Program termination · Lecture 3

Berkeley · Spring '09

Byron Cook

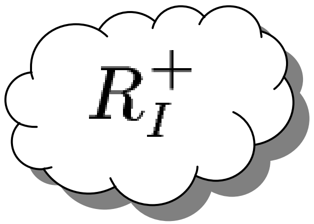
Summary from the last lecture

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

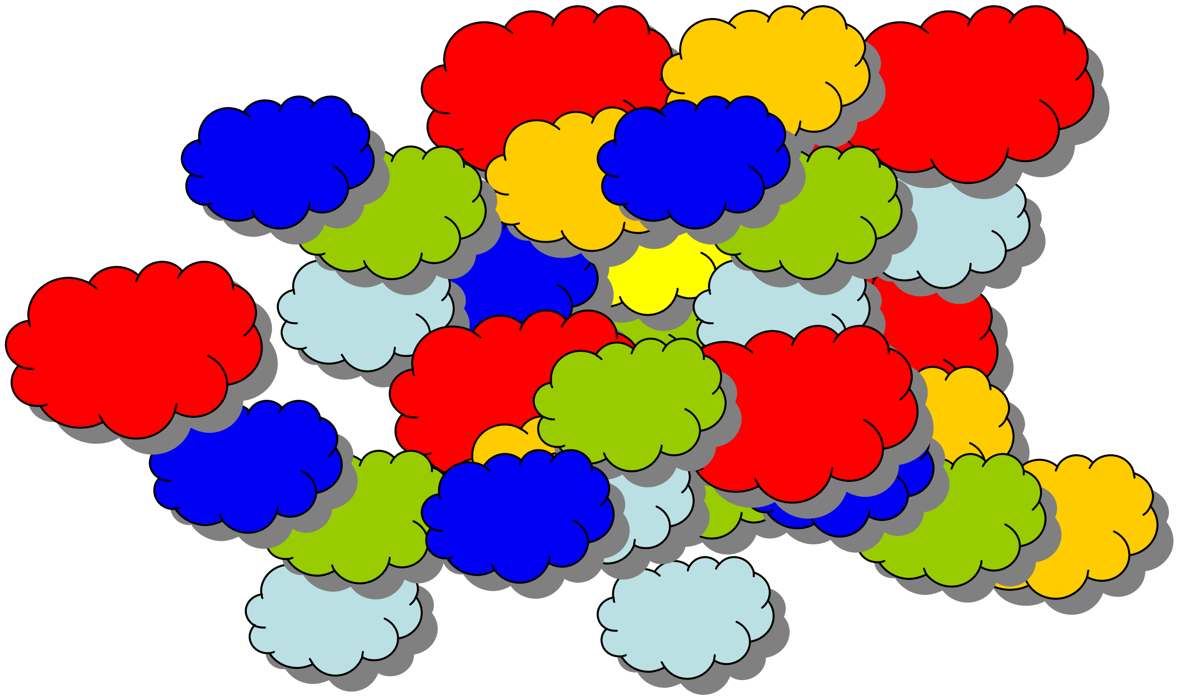
- Programs:
  - Arithmetic
  - Sequential
  - Non-recursive

- We simply fail when termination cannot be proved

Variance analysis

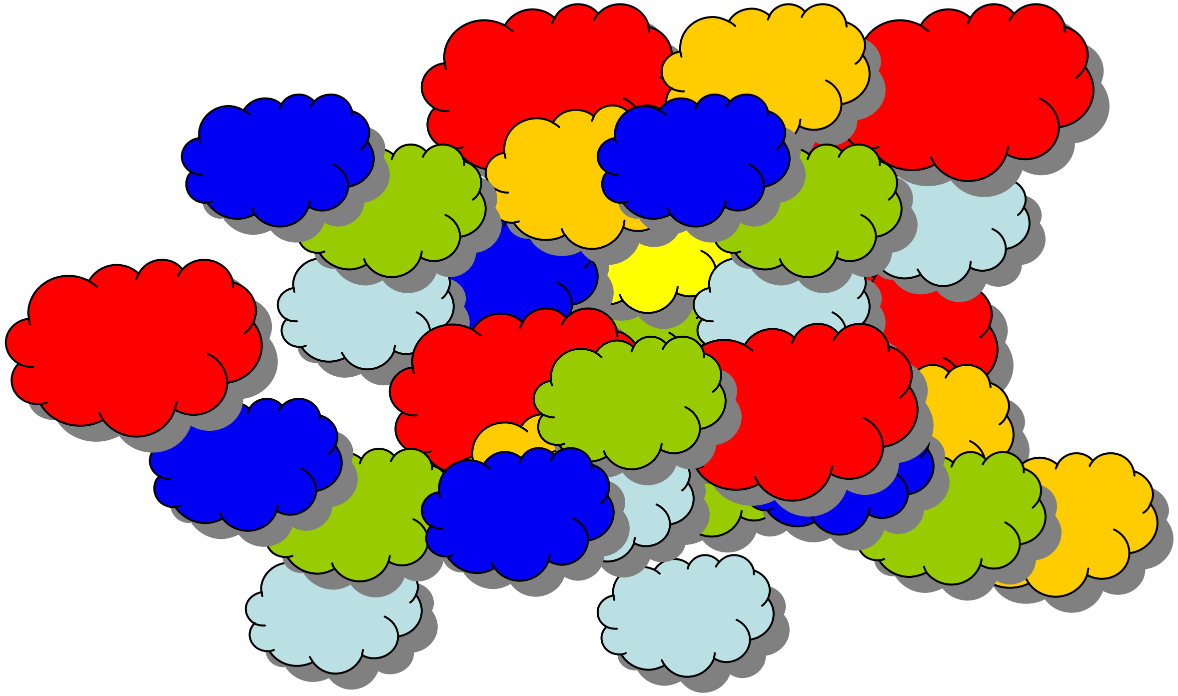


Variance analysis



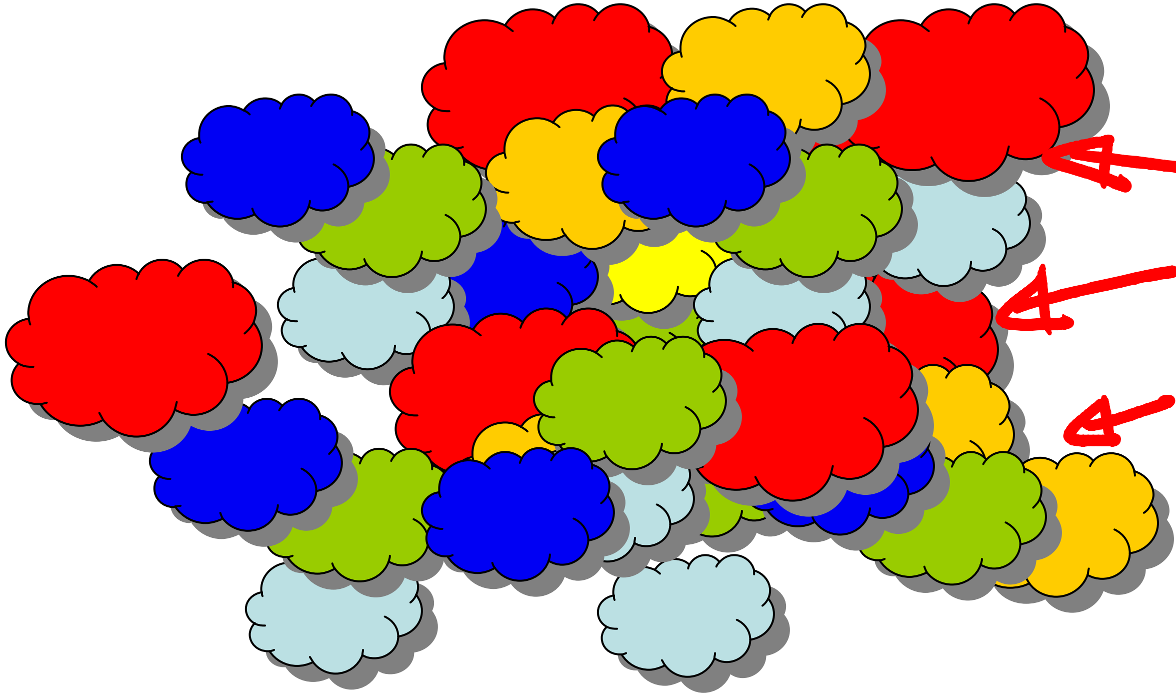
Variance analysis

$$R_I^+ \subseteq \unrhd_{f_1} \cup \unrhd_{f_2} \cup \ldots \cup \unrhd_{f_n}$$



Variance analysis

$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$

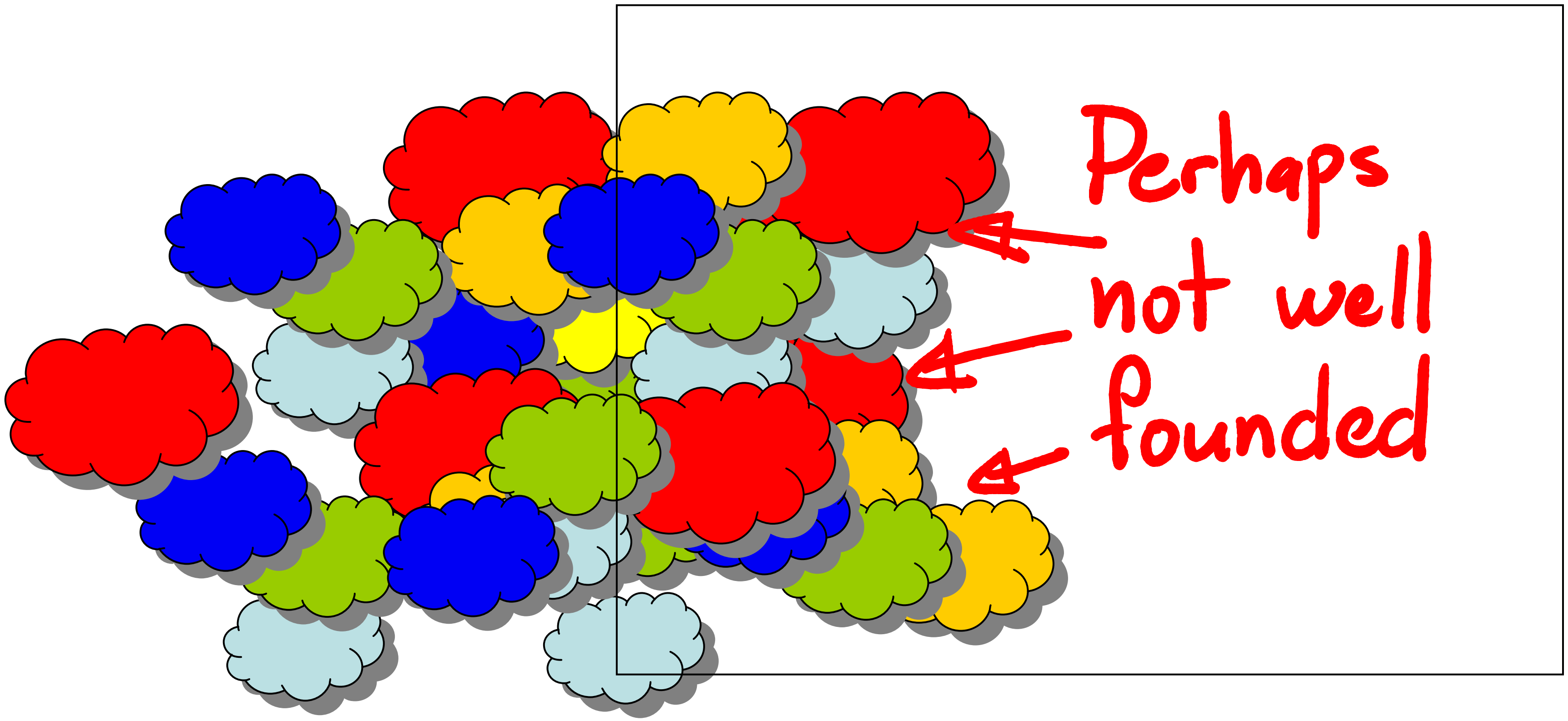


Perhaps not well founded

Variance analysis

Microsoft Research Cambridge

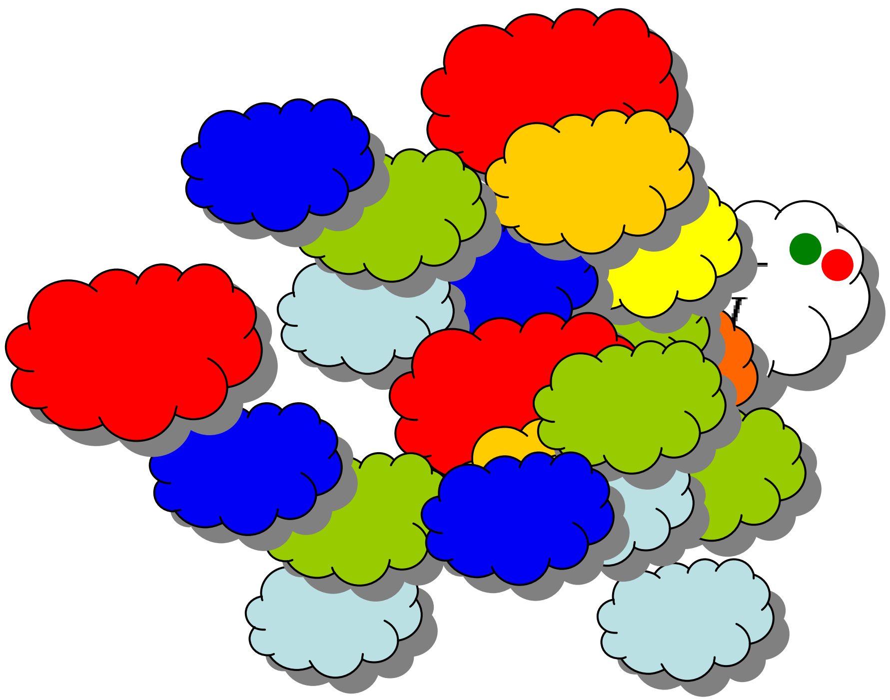
$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$



