Program termination · Lecture 3

Berkeley · Spring ’09

Byron Cook
We can build termination provers and analysis tools using mixtures of
- Symbolic model checkers for safety
- Program analysis tools
- Rank function synthesis engines

Programs:
- Arithmetic
- Sequential
- Non-recursive

We simply fail when termination cannot be proved
Variance analysis

$R^+_I$
Variance analysis
\[ R_I^+ \subseteq \bigcup \gtrdot f_1 \cup \gtrdot f_2 \cup \ldots \cup \gtrdot f_n \]
$R_I^+ \subseteq \geq f_1 \cup \geq f_2 \cup \ldots \cup \geq f_n$

Perhaps not well founded
$R^+_I \subseteq \bigvee f_1 \bigvee f_2 \bigvee \ldots \bigvee f_n$
\[ R_I^+ \subseteq \bigcup_{f_1 \geq f_2 \geq \ldots \geq f_n} \]
Refinement

$R^+_I \subseteq \uparrow \text{SYNTHESIS}(Q) \text{ (when successful)}$
\[ R_1^+ \subseteq \triangleright f_1 \cup \triangleright f_2 \cup \ldots \cup \triangleright f_n \]
Inclusion check can be made with induction if argument is carefully chosen
Recursive programs

Weakest preconditions
Recursive programs

Weakest preconditions
CFL-termination

Byron Cook
Microsoft Research

Andreas Podelski
Freiburg University

Andrey Rybalchenko
MPI-SWS

Abstract

CFL-reachability is the essence of partial correctness for recursive programs, where the qualifier CFL refers to the stack-based call-return discipline of program executions. Accordingly CFL-termination is the essence of total correctness for recursive programs. In this paper we present a program analysis method for CFL-termination. Until now, we had only program analysis methods for recursion or total correctness, but not both. We use the RHS framework [24] for interprocedural analysis to show how such methods can be integrated into a practical method for both.

Introduction

The extension of Hoare logic for reasoning about recursive programs is by now well-understood (see, e.g., [8]). In contrast, the treatment of recursion in program analysis continues to be an active research topic [3, 9–11, 13–15, 17, 23–27], as we continue to search for appropriate abstract domains for analyzing the stack as an infinite data structure. This issue is circumvented if one switches from a trace-based semantics to relational semantics (a procedure denotes a binary relation between entry and exit states). The drawback, however, is that one loses the direct connection to trace-based properties: reachability and termination, and thus partial and total correctness (or, more generally and especially for concurrent programs, safety and liveness). A breakthrough in this regard was obtained by the framework for interprocedural analysis in [24].

1

(e.g., [18, 19]): in both those cases the above-mentioned dichotomy between the trace-based semantics and the denotational (relational) semantics is not an issue. Our work differs from existing work on model checking of temporal properties (in generalization of termination and total correctness) for finite models augmented with one stack data structure (e.g., [1, 10, 16]) by the extension of its scope to general programs.

Our TERMINATOR termination prover [7] is, in some cases, capable of proving termination of recursive programs. These are cases when a precise relationship between the interplay between the stack and states in the transitive closure of the programs transition relation are not important, as we abstract this information away in previous work. TERMINATOR can, for example, prove the termination of Ackermann’s function, while it fails to prove the termination of Fibonacci’s function.

Our work distinguishes itself from both existing interprocedural analysis and model checking by the way abstraction is introduced. It is well-known that the finitary abstraction of valuations of infinite data structures is bound to lose the termination property. Instead, one needs to abstract pairs of consecutive states (e.g., by the fact that the variable x properly decreases its value). This “relational abstraction” of programs interferes in intricate ways with the abstraction of the ‘relational semantics’ of a procedure. This is one reason why it is practically mandatory to decompose the reasoning about recursion (which requires the abstraction of the ‘relational’ semantics) and the reasoning about termination (which requires ’re-
CFL-termination

Byron Cook
Microsoft Research

Andreas Podelski
Freiburg University

Andrey Rybalchenko
MPI-SWS

Abstract
CFL-reachability is the existence of terminating programs, where the call-return discipline ensures termination is the procedure. In this paper, we present methods for recursion in the RHS framework [20], methods can be integrated.

Introduction
The extension of Hoare’s treatment of recursion in program research topic [3, 9–11, 13–15, 20] search for appropriate abstract domains for an infinite data structure. This issue is crucial in switching from a trace-based semantics to relational semantics. The problem denotes a binary relation between elements, this drawback, however, is that one loses the domain properties: reachability and termination. This is partial and total correctness (or, more generally, safety) for concurrent programs, safety and liveness. A breakthrough in this regard was obtained by the framework for interprocedural analysis in [24].
Termination & recursion are orthogonal problems

Today:

- A new program transformation that returns semantically equivalent non-recursive programs
- Assumes an oracle for partial-correctness semantics
- Transformation is “parametric”
Recursive programs

```c
#include <stdio.h>

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

int main() {
    int x = nondet();
    int y = nondet();
    Ack(x, y);
    return 0;
}
```

File: c:\slam\src\terminator\demos\d2\test.c Line: 6. Function 'Ack'
Ackermann: ✓
Fibonacci: ????
$T := \emptyset$

while REACHABLE$\mathbb{B}(P, \ell, T)(\ell_{err})$ do

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

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

if SYNTHESIS([\pi_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 $T$”

