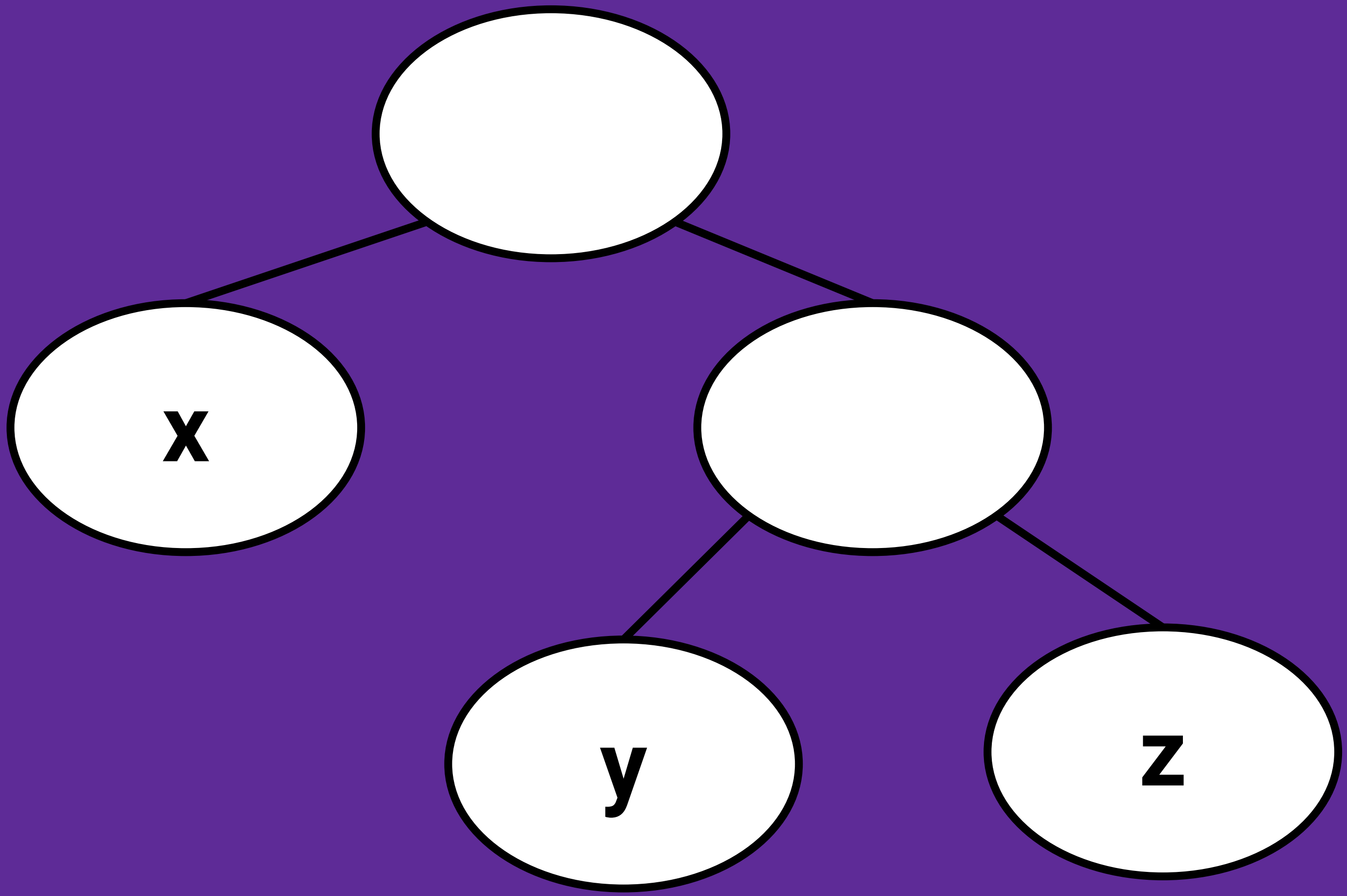
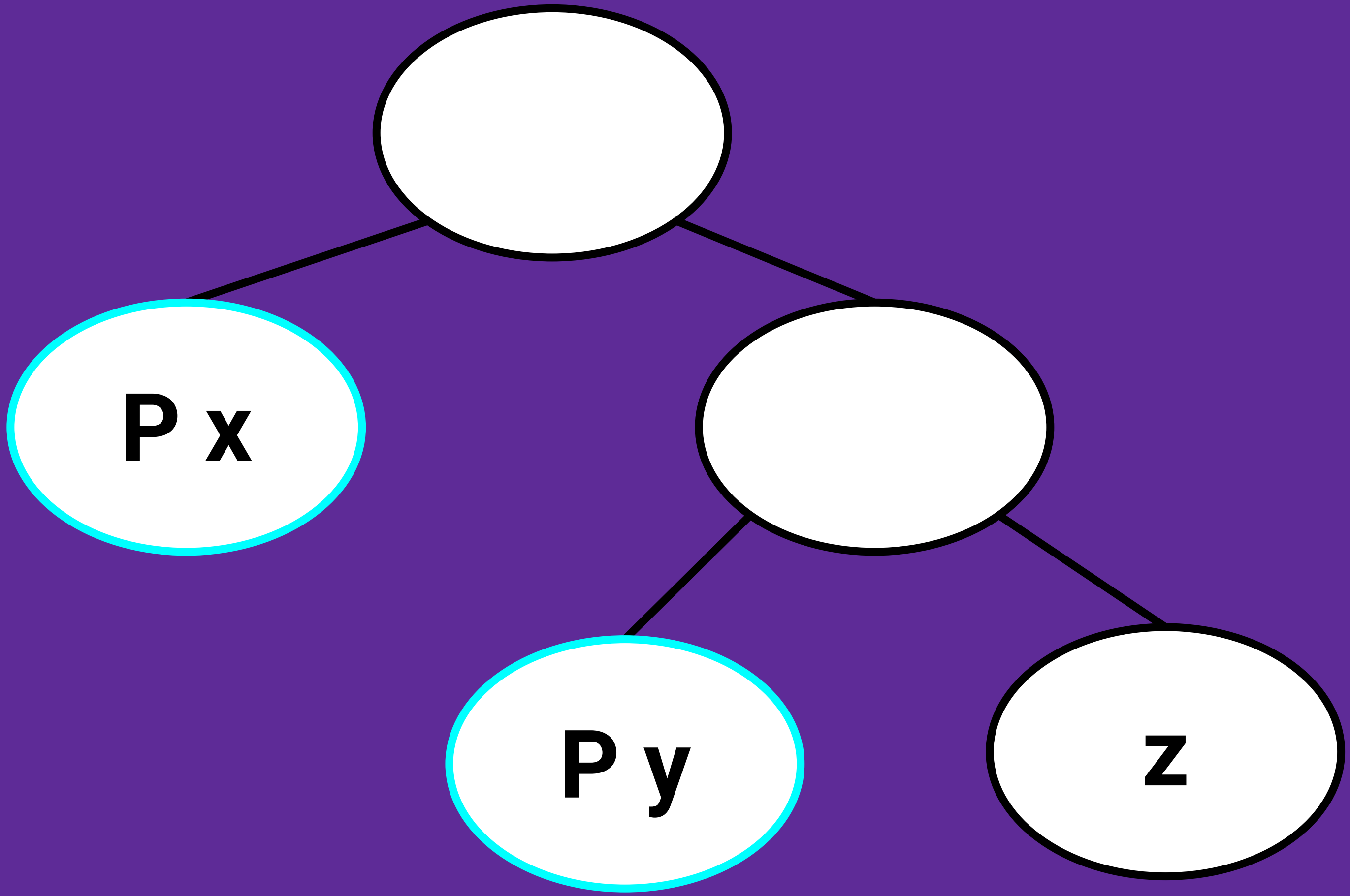
Bridging Neural & Symbolic Proof Automation

Talia Ringer
University of Illinois Urbana-Champaign

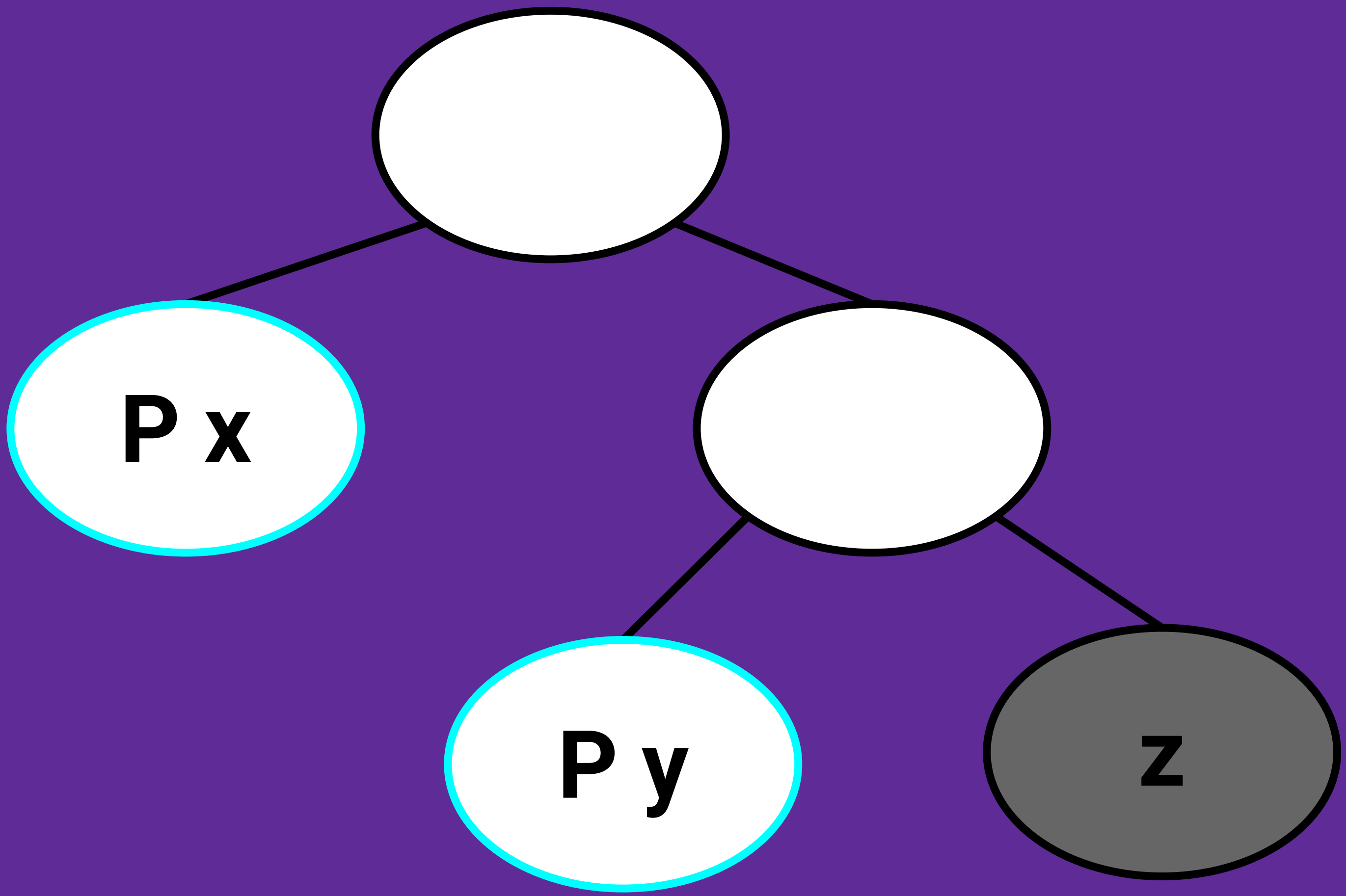
Tree Functions with ChatGPT



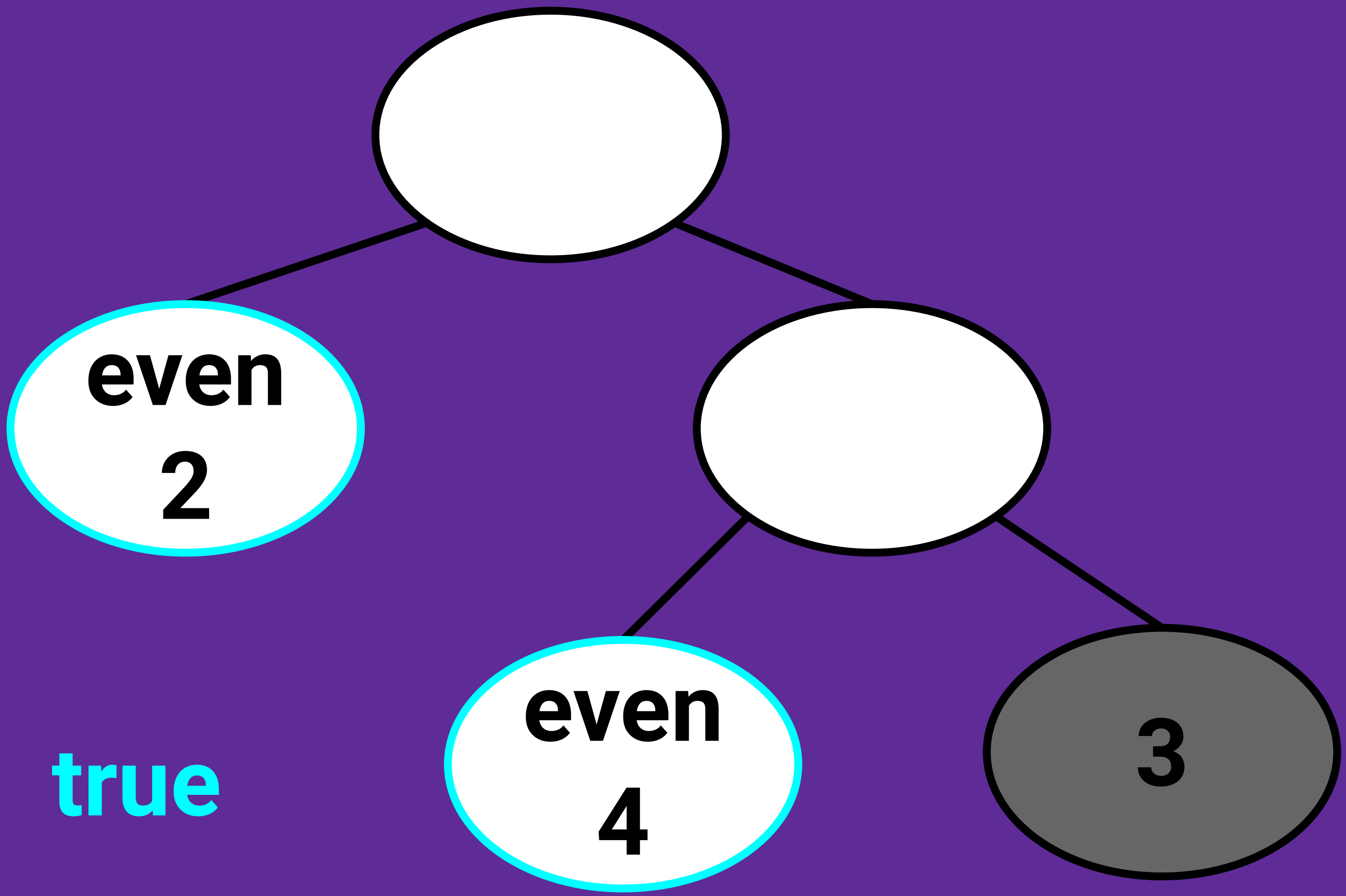
Tree Functions with ChatGPT



Tree Functions with ChatGPT

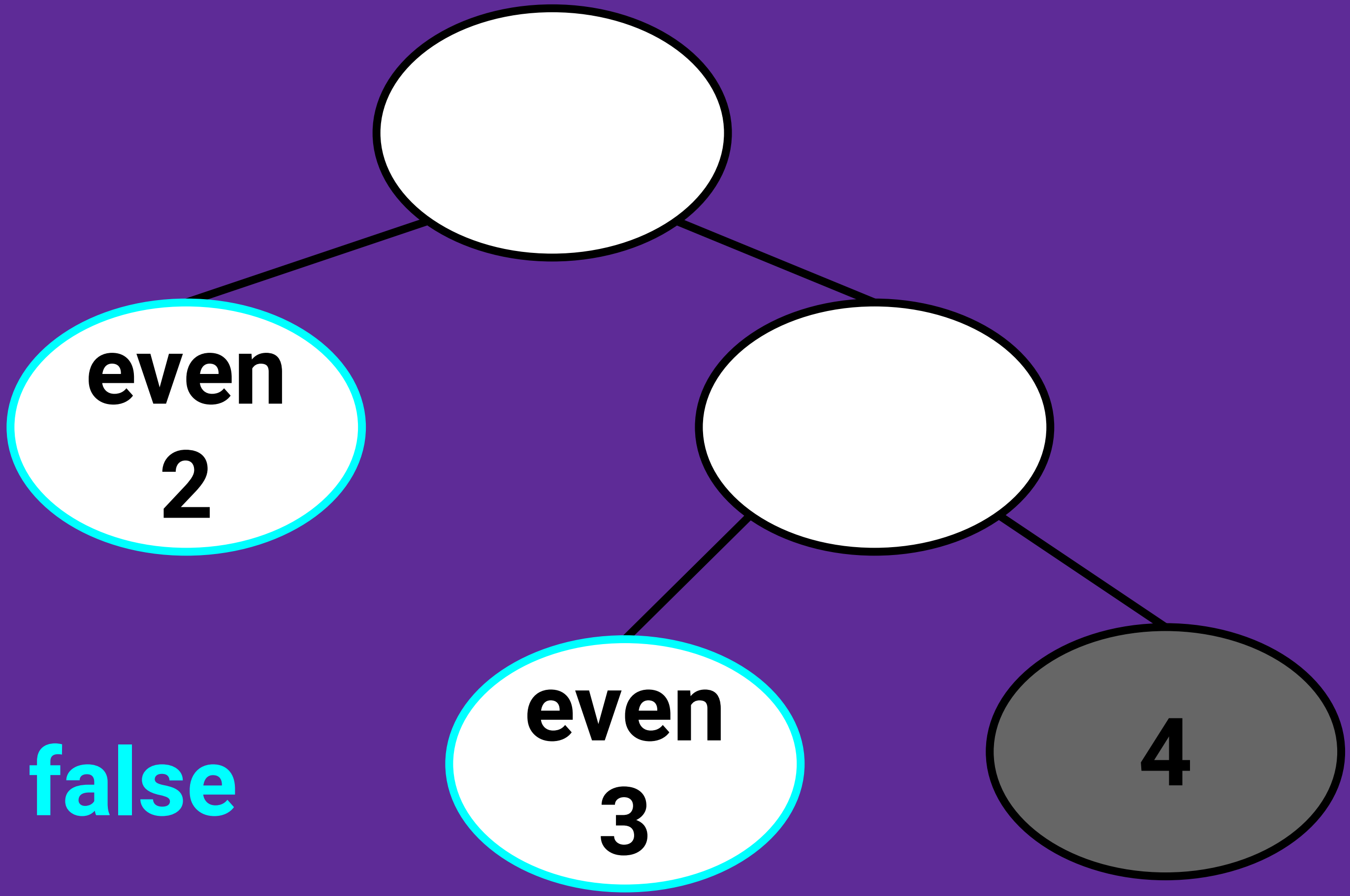


Tree Functions with ChatGPT



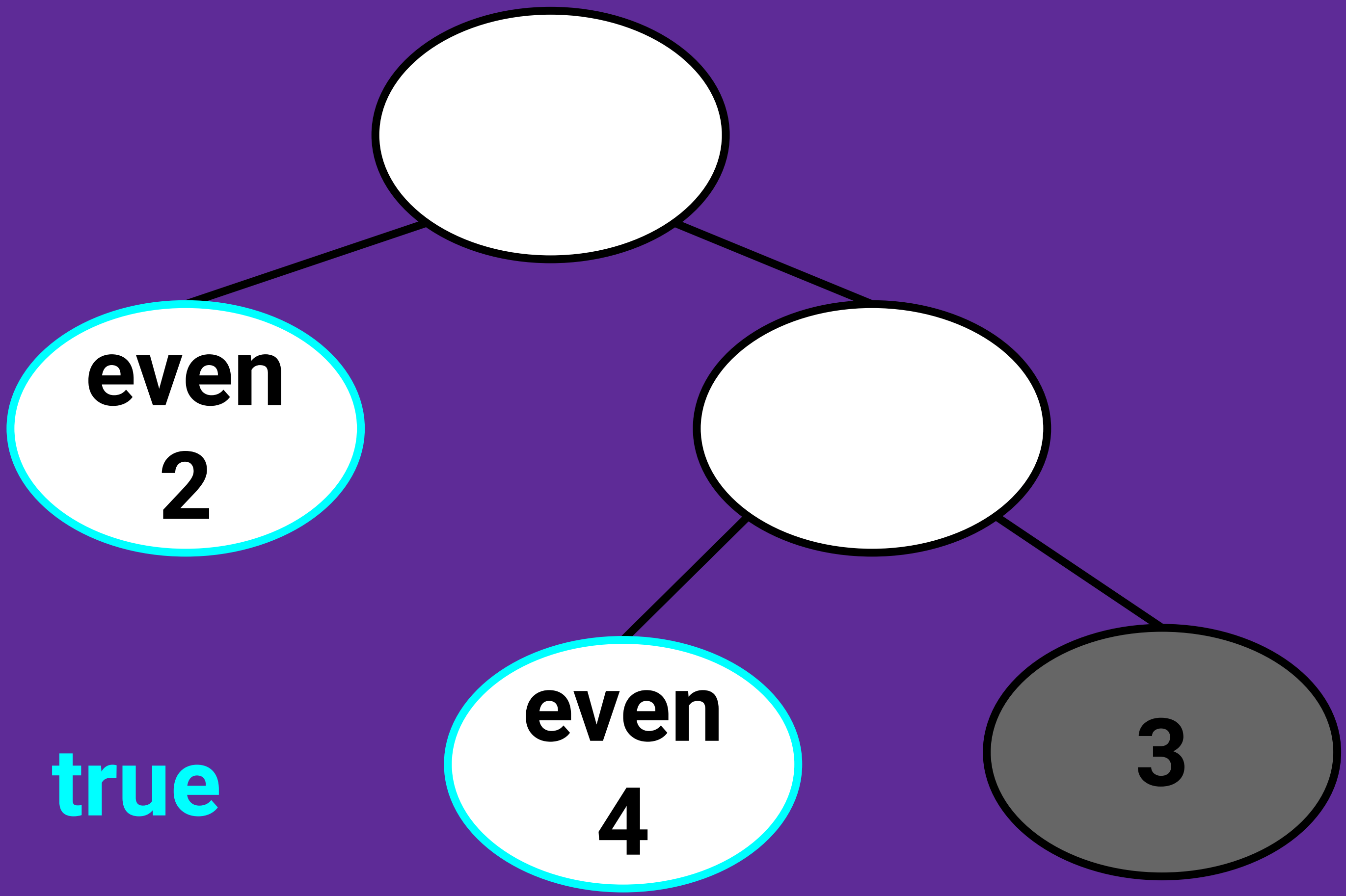
true

Tree Functions with ChatGPT



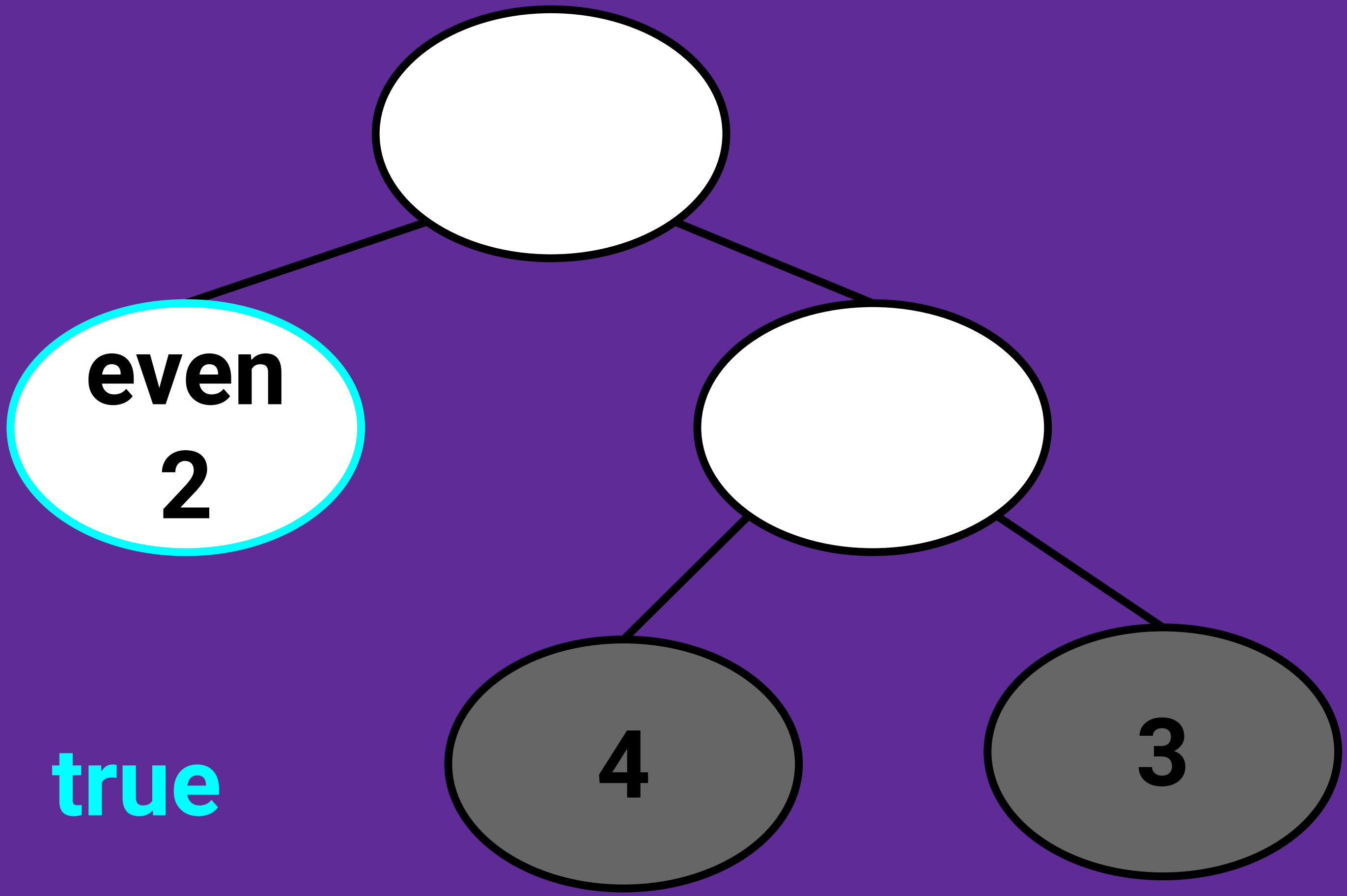
false

Tree Functions with ChatGPT



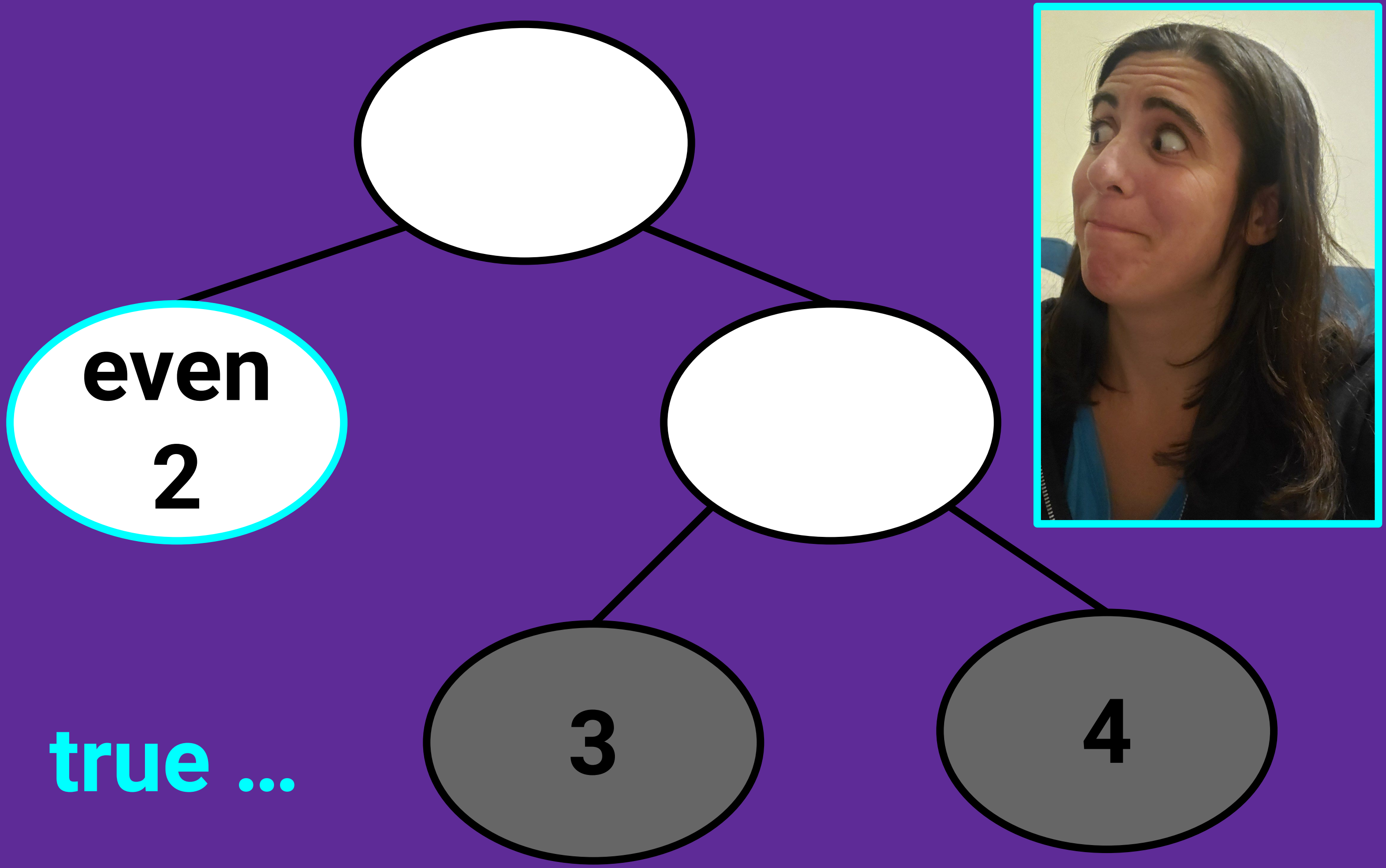
true

Tree Functions with ChatGPT



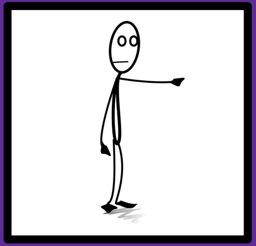
true

Tree Functions with ChatGPT



true ...