Perhaps not well founded

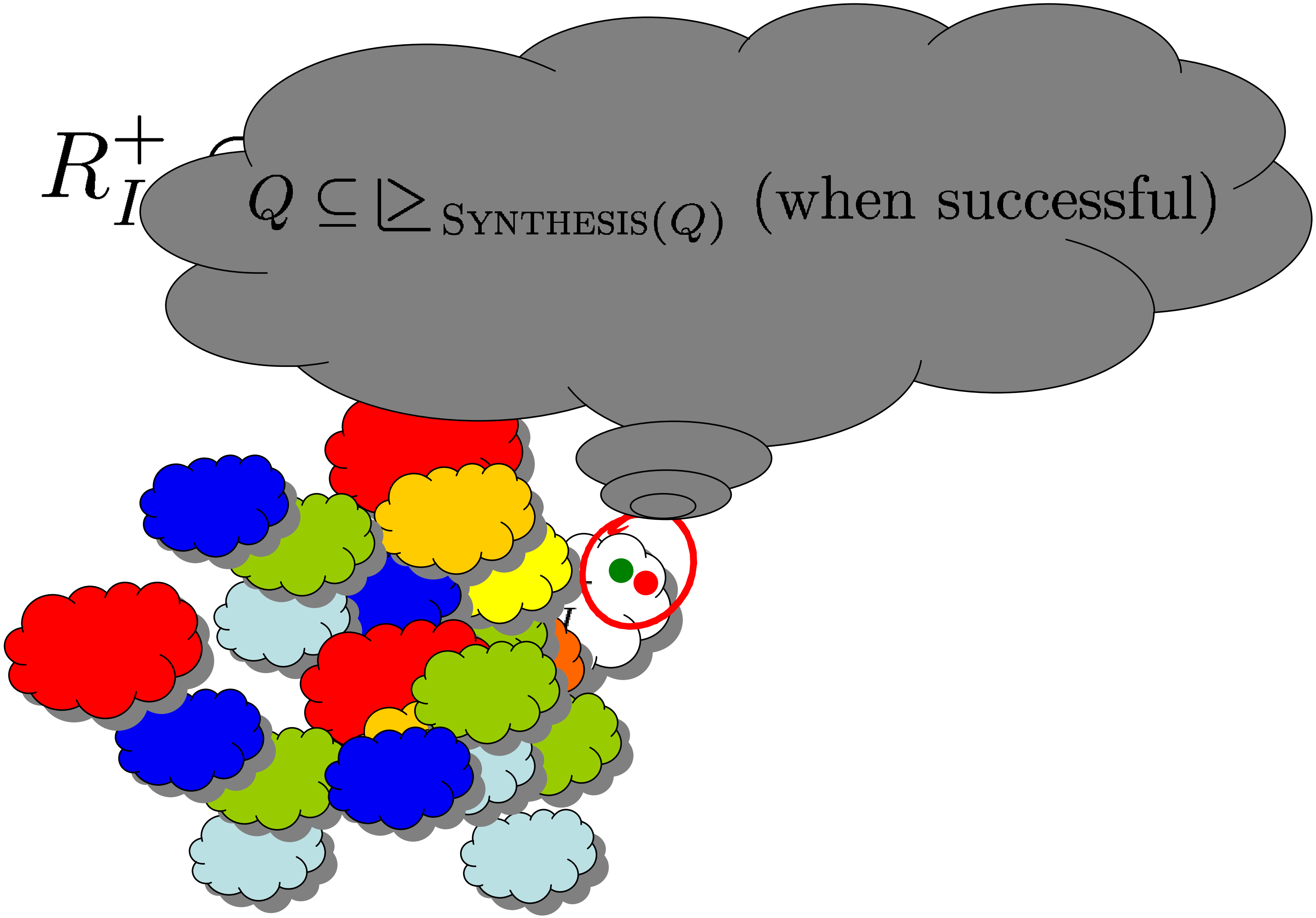
Variance analysis

$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$



Refinement

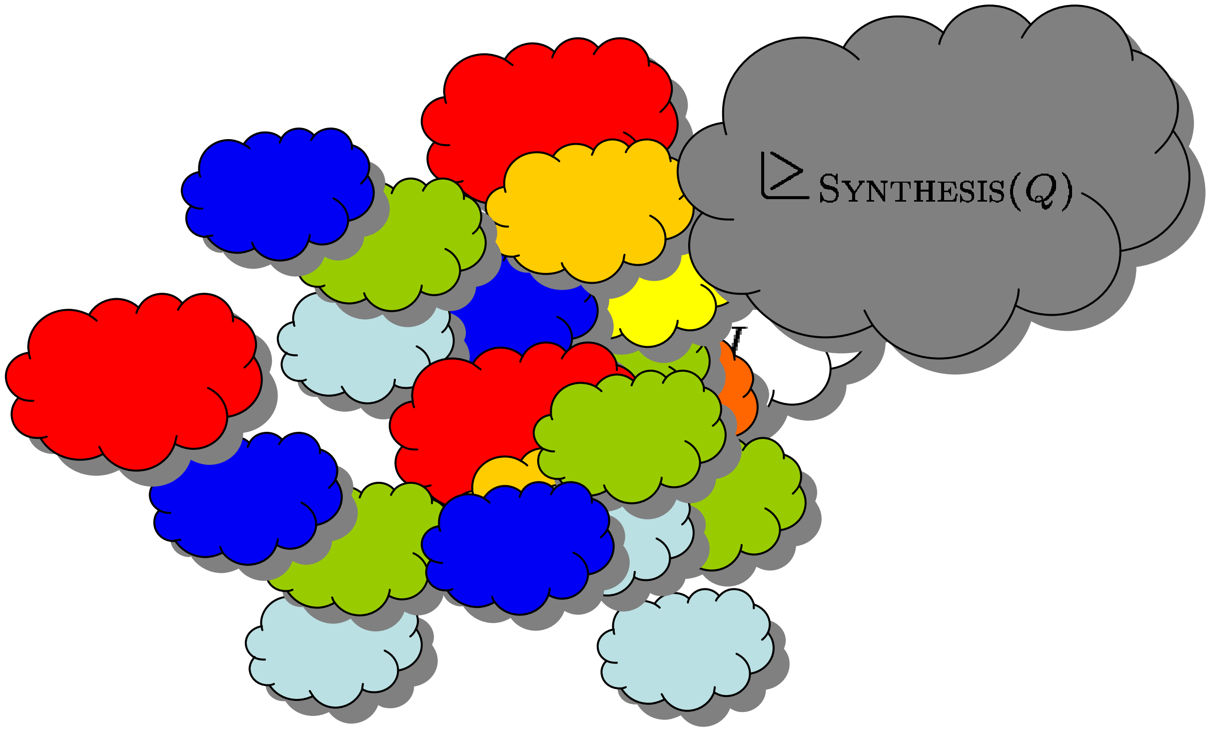
R_I^+



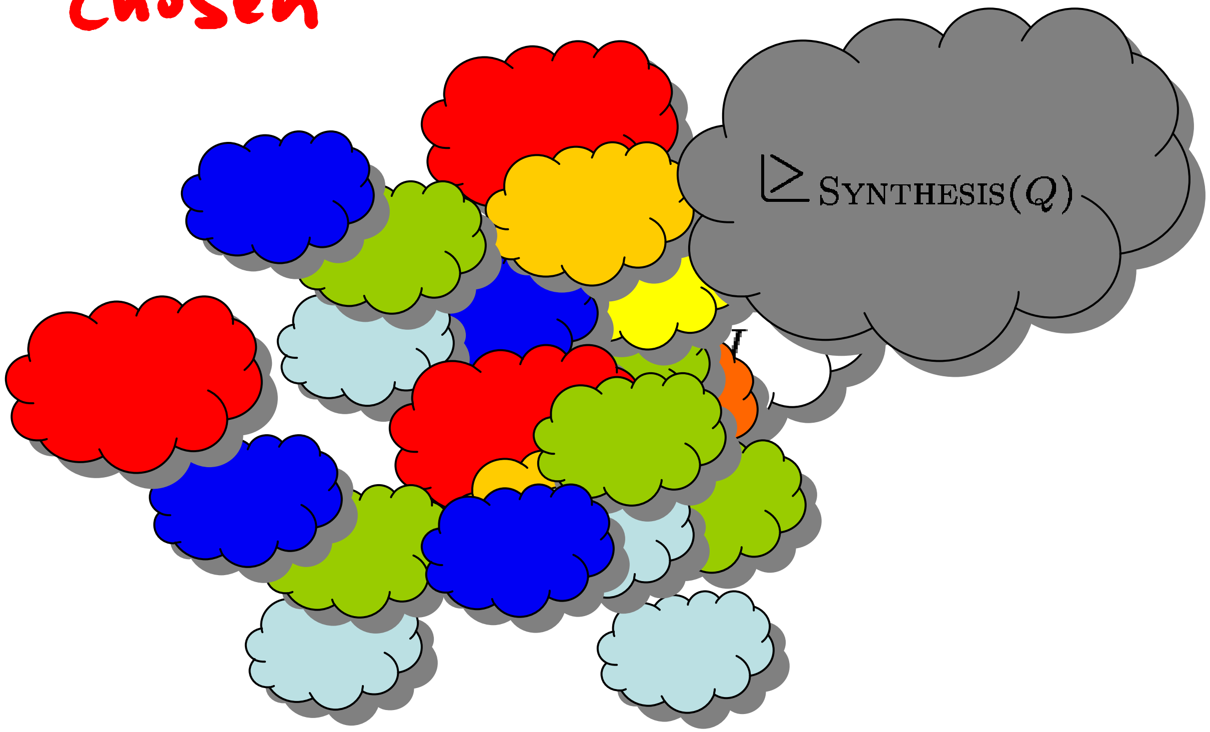
Refinement

Microsoft Research Cambridge

$$R_I^+ \subseteq \trianglerighteq_{f_1} \cup \trianglerighteq_{f_2} \cup \ldots \cup \trianglerighteq_{f_n}$$



Inclusion check can be made with induction if argument is carefully chosen

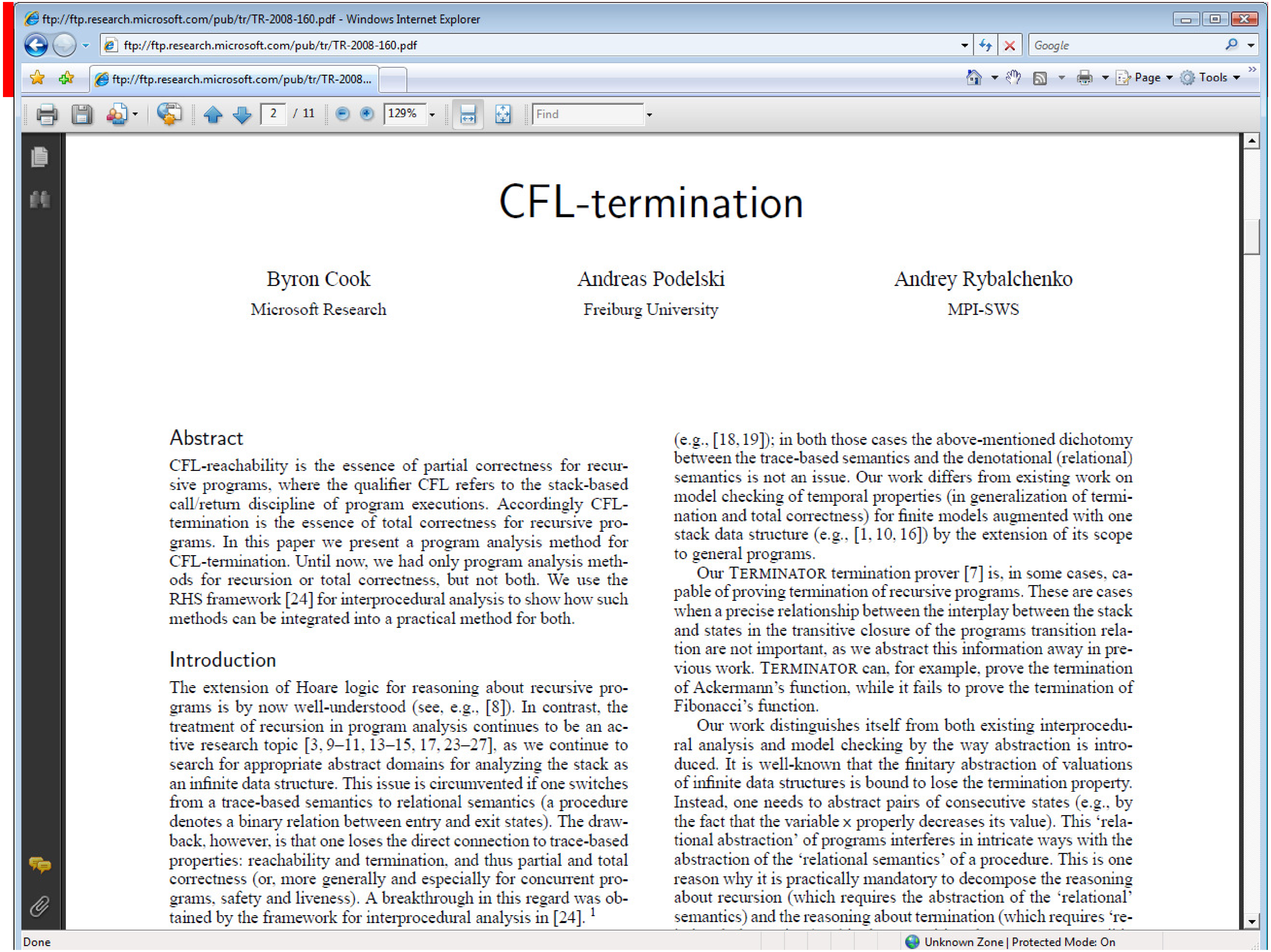


Outline

- Recursive programs
- Weakest preconditions

Outline

- Recursive programs
- Weakest preconditions



CFL-termination

Byron Cook
Microsoft Research

Andreas Podelski
Freiburg University

Andrey Rybalchenko
MPI-SWS

Abstract

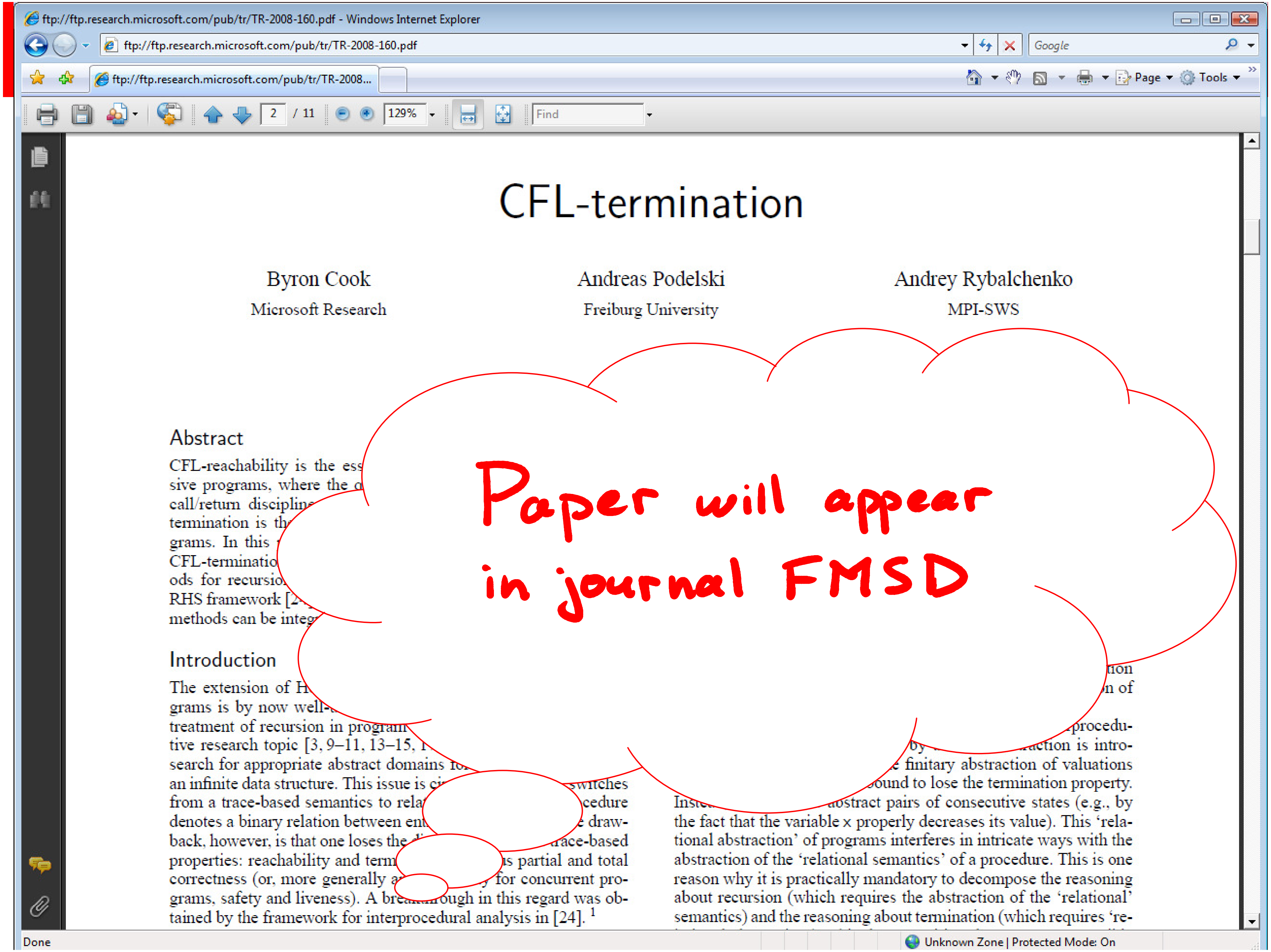
CFL-reachability is the essence of partial correctness for recursive programs, where the qualifier CFL refers to the stack-based call/return discipline of program executions. Accordingly CFL-termination is the essence of total correctness for recursive programs. In this paper we present a program analysis 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

Paper will appear in journal FMSD

Abstract

CFL-reachability is the ess
sive programs, where the c
call/return discipline
termination is th
grams. In this
CFL-terminatio
ods for recursio
RHS framework [2
methods can be integ

Introduction

The extension of H
grams is by now well-
treatment of recursion in program
tive research topic [3, 9–11, 13–15, 1
search for appropriate abstract domains fo
an infinite data structure. This issue is ci
from a trace-based semantics to rela
denotes a binary relation between ent
back, however, is that one loses the d
properties: reachability and term
correctness (or, more generally a
grams, safety and liveness). A brea
tained by the framework for interprocedural analysis in [24]. [1]

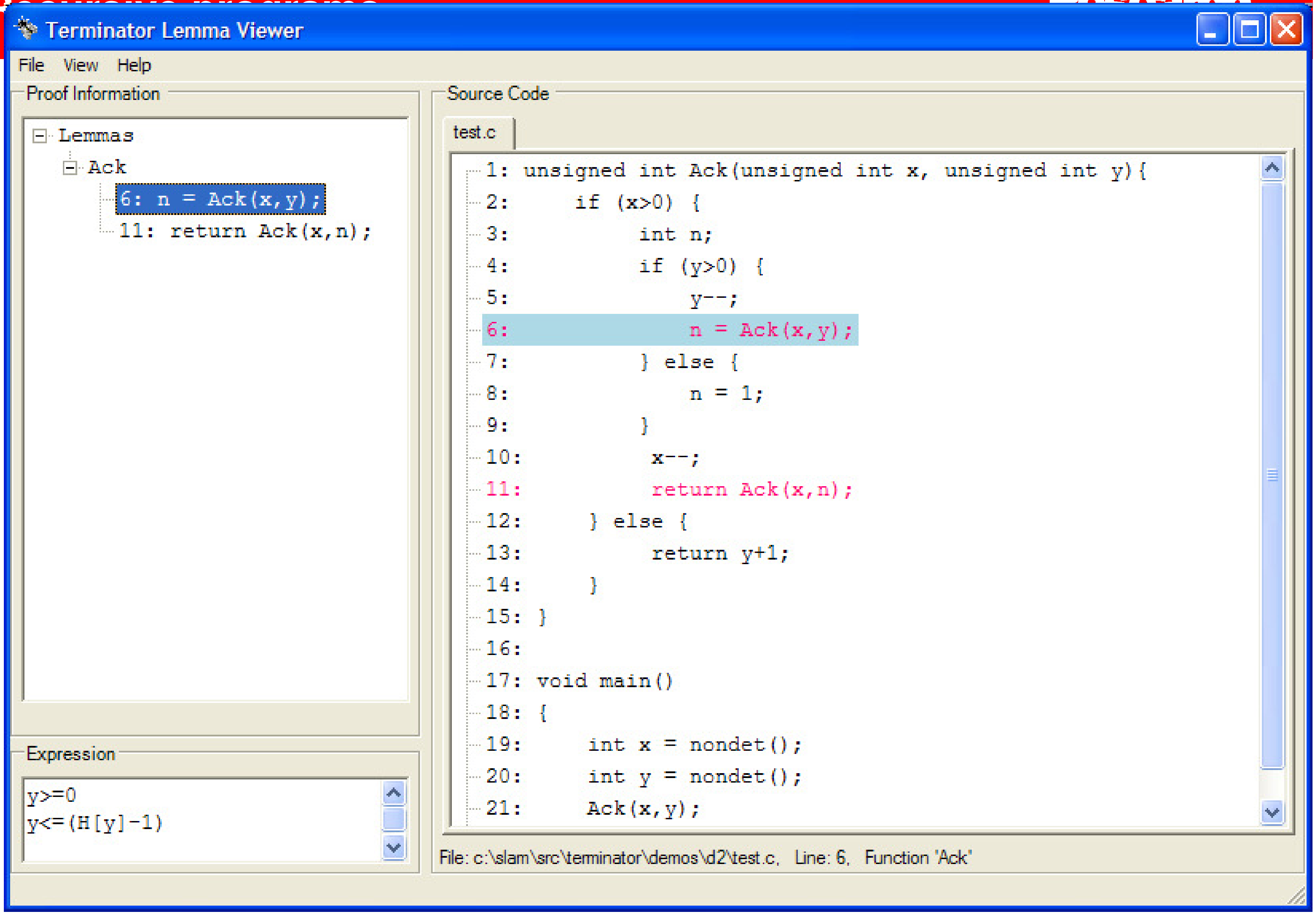
switches
cedure
e draw-
trace-based
is partial and total
for concurrent pro-
ough in this regard was ob-

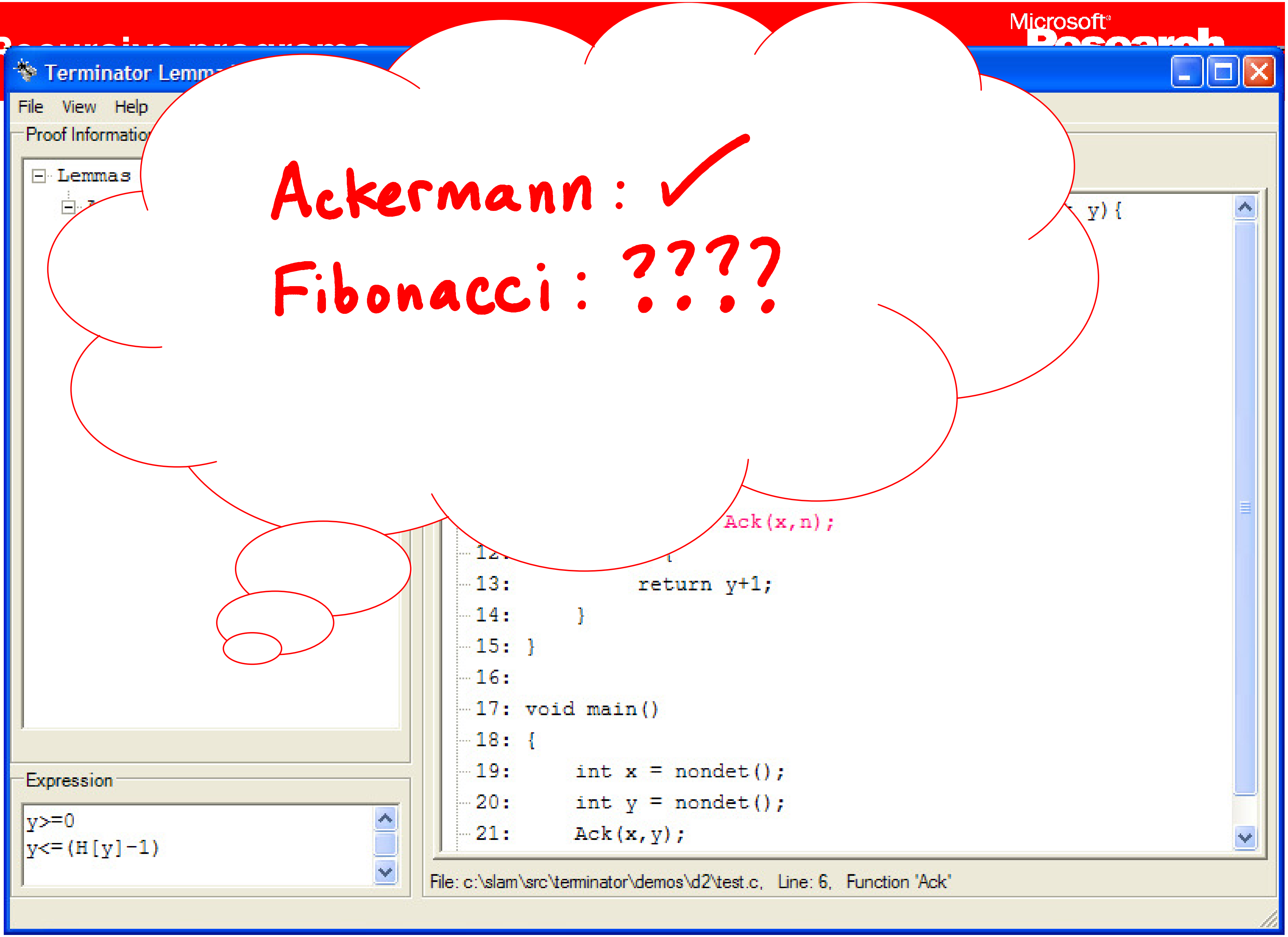
tion
n of
procedu-
action is intro-
e finitary abstraction of valuations
bound to lose the termination property.
Inste
abstract pairs of consecutive states (e.g., by
the fact that the variable x properly decreases its value). This 'relational abstraction' of programs interferes in intricate ways with 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-

