The next generation of formal verification tools

KeY Symposium 2026

Gidon Ernst
Interim Professor Software Engineering, University of Augsburg
Junior Professor Software Verification, LMU Munich
https://www.sosy-lab.org/people/ernst/

August 6, 2026

Foundations of
the next generation of formal verification tools

KeY Symposium 2026

Gidon Ernst
Interim Professor Software Engineering, University of Augsburg
Junior Professor Software Verification, LMU Munich
https://www.sosy-lab.org/people/ernst/

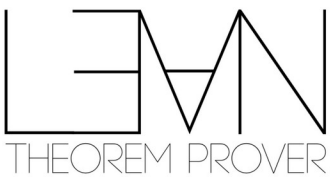
August 6, 2026

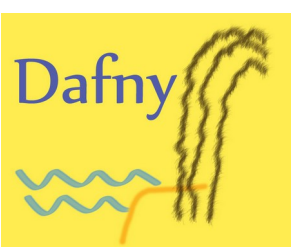
The current generation of tools

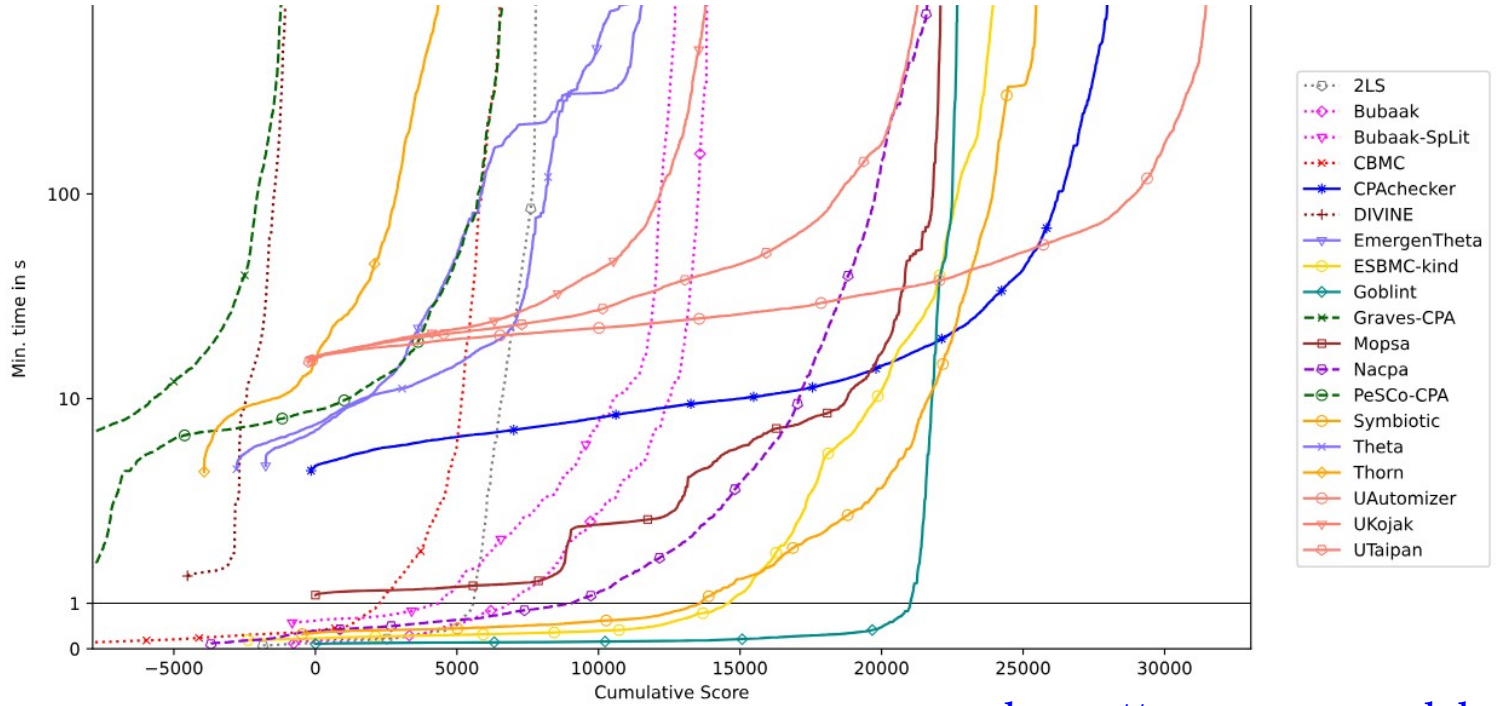
- they solve real and substantial problems

The current generation of tools

- they solve real and substantial problems







https://sv-comp.sosy-lab.org/

fully automatic
simpler properties

human-guided
expressive specs

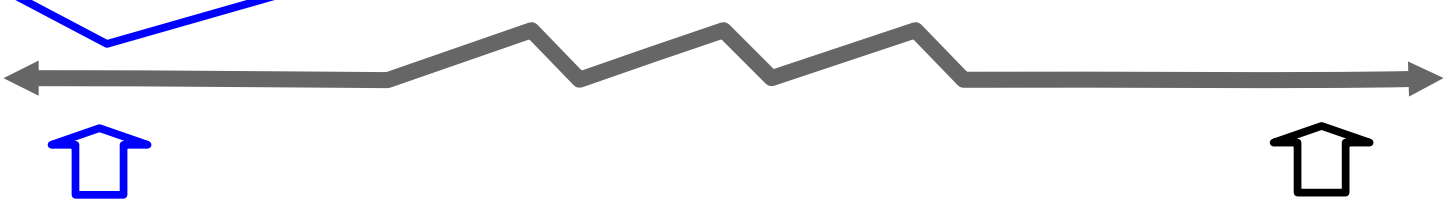
A billion SMT queries a day

By Neha Rungta

2022

Download | Copy BibTeX | Share

Amazon Web Services (AWS) is a cloud computing services provider that has made significant investments in applying formal methods to proving correctness of its internal systems and providing assurance of correctness to their end-users. In this paper, we focus on how we built abstractions and eliminated specifications to scale a verification engine for AWS access policies, Zelkova, to be usable by all AWS users. We present



fully automatic
simpler properties

human-guided
expressive specs

Fehler in Standardsortieralgorithmus mit formalen Methoden aufgedeckt

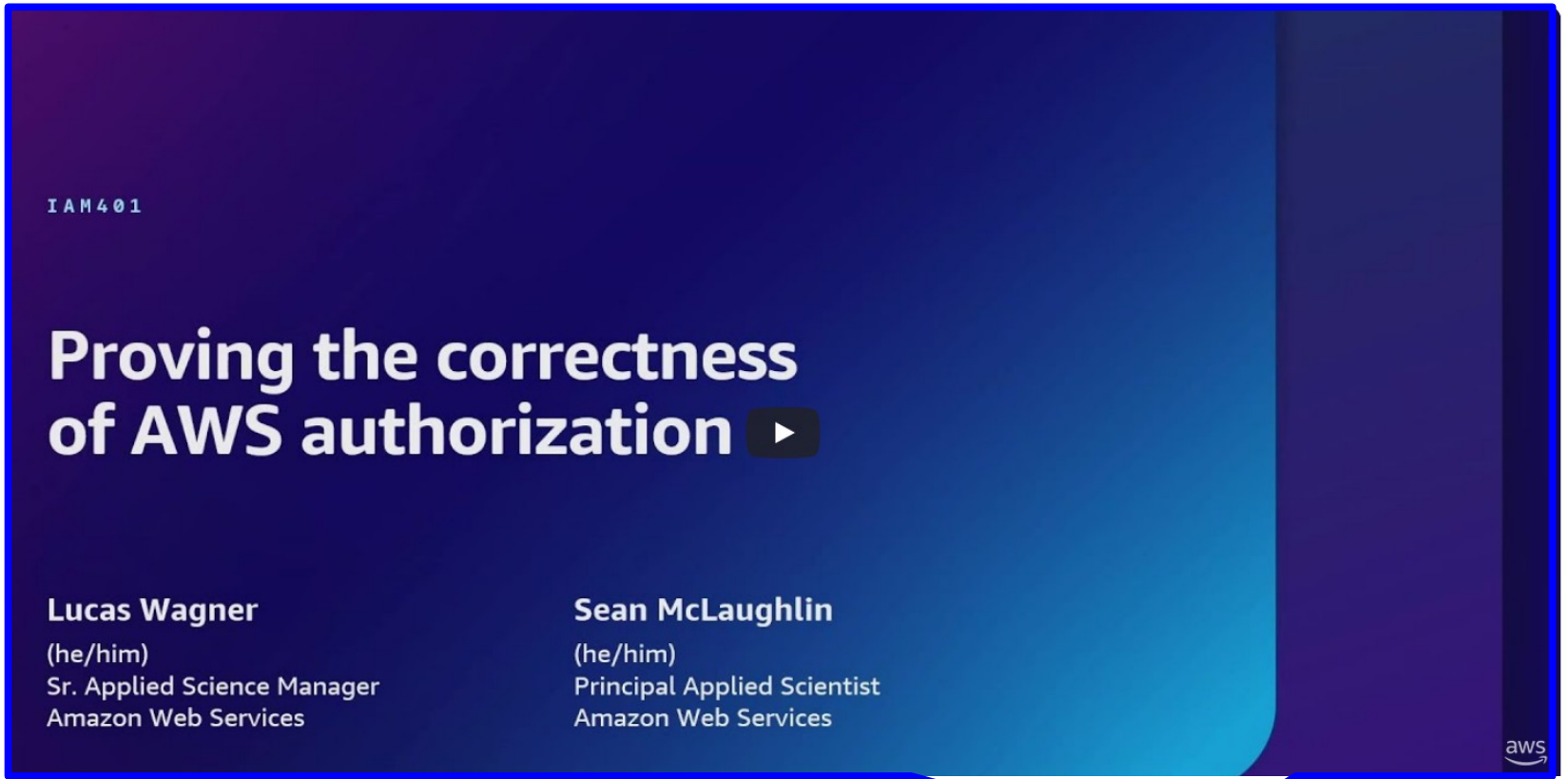
Android, Java und Groovy nutzen alle den TimSort-Algorithmus. Informatiker eines Verbundprojekts konnten mit Hilfe eines von ihnen entwickelten Tools nun einen Fehler in der Implementierung feststellen und beheben.