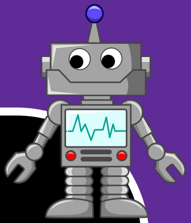
Tree Functions with ChatGPT



The code you provided doesn't count left leaves that occur on right branches, but I want it to. Can you fix it so that it does? For example, this code should print false since 3 is not even:

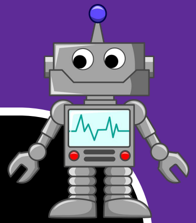
```
let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree)
```

Tree Functions with ChatGPT



```
let rec check_predicate_left_leaves predicate tree =
  let rec aux is_left = function
  | Leaf x -> if is_left then predicate x else true
  | Node (left, right) -> (aux true left) && (aux false right)
  in aux true tree

let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree)
```

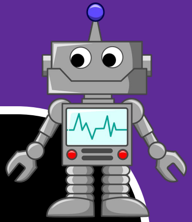
Tree Functions with ChatGPT



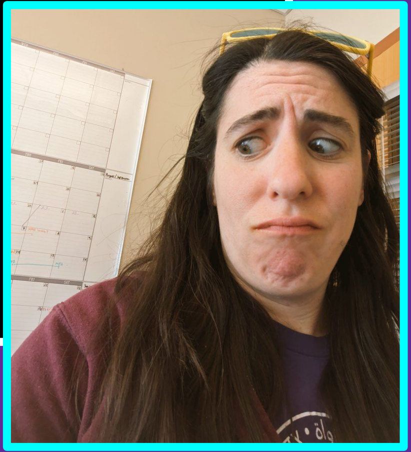
```
let rec check_predicate_left_leaves predicate tree =
  let rec aux is_left = function
  | Leaf x -> if is_left then predicate x else true
  | Node (left, right) -> (aux true left) && (aux false right)
  in aux true tree

let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree)
```

Passes test!

Tree Functions with ChatGPT



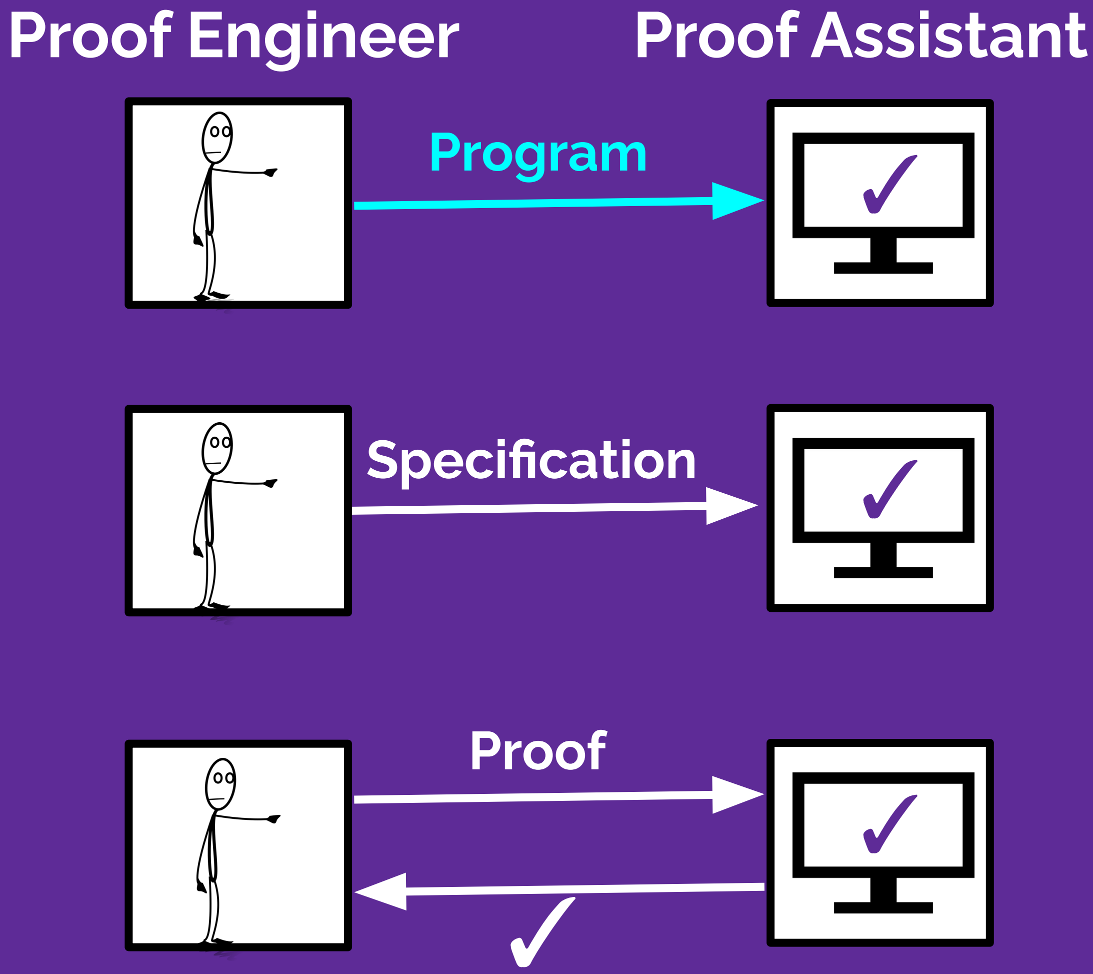
```
let rec check_predicate_left_leaves predicate tree =
  let rec aux is_left = function
  | Leaf x -> if is_left then predicate x else true
  | Node (left, right) -> (aux true left) && (aux false right)
  in aux true tree

let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree)
```



Passes test!

Proof Assistants

Proof Assistants for Verification



Proof Engineer

Proof Assistant

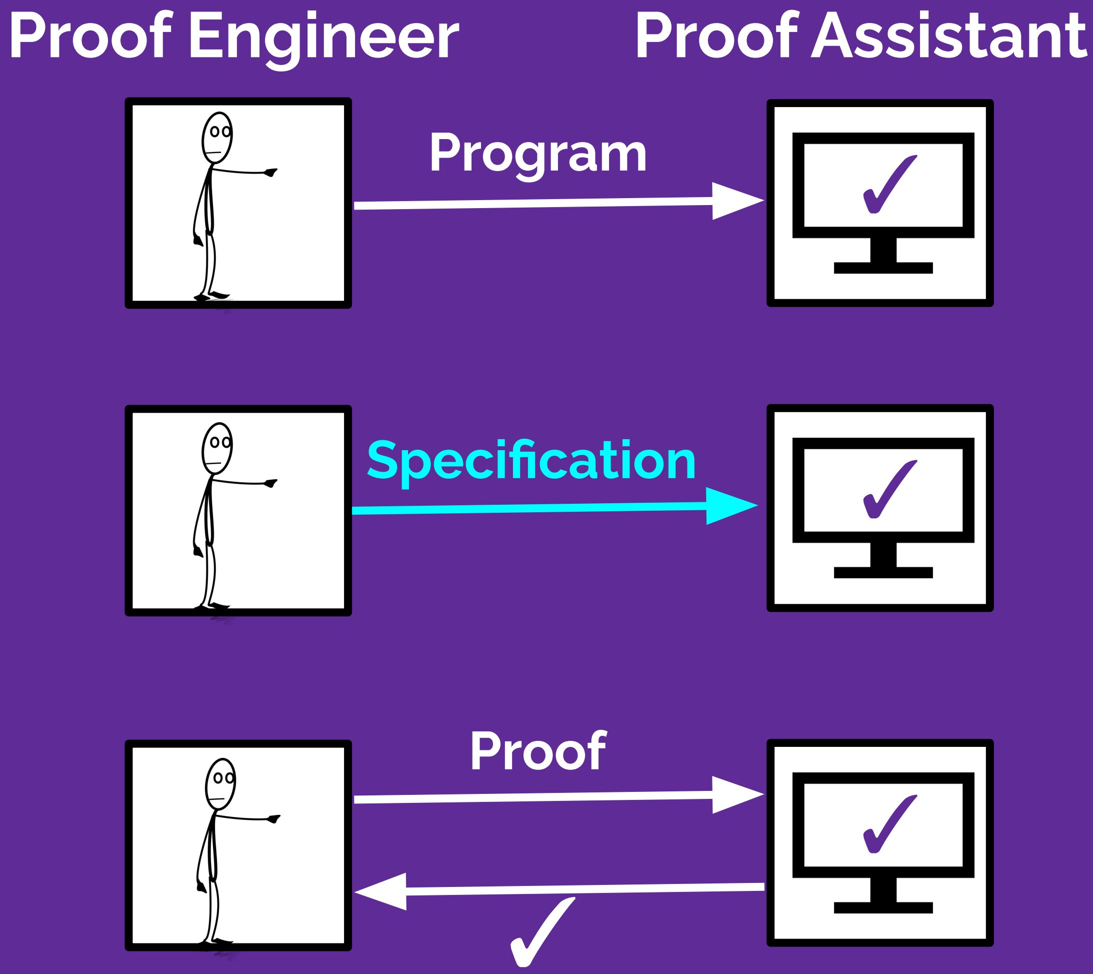
Program

Specification

Proof

✓

Proof Assistants for Verification



Proof Engineer

Proof Assistant

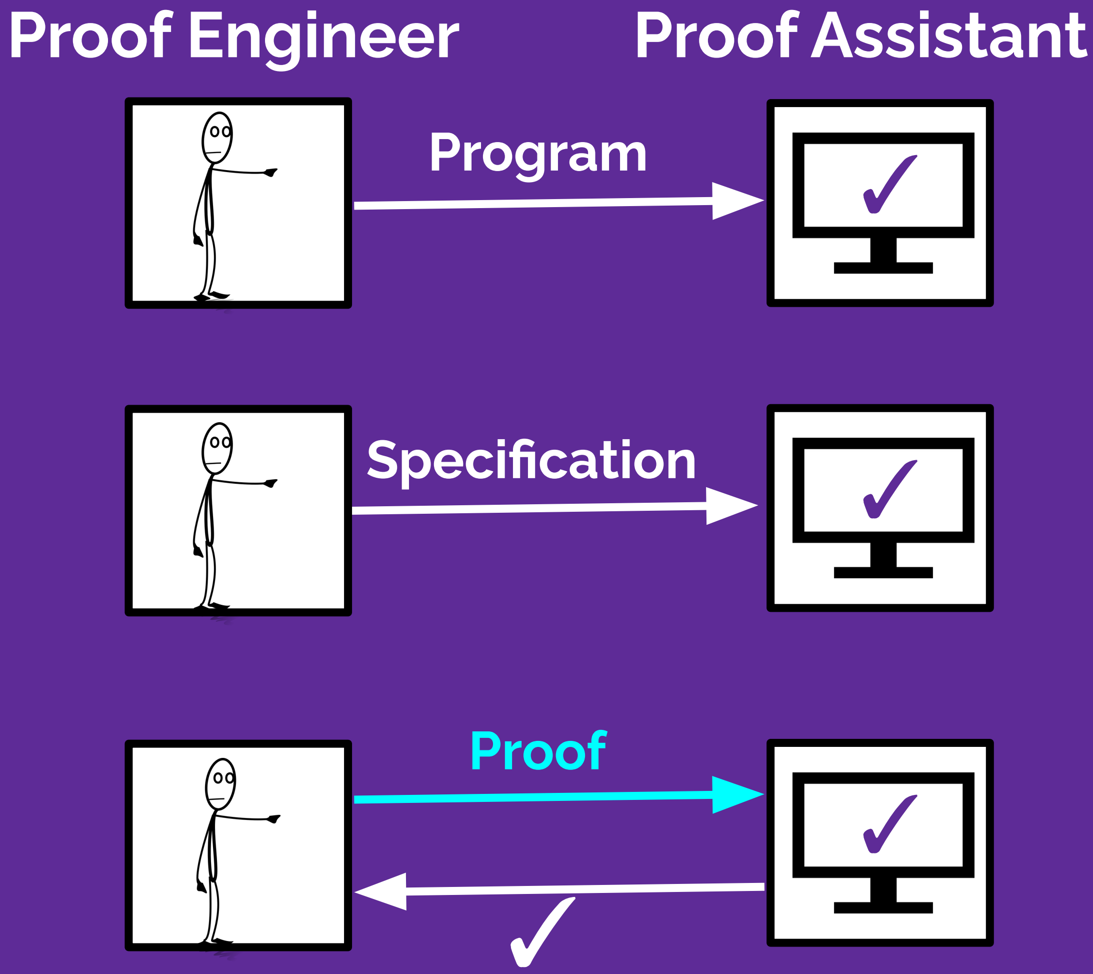
Program

Specification

Proof

✓

Proof Assistants for Verification



Proof Engineer **Proof Assistant**

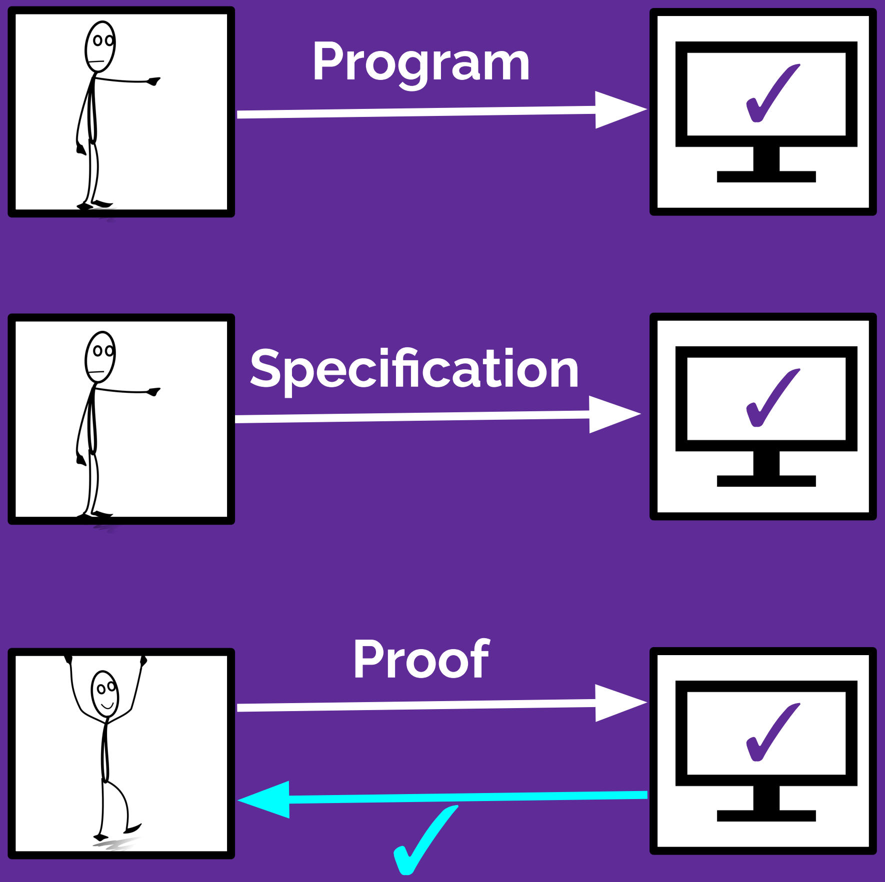
Program

Specification

Proof

✓

Proof Assistants for Verification



Proof Engineer | **Proof Assistant**

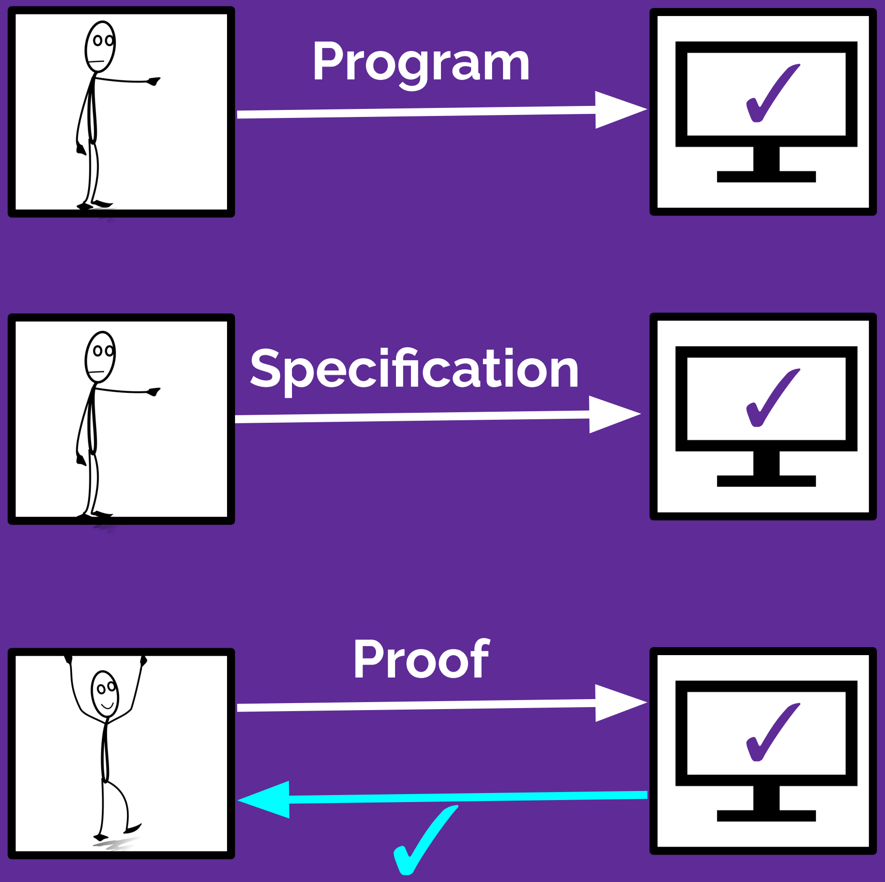
Program

Specification

Proof

✓

Proof Assistants for Verification



Proof Engineer

Proof Assistant

Program

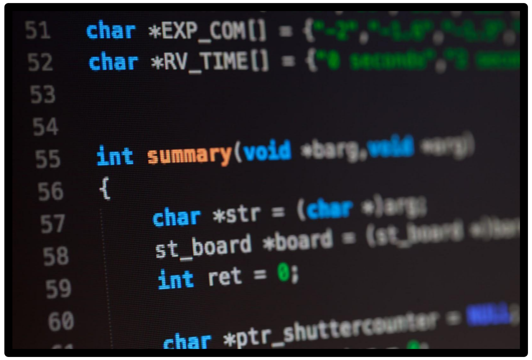
Specification

Proof

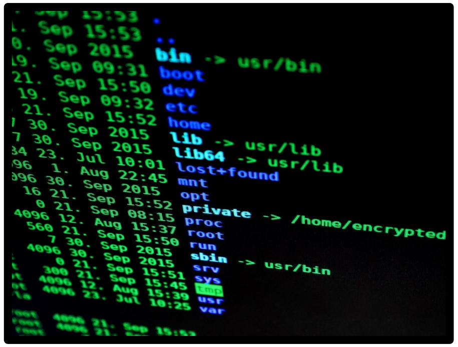
✓

16 Hours Later ...

Proof Assistants for Verification

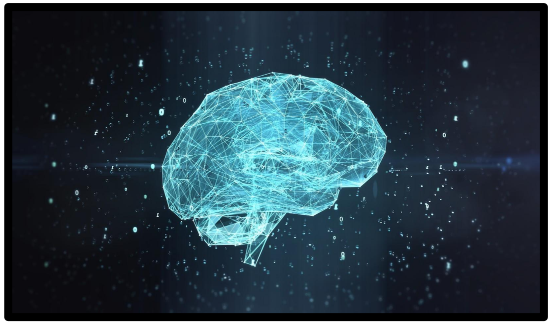


Compilers



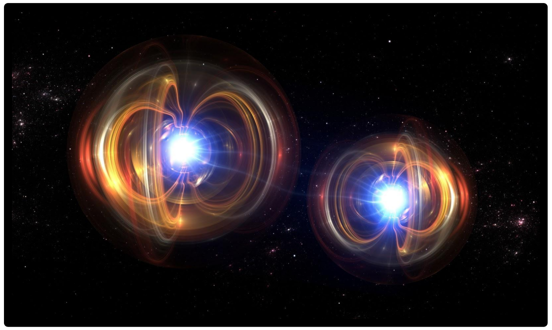
File Systems

Web Browsers



Machine Learning Systems

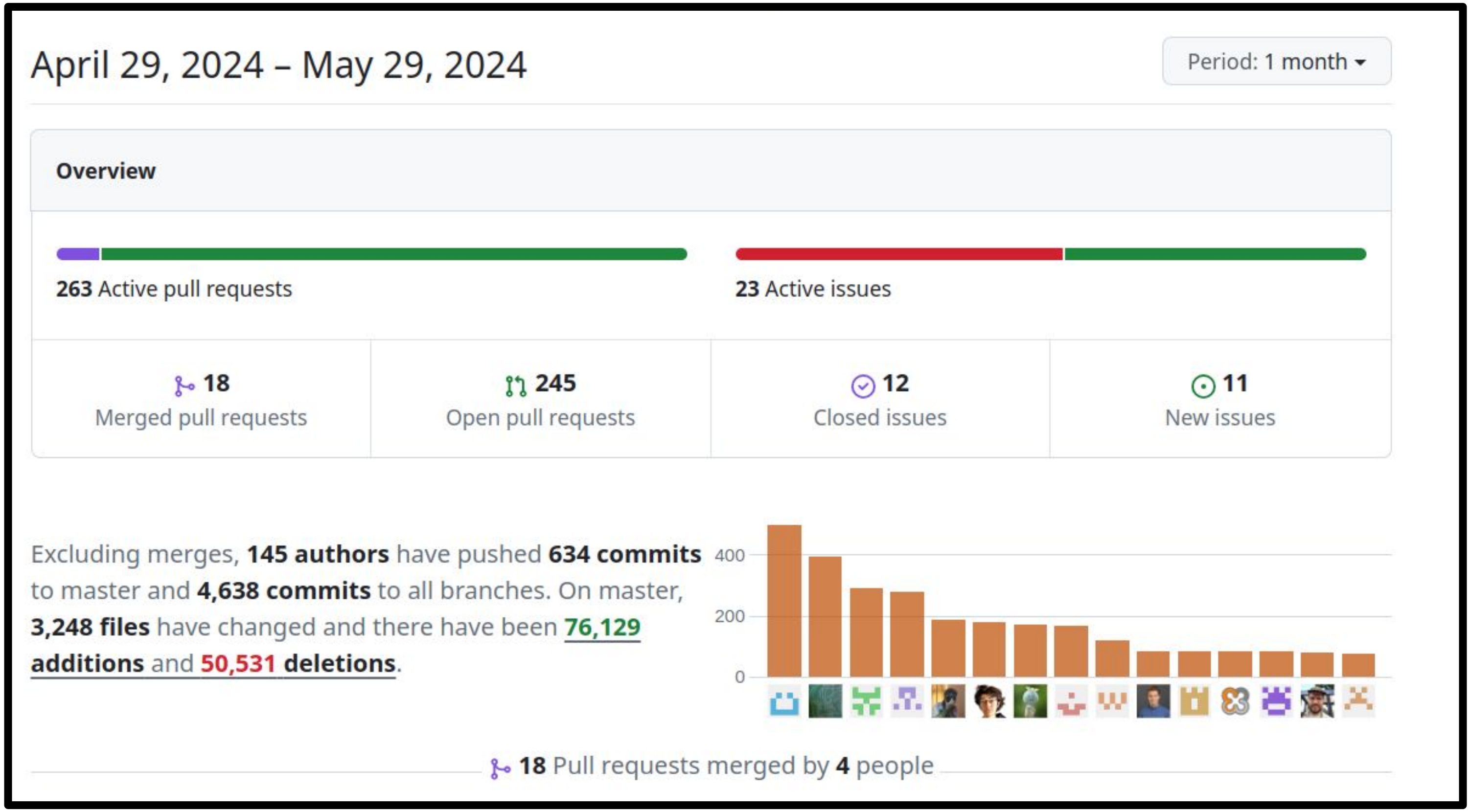
Operating Systems



Quantum Optimizers

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric and Zachary Tatlock (2019), QED at Large: A Survey of Engineering of Formally Verified Software, Foundations and Trends in Programming Languages: Vol. 5, No. 2-3, pp 102–281.

Proof Assistants for Math



April 29, 2024 – May 29, 2024

Overview

263 Active pull requests

23 Active issues

18 Merged pull requests	**245** Open pull requests	**12** Closed issues	**11** New issues

Excluding merges, **145 authors** have pushed **634 commits** to master and **4,638 commits** to all branches. On master, **3,248 files** have changed and there have been **76,129 additions** and **50,531 deletions**.

18 Pull requests merged by **4** people

https://github.com/leanprover-community/mathlib4/pulse/monthly

It’s still hard to write proofs.

Proof automation makes it easier to develop and maintain formal proofs using proof assistants.