Recursive programs

- 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”


Terminator Lemma Viewer
File View Help
Proof Information
Lemmas
Ack
6: n = Ack(x,y);
11: return Ack(x,n);
Expression
y>=0
y<=(H[y]-1)
Source Code
test.c
1: unsigned int Ack(unsigned int x, unsigned int y){
2: if (x>0) {
3: int n;
4: if (y>0) {
5: y--;
6: n = Ack(x,y);
7: } else {
8: n = 1;
9: }
10: x--;
11: return Ack(x,n);
12: } else {
13: return y+1;
14: }
15: }
16:
17: void main()
18: {
19: int x = nondet();
20: int y = nondet();
21: Ack(x,y);
File: c:\slam\src\terminator\demos\d2\test.c, Line: 6, Function 'Ack'


Terminator Lemm
File View Help
Proof Informatio
Lemmas
Ackermann : ✓
Fibonacci : ????
y){
Ack(x,n);
13: return y+1;
14: }
15: }
16:
17: void main()
18: {
19: int x = nondet();
20: int y = nondet();
21: Ack(x,y);
Expression
y>=0
y<=(H[y]-1)
File: c:\slam\src\terminator\demos\d2\test.c, Line: 6, Function 'Ack'

Recursive programs

Microsoft Research Cambridge

$T := \emptyset$
while $\boxed{\text{REACHABLE}}_{\boxplus(\mathcal{P},\ell,T)}(\ell_{err})$ **do**
  let π_s, π_c = lasso in $\boxplus(\mathcal{P},\ell,T)$ from 0 to ℓ, and ℓ to ℓ_{err}
  let $\rho = \alpha([\![\pi_c]\!]^*([\![\pi_s]\!]))$
  if SYNTHESIS$([\![\pi_c]\!]\downarrow_\rho)$ returns ranking relation f **then**
    $T := T \cup \trianglerighteq_f$
  else
    report "potential counterexample found: π_s,π_c"
  fi
od
report "termination proved with argument T"

We assume that REACHABLE supports recursion

Recursive programs

Microsoft Research Cambridge

$$
\begin{array}{l}
\textbf{procedure}\ \mathsf{fib}(\mathsf{x})\ \textbf{begin}\\
\ell_0 : \textbf{if}\ \mathsf{x} > 1\ \textbf{then begin}\\
\ell_1 : \qquad \mathsf{y} := \mathsf{fib}(\mathsf{x}-2)\\
\ell_2 : \qquad \mathsf{z} := \mathsf{fib}(\mathsf{x}-1)\\
\ell_3 : \qquad \textbf{return}\ \mathsf{y}+\mathsf{z}\\
\qquad \textbf{end}\\
\ell_4 \text{: } \textbf{return}\ 1\\
\textbf{end}
\end{array}
$$

Recursive programs

ℓ_1 : **use** $[\![\mathsf{y{:=}fib(x-2)}]\!]$

$\textbf{procedure}\ \mathsf{fib(x)}\ \textbf{beg}$

$\ell_0 : \textbf{if}\ \mathsf{x} > 1\ \textbf{then}\ \textbf{begi}$

$\ell_1 : \qquad \cancel{\mathsf{y} := \mathsf{fib}(\mathsf{x}-2)}$

$\ell_2 : \qquad \mathsf{z} := \mathsf{fib}(\mathsf{x}-1)$

$\ell_3 : \qquad \textbf{return}\ \mathsf{y}+\mathsf{z}$

$\qquad \textbf{end}$

$\ell_4 :\ \textbf{return}\ 1$

$\textbf{end}$

Recursive programs

$\textbf{procedure}$ fib(x) $\textbf{beg}$

ℓ_0 : $\textbf{if}$ x $>$ 1 $\textbf{then}$ $\textbf{begi}$

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

ℓ_2 : $\quad\quad$ z := fib(x $-$ 1)

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

$\quad\quad \textbf{end}$

ℓ_4: $\textbf{return}$ 1

$\textbf{end}$

ℓ_1 : $\textbf{use}$ $[\![$y:=fib(x $-$ 2)$]\!]$

x$'$:= $*$

y$'$:= $*$

z$'$:= $*$

$\textbf{assume}$($[\![$y:=fib(x $-$ 2)$]\!]$)

x := x$'$

y := y$'$

z := z$'$

Recursive programs

procedure fib(x) beg

ℓ_0 : if x > 1 then begi

ℓ_1 : ~~y := fib(x − 2)~~

ℓ_2 : z := fib(x

ℓ_3 :

en

ℓ_4: return 1

end

ℓ_1 : use $[\![$y:=fib(x − 2)$]\!]$

$\mathsf{x}' = \mathsf{x} \wedge \mathsf{y}' \geq 1 \wedge \mathsf{y}' \geq \mathsf{x} \wedge \mathsf{z}' = \mathsf{z}$

*

assume($[\![$y:=fib(x − 2)$]\!]$)

$\mathsf{x} := \mathsf{x}'$

$\mathsf{y} := \mathsf{y}'$

$\mathsf{z} := \mathsf{z}'$

Recursive programs

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then begin**

ℓ_1 : ~~y := fib(x − 2)~~

ℓ_2 : z := fib(x

ℓ_3 :

en

ℓ_4: **return** 1

end

$$x' = x \wedge y' \geq 1 \wedge y' \geq x \wedge z' = z$$

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"

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

Harder to execute

Easier to prove

- T… re orthogonal

- T… :
  - A new program transformation that returns semantically equivalent non-recursive programs
  - Assumes an oracle for partial-correctness semantics
  - Transformation is "parametric"

Microsoft Research Cambridge

→ [PLDI'06] transformation for termination is unaware of recursion

→ –

Harder to execute
Easier to prove

Overapproximation
preserves
soundness

→ T .

- A new program transformati
semantically equivalent non- sive programs
- Assumes an oracle for partial-correctness semantics
- Transformation is "parametric"

Microsoft Research Cambridge

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

Harder to execute
Easier to prove

- T

Underapproximation preserves completeness

- T .
  - A new program transformati semantically equivalent non- sive programs
  - Assumes an oracle for partial-correctness semantics
  - Transformation is "parametric"

Fibonacci

$$
\begin{array}{l}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} \\
\ell_1 : \qquad \mathsf{y} := \mathsf{fib}(\mathsf{x} - 2) \\
\ell_2 : \qquad \mathsf{z} := \mathsf{fib}(\mathsf{x} - 1) \\
\ell_3 : \qquad \textbf{return } \mathsf{y} + \mathsf{z} \\
\qquad \textbf{end} \\
\ell_4 : \textbf{return } 1 \\
\textbf{end}
\end{array}
$$

Fibonacci

$$
\begin{array}{l}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} \\
\ell_1 : \qquad \mathsf{y} := \mathsf{fib}(\mathsf{x} - 2) \\
\ell_2 : \qquad \mathsf{z} := \mathsf{fib}(\mathsf{x} - 1) \\
\ell_3 : \qquad \textbf{return } \mathsf{y} + \mathsf{z} \\
\qquad \textbf{end} \\
\ell_4 : \textbf{return } 1 \\
\textbf{end}
\end{array}
$$

Fibonacci

$\unrhd_{\mathsf{x}}$

$\mathbf{procedure}\ \mathsf{fib}(\mathsf{x})\ \mathbf{begin}$

$\ell_0 : \mathbf{if}\ \mathsf{x} > 1\ \mathbf{then\ begin}$

$\ell_1 : \quad \mathsf{y} := \mathsf{fib}(\mathsf{x} - 2)$

$\ell_2 : \quad \mathsf{z} := \mathsf{fib}(\mathsf{x} - 1)$

$\ell_3 : \quad \mathbf{return}\ \mathsf{y} + \mathsf{z}$

$\quad \mathbf{end}$

$\ell_4 :\ \mathbf{return}\ 1$

$\mathbf{end}$

Fibonacci

Microsoft Research Cambridge

$\unrhd_x$

procedure fib(x) **begin**

ℓ_0 : **if** $x > 1$ **then begin**

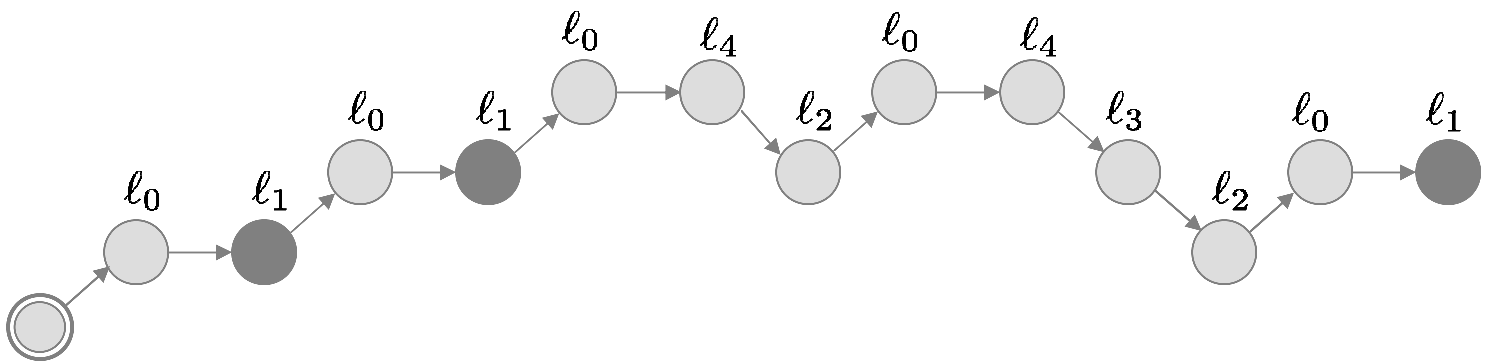
ℓ_1 : $\quad y := \mathrm{fib}(x-2)$

ℓ_2 : $\quad z := \mathrm{fib}(x-1)$

ℓ_3 : $\quad$ **return** $y + z$

end

ℓ_4: **return** 1



Fibonacci

$\trianglerighteq_{x}$

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then begin**

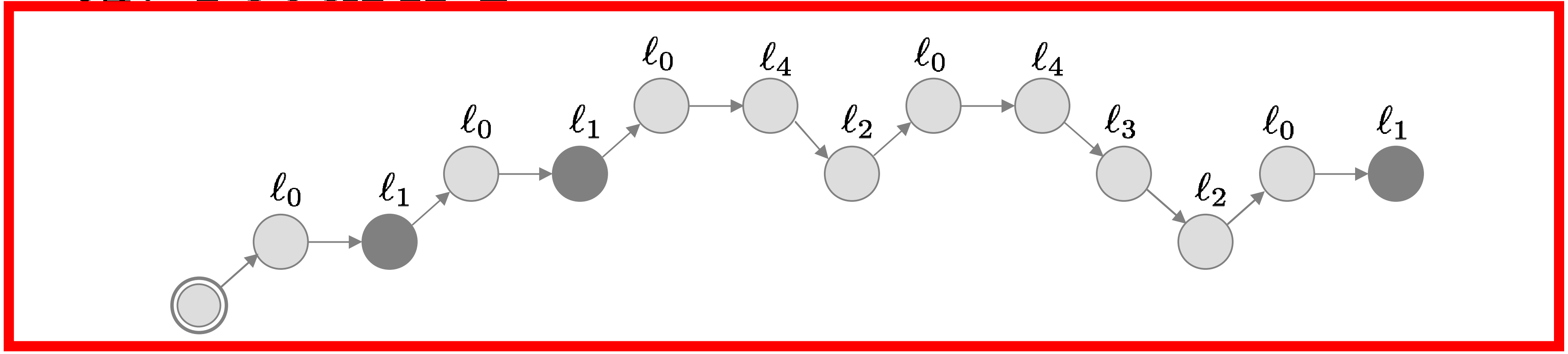
ℓ_1 : y := fib(x − 2)

ℓ_2 : z := fib(x − 1)

ℓ_3 : **return** y + z

end

ℓ_4: **return** 1



Fibonacci

$\unrhd_x$

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then begin**

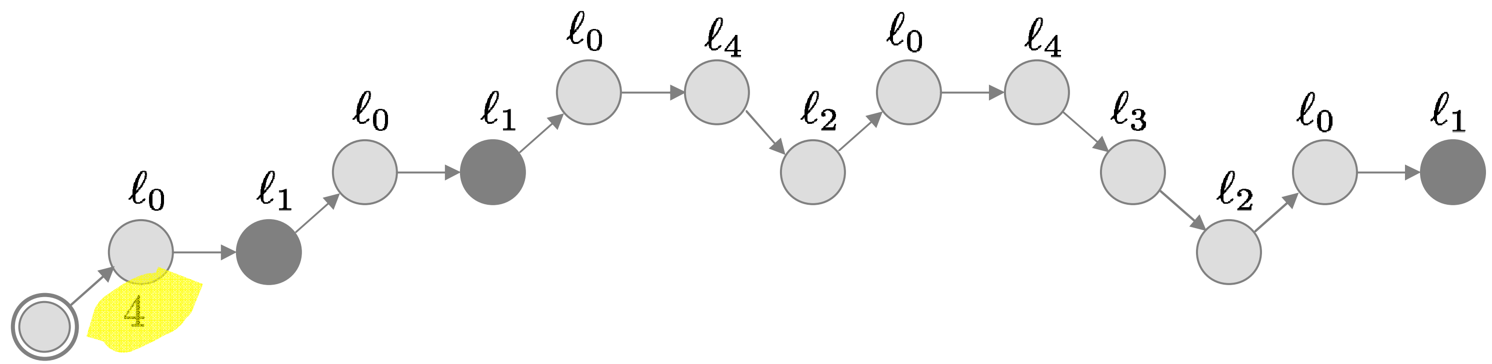
ℓ_1 : y := fib(x − 2)

ℓ_2 : z := fib(x − 1)

ℓ_3 : **return** y + z

end

ℓ_4: **return** 1



Fibonacci

$\trianglerighteq_x$

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then begin**

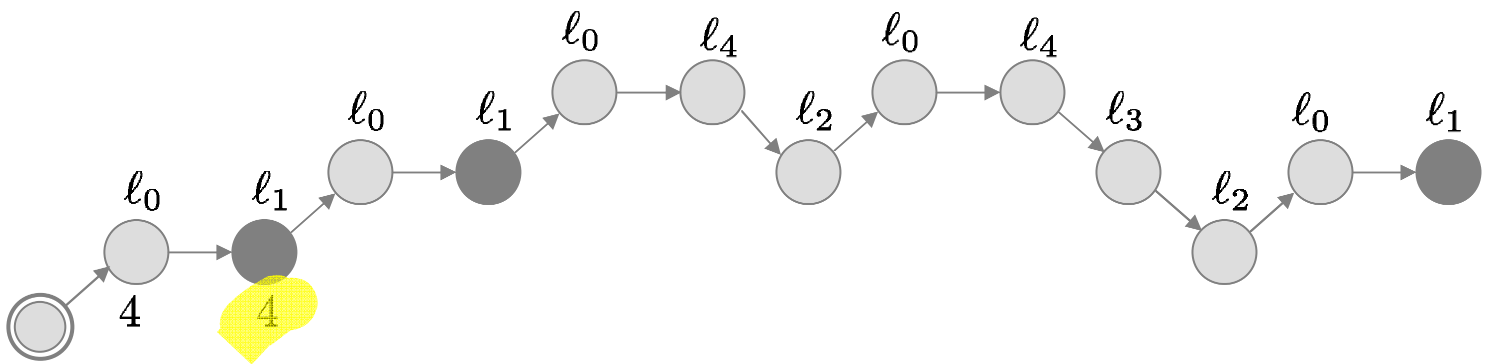
ℓ_1 : y := fib(x − 2)

ℓ_2 : z := fib(x − 1)

ℓ_3 : **return** y + z

end

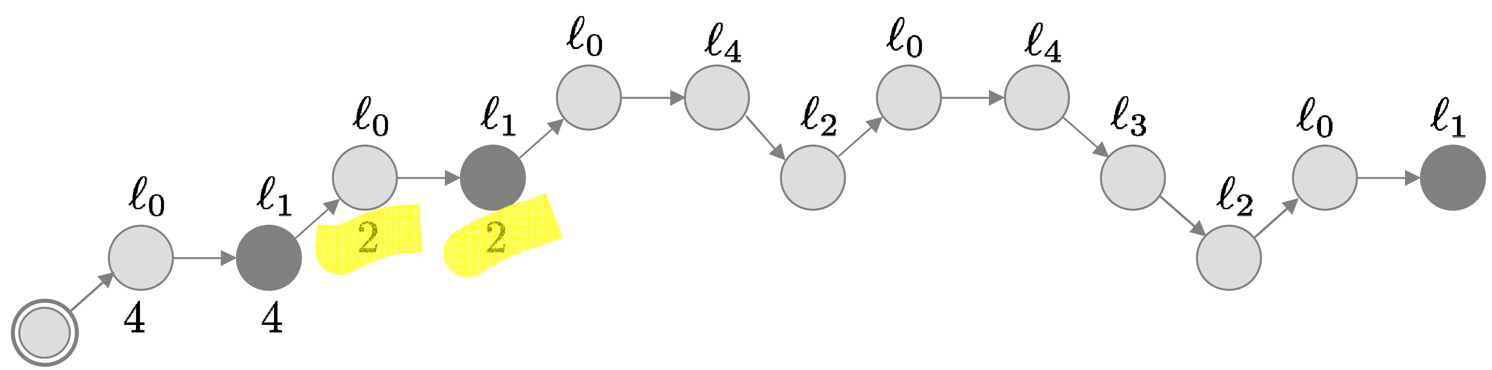
ℓ_4: **return** 1



Fibonacci

$\trianglerighteq_x$

procedure fib(x) **begin**
ℓ_0 : **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



