Program termination · Lecture 2

Berkeley · Spring ’09

Byron Cook
Review from the previous lecture

- Program termination = WF transition relation

- Proving WF can be reduced to finding a larger \textit{ranking relation}

- Accurate transition relations often too hard to compute
  - Supporting invariants needed to establish termination

- Unions of WF-relations not WF, but transitive closure can be used to offset the problem
Review from the previous lecture

- We can use variables with finite range to decompose termination proofs (e.g. the program counter)

- Linear ranking function synthesis is decidable
  - But linear ranking functions are often not enough..........
Review from the previous lecture

→ Finding linear ranking functions (for relations with only linear updates and conditions) is decidable

→ Not all WF linear relations have linear ranking functions, e.g.
  - $R \triangleq x > 0 \land x' = x - y \land y' = y + 1$
  - $R \triangleq x \geq 0 \land x' = -2x + 10$
  - Ackermann’s function
  - ............
Overview

- Notes on a representation for programs
- Checking termination arguments
- Refining termination arguments
- Induction
- Termination analysis
\[
[x := e]_V \triangleq \{(s, t) \mid \forall v \in V - \{x\}. \ s(v) = t(v) \land t(x) = e\}
\]
\[ [x := e]_V \triangleq \{(s, t) \mid \forall v \in V - \{x\}. s(v) = t(v) \land t(x) = e\} \]

\[ [x := *]_V \triangleq \{(s, t) \mid \forall v \in V - \{x\}. s(v) = t(v)\} \]
\)[x := e]_V \triangleq \{(s, t) \mid \forall v \in V - \{x\}. \ s(v) = t(v) \land t(x) = e\}

\)[x := \ast]_V \triangleq \{(s, t) \mid \forall v \in V - \{x\}. \ s(v) = t(v)\}

\)[\text{assume}(e)]_V \triangleq \{(s, t) \mid \forall v \in V. \ s(v) = t(v) \land e[s]\}
Programs

- Programs are rooted cyclic graphs where edges are annotated with finite command sequences.

- A “cutpoint set” can be computed by uniquely numbering the nodes and marking each node that can transition to a node that’s lower in the order.
The meaning of the program is a relation constructed from the graph and commands.

A special variable (not used in the program) pc is used to track program location.

Paths are sequences of pc valuations, traces are sequences of commands drawn from paths.
The *relational meaning* of a finite path/trace segment is the relational composition of the meaning of each command.

\[
\llbracket c_1; c_2; \ldots \rrbracket_{V, \rho} \triangleq \llbracket c_1 \rrbracket_{V, \rho} ; \llbracket c_2 \rrbracket_{V, \rho} ; \ldots
\]
The relational statement is the conjunction of each command:

\[ [c_1; c_2; \ldots] \forall x_i \exists x_0, x_1, y_0.
\]

\[
\begin{align*}
(s, t) &\text{ s.t. } \exists x_0, x_1, y_0. \\
\wedge &\left\{ \\
\quad s(x) &= x_0 \\
\quad s(y) &= y_0 \\
\quad x_1 &= x_0 + y_0 \\
\quad x_1 &> 0 \\
\quad t(x) &= x_1 \\
\quad t(y) &= y_0
\right\}
\end{align*}
\]

\[ x := x + y; \]
\[ \text{assume}(x > 0); \]
The relational statement is the conjunction of each command:

\[
\forall c_1, c_2, \ldots \left[ c_1; c_2; \ldots \right] \psi,
\]

\[
\frac{\text{x := x + y;}}{
\text{assume}(x > 0);}
\]

\[
(s, t) \text{ s.t. } \exists x_0, x_1, y_0.
\left\{
\begin{array}{l}
  s(x) = x_0 \\
  s(y) = y_0 \\
  x_1 = x_0 + y_0 \\
  x_1 > 0 \\
  t(x) = x_1 \\
  t(y) = y_0
\end{array}
\right\}
\]
Overview

→ Notes on a representation for programs

→ Checking termination arguments

→ Refining termination arguments

→ Induction

→ Termination analysis
Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R_I^+)_{pc=k} \subseteq Q \iff \neg \text{REACHABLE}_{\boxplus}(p,k,Q)(l_{err})$$
Transformations to reachability supporting closure

Defining $\square$ such that

$$(R_{\star I}^+)_{pc=k} \subseteq Q \iff \neg \text{REACHABLE}_{\square}(p,k,Q)(l_{err})$$

Tools like SLAM, BLAST, IMPACT, F-Soft, etc.
Transformations to reachability supporting closure

Defining $\square$ such that

$$(R_{\star I}^+)_{pc=k} \subseteq Q \text{ iff } \neg \text{REACHABLE}_{\square}(p,k,Q)(l_{err})$$
Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R_{*1}^{+})_{pc=k} \subseteq Q \text{ iff } \neg \text{REACHABLE}_{\boxplus}(p,k,Q)(l_{err})$$
Transformations to reachability supporting closure

Defining \( \boxplus \) such that

\[
(R^+_I)_{pc=k} \subseteq Q \iff \neg \text{REACHABLE}_{\boxplus}(p,k,Q)(l_{err})
\]

Using decomposition strategy we prove termination
Definition. Assume $[\mathcal{P}] = (I, R, S)$. A reachability engine $\text{REACHABLE}_{\mathcal{P}}(k)$ returns false when

$$R^*(I) \cap [pc = k] = \emptyset$$

If $\text{REACHABLE}_{\mathcal{P}}(k) = \text{true}$, then a witness path from $0$ to $k$ is returned.
Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R_+^I)_p^{c=k} \subseteq Q \iff \neg \text{REACHABLE}_{\boxplus}(p,k,Q)(l_{err})$$
Defining $\forall$ such that

$$(R_I^+)_{|_{pc=k}} \subseteq Q \iff \neg \text{REACHABLE}_{\forall}(p,k,Q)(l_{err})$$
Defining $\boxplus$ such that

$$(R_+^1|_{P_{\text{pc}=k}} \subseteq Q \iff \neg \text{REACHABLE}_{\boxplus}(p,k,Q)(l_{\text{err}}))$$
Transformations to reachability supporting closure