Traditional automation:

+ predictable
+ dependable
+ understandable
- limited in scope
- takes expertise to extend

Symbolic automation:

+ predictable
+ dependable
+ understandable
- limited in scope
- takes expertise to extend

Symbolic automation:

+ predictable
+ dependable
+ understandable
- limited in scope
- takes expertise to extend

Neural automation:

- *un*predictable
- *not* dependable
- *not* understandable

\+ *not very* limited in scope
\+ can take *little* expertise to extend

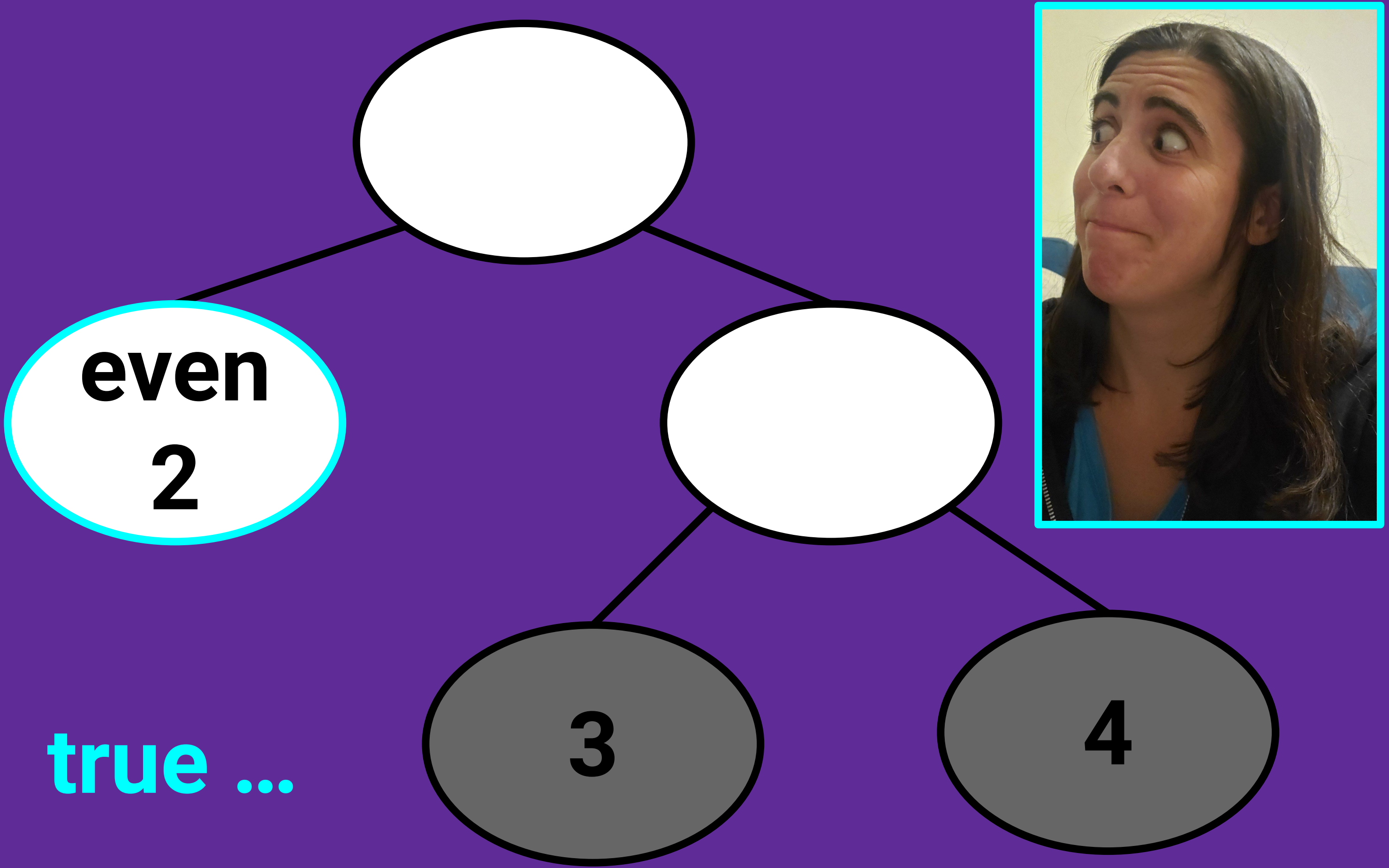
Best of both worlds?

+ predictable
+ dependable
+ understandable
+ *not very* limited in scope
+ can take *little* expertise to extend

Though proof assistants have come a long way, they are still hard for most people to use. **We can make this easier.**

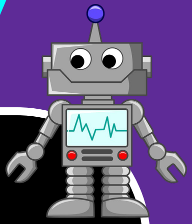
Though proof assistants have come a long way, they are still hard for most people to use. We must make this easier, now.

We Must Make Proofs Easier, Now

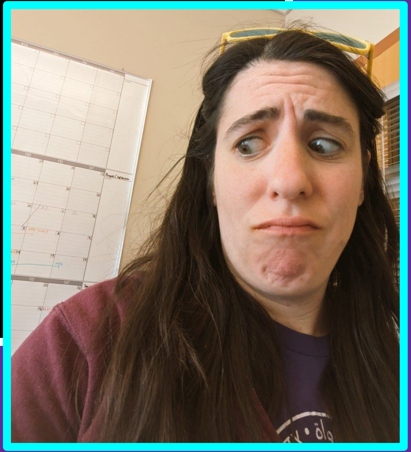


true …

We Must Make Proofs Easier, Now



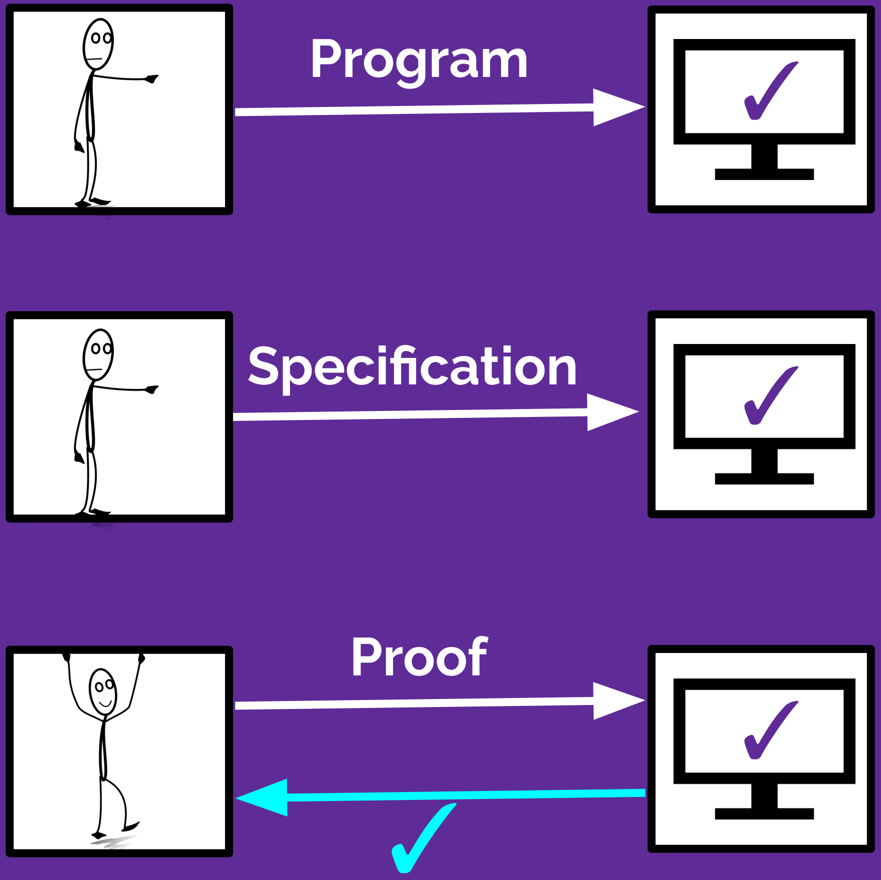
```
let rec check_predicate_left_leaves predicate tree =
  let rec aux is_left = function
  | Leaf x -> if is_left then predicate x else true
  | Node (left, right) -> (aux true left) && (aux false right)
  in aux true tree

let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree)
```



Passes test!

We Must Make Proofs Easier, Now

16 Hours Later ...



Proof Engineer

Proof Assistant

Program

Specification

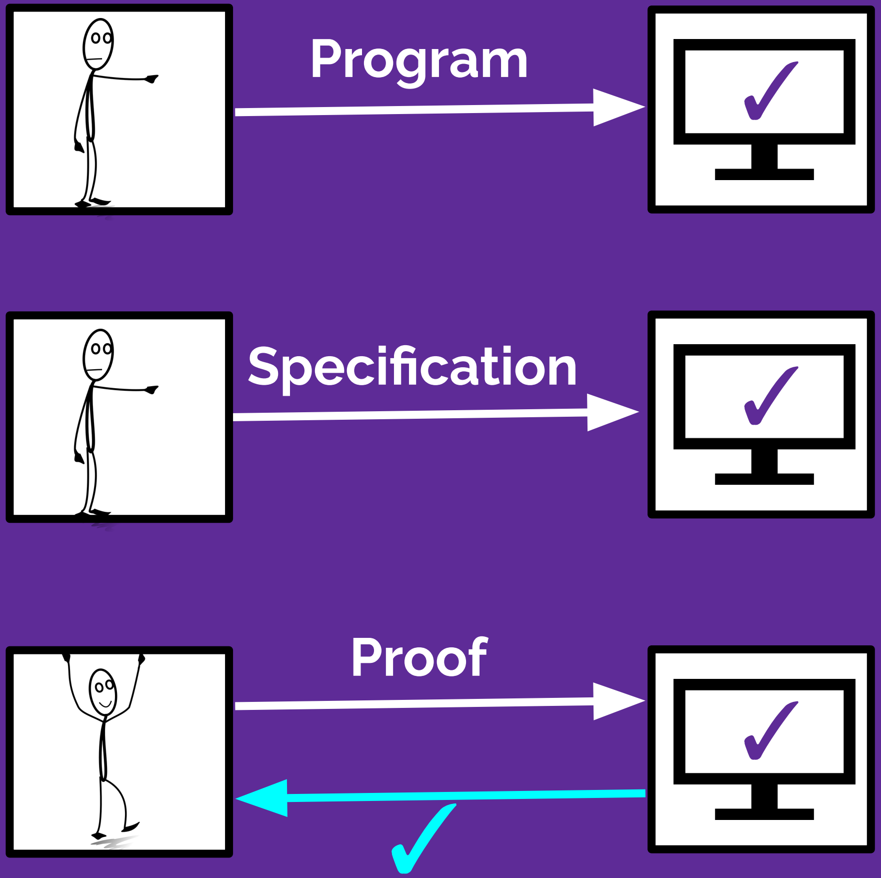
Proof

✓

We Must Make Proofs Easier, Now

Proof Engineer

Proof Assistant



Program

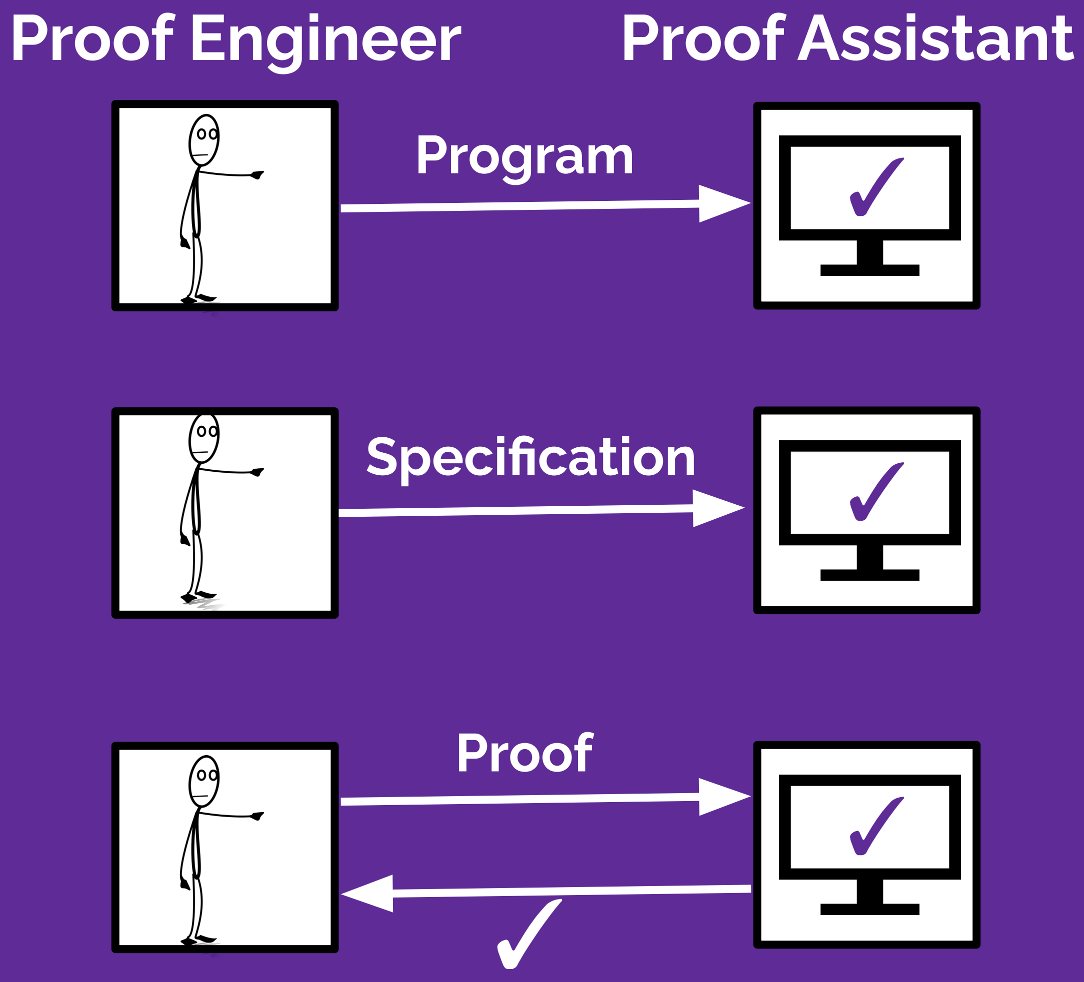
Specification

Proof

✓

1. Proof Assistants
2. Symbolic Automation
3. Neural Automation
4. Building Bridges
5. Opportunities

1. Proof Assistants
2. Symbolic Automation
3. Neural Automation
4. Building Bridges
5. Opportunities



Proof Engineer

Proof Assistant

Program

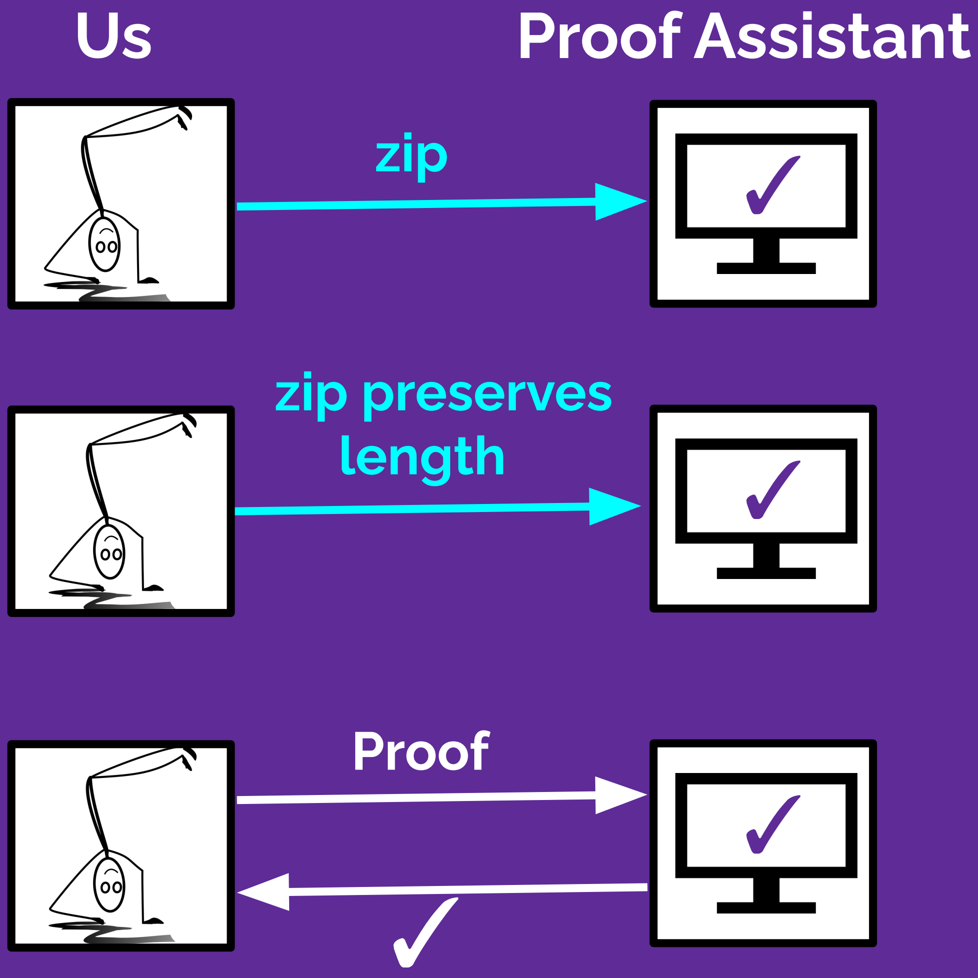
Specification

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Proof Assistant

zip

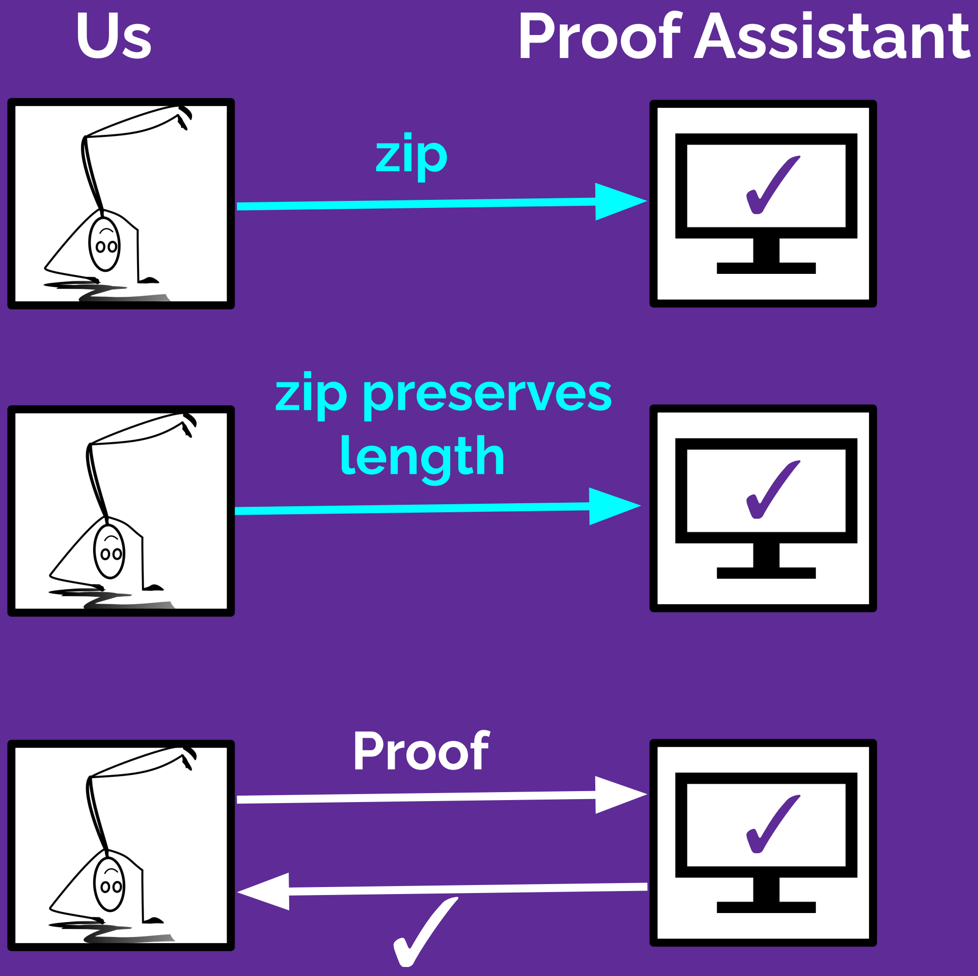
zip preserves length

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Proof Assistant

zip

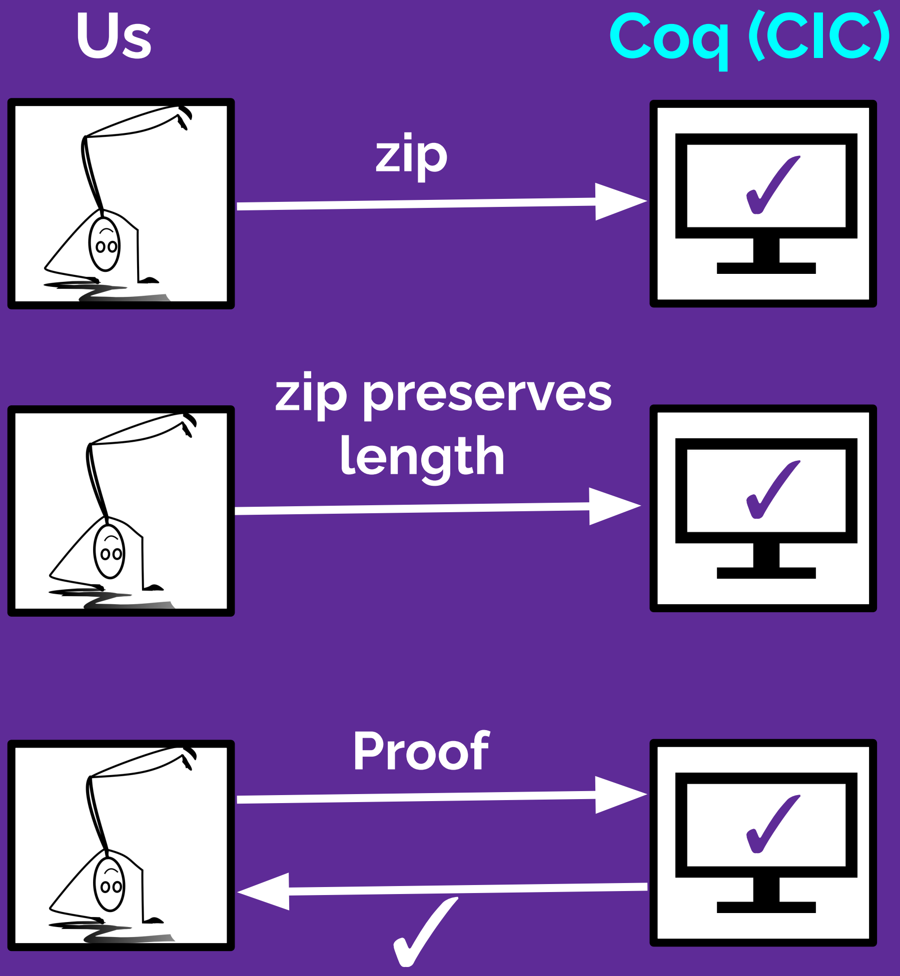
zip preserves length

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Coq (CIC)

zip

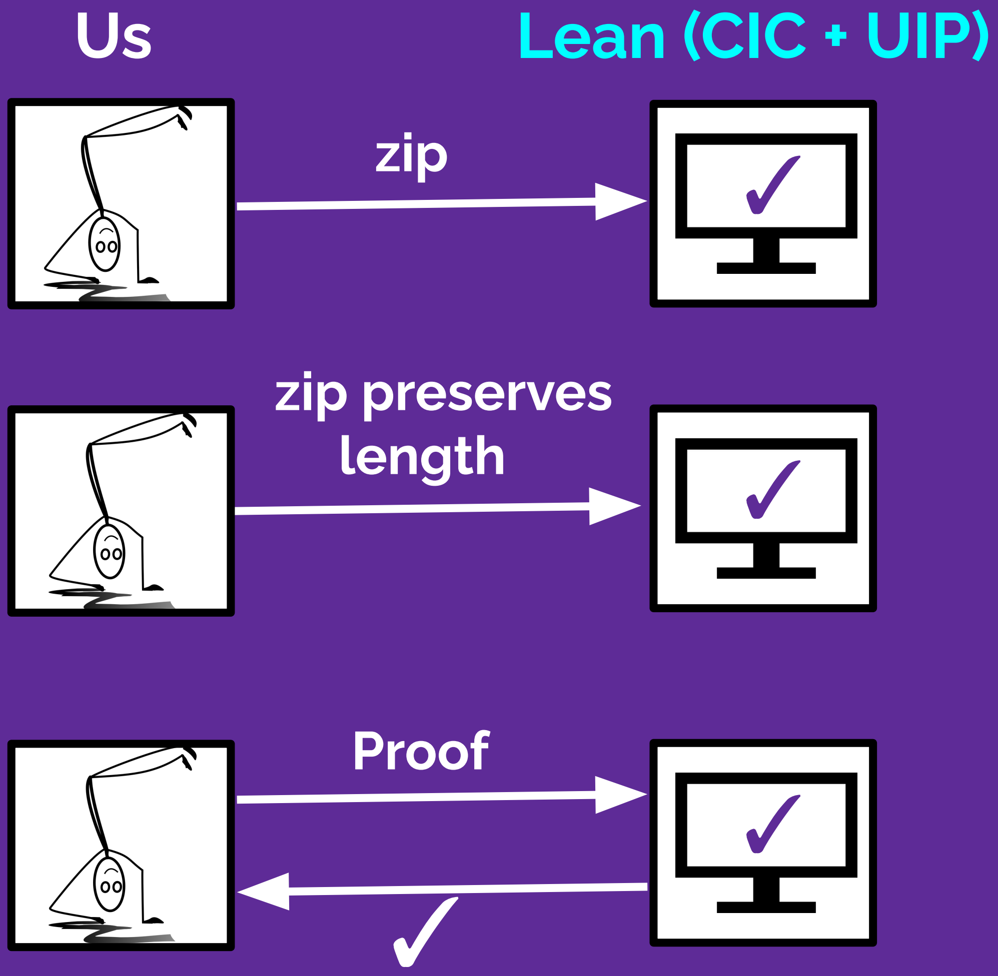
zip preserves length

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Lean (CIC + UIP)

zip

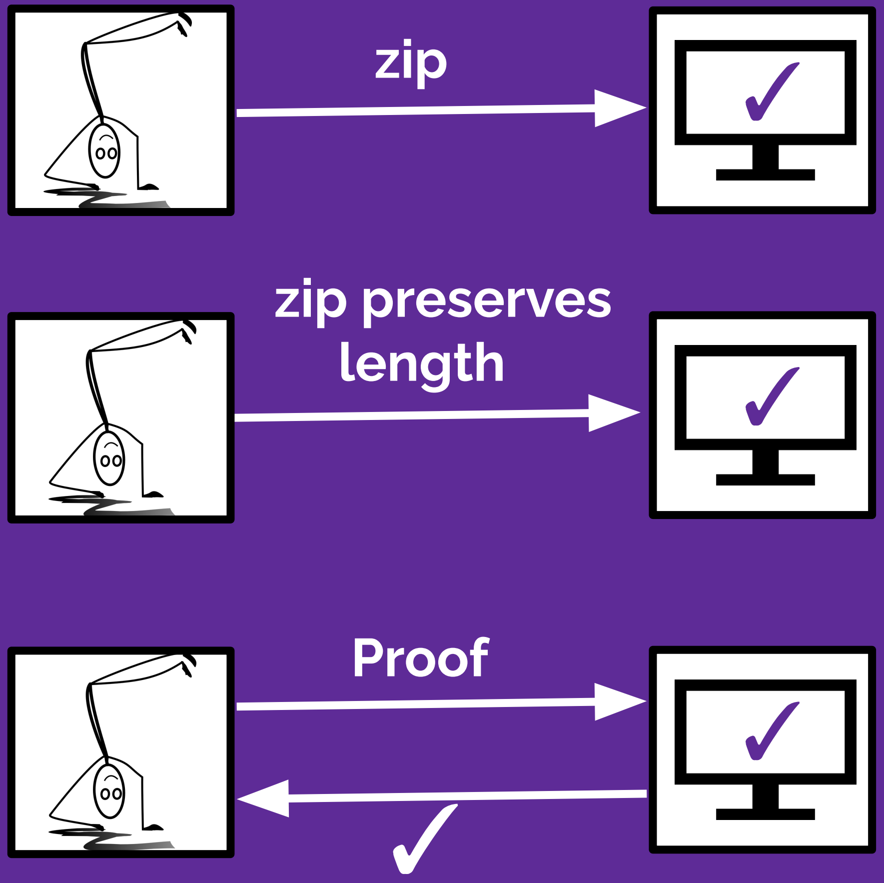
zip preserves length

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Cubical Agda (Cubical)

