Proofs That Count

Zachary Kincaid[1]
Azadeh Farzan[1] Andreas Podelski[2]

[1]University of Toronto
[2]University of Freiburg

January 22, 2014

Software verification

Goal

Given a program P and a specification $\varphi_{\mathbf{pre}}/\varphi_{\mathbf{post}}$, prove

$$\{\varphi_{\mathbf{pre}}\}P\{\varphi_{\mathbf{post}}\}$$

Software verification

Concurrent

Goal

Given a program P and a specification $\varphi_{\mathbf{pre}}/\varphi_{\mathbf{post}}$, prove

$$\{\varphi_{\mathbf{pre}}\}P\{\varphi_{\mathbf{post}}\}$$

Software verification

Concurrent

Unboundedly many threads

Goal

Given a program P and a specification $\varphi_{\mathbf{pre}}/\varphi_{\mathbf{post}}$, prove

$$\{\varphi_{\mathbf{pre}}\}P\{\varphi_{\mathbf{post}}\}$$

Software verification

Concurrent

Unboundedly many threads

Goal

Given a program P and a specification $\varphi_{\mathbf{pre}}/\varphi_{\mathbf{post}}$, prove

$$\{\varphi_{\mathbf{pre}}\}P\{\varphi_{\mathbf{post}}\}$$

- Proofs for concurrent programs sometimes make use of *counting* arguments.

Software verification

Concurrent

Unboundedly many threads

Goal

Given a program P and a specification $\varphi_{\mathbf{pre}}/\varphi_{\mathbf{post}}$, prove

$$\{\varphi_{\mathbf{pre}}\}P\{\varphi_{\mathbf{post}}\}$$

- Proofs for concurrent programs sometimes make use of *counting* arguments.
  - Readers/Writers protocol: “the number of active readers”

Software verification

Concurrent

Unboundedly many threads

Goal

Given a program P and a specification $\varphi_{\mathbf{pre}}/\varphi_{\mathbf{post}}$, prove

$$\{\varphi_{\mathbf{pre}}\}P\{\varphi_{\mathbf{post}}\}$$

- Proofs for concurrent programs sometimes make use of *counting* arguments.
  - Readers/Writers protocol: "the number of active readers"
  - Ticket protocol: "the number of processes with a smaller ticket"

What is a counting argument?

A counting argument is a proof that a program satisfies its specification which uses auxiliary *counters*:

- Can be used in assertions.
- *Auxiliary* (or *ghost*) variables: do not appear in the program.
  Think: Owicki-Gries.

Example

Precondition: $\{s = t = 0\}$

```
t++
assert(t > s)
s++
```

Example

Precondition: $\{s = t = 0\}$

```
1:  t++                 ||  1:  t++
2:  assert(t > s)       ||  2:  assert(t > s)
3:  s++                 ||  3:  s++
```

Example

Precondition: $\{s = t = 0\}$

```
1: t++                ||  1: t++
2: assert(t > s)      ||  2: assert(t > s)
3: s++                ||  3: s++
```

There is *no* Owicki-Gries proof that does not use auxiliary variables.

Example

Precondition: $\{s = t = 0\}$

`1: t++`
`2: assert(t > s)`
`3: s++`
	$\|\|$	`1: t++`
		`2: assert(t > s)`
		`3: s++`
			$\|\| \cdots \|\|$	`1: t++`
				`2: assert(t > s)`
				`3: s++`

Example

Precondition: $\{s = t = 0\}$

```
1:  t++                 ||  1:  t++                 ||  ...  ||  1:  t++
2:  assert(t > s)       ||  2:  assert(t > s)       ||       ||  2:  assert(t > s)
3:  s++                 ||  3:  s++                 ||       ||  3:  s++
```

Inductive invariant:

$\#2 + \#3 = t - s$

Example

Precondition: $\{s = t = 0\}$

```
1:  t++                 ||  1:  t++                 ||       ||  1:  t++
2:  assert(t > s)       ||  2:  assert(t > s)       ||  ···  ||  2:  assert(t > s)
3:  s++                 ||  3:  s++                 ||       ||  3:  s++
```

Inductive invariant:

$\#2 + \#3 = t - s$

# of threads at line 2

# of threads at line 3

Challenges

How do we formalize counting arguments?

Challenges

How do we formalize counting arguments?