Defining $\boxplus$ such that

$$(R^\downarrow_{I}^+_{pc=k}) \subseteq Q \text{ iff } \neg \text{REACHABLE}_{\boxplus}(p,k,Q)(l_{err})$$

\[\text{Diagram:}\]

$S$

$\bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet$
Defining $\boxplus$ such that

$$(R_{\mathcal{I}^{+}}|_{\text{pc}=k}) \subseteq Q \iff \neg \text{REACHABLE}_{\boxplus}(p, k, Q)(l_{\text{err}})$$
Defining \( \boxplus \) such that

\[
(R_{I}^+_{\downarrow I})_{pc=k} \subseteq Q \iff \neg \text{REACHABLE}_{\boxplus}(p,k,Q)(l_{err})
\]

\( s \xrightarrow{\delta} x \xrightarrow{+1} \text{assume}(x > 0) \xrightarrow{y := y + 1} \text{assume}(k > 0) \xrightarrow{\text{skip}} \) 

Is \((s, t) \in \emptyset\) ?
Transformations to reachability supporting closure

$$\Pi_{x := x + 1 ; \text{assume}(x > 0)} ; \ldots \Pi_{\forall R(\pm)} \leq \emptyset ?$$

\[ (R'|_I^+) \mid_{\forall I} \iff \text{REACHABLE}_{\forall}(P, k, Q)(l_{err}) \]

\[
S \quad x := x + 1 \quad \text{assume}(x > 0) \quad y := y + 1 \quad \text{assume}(k > 0) \quad \text{skip} \quad \downarrow
\]

\[ (s, t) \in \emptyset ? \]
Transformations to reachability supporting closure

```
copied = 0;
.
.
.
while(*) {
    k: loop body
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
            .
            .
            copied = 1;
        }
    }
    } else {
        assert(\varnothing);
    }
}
Transformations to reachability supporting closure

while(*) {
    \textit{k: loop body}
}

Transformations to reachability supporting closure

copied = 0;
  .
  .
  .

while(*) {
  if (copied==0) {
    if (*) {
      if (*) {
        old_x = x;
        old_y = y;
        .
        .
        copied = 1;
      }
    } else {
      assert(0);
    }
    .
    loop body
  }
}
Transformations to reachability supporting closure

\[ \text{assert}(e) \quad \Delta \quad \text{if} \ (\neg e) \ \{ \ \text{loop body} \ \}
\]

\[ \text{while}(*) \{
\]
\[ \quad \text{if} \ (\text{copied}==0) \{
\]
\[ \quad \quad \text{if} \ (*) \{
\]
\[ \quad \quad \quad \text{old}_x = x;
\]
\[ \quad \quad \quad \text{old}_y = y;
\]
\[ \quad \quad \}
\]
\[ \quad \quad \text{copied} = 1;
\]
\[ \quad \}
\]
\[ \quad \text{else} \{
\]
\[ \quad \quad \text{assert}(\emptyset);
\]
\[ \}
\]

\[ \}
\]
copied = 0;
    
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
            
            copied = 1;
        }
        } else {
            assert(0);
        }
    }
    }

loop body
while(*) { 
    if (copied==0) { 
        if (*) { 
            old_x = x; 
            old_y = y; 
            copied = 1; 
        } else { 
            assert(0); 
        } 
    } else { 
        assert(0); 
    } 
}
Transformations to reachability supporting closure

```c
while(*) {
    if (copied) {
        if (*) {
            old_x = x;
            old_y = ...;
        }
        copied = 1;
    } else {
        assert(false);
    }
}
```

old_x > x
\land
old_x \geq 0

loop body
Termination Proofs for Systems Code

Byron Cook
Microsoft Research
bycook@microsoft.com

Andreas Podelski
Max-Planck-Institut für Informatik
podelski@mpi-sb.mpg.de

Andrey Rybalchenko
Max-Planck-Institut für Informatik and
EPFL
rybal@mpi-sb.mpg.de and
andrey.rybalchenko@epfl.ch

Abstract

Program termination is central to the process of ensuring that systems code can always react. We describe a new program termination prover that performs a path-sensitive and context-sensitive program analysis and provides capacity for large program fragments (i.e., more than 20,000 lines of code) together with support for programming language features such as arbitrarily nested loops, pointers, function-pointers, side-effects, etc. We also present experimental results on device driver dispatch routines from the Windows operating system. The most distinguishing aspect of our tool is how it shifts the balance between the two tasks of constructing and respectively checking the termination argument. Checking becomes the hard step. In this paper we show how we solve the corresponding challenge of checking with binary reachability analysis.

Categories and Subject Descriptors
D.2.4 [Software]: Software Engineering—Program Verification; D.4.5 [Software]: Operating Systems—Reliability

request packet and FdcData->TopOfStack is the pointer to another serial-based device driver). In the case where the other device driver returns a return-value that indicates success, but places 0 in PioStatusBlock->Information, the serial enumeration driver will fail to increment the value pointed to by nActual (line 66), possibly causing the driver to infinitely execute this loop and not return to its calling context. The consequence of this error is that the computer’s serial devices could become non-responsive. Worse yet, depending on what actions the other device driver takes, this loop may cause repeated acquiring and releasing of kernel resources (memory, locks, etc) at high priority and excessive physical bus activity. This extra work stresses the operating system, the other drivers, and the user applications running on the system, which may cause them to crash or become non-responsive too.

This example demonstrates how a notion of termination is central to the process of ensuring that reactive systems can always react. Until now no automatic termination tool has ever been able to provide a capacity for large program fragments (>70,000 lines).
Nesting of loops allows us to isolate pieces of the program

- When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

- When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop
Nesting of loops allows us to isolate pieces of the program

→ When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

→ When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop
Isolation

Nesting of loops allows us to isolate pieces of the program

→ When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

→ When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop

`while(x<y) {
  k = nondet();
  while(k>0) {
    x--;
  }
  x++;
}`

Comes for free
Isolation

Nesting of loops allows us to isolate pieces of the program

→ When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop

→ When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop
Nesting of loops allows us to isolate pieces of the program

- When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop.

- When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop.

We don't support this yet......
Nesting of loops allows us to isolate pieces of the program

- When proving well-foundedness of cutpoints in inner loops, we can ignore non-termination of the enclosing loop.

- When proving well-foundedness of cutpoints in outer loops, we can ignore non-termination of the inner loop.

We don’t support this yet......
Transformations to reachability supporting closure