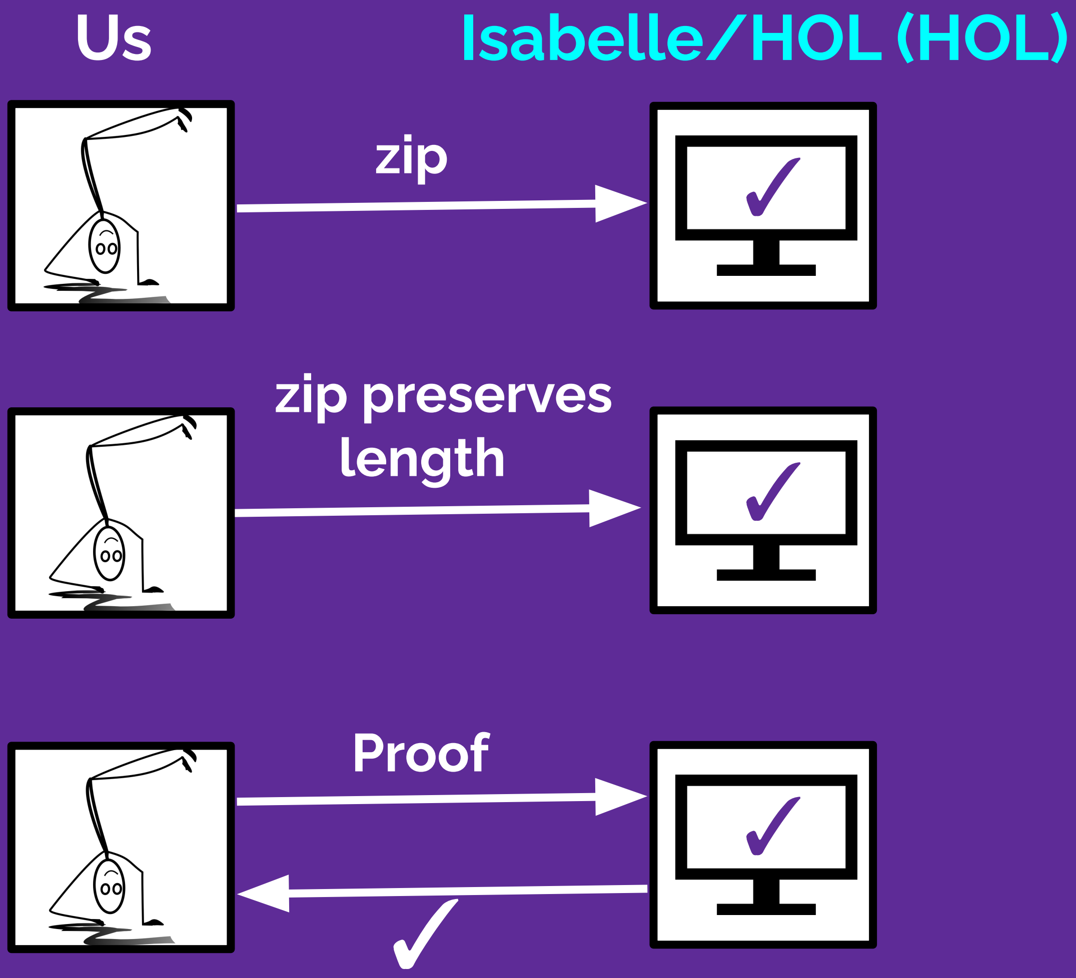
zip

zip preserves length

Proof

✓

List Zip Preserves Length



Us

Isabelle/HOL (HOL)

zip

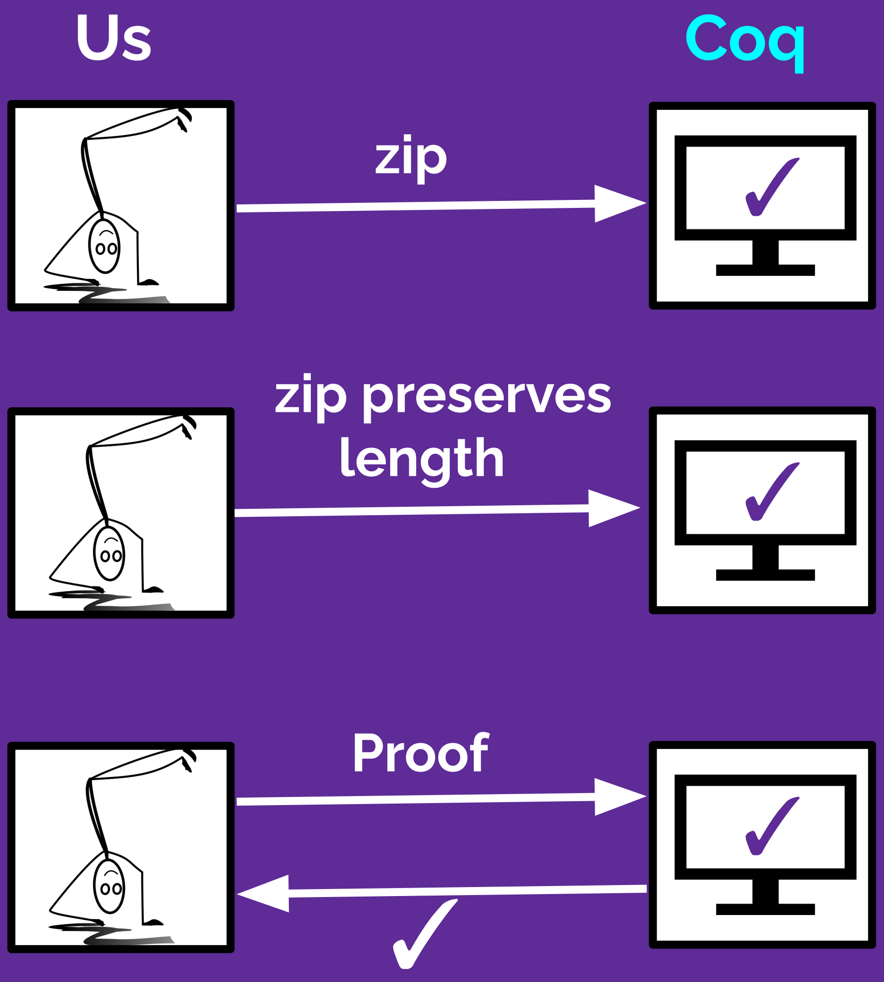
zip preserves length

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Coq

zip

zip preserves length

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length

```
Inductive list T :=
| nil : list T
| cons : T → list T → list T
```

Proof Assistants (Part 1 of 5)

List Zip Preserves Length

```
Inductive list T :=
| nil : list T
| cons : T → list T → list T
```

Proof Assistants (Part 1 of 5)

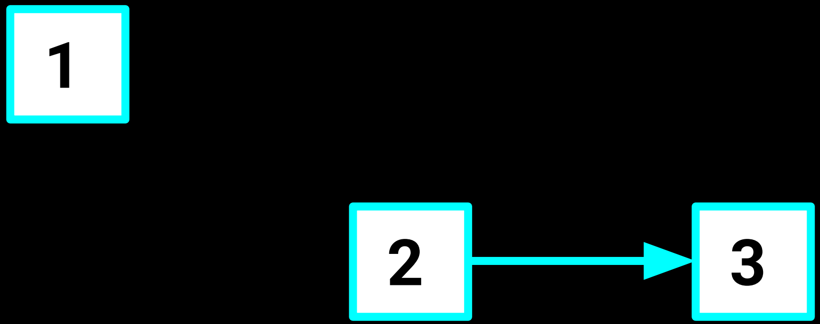
List Zip Preserves Length

Inductive list T :=
| **nil : list T** (* [ ] *)
| cons : T → list T → list T

Proof Assistants (Part 1 of 5)

List Zip Preserves Length

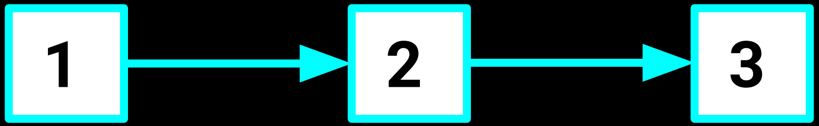
```
Inductive list T :=
| nil : list T (* [ ] *)
| cons : T → list T → list T  (* t :: l *)
```



Proof Assistants (Part 1 of 5)

List Zip Preserves Length

```
Inductive list T :=
| nil : list T (* [ ] *)
| cons : T → list T → list T (* t :: l *)
```



Proof Assistants (Part 1 of 5)

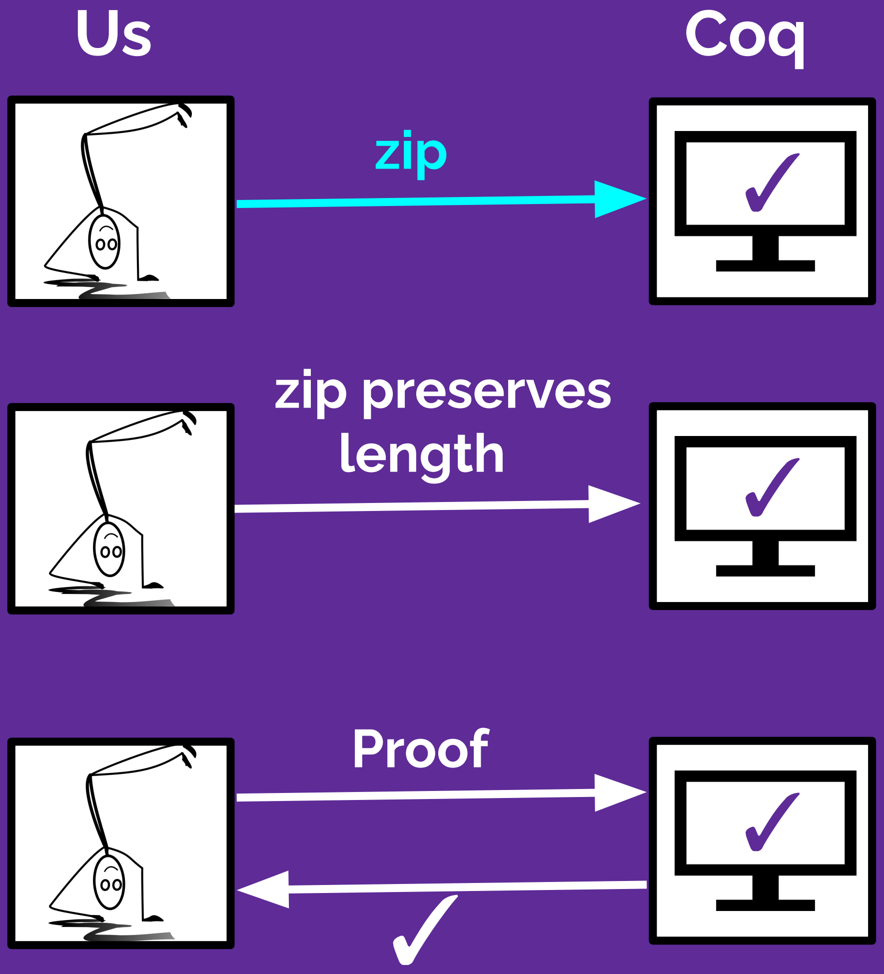
List Zip Preserves **Length**

Inductive list T :=
| nil : list T (* [ ] *)
| cons : T → list T → list T (* t :: l *)

length : list T → nat

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Coq

zip

zip preserves length

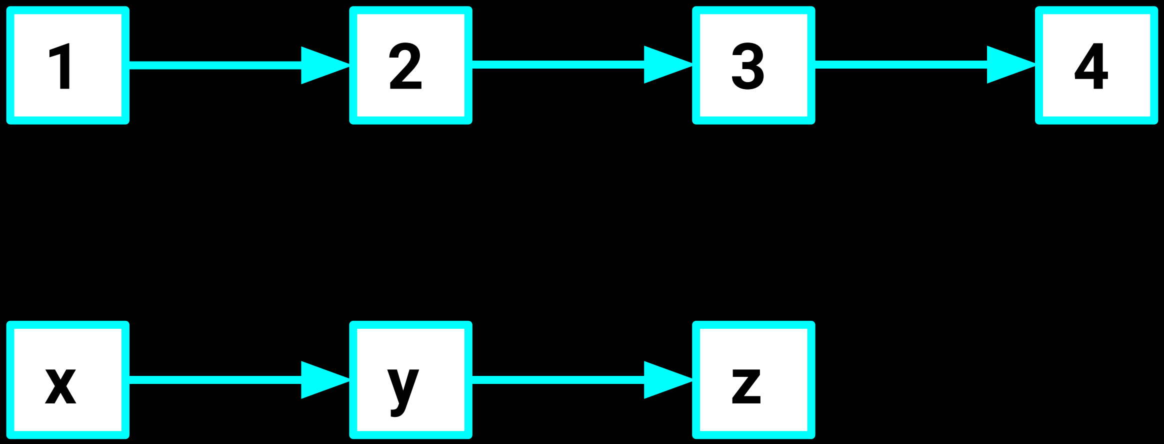
Proof

✓

List Zip Preserves Length

Modified from hs-to-coq

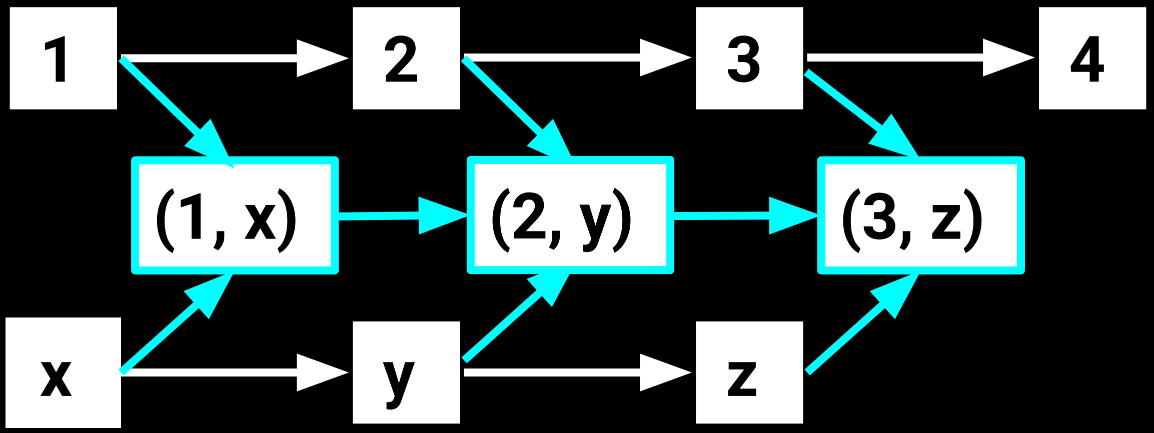
```
Fixpoint zip {A B} (l1 : list A) (l2 : list B) : list (A * B) :=
  match l1, l2 with
  | [ ], _ -> [ ]
  | _, [ ] -> [ ]
  | h1 :: tl1, h2 :: tl2 -> (h1, h2) :: (zip tl1 tl2)
  end.
```



List Zip Preserves Length

Modified from hs-to-coq

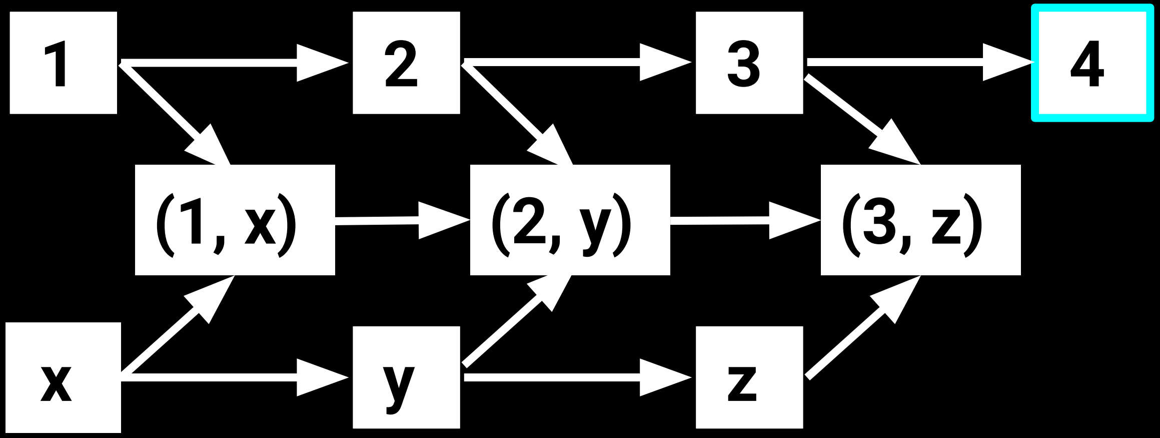
```
Fixpoint zip {A B} (l1 : list A) (l2 : list B) : list (A * B) :=
  match l1, l2 with
  | [ ], _ -> [ ]
  | _, [ ] -> [ ]
  | h1 :: tl1, h2 :: tl2 -> (h1, h2) :: (zip tl1 tl2)
  end.
```



List **Zip** Preserves Length

Modified from hs-to-coq

```
Fixpoint zip {A B} (l1 : list A) (l2 : list B) : list (A * B) :=
  match l1, l2 with
  | [ ], _ -> [ ]
  | _, [ ] -> [ ]
  | h1 :: tl1, h2 :: tl2 -> (h1, h2) :: (zip tl1 tl2)
  end.
```

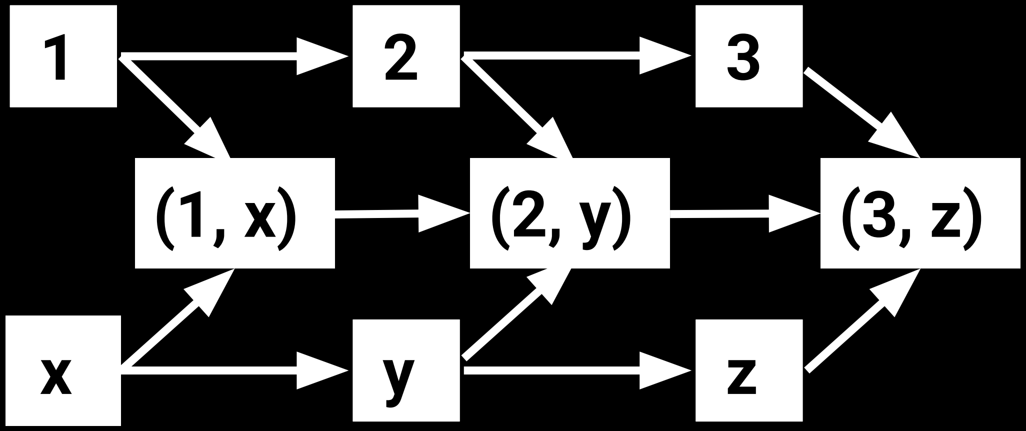


Proof Assistants (Part 1 of 5)

List Zip Preserves Length

Modified from hs-to-coq

```
Fixpoint zip {A B} (l1 : list A) (l2 : list B) : list (A * B) :=
  match l1, l2 with
  | [ ], _ -> [ ]
  | _, [ ] -> [ ]
  | h1 :: tl1, h2 :: tl2 -> (h1, h2) :: (zip tl1 tl2)
  end.
```

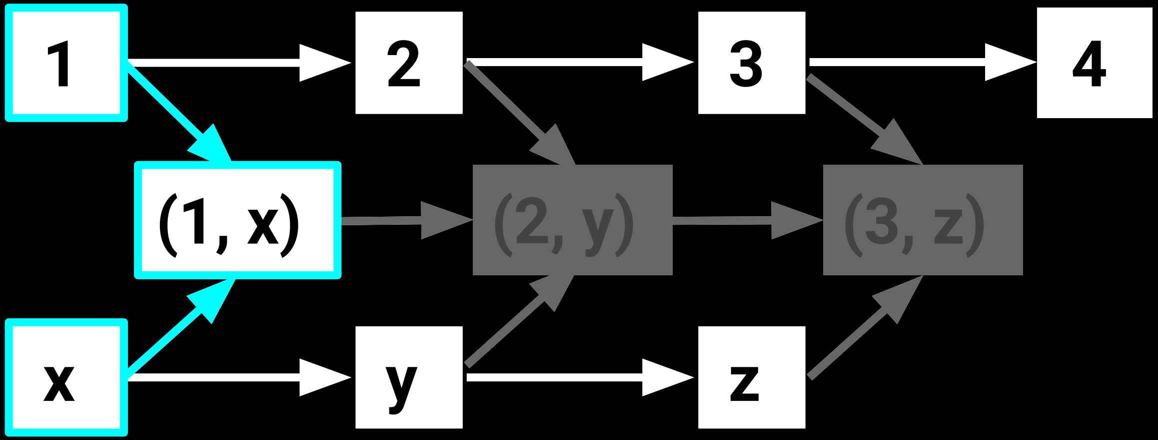


Proof Assistants (Part 1 of 5)

List Zip Preserves Length

Modified from hs-to-coq

```
Fixpoint zip {A B} (l1 : list A) (l2 : list B) : list (A * B) :=
  match l1, l2 with
  | [ ], _ -> [ ]
  | _, [ ] -> [ ]
  | h1 :: tl1, h2 :: tl2 -> (h1, h2) :: (zip tl1 tl2)
  end.
```

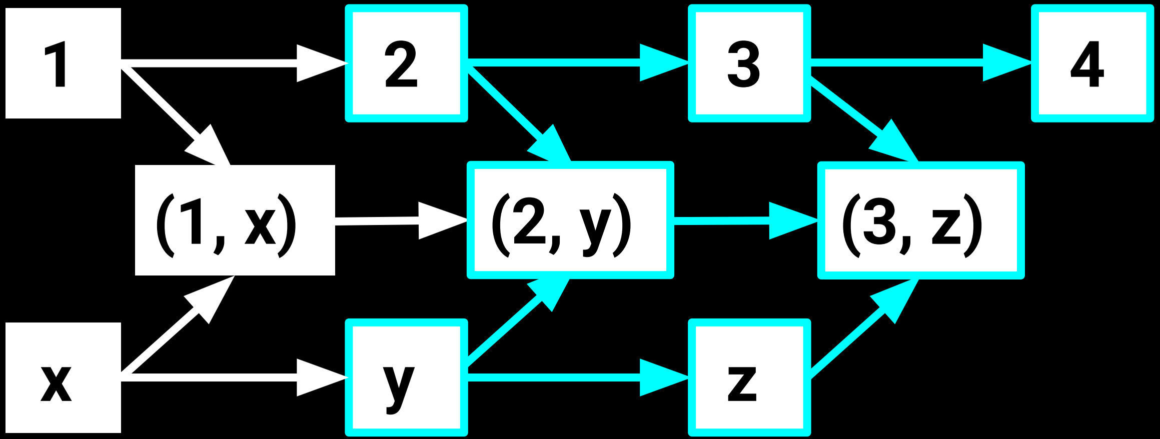


Proof Assistants (Part 1 of 5)

List **Zip** Preserves Length

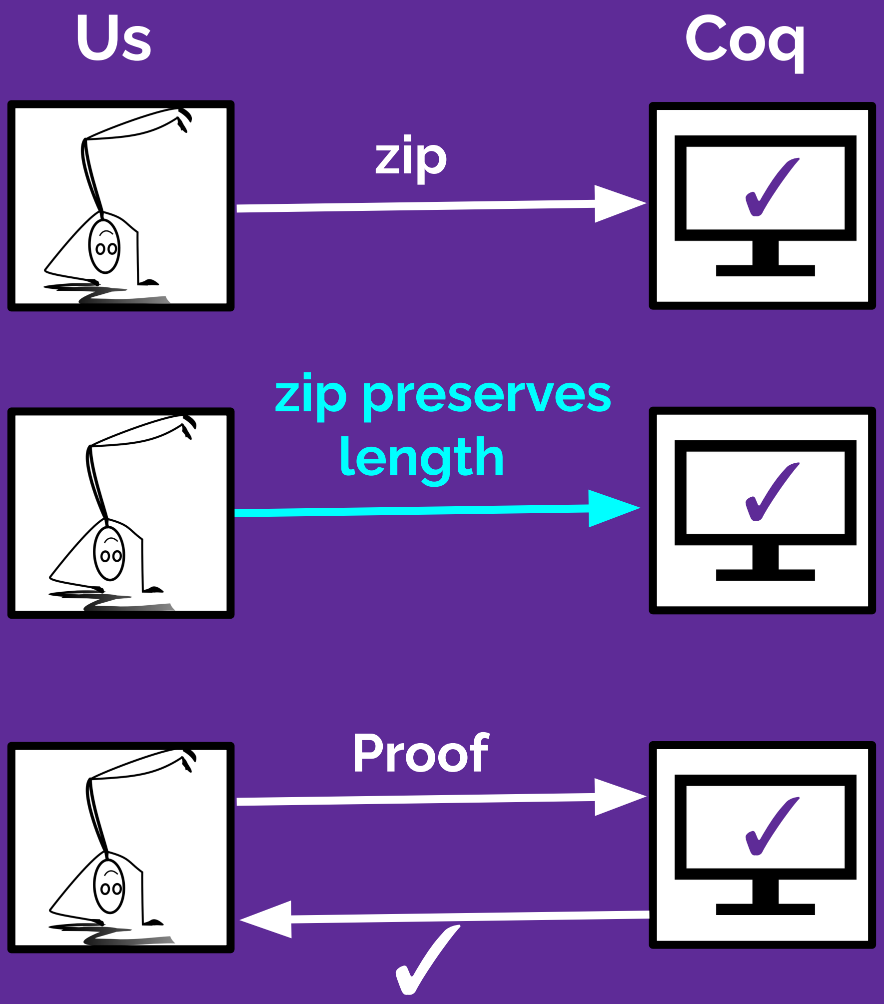
Modified from hs-to-coq

```
Fixpoint zip {A B} (l1 : list A) (l2 : list B) : list (A * B) :=
  match l1, l2 with
  | [ ], _ -> [ ]
  | _, [ ] -> [ ]
  | h1 :: tl1, h2 :: tl2 -> (h1, h2) :: (zip tl1 tl2)
  end.
```



Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Coq

zip

zip preserves length

Proof

✓

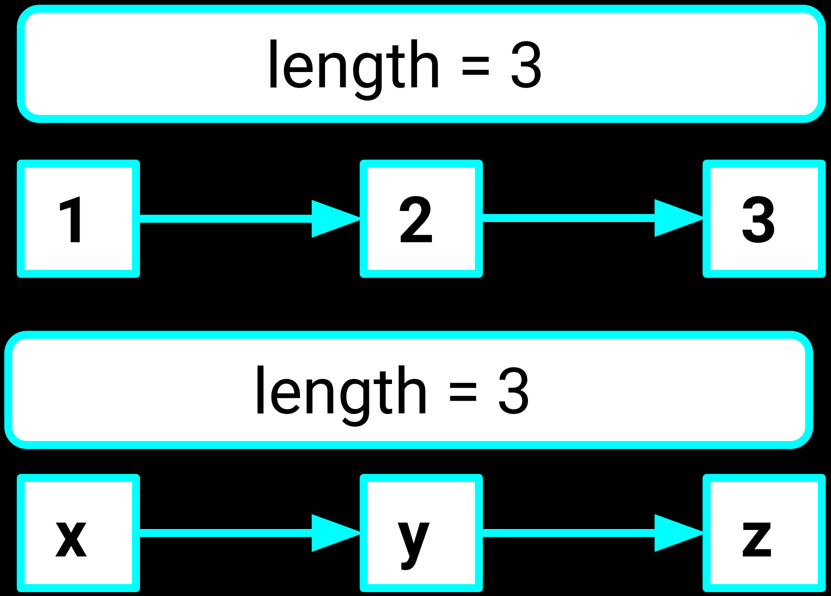
Proof Assistants (Part 1 of 5)

List Zip Preserves Length

```
Theorem zip_preserves_length :
  ∀ {A B} (l1 : list A) (l2 : list B),
   length l1 = length l2 →
   length (zip l1 l2) = length l1.
```