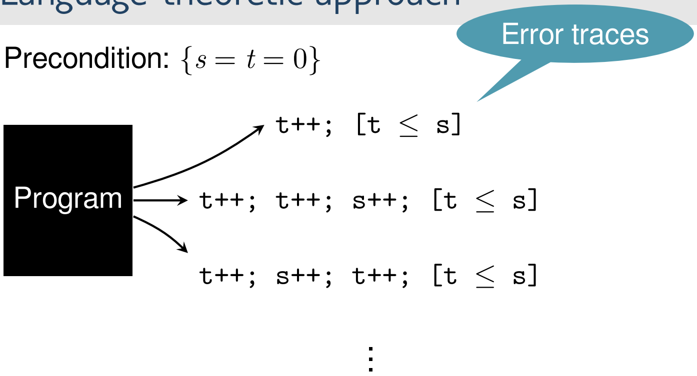
How do we synthesize counting arguments automatically?

Precondition: $\{s = t = 0\}$

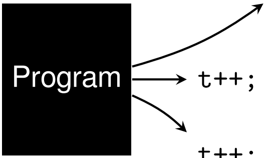
```
t++                  ||  t++                  ||       ||  t++
assert(t > s)        ||  assert(t > s)        ||  ...  ||  assert(t > s)
s++                  ||  s++                  ||       ||  s++
```

Language-theoretic approach

Precondition: $\{s = t = 0\}$



Precondition: $\{s = t = 0\}$




```
t++; [t ≤ s]

t++; t++; s++; [t ≤ s]

t++; s++; t++; [t ≤ s]
```

$\vdots$

$\forall \tau \in \mathcal{L}(\text{Proof}).\{\varphi_{\text{pre}}\}\tau\{\varphi_{\text{post}}\}$

Language-theoretic approach

Precondition: $\{s = t = 0\}$

Program

`t++; [t ≤ s]`

`t++; t++; s++; [t ≤ s]`

`t++; s++; t++; [t ≤ s]`

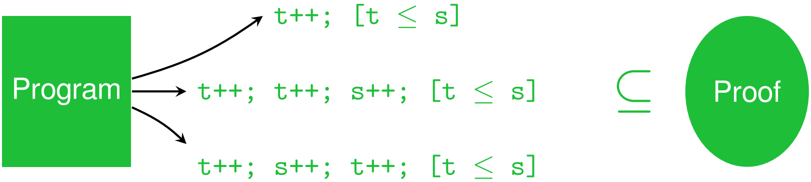
⋮

$\subseteq$ Proof

$\forall \tau \in \mathcal{L}(\text{Proof}).\{\varphi_{\text{pre}}\}\tau\{\varphi_{\text{post}}\}$

Language-theoretic approach

Precondition: $\{s = t = 0\}$



$\forall \tau \in \mathcal{L}(\text{Proof}).\{\varphi_{\text{pre}}\}\tau\{\varphi_{\text{post}}\}$

Proof rule

If there exists a *Proof* such that $\mathcal{L}(\textit{Program}) \subseteq \mathcal{L}(\textit{Proof})$, then $\{\varphi_{\mathbf{pre}}\}\textit{Program}\{\varphi_{\mathbf{post}}\}$

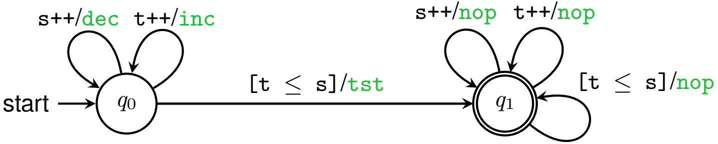
Counting proofs

Counting proof = counting automaton + inductive annotation

Counting proofs

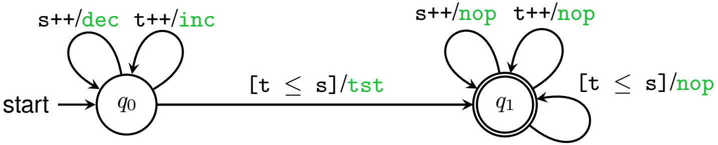
Counting proof = counting automaton + inductive annotation

- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}



Counting proof = counting automaton + inductive annotation

- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}

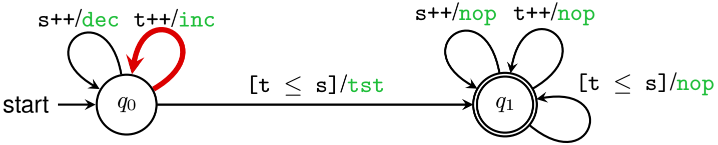


q_0
$k = 0$

Counting proofs

Counting proof = counting automaton + inductive annotation

- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}

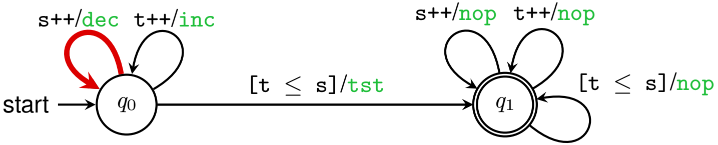


$$\begin{matrix} q_0 \\ k=0 \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \end{matrix}$$