We assume that REACHABLE supports recursion
procedure fib(x) begin
  \( \ell_0 : \text{if } x > 1 \text{ then begin} \)
  \( \ell_1 : \quad y := \text{fib}(x - 2) \)
  \( \ell_2 : \quad z := \text{fib}(x - 1) \)
  \( \ell_3 : \quad \text{return } y + z \)
    \quad \text{end} \)
  \( \ell_4 : \text{return } 1 \)
end
procedure fib(x) begin
  \( \ell_0 : \) if \( x > 1 \) then begin
    \( \ell_1 : \) \( y := \text{fib}(x - 2) \)
    \( \ell_2 : \) \( z := \text{fib}(x - 1) \)
    \( \ell_3 : \) return \( y + z \)
  end
  \( \ell_4 : \) return 1
end
procedure fib(x) begin
  \(\ell_1\) : use \([y:=\text{fib}(x - 2)]\)
  \(\ell_0\) : if \(x > 1\) then begin
    \(\ell_1\) : \(y := \text{fib}(x - 2)\)
    \(\ell_2\) : \(z := \text{fib}(x - 1)\)
    \(\ell_3\) : return \(y + z\)
  end
  \(\ell_4\) : return 1
end
Recursive programs

procedure fib(x) begin
  $\ell_0$: if $x > 1$ then begin
  $\ell_1$:     $y := \text{fib}(x - 2)$
  $\ell_2$:     $z := \text{fib}(x - 1)$
  $\ell_3$:     $x' = x \land y' \geq 1 \land y' \geq x \land z' = z$
  end
  $\ell_4$: return 1
end
Recursive programs

procedure fib(x) begin
  \( l_0 : \text{if } x > 1 \text{ then begin} \)
  \( l_1 : \quad y := \text{fib}(x - 2) \)
  \( l_2 : \quad z := \text{fib}(x - 1) \)
  \( l_3 : \quad x' = x \land y' \geq 1 \land y' \geq x \land z' = z \)
  end
  \( l_4 : \text{return } 1 \)
end

Level of precision determined on demand during the proof
Recursive programs

- [PLDI’06] transformation for termination is unaware of recursion

- Termination & recursion are orthogonal problems

- Today:
  - A new program transformation that returns semantically equivalent non-recursive programs
  - Assumes an oracle for partial-correctness semantics
  - Transformation is “parametric”
Recursive programs

- [PLDI’06] transformation for termination is unaware of recursion

- Termination & recursion are orthogonal

- Harder to execute
- Easier to prove

- Today:
  - A new program transformation that returns semantically equivalent non-recursive programs
  - Assumes an oracle for partial-correctness semantics
  - Transformation is “parametric”
Recursive programs

[PLDI’06] transformation for termination is unaware of recursion

- Harder to execute
- Easier to prove

Today:

\[ \text{square4} \]

A new program transformation that returns semantically equivalent non-recursive programs

- Assumes an oracle for partial-correctness semantics
- Transformation is “parametric”
Recursive programs

[PLDI’06] transformation for termination is unaware of recursion

- Harder to execute
- Easier to prove

Today:

A new program transformation that returns semantically equivalent non-recursive programs

- Assumes an oracle for partial-correctness semantics
- Transformation is “parametric”
procedure fib(x) begin
\[ l_0 : \text{if } x > 1 \text{ then begin} \]
\[ l_1 : \quad y := \text{fib}(x - 2) \]
\[ l_2 : \quad z := \text{fib}(x - 1) \]
\[ l_3 : \quad \text{return } y + z \]
\[ \text{end} \]
\[ l_4 : \text{return } 1 \]
end
procedure fib(x) begin
\[ \ell_0 : \text{if } x > 1 \text{ then begin} \]
\[ \ell_1 : \quad y := \text{fib}(x - 2) \]
\[ \ell_2 : \quad z := \text{fib}(x - 1) \]
\[ \ell_3 : \quad \text{return } y + z \]
\[ \text{end} \]
\[ \ell_4 : \text{return } 1 \]
\[ \text{end} \]
procedure fib(x) begin  
\[ \ell_0 : \text{if } x > 1 \text{ then begin} \]
\[ \ell_1 : \quad y := \text{fib}(x - 2) \]
\[ \ell_2 : \quad z := \text{fib}(x - 1) \]
\[ \ell_3 : \quad \text{return } y + z \]
end  
\[ \ell_4 : \text{return 1} \]
end
procedure fib(x) begin
  \( \ell_0 : \) if \( x > 1 \) then begin
  \( \ell_1 : \) y := fib(x - 2)
  \( \ell_2 : \) z := fib(x - 1)
  \( \ell_3 : \) return y + z
  end
  \( \ell_4 : \) return 1
procedure fib(x) begin
  if x > 1 then begin
    \( l_1 \) : y := fib(x - 2)
    \( l_2 \) : z := fib(x - 1)
    \( l_3 \) : return y + z
  end
\( l_4 \) : return 1
procedure fib(x) begin
  if x > 1 then begin
    \( l_1 \) : \( y := \text{fib}(x - 2) \)
    \( l_2 \) : \( z := \text{fib}(x - 1) \)
    \( l_3 \) : return \( y + z \)
  end
\( l_4 \) : return 1
procedure fib(x) begin

