
00: iconst_1
01: istore_1
02: aload_0
03: ifnull 17
06: aload_0
07: getfield n
10: astore_0
11: iinc 1, 1
14: goto 2
17: iload_1
18: ireturn
```




1. Only consider SCCs!

# 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
```




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

$$\mathsf{f}_{B'}(\mathsf{L}(o'_4), i_1) \quad \to \quad \mathsf{f}_{B'}(o'_4, i_1 + 1)$$

# 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]

# 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]

# 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

# 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*

| | **Yes** | **No** | **Fail** | **Run** (s) |
|---|---|---|---|---|
| AProVE | 289 | 125 | 27 | 16.6 |
| Julia | 205 | 79 | 157 | 6.5 |
| COSTA | 163 | 0 | 278 | 13.1 |

# Experimental results

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

| | **Yes** | **No** | **Fail** | **Run** (s) |
|---|---|---|---|---|
| AProVE | 289 | 125 | 27 | 16.6 |
| Julia | 205 | 79 | 157 | 6.5 |
| COSTA | 163 | 0 | 278 | 13.1 |

- Won Termination Competition 2012/2013

# Experimental results

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

| | **Yes** | **No** | **Fail** | **Run** (s) |
|---|---|---|---|---|
| AProVE | 289 | 125 | 27 | 16.6 |
| Julia | 205 | 79 | 157 | 6.5 |
| COSTA | 163 | 0 | 278 | 13.1 |

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

# Experimental results

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

| | **Yes** | **No** | **Fail** | **Run** (s) |
|---|---|---|---|---|
| AProVE | 289 | 125 | 27 | 16.6 |
| Julia | 205 | 79 | 157 | 6.5 |
| COSTA | 163 | 0 | 278 | 13.1 |

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

**Specialized abstract domains:**

- **easy to automate**
- **very effective**

# Termination Analysis: Invariants and Rank Functions

**Example**

$$
\begin{array}{l}
\mathrm{y} := 1; \\
\textbf{while } \mathrm{x} > 0 \textbf{ do} \\
\quad \mathrm{x} := \mathrm{x} - \mathrm{y}; \\
\quad \mathrm{y} := \mathrm{y} + 1; \\
\textbf{done}
\end{array}
$$

- Invariant $y > 0$ and rank function $x$ prove termination
- How do we know that we need $y > 0$? $\curvearrowright$ $x$ requires it

# Termination Analysis: Invariants and Rank Functions

**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 $y > 0$? $\curvearrowright$ $x$ requires it
- How do we know that $x$ is a RF? $\curvearrowright$ $y > 0$ proves it

# Termination by iterative strengthening: Idea

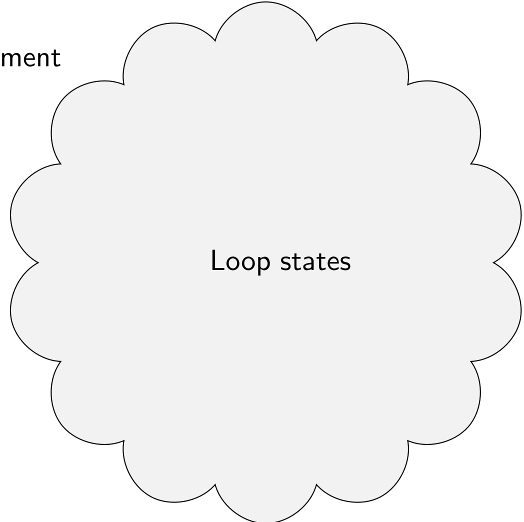
1. Safety: Provide samples (Counterexamples)
2. Rank tool: Find specific termination argument
3. Safety: Prove generality, or 1

# Termination by iterative strengthening: Idea

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

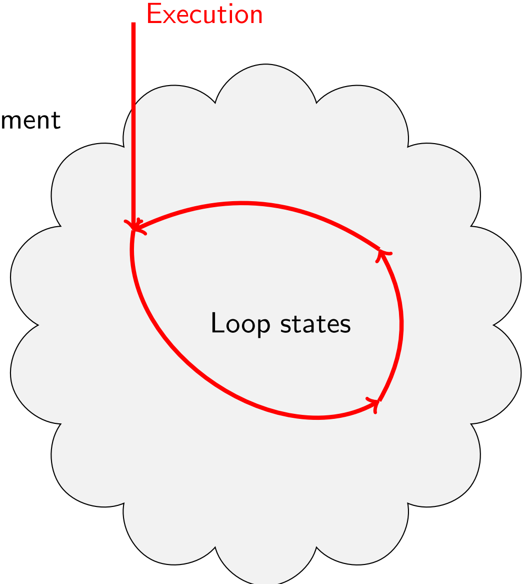
# Termination by iterative strengthening

Find counterexample
then strengthen argument



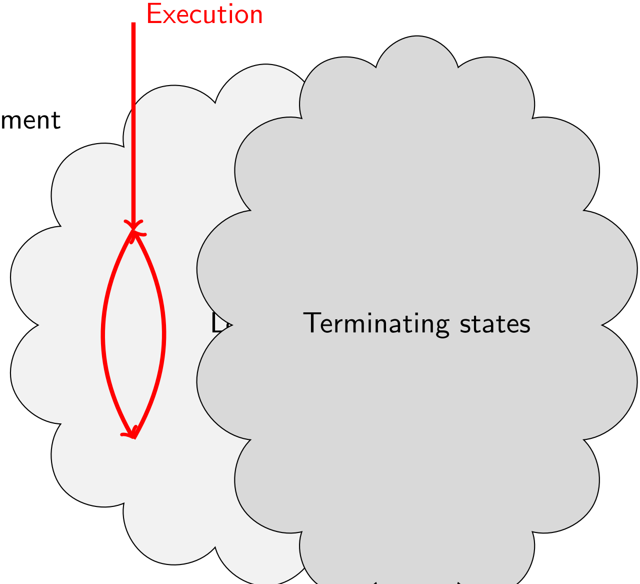
# Termination by iterative strengthening