Counting proofs

Counting proof = counting automaton + inductive annotation

- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}

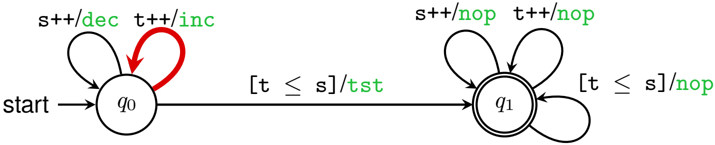


$$\begin{matrix} q_0 \\ k=0 \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \end{matrix} \xrightarrow[\texttt{dec}]{\texttt{s++}} \begin{matrix} q_0 \\ k=0 \end{matrix}$$

Counting proofs

Counting proof = counting automaton + inductive annotation

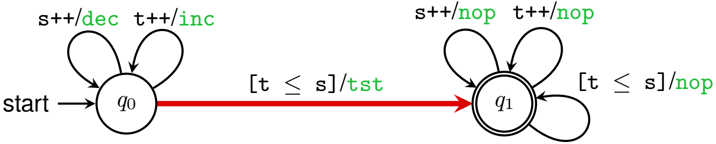
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in \{$inc, dec, tst, nop$\}$



$$\begin{matrix} q_0 \\ k=0 \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \end{matrix} \xrightarrow[\texttt{dec}]{\texttt{s++}} \begin{matrix} q_0 \\ k=0 \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \end{matrix}$$

Counting proof = counting automaton + inductive annotation

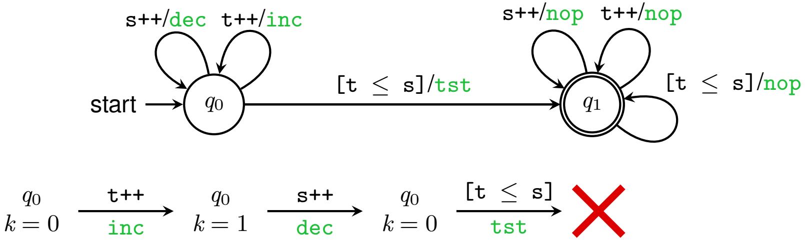
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}



$$\begin{matrix} q_0 \\ k=0 \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \end{matrix} \xrightarrow[\texttt{dec}]{\texttt{s++}} \begin{matrix} q_0 \\ k=0 \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \end{matrix} \xrightarrow[\texttt{tst}]{\texttt{[t} \leq \texttt{s]}} \begin{matrix} q_1 \\ k=1 \end{matrix}$$

Counting proof = counting automaton + inductive annotation

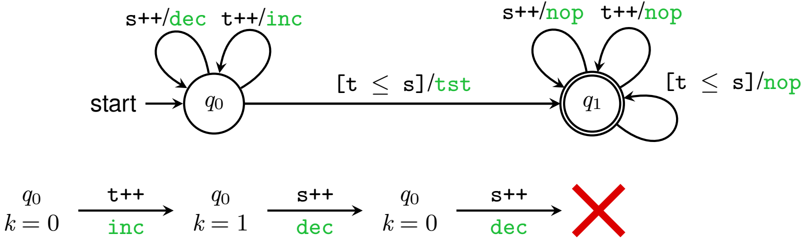
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}



Counting proofs

Counting proof = counting automaton + inductive annotation

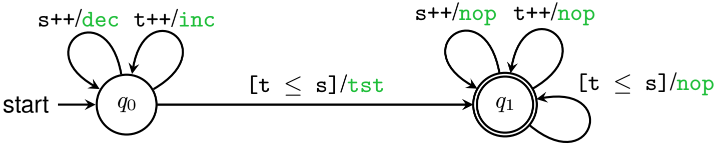
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}



Counting proofs

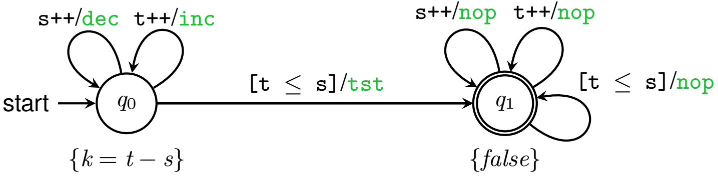
Counting proof = counting automaton + inductive annotation

- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}
- *Inductive annotation* = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)



Counting proof = counting automaton + inductive annotation

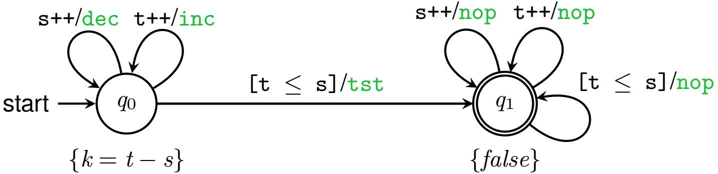
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables. *Assume one counter variable for this talk.* Transitions are labeled by a counter action $\in \{\text{inc}, \text{dec}, \text{tst}, \text{nop}\}$
- *Inductive annotation* = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)



Counting proofs

Counting proof = counting automaton + inductive annotation

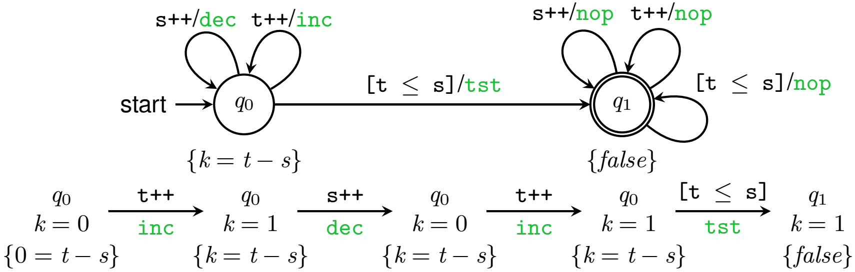
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}
- *Inductive annotation* = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)



$$\begin{matrix} q_0 \\ k=0 \\ \{k=t-s\} \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \\ \{k=t-s\} \end{matrix} \xrightarrow[\texttt{dec}]{\texttt{s++}} \begin{matrix} q_0 \\ k=0 \\ \{k=t-s\} \end{matrix} \xrightarrow[\texttt{inc}]{\texttt{t++}} \begin{matrix} q_0 \\ k=1 \\ \{k=t-s\} \end{matrix} \xrightarrow[\texttt{tst}]{[\texttt{t} \le \texttt{s}]} \begin{matrix} q_1 \\ k=1 \\ \{false\} \end{matrix}$$

Counting proof = counting automaton + inductive annotation

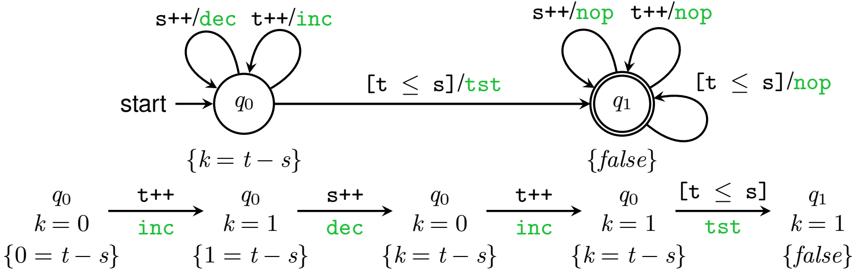
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}
- *Inductive annotation* = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)



Counting proofs

Counting proof = counting automaton + inductive annotation

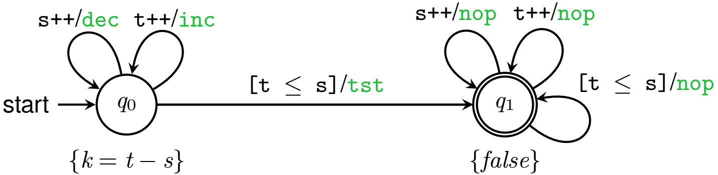
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in \{\text{inc}, \text{dec}, \text{tst}, \text{nop}\}$
- *Inductive annotation* = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)