List Zip Preserves Length

```
Theorem zip_preserves_length :
  ∀ {A B} (l1 : list A) (l2 : list B),
   length l1 = length l2 →
   length (zip l1 l2) = length l1.
```



length = 3

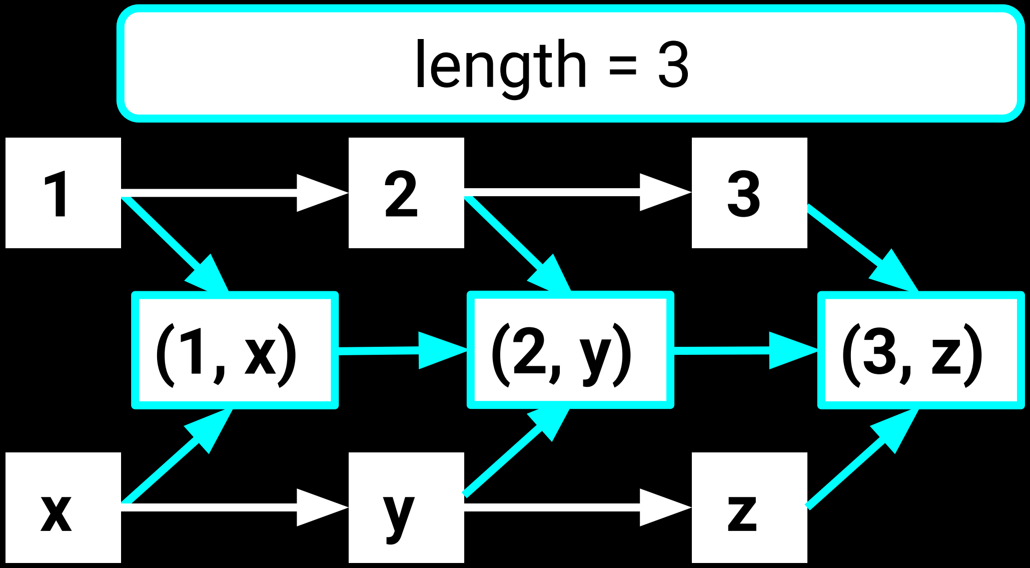
1 → 2 → 3

length = 3

x → y → z

List Zip Preserves Length

```
Theorem zip_preserves_length :
  ∀ {A B} (l1 : list A) (l2 : list B),
   length l1  = length l2 →
   length (zip l1 l2) = length l1.
```



length = 3

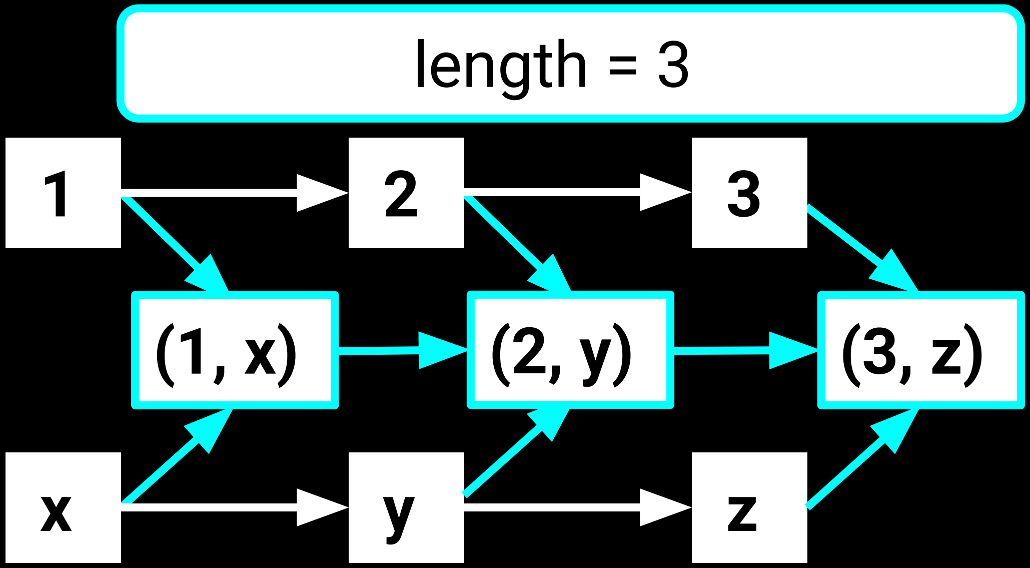
1 2 3

(1, x) (2, y) (3, z)

x y z

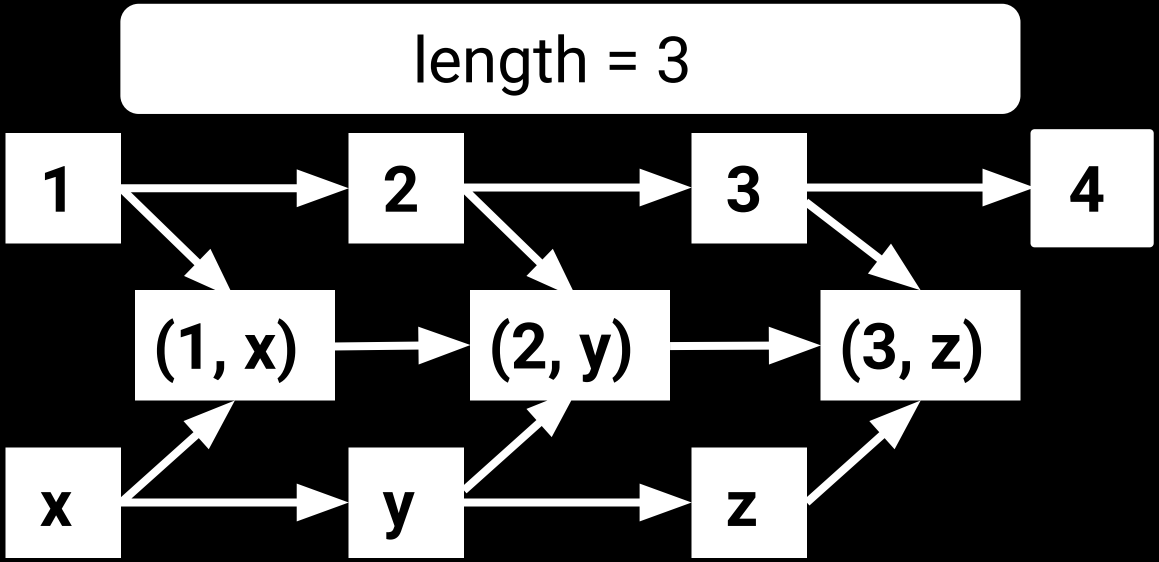
List Zip Preserves Length

```
Theorem zip_preserves_length :
  ∀ {A B} (l1 : list A) (l2 : list B),
   length l1 = length l2 →
   length (zip l1 l2) = length l1.
```



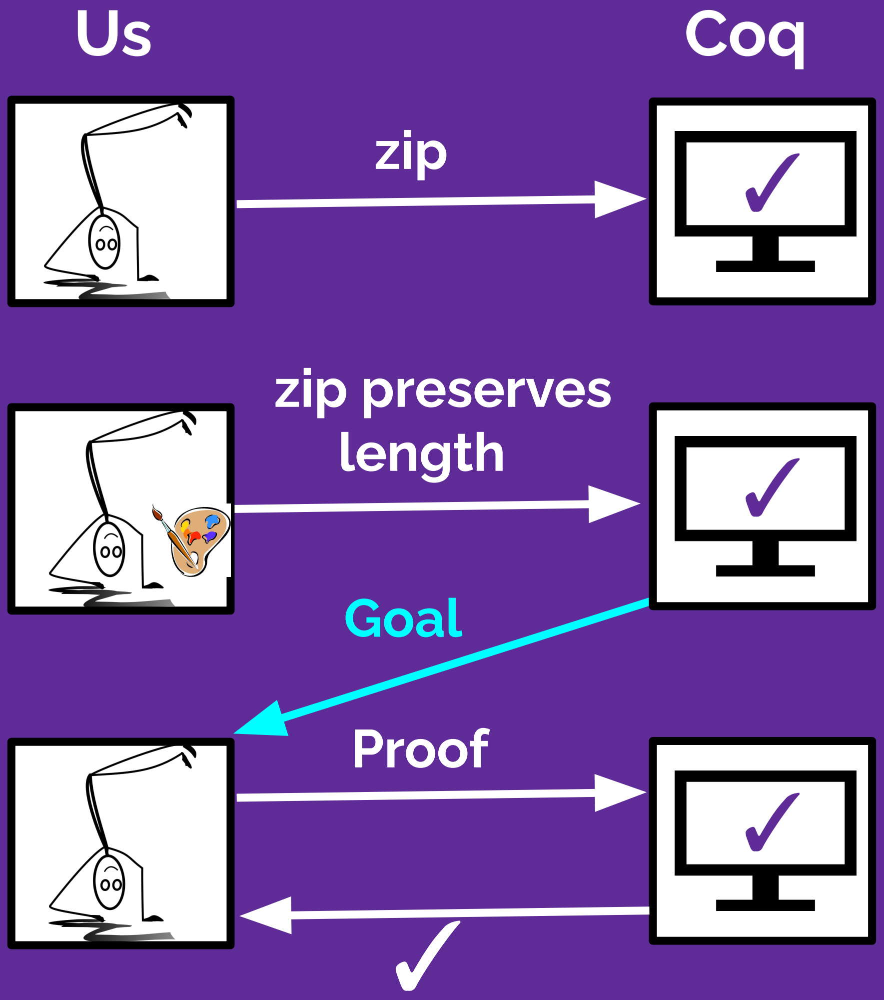
List Zip Preserves Length

```
Theorem zip_preserves_length :
  ∀ {A B} (l1 : list A) (l2 : list B),
   length (zip l1 l2) = min (length l1) (length l2).
```



Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Us

Coq

zip

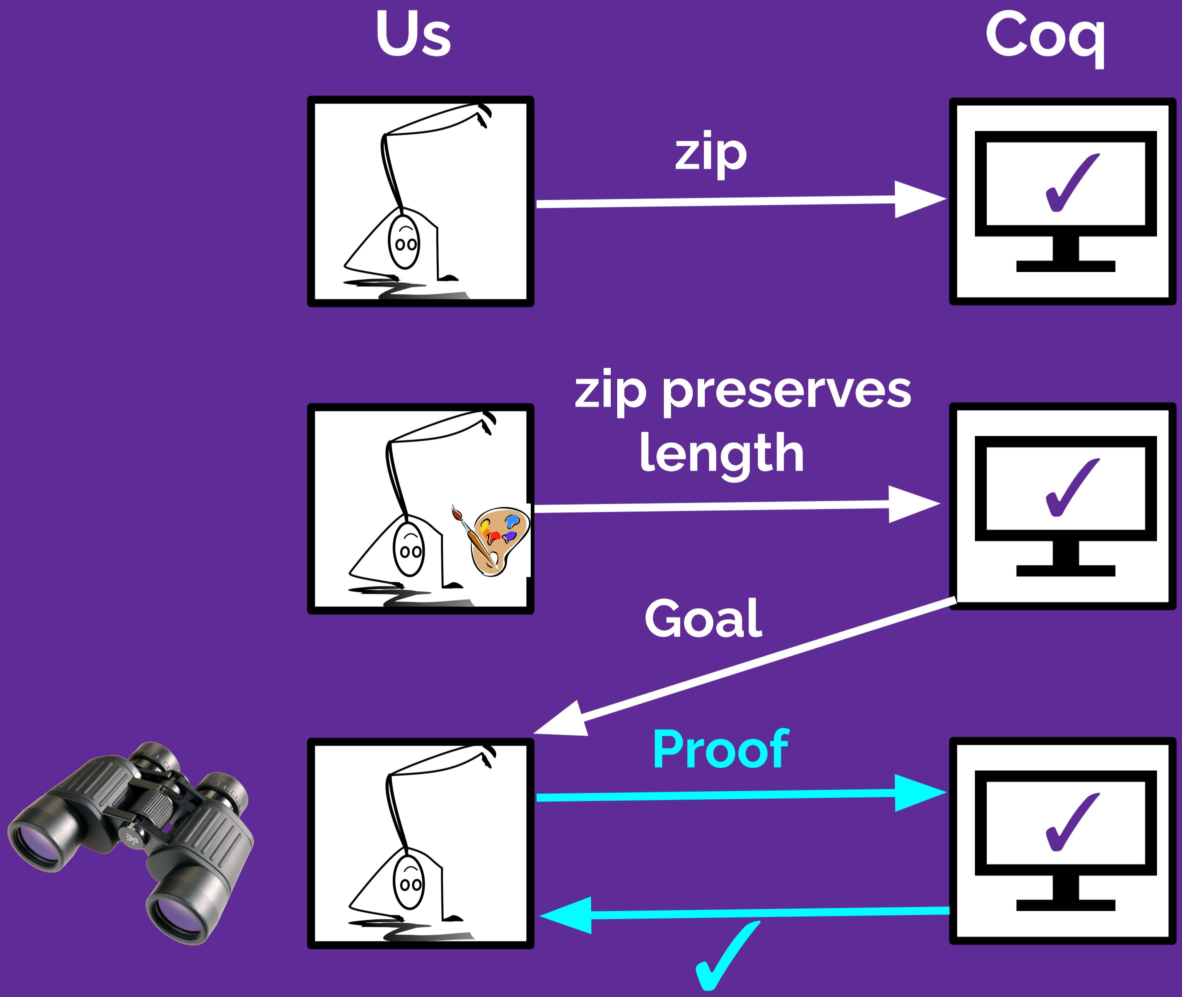
zip preserves length

Goal

Proof

✓

List Zip Preserves Length



Us

Coq

zip

zip preserves length

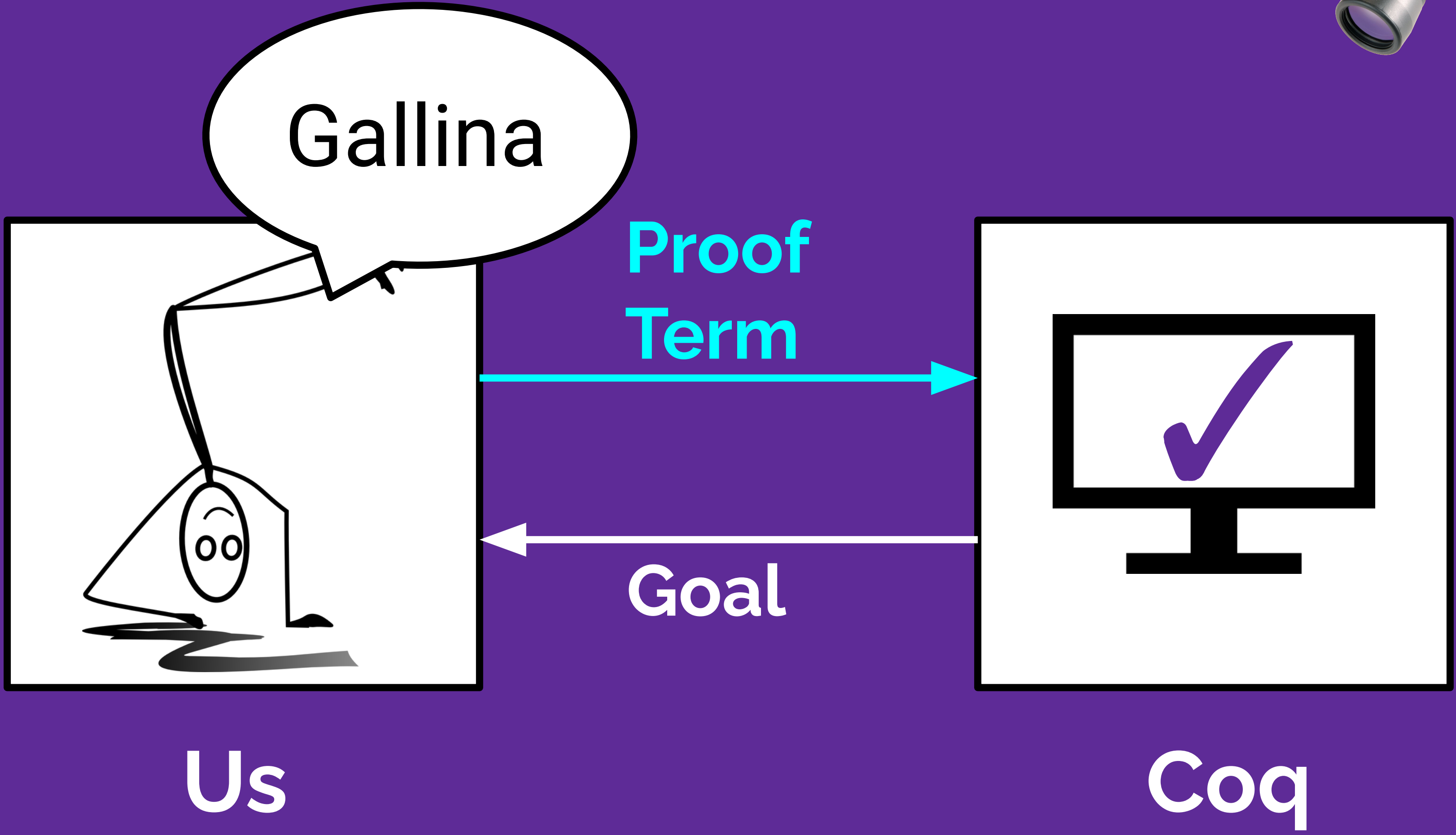
Goal

Proof

✓

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Gallina

Proof Term

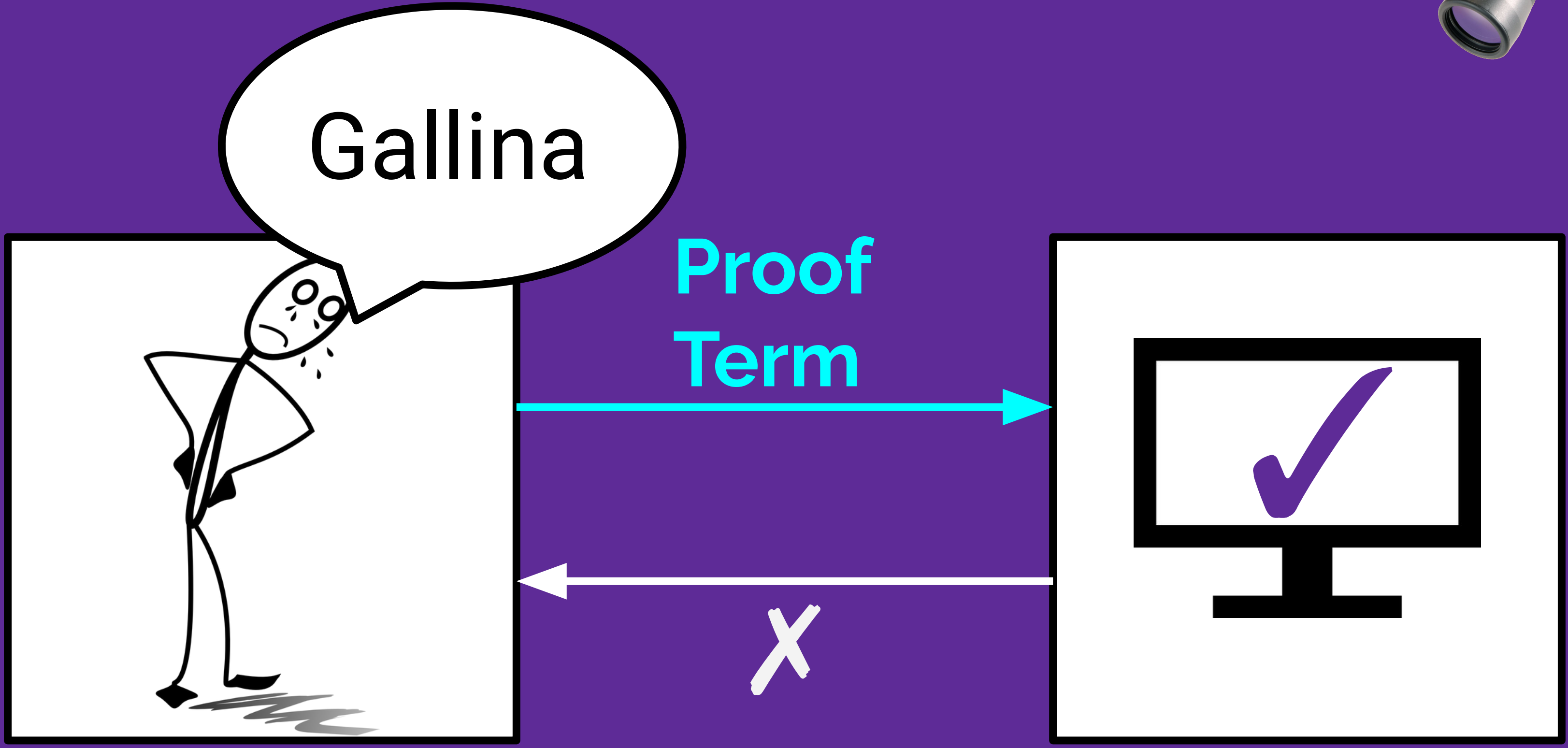
Goal

Us

Coq

Proof Assistants (Part 1 of 5)

List Zip Preserves Length



Gallina

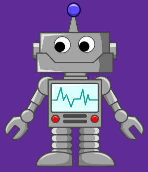
Proof Term

✗

Us

Coq

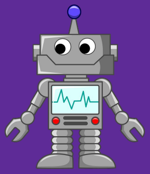
1. Proof Assistants
2. Symbolic Automation
3. Neural Automation
4. Building Bridges
5. Opportunities



Proof Automation

(The kind you're used to.)

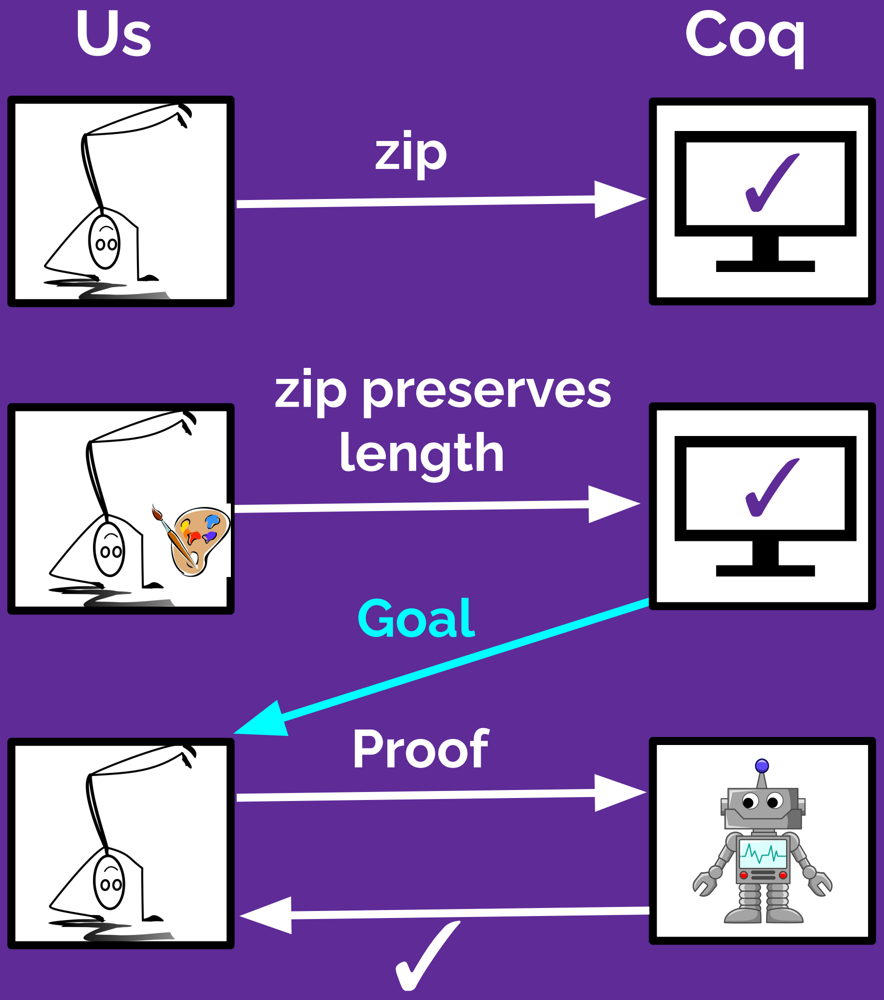
Symbolic Automation (Part 2 of 5)



Example: Tactics

Symbolic Automation (Part 2 of 5)

Example: Tactics



Us

Coq

zip

zip preserves length

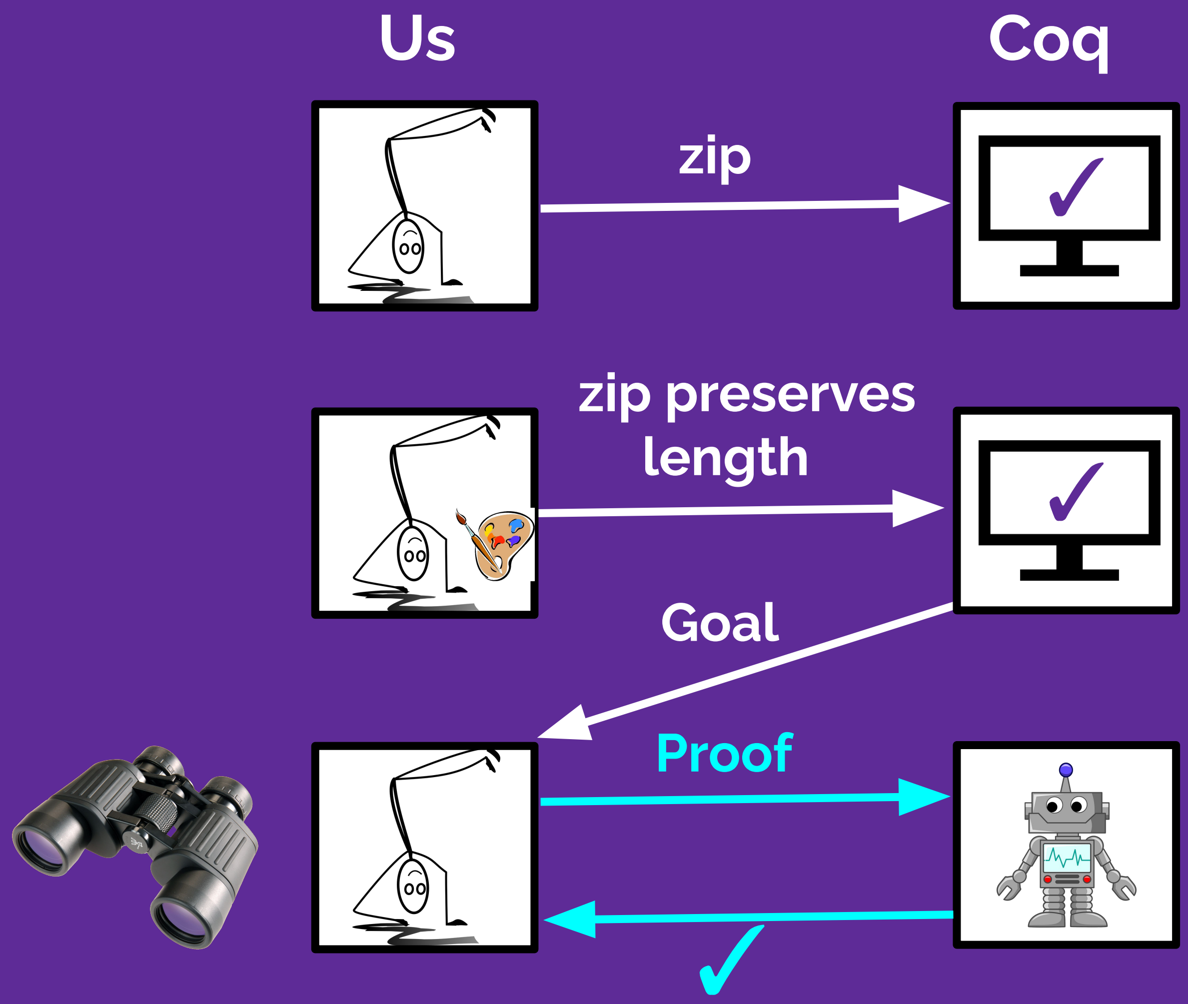
Goal

Proof

✓

Symbolic Automation (Part 2 of 5)

Example: Tactics



Us

Coq

zip

zip preserves length

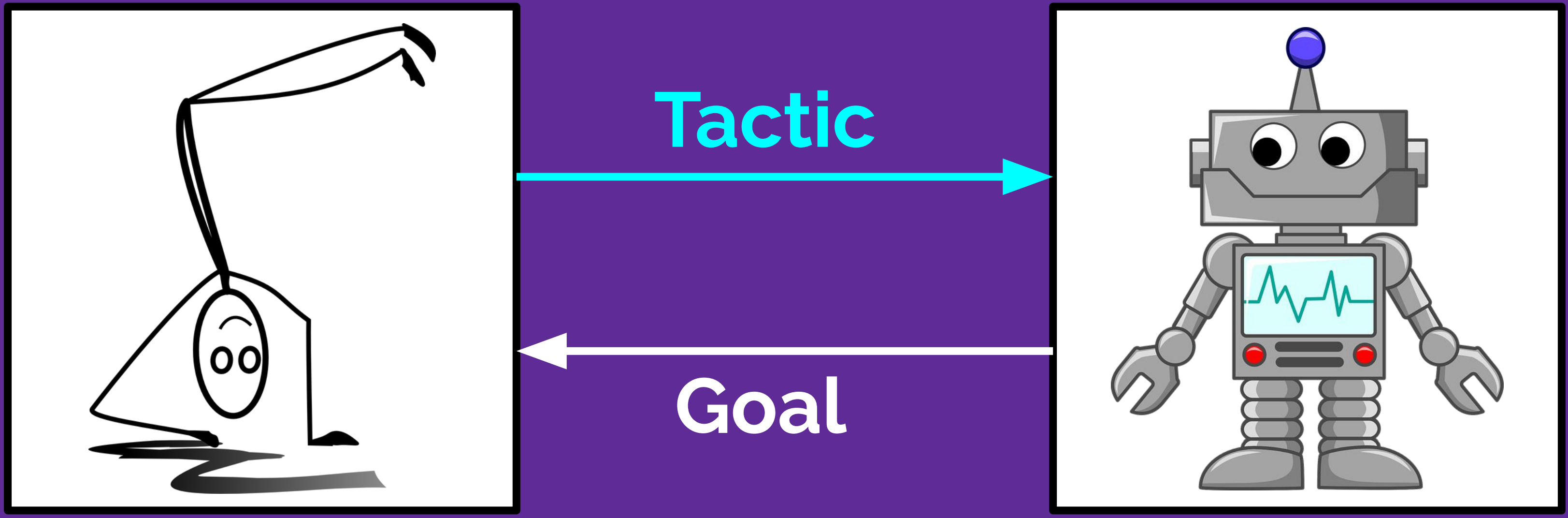
Goal

Proof

✓

Symbolic Automation (Part 2 of 5)