Find counterexample
then strengthen argument



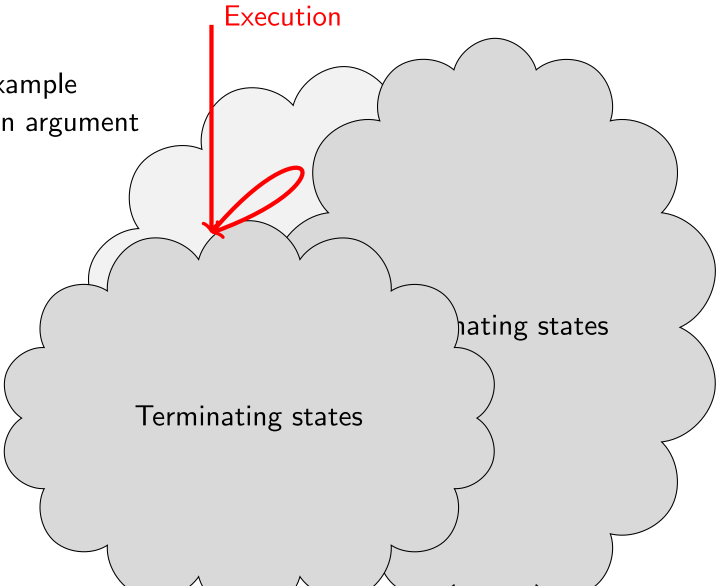
# Termination by iterative strengthening

Find counterexample
then strengthen argument



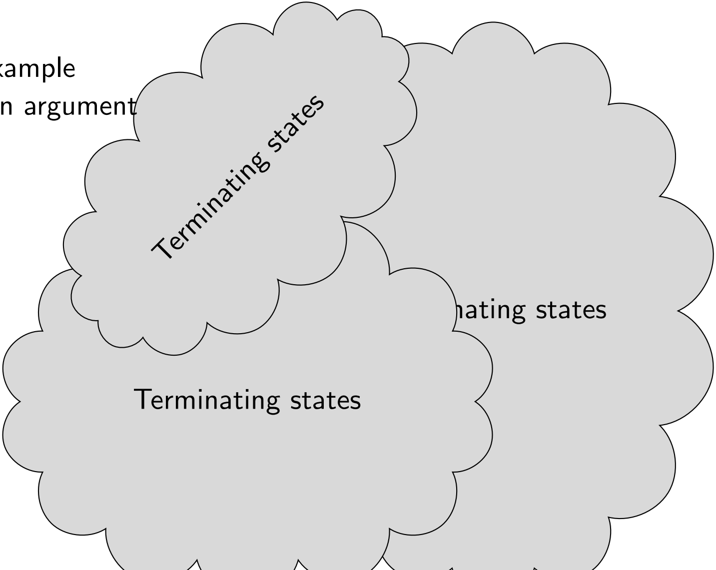
# Termination by iterative strengthening

Find counterexample
then strengthen argument



# Termination by iterative strengthening

Find counterexample
then strengthen argument



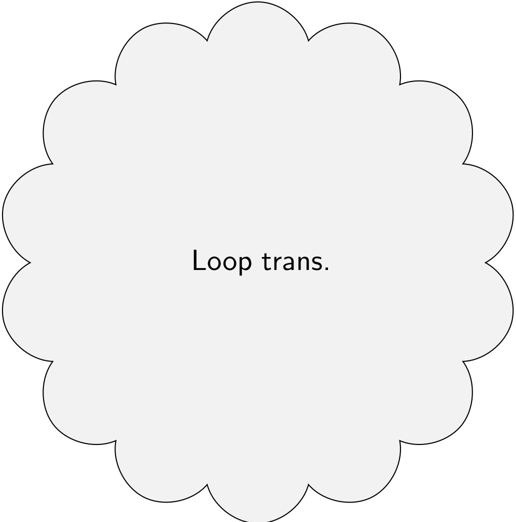
# Termination by iterative strengthening: Worst case

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

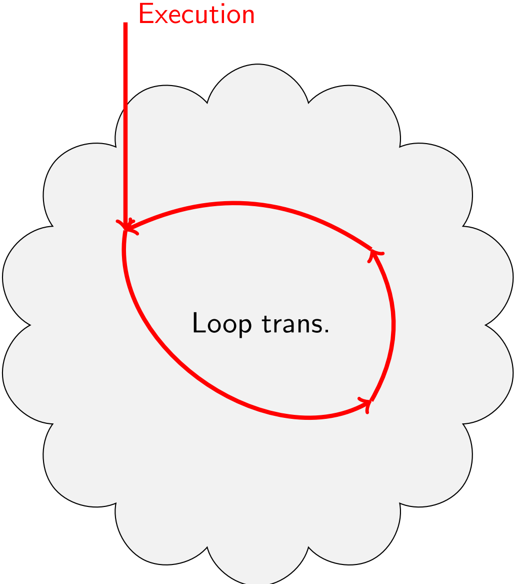
# Termination by iterative strengthening: Worst case

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

# Termination by iterative simplification

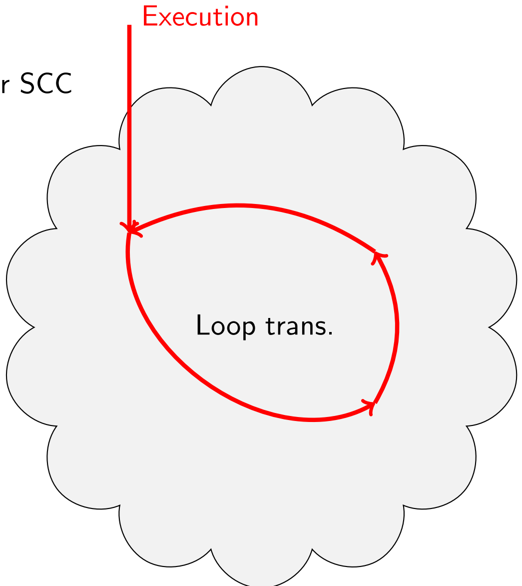


# Termination by iterative simplification



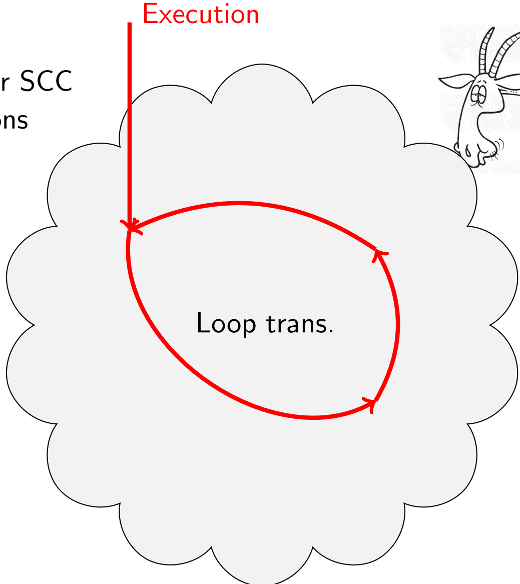
# Termination by iterative simplification