Fibonacci

$\trianglerighteq_x$

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then begin**

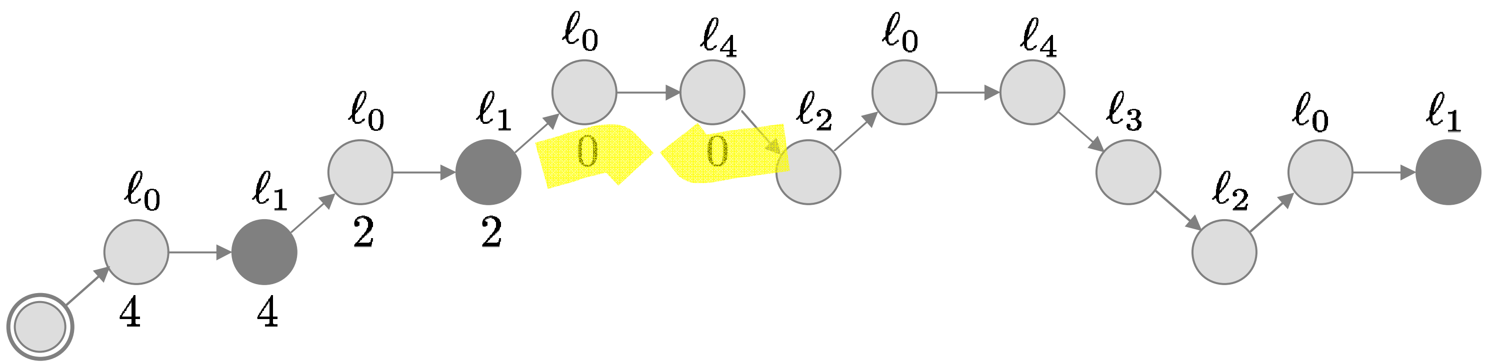
ℓ_1 : y := fib(x − 2)

ℓ_2 : z := fib(x − 1)

ℓ_3 : **return** y + z

end

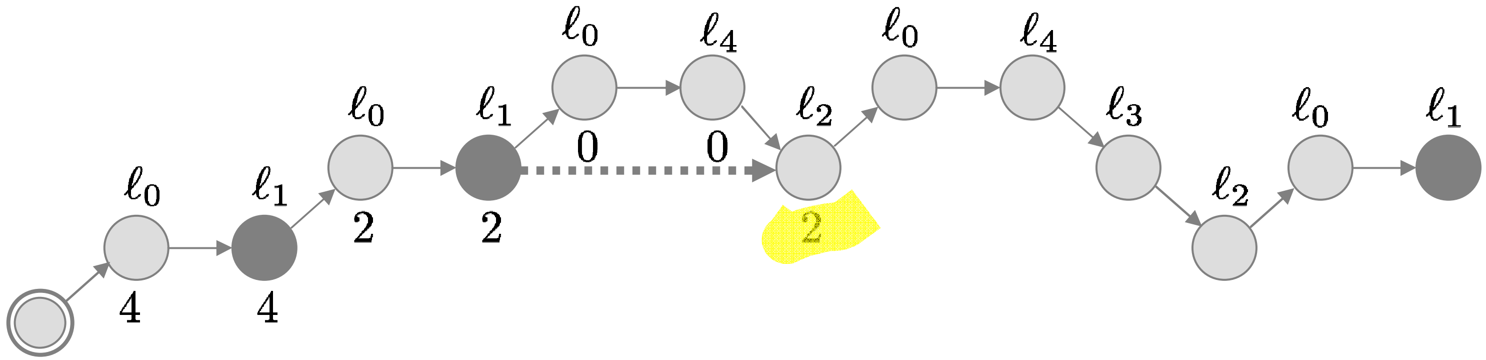
ℓ_4: **return** 1



Fibonacci

$\trianglerighteq_x$

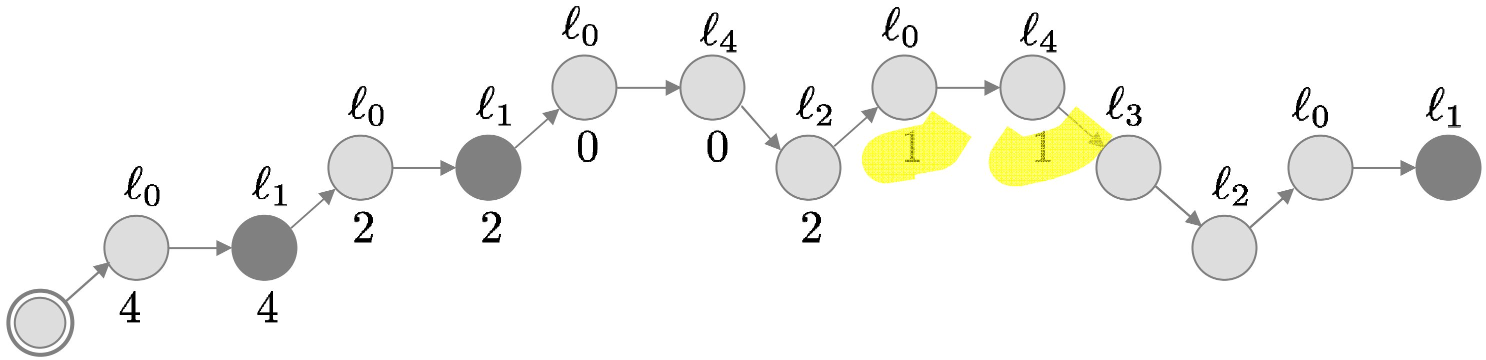
$$
\begin{array}{ll}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} & \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} & \\
\ell_1 : & \mathsf{y} := \mathsf{fib}(\mathsf{x} - 2) \\
\ell_2 : & \mathsf{z} := \mathsf{fib}(\mathsf{x} - 1) \\
\ell_3 : & \textbf{return } \mathsf{y} + \mathsf{z} \\
\quad \textbf{end} & \\
\ell_4 : \textbf{return } 1 &
\end{array}
$$



Fibonacci

$\unrhd_x$

$$
\begin{array}{ll}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} & \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} & \\
\ell_1 : & \mathsf{y} := \mathsf{fib}(\mathsf{x} - 2) \\
\ell_2 : & \mathsf{z} := \mathsf{fib}(\mathsf{x} - 1) \\
\ell_3 : & \textbf{return } \mathsf{y} + \mathsf{z} \\
\quad \textbf{end} & \\
\ell_4 : \textbf{return } 1 &
\end{array}
$$



Fibonacci

$\trianglerighteq_x$

procedure fib(x) **begin**

ℓ_0 : **if** $x > 1$ **then begin**

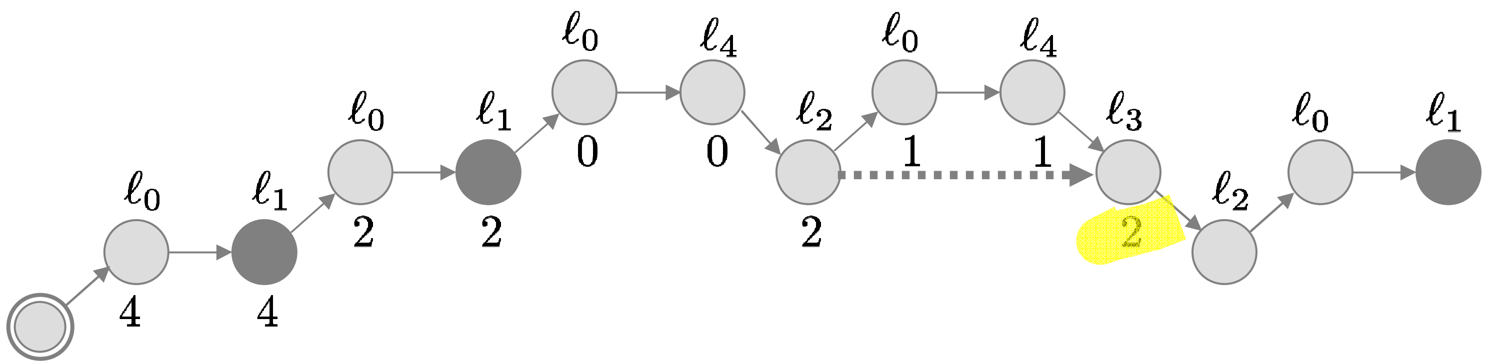
ℓ_1 : $\quad y :=$ fib$(x-2)$

ℓ_2 : $\quad z :=$ fib$(x-1)$

ℓ_3 : $\quad$ **return** $y + z$

$\quad$ **end**

ℓ_4: **return** 1



Fibonacci

$\trianglerighteq_x$

procedure fib(x) **begin**

ℓ_0 : **if** $x > 1$ **then begin**

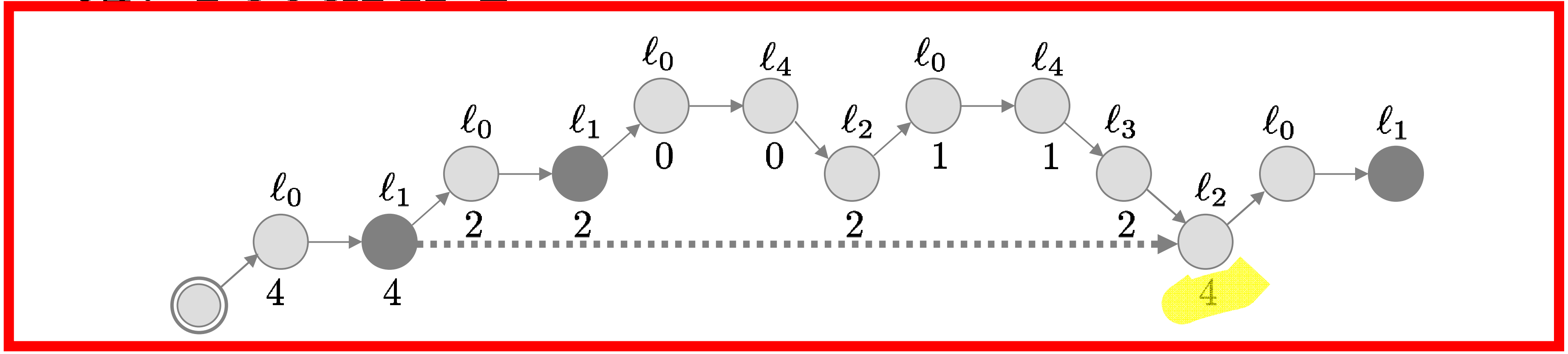
ℓ_1 : $\quad y := \text{fib}(x - 2)$

ℓ_2 : $\quad z := \text{fib}(x - 1)$

ℓ_3 : $\quad$ **return** $y + z$

end

ℓ_4: **return** 1



Fibonacci

$\trianglerighteq_x$

procedure fib(x) **begin**

ℓ_0 : **if** $x > 1$ **then begin**

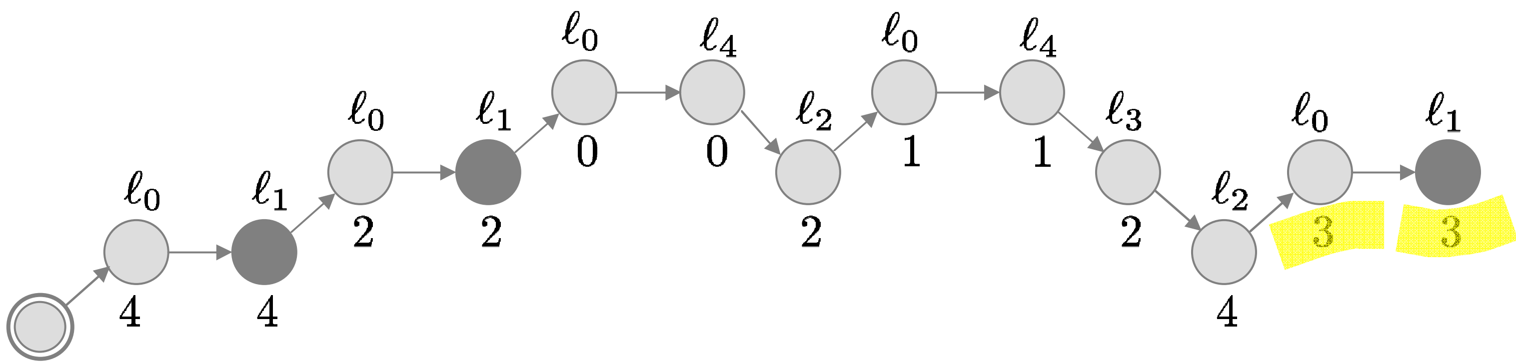
ℓ_1 : $\quad y := \text{fib}(x-2)$

ℓ_2 : $\quad z := \text{fib}(x-1)$

ℓ_3 : $\quad$ **return** $y + z$

end

ℓ_4: **return** 1



Fibonacci

Microsoft Research Cambridge

$\trianglerighteq_{\mathsf{x}}$

$\textbf{procedure}\ \mathsf{fib(x)}\ \textbf{begin}$

$\ell_0 :\ \textbf{if}\ \mathsf{x} > 1\ \textbf{then begin}$

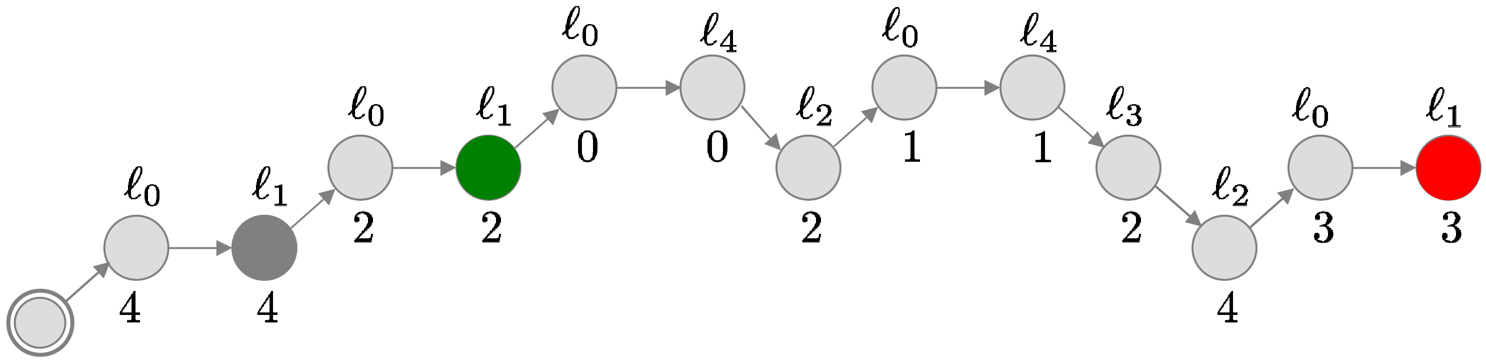
$\ell_1 :\qquad \mathsf{y} := \mathsf{fib(x-2)}$

$\ell_2 :\qquad \mathsf{z} := \mathsf{fib(x-1)}$

$\ell_3 :\qquad \textbf{return}\ \mathsf{y} + \mathsf{z}$

$\qquad \textbf{end}$

$\ell_4 :\ \textbf{return}\ 1$



Fibonacci

$\trianglerighteq_x$

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then begin**

ℓ_1 : y := fib(x − 2)

ℓ_2 : z := fib(x − 1)

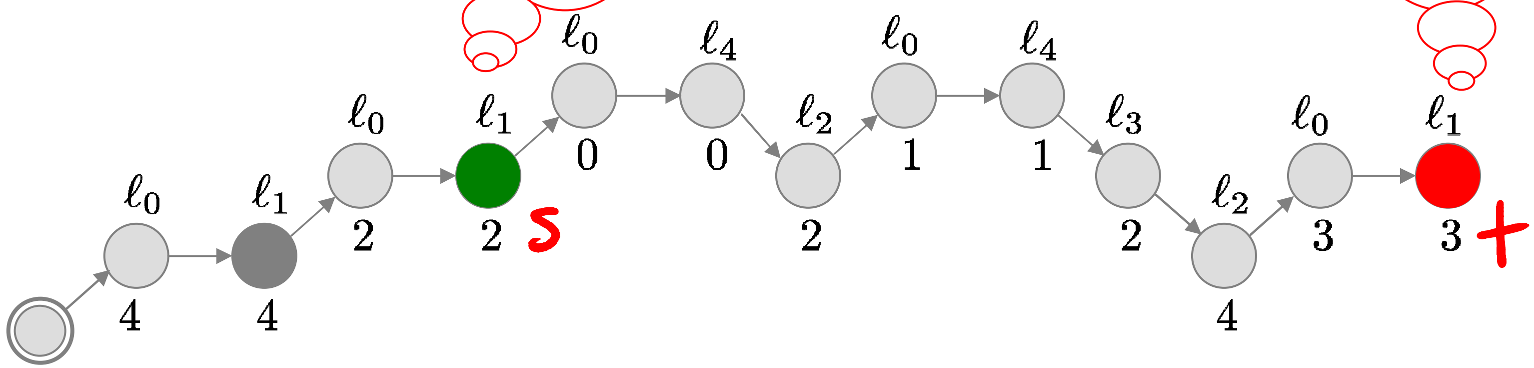
ℓ_3 : **return**

end

ℓ_4: **return**

x = 2

x = 3



Fibonacci

Microsoft Research Cambridge

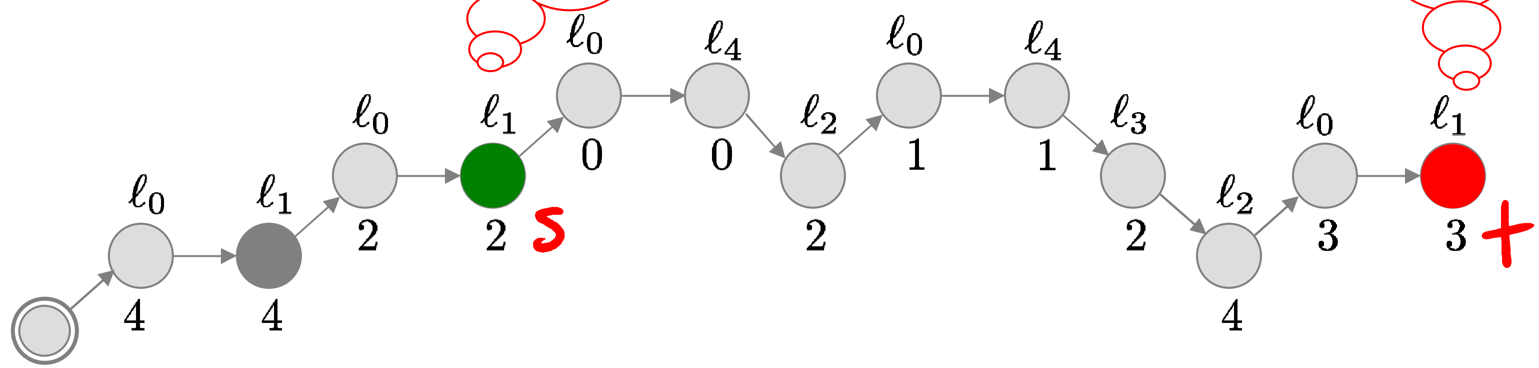
$\sqsupseteq_x$

$$
\begin{aligned}
&\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin}\\
&\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin}\\
&\ell_1 : \quad \mathsf{y} := \mathsf{fib}(\mathsf{x}-2)\\
&\ell_2 : \quad \mathsf{z} := \mathsf{fib}(\mathsf{x}-1)\\
&\ell_3 : \quad \textbf{return}\\
&\quad \textbf{end}\\
&\ell_4 : \textbf{return}
\end{aligned}
$$

$\boxed{(s,t) \notin \sqsupseteq_x}$

$\mathsf{x} = 2$

$\mathsf{x} = 3$



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"