Counting proofs

Counting proof = counting automaton + inductive annotation

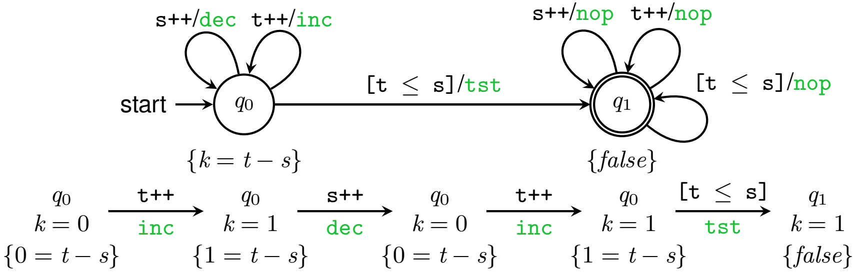
- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}
- *Inductive annotation* = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)



$$\begin{array}{ccccccccc} q_0 & \xrightarrow[\texttt{inc}]{\texttt{t++}} & q_0 & \xrightarrow[\texttt{dec}]{\texttt{s++}} & q_0 & \xrightarrow[\texttt{inc}]{\texttt{t++}} & q_0 & \xrightarrow[\texttt{tst}]{[\texttt{t} \leq \texttt{s}]} & q_1 \\ k=0 & & k=1 & & k=0 & & k=1 & & k=1 \\ \{0 = t - s\} & & \{1 = t - s\} & & \{0 = t - s\} & & \{k = t - s\} & & \{false\} \end{array}$$

Counting proof = counting automaton + inductive annotation

- *Counting automaton* = DFA with additional $\mathbb{N}$-valued counter variables.
  Assume one counter variable for this talk.
  Transitions are labeled by a counter action $\in$ {inc, dec, tst, nop}
- *Inductive annotation* = assignment of assertions to counting automaton states (think: Floyd/Hoare annotation)

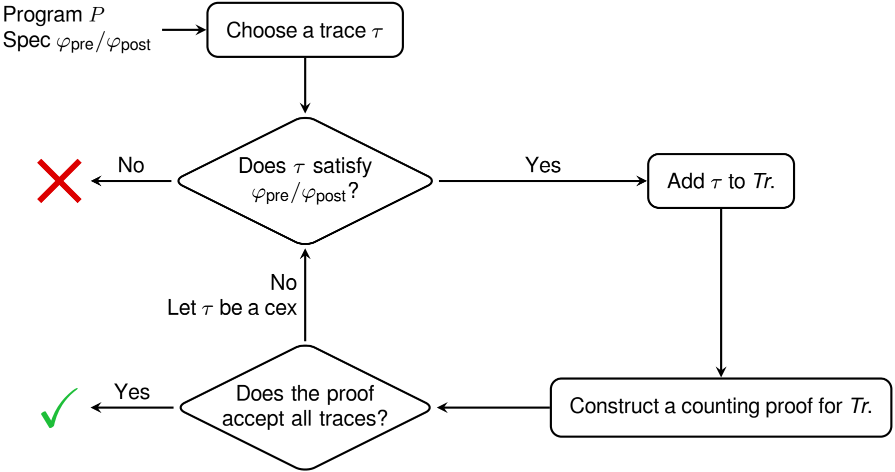


Challenges

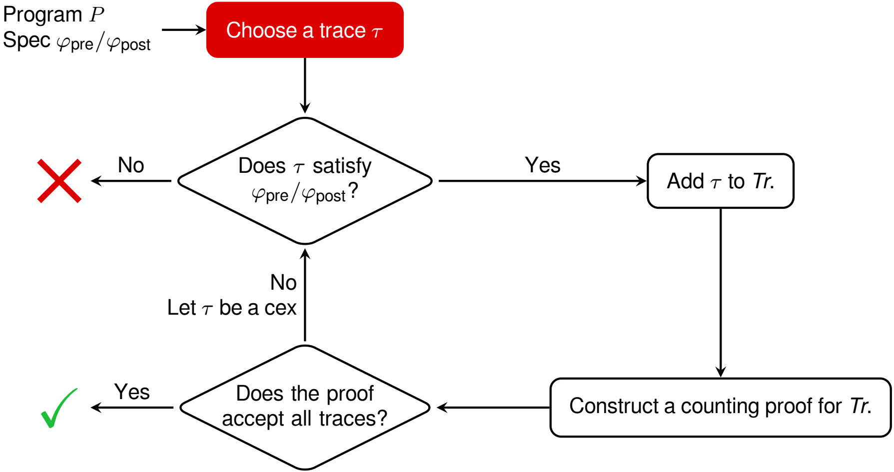
How do we formalize counting arguments?

How do we synthesize counting arguments automatically?