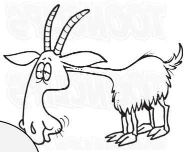
Find rank function for SCC



# Termination by iterative simplification

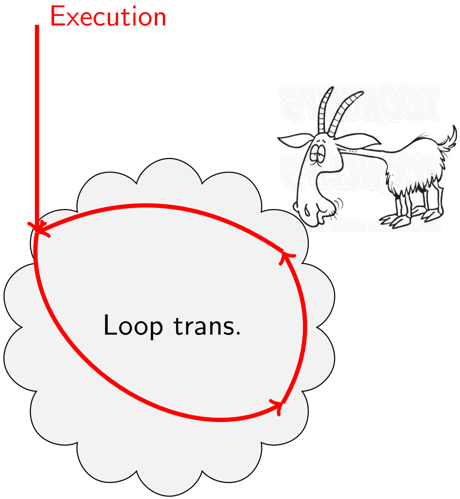
Find rank function for SCC
then remove transitions



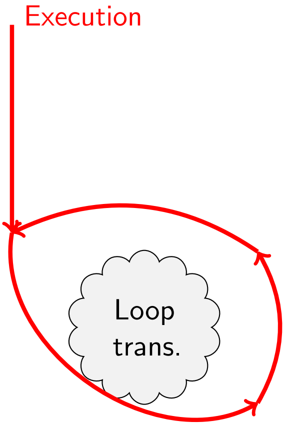
# Termination by iterative simplification

Find rank function for SCC
then remove transitions



# Termination by iterative simplification

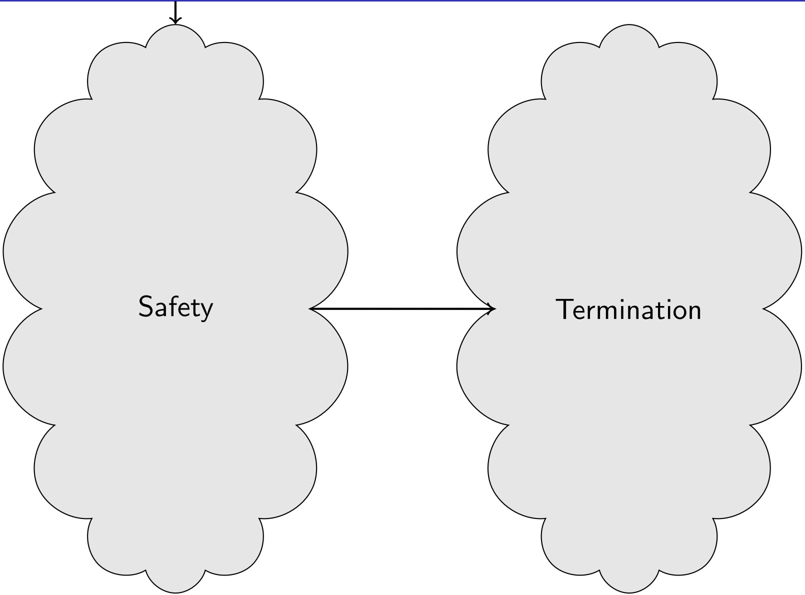
Find rank function for SCC
then remove transitions