$\trianglerighteq_x$

procedure fib(x) **begin**

ℓ_0 : **if** $x > 1$ **then begin**

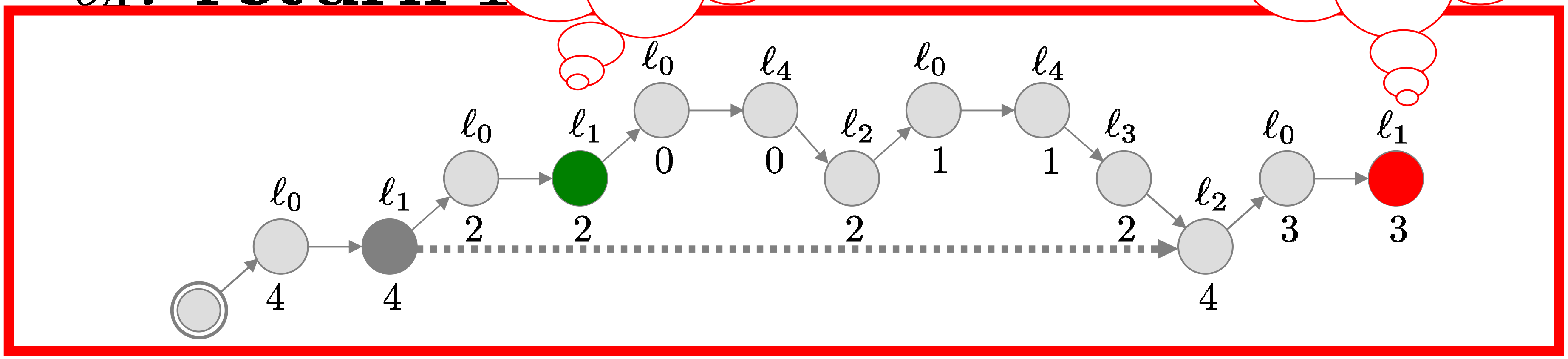
ℓ_1 : $y := \mathsf{fib}(x-2)$

ℓ_2 : $z := \mathsf{fib}(x-1)$

ℓ_3 : **return**

end

ℓ_4: **return**



Removing recursion

procedure fib(x) begin

ℓ_0 : if x > 1 then be

ℓ_1 : y := fib(x −

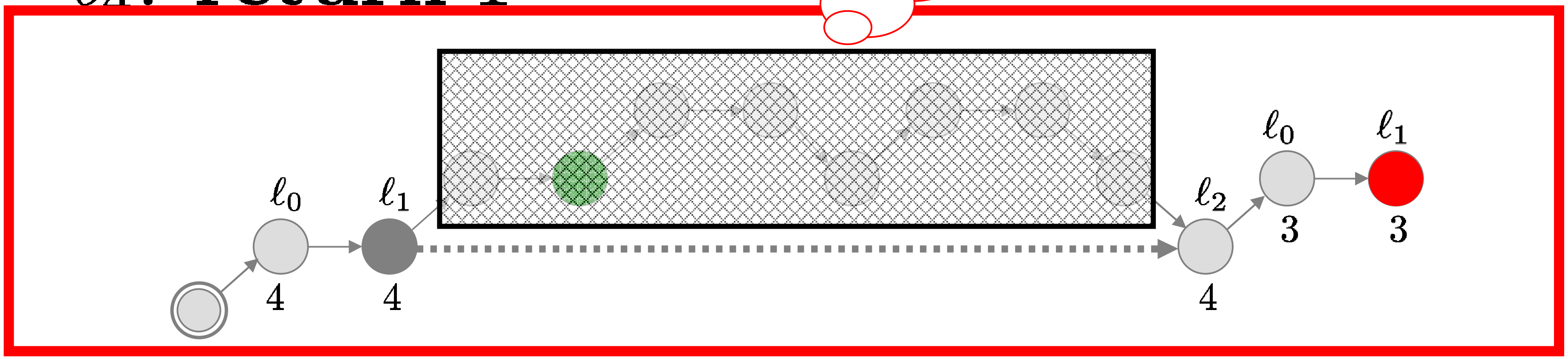
ℓ_2 : z := fib(x −

ℓ_3 : return y +

end

ℓ_4: return 1

This case can happen only when the command terminates



Removing recursion

Microsoft Research Cambridge

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then** **be**

ℓ_1 : y := fib(x −

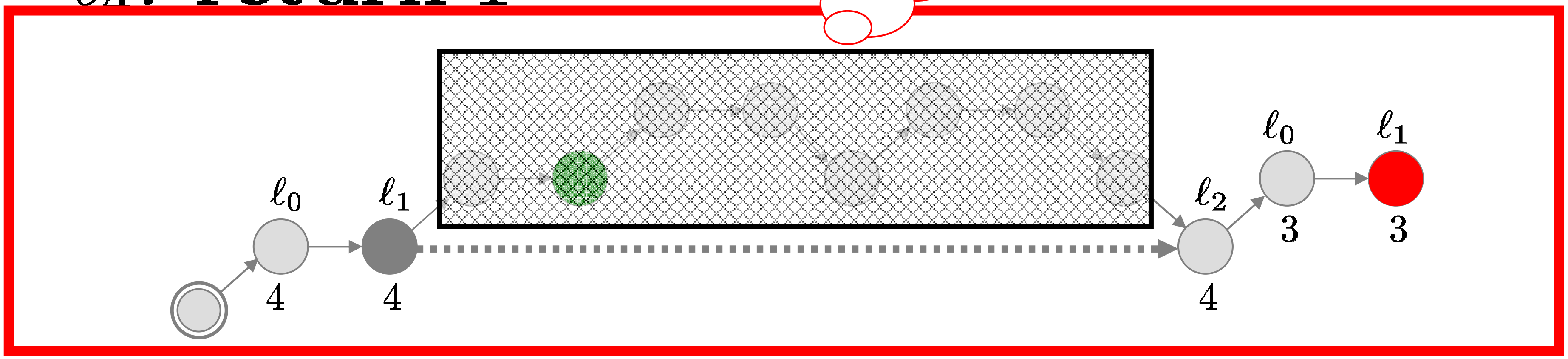
ℓ_2 : z := fib(x −

ℓ_3 : **return** y +

end

ℓ_4: **return** 1

If the command terminates we are better off with the summary



Removing recursion

procedure fib(x) begin

ℓ_0 : **if** x > 1 **then** **be**

ℓ_1 : y := fib(x −

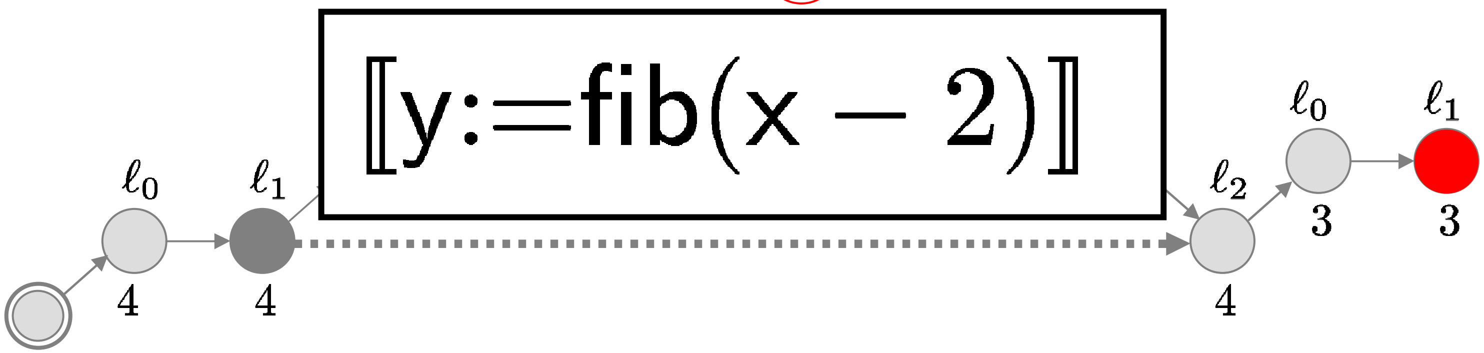
ℓ_2 : z := fib(x −

ℓ_3 : **return** y +

end

ℓ_4: **return** 1

If the command terminates we are better off with the summary



Removing recursion

$\textbf{procedure}$ fib(x) $\textbf{begin}$

ℓ_0 : $\textbf{if}$ x > 1 $\textbf{then be}$

ℓ_1 : y := fib(x −

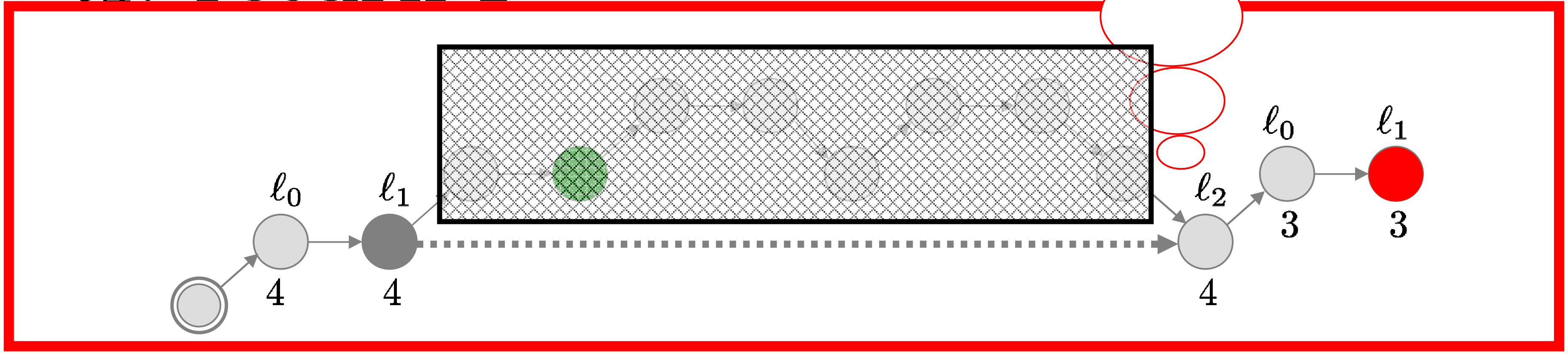
ℓ_2 : z := fib(x −

ℓ_3 : $\textbf{return}$ y +

$\textbf{end}$

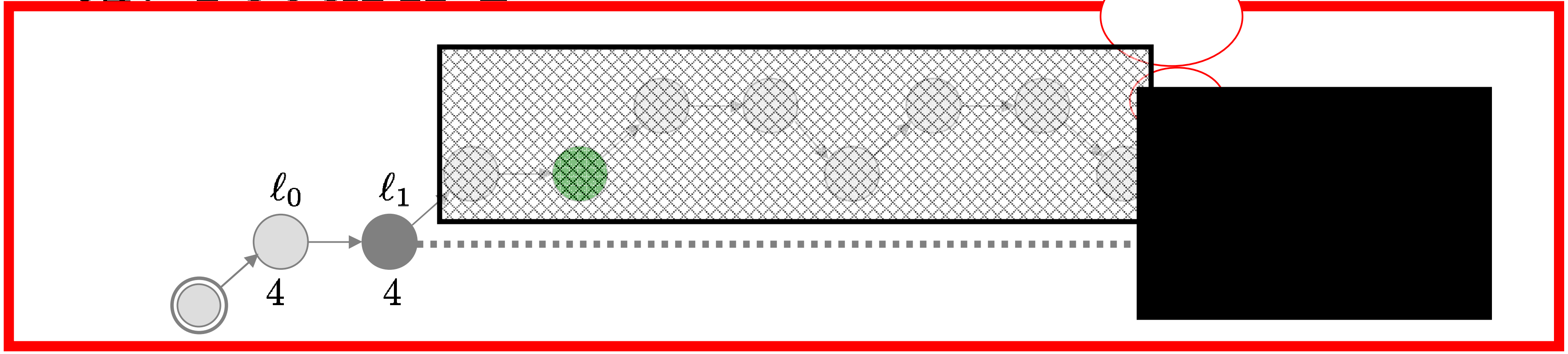
ℓ_4: $\textbf{return}$ 1

If the command doesn't terminate we'll never get back to this stack depth



procedure fib(x) **begin**
ℓ_0 : **if** x > 1 **then** **be**
ℓ_1 : y := fib(x −
ℓ_2 : z := fib(x −
ℓ_3 : **return** y +
end
ℓ_4: **return** 1

If the command doesn't terminate we'll never get back to this stack depth



Removing recursion

procedure fib(x) **begin**

ℓ_0 : **if** x > 1 **then begin**

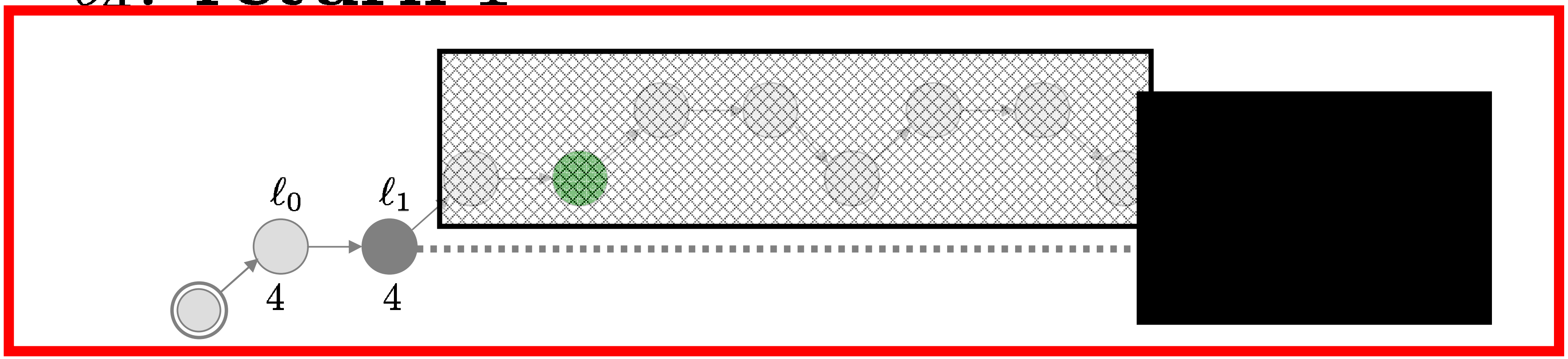
ℓ_1 : y := fib(x − 2)

ℓ_2 : z := fib(x − 1)

ℓ_3 : **return** y + z

end

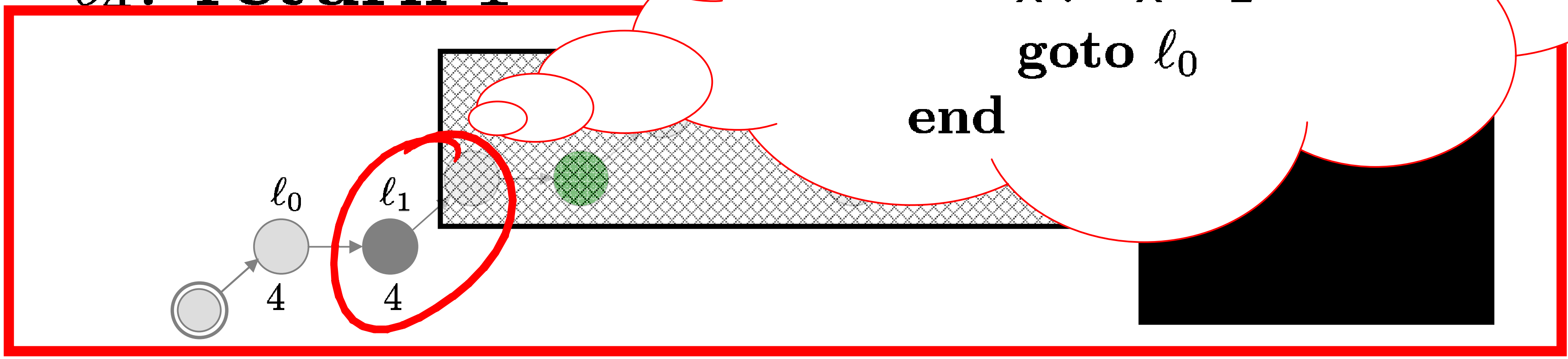
ℓ_4: **return** 1



Removing recursion

$$
\begin{array}{ll}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} & \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} & \\
\ell_1 : \qquad \mathsf{y} := \mathsf{fib}(\mathsf{x}-2) & \\
\ell_2 : \qquad \mathsf{z} := \mathsf{fib}(\mathsf{x}-1) & \\
\ell_3 : \qquad \textbf{return } \mathsf{y} + & \\
\qquad \textbf{end} & \\
\ell_4 : \textbf{ return } 1 &
\end{array}
$$

$$
\begin{array}{l}
\ell_1 : \textbf{if } * \textbf{ then begin} \\
\qquad \textbf{use } [\![\mathsf{y}{:=}\mathsf{fib}(\mathsf{x}-2)]\!] \\
\textbf{end else begin} \\
\qquad \mathsf{x} := \mathsf{x} - 2 \\
\qquad \textbf{goto } \ell_0 \\
\textbf{end}
\end{array}
$$



Removing recursion

$$\textbf{procedure}\ \mathsf{fib(x)}\ \textbf{begin}$$
$$\ell_0 : \textbf{if}\ \mathsf{x} > 1\ \textbf{then begin}$$
$$\ell_1 : \quad \mathsf{y} := \mathsf{fib(x-2)}$$
$$\ell_2 : \quad \mathsf{z} := \mathsf{fib(x-1)}$$

Recursive programs

→ Program transformation for termination is unaware of recursion

→ ... ecursion are

Hard to execute but easier to prove

→ T... .

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

16

$$\ell_1 : \textbf{if} * \textbf{then begin}$$
$$\quad \textbf{use}\ [\![\mathsf{y{:=}fib(x-2)}]\!]$$
$$\textbf{end else begin}$$
$$\quad \mathsf{x} := \mathsf{x} - 2$$
$$\quad \textbf{goto}\ \ell_0$$
$$\textbf{end}$$

Removing recursion

$$
\begin{array}{l}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} \\
\ell_1 : \qquad \textbf{if } * \ldots \\
\ell_2 : \qquad \textbf{if } * \ldots \\
\ell_3 : \qquad \textbf{return } \mathsf{y} + \mathsf{z} \\
\qquad \textbf{end} \\
\ell_4 \text{: } \textbf{return } 1 \\
\textbf{end}
\end{array}
$$

Removing recursion

$$
\begin{array}{l}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} \\
\ell_1 : \qquad \textbf{if } * \ldots \\
\ell_2 : \qquad \textbf{if } * \ldots \\
\ell_3 : \qquad \textbf{return } \mathsf{y} + \mathsf{z} \\
\qquad \textbf{end} \\
\ell_4 : \textbf{return } 1 \\
\textbf{end}
\end{array}
$$

Removing recursion

procedure fib(x) begin

ℓ_0 : if x > 1 then begin

ℓ_1 : if $* \ldots$

assume(false)

ℓ_2 : if $* \ldots$

ℓ_3 : return y + z

end

assume(false)

ℓ_4: return 1

end

Removing recursion

$$
\begin{array}{l}
\textbf{procedure } \mathsf{fib}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \textbf{if } \mathsf{x} > 1 \textbf{ then begin} \\
\ell_1 : \qquad \textbf{if } * \ldots \\
\ell_2 : \qquad \textbf{if } * \ldots \\
\ell_3 : \qquad \textbf{assume(false)} \\
\qquad \textbf{end} \\
\ell_4 : \textbf{ assume(false)} \\
\textbf{end}
\end{array}
$$

procedure fib(x) begin

ℓ_0 : if x $>$ 1 then begin

ℓ_1 : if $*$ ...

ℓ_2 : if $*$ ...