```c
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
            .
            .
            copied = 1;
        }
    } else {
        assert(0);
    }
}

loop body

copied = 0;
```
Transformations to reachability supporting closure

```c
while(*) {
    if (copied == 0) {
        // Decomposition story now more complex .......
    }
    copied = 0;
}
```
Transformations to reachability supporting closure

"l is not visited infinitely often so long as the context is not entered infinitely often"
Overview

- Notes on a representation for programs
- Checking termination arguments
- Refining termination arguments
- Induction
- Termination analysis
**Definition.** SYNTHESIS : $\langle S \leftrightarrow S \rangle \rightarrow (S \rightarrow \mathbb{N})$ is a partial function such that $R \subseteq \nabla_{\text{SYNTHESIS}(R)}$ when SYNTHESIS($R$) is defined.
\( T := \emptyset \)

\textbf{while} \( \text{REACHABLE}_{\Psi}(\mathcal{P}, \ell, T)(\ell_{err}) \) \textbf{do}

\hspace{1em} let \( \pi_s, \pi_c = \text{lasso in } \Psi(\mathcal{P}, \ell, T) \) from 0 to \( \ell \), and \( \ell \) to \( \ell_{err} \)

\hspace{1em} let \( \rho = \alpha([\pi_c]*/([\pi_s]*/)) \)

\hspace{1em} if \( \text{SYNTHESIS}([\pi_c]_\rho) \) returns ranking relation \( f \) then

\hspace{2em} \( T := T \cup \geq_f \)

\hspace{1em} \textbf{else}

\hspace{2em} \textbf{report} "potential counterexample found: } \pi_s, \pi_c"

\hspace{1em} \textbf{fi}

\textbf{od}

\textbf{report} "termination proved with argument } T"
$\rho \supseteq \mathcal{P}_c^* (\mathcal{P}_s)$

$T := \emptyset$

while REACH

let $\pi_s, \pi_c \in \mathcal{P}(P, \emptyset, \emptyset)$ from 0 to $l$, and $l$ to $l_{err}$

let $\rho = \alpha(\mathcal{P}_c^* (\mathcal{P}_s))$

if SYNTHESIS($\mathcal{P}_c\rho$) returns ranking relation $f$ then

$T := T \cup \geq_f$

else

report "potential counterexample found: $\pi_s, \pi_c$"

fi

od

report "termination proved with argument $T$"
$$\rho \supseteq [\pi_c]^*([\pi_s])$$

\[ T := \emptyset \]

\textbf{while REACH}\!

let \( \pi_s, \pi_c \) - also in \( \mathbf{R} \)

let \( \rho = \alpha([\pi_c]^*([\pi_s])) \)

\textbf{if SYNTHESIS}([\pi_c]_\rho)

\( T := T \cup \geq_f \)

\textbf{else}

\textbf{report} "potential counterexample found: \( \pi_s, \pi_c \)"

\textbf{fi}

\textbf{od}

\textbf{report} "termination proved with argument } T↖
Abstraction Refinement for Termination

Byron Cook\textsuperscript{1}, Andreas Podelski\textsuperscript{2}, and Andrey Rybalchenko\textsuperscript{2}

\textsuperscript{1} Microsoft Research, Cambridge
\textsuperscript{2} Max-Planck-Institut für Informatik, Saarbrücken

\textbf{Abstract}. Abstraction can often lead to spurious counterexamples. Counterexample-guided abstraction refinement is a method of strengthening abstractions based on the analysis of these spurious counterexamples. For invariance properties, a counterexample is a finite trace that violates the invariant; it is spurious if it is possible in the abstraction but not in the original system. When proving termination or other liveness properties of infinite-state systems, a useful notion of spurious counterexamples has remained an open problem. For this reason, no counterexample-guided abstraction refinement algorithm was known for termination. In this paper, we address this problem and present the first known automatic counterexample-guided abstraction refinement algorithm for termination proofs. We exploit recent results on transition invariants and transition predicate abstraction. We identify two reasons for spuriousness: abstractions that are too coarse, and candidate transition invariants that are too strong. Our counterexample-guided abstraction refinement algorithm successively weakens candidate transition invariants and refines the abstraction.
Refinement