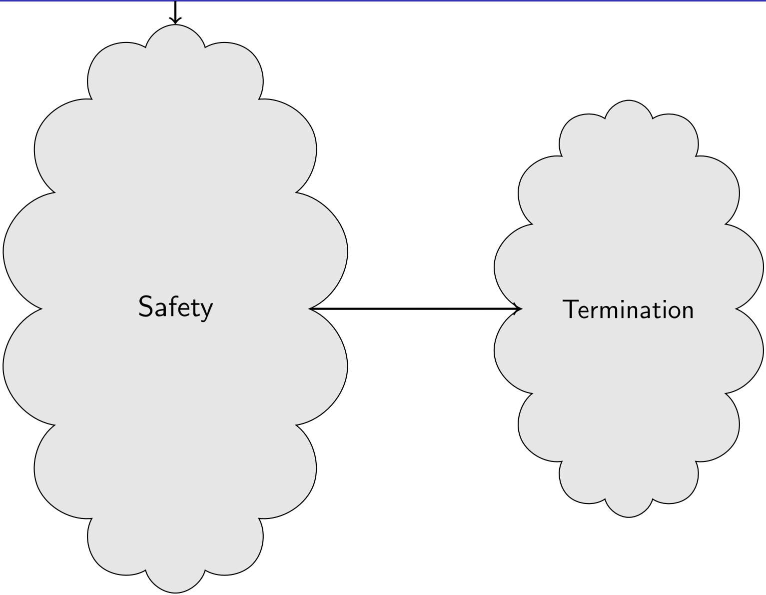
# Termination by cooperation

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

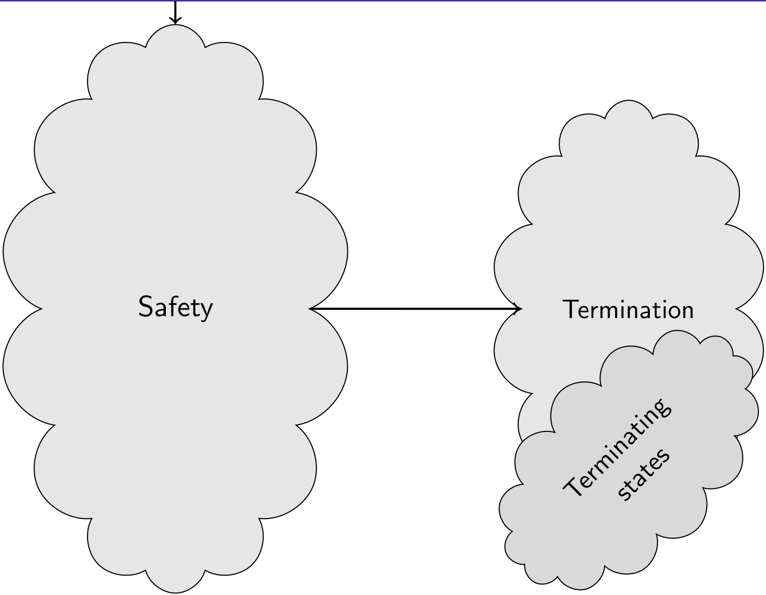
# Cooperation: High-level view



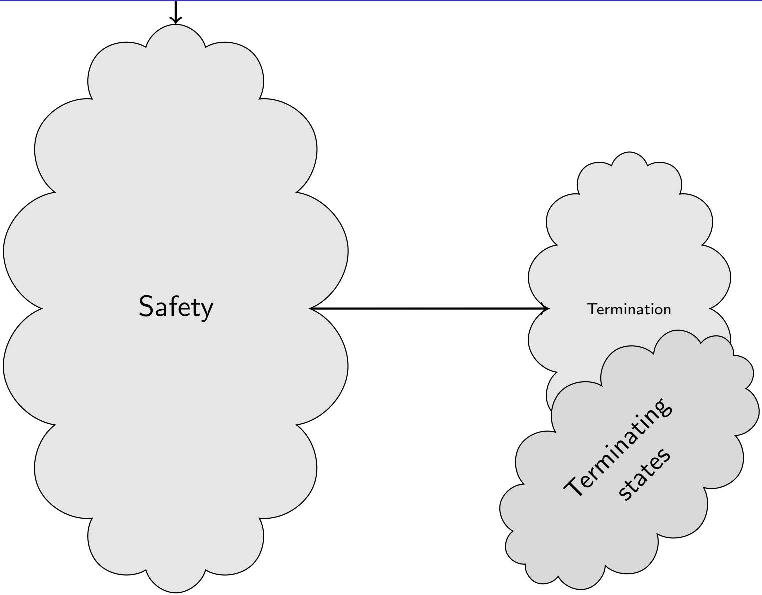
# Cooperation: High-level view



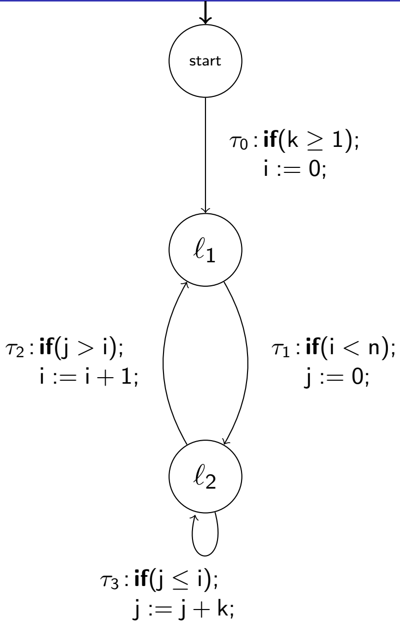
# Cooperation: High-level view



# Cooperation: High-level view



# Cooperation: High-level view



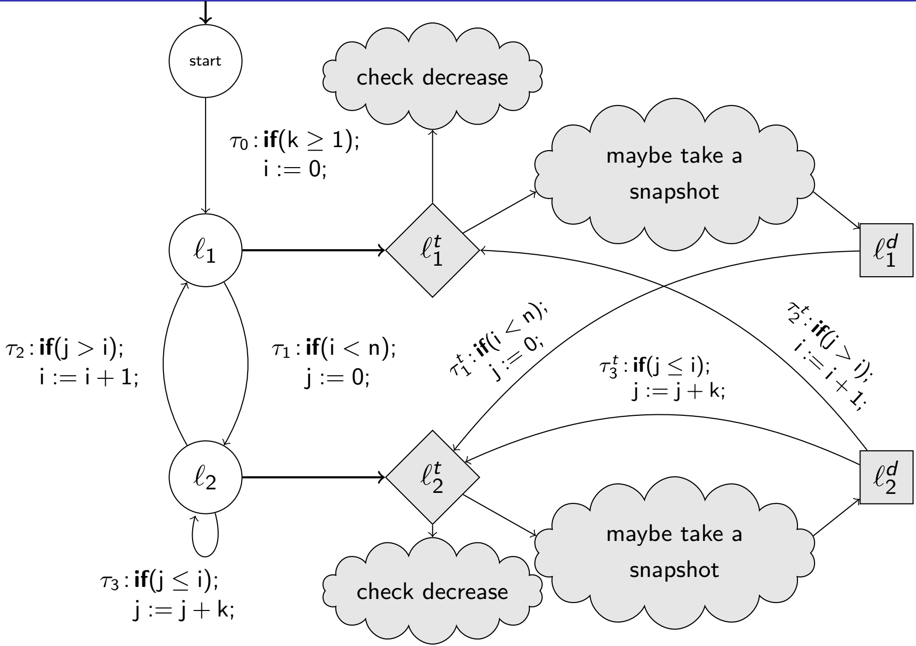

**Example (Source)**

**if** $k \geq 1$ **then**
  $i := 0$;
$\ell_1$ **while** $i < n$ **do**
    $j := 0$;

$\ell_2$ **while** $j \leq i$ **do**
      $j := j + k$;
    **done**

    $i := i + 1$;
  **done**
**fi**

# Cooperation: High-level view



# Cooperation

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

# Cooperation

Intuition:

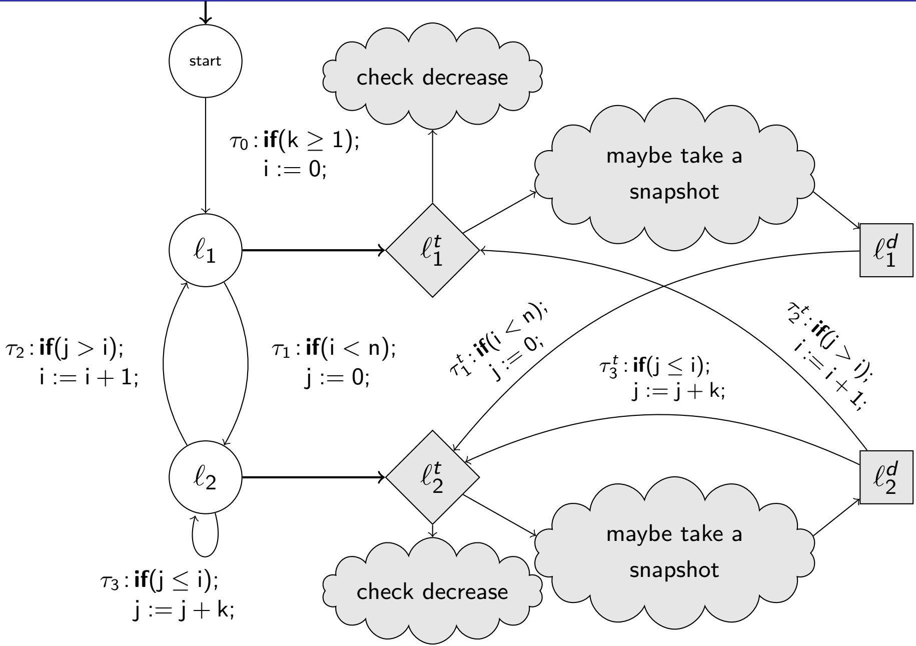
- **Safety subgraph**: original program
- **Termination subgraph**: instrumented copy

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

Approach:

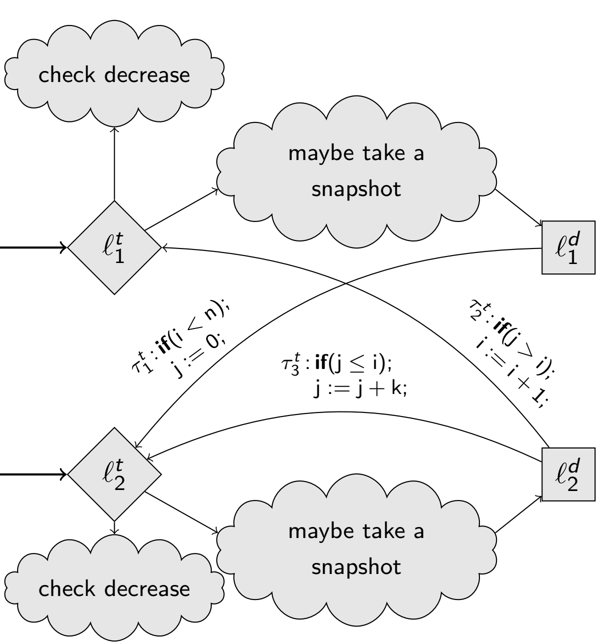
- Analyze whole SCC, not counterexample slice
- Remove transitions after proof

# Cooperation: Simplification



# Cooperation: Simplification

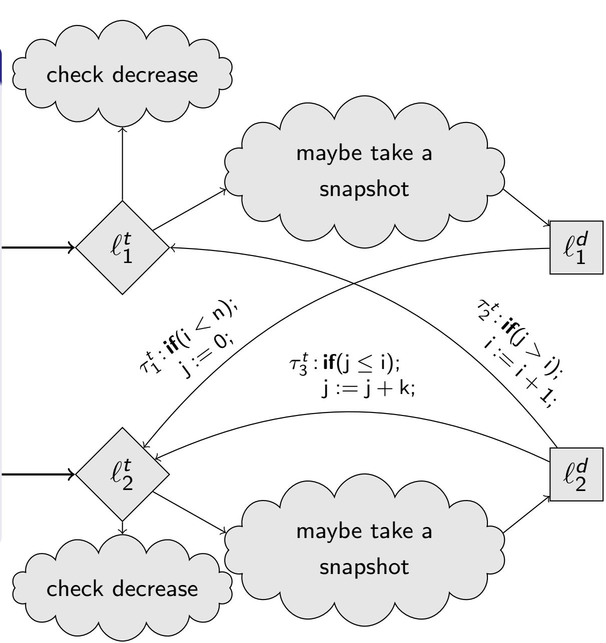
## Simplification



# Cooperation: Simplification

## Simplification

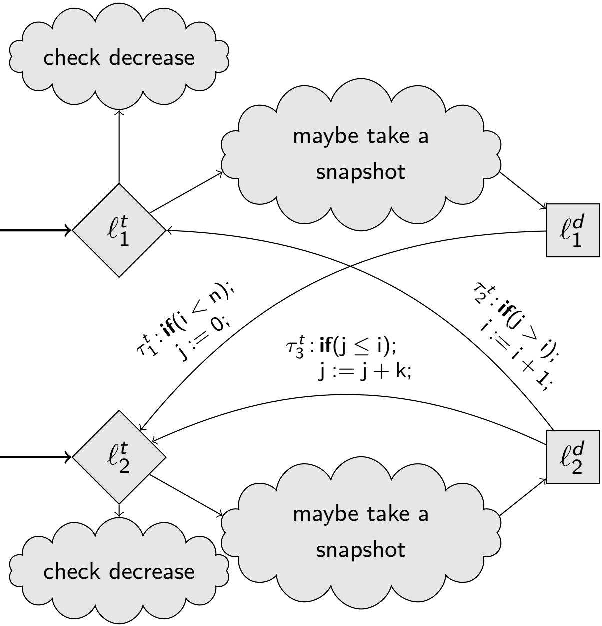
1. Find SCC $\mathcal{S}$ in termination graph:
   $$\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$$



# Cooperation: Simplification

## Simplification

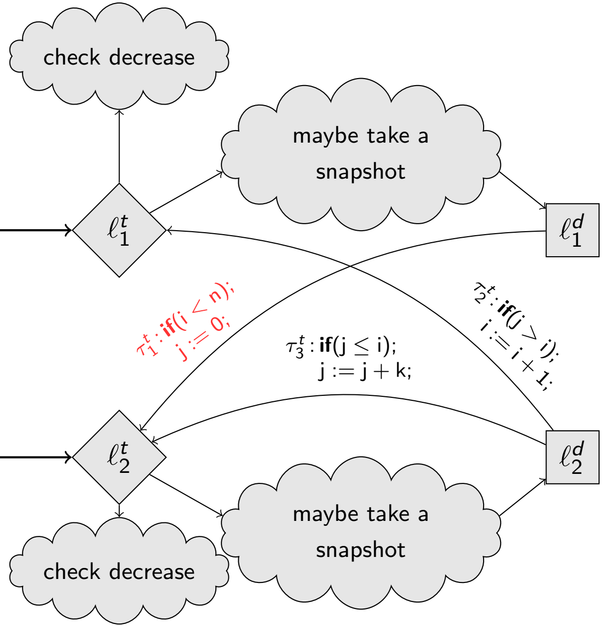
1. Find SCC $\mathcal{S}$ in termination graph:
$$\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$$
2. Find $\mathcal{S}$-orienting RF:
$$f^1_{\ell_1^t}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i} + 1$$
$$f^1_{\ell_1^d}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i} + 1$$
$$f^1_{\ell_2^t}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i}$$
$$f^1_{\ell_2^d}(\mathsf{i},\mathsf{j},\mathsf{k},\mathsf{n}) = \mathsf{n} - \mathsf{i}$$