$l_0: \text{ if } x > 1 \text{ then begin}$

$l_1: \quad y := \text{fib}(x - 2)$

$l_2: \quad z := \text{fib}(x - 1)$

$l_3: \quad \text{return } y + z$

end

$l_4: \text{return 1}$
procedure fib(x) begin
    if x > 1 then begin
        y := fib(x - 2)
        z := fib(x - 1)
        return y + z
    end
end

return 1
procedure fib(x) begin
  if x > 1 then begin
    \( l_1 \) : \( y := \text{fib}(x - 2) \)
    \( l_2 \) : \( z := \text{fib}(x - 1) \)
    \( l_3 \) : return \( y + z \)
  end
end
return 1
procedure fib(x) begin
  \( l_0 \): if \( x > 1 \) then begin
  \( l_1 \): \( y := \text{fib}(x - 2) \)
  \( l_2 \): \( z := \text{fib}(x - 1) \)
  \( l_3 \): return \( y + z \)
  end
  \( l_4 \): return 1
procedure fib(x) begin
  if x > 1 then begin
    \[ l_1 : \quad y \leftarrow \text{fib}(x - 2) \]
    \[ l_2 : \quad z \leftarrow \text{fib}(x - 1) \]
    \[ l_3 : \quad \text{return } y + z \]
  end
return 1
procedure fib(x) begin
  \( l_0: \text{ if } x > 1 \text{ then begin} \)
  \( l_1: \quad y := \text{fib}(x - 2) \)
  \( l_2: \quad z := \text{fib}(x - 1) \)
  \( l_3: \quad \text{return } y + z \)
  \text{end} \)
  \( l_4: \text{ return } 1 \)
procedure fib(x) begin
    \( l_0 \): if \( x > 1 \) then begin
        \( l_1 \): \( y := \text{fib}(x - 2) \)
        \( l_2 \): \( z := \text{fib}(x - 1) \)
        \( l_3 \): return \( y + z \)
    end
    \( l_4 \): return 1
procedure fib(x) begin
  \( l_0 \): if \( x > 1 \) then begin
  \( l_1 \): \( y := \text{fib}(x - 2) \)
  \( l_2 \): \( z := \text{fib}(x - 1) \)
  \( l_3 \): return \( y + z \)
  end
\( l_4 \): return 1
procedure fib(x) begin
\[ \ell_0 : \text{if } x > 1 \text{ then begin} \]
\[ \ell_1 : \quad y := \text{fib}(x - 2) \]
\[ \ell_2 : \quad z := \text{fib}(x - 1) \]
\[ \ell_3 : \quad \text{return } y + z \]
end
\[ \ell_4 : \text{return } 1 \]
procedure fib(x) begin
    \( l_0 : \) if \( x > 1 \) then begin
        \( l_1 : \) \( y := \text{fib}(x - 2) \)
        \( l_2 : \) \( z := \text{fib}(x - 1) \)
        \( l_3 : \) return
    end
    \( l_4 : \) return 1
end

\[ x = 2 \]
\[ x = 3 \]
procedure fib(x) begin
\( l_0: \) if \( x > 1 \) then begin
\( l_1: \) \( y := \text{fib}(x - 2) \)
\( l_2: \) \( z := \text{fib}(x - 1) \)
\( l_3: \) return
end
\( l_4: \) return 1
end
[PLDI’06] transformation for termination is unaware of recursion

Termination & recursion are orthogonal problems

Today:
- A new program transformation that returns semantically equivalent non-recursive programs
- Assumes an oracle for partial-correctness semantics
- Transformation is “parametric”
Removing recursion

procedure fib(x) begin
\[ \geq x \]
\[ \ell_0 : \text{if } x > 1 \text{ then begin} \]
\[ \ell_1 : \ y := \text{fib}(x - 2) \]
\[ \ell_2 : \ z := \text{fib}(x - 1) \]
\[ \ell_3 : \ \text{return} \]
end
\[ \ell_4 : \text{return} 1 \]
Removing recursion

procedure fib(x) begin
  \( l_0 : \) if \( x > 1 \) then begin
    \( l_1 : \) \( y := \text{fib}(x - 1) \)
    \( l_2 : \) \( z := \text{fib}(x - 2) \)
    \( l_3 : \) return \( y + z \)
  end
  \( l_4 : \) return 1
end

This case can happen only when the command terminates.
Removing recursion

procedure fib(x) begin
  \( l_0: \) if \( x > 1 \) then begin
  \( l_1: \) y := fib(x - 1)
  \( l_2: \) z := fib(x - 2)
  \( l_3: \) return y + z
  end

\( l_4: \) return 1

If the command terminates we are better off with the summary.
procedure fib(x) begin
  \( \ell_0 : \text{if } x > 1 \text{ then begin} \)
  \( \ell_1 : \quad y := \text{fib}(x - 1) \)
  \( \ell_2 : \quad z := \text{fib}(x - 1) \)
  \( \ell_3 : \quad \text{return } y + z \)
  \end \)
  \( \ell_4 : \text{return } 1 \)