```c
example1.c

12:   x = nondet();
13:   y = nondet();
14:
15:   while(x>0 && y>0) {
16:       if (nondet()) {
17:           y++;
18:           x--;
19:       } else {
20:           y--;
21:       }
22:
23:
24:       if (copied==0) {
25:           if (nondet()) {
26:               oldx = x;
27:               oldy = y;
28:               copied=1;
29:           } else {
30:               if (!0) {
31:                   SLIC_ERROR=0;
32:               }
33:           }
34:       }
35:
```
while(x>0 && y>0) {
    if (nondet()) {
        y++;
        x--;
    } else {
        y--;
    }
}
refinement
Refinement

```c
x = nonet();
y = nonet();
while(x>0 && y>0) {
    if (nonet()) {
        y++;
        x--;
    } else {
        y--;
    }
    if (copied==0) {
        if (nonet()) {
            oldx = x;
            oldy = y;
            copied=1;
        }
        while(x>0 && y>0) {
            y--;
            if (copied==0) {
                if (nonet()) {
                    oldx = x;
                    oldy = y;
                    copied=1;
                }
            } else {
                if (!0) {
                    SLIC_ERROR:0;
                }
            }
        }
    } else {
        if (!0) {
            SLIC_ERROR:0;
        }
    }
```
Refinement
\[ \pi_c \](oldx, oldy), (x, y) = \exists x_0, y_0, y_1.

\[ \land \begin{cases} 
\text{oldx} = x_0 \\
\text{oldy} = y_0 \\
x_0 > 0 \\
y_0 > 0 \\
y_1 = y_0 - 1 \\
x = x_0 \\
y = y_1
\end{cases} \]
Refinement

Proclusos ranking function \( y,0 \)

\[
\begin{align*}
\text{oldx} &= x_0 \\
\text{oldy} &= y_0 \\
x_0 &> 0 \\
y_0 &> 0 \\
y_1 &= y_0 - 1 \\
x &= x_0 \\
y &= y_1
\end{align*}
\]
\(( \text{oldx}, \text{oldy}) \), \((x, y) = \exists x_0, y_0, y_1.\)
\[(oldx, oldy), (x, y) = \exists x_0, y_0, y_1.\]
Refinement

ACU 1, AFU 1: G(t_y_1_1_1_2_0,0_0_1) = S(<L(y_1_1_1_2),0_0_29_2_0_0_0>), (<S(<L(y_1_1_1_2),0_0_29_2_0_0_0>_1)_1>_1>0), L(y_1_1_1_2) = <S(<L(y_1_1_1_2),0_0_29_2_0_0_0>_1)_1>_1>1, L(y_1_1_1_2) = S(<L(y_1_1_1_2),0_0_29_2_0_0_0>_1)_1>1

SLAM: iter-begin 9

AR: no new preds at iter 9
AR: calling constrain at iter 9
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp >bebop.out 2>&1

AR: bebop generated a path
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp >bebop.out 2>&1

AR: bebop generated a path
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp >bebop.out 2>&1

AR: bebop generated a path
AR: calling bebop bebop -o -t -c -m -u -l SLIC_ERROR -f slam.bp >bebop.out 2>&1

AR: bebop is raising completed
AR: watch_startup_end Completed
SLAM: watch_startup_end 0, iter 9
AR: saving preds
Program [ example.c ] passed property
Saving termination lemmas to "witness.tt"
Time: 218.408
void main()
{
    int x, y;

    x = nondet();
    y = nondet();

    while(x > 0 && y > 0) {
        if (nondet()) {
            y++;
            x--;
        } else {
            y--;
        }
    }
}
Cutpoint

```c
void main()
{
    int x, y;
    x = nondet();
    y = nondet();
    while(x>0 && y>0) {
        if (nondet()) {
            y++;
            x--;
        } else {
            y--;
        }
    }
}
```
Example

```c
unsigned int Ack(unsigned int x, unsigned int y)
{
    if (x > 0) {
        int n;
        if (y > 0) {
            y--;
            n = Ack(x, y);
        }
        else {
            n = 1;
        }
        x--;
        return Ack(x, n);
    }
    else {
        return y + 1;
    }
}

void main()
{
    int x = nondet();
    int y = nondet();
    Ack(x, y);
}
```
```c
unsigned int Ack(unsigned int x, unsigned int y)
{
    if (x>0) {
        int n;
        if (y>0) {
            y--;
        }
        else {
            n = Ack(x,y);
        }
        n = 1;
    } else {
        return Ack(x,n);
    }
    return y+1;
}

void main()
{
    int x = nondet();
    int y = nondet();
    Ack(x,y);
}
```
We luck out in this case, accurate support for recursion not needed.
Examples

unsigned int Ack(unsigned int x, unsigned int y) {
    if (x > 0) {
        int n;
        if (y > 0) {
            y--;
            n = Ack(x, y);
        } else {
            n = 1;
        }
    } else {
        //x--;
        return Ack(x, n);
    } else {
        return y + 1;
    }
}

void main()
{
    int x = nonet();
    int y = nonet();
    Ack(x, y);
}
unsigned int Ack(unsigned int x, unsigned int y) {
    if (x > 0) {
        int n;
        if (y > 0) {
            y--;
            n = Ack(x, y);
        } else {
            n = 1;
        }
    }
    //x--;
    return Ack(x, n);
    else {
        return y + 1;
    }
}
void main() {
    int x = nondet();
    int y = nondet();
    Ack(x, y);
}
Examples