115

https://heise.de/-2570944

fully automatic
simpler properties

human-guided
expressive specs



fully automatic
simpler properties

human-guided
expressive specs

The current generation of tools

- they solve real and substantial problems

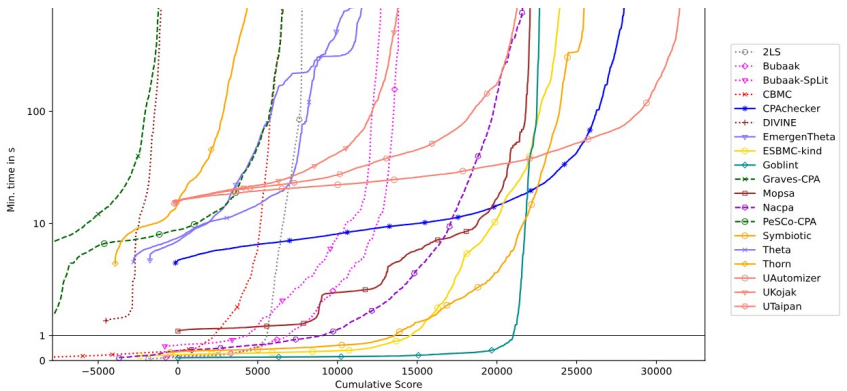
- great! thanks for your attention. the end.

The current generation of tools

- they solve real and substantial problems

- great! we have come a long way but…

  verification is still difficult



fully automatic
simpler properties

open challenge

human-guided
expressive specs

Closing the verification loop: Observability-driven harnesses for building with agents

AI HARNESS ENGINEERING

PUBLISHED Mar 9, 2026

READ TIME 12m


Alp Keles


Jai Menon


Sesh Nalla

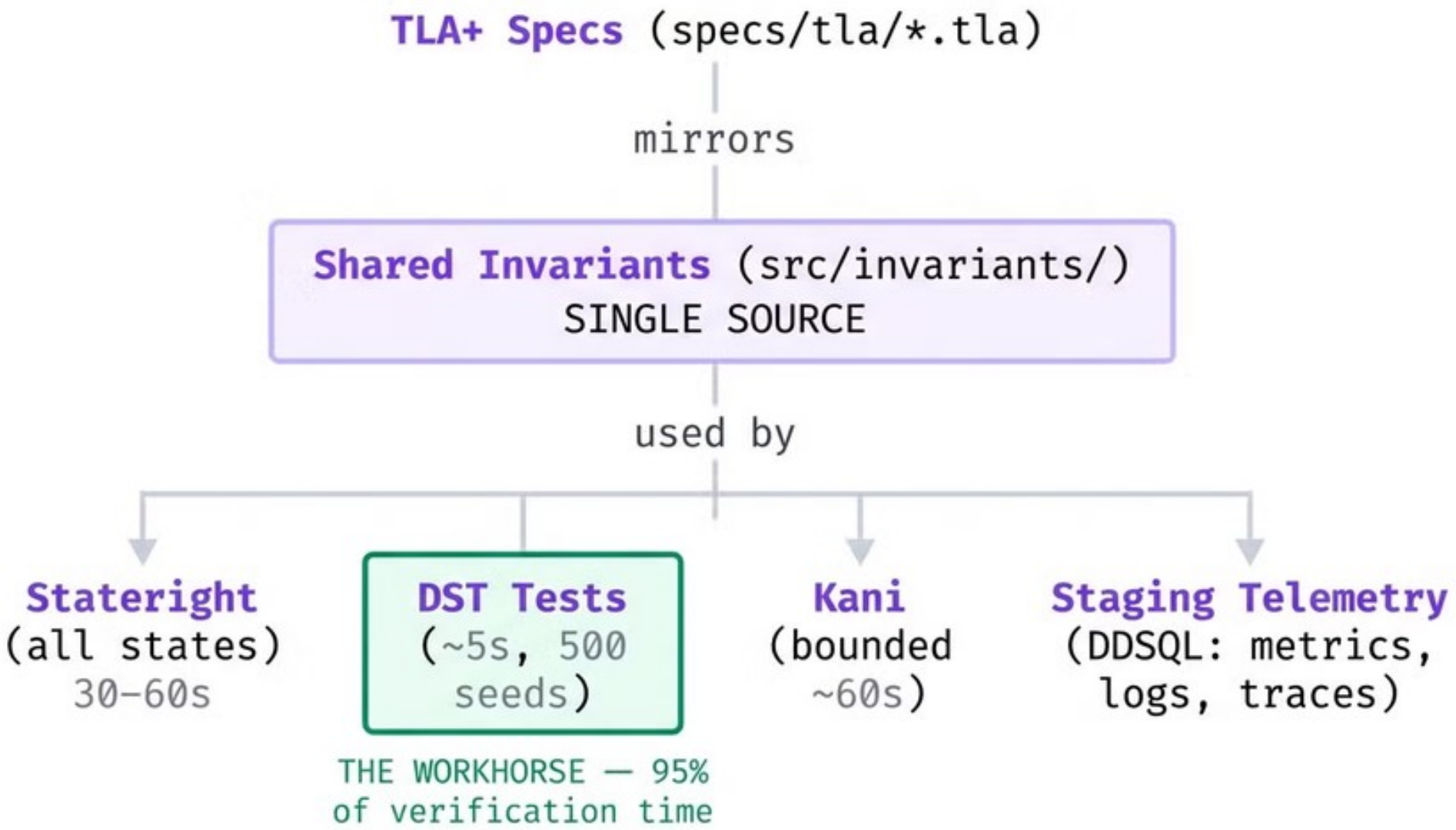

Vyom Shah

AI agents can now produce software faster than any team can verify it. The bottleneck has moved from writing code to trusting what was written.

https://www.datadoghq.com/blog/ai/harness-first-agents/

Why bother at all?

"the bottleneck has moved from writing code to **trusting what was written**"


TLA+ Specs (specs/tla/*.tla)
mirrors
Shared Invariants (src/invariants/)
SINGLE SOURCE
used by
Stateright
(all states)
30–60s
DST Tests
(~5s, 500
seeds)
THE WORKHORSE — 95%
of verification time
Kani
(bounded
~60s)
Staging Telemetry
(DDSQL: metrics,
logs, traces)

The current generation of tools

- they solve real and substantial problems

- great! we have come a long way but...

  verification is still difficult but worthwhile

- what are we doing and are we doing it right?

Claims

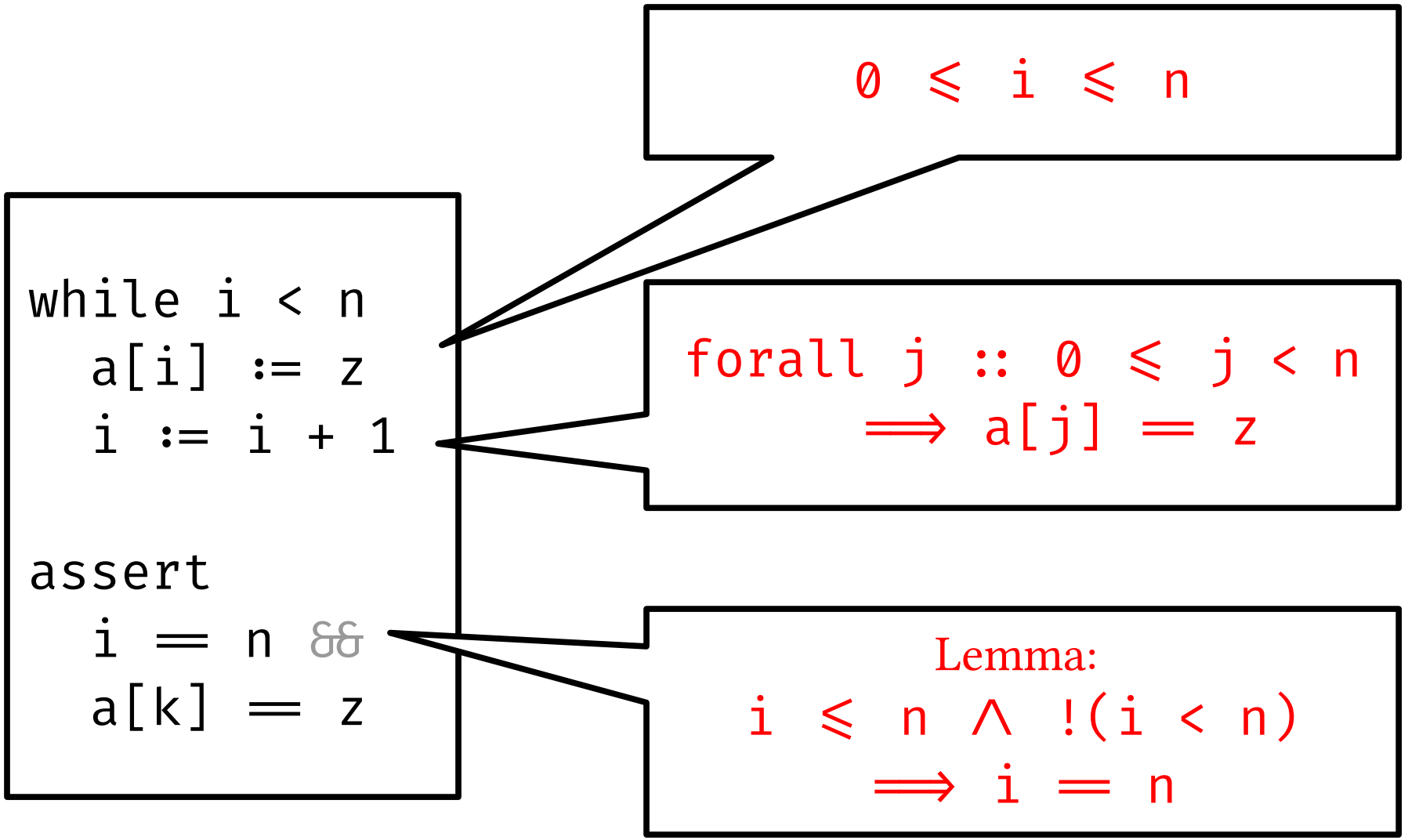
- Invariants are a bad representation of correctness arguments
- Lemmas and auxiliary definitions are often a severe bottleneck
- Algorithmic inference is a bad method for routine tasks
- Incremental reasoning is essential for complex tasks