If the command terminates we are better off with the summary

\[ y := \text{fib}(x - 2) \]
procedure fib(x) begin
  \( \ell_0 : \text{if } x > 1 \text{ then begin} \)
  \( \ell_1 : \ y := \text{fib}(x - 1) \)
  \( \ell_2 : \ z := \text{fib}(x - 2) \)
  \( \ell_3 : \ \text{return } y + z \)
  \text{end} \)
  \( \ell_4 : \text{return } 1 \)
procedure fib(x) begin
  \( \ell_0 : \text{if } x > 1 \text{ then begin} \)
  \( \ell_1 : \quad y := \text{fib}(x - 1) \)
  \( \ell_2 : \quad z := \text{fib}(x - 2) \)
  \( \ell_3 : \quad \text{return } y + z \)
  \( \end \)
  \( \ell_4 : \text{return } 1 \)
procedure fib(x) begin
\[ \ell_0 : \text{if } x > 1 \text{ then begin} \]
\[ \ell_1 : \quad y := \text{fib}(x - 2) \]
\[ \ell_2 : \quad z := \text{fib}(x - 1) \]
\[ \ell_3 : \quad \text{return } y + z \]
end
\[ \ell_4 : \text{return } 1 \]
procedure fib(x) begin
\[ \ell_0 : \text{if } x > 1 \text{ then begin} \]
\[ \ell_1 : \quad y := \text{fib}(x - 2) \]
\[ \ell_2 : \quad z := \text{fib}(x - 1) \]
\[ \ell_3 : \quad \text{return } y + z \]
end
\[ \ell_4 : \text{return } 1 \]
procedure fib(x) begin
\[ \ell_0 : \text{if } x > 1 \text{ then begin} \]
\[ \ell_1 : \quad y := \text{fib}(x - 2) \]
\[ \ell_2 : \quad z := \text{fib}(x - 1) \]
\[ \ell_1 : \text{if } * \text{ then begin} \]
\[ \quad \text{use } [y := \text{fib}(x - 2)] \]
\[ \quad \text{end else begin} \]
\[ \quad x := x - 2 \]
\[ \quad \text{goto } \ell_0 \]
\[ \text{end} \]

Recursive programs

- Program transformation for termination is unaware of recursion
- Hard to execute but easier to prove
- A new program transformation that returns semantically equivalent non-recursive programs
- Assumes an oracle for partial-correctness semantics
- Transformation is “parametric”
procedure fib(x) begin

$l_0 : \text{ if } x > 1 \text{ then begin}$

$l_1 : \quad \text{ if } \ast \ldots$

$l_2 : \quad \text{ if } \ast \ldots$

$l_3 : \quad \text{return } y + z$

\hspace{1cm} \text{end}$

$l_4 : \text{return 1}$

end
procedure fib(x) begin
\[ l_0 : \text{if } x > 1 \text{ then begin } \]
\[ l_1 : \quad \text{if } \ldots \]
\[ l_2 : \quad \text{if } \ldots \]
\[ l_3 : \quad \text{return } y + z \]
\[ \text{end} \]
\[ l_4 : \text{return } 1 \]
end
procedure fib(x) begin
\[\ell_0 : \text{if } x > 1 \text{ then begin}\]
\[\ell_1 : \quad \text{if } \ast \ldots\]
\[\ell_2 : \quad \text{if } \ast \ldots\]
\[\ell_3 : \quad \text{return } y + z\]
\[\text{end}\]
\[\ell_4 : \quad \text{return } 1\]
\[\text{end}\]
procedure fib(x) begin
  \( l_0 : \text{if } x > 1 \text{ then begin} \)
  \( l_1 : \text{if \ldots} \)
  \( l_2 : \text{if \ldots} \)
  \( l_3 : \text{assume(false)} \)
  \text{end} \)
  \( l_4 : \text{assume(false)} \)
end
procedure fib(x) begin
\[ l_0 : \text{if } x > 1 \text{ then begin} \]
\[ l_1 : \quad \text{if } \ast \ldots \]
\[ l_2 : \quad \text{if } \ast \ldots \]
\[ l_3 : \quad \text{assume(false)} \]
end
\[ l_4 : \quad \text{assume(false)} \]
end

"a wrong turn was made somewhere along the way......."
procedure f(x) begin
\[ \ell_0 : \text{if } x = 1 \text{ then begin} \]
\[ \ell_1 : \quad f(0) \]
\[ \quad \text{end else begin} \]
\[ \ell_2 : \quad f(1) \]
\[ \quad \text{end} \]
\[ \ell_3 : \text{return} \]
end
procedure f(x) begin
\[ \ell_0: \text{if } x = 1 \text{ then begin} \]
\[ \ell_1: \quad f(0) \quad \leftarrow \]
\[ \text{end else begin} \]
\[ \ell_2: \quad f(1) \quad \leftarrow \]
\[ \text{end} \]
\[ \ell_3: \text{return} \]
end \text{ assume(false)}