ℓ_3 : assume(false)

end

ℓ_4: assume(false)

end

"a wrong turn was made somewhere along the way......"

$$
\begin{array}{l}
\textbf{procedure } \mathsf{f}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \textbf{if } \mathsf{x} = 1 \textbf{ then begin} \\
\ell_1 : \qquad \mathsf{f}(0) \\
\qquad \textbf{end else begin} \\
\ell_2 : \qquad \mathsf{f}(1) \\
\qquad \textbf{end} \\
\ell_3 : \textbf{ return} \\
\textbf{end}
\end{array}
$$

Microsoft Research Cambridge

```
procedure f(x) begin
ℓ0 : if x = 1 then begin
ℓ1 :        f(0)
       end else begin
ℓ2 :        f(1)
       end
ℓ3: ~~return~~
end   assume(false)
```

if * then
    assume(false)
else
    x := ....
    goto ℓ0

Removing recursion

Microsoft Research Cambridge

procedure f(x) **begin**

ℓ_0 : **if** x = 1 **then begin**

ℓ_1 : f(0)

end else begin

ℓ_2 : f(1)

end

ℓ_3: ~~**return**~~

end assume(false)

then branch is never the right choice.....

if * then
assume(false)
else
x := ....
goto ℓ_0

$$
\begin{array}{l}
\textbf{procedure } \mathsf{f}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \mathsf{y} := 1 \\
\ell_1 : \textbf{while } \mathsf{y} \geq 0 \textbf{ begin} \\
\ell_2 : \qquad \mathsf{f}(\mathsf{x} - \mathsf{y}) \\
\ell_3 : \qquad \mathsf{y} := \mathsf{y} - 1 \\
\qquad \textbf{end} \\
\ell_4 : \textbf{ return} \\
\textbf{end}
\end{array}
$$

Removing recursion

$$
\begin{array}{l}
\textbf{procedure } \mathsf{f}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \mathsf{y} := 1 \\
\ell_1 : \textbf{while } \mathsf{y} \geq 0 \textbf{ begin} \\
\ell_2 : \qquad \mathsf{f}(\mathsf{x} - \mathsf{y}) \\
\ell_3 : \qquad \mathsf{y} := \mathsf{y} - 1 \\
\qquad \textbf{end} \\
\ell_4 : \textbf{ return} \\
\textbf{end}
\end{array}
$$

Counter example to termination need both cases in the "if*"

Removing recursion

Microsoft Research Cambridge

$$
\begin{array}{l}
\textbf{procedure } \mathsf{f}(\mathsf{x}) \textbf{ begin} \\
\ell_0 : \mathsf{y} := 1 \\
\ell_1 : \textbf{while } \mathsf{y} \geq 0 \textbf{ begin} \\
\ell_2 : \qquad \mathsf{f}(\mathsf{x} - \mathsf{y}) \\
\ell_3 : \qquad \mathsf{y} := \mathsf{y} - 1 \\
\qquad \textbf{end} \\
\ell_4 : \textbf{ return} \\
\textbf{end}
\end{array}
$$

$\approxeq$ f(x−1);
f(x);

Removing recursion

$$
\begin{array}{l}
\textbf{procedure } \mathsf{f(x)} \textbf{ begin} \\
\ell_0 : \mathsf{y} := 1 \\
\ell_1 : \textbf{if } \mathsf{x} \geq 0 \textbf{ begin} \\
\ell_2 : \qquad\quad \mathsf{y} := \mathsf{x} * \mathsf{f}(\mathsf{x} - 1) \\
\qquad \textbf{end} \\
\ell_3 \text{:} \ \textbf{return } \mathsf{y} \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\textbf{procedure } \mathsf{f(x)} \textbf{ begin} \\
\ell_0 : \mathsf{y} := 1 \\
\ell_1 : \textbf{if } \mathsf{x} \geq 0 \textbf{ begin} \\
\ell_2 : \qquad \mathsf{y} := \mathsf{x} * \boxed{\mathsf{f(x-1)}} \\
\qquad \textbf{end} \\
\ell_3 : \textbf{ return } \mathsf{y} \\
\textbf{end}
\end{array}
$$

summary

"true"

suffices.....

Underapproximation

- Imagine that we have relational summaries that underapproximate partial-correctness semantics

- We can use these summaries to prove non-terminating using the same technique

$$\begin{array}{l}\textbf{while } \mathsf{x} > 0 \textbf{ do}\\ \qquad \mathsf{f}(\mathsf{x});\\ \qquad \mathsf{x} := \mathsf{x} + 1;\\ \textbf{done}\end{array}$$

Underapproximation

```
procedure f(x) begin
    if x > 0 then
        f(x − 1);
    fi
    return;
end
```

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

Underapproximation

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

$$
\begin{array}{l}
\textbf{while } \mathrm{x} > 0 \textbf{ do} \\
\qquad \mathrm{f(x)}; \\
\qquad \mathrm{x} := \mathrm{x} + 1; \\
\textbf{done}
\end{array}
$$

Underapproximation

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

```
while x > 0 do
    use x' = x;
    x := x + 1;
done
```

Underapproximation

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

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

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

Implementation

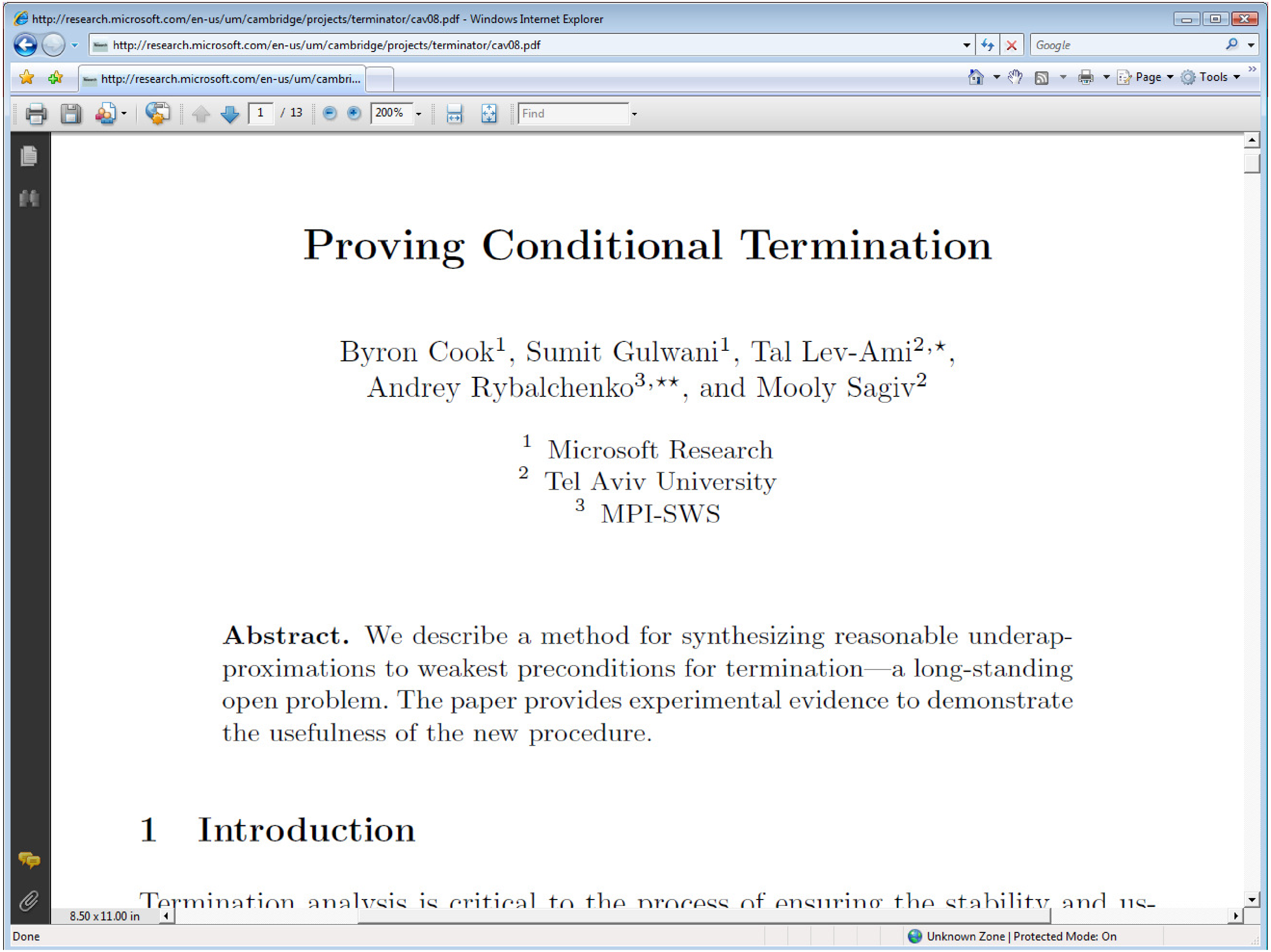
Microsoft Research Cambridge

```
procedure fib(x) begin
ℓ0 : if x > 1 then begin
ℓ1 :     if * then
             y := fib'(x − 2);
         else
             x := x − 2;
             goto ℓ0;
         fi
ℓ2 :     if * then
             z := fib'(x − 1);
         else
             x := x − 1;
             goto ℓ0;
         fi
ℓ3 :     return y + z;
     end
ℓ4: return 1;
end
```

```
procedure fib'(x) begin
    if x > 1 then begin
        /* IGNORE CUTPOINT */
        y := fib'(x − 2);
        /* IGNORE CUTPOINT */
        z := fib'(x − 1);
        return y + z;
    end
    return 1;
end
```

Outline

- Recursive programs
- Weakest preconditions



Proving Conditional Termination

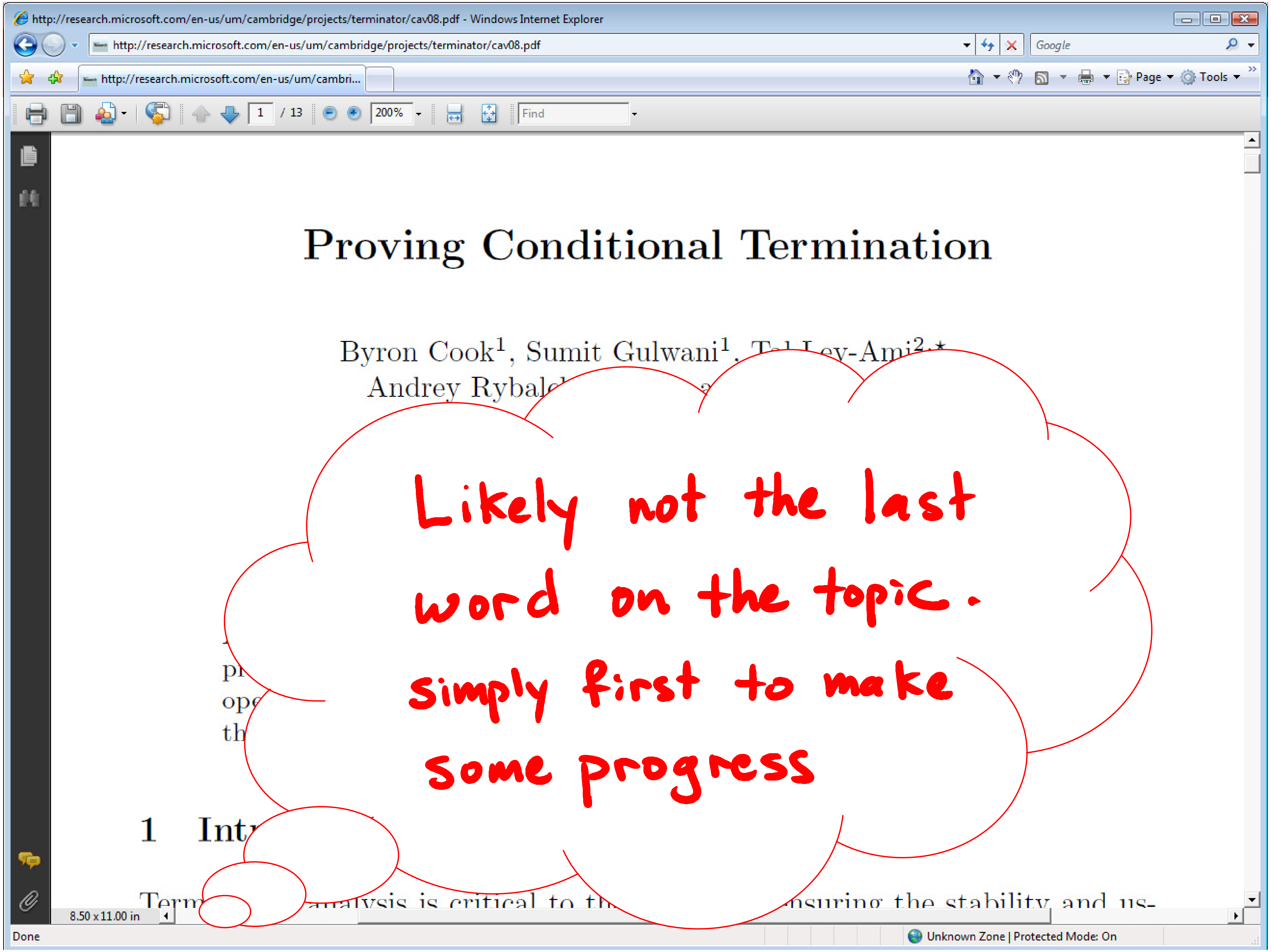
Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,⋆],
Andrey Rybalchenko[3,⋆⋆], and Mooly Sagiv[2]

[1] Microsoft Research
[2] Tel Aviv University
[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-


http://research.microsoft.com/en-us/um/cambridge/projects/terminator/cav08.pdf - Windows Internet Explorer
Proving Conditional Termination
Byron Cook[1], Sumit Gulwani[1], Tal Lev-Ami[2,*]
Likely not the last word on the topic.
Simply first to make some progress
Done
Unknown Zone | Protected Mode: On

Underapproximating weakest preconditions

$$\textsc{TerminationProver}(P) \subseteq \mathrm{WP}(P, \mathbf{true})$$

$$\textsc{TerminationProver}(P) \subseteq \mathrm{WP}(P, \mathbf{true})$$

returns true or false

Underapproximating weakest preconditions

Microsoft Research Cambridge

$$\textsc{TerminationProver}(P) \subseteq \boxed{\mathrm{WP}}(P, \mathbf{true})$$

Not WLP

$$\textsc{TerminationProver}(P) \subseteq \mathrm{WP}(P, \mathbf{true})$$

Wanted: the <u>right</u> precondition

Underapproximating weakest preconditions

$$\textsc{TerminationProver}(P) \subseteq \mathrm{WP}(P, \mathbf{true})$$

$$wp(P, c) = wlp(P, c) \wedge wp(P, true)$$

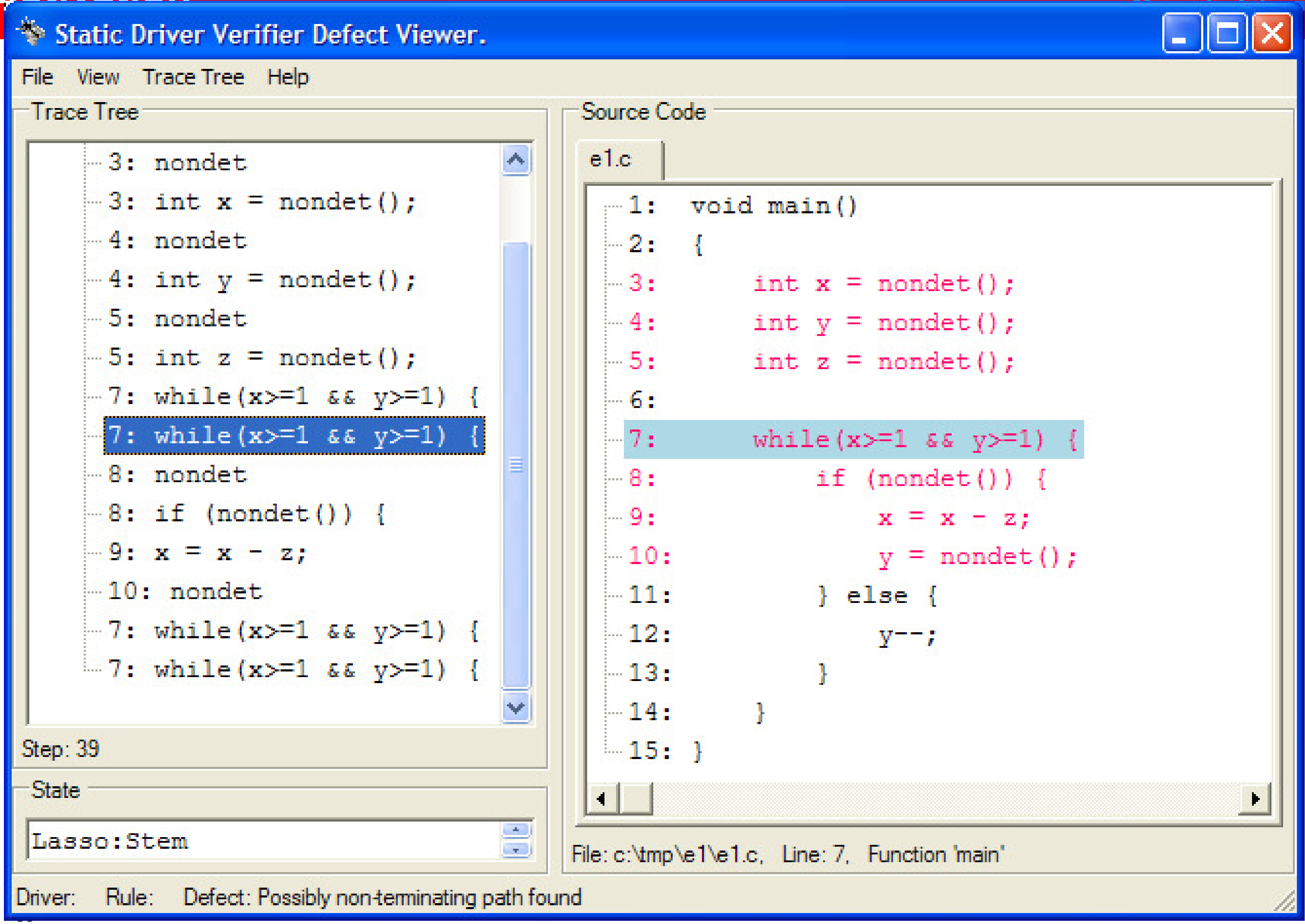
Underapproximating weakest preconditions

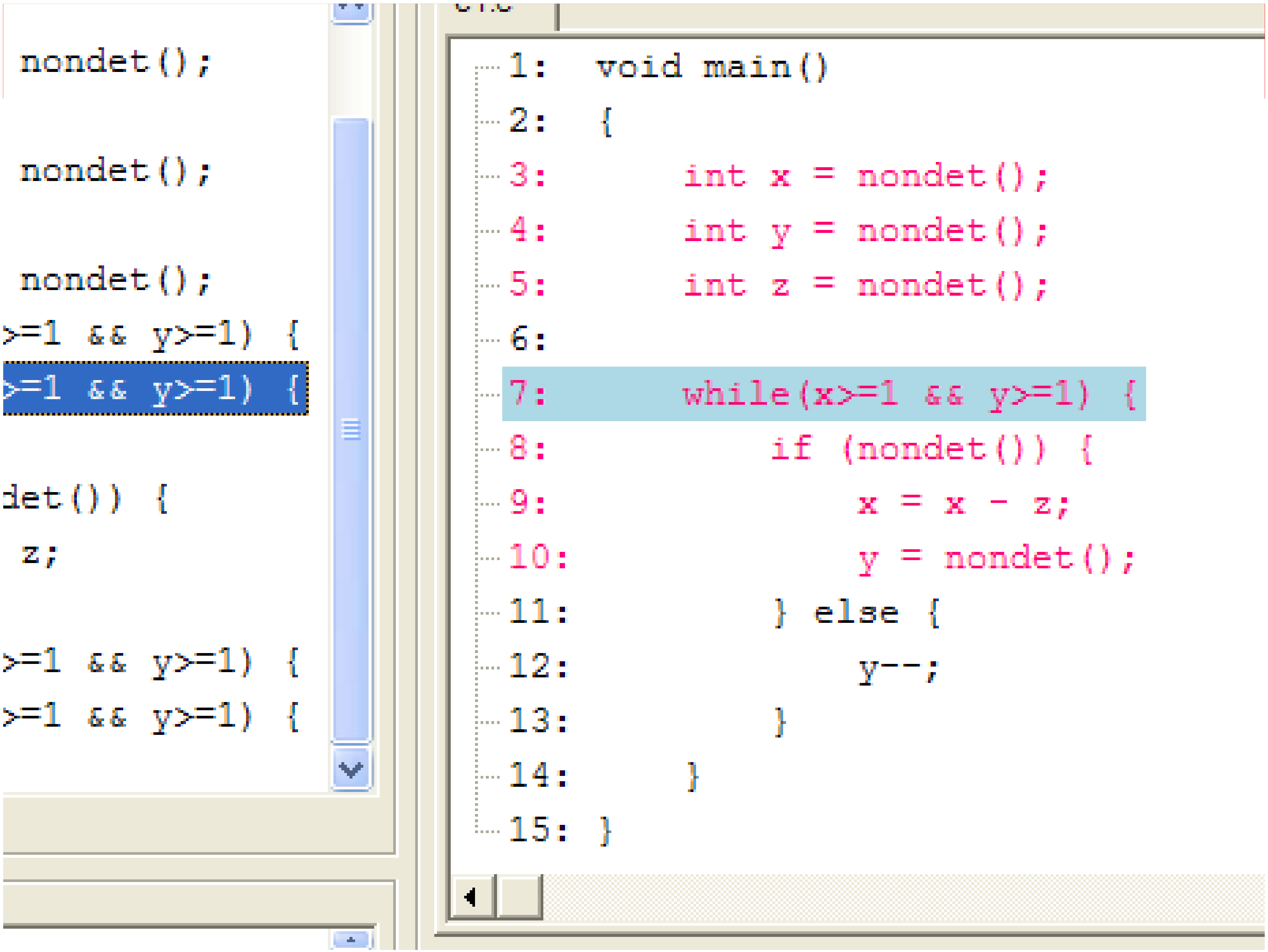
Techniques available for underapproximating WLP