```c
void main()
{
    int x, y;

    x = nondet();
    y = nondet();

    while (x > 0) {
        if (y < 0) {} else {
            x = x + y;
            y--;}
    }
}
```
Examples

Notice the trick.
Overview

→ Notes on a representation for programs

→ Checking termination arguments

→ Refining termination arguments

→ Induction

→ Termination analysis
\( T := \emptyset \)

\textbf{while} \texttt{Reachable}(\( R, \ell, T, \ell_{err} \)) \textbf{do}

\hspace{1em} let \( \pi_s, \pi_c = \text{lasso in } R, \ell, T \) from 0 to \( \ell \), and \( \ell \) to \( \ell_{err} \)

\hspace{1em} let \( \rho = \alpha([\pi_c]([\pi_s](T))) \)

\hspace{1em} if \texttt{Synthesis}(\([\pi_c] \cap \rho \times \rho\)) returns ranking function \( f \) then

\hspace{2em} \( T := T \cup \geq_f \)

\hspace{1em} \textbf{else}

\hspace{2em} report "potential counterexample found: \( \pi_s, \pi_c \)"

\hspace{1em} fi

\textbf{od}

\textbf{report} "termination proved with argument \( T \)"
\[ T := \emptyset \]

\begin{algorithm}
\textbf{while} \texttt{Reachable}(\texttt{Reachable}(R, \ell, T), \ell_{err}) \textbf{do}

\begin{itemize}
  \item \texttt{let} \( \pi_s, \pi_c = \text{lasso in } \texttt{Reachable}(R, \ell, T) \) \text{ from } 0 \text{ to } \ell, \text{ and } \ell \text{ to } \ell_{err} \)
  \item \texttt{let} \( \rho = \alpha([\pi_c]^*([\pi_s](T))) \)
  \item \texttt{if} \( \texttt{Synthesis}([\pi_c] \cap \rho \times \rho) \) \text{ returns ranking function } f \text{ then}
    \begin{align*}
      T &:= T \cup \geq_f \\
    \end{align*}
  \item \texttt{else}
    \begin{itemize}
      \item \texttt{report} "potential counterexample found: } \pi_s, \pi_c \)
    \end{itemize}
  \item \texttt{fi}
\end{itemize}
\textbf{od}

\texttt{report} "termination proved with argument } T \)
$T := \emptyset$

while Reachable($\emptyset$, $R$, $\ell$, $T$, $\ell_{err}$) do
    let $\pi_s, \pi_c = \text{lasso in } \emptyset(R, \ell, T) \text{ from } 0 \text{ to } \ell$, and $\ell$ to $\ell_{err}$
    let $\rho = \alpha([\pi_c]^*([\pi_s](T)))$
    if Synthesis([\pi_c] \cap \rho \times \rho) returns ranking function $f$ then
        $T := T \cup \geq_f$
    else
        report "potential counterexample found: $\pi_s, \pi_c$
    fi
od

report "termination proved with argument $T"
\[
T := \emptyset \\
\textbf{while} \ \text{Reachable}(\mathcal{R}, l, T), l_{err}) \ \textbf{do} \\
\quad \text{let} \ \pi_s, \pi_c = \text{lasso in } \mathcal{R}(R, l, T) \ \text{from} \ 0 \ \text{to} \ l, \ \text{and} \ l \ \text{to} \ l_{err} \\
\quad \text{let} \ \rho = \alpha([\pi_c]^*([\pi_s](T))) \\
\quad \text{if} \ \text{Synthesis}([\pi_c] \cap \rho \times \rho) \ \text{returns ranking function} \ f \ \text{then} \\
\quad \quad T := T \cup \geq_f \\
\quad \text{else} \\
\quad \quad \text{report} \ \text{"potential counterexample found: } \pi_s, \pi_c \\
\quad \text{fi} \\
\textbf{od} \\
\textbf{report} \ \text{"termination proved with argument } T
\[ T := \emptyset \]

\textbf{while} Reachable(\( R, \ell, T \), \( \ell_{err} \)) \textbf{do}

\textbf{let} \( \pi_s, \pi_c = \text{lasso in } R, \ell, T \) \text{ from } 0 \text{ to } \ell, \text{ and } \ell \text{ to } \ell_{err} \)

\textbf{let} \( \rho = \alpha(\[\pi_c\] \ast ([\pi_s](T))) \)

\textbf{if} Synthesis(\([\pi_c] \cap \rho \times \rho\)) \text{ returns ranking function } f \text{ then}

\[ T := T \cup \geq_f \]

\textbf{else}

\textbf{report} “potential counterexample found: } \pi_s, \pi_c \)

\textbf{fi}

\textbf{od}

\textbf{report} “termination proved with argument } T \)
$T := \emptyset$

while \text{REACHABLE}(\mathbb{R}(R, l, T), \ell_{err}) do

let $\pi_s, \pi_c = \text{lasso in } \mathbb{R}(R, l, T) \text{ from } 0 \text{ to } l$, and $l$ to $\ell_{err}$

let $\rho = \alpha([\pi_c]^*([\pi_s](T)))$

if \text{SYNTHESIS}([\pi_c] \cap \rho \times \rho) \text{ returns ranking function } f \text{ then}

$T := T \cup \succeq_f$

else

report "potential counterexample found: $\pi_s, \pi_c$"

fi

od

report "termination proved with argument $T$"
Induction check

\[ T := \emptyset \]

while \textsc{Reachable}(\mathbb{R}(R, l, T), l_{err}) do

let \( a, \pi_c = \text{lasso in } \mathbb{R}(R, l, T) \) from 0 to \( l \), and \( l \) to \( l_{err} \)

let \( \rho = \alpha([\pi_c]^*([\pi_s](T))) \)

if \textsc{Synthesis}([\pi_c] \cap \rho \times \rho) returns ranking function \( f \) then

\[ T := T \cup \geq_f \]

else

report "potential counterexample found: \( a, \pi_c \)"

fi

od

report "termination proved with argument \( T \)"
\[ T := \emptyset \]

while Reachable(\( R, l, T \), \( l_{err} \)) do

let \( \pi_a, \pi_c = \text{lasso in } R(l, T) \) from 0 to \( l \), and \( l \) to \( l_{err} \)

let \( \rho = \alpha([\pi_c] \cdot ([\pi_a](T))) \)

if Synthesis([\pi_c] \setminus \{\rho \times \rho\}) returns ranking function \( f \) then

\[ T := T \cup \geq_f \]

else

report "potential counterexample found: \( \pi_a, \pi_c \)"

fi

od

report "termination proved with argument \( T \)"

Could use the invariant at location \( l \)
Ranking Abstractions

Aziem Chawdhary\textsuperscript{1}, Byron Cook\textsuperscript{2}, Sumit Gulwani\textsuperscript{2}, Mooly Sagiv\textsuperscript{3}, and Hongseok Yang\textsuperscript{1}

\textsuperscript{1} Queen Mary, University of London
\textsuperscript{2} Microsoft Research
\textsuperscript{3} Tel Aviv University

Abstract. We propose an abstract interpretation algorithm for proving that a program terminates on all inputs. The algorithm uses a novel abstract domain which uses ranking relations to conservatively represent relations between intermediate program states. One of the attractive aspects of the algorithm is that it abstracts information that is usually not important for proving termination such as program invariants and yet it distinguishes between different reasons for termination which are not usually maintained in existing abstract domains. We have implemented a prototype of the algorithm and shown that in practice it is fast and precise.
The bad news

Bad news: lassos don’t always lead to progress ........
Bad news: lassos don’t always lead to progress ........

\[
R_I^+ \subseteq \bigg\uparrow f_1 \cup \bigg\uparrow f_2 \cup \ldots \cup \bigg\uparrow f_n
\]
Bad news: lassos don't always lead to progress ........

\[ R^+_I \subseteq \triangleright f_1 \cup \triangleright f_2 \cup \ldots \cup \triangleright f_n \cup \triangleright f_{n+1} \cup \ldots ? \]
Bad news: lassos don’t always lead to progress .......