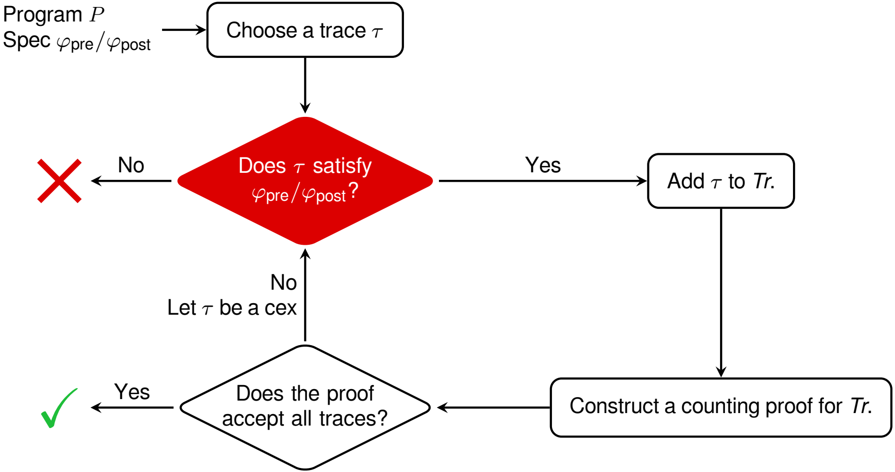
“Learning” a counting argument



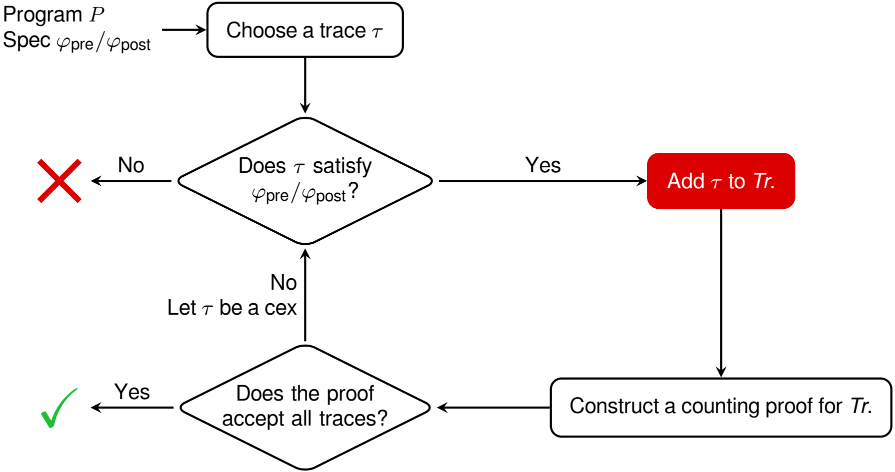
“Learning” a counting argument



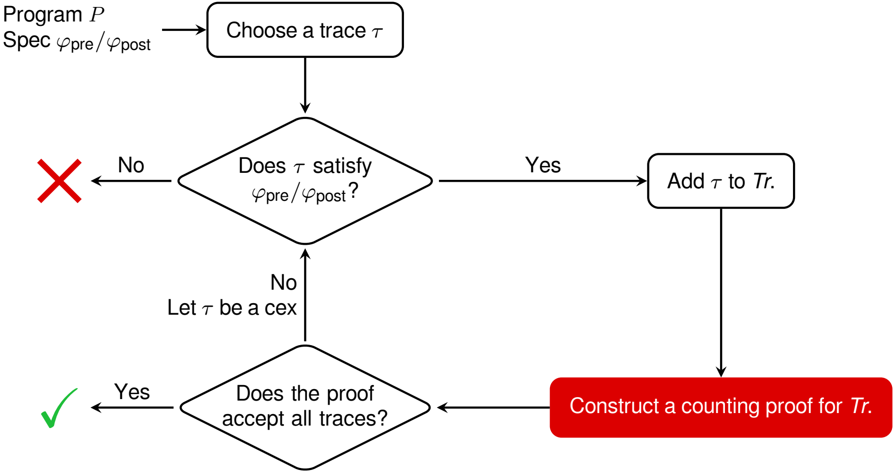
“Learning” a counting argument



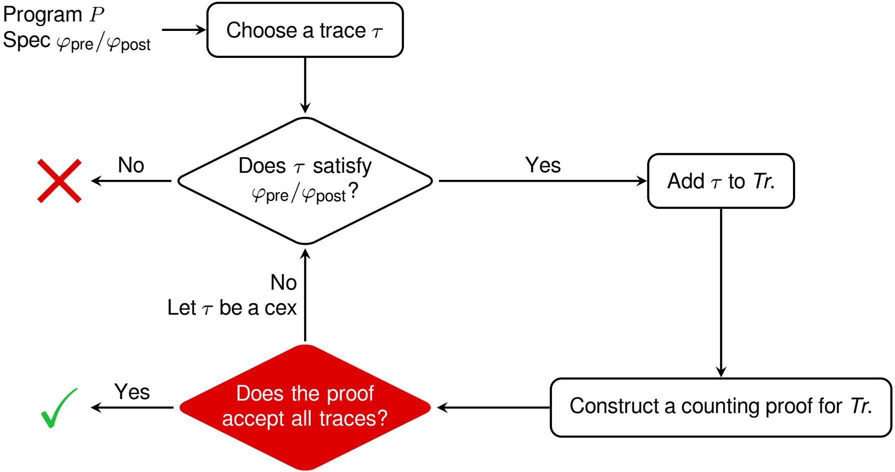
“Learning” a counting argument



"Learning" a counting argument



"Learning" a counting argument



Constructing a counting proof

Goal

Given a finite set of traces *Tr* and a spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$, construct a counting proof $\langle A, \varphi\rangle$ such that *Tr* $\subseteq \mathcal{L}(A)$.

Constructing a counting proof requires us to find a counting automaton and an inductive annotation *simultaneously*.

- Insight #1: Bounded synthesis is decidable
  - Bound the size of the counting proof (think: # of states)
  - Encode bounded proof synthesis as a formula in a decidable theory (QF_UFNRA)
  - Use uninterpreted function symbols to encode the transition relation.
  - Use Farkas' lemma to generate constraints searching for an inductive annotation (*á la* Colón et al.[a])

[a] Linear Invariant Generation using Non-linear Constraint Solving, CAV'03

Constructing a counting proof

Goal