\text{if } \ast \text{ then assume(false) else}
\quad x: = \ldots \quad \text{goto } \ell_0
procedure f(x) begin
  \textcolor{red}{l_0: \textbf{if} x = 1 \textbf{then} \begin{align*}
  & f(0) \\
  & \textbf{end else} \begin{align*}
  & f(1) \\
  & \textbf{end}
\end{align*}
\end{align*}}
  \textbf{end assume(false)}
end  \textbf{assume(false)}
procedure f(x) begin
  \ell_0 : y := 1
  \ell_1 : while y \geq 0 begin
  \ell_2 : \quad f(x - y)
  \ell_3 : \quad y := y - 1
          \quad end
  \ell_4: \quad return
end
procedure f(x) begin
\ell_0 : y := 1
\ell_1 : while y \geq 0 begin
\ell_2 : \quad f(x - y)
\ell_3 : \quad y := y - 1
\quad end
\ell_4: \quad \text{return}
end

Counter example
to termination
need both cases
in the "if"
procedure f(x) begin
\[ l_0 : y := 1 \]
\[ l_1 : \text{while } y \geq 0 \text{ begin} \]
\[ l_2 : \quad f(x - y) \]
\[ l_3 : \quad y := y - 1 \]
\[ \text{end} \]
\[ l_4 : \text{return} \]
\[ \text{end} \]
procedure f(x) begin
  $l_0$: $y := 1$
  $l_1$: if $x \geq 0$ begin
  $l_2$: $y := x \times f(x - 1)$
  end
  $l_3$: return $y$
end
procedure f(x) begin

$l_0 : y := 1$

$l_1 : \text{if } x \geq 0 \text{ begin}$

$l_2 : y := x \times f(x - 1)$

end

$l_3 : \text{return } y$

end

Summary
"true" suffices......
Imagine that we have relational summaries that underapproximate partial-correctness semantics.

We can use these summaries to prove non-terminating using the same technique.
while $x > 0$ do
    $f(x)$;
    $x := x + 1$;

done
procedure f(x) begin
    if x > 0 then
        f(x - 1);
    fi
    return;
end

while x > 0 do
    f(x);
    x := x + 1;
end

done
Summarize using only variables from the cone-of-influence of the ranking function

```plaintext
while x > 0 do
    f(x);
    x := x + 1;

done
```
Summarize using only variables from the cone-of-influence of the ranking function

\[
\text{while } x > 0 \text{ do}
\]
\[
\text{use } x' = x;
\]
\[
x := x + 1;
\]
\[
\text{done}
\]
Summarize using only variables from the cone of influence of the ranking function

\[
\text{while } x > 0 \text{ do}
\]
\[
\text{use } x' = x;
\]
\[
x := x + 1;
\]
\[
done
\]

Non termination easy to prove
**Discussion**

- **Semantics preserving recursion elimination**
  - Assumes (perhaps an overapproximation of) partial-correctness semantics
  - Transformed program is harder to execute, but simplifies proof of program termination
  - Shows that termination and recursion are somehow orthogonal
    - Similar to observations about the heap

- **Transformation case-splits on termination from a given state**
  - Doesn’t terminate? *Throw away the stack*........
  - Does terminate? *Use a summary*........

- **Implementation is a snap!**
  - Termination for non-recursive programs + relational RHS
  - Standard techniques used to refine RHS summaries on-demand
procedure fib(x) begin
\[\ell_0: \text{if } x > 1 \text{ then begin}\]
\[\ell_1: \text{if } * \text{ then}\]
\[y := \text{fib}'(x - 2);\]
\[\text{else}\]
\[x := x - 2;\]
\[\text{goto } \ell_0;\]
\[\text{fi}\]
\[\ell_2: \text{if } * \text{ then}\]
\[z := \text{fib}'(x - 1);\]
\[\text{else}\]
\[x := x - 1;\]
\[\text{goto } \ell_0;\]
\[\text{fi}\]
\[\ell_3: \text{return } y + z;\]
\[\text{end}\]
\[\ell_4: \text{return } 1;\]
\[\text{end}\]

procedure fib'(x) begin
\[\text{if } x > 1 \text{ then begin}\]
\[\text{/* IGNORE CUTPOINT */}\]
\[y := \text{fib}'(x - 2);\]
\[\text{/* IGNORE CUTPOINT */}\]
\[z := \text{fib}'(x - 1);\]
\[\text{return } y + z;\]
\[\text{end}\]
\[\text{return } 1;\]
\[\text{end}\]
Recursive programs

Weakest preconditions
Proving Conditional Termination

Byron Cook\textsuperscript{1}, Sumit Gulwani\textsuperscript{1}, Tal Lev-Ami\textsuperscript{2,\ast}, Andrey Rybalchenko\textsuperscript{3,\ast\ast}, and Mooly Sagiv\textsuperscript{2}

\textsuperscript{1} Microsoft Research \quad \textsuperscript{2} Tel Aviv University \quad \textsuperscript{3} MPI-SWS

Abstract. We describe a method for synthesizing reasonable underapproximations to weakest preconditions for termination—a long-standing open problem. The paper provides experimental evidence to demonstrate the usefulness of the new procedure.

1 Introduction

Termination analysis is critical to the process of ensuring the stability and us-
Proving Conditional Termination

Byron Cook¹, Sumit Gulwani¹, Tal Levy-Ami², Andrey Rybalchenko²

1 Introduction

Termination analysis is critical to the correctness of programs, ensuring the stability and usability of systems. It involves determining whether a program will terminate for all possible inputs, which is a fundamental problem in computer science.