(much of this is widely accepted)

Claims

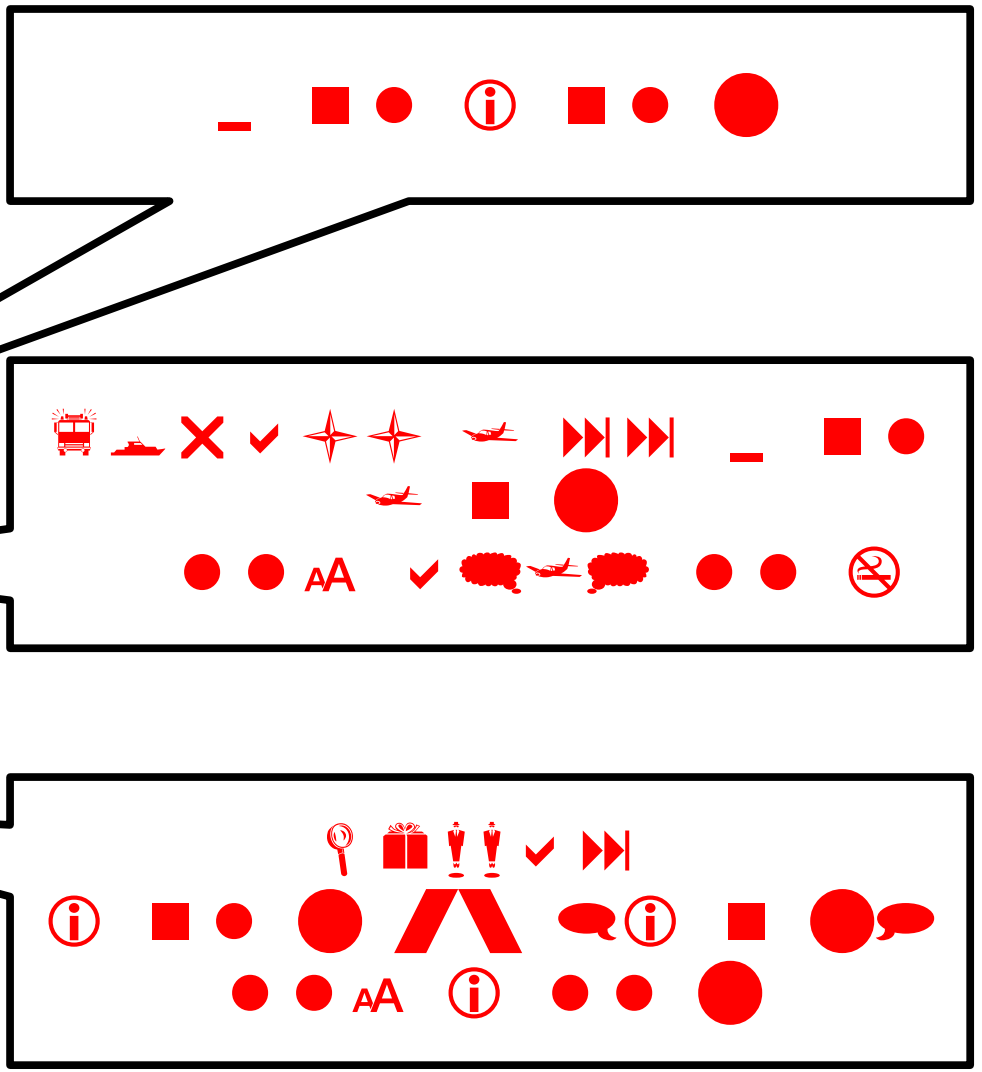
- Invariants are a bad representation of correctness arguments
- Lemmas and auxiliary definitions are often a severe bottleneck
- Algorithmic inference is a bad method for routine tasks
- Incremental reasoning is essential for complex tasks