Example: Tactics



Tactic

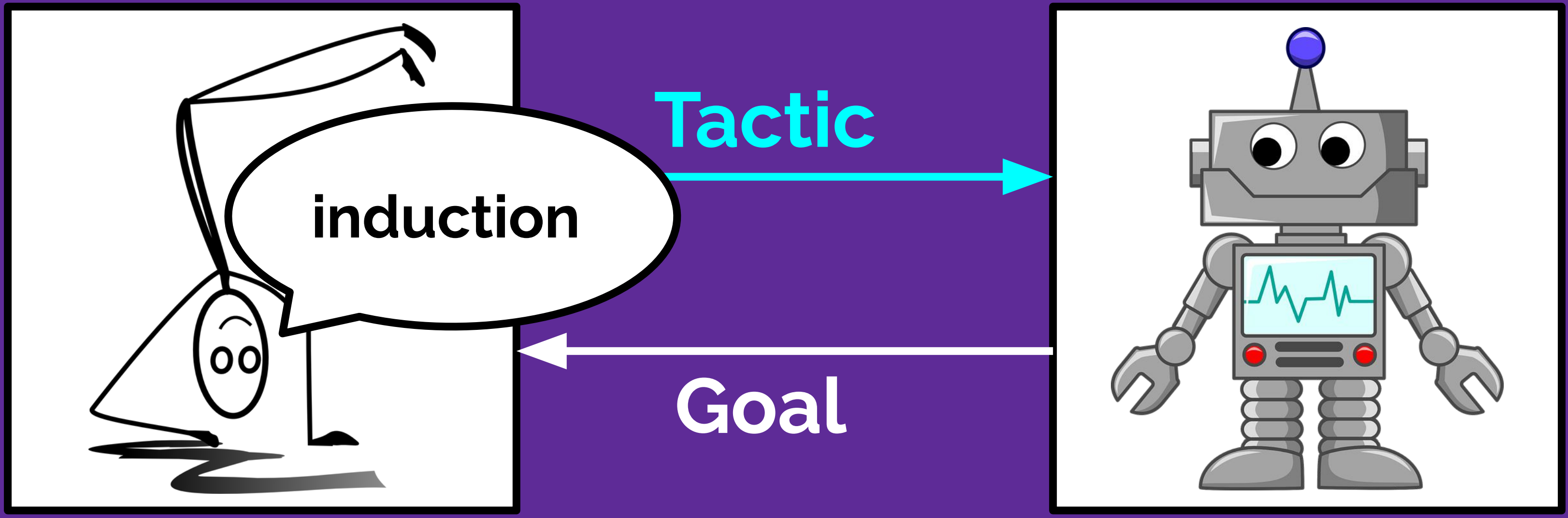
Goal

Us

Coq

Symbolic Automation (Part 2 of 5)

Example: Tactics



Tactic

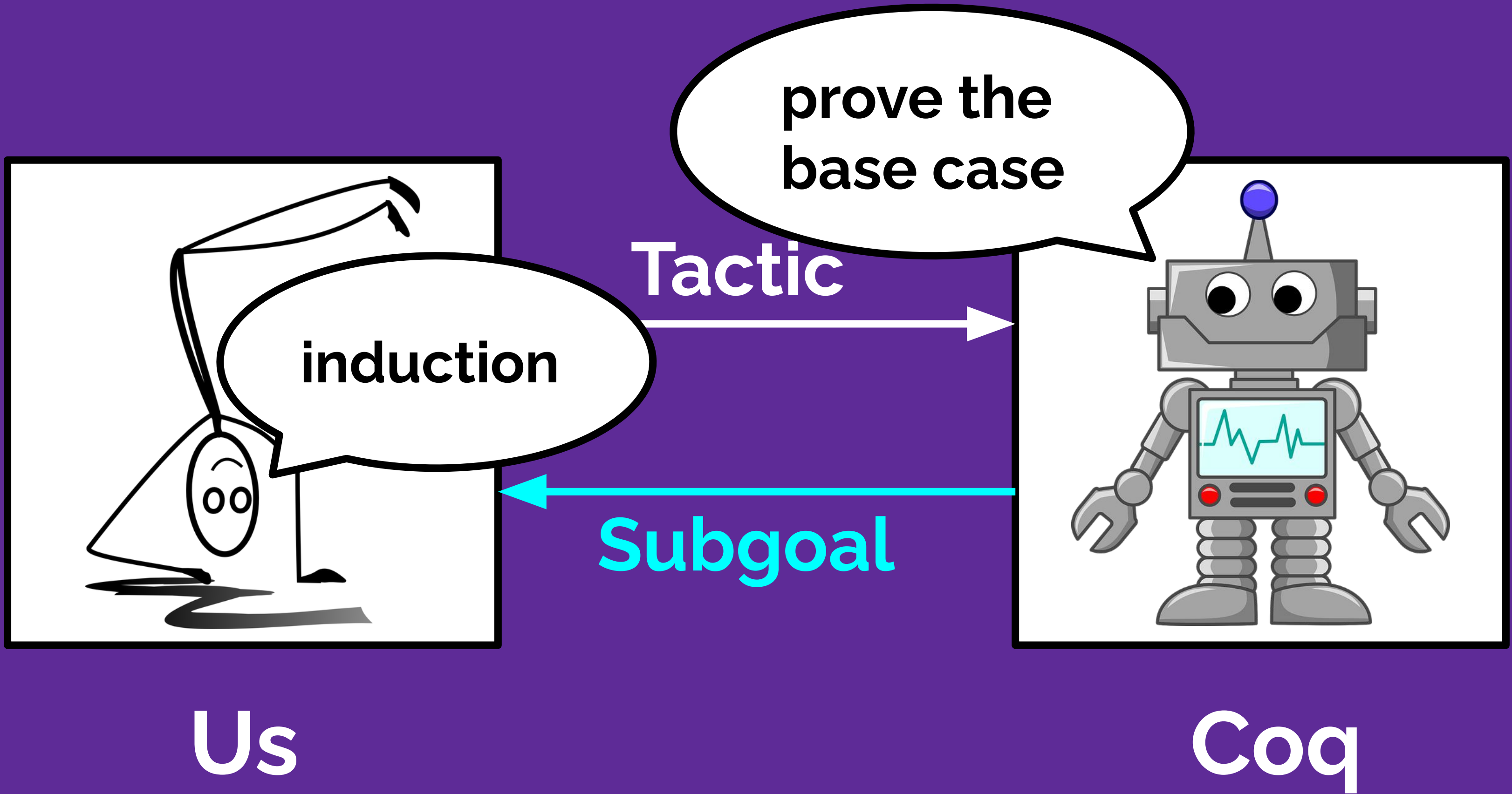
Goal

Us

Coq

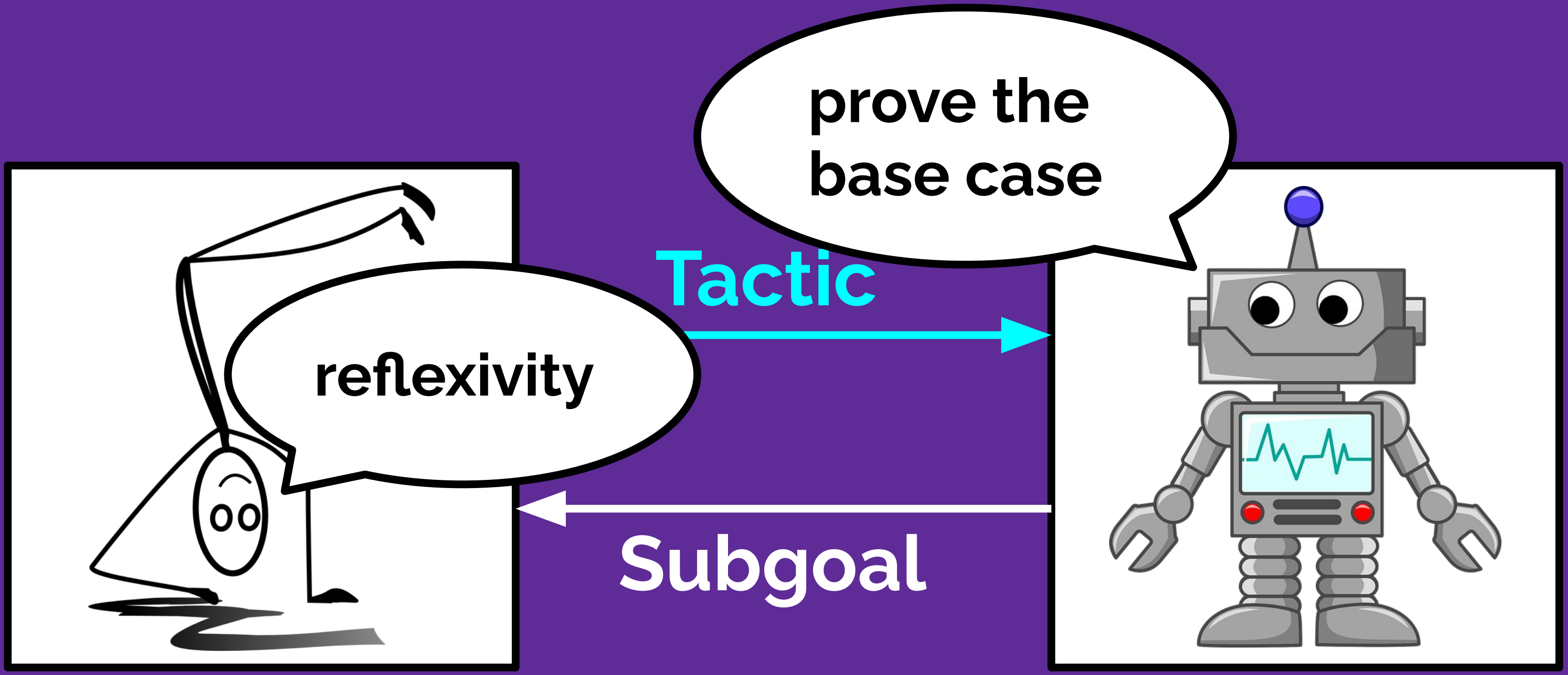
Symbolic Automation (Part 2 of 5)

Example: Tactics



Symbolic Automation (Part 2 of 5)

Example: Tactics



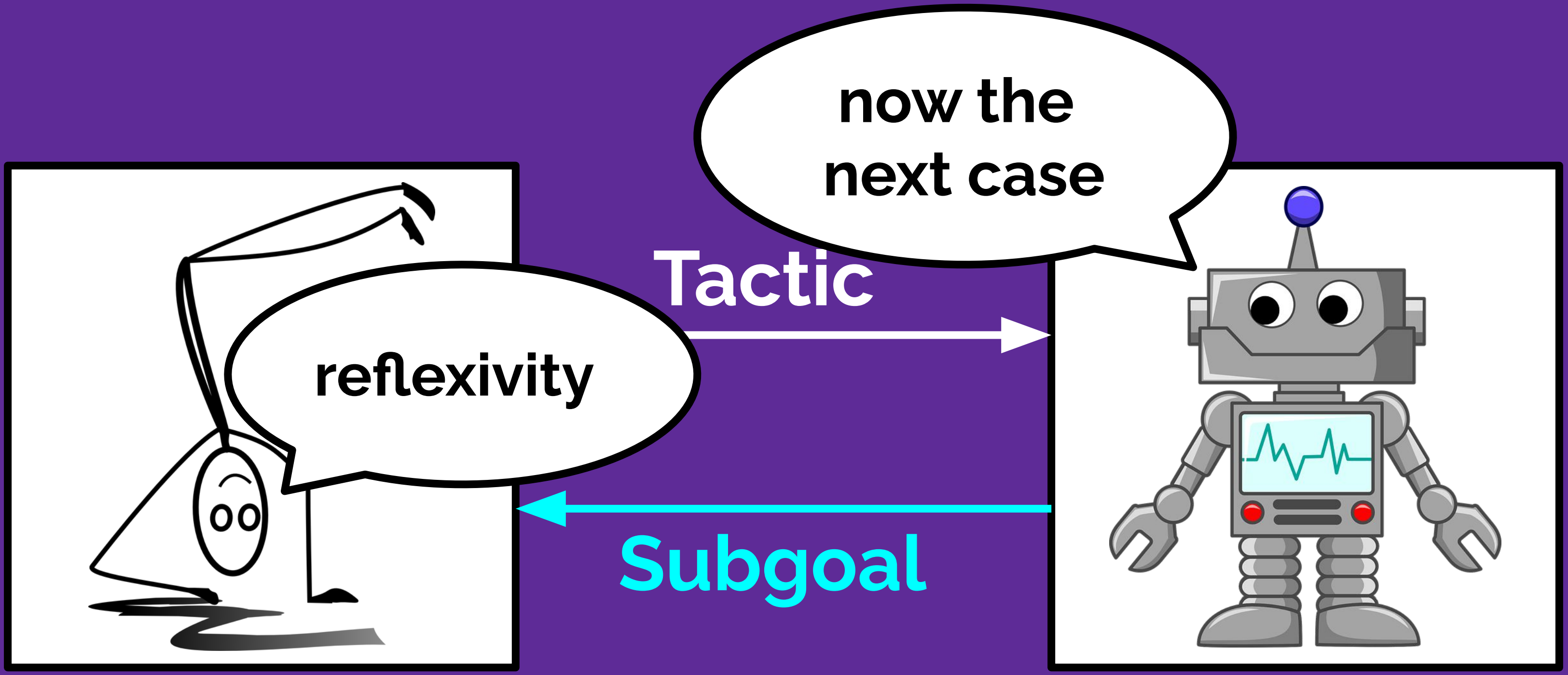
Tactic

Subgoal

Us

Coq

Example: Tactics



Tactic

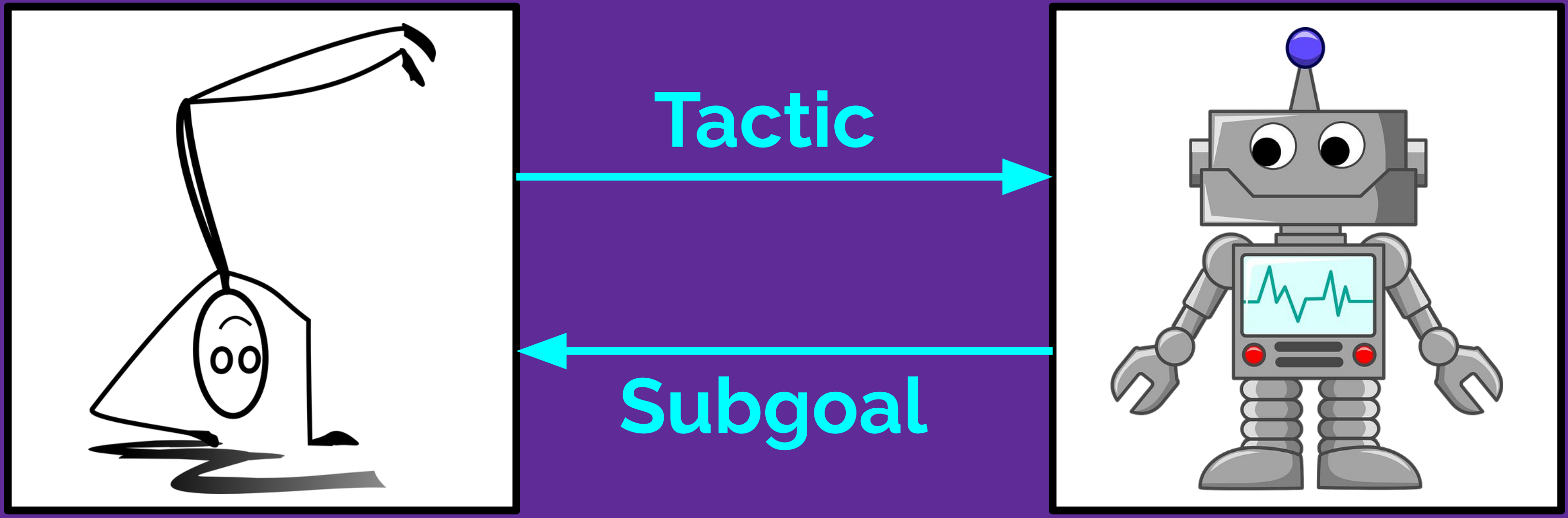
Subgoal

Us

Coq

Symbolic Automation (Part 2 of 5)

Example: Tactics



Tactic

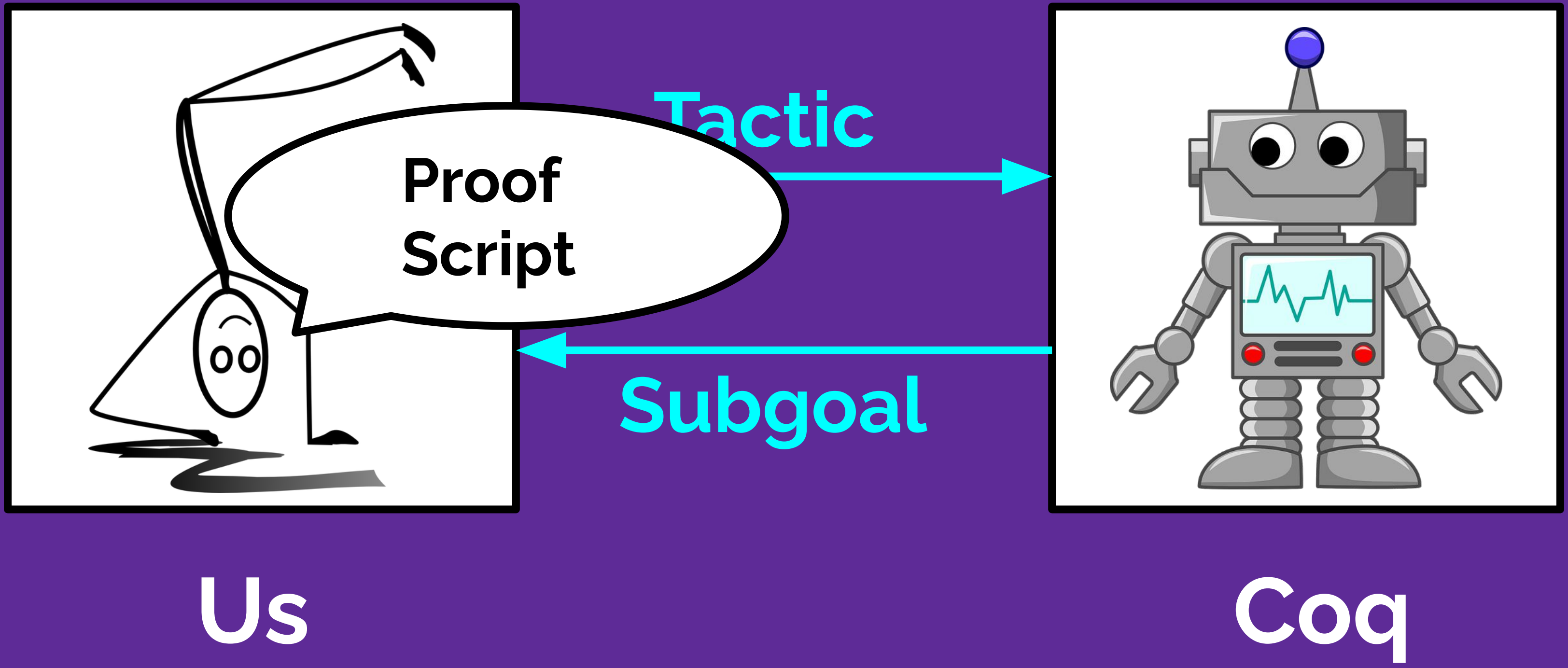
Subgoal

Us

Coq

Symbolic Automation (Part 2 of 5)

Example: Tactics



Proof Script

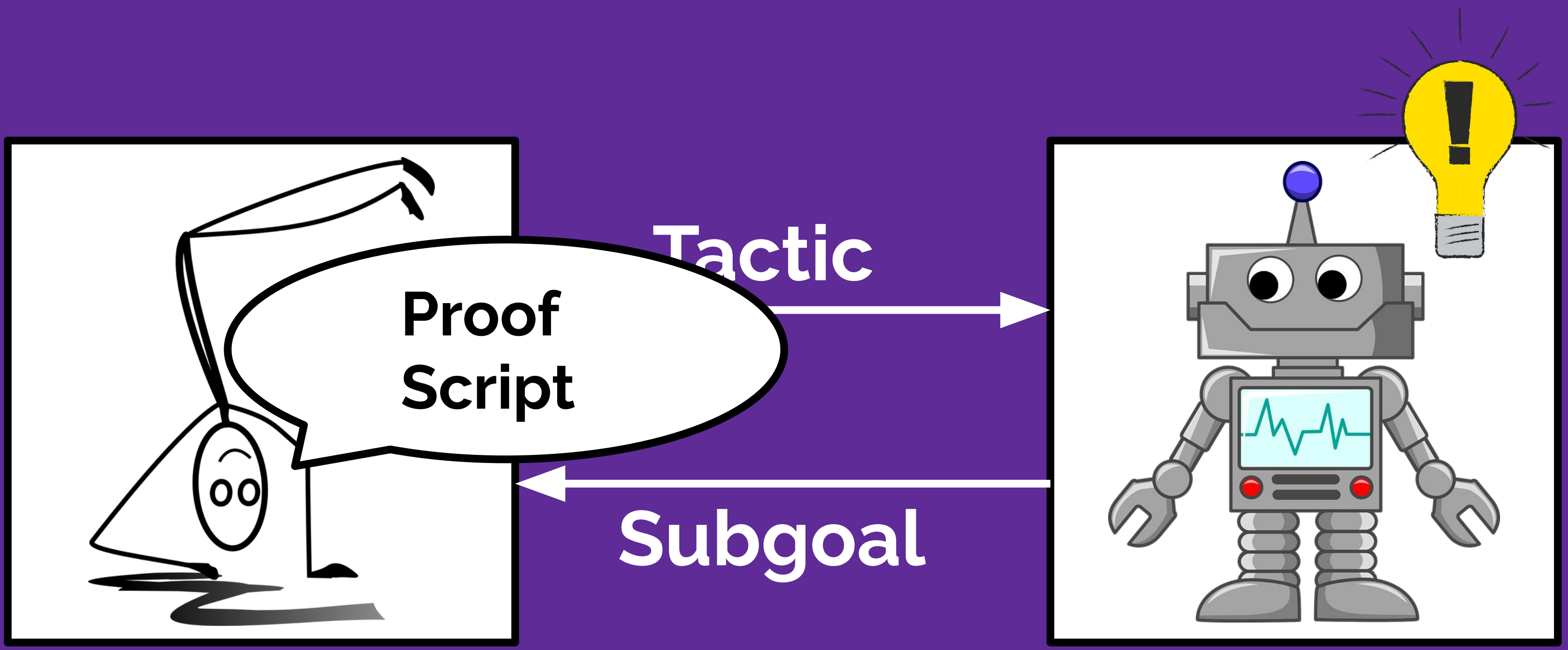
Tactic

Subgoal

Us

Coq

Example: Tactics

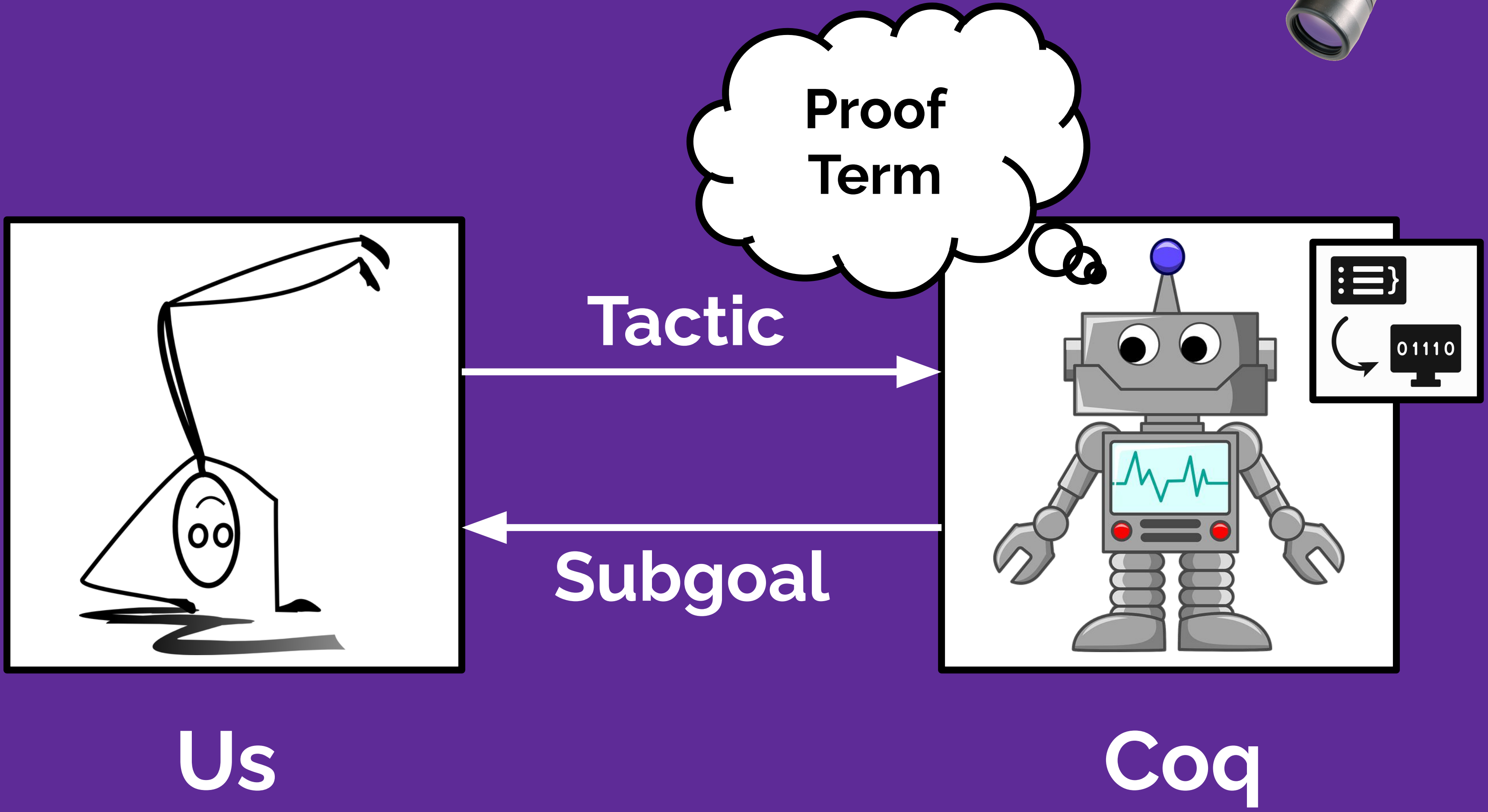


Us

Coq

Symbolic Automation (Part 2 of 5)

Example: Tactics



Proof Term

Tactic

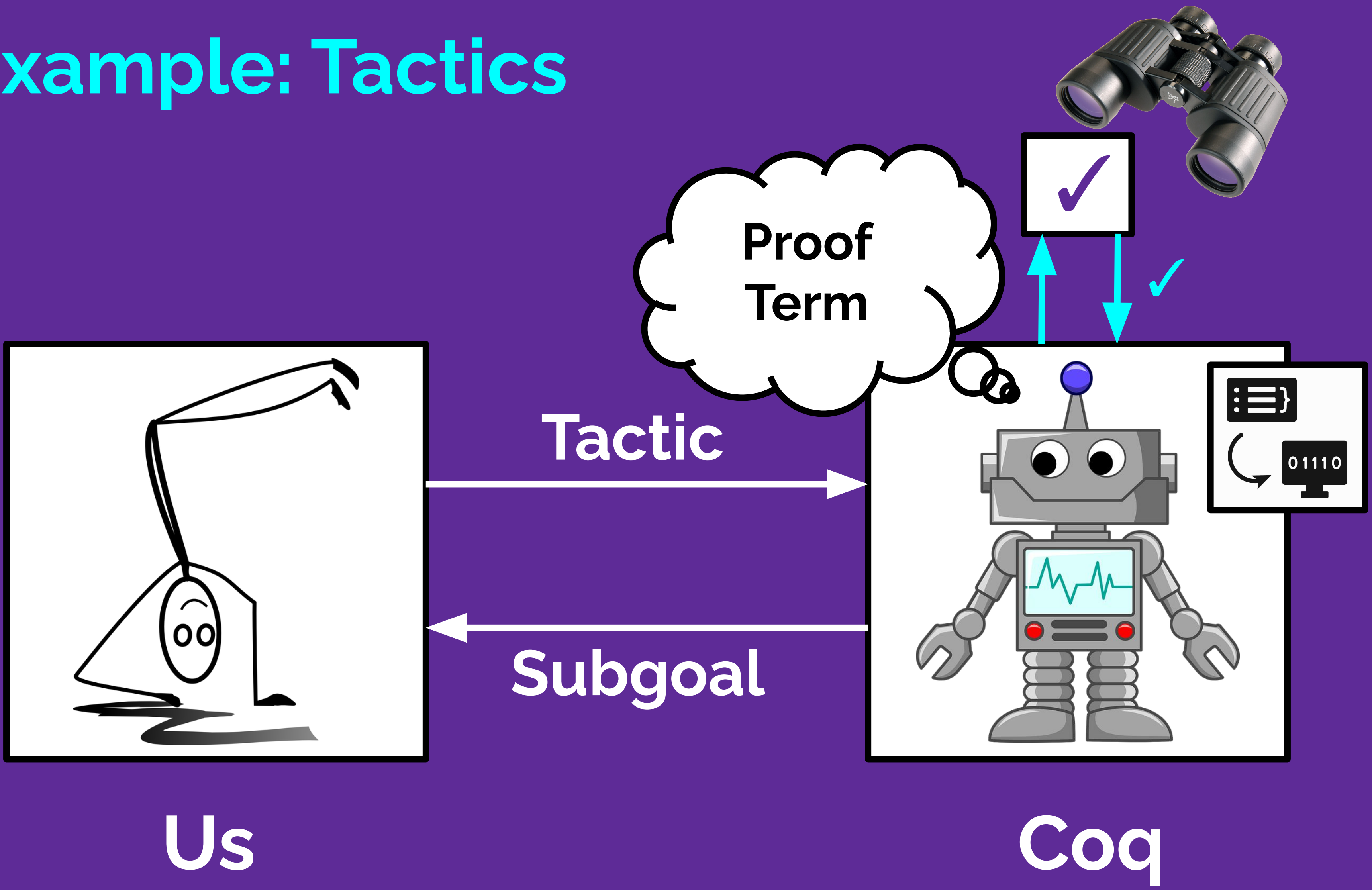
Subgoal

Us

Coq

Symbolic Automation (Part 2 of 5)