```cpp
1 while(x>0) {
2   y = x;
3   while(y>0) {
4     y = y - 1;
5   }
6   x = x + 1;
7 }
```
The bad news

Bad news: lassos don't always lead to progress ..........

1     while(x>0) {
2         y = x;
3         while(y>0) {
4             y = y - 1;
5         }
6     x = x + 1;
7 }
Bad news: lassos don't always lead to progress .......

```c
while(x>0) {
    y = x;
    while(y>0) {
        y = y - 1;
    }
    x = x + 1;
}
```
Bad news: lassos don’t always lead to progress .......

1: \( x_0 > 0 \)
2: \( y_1 = x_0 \)
3: \( y_1 > 0 \)
4: \( y_2 = y_1 - 1 \)
3: \( y_2 > 0 \)
4: \( y_3 = y_2 - 1 \)
3: \( y_3 \leq 0 \)
6: \( x_1 = x_0 + 1 \)

1: while(x>0) {
2: \hspace{1em} y = x;
3: \hspace{1em} while(y>0) {
4: \hspace{2em} y = y - 1;
5: \hspace{1em} }
6: \hspace{1em} x = x + 1;
7: }

86
The bad news

Bad lead

$y_2 = 1$

$y_3 = 0$

1: while($x > 0$) {
2:     $y = x$;
3:     while($y > 0$) {
4:         $y = y - 1$;
5:     }
6:     $x = x + 1$;
7: }

1: $x_0 > 0$
2: $y_1 = x_0$
3: $y_1 > 0$
4: $y_2 = y_1 - 1$
3: $y_2 > 0$
4: $y_3 = y_2 - 1$
3: $y_3 \leq 0$
6: $x_1 = x_0 + 1$
The bad news

Bad lead always

\[ y_1 = 2 \]
\[ y_2 = 1 \]
\[ y_3 = 0 \]

1. while(x>0) {
2. \quad y = x;
3. \quad while(y>0) {
4. \quad \quad y = y - 1;
5. \quad }
6. \quad x = x + 1;
7. }

1. \quad x_0 > 0
2. \quad y_1 = x_0
3. \quad y_1 > 0
4. \quad y_2 = y_1 - 1
3. \quad y_2 > 0
4. \quad y_3 = y_2 - 1
3. \quad y_3 \leq 0
6. \quad x_1 = x_0 + 1
The bad news

Bad lead

\[ y_1 = 2 \quad x_0 = 2 \]
\[ y_2 = 1 \]
\[ y_3 = 0 \]

1. While (x > 0) {
2.     y = x;
3.         while (y > 0) {
4.             y = y - 1;
5.         }
6.     x = x + 1;
7. }

1. \( x_0 > 0 \)
2. \( y_1 = x_0 \)
3. \( y_1 > 0 \)
4. \( y_2 = y_1 - 1 \)
3. \( y_2 > 0 \)
4. \( y_3 = y_2 - 1 \)
3. \( y_3 \leq 0 \)
6. \( x_1 = x_0 + 1 \)
The bad news

Bad lead always

\[ y_1 = 2 \quad x_0 = 2 \quad x_1 = 3 \]
\[ y_2 = 1 \]
\[ y_3 = 0 \]

1: \( x_0 > 0 \)
2: \( y_1 = x_0 \)
3: \( y_1 > 0 \)
4: \( y_2 = y_1 - 1 \)
3: \( y_2 > 0 \)
4: \( y_3 = y_2 - 1 \)
3: \( y_3 \leq 0 \)
6: \( x_1 = x_0 + 1 \)

1 while(x>0) {
2 y = x;
3 while(y>0) {
4 y = y - 1;
5 }
6 x = x + 1;
7 }
The bad news

Bad lead always

\[
\begin{align*}
  y_1 &= 2 \\
  x_0 &= 2 \\
  x_1 &= 3 \\
  y_2 &= 1 \\
  y_3 &= 0
\end{align*}
\]

1. while(x>0) {
2.     y = x;
3.     while(y>0) {
4.         y = y - 1;
5.     }
6.     x = x + 1;
7. }

1: \ x_0 > 0 \\
2: \ y_1 = x_0 \\
3: \ y_1 > 0 \\
4: \ y_2 = y_1 - 1 \\
3: \ y_2 > 0 \\
4: \ y_3 = y_2 - 1 \\
3: \ y_3 \leq 0 \\
6: \ x_1 = x_0 + 1

Thus: trivially well founded
More bad news

Standard symbolic model checkers don't really like the validity checking task
More bad news

Standard symbolic model checkers don't really like the validity checking task.

\[ R_I^+ \subseteq \geq f_1 \cup \geq f_2 \cup \ldots \cup \geq f_n \]
More bad news

Standard
don't
task

SLAM usually doesn't terminate unless modified significantly

\[ R_I^+ \subseteq \bigcup \geq f_1 \cup \bigcup \geq f_2 \cup \ldots \cup \bigcup \geq f_n \]
Overview

→ Notes on a representation for programs
→ Checking termination arguments
→ Refining termination arguments
→ Induction
→ Termination analysis
Transformations to reachability supporting closure

copied = 0;
.
.
.
while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
        }
        copied = 1;
    } else {
        assert(0);
    }

    loop body
}
Transformations to reachability supporting closure

copied = 0;

while(*) {
    if (copied==0) {
        if (*) {
            if (*) {
                old_x = x;
                old_y = y;
                
                copied = 1;
            }
        } else {
            skip;  // Corrected
        }
    }
    loop body
}
Transformations to reachability supporting closure