# Cooperation: Simplification

## Simplification

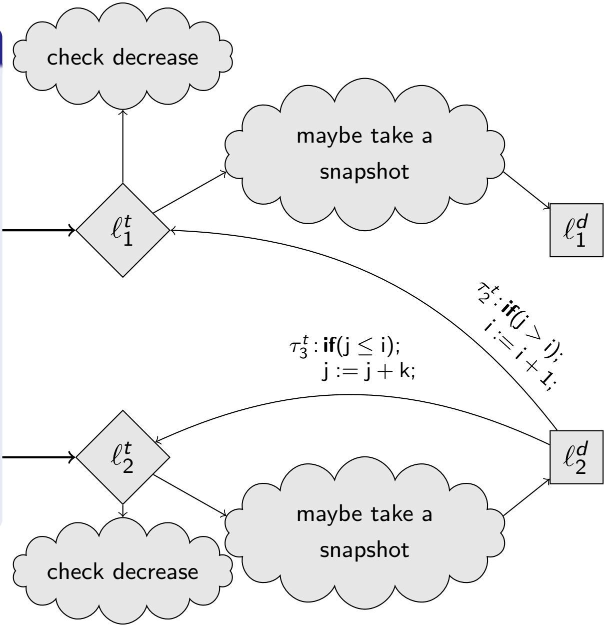
1. Find SCC $\mathcal{S}$ in termination graph:
   $$\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$$
2. Find $\mathcal{S}$-orienting RF:
   $$f^1_{\ell_1^t}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i} + 1$$
   $$f^1_{\ell_1^d}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i} + 1$$
   $$f^1_{\ell_2^t}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i}$$
   $$f^1_{\ell_2^d}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i}$$
3. Delete decr./bounded



# Cooperation: Simplification

**Simplification**

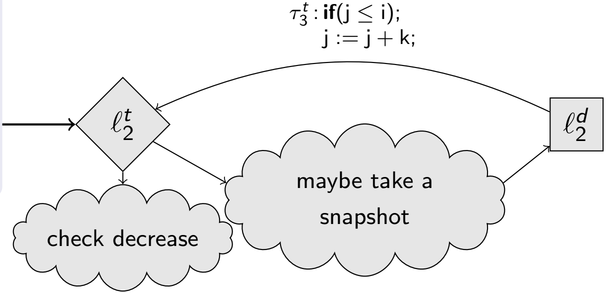
1. Find SCC $\mathcal{S}$ in termination graph:
$$\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$$
2. Find $\mathcal{S}$-orienting RF:
$$f^1_{\ell_1^t}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i} + 1$$
$$f^1_{\ell_1^d}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i} + 1$$
$$f^1_{\ell_2^t}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i}$$
$$f^1_{\ell_2^d}(\mathsf{i,j,k,n}) = \mathsf{n} - \mathsf{i}$$
3. Delete decr./bounded