Example: Tactics



Proof Term

✓

✓

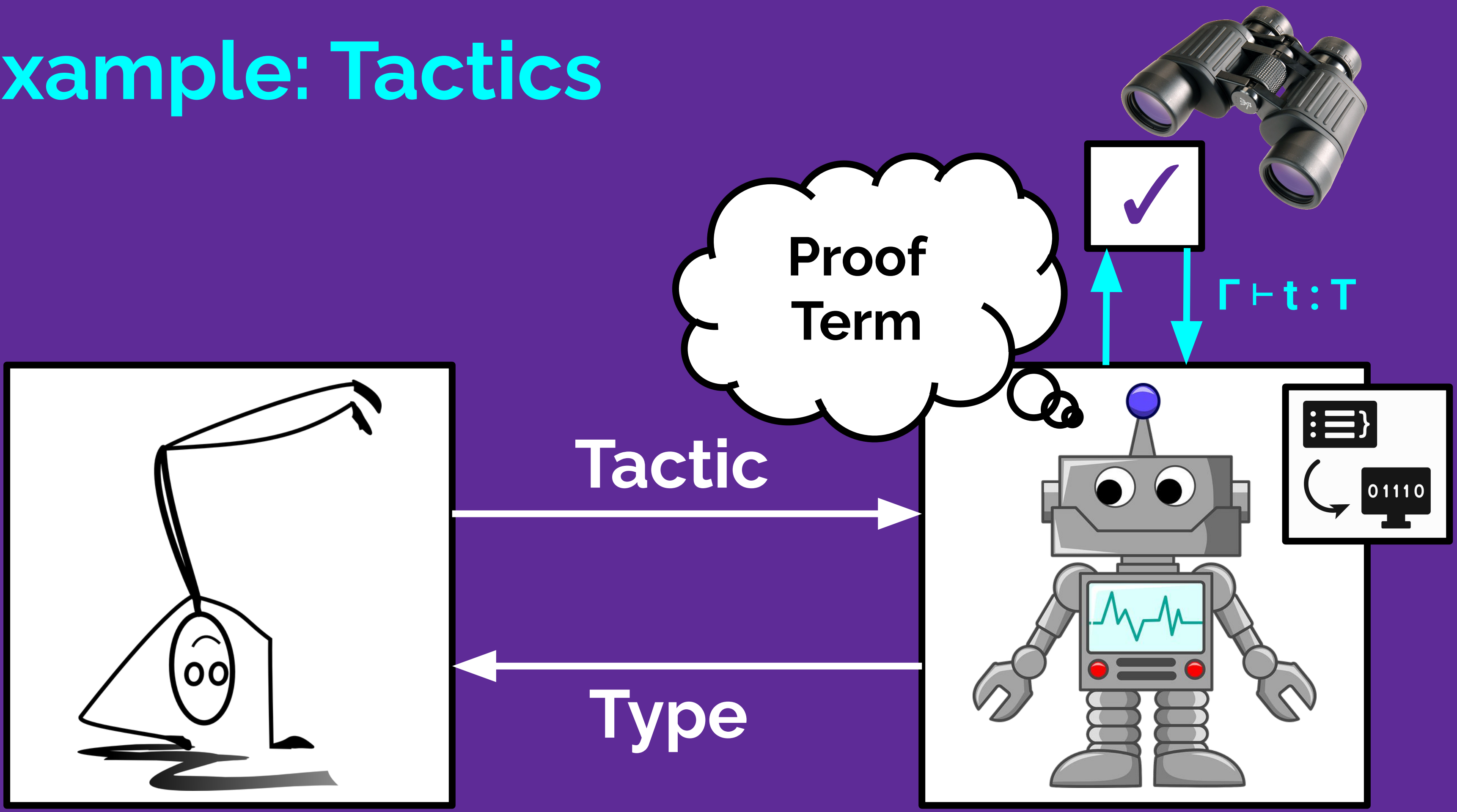
Tactic

Subgoal

Us

Coq

Example: Tactics



Proof Term

✓

Γ ⊢ t : T

Tactic

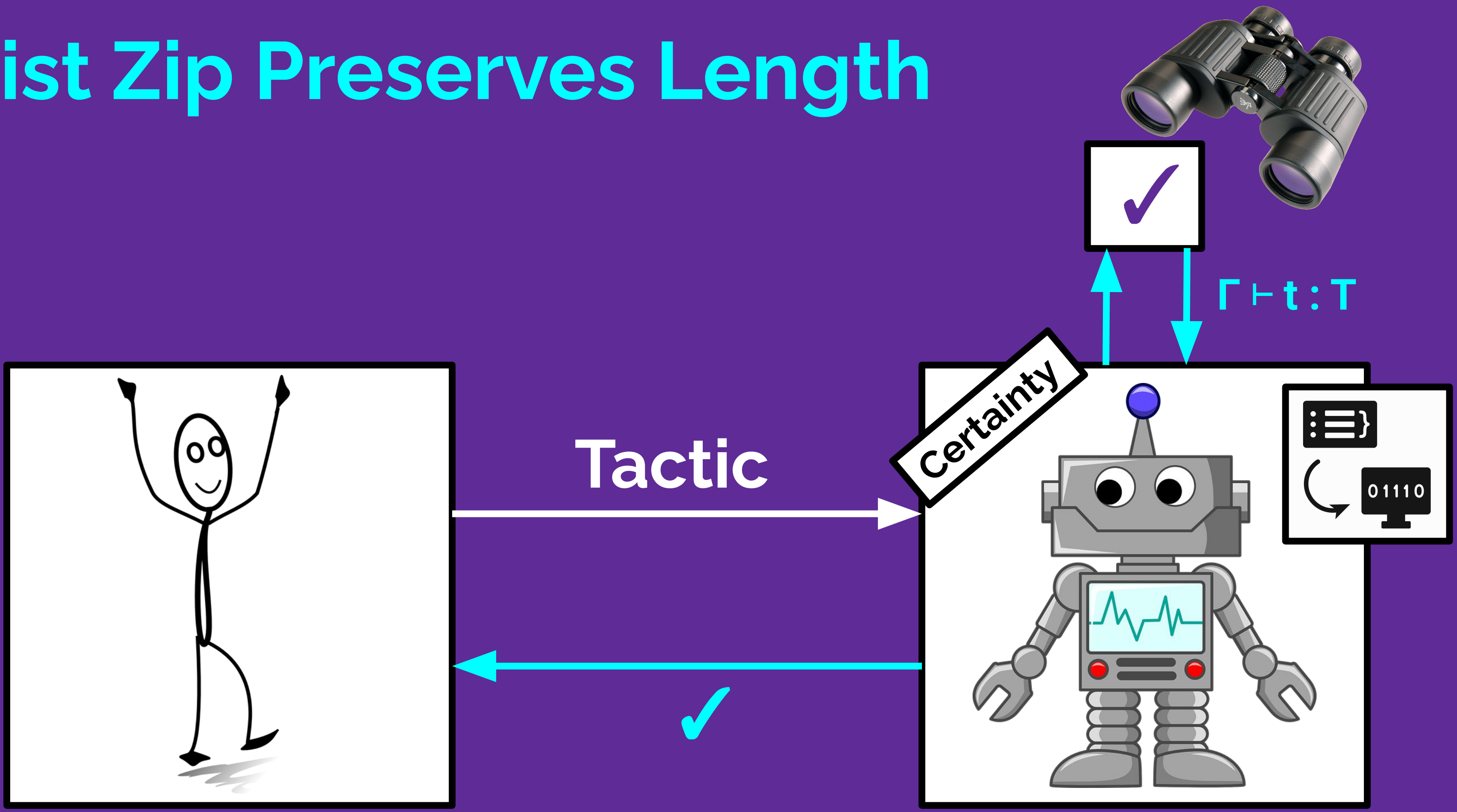
Type

Us

Coq

Symbolic Automation (Part 2 of 5)

List Zip Preserves Length



✓

$\Gamma \vdash t : T$

Tactic

✓

Certainty

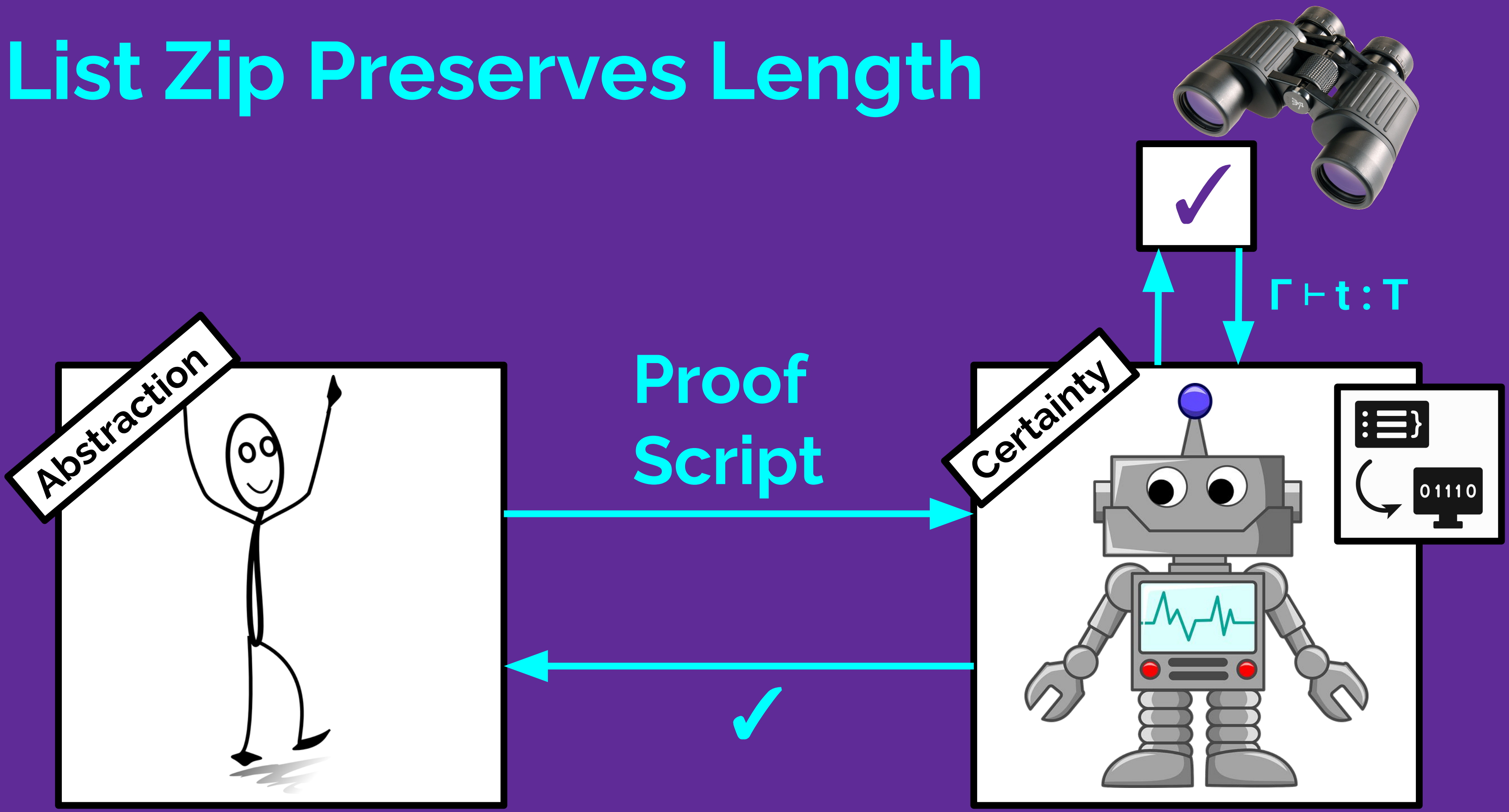
01110

Us

Coq

Symbolic Automation (Part 2 of 5)

List Zip Preserves Length



Γ ⊢ t : T

✓

Proof Script

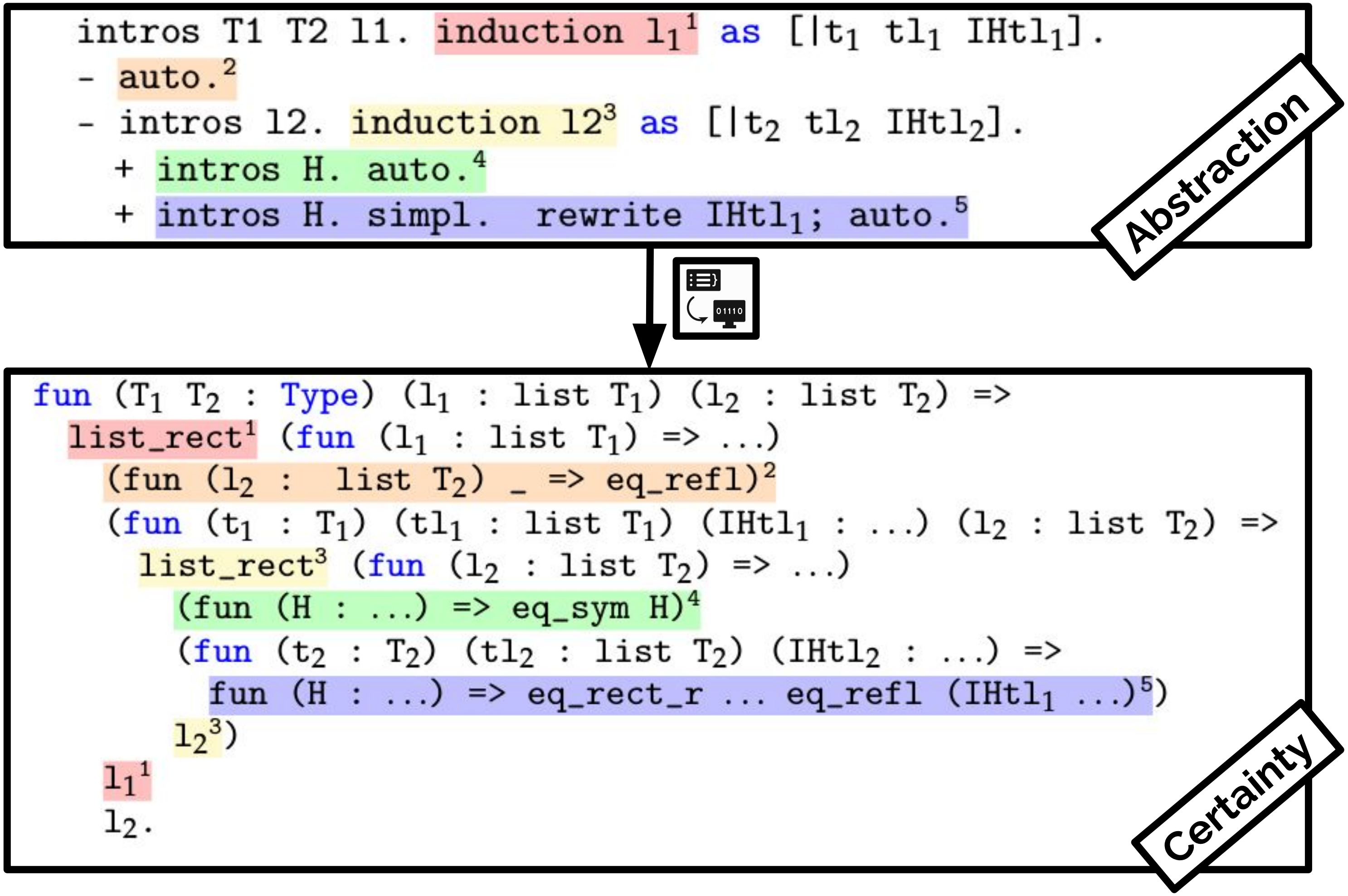
✓

Us

Coq

Symbolic Automation (Part 2 of 5)

List Zip Preserves Length

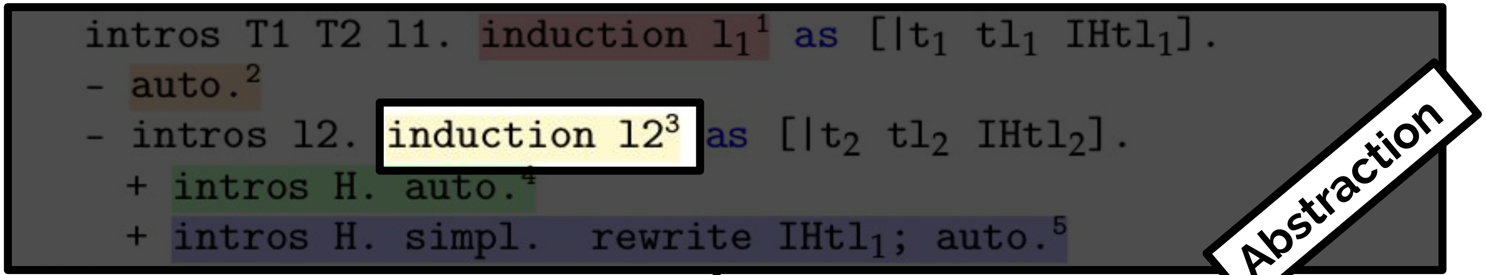


```
intros T1 T2 l1. induction l1^1 as [|t1 tl1 IHtl1].
- auto.^2
- intros l2. induction l2^3 as [|t2 tl2 IHtl2].
  + intros H. auto.^4
  + intros H. simpl.  rewrite IHtl1; auto.^5
```

```
fun (T1 T2 : Type) (l1 : list T1) (l2 : list T2) =>
  list_rect^1 (fun (l1 : list T1) => ...)
   (fun (l2 :  list T2) _ => eq_refl)^2
   (fun (t1 : T1) (tl1 : list T1) (IHtl1 : ...) (l2 : list T2) =>
    list_rect^3 (fun (l2 : list T2) => ...)
     (fun (H : ...) => eq_sym H)^4
     (fun (t2 : T2) (tl2 : list T2) (IHtl2 : ...) =>
      fun (H : ...) => eq_rect_r ... eq_refl (IHtl1 ...)^5)
     l2^3)
   l1^1
   l2.
```

Symbolic Automation (Part 2 of 5)

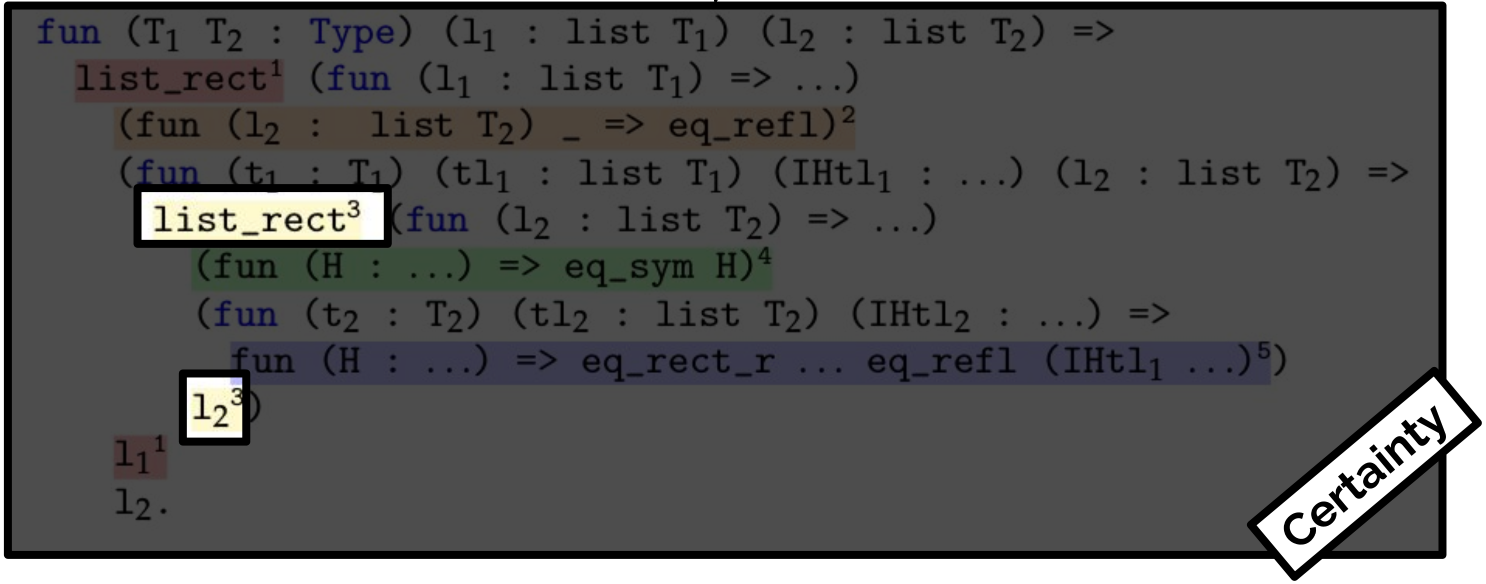
List Zip Preserves Length



```
intros T1 T2 l1. induction l1[1] as [|t1 tl1 IHtl1].
- auto.[2]
- intros l2. induction l2[3] as [|t2 tl2 IHtl2].
  + intros H. auto.[4]
  + intros H. simpl. rewrite IHtl1; auto.[5]
```

Abstraction

Induction => Induction Principles

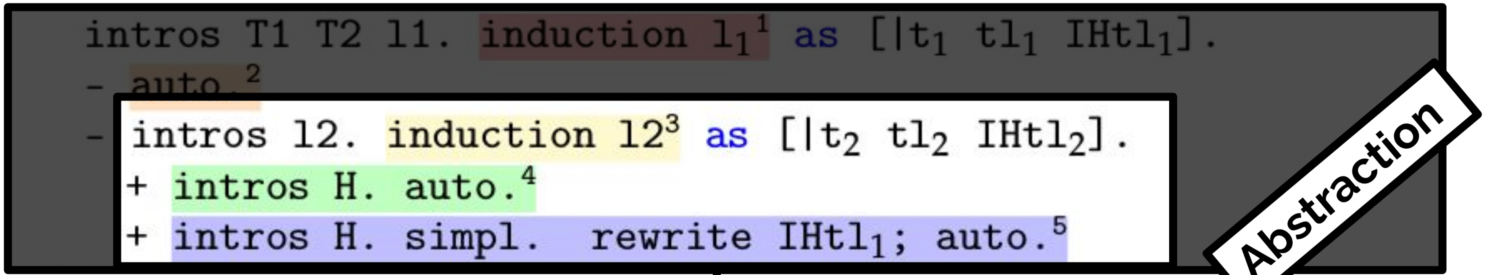


```
fun (T1 T2 : Type) (l1 : list T1) (l2 : list T2) =>
  list_rect[1] (fun (l1 : list T1) => ...)
    (fun (l2 : list T2) _ => eq_refl)[2]
    (fun (t1 : T1) (tl1 : list T1) (IHtl1 : ...) (l2 : list T2) =>
     list_rect[3] (fun (l2 : list T2) => ...)
       (fun (H : ...) => eq_sym H)[4]
       (fun (t2 : T2) (tl2 : list T2) (IHtl2 : ...) =>
         fun (H : ...) => eq_rect_r ... eq_refl (IHtl1 ...)[5])
      l2[3])
    l1[1]
    l2.
```

Certainty

Symbolic Automation (Part 2 of 5)

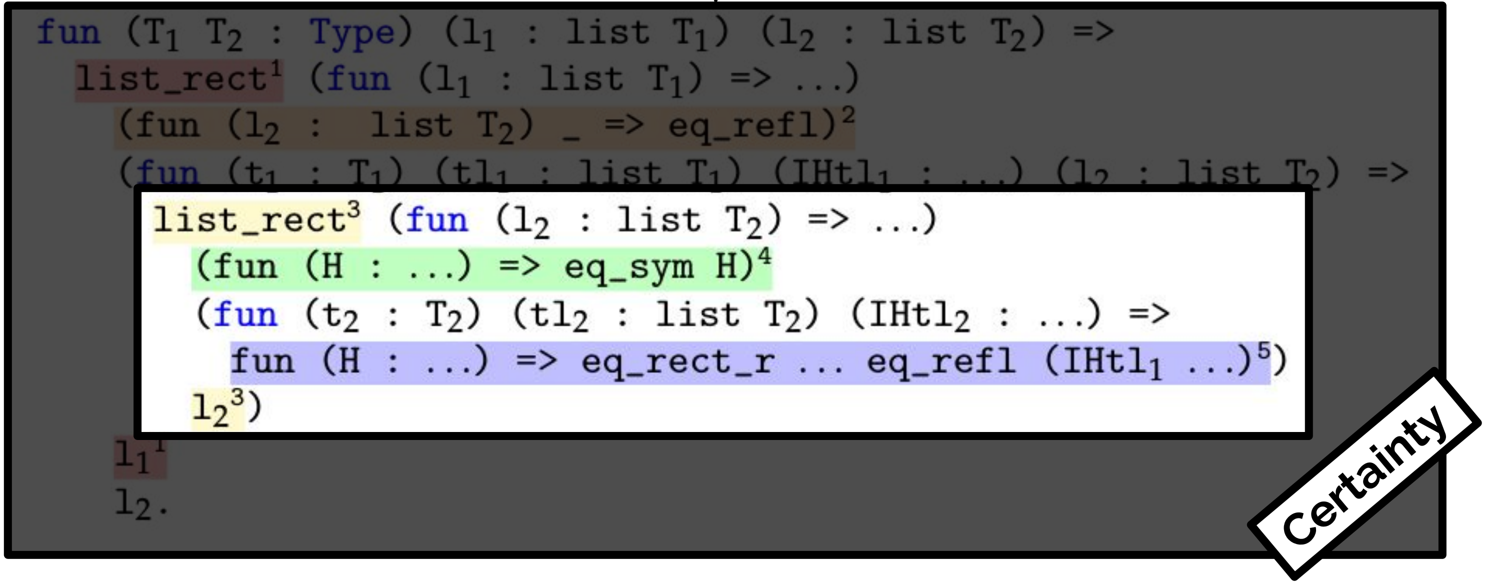
List Zip Preserves Length



```
intros T1 T2 l1. induction l₁¹ as [|t₁ tl₁ IHtl₁].
 - auto.²
 - intros l2. induction l2³ as [|t₂ tl₂ IHtl₂].
   + intros H. auto.⁴
   + intros H. simpl.  rewrite IHtl₁; auto.⁵
```

Abstraction

Induction => Induction Principles



```
fun (T₁ T₂ : Type) (l₁ : list T₁) (l₂ : list T₂) =>
  list_rect¹ (fun (l₁ : list T₁) => ...)
   (fun (l₂ :  list T₂) _ => eq_refl)²
   (fun (t₁ : T₁) (tl₁ : list T₁) (IHtl₁ : ...) (l₂ : list T₂) =>
    list_rect³ (fun (l₂ : list T₂) => ...)
      (fun (H : ...) => eq_sym H)⁴
      (fun (t₂ : T₂) (tl₂ : list T₂) (IHtl₂ : ...) =>
        fun (H : ...) => eq_rect_r ... eq_refl (IHtl₁ ...)⁵)
      l₂³)
   l₁¹
   l₂.
```

Certainty

Symbolic Automation (Part 2 of 5)

Kinds of Automation

Tactics
Reflection
Custom proof modes
Proof procedures
Plugins
Proof repair
Hammers

Symbolic Automation (Part 2 of 5)

Kinds of Automation

Tactics
Reflection
Custom proof modes
Proof procedures
Plugins
Proof repair
Hammers

Symbolic Automation (Part 2 of 5)

This automation can do basically anything, yet still **preserve correctness**.