Floyd/Hoare logic and invariants

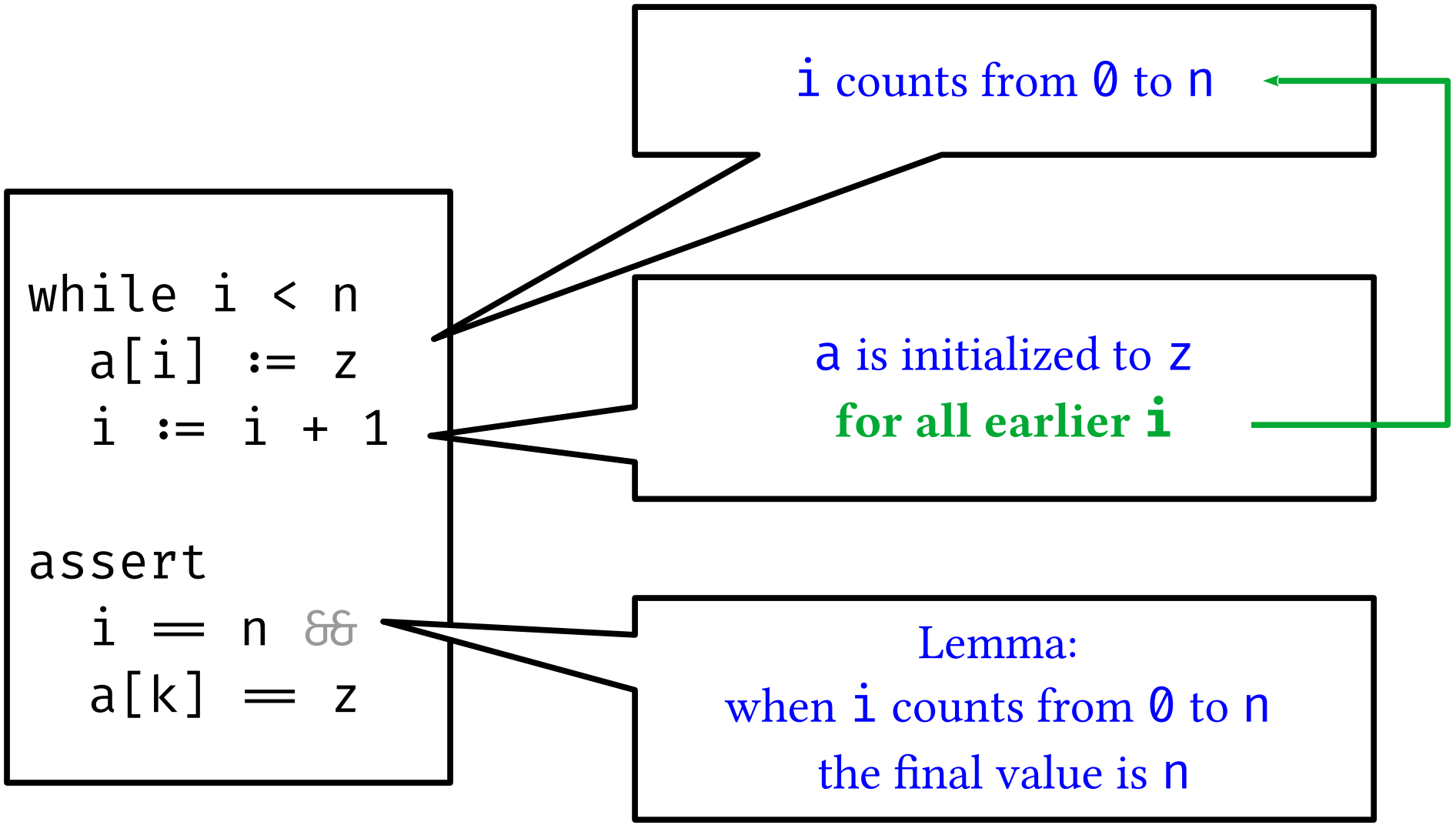


Floyd/Hoare logic and invariants

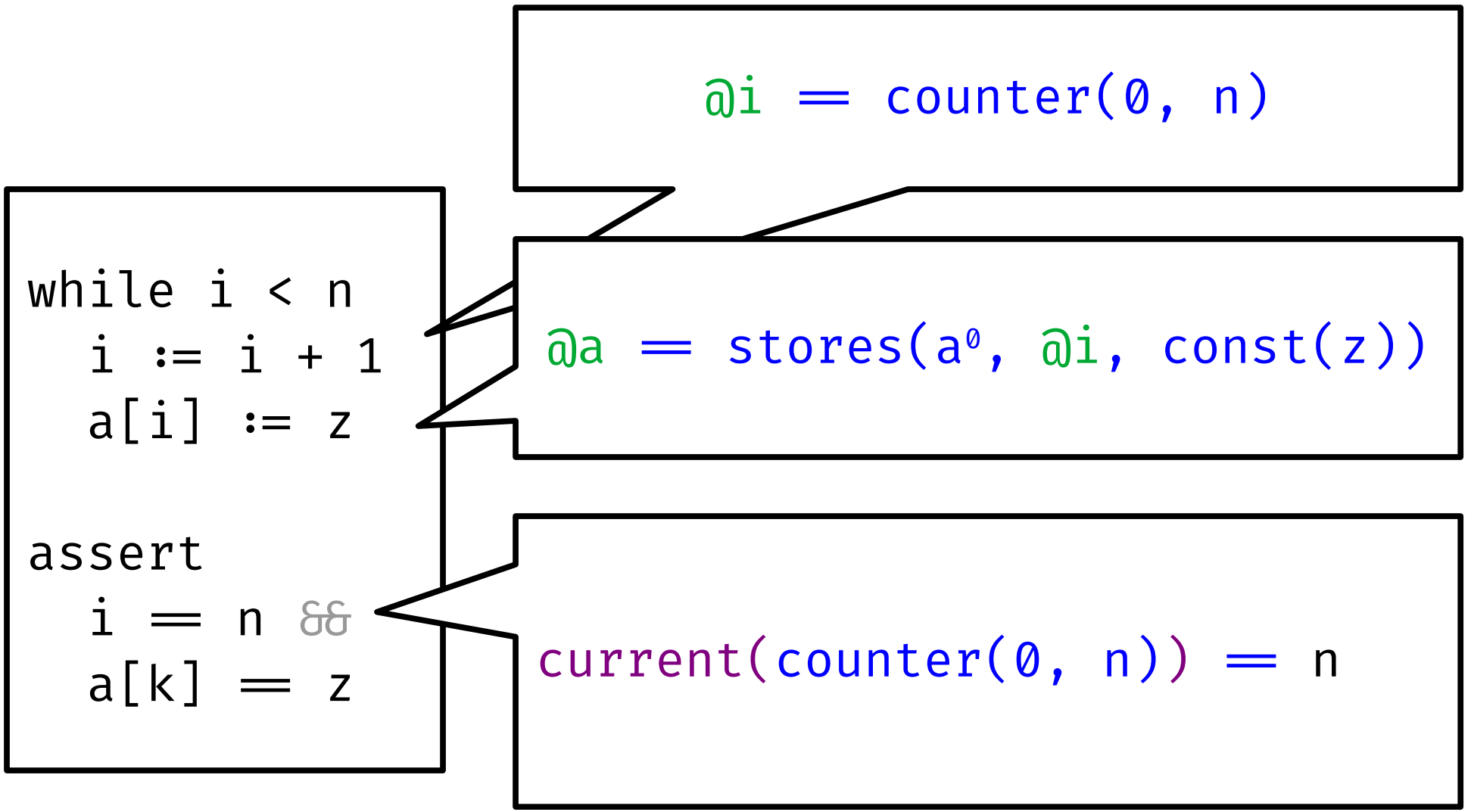
```
while i < n
  a[i] := z
  i := i + 1

assert
  i == n &&
  a[k] == z
```



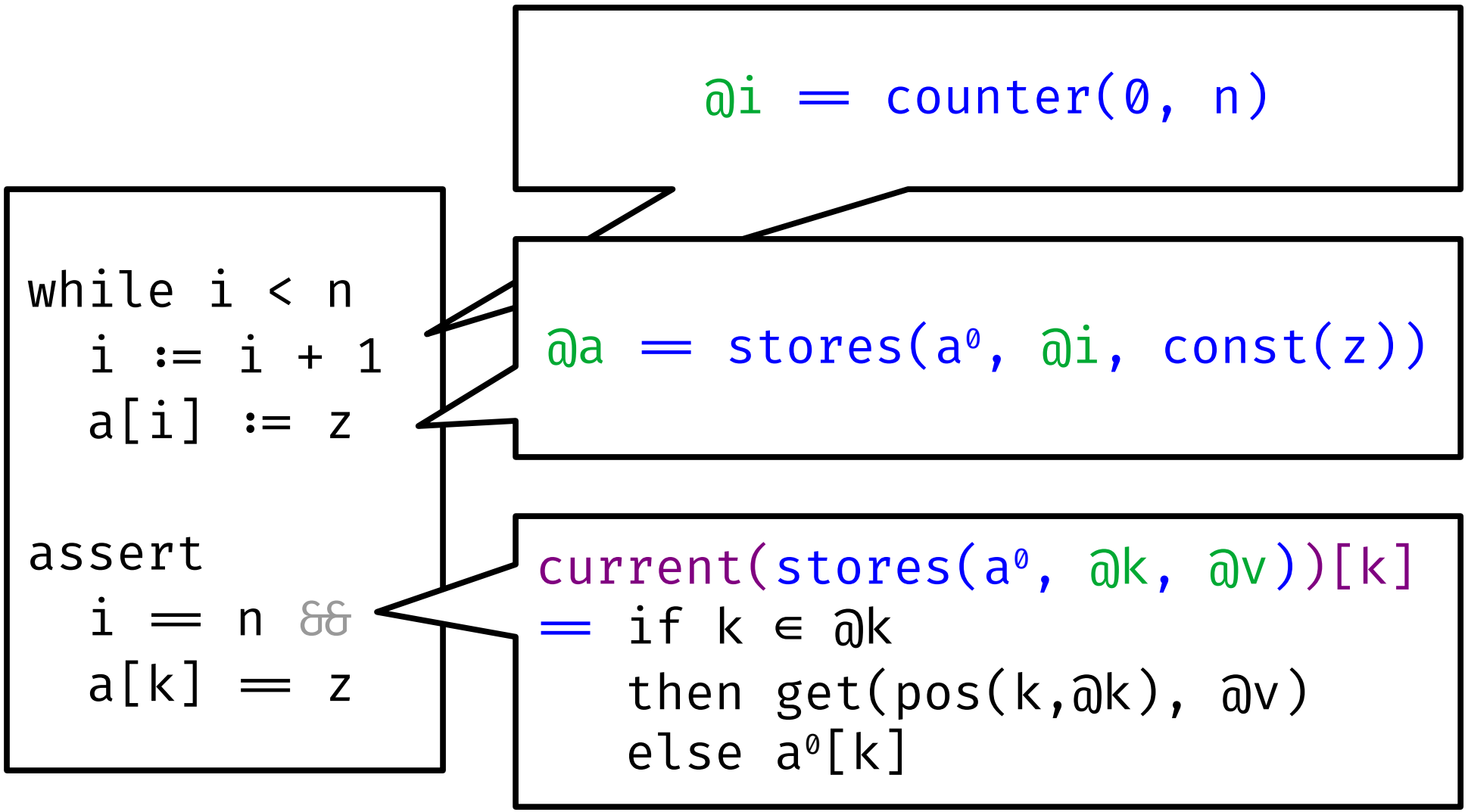
A much better explanation



Trace Equation Systems (TEQs)



Trace Equation Systems (TEQs)



Claims

- Invariants are a bad representation of correctness arguments
- Lemmas and auxiliary definitions are often a severe bottleneck

  [Baumann & Beckert 12]
- Algorithmic inference is a bad method for routine tasks
- Incremental reasoning is essential for complex tasks

“Annotation inference is the problem”

$$\frac{\exists\, I. \qquad P \Rightarrow I \qquad \{I \wedge \phi\}\ c\ \{I\} \qquad I \wedge \neg\phi \Rightarrow Q}{\{P\}\ \textbf{while}\ \phi\ \textbf{do}\ c\ \{Q\}}$$

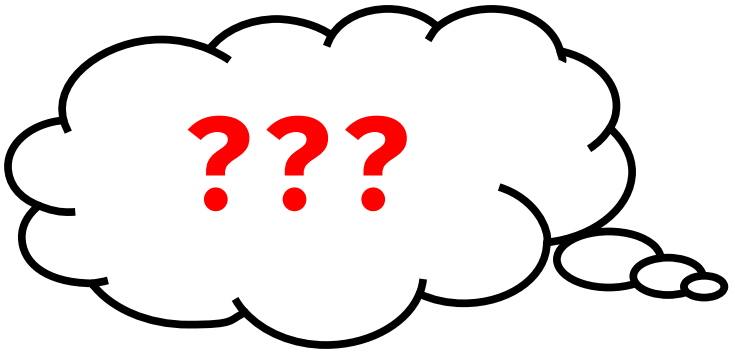
"Annotation inference is the problem"