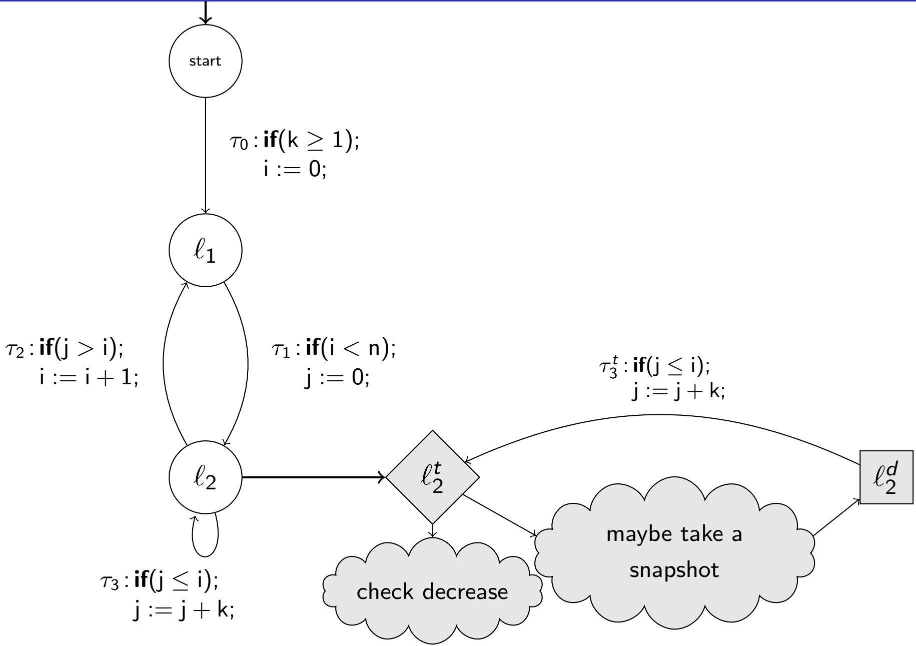
# Cooperation: Simplification

## Simplification

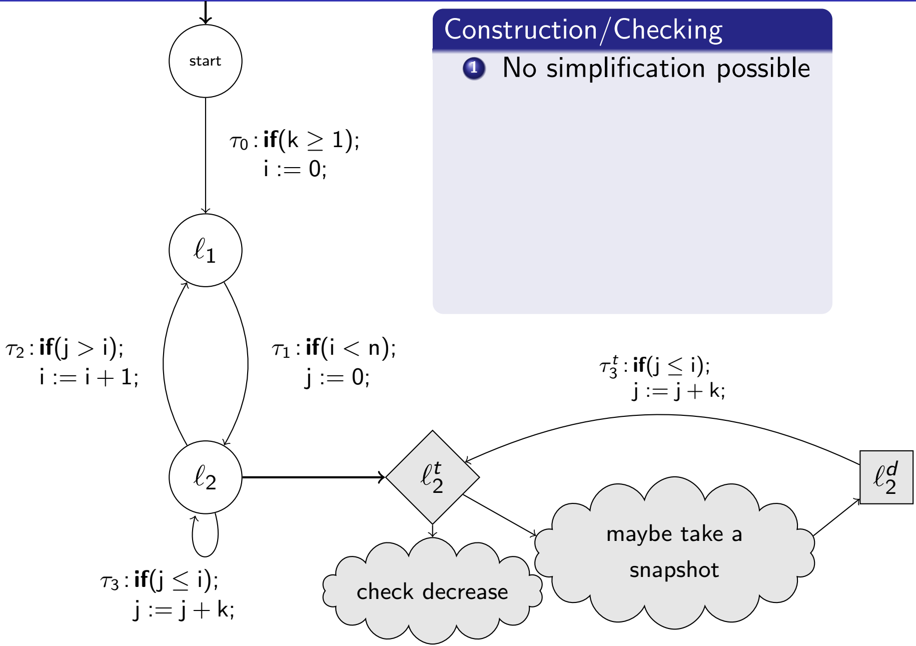
1. Find SCC $\mathcal{S}$ in termination graph:
$$\ell_1^t, \ell_1^d, \ell_2^t, \ell_2^d$$
2. Find $\mathcal{S}$-orienting RF:
$$f^1_{\ell_1^t}(\mathrm{i,j,k,n}) = \mathrm{n} - \mathrm{i} + 1$$
$$f^1_{\ell_1^d}(\mathrm{i,j,k,n}) = \mathrm{n} - \mathrm{i} + 1$$
$$f^1_{\ell_2^t}(\mathrm{i,j,k,n}) = \mathrm{n} - \mathrm{i}$$
$$f^1_{\ell_2^d}(\mathrm{i,j,k,n}) = \mathrm{n} - \mathrm{i}$$
3. Delete decr./bounded
4. Clean up