```c
copied = 0;

while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
        }
        copied = 1;
    } else { skip; }
}
```

If the invariant here is disjunctively WF, then we have proved termination.

*loop body*
Transformations to reachability supporting closure

```c

```
copied = 0;
.
.
.

while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
            .
            copied = 1;
        }
    } else {
        skip;
    }
    .
    loop body
}
```
Transformations to reachability supporting closure

```c
	copied = 0;

assume (inv_C)

old_x = x
old_y = y
goto

while(*) {
    if (copied==0) {
        if (*) {
            old_x = x;
            old_y = y;
            ...
            copied = 1;
        }
    } else {
        skip;
    }

    loop body
```
Transformations to reachability supporting closure

```plaintext
assume (inv_c)
old_x = x
old_y = y
goto

copied = 0;

while(*) {

}

loop body
```
Transformations to reachability supporting closure

```
assume (invC)
old_x = x
old_y = y
goTo
```

```
copied = 0;

while(*) {
  skip;
}
```

```
loop body
```
Transformations to removing closing curly brackets:

\[ \text{old-}x \geq x+1 \]
\[ \text{old-}y \geq y \]
\[ x \geq 0 \]
\[ y \geq 0 \]

\[
\text{loop body}
\]

\[
\text{Skip;}
\]

\[
\text{Skip;}
\]

\[
\text{Skip;}
\]
Variance analysis
Variance analysis

Overapproximation of relation encoded in a state
Use off-the-shelf abstract interpretation techniques to compute inclusion.
Variance analysis

\[ R^*_{\text{dC}} \]

\[ f_1 \underline{f_2} \underline{u} \underline{n} \cdots \underline{u} \underline{n} \]
Variance Analyses From Invariance Analyses

Josh Berdine  
Microsoft Research  
jib@microsoft.com

Aziem Chawdhary  
Queen Mary, University of London  
aziem@dcs.qmul.ac.uk

Byron Cook  
Microsoft Research  
bycook@microsoft.com

Dino Distefano  
Queen Mary, University of London  
ddino@dcs.qmul.ac.uk

Peter O’Hearn  
Queen Mary, University of London  
ohearn@dcs.qmul.ac.uk

Abstract

An invariance assertion for a program location \( \ell \) is a statement that always holds at \( \ell \) during execution of the program. Program invariance analyses infer invariance assertions that can be useful when trying to prove safety properties. We use the term variance assertion to mean a statement that holds between any state at \( \ell \) and any previous state that was also at \( \ell \). This paper is concerned with the development of analyses for variance assertions and their application to program termination and liveness properties. We describe the user (or algorithm calling the invariance analysis) might try to refine the abstraction. For example, if the tool is based on abstract interpretation they may choose to improve the abstraction by delaying the widening operation [28], using dynamic partitioning [33], employing a different abstract domain, etc.

The aim of this paper is to develop an analogous set of tools for program termination and liveness: we introduce a class of tools called variance analyses which infer assertions, called variance assertions, that hold between any state at a location \( \ell \) and any...
01 \texttt{VarianceAnalysis}(P, L, I^#) \{
02 \hspace{1em} IAs := \texttt{InvarianceAnalysis}(P, I^#)
03 \hspace{1em} \textbf{foreach} \ l \in L \{
04 \hspace{2em} \texttt{LTPreds}[\ell] := \texttt{true}
05 \hspace{2em} O := \texttt{Isolate}(P, L, \ell)
06 \hspace{2em} \textbf{foreach} \ q \in IAs \text{ such that } \texttt{pc}(q) = \ell \{
07 \hspace{3em} VAs := \texttt{InvarianceAnalysis}(O, \text{Step}(O, \{\texttt{Seed}(q)\}))
08 \hspace{2em} \textbf{foreach} \ r \in VAs \{
09 \hspace{3em} \textbf{if} \ \texttt{pc}(r) = \ell \land \neg \texttt{WellFounded}(r) \{
10 \hspace{4em} \texttt{LTPreds}[\ell] := \texttt{false}
11 \hspace{3em} \}
12 \hspace{2em} \}
13 \hspace{1em} \}
14 \hspace{1em} \}
15 \hspace{1em} \textbf{return} \ \texttt{LTPreds}
16 \}
Variance analysis