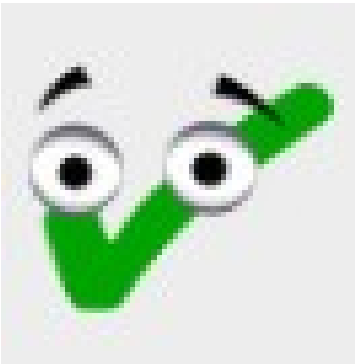
has trivial solution:
(**assume** ϕ; c)*
(think: ACL2, rippling)

$$\exists\, I. \qquad P \Rightarrow I \qquad \{I \wedge \phi\}\ c\ \{I\} \qquad I \wedge \neg\phi \Rightarrow Q$$
$$\{P\}\ \mathbf{while}\ \phi\ \mathbf{do}\ c\ \{Q\}$$

Current boundary of automation

```
for i = 0..n
  s += a[i]

for i = 0..n
  s -= a[i]

assert
  s == 0
```





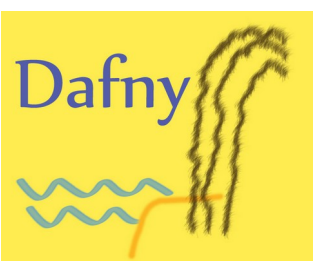
Current deductive approach

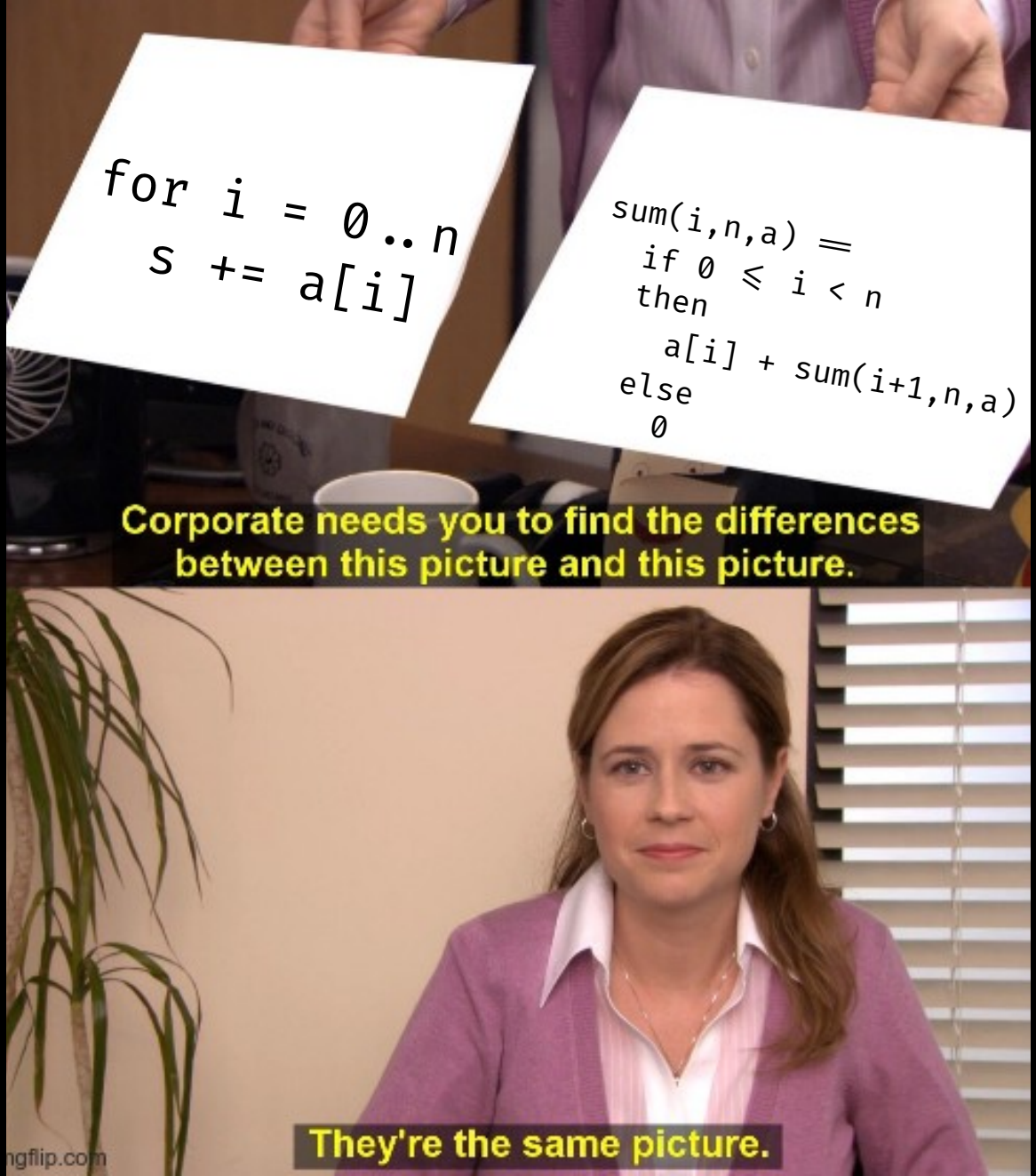
program

```
for i = 0..n
  s += a[i]

for i = 0..n
  s -= a[i]

assert
  s == 0
```

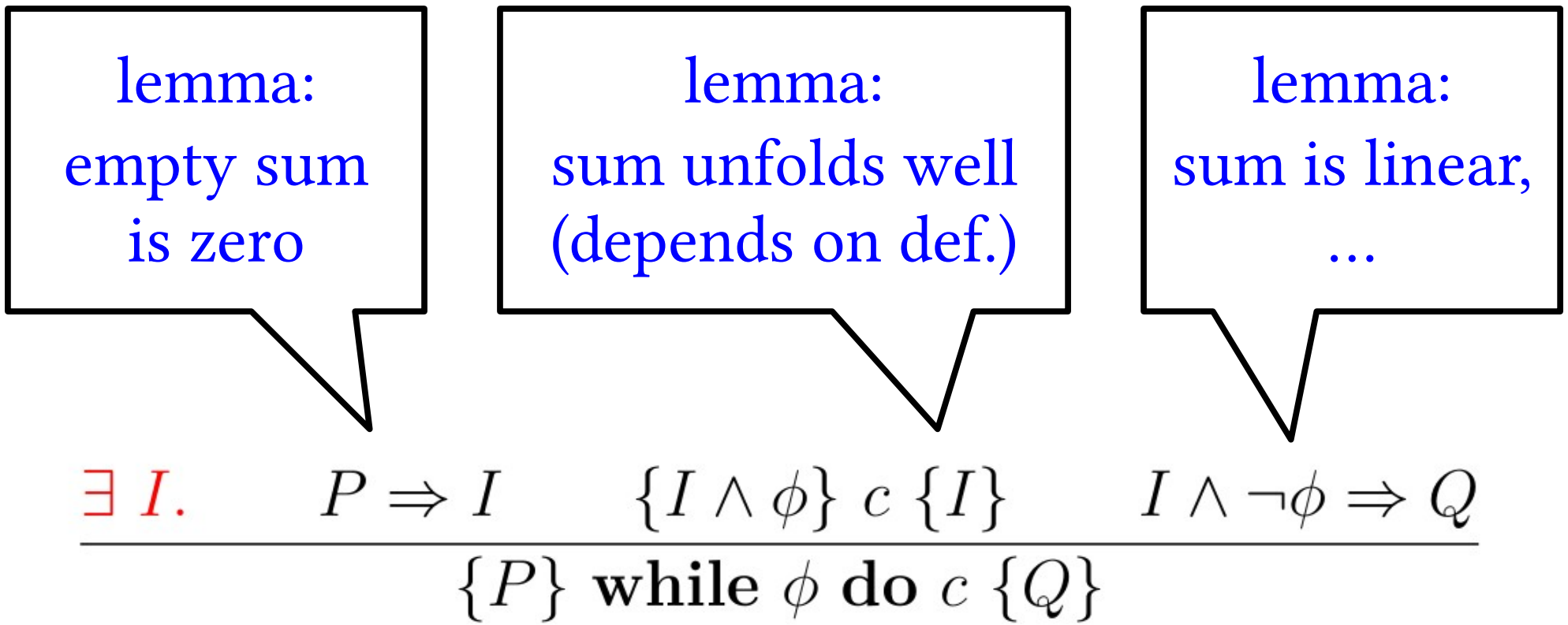
logic

```
fun sum(i,n,a) {
  if 0 ⩽ i < n
  then a[i]+sum(i+1,n,a)
  else 0
}
```




for i = 0..n
s += a[i]
sum(i,n,a) =
if 0 ≤ i < n
then
a[i] + sum(i+1,n,a)
else
0
Corporate needs you to find the differences between this picture and this picture.
They're the same picture.