Likely not the last word on the topic, simply first to make some progress.
\text{TERRMINATIONPROVER}(P) \subseteq \text{WP}(P, \text{true})
TERMINATIONPROVER(P) ⊆ WP(P, true)

returns \text{true} or \text{false}
TerminationProver($P$) \subseteq WP(P, true)

Not WLP
\textsc{TerminationProver}(P) \subseteq \text{WP}(P, \text{true})

Wanted: the right precondition
TerminationProver\( (P) \subseteq WP(P, true) \)

\[ WP(P, C) = WLP(P, C) \land WP(P, true) \]
Underapproximating weakest preconditions

WP(P, true) = WLP(P, C) \land WP(P, true)
Motivation

Automatic termination/liveness proving is now a reality. Advanced termination/liveness tools now supporting concurrency, pointers, heap, recursion, omega-regular properties, counterexample-generation, etc.

Tools:
- Terminator (currently being transferred into Windows SDV product)
- ARMC (Andrey's publicly available version)
- Polyrank (from Bradley, Manna, Sipma)
- T2 (in development for my book and CMU course)
Motivation

Automatic termination/liveness proving is now a reality. Advanced termination/liveness tools now supporting:

- Concurrency
- Pointers
- Heap
- Recursion
- Omega-regular properties
- Counterexample-generation, etc.

Tools:

- Terminator (currently being transferred into Windows SDV product)
- ARMC (Andrey's publicly available version)
- Polyrank (from Bradley, Manna, Sipma)
- T2 (in development for my book and CMU course)

```c
void main()
{
    int x = nondet();
    int y = nondet();
    int z = nondet();

    while(x>=1 && y>=1) {
        if (nondet()) {
            x = x - z;
            y = nondet();
        } else {
            y--;
        }
    }
}
```
Motivation

Automatic termination/liveness proving is now a reality

Advanced termination/liveness tools now supporting
- Concurrency,
- Pointers,
- Heap,
- Recursion,
- Omega-regular properties,
- Counterexample-generation,
- etc

Tools:
- Terminator (currently being transferred into Windows SDV product)
- ARMC (Andrey's publicly available version)
- Polyrank (from Bradley, Manna, Sipma)
- T2 (in development for my book and CMU course)

Does this code always terminate?

```
#define nondet() (nonexistent)

f() {
    z;
    if (nondet()) {
        x = x - z;
        y = nondet();
    } else {
        y--;
    }
}
```
Motivation

Automatic termination/liveness proving is now a reality. Advanced termination/liveness tools now supporting concurrency, pointers, heap, recursion, omega-regular properties, counterexample-generation, etc.

Tools:
- Terminator (currently being transferred into Windows SDV product)
- ARMC (Andrey's publicly available version)
- Polyrank (from Bradley, Manna, Sipma)
- T2 (in development for my book and CMU course)

What's the weakest precondition?
Automatic termination/liveness proving is now a reality. Advanced termination/liveness tools now supporting concurrency, pointers, heap, recursion, omega-regular properties, counterexample-generation, etc.

Tools:
- Terminator (currently being transferred into Windows SDV product)
- ARMC (Andrey's publicly available version)
- Polyrank (from Bradley, Manna, Sipma)
- T2 (in development for my book and CMU course)
Motivation

Automatic termination/liveness proving is now a reality. Advanced termination/liveness tools now supporting concurrency, pointers, heap, recursion, omega-regular properties, counterexample-generation, etc.

Tools:
- Terminator (currently being transferred into Windows SDV product)
- ARMC (Andrey's publicly available version)
- Polyrank (from Bradley, Manna, Sipma)
- T2 (in development for my book and CMU course)
Motivation

 Couldn’t prove $[\pi_c]_*[\pi_s]$ well founded.
{(s, t) | f(s) > f(t) \land f(s) \geq 0 \implies \text{can't find an } f : S \to \mathbb{Z} \}
Couldn’t find an $f : S \rightarrow \mathbb{Z}$

$\left[\pi_c\right] \ast \left[\pi_s\right] \subseteq$

$C := \text{true}$

\textbf{while} $\neg \text{TERMINATOR}(P, C)$ \textbf{do}

\textbf{let} $(\pi_s, \pi_c)$ be the counterexample to termination

$C := C \land \text{PRESYNTH}(\pi_s, \pi_c)$

\textbf{od}

\textbf{return} $C$
• Find a set $B$ such that for all $b \in B$

$$\left[\pi_c\right]_\star\left[\pi_s\right] \subseteq \{(s, t) \mid b(s) \geq 0\}$$

• For each $b \in B$, find a set of states $Q_b$ s.t.

$$\left(\left[\pi_c\right]_\star\left[\pi_s\right]\right)_\downarrow Q_b \subseteq \{(s, t) \mid b(s) > b(t)\}$$

• Return $\bigcup_{b \in B} \text{WLP}(\left(\left[\pi_c\right]_\star\left[\pi_s\right]\right)^*, Q_b)$
PreSynth algorithm