- **Strategy:**
  - Use abstract interpretation techniques to compute (disjunctive) over-approximation
  - Check that the parts of the disjunction are well founded

- **Advantages:**
  - Can use existing abstract interpretation tools to compute overapproximation
  - Always terminates
  - Fast

- **Disadvantages:**
  - No counterexamples
  - Less accurate than refinement-based approach
  - Abstract domains (currently) not built for our application
    - Widening can be too aggressive
    - Redundant information kept
the smallest ordinal $\alpha$ such that $\pi$ admits a ranking function with values $< \alpha$ is the \textit{ranking height} of $\pi$. The following observation may be helpful in establishing termination.

\textbf{Lemma 1 (Covering Observation).} \textit{Any transitive relation covered by finitely many well-founded relations is well-founded.}

In other words, if relations $U_1, \ldots, U_n$ are well-founded and $R \subseteq U_1 \cup \cdots \cup U_n$ is a transitive relation, then $R$ is well-founded.

Apparently this observation was made independently a number of times, and each time it was related to the termination problem. As far as we know, the observation was made first by Alfons Geser in [1990, page 31]. A weaker form of the observation, in which the relations $U_i$ are required to be transitive, had been proposed as a question on the web by Geser, and he informed us that he received proofs of it from Jean-Pierre Jouannaud, Werner Nutt, Franz Baader, George McNulty, Thomas Streicher, and Dieter Hofbauer; see [Lescanne, discussion list, items 38-42] for all but the last two of these. Both of our two referees pointed out that the observation was made independently in [Lee et al. 2001]. One of them wrote that “the covering observation lies at the heart of” [Lee et al. 2001] where it “is used implicitly in Theorem 4.” The other referee pointed out that the covering observation was made independently in [Dershowitz et al. 2001] and in [Codish et al. 2003]; see [Bruynooghe et al. ] in this connection. Recently the covering observation was rediscovered in [Podolski and Rybakchenko 2004] and was used for proving termination in [Podelski and Rybakchenko 2005; Cook et al. 2006; Berdine et al. 2007]. A stronger version of the covering observation, using a hypothesis that is weaker (but more complicated) than transitivity of $R$, was given in [Doornbos and Von Karger 1998].

The covering observation is proved by a straightforward application of the infinite version of Ramsey’s theorem. The transitivity of $R$ is essential here. If $a, b$ are distinct elements then the relation $\{(a,b),(b,a)\}$ is covered by the well-founded relations $\{(a,b)\}$ and $\{(b,a)\}$ but is not well-founded.

\textit{Example 2.} Let $\pi_1$ be the program
the smallest ordinal \( \alpha \) such that \( \pi \) admits a ranking function with values \( < \alpha \) is the \textit{ranking height} of \( \pi \). The following observation may be helpful in establishing termination.

**Lemma 1 (Covering Observation).** Any transitive relation covered by finitely many well-founded relations is well-founded.

In other words, if relations \( U_1, \ldots, U_n \) are well-founded and \( R \subseteq U_1 \cup \cdots \cup U_n \) is a transitive relation, then \( R \) is well-founded.

Apparently this observation was made independently a number of times, and each time it was related to the termination problem. As far as we know, the observation was made first by Alfons Geser in [1990, page 31]. A weaker form of the observation, in which the relations \( U_i \) are required to be transitive, had been proposed as a question on the web by Geser, and he informed us that he received proofs of it from Jean-Pierre Jouannaud, Werner Nutt, Franz Baader, George McNulty, Thomas Streicher, and Dieter Hofbauer; see [Lescanne, discussion list, items 38-42] for all but the last two of these. Both of our two referees pointed out that the observation was made independently in [Lee et al. 2001]. One of them wrote that “the covering observation lies at the heart of” [Lee et al. 2001] where it “is used implicitly in Theorem 4.” The other referee pointed out that the covering observation was made independently in [Dershowitz et al. 2001] and in [Codish et al. 2003]; see [Bruynooghe et al. 2007] in this connection. Recently the covering observation was rediscovered in [Podelski and Rybakchenko 2004] and was used for proving termination in [Podelski and Rybakchenko 2005; Cook et al. 2006; Berdine et al. 2007]. A stronger version of the covering observation, using a hypothesis that is weaker (but more complicated) than transitivity of \( R \), was given in [Doornbos and Von Karger 1998].

The covering observation is proved by a straightforward application of the infinite version of Ramsey’s theorem. The transitivity of \( R \) is essential here. If \( a, b \) are distinct elements then the relation \( \{(a, b), (b, a)\} \) is covered by the well-founded relations \( \{(a, b)\} \) and \( \{(b, a)\} \) but is not well-founded.

**Example 2.** Let \( \pi_1 \) be the program
the smallest ordinal α such that π admits a ranking function with values < α is the ranking height of π. The following observations may be useful in establishing termination.

What about size-change?
Termination proof rule

\[ R^+ \subseteq \triangleright_a \cup \triangleright_b \cup \ldots \cup \triangleright_z \]
Termination proof rule

\[ R^+ \subseteq \triangleright_a \cup \triangleright_b \cup \ldots \cup \triangleright_z \]

All program variables used
Termination proof rule

\[ R^+ \subseteq \geq_a \cup \geq_b \cup \cdots \cup \geq_z \]

\[ d(R)^+ \uparrow \quad \uparrow \]

All program variables used