Use of lemmas for code with auxiliary functions



Claims

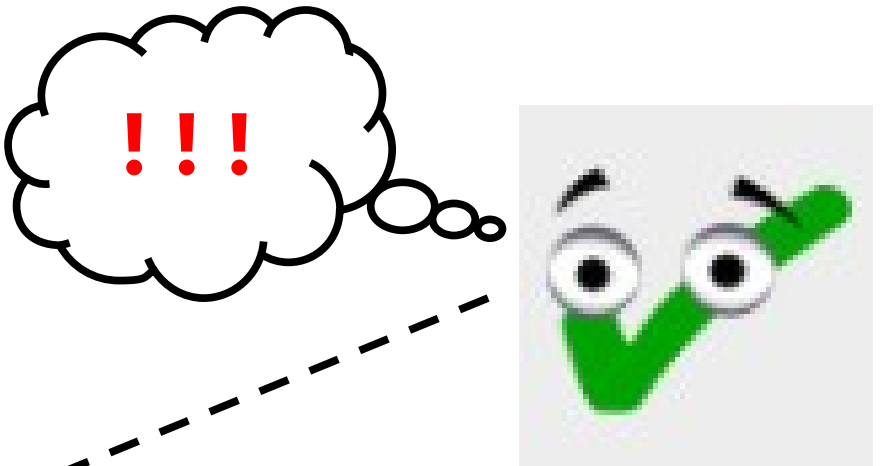
- Invariants are a bad representation of correctness arguments
- Lemmas and auxiliary definitions are often a severe bottleneck
- Algorithmic inference is a bad method for routine tasks
- Incremental reasoning is essential for complex tasks

“Model checking is dead”

(famous person at CAV, reportedly)

Algorithmic invariant inference

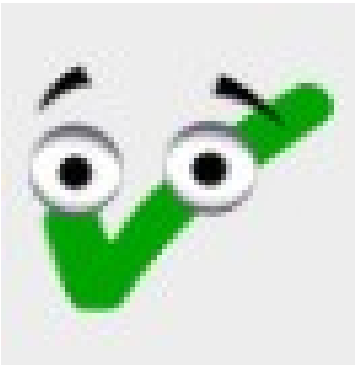
```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```



Algorithmic invariant inference

```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```

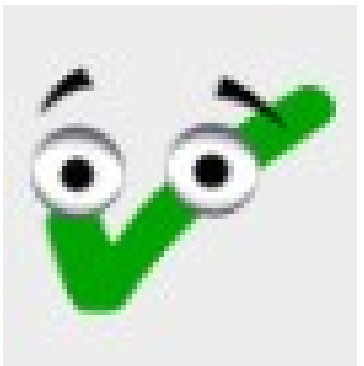
$\text{true} \wedge \neg(i < n) \implies i = n \; ?$



Algorithmic invariant inference

```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```

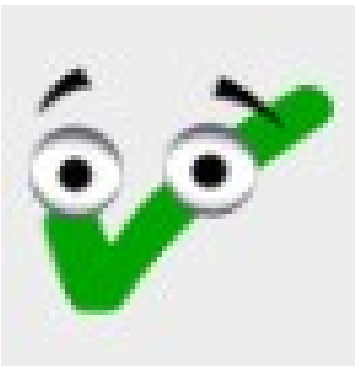
```
true ∧ !(i < n)
⟹ i == n ?

no. let me see if
the CEX is
reachable ...
```



Algorithmic invariant inference

```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```

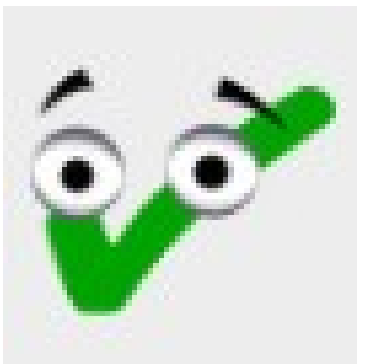
```
true ∧ !(i < n)
⟹ i = n ?

no.
CEX not real.
```



Algorithmic invariant inference

```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```

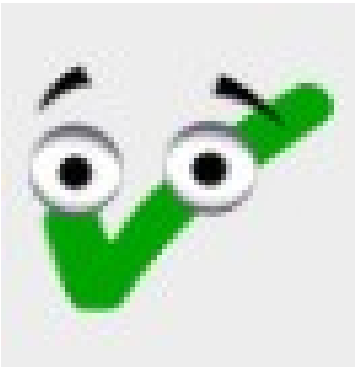
```
true ∧ !(i < n)
⟹ i = n ?

no.
CEX not real.
let me
interpolate that
for you!
```



Algorithmic invariant inference

```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```

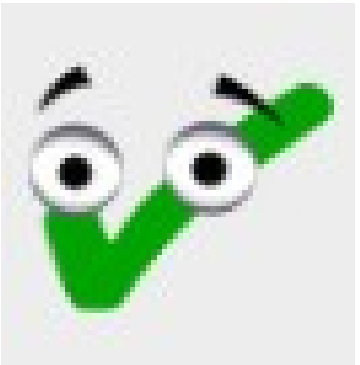
```
true ∧ !(i < n)
⟹ i = n ?

no.
CEX not real.
interpolant:
  i ⩽ n
```



Algorithmic invariant inference

```
i = 0

while i < n
  i += 1
  ...

assert
  i == n
```

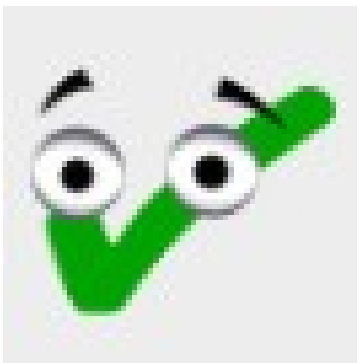
$i \leqslant n \wedge$
$\quad !(i < n)$
$\Longrightarrow i = n$?



Algorithmic invariant inference

```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```

```
i ⩽ n ∧
  !(i < n)
⟹ i = n ?

yes.

you are right!
it's a counter
```



Invariants from known principles

```
i = 0

while i < n
   i += 1
   ...

assert
   i == n
```

loop acceleration:
`i` counts from `0` to `n`

Claims

- Invariants are a bad representation of correctness arguments
- Lemmas and auxiliary definitions are often a severe bottleneck
- Algorithmic inference is a bad method for routine tasks
- Incremental reasoning is essential for complex tasks

“Interaction of is coming back”

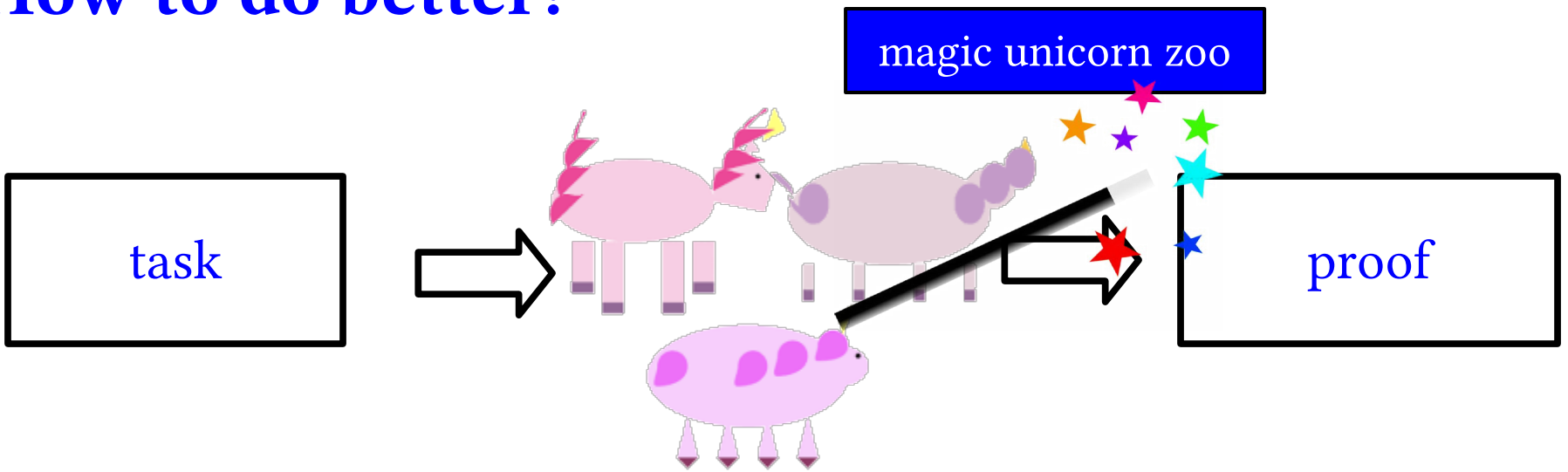
(Bernhard Steffen, yesterday)

Access to (useful) intermediate results?