- For
  \[ C := \text{true} \]
  \[ \text{while } \neg \text{TERMINATOR}(P, C) \text{ do} \]
  \[ \text{let } (\pi_s, \pi_c) \text{ be the counterexample to termination} \]
  \[ C := C \land \text{PRESYNTH}(\pi_s, \pi_c) \]
  \[ \text{od} \]
- Return
  \[ \bigcup_{b \in B} \text{WLP}((\lfloor [\pi_c] \rfloor \searrow [\pi_s])^*, Q_b) \]
\[ R(X, X') := \left[ \pi_c \right]_\star \left[ \pi_s \right] \]
\[ C(X) := \text{false;} \]
\[ B(X) := \text{QELIM}(\exists X'. \ R(X, X')) \]
\textbf{foreach} conjunct \( b(X) \geq 0 \) in \( B(X) \) \textbf{do}
\[ Q_b(X) := \text{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \]
\[ C(X) := C(X) \lor \text{WLP}(R^*(X, X'), Q_b(X)) \]
\textbf{done}
\textbf{return} \( C(X) \)
\[ R = x' = x - z \]
\[ \land x \geq 1 \]
\[ \land y \geq 1 \]
\[ \land z' = z \]
\[ X = \{ x, y, z \} \]
\[ X' = \{ x', y', z' \} \]
Motivation

Automatic termination/liveness proving is now a reality.

Advanced termination/liveness tools now supporting:
- Concurrency,
- Pointers,
- Heap,
- Recursion,
- Omega-regular properties,
- Counterexample-generation,
- etc.

Tools:
- Terminator (currently being transferred into Windows SDV product)
- ARMC (Andrey's publicly available version)
- Polyrank (from Bradley, Manna, Sipma)
- T2 (in development for my book and CMU course)

```c
void main()
{
    int x = nondet();
    int y = nondet();
    int z = nondet();

    while(x>=1 && y>=1) {
        if (nondet()) {
            x = x - z;
            y = nondet();
        } else {
            y--;
        }
    }
}
```
Example

\[ R(X, X') := [\pi_c]\pi_s \]

\[ C(X) := \text{false}; \]

\[ B(X) := \text{QELIM}(\exists X'. R(X, X')) \]

\textbf{foreach conjunct } \( b(X) \geq 0 \) in \( B(X) \) \textbf{do}

\[ Q_b(X) := \text{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \]

\[ C(X) := C(X) \lor \text{WLP}(R^*(X, X'), Q_b(X)) \]

\textbf{done}

\textbf{return } C(X)
Example

\[ R(X, X') := \left[ \pi_c \right]_* \left[ \pi_s \right] \]

\[ C(X) := \text{false;} \]

\[ B(X) := \text{QELIM}(\exists X'. R(X, X')) \]

\textbf{foreach conjunct } \( b(X) \geq 0 \text{ in } B(X) \text{ do} \]

\[ Q_b(X) := \text{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \]

\[ C'(X) := C(X) \lor \text{WLP}(R^*(X, X'), Q_b(X)) \]

\textbf{done}

\textbf{return } C(X)
Example

\[ Q\text{ELIM}(\exists x', y', z'. R) = x - 1 \geq 0 \land y - 1 \geq 0 \]

\[ R = x' = x - z \]
\[ \land x \geq 1 \]
\[ \land y \geq 1 \]
\[ \land z' = z \]

\[ X = \{x, y, z\} \]

\[ X' = \{x', y', z'\} \]

\[ R(X, X') := \exists x' \cdot \pi_s \]

\[ C(X) := \]

\[ B(X) := Q\text{ELIM}(\exists X'. R(X, X')) \]

foreach conjunct \( b(X) \geq 0 \) in \( B(X) \) do

\[ Q_b(X) := Q\text{ELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \]

\[ C(X) := C(X) \lor \text{WLP}(R^*(X, X'), Q_b(X)) \]

done

return \( C(X) \)
Example

\[
\text{QELIM}(\exists x', y', z'. \ R) = x - 1 \geq 0 \land y - 1 \geq 0
\]

\[
R(X, X') := \Pi_{s}^{\ast}
\]

\[
C(X) := \text{QELIM}(\exists X'. \ R(X, X'))
\]

\[
B(X) := \text{QELIM}(\exists X'. \ R(X, X'))
\]

\[
\textbf{foreach conjunct } b(X) \geq 0 \text{ in } B(X) \textbf{ do}
\]

\[
Q_b(X) := \text{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X))
\]

\[
C(X) := C(X) \lor \text{WLP}(R^*(X, X'), Q_b(X))
\]

\[
\text{done}
\]

\[
\text{return } C(X)
\]
Example

\[ R = x' = x - z \]
\[ \land x \geq 1 \]
\[ \land y \geq 1 \]
\[ \land z' = z \]
\[ X = \{x, y, z\} \]
\[ X' = \{x', y', z'\} \]

\[ QELIM(\exists x', y', z'. \ R) = x - 1 \geq 0 \land y - 1 \geq 0 \]

\[ R(X, X') \text{ i.e. } x - 1 \text{ or } y - 1 \]
\[ C(X) \]
\[ B(X) := QELIM(\forall X'. \ R(X, X')) \]
\[ \text{foreach conjunct } b(X) \geq 0 \text{ in } B(X) \text{ do} \]
\[ Q_b(X) := QELIM(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \]
\[ C(X) := C(X) \lor WLP(R^*(X, X'), Q_b(X)) \]
\[ \text{done} \]
\[ \text{return } C(X) \]
Example

\[ R(X, X') := \left[ \pi_c \right] \star \left[ \pi_s \right] \]