# Cooperation



# Cooperation: Invariants




## Construction/Checking

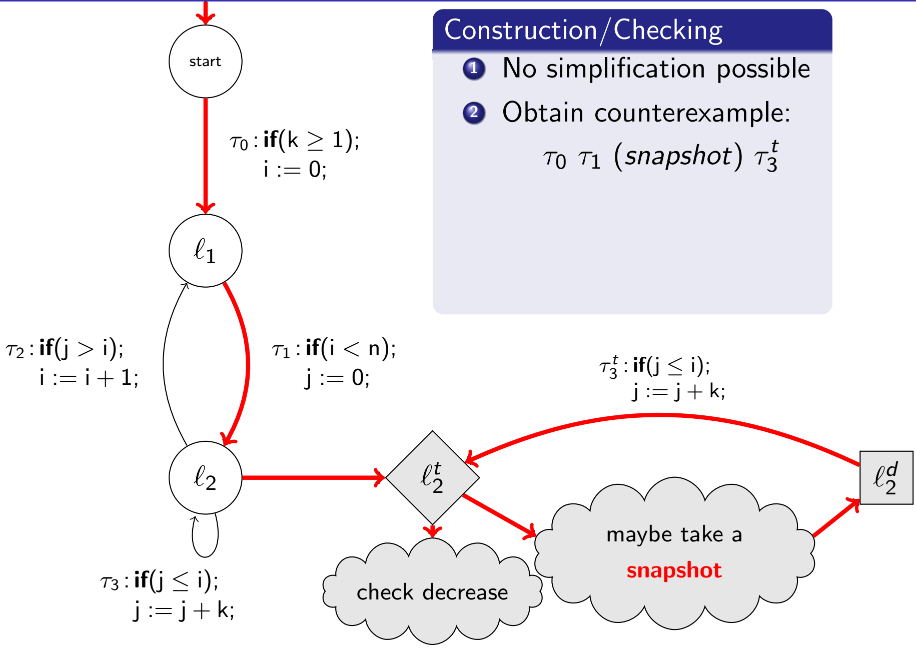
1. No simplification possible

# Cooperation: Invariants

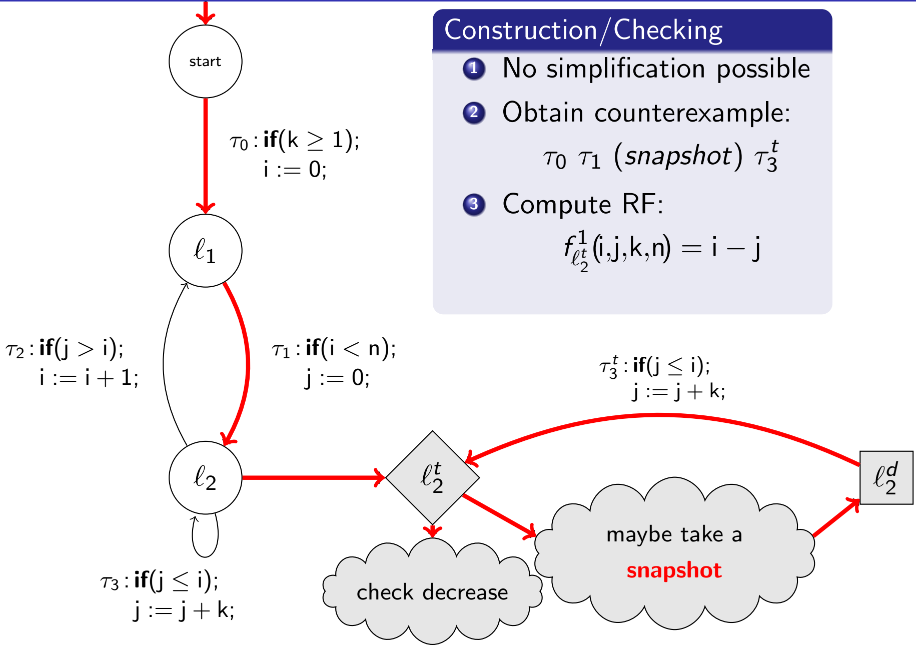
## Construction/Checking

1. No simplification possible
2. Obtain counterexample:

$$\tau_0 \ \tau_1 \ (\textit{snapshot}) \ \tau_3^t$$



# Cooperation: Invariants




## Construction/Checking

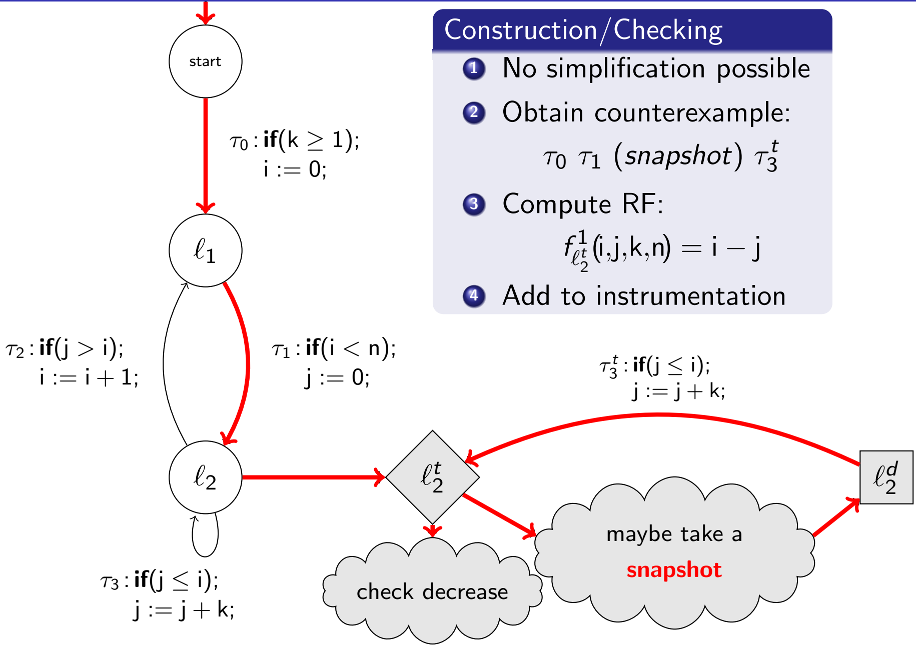
1. No simplification possible
2. Obtain counterexample:

   $$\tau_0 \ \tau_1 \ (\textit{snapshot}) \ \tau_3^t$$

3. Compute RF:

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

# Cooperation: Invariants




## Construction/Checking

1. No simplification possible
2. Obtain counterexample:

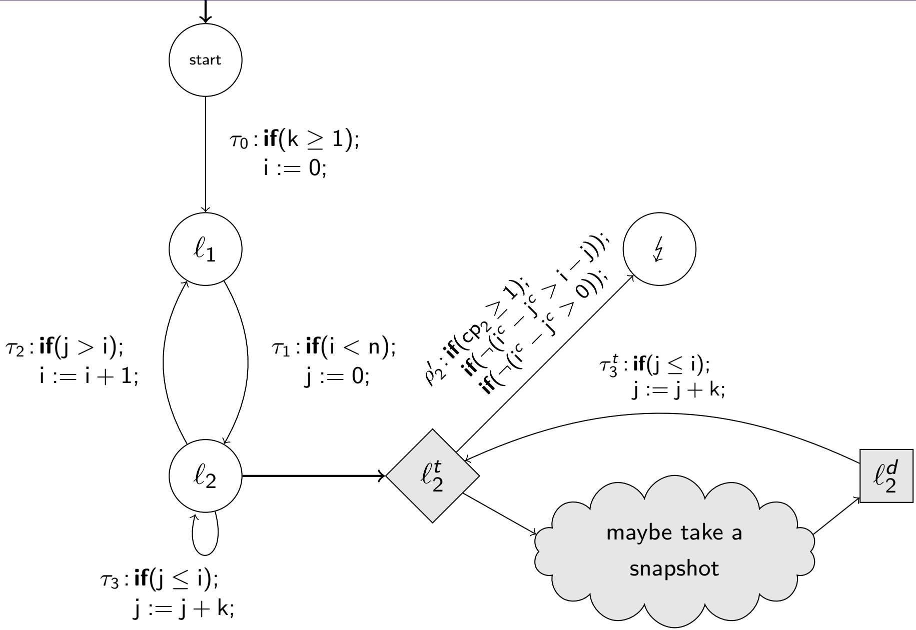
$$\tau_0\ \tau_1\ (\mathit{snapshot})\ \tau_3^t$$

3. Compute RF:

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

4. Add to instrumentation

# Cooperation: Invariants



# Cooperation: Evaluation

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, ...

| | Term (#) | Term (avg. s) |
|---|---|---|
| Cooperating-T2 | 245 | 3.42 |
| AProVE | 197 | 2.21 |
| KITTeL | 196 | 4.65 |
| T2 | 189 | 5.15 |
| AProVE+Interproc | 185 | 1.53 |
| Terminator | 177 | 4.99 |
| Size-Change/MCNP | 156 | 17.50 |
| ARMC | 138 | 16.16 |

# Cooperation: Evaluation

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

| | Term (#) | Term (avg. s) |
|---|---|---|
| Cooperating-T2 | 245 | 3.42 |
| AProVE | 197 | 2.21 |
| KITTeL | 196 | 4.65 |
| T2 | 189 | 5.15 |
| AProVE+Interproc | 185 | 1.53 |
| Terminator | 177 | 4.99 |
| Size-Change/MCNP | 156 | 17.50 |
| ARMC | 138 | 16.16 |

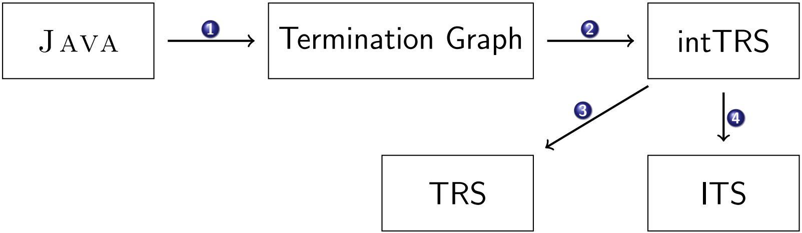
Sources available: `http://research.microsoft.com/en-us/projects/t2/`

# Program termination: Our approach

$$\boxed{\text{JAVA}} \xrightarrow{❶} \boxed{\text{Termination Graph}} \xrightarrow{❷} \boxed{\text{intTRS}}$$

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"

# 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

# 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