```
for i = 0..n
  s += a[i]

for i = 0..n
  s -= a[i]

assert
  s == 0
```

```
first loop:
0 ⩽ i ⩽ n

second loop:
0 ⩽ i ⩽ n

let me unroll that
…
a[0] - a[0] == 0
a[0] + a[1]
  - a[0] - a[1] == 0
…
(timeout)
```

How to do better?



How to do better?

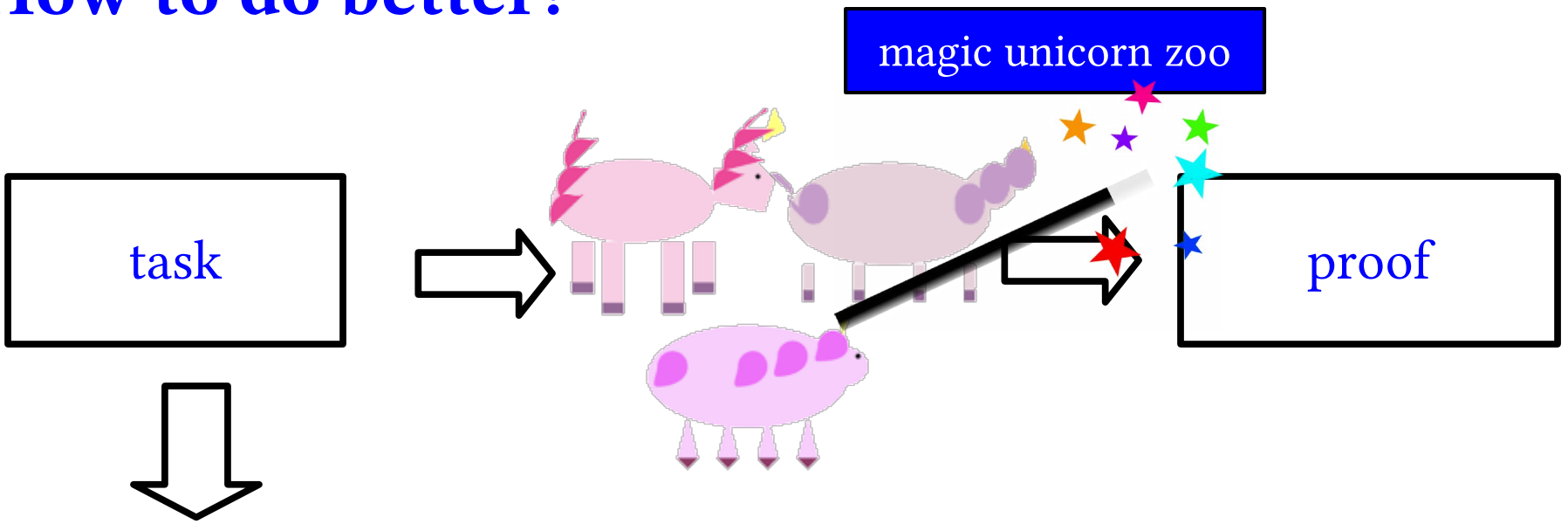
Why are LLMs good?

How to do better?

Why are LLMs good?

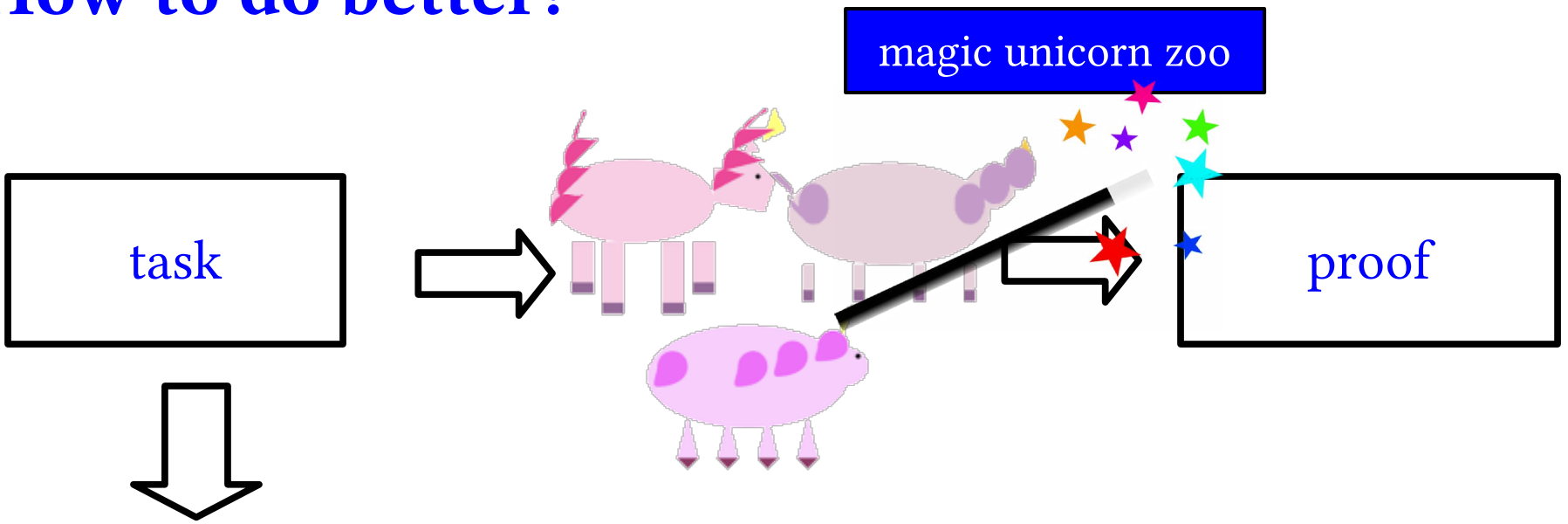
1. have effective problem representation
2. know the solution

How to do better?

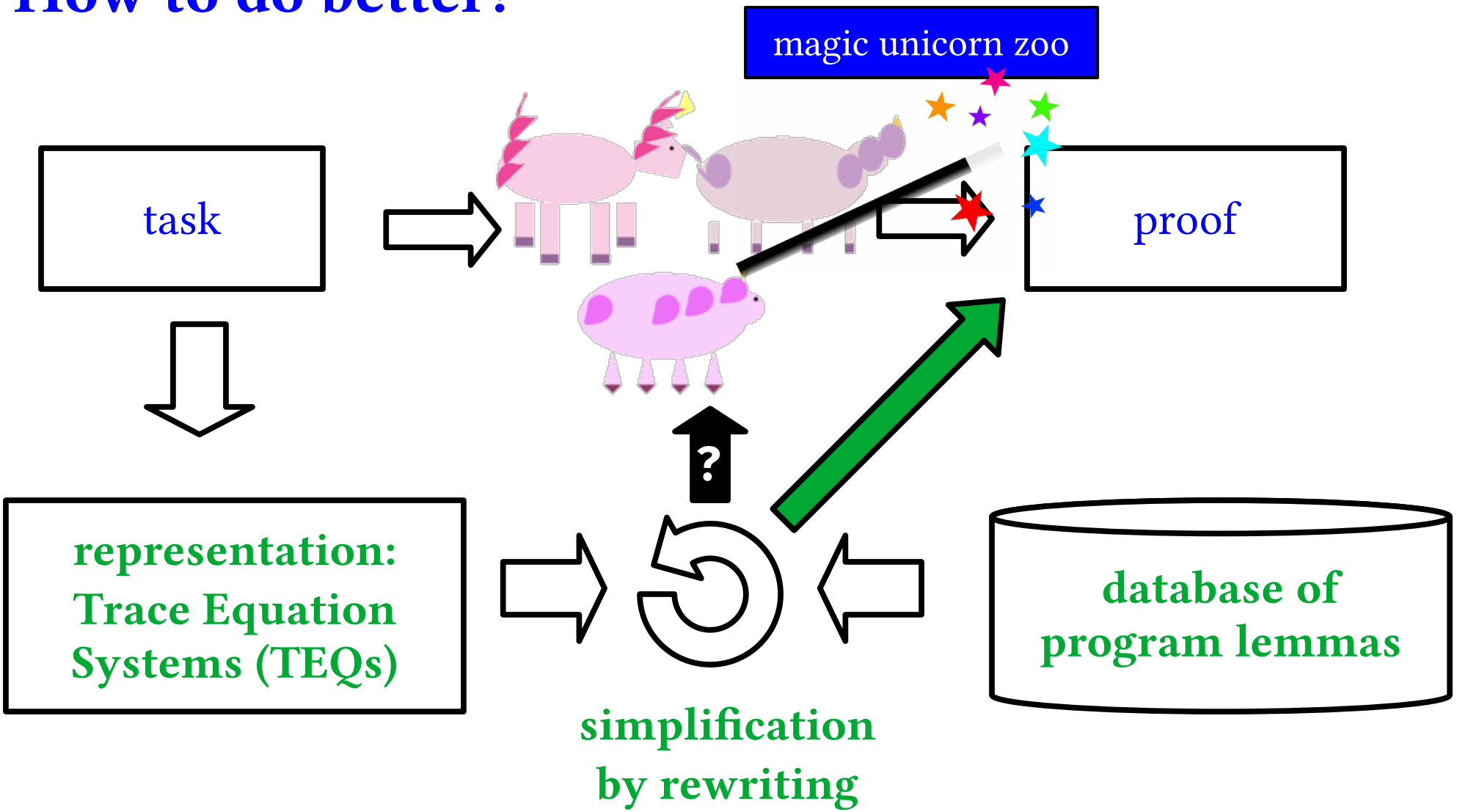


representation: Trace Equation Systems (TEQs)

How to do better?



How to do better?



TEQs are a “variable-modular” representation

```
@xs == trace(init, for y ← ys do x := u(x,y))
```

dependencies

update rule

```
x == current(@xs)
```

program variable

```
current(trace(init, ys, upd))
  == foldl(init, ys, upd)
```

TEQs enable algebraic reasoning