$\mathrm{P}(P, \mathbf{true})$

$wp(P, C) = WLP(P, C) \wedge wp(P, \mathrm{true})$

Motivation


Static Driver Verifier Defect Viewer.
File View Trace Tree Help
Trace Tree
3: nondet
3: int x = nondet();
4: nondet
4: int y = nondet();
5: nondet
5: int z = nondet();
7: while(x>=1 && y>=1) {
7: while(x>=1 && y>=1) {
8: nondet
8: if (nondet()) {
9: x = x - z;
10: nondet
7: while(x>=1 && y>=1) {
7: while(x>=1 && y>=1) {
Step: 39
State
Lasso:Stem
Source Code
e1.c
1: void main()
2: {
3: int x = nondet();
4: int y = nondet();
5: int z = nondet();
6:
7: while(x>=1 && y>=1) {
8: if (nondet()) {
9: x = x - z;
10: y = nondet();
11: } else {
12: y--;
13: }
14: }
15: }
File: c:\tmp\e1\e1.c, Line: 7, Function 'main'
Driver: Rule: Defect: Possibly non-terminating path found

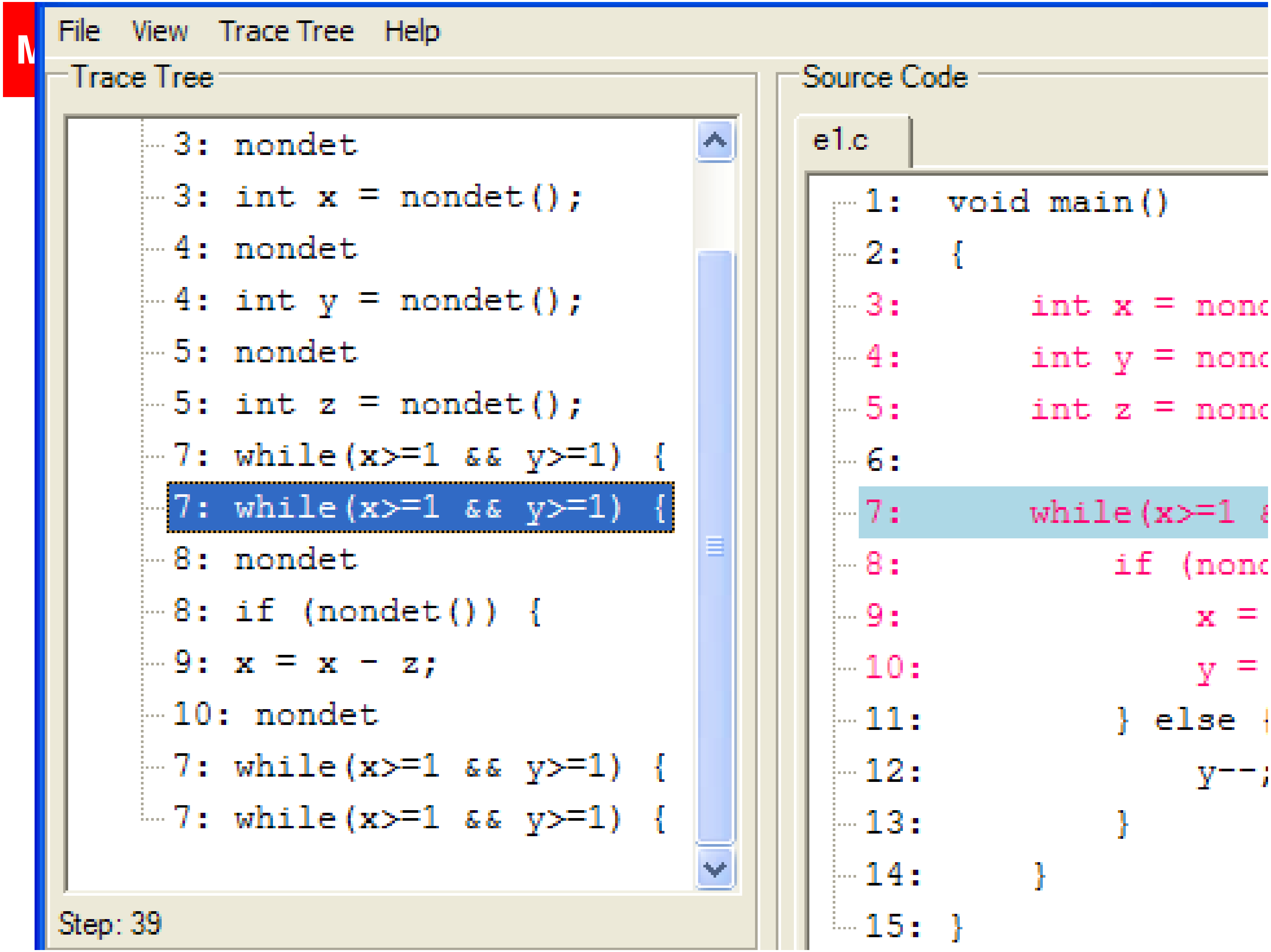

nondet();
nondet();
nondet();
>=1 && y>=1) {
>=1 && y>=1) {
det()) {
z;
>=1 && y>=1) {
>=1 && y>=1) {
1: void main()
2: {
3: int x = nondet();
4: int y = nondet();
5: int z = nondet();
6:
7: while(x>=1 && y>=1) {
8: if (nondet()) {
9: x = x - z;
10: y = nondet();
11: } else {
12: y--;
13: }
14: }
15: }

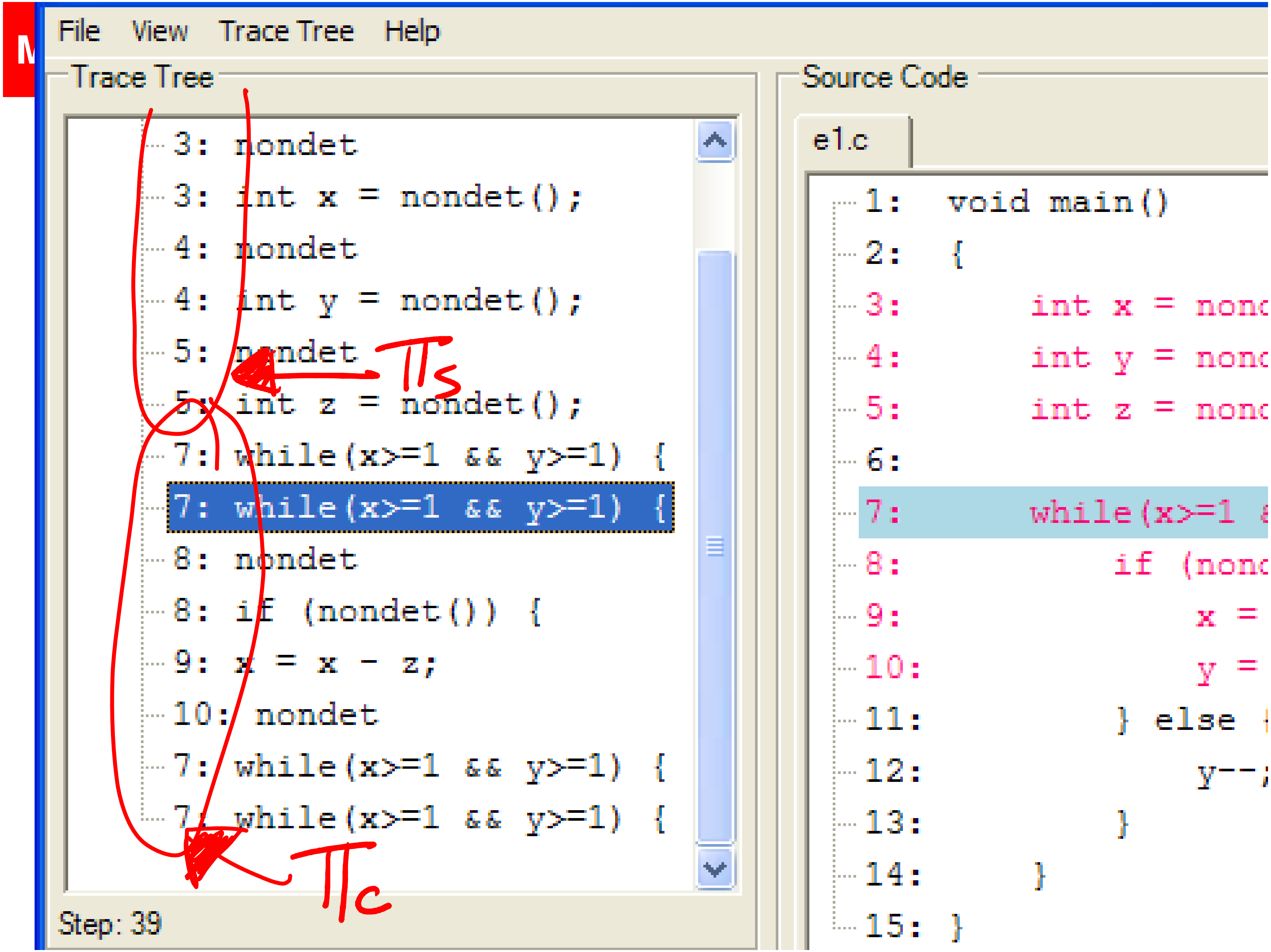
Does this code always terminate?

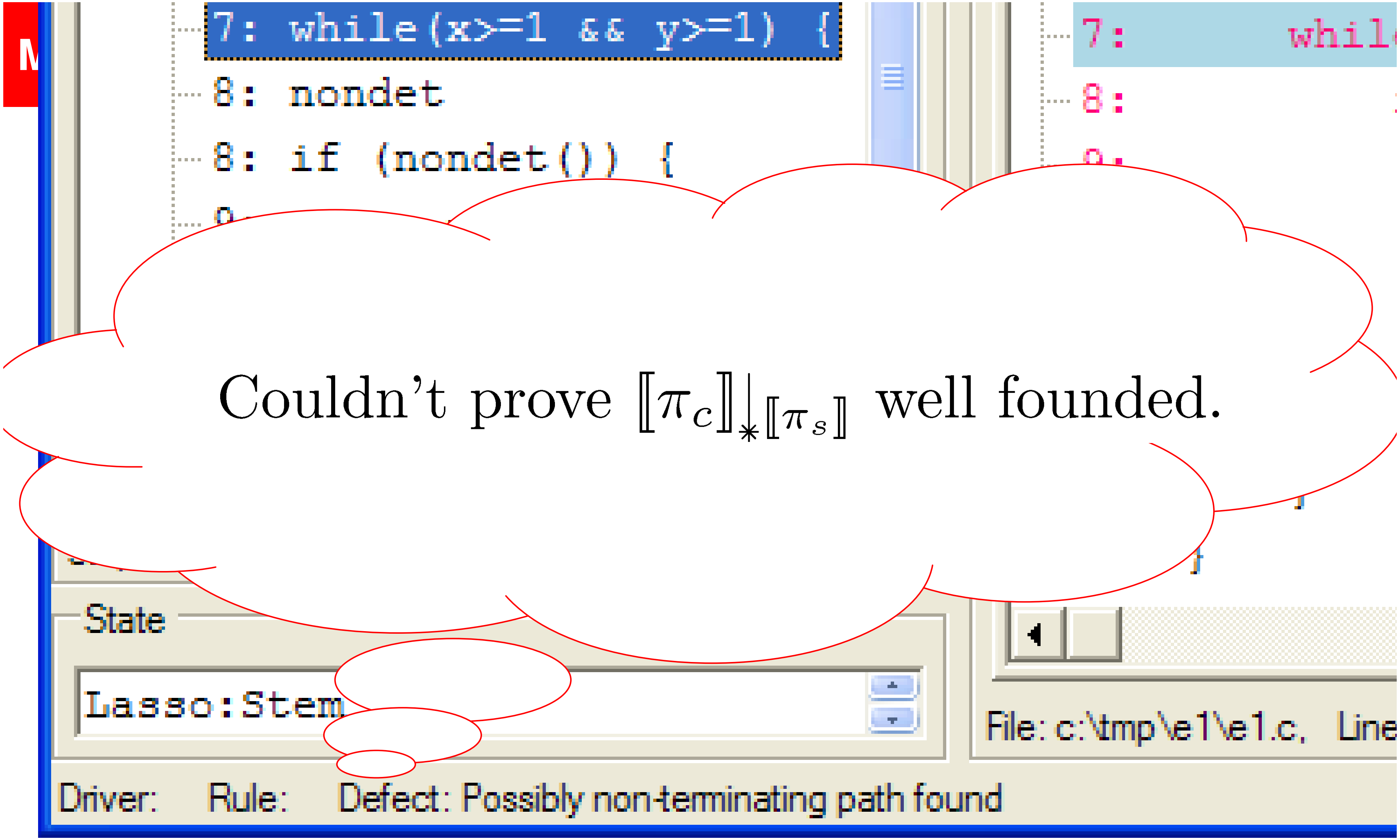
```
                              t();
                          nondet();
                        = nondet();

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

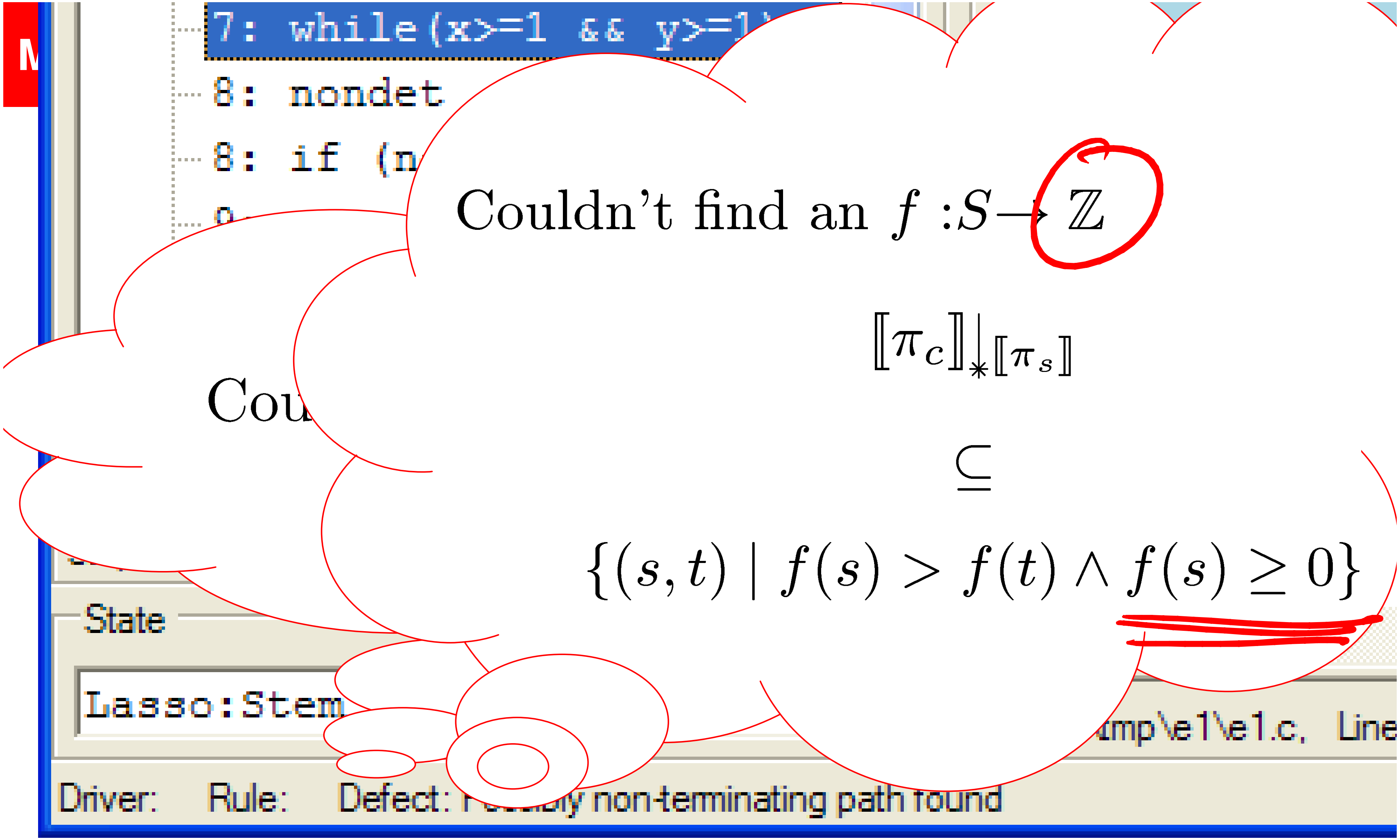
What's the weakest precondition?

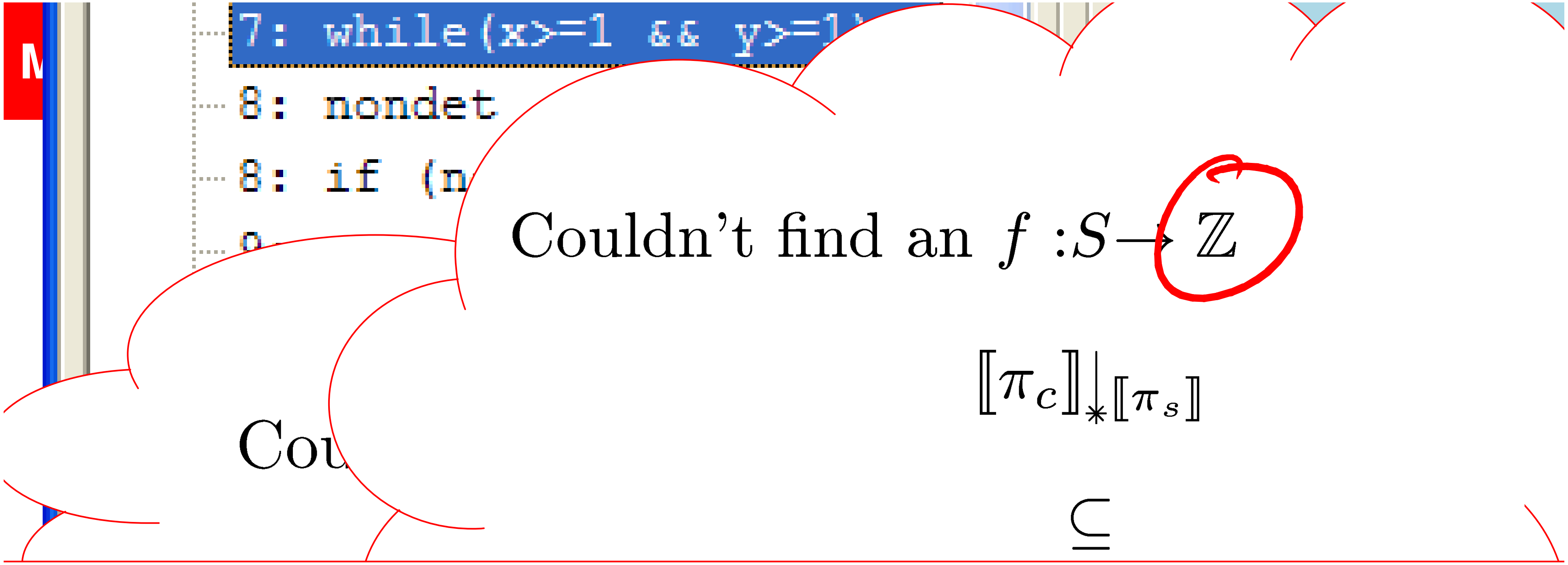
```
    while(x>=1 && y>=1) {
        if (nondet()) {
            x = x - z;
            y = nondet();
        } else {
            y--;
        }
    }
}
```


File View Trace Tree Help
Trace Tree
3: nondet
3: int x = nondet();
4: nondet
4: int y = nondet();
5: nondet
5: int z = nondet();
7: while(x>=1 && y>=1) {
7: while(x>=1 && y>=1) {
8: nondet
8: if (nondet()) {
9: x = x - z;
10: nondet
7: while(x>=1 && y>=1) {
7: while(x>=1 && y>=1) {
Step: 39
Source Code
e1.c
1: void main()
2: {
3: int x = nonc
4: int y = nonc
5: int z = nonc
6:
7: while(x>=1
8: if (nonc
9: x =
10: y =
11: } else
12: y--;
13: }
14: }
15: }


File View Trace Tree Help
Trace Tree
3: nondet
3: int x = nondet();
4: nondet
4: int y = nondet();
5: nondet
5: int z = nondet();
7: while(x>=1 && y>=1) {
7: while(x>=1 && y>=1) {
8: nondet
8: if (nondet()) {
9: x = x - z;
10: nondet
7: while(x>=1 && y>=1) {
7: while(x>=1 && y>=1) {
Step: 39
Source Code
e1.c
1: void main()
2: {
3: int x = nond
4: int y = nond
5: int z = nond
6:
7: while(x>=1 &
8: if (nond
9: x =
10: y =
11: } else {
12: y--;
13: }
14: }
15: }
Π_s
Π_c


7: while(x>=1 && y>=1) {
8: nondet
8: if (nondet()) {
Couldn't prove $[\![\pi_c]\!]\downarrow_{*[\![\pi_s]\!]}$ well founded.
State
Lasso:Stem
File: c:\tmp\e1\e1.c,
Driver:
Rule:
Defect: Possibly non-teminating path found


7: while(x>=1 && y>=1)
8: nondet
8: if (n
Couldn't find an $f : S \to \mathbb{Z}$
$[\![\pi_c]\!]\downarrow_{*[\![\pi_s]\!]}$
$\subseteq$
$\{(s,t) \mid f(s) > f(t) \wedge f(s) \geq 0\}$
Cou
State
Lasso:Stem
mp\e1\e1.c, Line
Driver: Rule: Defect:
non-teminating path found



C := true

while $\neg$Terminator(P, C) **do**

    let (π_s, π_c) be the counterexample to termination

    $C := C \wedge \textsc{PreSynth}(\pi_s, \pi_c)$

od

return C

PreSynth algorithm

- Find a set B such that for all $b \in B$

$$[\![\pi_c]\!]\downarrow_{*}[\![\pi_s]\!] \subseteq \{(s,t) \mid b(s) \geq 0\}$$

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

$$([\![\pi_c]\!]\downarrow_{*}[\![\pi_s]\!])\downarrow Q_b \subseteq \{(s,t) \mid b(s) > b(t)\}$$

- Return $\bigcup_{b \in B} \mathsf{WLP}(([\![\pi_c]\!]\downarrow_{*}[\![\pi_s]\!])^*, Q_b)$

PreSynth alg

C := true
while $\neg$Terminator(P, C) **do**
  let (π_s, π_c) be the counterexample to termination
  $C := C \wedge \textsc{PreSynth}(\pi_s, \pi_c)$
od
return C

- Return $\bigcup_{b\in B} \mathsf{WLP}(([\![\pi_c]\!]\downarrow_{*[\![\pi_s]\!]})^*, Q_b)$

Implementation

$$
\begin{array}{l}
R(X, X') := [\![\pi_c]\!]\downarrow_{*[\![\pi_s]\!]} \\
C(X) := \mathsf{false}; \\
B(X) := \mathsf{QELIM}(\exists X'.\ R(X, X')) \\
\textbf{foreach } \text{conjunct } b(X) \geq 0 \text{ in } B(X) \textbf{ do} \\
\quad\quad Q_b(X) := \mathsf{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X)) \\
\quad\quad C(X) := C(X) \vee \mathsf{WLP}(R^*(X, X'), Q_b(X)) \\
\textbf{done} \\
\textbf{return } C(X)
\end{array}
$$

Example

$$
\begin{aligned}
R &= \mathsf{x}' = \mathsf{x} - \mathsf{z} \\
&\wedge \mathsf{x} \geq 1 \\
&\wedge \mathsf{y} \geq 1 \\
&\wedge \mathsf{z}' = \mathsf{z} \\
X &= \{\mathsf{x}, \mathsf{y}, \mathsf{z}\} \\
X' &= \{\mathsf{x}', \mathsf{y}', \mathsf{z}'\}
\end{aligned}
$$