\( C(X) := \text{false}; \)

\( B(X) := \text{QELIM}(\exists X'. R(X, X')) \)

\textbf{foreach} conjunct \( b(X) \geq 0 \) in \( B(X) \) \textbf{do}

\[ Q_b(X) := \text{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \]

\[ C'(X) := C(X) \lor \text{WLP}(R^*(X, X'), Q_b(X)) \]

\textbf{done}

\textbf{return} \( C(X) \)
Example

\[ R = x' = x - z \]
\[ \land \quad x \geq 1 \]
\[ \land \quad z \geq 1 \]

\[
QELIM(\forall x', y', z'. \ R \Rightarrow x - 1 > x' - 1) = z \leq -1
\]

\[
C(X) := R
\]

\[
B(X) := QELIM(\exists X'. \ R(x', x', x', x'))
\]

\[
\text{foreach conjunct } b(X) \geq 0 \text{ in } B(X) \text{ do}
\]

\[
Q_b(X) := QELIM(\forall X'. \ R(X, X') \Rightarrow b(X) > b(X))
\]

\[
\hat{C}(X) := C(X) \lor WLP(R^*(X, X'), Q_b(X))
\]

\[
done
\]

\[
\text{return } C(X)
\]
Example

\[ R = x' = x - z \]
\[ \land x \geq 1 \]
\[ x' \geq 1 \]

\[ \text{QELIM}(\forall x', y', z'. R \Rightarrow x - 1 > x' - 1) = z \leq -1 \]

\[ C(X) := R \]
\[ B(X) := \text{QELIM}(\exists X'. R(X, X')) \]

\textbf{foreach} conjunct \( b(X) \geq 0 \) \textbf{do}

\[ Q_b(X) := \text{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \]
\[ C(X) := C(X) \lor \text{WLP}(R^*(X, X'), Q_b(X)) \]

\textbf{done}

\textbf{return} \( C(X) \)
Example

\[ R = x' = x - z \]
\[ \land \quad x \geq 1 \]
\[ \land \quad z \geq 1 \]

\[ \text{QELIM}\left(\forall x', y', z'. \; R \Rightarrow x - 1 > x' - 1\right) = z \leq -1 \]

\[ C(X) := \text{WLP}\left(\forall z. \; R^* (X, X') \Rightarrow z \leq -1 \lor x < 1 \lor y < 1\right) \]

\[ B(X) \quad \text{foreach} \quad \text{conj} \]

\[ Q_b(X) := \text{QELIM}\left(\forall z. \; R(X, X') \Rightarrow b(X) > b(X)\right) \]

\[ C(X) := C(X) \lor \text{WLP}\left(R^* (X, X'), Q_b(X)\right) \]

\text{done}

\text{return } C(X)
Other examples

```java
// @requires true;
while(x>0){
  x=x+y;
  y=y+z;
}
```
Other examples

\[ x \leq 0 \lor x + y \leq 0 \lor x + 2y + z \leq 0 \lor x + 3y + 3z \leq 0 \lor z < 0 \lor (z \leq 0 \land y < 0) \]

// @requires true;
while(x>0){
    x=x+y;
    y=y+z;
}

110
// @requires true;
while(x<=N){
    if (*) {
        x=2*x+y;
        y=y+1;
    } else {
        x++;
    }
}
Other examples

```
// @requires true;
while(x<=N){
    if (*) {
        x=2*x+y;
        y=y+1;
    } else {
        x++;
    }
}
```

$x > N \lor x + y \geq 0$
Other examples

// @requires true;
while (x >= 0) {
    x = -2*x + 10;
}


x > 5 ∨ x < 0

// @requires true;
while(x>=0){
    x= -2*x + 10;
}

while (x != y) {
    if (x > y) {
        x = x - y;
    } else
        y = y - x;
}
Other examples

\[ x = y \lor (x > 0 \land y > 0) \]

```c
while (x!=y) {
    if (x>y) {
        x = x - y;
    } else 
        y = y - x;
 }
```
Other examples

```
// @requires n>200 and y<9;
x = 0;
while (1) {
    if (x<n) {
        x=x+y;
        if (x>=200) break;
    }
}
```
Synthesis technique can help improve power of the termination prover

Key idea: Found precondition can be used as case split
Improving termination provers

```c
1: void main()
2: {
3:     int x, y;
4:     x = nondet();
5:     y = nondet();
6:     while (x > 0) {
7:         if (y < 0) {} else {
8:             x = x + y;
9:             y--;}
10: }
11: }
```
while (x>0) 
{
    int x, y;
    x = nondet();
    y = nondet();
    while (x>0) {
        if (y>0) {
        } else {
            x = x + y;
            y--;
        }
    }
}
void main()
{
    int x = number;
    y = number();
    while (x > 0) {
        if (y < 0) { else }
        x = x + y;
        y--;
    }
}
procedure PHASED_PRESYNTH(I, R)
begin
    C := PRESYNTH(I, R)
    if C is non-empty then
        C := C \cup PHASED_PRESYNTH(I, R \downarrow_{\sim C})
    fi
    return C
end
Recursive programs

Weakest preconditions