```
@is == trace(init, for k ← ks do i := i + 1)
    ⇝ range(0, 1, |ks|)

  i == current(@is)
    ⇝ |ks|
```

```
current(range(a, c, n))
        ==[lemma]  a + c*n
```

DEMO
array init, summation

Trace Equation Systems: Evaluation

Benchmark	# Tasks	TEQ ✓	TEQ err	TEQ+Z3 ✓	TEQ+Z3 err	Reference	
MaxPranq	17	16	0	16	0	MaxPranq †	17
Rapid	33	14	0	15	0	Rapid	30
Σ MonoCera	150	52	10	62	23	MonoCera	63
min	17	7	0	10	0	9	
max	12	5	3	5	3	8	
sum	26	5	3	6	7	16	
forall	96	35	7	41	13	30	
Arrays (safe)	320	68	32	86	‡ 87	Symbiotic ∗	68
Loops (safe)	551	44	146	71	158	aise ∗	405

Trace Equation Systems for Program Verification

[Coenen, Ernst, submitted 2026]

Contribution: New foundations

Trace Equation Systems for Program Verification

[Coenen, Ernst, submitted 2026]

Contribution: New foundations

TEQs represent constraints from loops

- view a the loop as its own specification – but enable tractable automation
- syntactic, sound, complete → retain expressiveness
- isolate program fragments → compositional reasoning

Trace Equation Systems for Program Verification

[Coenen, Ernst, submitted 2026]

Contribution: New foundations

TEQs represent constraints from loops

- view a the loop as its own specification – but enable tractable automation
- syntactic, sound, complete → retain expressiveness
- isolate program fragments → compositional reasoning

TEQs support incremental verification

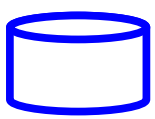
- reason by program transformation – with an algebraic rewrite theory
- lemmas to give names to program fragments (counter, array copy, …)
- lemmas to explain program fragments some contexts (comparison, lookup, …)

Trace Equation Systems: Current Status

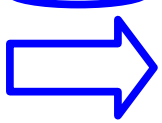
- Theory (including Isabelle/HOL formalization)
- Tool (most capabilities, still prototype)

Trace Equation Systems: Current Status

- Theory (including Isabelle/HOL formalization)
- Tool (most capabilities, still prototype)
- Lemma database and reasoning principles



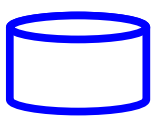
counters, array lookup + update, summations

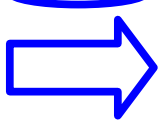


next steps: learn how to deal with more complex code

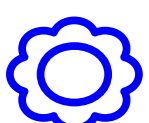
Trace Equation Systems: Current Status

- Theory (including Isabelle/HOL formalization)
- Tool (most capabilities, still prototype)
- Lemma database and reasoning principles

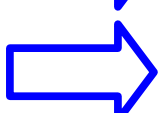

counters, array lookup + update, summations


next steps: learn how to deal with more complex code

- Evaluation


so far: promising results without overly specific lemmas


next steps: cover entire benchmark set, extract principles


later steps: real-world code

Some related work (in parts motivating/subsumed)

- CHCs: canonical “state-modular” representatio
- TEQs: canonical “variable-modular” representation
- Bundy+ [93], Unno [17] goal-directed induction
- Daca+ [CAV 16] Array Folds Logic
- Frohn [TACAS 20] modular acceleration calculus
- Georgiou+ [FMCAD 20] Trace Logic
- Chakraborty+ [TACAS 20, CAV 21] transformations
- Amilon+ [CAV 23] instrumentation + CHCs
- Frohn [CAV 26] acceleration of array lookups

Outlook and Discussion

- What's the scope & limitations of TEQs?
  Notably: how to treat recursive procedures?

- How to combine with traditional methods?

- How to build interaction around TEQs?

- How to infer lemmas automatically?
  LemmaCalc [Ernst, Fedyukovich iFM 2025]

On the abstraction level of interactive proofs

But AI…

https://mistral.ai/news/leanstral/

AvlRotation · LEAN

Copy

```
/-- Left rotation, applied at a node whose right child is itself a node.
    If the right child is a leaf there is nothing to rotate, and we
    return the tree unchanged (the standard convention). -/
def rotateLeft : Tree α → Tree α
  | node l x (node rl y rr) => node (node l x rl) y rr
  | t => t

/-! ### 1. Rotation preserves the multiset (here: the exact sequence) of elements -/

theorem rotateLeft_toList (l : Tree α) (x : α) (rl : Tree α) (y : α) (rr : Tree α) :
    toList (rotateLeft (node l x (node rl y rr)))
      = toList (node l x (node rl y rr)) := by
  simp only [rotateLeft, toList, List.append_assoc]
```

https://claude.ai/chat/688827fb-f1ec-41b9-9ab1-c4aa7fa8c5cf

AvlRotation · LEAN

Copy

```
/-- Left rotation, applied at a node whose right child is itself a node.
    If the right child is a leaf there is nothing to rotate, and we
    return the tree unchanged (the standard convention). -/
def rotateLeft : Tree α → Tree α
  | node l x (node rl y rr) => node (node l x rl) y rr
  | t => t

/-! ### 1. Rotation preserves the multiset (here: the exact sequence) of elements -/

theorem rotateLeft_toList (l : Tree α) (x : α) (rl : Tree α) (y : α) (rr : Tree α) :
    toList (rotateLeft (node l x (node rl y rr)))
      = toList (node l x (node rl y rr)) := by
  simp only [rotateLeft, toList, List.append_assoc]
```

not principled approach

https://claude.ai/chat/688827fb-f1ec-41b9-9ab1-c4aa7fa8c5cf

AvlRotation · LEAN

Copy

```
/-! ### 2. Rotation preserves the BST invariant -/

theorem rotateLeft_BST [LT α] (lt_trans : ∀ {a b c : α}, a < b → b < c → a < c)
    (l : Tree α) (x : α) (rl : Tree α) (y : α) (rr : Tree α)
    (h : BST (α := α) (node l x (node rl y rr))) :
    BST (α := α) (rotateLeft (node l x (node rl y rr))) := by
  obtain ⟨hlx, hxRight, hBSTl, hBSTright⟩ := h
  obtain ⟨hrly, hyrr, hBSTrl, hBSTrr⟩ := hBSTright
  have hxy : x < y := hxRight y (by simp [toList])
  have hxrl : ∀ z ∈ toList rl, x < z := by
    intro z hz
    exact hxRight z (by simp [toList]; exact Or.inl hz)
  have hBSTlxrl : BST (α := α) (node l x rl) := ⟨hlx, hxrl, hBSTl, hBSTrl⟩
  have hleftY : ∀ z ∈ toList (node l x rl), z < y := by
    intro z hz
    simp only [toList, List.mem_append, List.mem_singleton] at hz
    rcases hz with (hzl | hzeq) | hzrl
    · exact lt_trans (hlx z hzl) hxy
    · exact hzeq ▸ hxy
    · exact hrly z hzrl
  exact ⟨hleftY, hyrr, hBSTlxrl, hBSTrr⟩
```

https://claude.ai/chat/688827fb-f1ec-41b9-9ab1-c4aa7fa8c5cf

AvlRotation · LEAN

Copy

```
/-! ### 2. Rotation preserves the BST invariant -/

theorem rotateLeft_BST [LT α] (lt_trans : ∀ {a b c : α}, a < b → b < c → a < c)
    (l : Tree α) (x : α) (rl : Tree α) (y : α) (rr : Tree α)
    (h : BST (α := α) (node l x (node rl y rr))) :
    BST (α := α) (rotateLeft (node l x (node rl y rr))) := by
  obtain ⟨hlx, hxRight, hBSTl, hBSTright⟩ := h
  obtain ⟨hrly, hyrr, hBSTrl, hBSTrr⟩ := hBSTright
  have hxy : x < y := hxRight y (by simp [toList])
  have hxrl : ∀ z ∈ toList rl, x < z := by
    intro z hz
    exact hxRight z (by simp [toList]; exact Or.inl hz)
  have hBSTlxrl : BST (α := α) (node l x rl) := ⟨hlx, hxrl, hBSTl, hBSTrl⟩
  have hleftY : ∀ z ∈ toList (node l x rl), z < y := by
    intro z hz
    simp only [toList, List.mem_append, List.mem_singleton] at hz
    rcases hz with (hzl | hzeq) | hzrl
    · exact lt_trans (hlx z hzl) hxy
    · exact hzeq ▸ hxy
    · exact hrly z hzrl
  exact ⟨hleftY, hyrr, hBSTlxrl, hBSTrr⟩
```

≃ assembler

https://claude.ai/chat/688827fb-f1ec-41b9-9ab1-c4aa7fa8c5cf

```
fun content where
"content Leaf = []" |
"content (Node l x r) = content l @ [x] @ content r"

fun BST where
"BST Leaf = True" |
"BST (Node l x r) = ((∀ y∈set (content l). y < x) /

fun rotate_left where
"rotate_left (Node l x (Node rl y rr)) = Node (Node
"rotate_left t = t"

lemma content_rotate_left:
  "content (rotate_left t) = content t"
  sledgehammer
  by (smt (verit) append_assoc content.simps(2) rotate_left.elims)

lemma BST_rotate_left:
  fixes t :: "int tree"
  assumes "BST t"
  shows   "BST (rotate_left t)"
  using assms
  by (cases t rule: rotate_left.cases) auto
```

good abstraction level:
hammer + lemmas
(here found automatically)

even better:
both lemmas have an
obvious, “canonical” proof

Further features of next geneneration of tools

- AI enabled
- high-level & explainable reasoning at the level of the problem domain and input language (e.g., boilerplate and internal representation is suppressed)
- actually support full programming languages & libraries
- interoperable: types of tools, shared frontends, testing
- flexible correctness concerns (e.g. security, underapproximation, trace- and hyperproperties)
- integrate better with SE (people, processes, toolchains, documentation)

Verification with Trace Equation Systems