Given a finite set of traces *Tr* and a spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$, construct a counting proof $\langle A, \varphi \rangle$ such that $\mathit{Tr} \subseteq \mathcal{L}(A)$.

Constructing a counting proof requires us to find a counting automaton and an inductive annotation *simultaneously*.

- Insight #2: Occam's Razor – search for a "small" proof. More likely to generalize & use counters!

$$\tau = \texttt{t++}; \texttt{s++}; \texttt{t++}; \texttt{[t} \leq \texttt{s]}$$

Constructing a counting proof

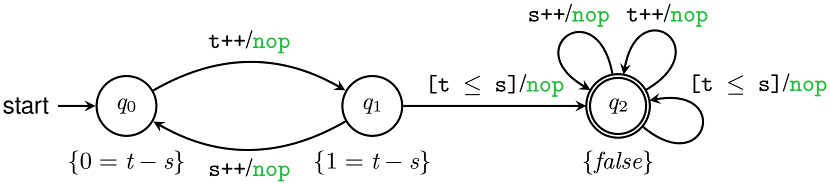
Goal

Given a finite set of traces *Tr* and a spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$, construct a counting proof $\langle A, \varphi \rangle$ such that $Tr \subseteq \mathcal{L}(A)$.

Constructing a counting proof requires us to find a counting automaton and an inductive annotation *simultaneously*.

- Insight #2: Occam's Razor – search for a "small" proof. More likely to generalize & use counters!

$$\tau = \texttt{t++}; \texttt{s++}; \texttt{t++}; [\texttt{t} \leq \texttt{s}]$$



Constructing a counting proof

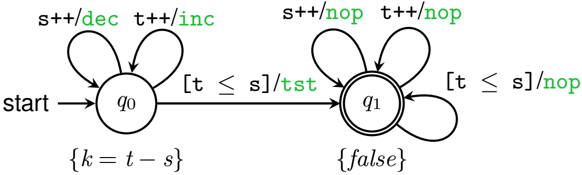
Goal

Given a finite set of traces *Tr* and a spec $\varphi_{\text{pre}}/\varphi_{\text{post}}$, construct a counting proof $\langle A, \varphi \rangle$ such that $Tr \subseteq \mathcal{L}(A)$.