```
nondet();

nondet();

nondet();
>=1 && y>=1) {
>=1 && y>=1) {

det()) {
z;

>=1 && y>=1) {
>=1 && y>=1) {
```

```
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

$$
\begin{array}{lcl}
R & = & \mathsf{x}' = \mathsf{x} - \mathsf{z} \\
 & \wedge & \mathsf{x} \geq 1 \\
 & \wedge & \mathsf{y} \geq 1 \\
 & \wedge & \mathsf{z}' = \mathsf{z} \\
X & = & \{\mathsf{x}, \mathsf{y}, \mathsf{z}\} \\
X' & = & \{\mathsf{x}', \mathsf{y}', \mathsf{z}'\}
\end{array}
$$

$R(X, X') := [\![\pi_c]\!]|_{*[\![\pi_s]\!]}$
$C(X) := \mathsf{false};$
$B(X) := \mathsf{QELIM}(\exists X'.\ R(X, X'))$
foreach conjunct $b(X) \geq 0$ in $B(X)$ **do**
$\quad Q_b(X) := \mathsf{QELIM}(\forall X'.R(X, X') \Rightarrow b(X) > b(X))$
$\quad C(X) := C(X) \vee \mathsf{WLP}(R^*(X, X'), Q_b(X))$
done
return $C(X)$

Example

$$\begin{array}{lcl} R & = & \mathsf{x}' = \mathsf{x} - \mathsf{z} \\ & \wedge & \mathsf{x} \geq 1 \\ & \wedge & \mathsf{y} \geq 1 \\ & \wedge & \mathsf{z}' = \mathsf{z} \\ X & = & \{\mathsf{x}, \mathsf{y}, \mathsf{z}\} \\ X' & = & \{\mathsf{x}', \mathsf{y}', \mathsf{z}'\} \end{array}$$

$R(X, X') := [\![\pi_c]\!]\downarrow_{*}[\![\pi_s]\!]$
$C(X) := \mathsf{false};$
$\boxed{B(X) := \mathsf{QELIM}(\exists X'.\ R(X, X'))}$
foreach conjunct $b(X) \geq 0$ in $B(X)$ **do**
 $Q_b(X) := \mathsf{QELIM}(\forall X'.R(X, X') \Rightarrow b(X) > b(X))$
 $C(X) := C(X) \vee \mathsf{WLP}(R^*(X, X'), Q_b(X))$
done
return $C(X)$

Example

$$
\begin{aligned}
R &= \mathsf{x}' = \mathsf{x} - \mathsf{z} \\
&\wedge \ \mathsf{x} \geq 1 \\
&\wedge \ \mathsf{y} \geq 1 \\
&\wedge \ \mathsf{z}' = \mathsf{z} \\
X &= \{\mathsf{x}, \mathsf{y}, \mathsf{z}\} \\
X' &= \{\mathsf{x}', \mathsf{y}', \mathsf{z}'\}
\end{aligned}
$$

$$\mathsf{QELIM}(\exists \mathsf{x}', \mathsf{y}', \mathsf{z}'.\ R) = \mathsf{x} - 1 \geq 0 \wedge \mathsf{y} - 1 \geq 0$$

$R(X, X') := [\![\pi_s]\!]_*[\![\pi_s]\!]$

$C(X) :=$ false;

$\boxed{B(X) := \mathsf{QELIM}(\exists X'.\ R(X, X'))}$

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

$\quad Q_b(X) := \mathsf{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X))$

$\quad C(X) := C(X) \vee \mathsf{WLP}(R^*(X, X'), Q_b(X))$

done

return $C(X)$

Example

$$\mathsf{QELIM}(\exists \mathsf{x}', \mathsf{y}', \mathsf{z}'.\ R) = \mathsf{x} - 1 \geq 0 \wedge \mathsf{y} - 1 \geq 0$$

$$
\begin{array}{rcl}
R & = & \mathsf{x}' = \mathsf{x} - \mathsf{z} \\
 & \wedge & \mathsf{x} \geq 1 \\
 & \wedge & \mathsf{y} \geq 1 \\
 & \wedge & \mathsf{z}' = \mathsf{z} \\
X & = & \{\mathsf{x}, \mathsf{y}, \mathsf{z}\} \\
X' & = & \{\mathsf{x}', \mathsf{y}', \mathsf{z}'\}
\end{array}
$$

$R(X, X') := [\![\pi_s]\!]_*[\![\pi_s]\!]$

$C(X) := \mathsf{false};$

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

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

$\quad Q_b(X) := \mathsf{QELIM}(\forall X'. R(X, X') \Rightarrow b(X) > b(X))$

$\quad C(X) := C(X) \vee \mathsf{WLP}(R^*(X, X'), Q_b(X))$

done

return $C(X)$

$$
\begin{aligned}
R &= x' = x - z \\
&\wedge\ x \geq 1 \\
&\wedge\ y \geq 1 \\
&\wedge\ z' = z \\
X &= \{x, y, z\} \\
X' &= \{x', y', z'\}
\end{aligned}
$$

QELIM$(\exists x', y', z'.\ R) = x - 1 \geq 0 \wedge y - 1 \geq 0$

$i.e.$ $x - 1$ or $y - 1$

$R(X, X$ …

$C(X)$ …

$B(X) :=$ QELI… $X'.\ R(X, X'))$

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

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

$\quad C(X) := C(X) \vee$ WLP$(R^*(X, X'), Q_b(X))$

done

return $C(X)$

Example

Microsoft Research Cambridge

$$
\begin{array}{rcl}
R & = & \mathsf{x}' = \mathsf{x} - \mathsf{z} \\
 & \wedge & \mathsf{x} \geq 1 \\
 & \wedge & \mathsf{y} \geq 1 \\
 & \wedge & \mathsf{z}' = \mathsf{z} \\
X & = & \{\mathsf{x}, \mathsf{y}, \mathsf{z}\} \\
X' & = & \{\mathsf{x}', \mathsf{y}', \mathsf{z}'\}
\end{array}
$$

$R(X, X') := [\![\pi_c]\!]\downarrow_{*[\![\pi_s]\!]}$

$C(X) :=$ false;

$B(X) :=$ QELIM$(\exists X'.\ R(X, X'))$

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

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

  $C(X) := C(X) \vee$ WLP$(R^*(X, X'), Q_b(X))$

done

return $C(X)$

Example

$$R = x' = x - z \wedge x \geq 1 \wedge y \geq 1$$

$\dots, y', z'\}$

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

$C(X) :=$ false

$B(X) := \mathsf{QELIM}(\exists X'.\ R(\dots))$

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

$\quad Q_b(X) := \mathsf{QELIM}(\forall X'.R(X, X') \Rightarrow b(X) > b(X))$

$\quad C(X) := C(X) \vee \mathsf{WLP}(R^*(X, X'), Q_b(X))$

done

return $C(X)$

Example

$$R = \mathsf{x}' = \mathsf{x} - \mathsf{z}$$
$$\wedge \ \mathsf{x} \geq 1$$
$$\wedge \ \mathsf{y} \geq 1$$

$\ldots, \mathsf{y}', \mathsf{z}'\}$

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

$C(X) := \mathsf{false}$

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

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

$\quad Q_b(X) := \mathsf{QELIM}(\forall X'.\ R(X, X') \Rightarrow b(X) > b(X))$

$\quad C(X) := C(X) \vee \boxed{\mathsf{WLP}(R^*(X, X'), Q_b(X)}))$

done

return $C(X)$

Example

$$R = \mathsf{x}' = \mathsf{x} - \mathsf{z} \wedge \mathsf{x} \geq 1 \wedge \mathsf{y} \geq 1$$

$$\ldots, \mathsf{y}', \mathsf{z}'\}$$

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

$$\mathsf{WLP}(R^*, \mathsf{z} \leq -1) = \mathsf{z} \leq -1 \vee \mathsf{x} < 1 \vee \mathsf{y} < 1$$

$C(X) :=$ ...
$B(X)$ ...
foreach ...
$\quad Q_b(X) := \mathsf{QELIM}(\ldots R(X, X') \Rightarrow b(X) > b(X))$
$\quad C(X) := C(X) \vee \boxed{\mathsf{WLP}(R^*(X, X'), Q_b(X))}$
done
return $C(X)$

Other examples

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

Other examples

$$x \le 0 \vee x + y \le 0 \vee x + 2y + z \le 0 \vee x + 3y + 3z \le 0 \vee z < 0 \vee (z \le 0 \wedge y < 0)$$

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

Other examples

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

Other examples

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

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

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

Other examples

$$x > 5 \vee x < 0$$

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

Other examples

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

Other examples

$$x = y \vee (x > 0 \wedge y > 0)$$

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

Other examples

$y > 0$

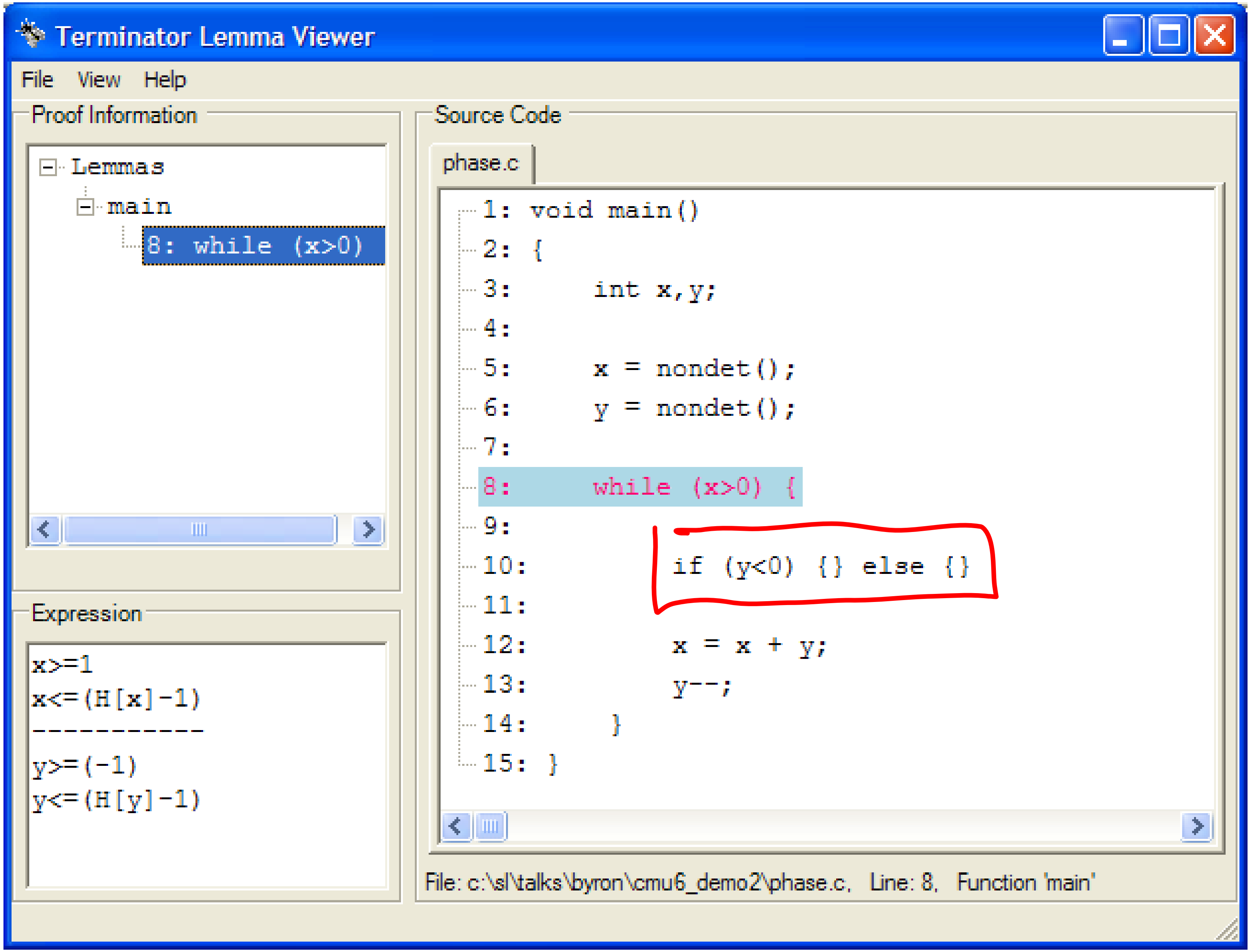
```
// @requires n>200 and y<9;
x = 0;
while (1) {
   if (x<n) {
      x=x+y;
      if (x>=200) break;
   }
}
```

Improving termination provers

- Synthesis technique can help improve power of the termination prover

- Key idea: Found precondition can be used as case split

Improving termination provers



while (x>0)

Source Code

phase.c

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

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

        while (x>0) {

                if (y<0) {} else {}

                x = x + y;
                y--;
        }
}
```

while (x>0)

Source Code

phase.c

```
void main()
{
    int
    
    x = non
    y = no ();
    
    while (x>0) {
    
        ~~if (y<0) {} else {}~~
        
        x = x + y;
        y--;
    }
}
```

$y \leq -1$

$$
\begin{array}{l}
\textbf{procedure } \textsc{PhasedPreSynth}(I, R) \\
\textbf{begin} \\
\quad C := \textsc{PreSynth}(I, R) \\
\quad \textbf{if } C \text{ is non-empty } \textbf{then} \\
\quad\quad C := C \cup \textsc{PhasedPreSynth}(I, R\downarrow_{\neg C}) \\
\quad \textbf{fi} \\
\quad \textbf{return } C \\
\textbf{end}
\end{array}
$$

Outline

- Recursive programs
- Weakest preconditions