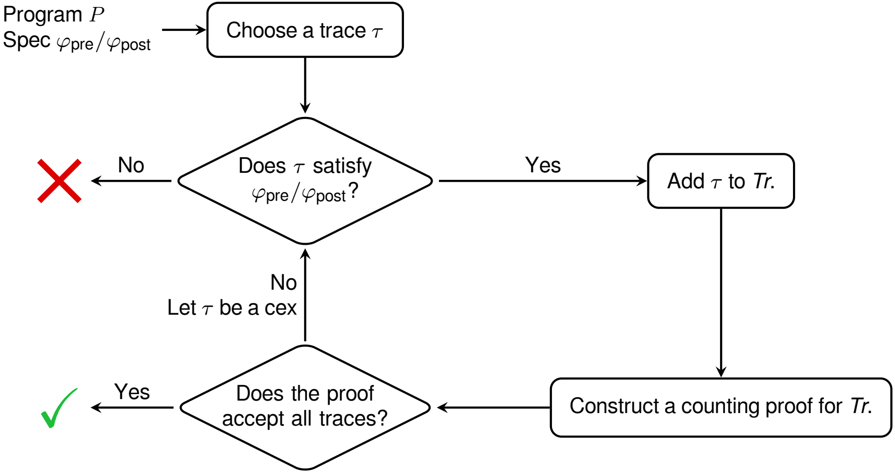
Constructing a counting proof requires us to find a counting automaton and an inductive annotation *simultaneously*.

- Insight #2: Occam's Razor – search for a "small" proof. More likely to generalize & use counters!

$$\tau = \texttt{t++}; \texttt{s++}; \texttt{t++}; [\texttt{t} \leq \texttt{s}]$$



“Learning” a counting argument



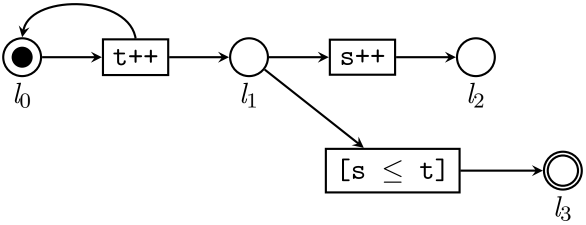
Control flow nets

Control flow net $=$ Petri net $+$ program commands

Control flow nets

Control flow net $=$ Petri net $+$ program commands

```
1:  t++                 ||  1:  t++                 ||       ||  1:  t++
2:  assert(t > s)       ||  2:  assert(t > s)       ||  ···  ||  2:  assert(t > s)
3:  s++                 ||  3:  s++                 ||       ||  3:  s++
```



Represents the set of error traces for the program.

Proof checking

Theorem

Let P be a control flow net, and let A be a counting automaton. The problem of determining whether $\mathcal{L}(P) \subseteq \mathcal{L}(A)$ is decidable.

Proof checking

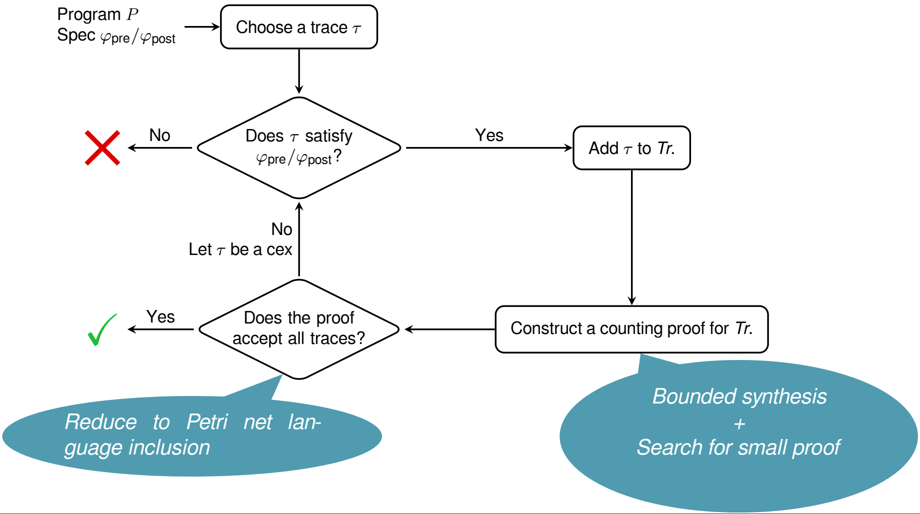
Theorem

Let P be a control flow net, and let A be a counting automaton. The problem of determining whether $\mathcal{L}(P) \subseteq \mathcal{L}(A)$ is decidable.

- Reduction to Petri net language inclusion.

Summary

We can automate synthesis of a class of auxiliary variables!



What's next?

- Implementation & Evaluation
  - Practical algorithm for inclusion?
    - Ultimately, inclusion relies on a reduction to Petri net reachability.
  - Practical nonlinear constraint solving?
- Synthesize other classes of auxiliary variables?