De Bruijn Criterion

Symbolic Automation (Part 2 of 5)

Checking the Proof

Producing the Proof

Symbolic Automation (Part 2 of 5)

Checking the Proof

Producing the Proof

Symbolic Automation (Part 2 of 5)

Checking the Proof

Tactics

Producing the Proof

Symbolic Automation (Part 2 of 5)

Checking the Proof

Tactics

Domain-Specific Heuristics

Producing the Proof

Symbolic Automation (Part 2 of 5)

Checking the Proof

Tactics

Domain-Specific Heuristics

Proof Transformations

Producing the Proof

Symbolic Automation (Part 2 of 5)

Checking the Proof

Tactics

Domain-Specific Heuristics

Proof Transformations

Producing the Proof

Scary Programs

Symbolic Automation (Part 2 of 5)

Checking the Proof

Tactics

Domain-Specific Heuristics

Proof Transformations

Producing the Proof

Scary Programs

Symbolic Automation (Part 2 of 5)

Checking the Proof

Small & Human-Readable Logic/Type Checker

Tactics

Domain-Specific Heuristics

Proof Transformations

Producing the Proof

Scary Programs

Symbolic Automation (Part 2 of 5)

Checking the Proof

Small Logical Kernel

Tactics

Domain-Specific Heuristics

Proof Transformations

Producing the Proof

Scary Programs

Symbolic Automation (Part 2 of 5)

With de Bruijn, as long as you don’t touch the kernel, your automation is safe.

Symbolic Automation (Part 2 of 5)

With de Bruijn, as long as you don't touch the kernel, your automation is safe.*

* If your specification is OK, your kernel has no bugs, and you don't assume contradictory or false axioms.

Symbolic automation:

+ predictable
+ dependable
+ understandable
- limited in scope
- takes expertise to extend

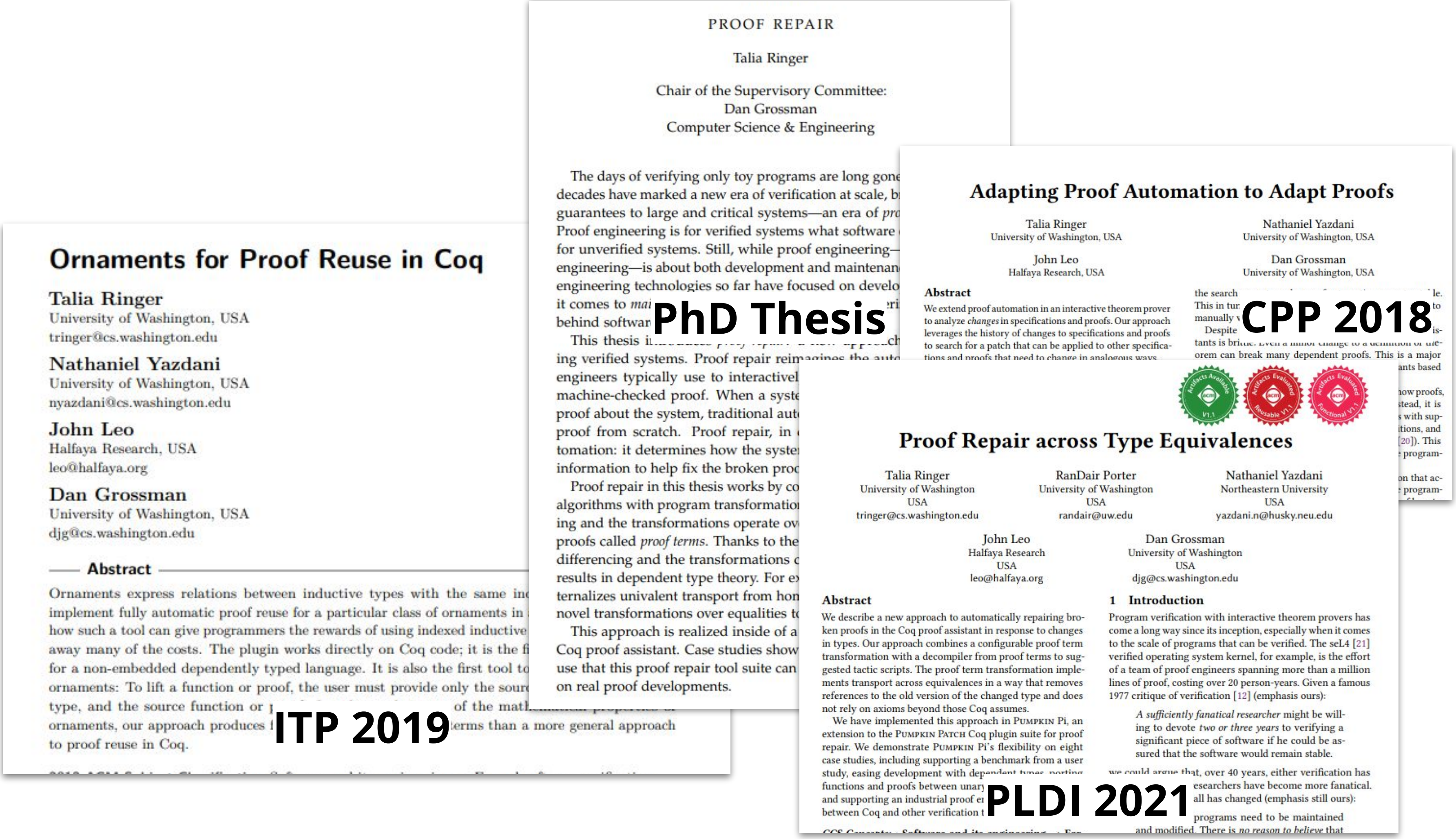
Symbolic Automation (Part 2 of 5)

Symbolic proof repair:

+ predictable
+ dependable
+ understandable
- limited in scope
- takes expertise to extend

Symbolic Automation (Part 2 of 5)

Example: Proof Repair



Ornaments for Proof Reuse in Coq

Talia Ringer
University of Washington, USA
tringer@cs.washington.edu

Nathaniel Yazdani
University of Washington, USA
nyazdani@cs.washington.edu

John Leo
Halfaya Research, USA
leo@halfaya.org

Dan Grossman
University of Washington, USA
djg@cs.washington.edu

Abstract

Ornaments express relations between inductive types with the same in
implement fully automatic proof reuse for a particular class of ornaments in
how such a tool can give programmers the rewards of using indexed inductive
away many of the costs. The plugin works directly on Coq code; it is the fi
for a non-embedded dependently typed language. It is also the first tool to
ornaments: To lift a function or proof, the user must provide only the sourc
type, and the source function or
of the math
ornaments, our approach produces
terms than a more general approach
to proof reuse in Coq.

ITP 2019

PROOF REPAIR

Talia Ringer

Chair of the Supervisory Committee:
Dan Grossman
Computer Science & Engineering

The days of verifying only toy programs are long gone
decades have marked a new era of verification at scale, b
guarantees to large and critical systems—an era of pro
Proof engineering is for verified systems what software
for unverified systems. Still, while proof engineering—
engineering—is about both development and maintenan
engineering technologies so far have focused on develo
it comes to mai
behind softwar
This thesis i
ing verified systems. Proof repair reimagines the auto
engineers typically use to interactivel
machine-checked proof. When a syste
proof about the system, traditional aut
proof from scratch. Proof repair, in
tomation: it determines how the syster
information to help fix the broken proo
Proof repair in this thesis works by co
algorithms with program transformatio
ing and the transformations operate ov
proofs called *proof terms*. Thanks to the
differencing and the transformations c
results in dependent type theory. For ex
ternalizes univalent transport from hon
novel transformations over equalities t
This approach is realized inside of a
Coq proof assistant. Case studies show
use that this proof repair tool suite can
on real proof developments.

PhD Thesis

Adapting Proof Automation to Adapt Proofs

Talia Ringer
University of Washington, USA

Nathaniel Yazdani
University of Washington, USA

John Leo
Halfaya Research, USA

Dan Grossman
University of Washington, USA

Abstract

We extend proof automation in an interactive theorem prover to analyze *changes* in specifications and proofs. Our approach leverages the history of changes to specifications and proofs to search for a patch that can be applied to other specifications and proofs that need to change in analogous ways.

the search
This in tur
manually
Despite
tants is brittle. Even a minor change to a definition or the-
orem can break many dependent proofs. This is a major

CPP 2018

Proof Repair across Type Equivalences

Talia Ringer
University of Washington
USA
tringer@cs.washington.edu

RanDair Porter
University of Washington
USA
randair@uw.edu

Nathaniel Yazdani
Northeastern University
USA
yazdani.n@husky.neu.edu

John Leo
Halfaya Research
USA
leo@halfaya.org

Dan Grossman
University of Washington
USA
djg@cs.washington.edu

Abstract

We describe a new approach to automatically repairing broken proofs in the Coq proof assistant in response to changes in types. Our approach combines a configurable proof term transformation with a decompiler from proof terms to suggested tactic scripts. The proof term transformation implements transport across equivalences in a way that removes references to the old version of the changed type and does not rely on axioms beyond those Coq assumes.

We have implemented this approach in Pumpkin Pi, an extension to the Pumpkin Patch Coq plugin suite for proof repair. We demonstrate Pumpkin Pi's flexibility on eight case studies, including supporting a benchmark from a user study, easing development with dependent types, porting functions and proofs between unary
and supporting an industrial proof e
between Coq and other verification t

1 Introduction

Program verification with interactive theorem provers has come a long way since its inception, especially when it comes to the scale of programs that can be verified. The seL4 [21] verified operating system kernel, for example, is the effort of a team of proof engineers spanning more than a million lines of proof, costing over 20 person-years. Given a famous 1977 critique of verification [12] (emphasis ours):

> *A sufficiently fanatical researcher* might be willing to devote *two or three years* to verifying a significant piece of software if he could be assured that the software would remain stable.

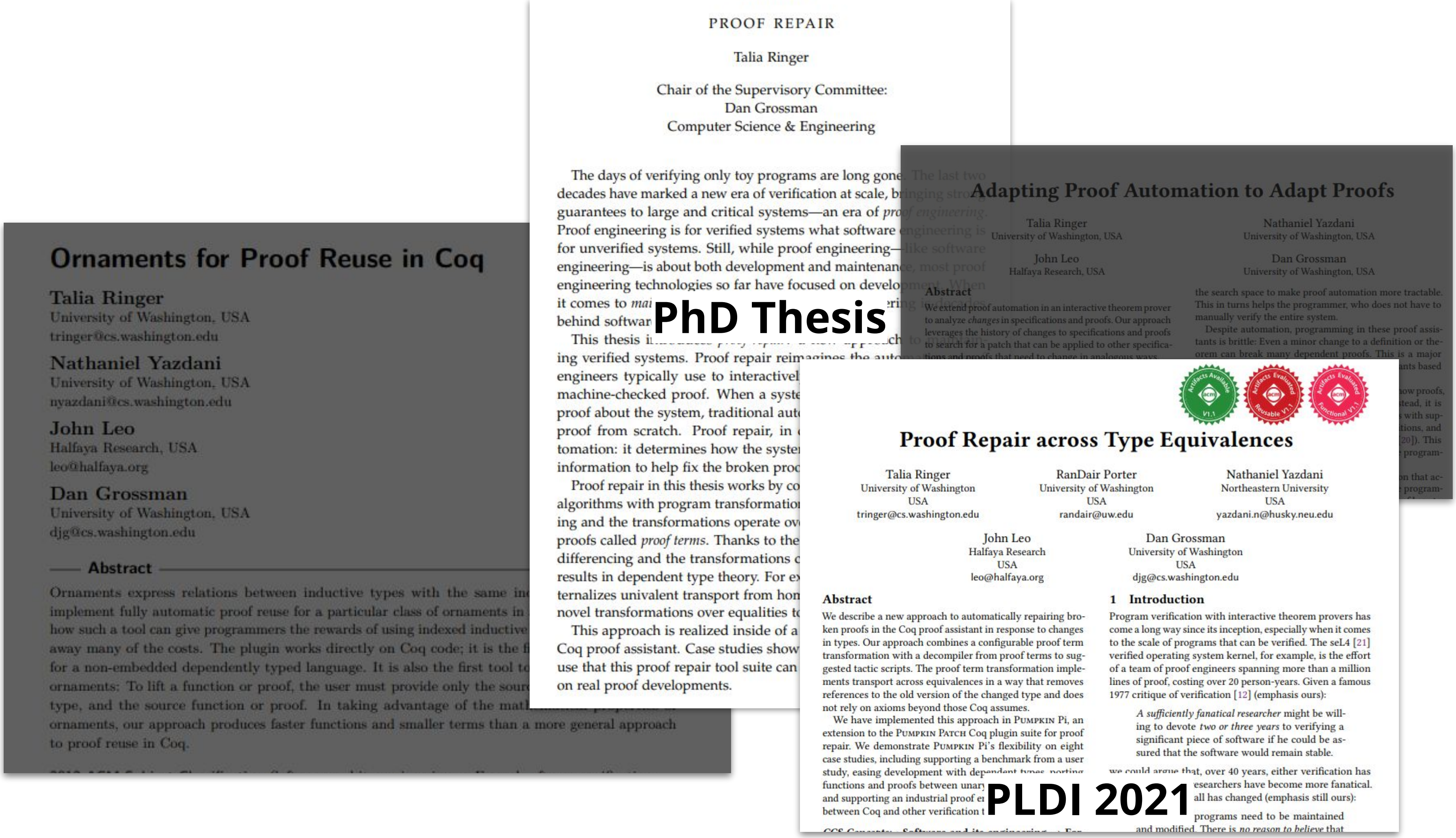
we could argue that, over 40 years, either verification has
esearchers have become more fanatical.
all has changed (emphasis still ours):

programs need to be maintained
and modified. There is *no reason to believe* that

PLDI 2021

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)



Ornaments for Proof Reuse in Coq

Talia Ringer
University of Washington, USA
tringer@cs.washington.edu

Nathaniel Yazdani
University of Washington, USA
nyazdani@cs.washington.edu

John Leo
Halfaya Research, USA
leo@halfaya.org

Dan Grossman
University of Washington, USA
djg@cs.washington.edu

Abstract

Ornaments express relations between inductive types with the same in
implement fully automatic proof reuse for a particular class of ornaments in
how such a tool can give programmers the rewards of using indexed inductive
away many of the costs. The plugin works directly on Coq code; it is the fi
for a non-embedded dependently typed language. It is also the first tool to
ornaments: To lift a function or proof, the user must provide only the sour
type, and the source function or proof. In taking advantage of the math
ornaments, our approach produces faster functions and smaller terms than a more general approach
to proof reuse in Coq.

PROOF REPAIR

Talia Ringer

Chair of the Supervisory Committee:
Dan Grossman
Computer Science & Engineering

The days of verifying only toy programs are long gone. The last two
decades have marked a new era of verification at scale, bringing strong
guarantees to large and critical systems—an era of *proof engineering*.
Proof engineering is for verified systems what software engineering is
for unverified systems. Still, while proof engineering—like software
engineering—is about both development and maintenance, most proof
engineering technologies so far have focused on development. When
it comes to *mai*
behind softwar

This thesis i
ing verified systems. Proof repair reimagines the auto
engineers typically use to interactivel
machine-checked proof. When a syste
proof about the system, traditional aut
proof from scratch. Proof repair, in
tomation: it determines how the syste
information to help fix the broken proo

Proof repair in this thesis works by co
algorithms with program transformatio
ing and the transformations operate ov
proofs called *proof terms*. Thanks to the
differencing and the transformations c
results in dependent type theory. For ex
ternalizes univalent transport from hom
novel transformations over equalities to

This approach is realized inside of a
Coq proof assistant. Case studies show
use that this proof repair tool suite can
on real proof developments.

PhD Thesis

Adapting Proof Automation to Adapt Proofs

Talia Ringer
University of Washington, USA

Nathaniel Yazdani
University of Washington, USA

John Leo
Halfaya Research, USA

Dan Grossman
University of Washington, USA

Abstract

We extend proof automation in an interactive theorem prover to analyze *changes* in specifications and proofs. Our approach leverages the history of changes to specifications and proofs to search for a patch that can be applied to other specifications and proofs that need to change in analogous ways.

the search space to make proof automation more tractable. This in turns helps the programmer, who does not have to manually verify the entire system.

Despite automation, programming in these proof assistants is brittle: Even a minor change to a definition or theorem can break many dependent proofs. This is a major
ants based

ow proofs,
tead, it is
s with sup-
tions, and
20]). This
program-

n that ac-
program-

Proof Repair across Type Equivalences

Talia Ringer
University of Washington
USA
tringer@cs.washington.edu

RanDair Porter
University of Washington
USA
randair@uw.edu

Nathaniel Yazdani
Northeastern University
USA
yazdani.n@husky.neu.edu

John Leo
Halfaya Research
USA
leo@halfaya.org

Dan Grossman
University of Washington
USA
djg@cs.washington.edu

Abstract

We describe a new approach to automatically repairing broken proofs in the Coq proof assistant in response to changes in types. Our approach combines a configurable proof term transformation with a decompiler from proof terms to suggested tactic scripts. The proof term transformation implements transport across equivalences in a way that removes references to the old version of the changed type and does not rely on axioms beyond those Coq assumes.

We have implemented this approach in Pumpkin Pi, an extension to the Pumpkin Patch Coq plugin suite for proof repair. We demonstrate Pumpkin Pi's flexibility on eight case studies, including supporting a benchmark from a user study, easing development with dependent types, porting functions and proofs between unary
and supporting an industrial proof en
between Coq and other verification t

1 Introduction

Program verification with interactive theorem provers has come a long way since its inception, especially when it comes to the scale of programs that can be verified. The seL4 [21] verified operating system kernel, for example, is the effort of a team of proof engineers spanning more than a million lines of proof, costing over 20 person-years. Given a famous 1977 critique of verification [12] (emphasis ours):

> *A sufficiently fanatical researcher* might be willing to devote *two or three years* to verifying a significant piece of software if he could be assured that the software would remain stable.

we could argue that, over 40 years, either verification has
esearchers have become more fanatical.
all has changed (emphasis still ours):

> programs need to be maintained
> and modified. There is *no reason to believe* that

PLDI 2021

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

You have changed a datatype, and now the standard library is broken!

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

```
Inductive list T :=
| nil : list T
| cons : T → list T → list T
```

```
(* Repair all 451 functions & proofs: *)
Repair Module Old.list New.list in StdLib.
```

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

```
Inductive list T :=
| cons : T → list T → list T
| nil : list T
```

```
(* Repair all 451 functions & proofs: *)
Repair Module Old.list New.list in StdLib.
```

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

```
Inductive list T :=
| cons : T → list T → list T
| nil : list T
```

```
(* Repair all 451 functions & proofs: *)
Repair Module Old.list New.list in StdLib.
```

451 functions & proofs, 25 seconds

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Traditional proof repair:

+ predictable
+ dependable
+ understandable
- limited in scope
- takes expertise to extend

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable

PUMPKIN Pi supports any change described by a type equivalence.

The Univalent Foundations Program. 2013. **Homotopy Type Theory: Univalent Foundations of Mathematics.** Institute for Advanced Study.

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable

PUMPKIN Pi supports any change described by a type equivalence.

The Univalent Foundations Program. 2013. **Homotopy Type Theory: Univalent Foundations of Mathematics.** Institute for Advanced Study.

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable

$$\begin{tikzcd}
\text{Old.list T} \arrow[r, bend left, "\textbf{swap}"] & \text{New.list T} \arrow[l, bend left, "\textbf{swap\_back}"]
\end{tikzcd}$$

Example: Proof Repair (PUMPKIN Pi)

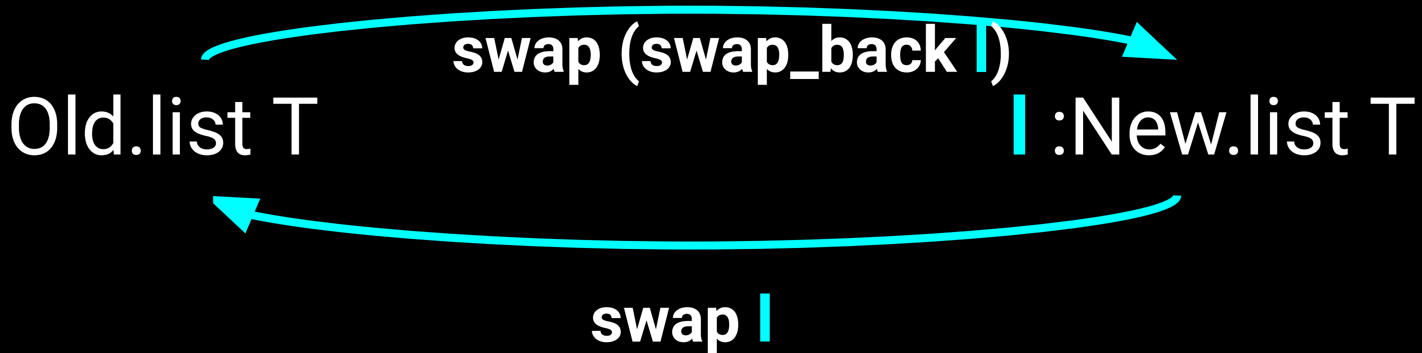
Predictable (Equivalences), Dependable

$$\begin{tikzcd}
l : \text{Old.list } T \arrow[r, bend left, "\textbf{swap } l"] & \text{New.list } T \arrow[l, bend left, "\textbf{swap\_back (swap } l\textbf{)}"]
\end{tikzcd}$$

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable



swap (swap_back l)

Old.list T

l :New.list T

swap l

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable

update

old

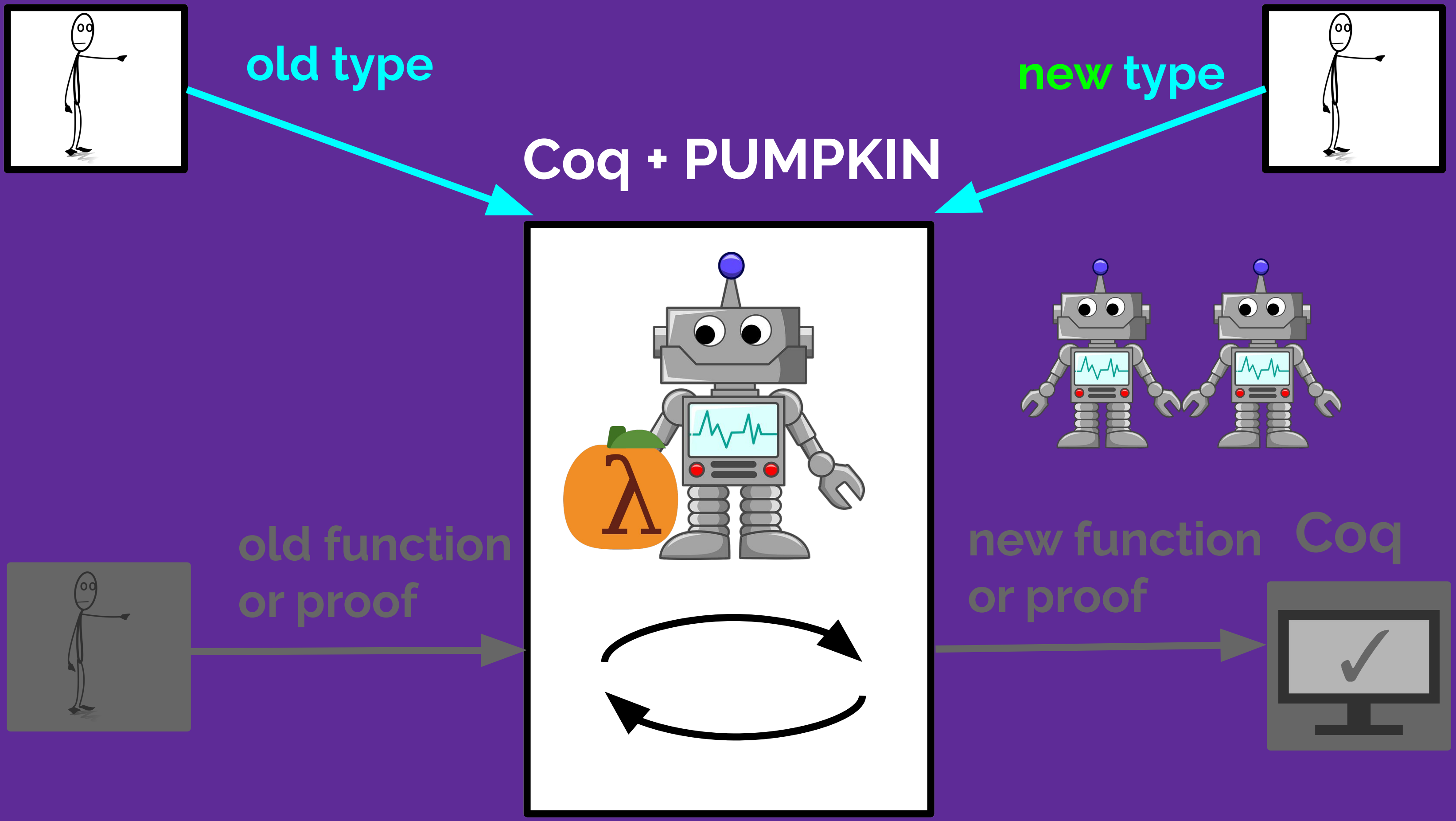
new

revert

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable



old type

new type

Coq + PUMPKIN

old function
or proof

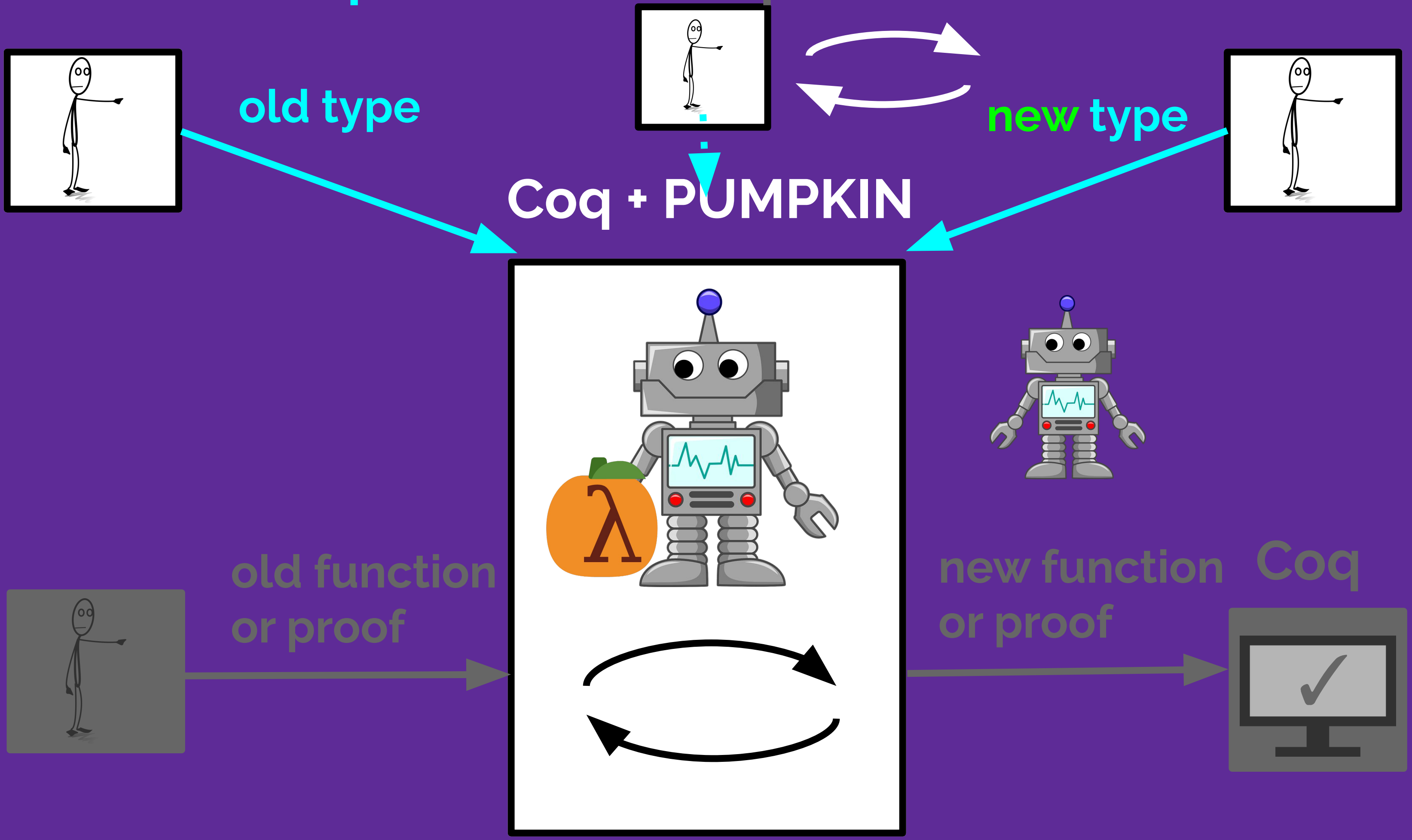
new function
or proof

Coq

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable



old type

new type

Coq + PUMPKIN

old function or proof

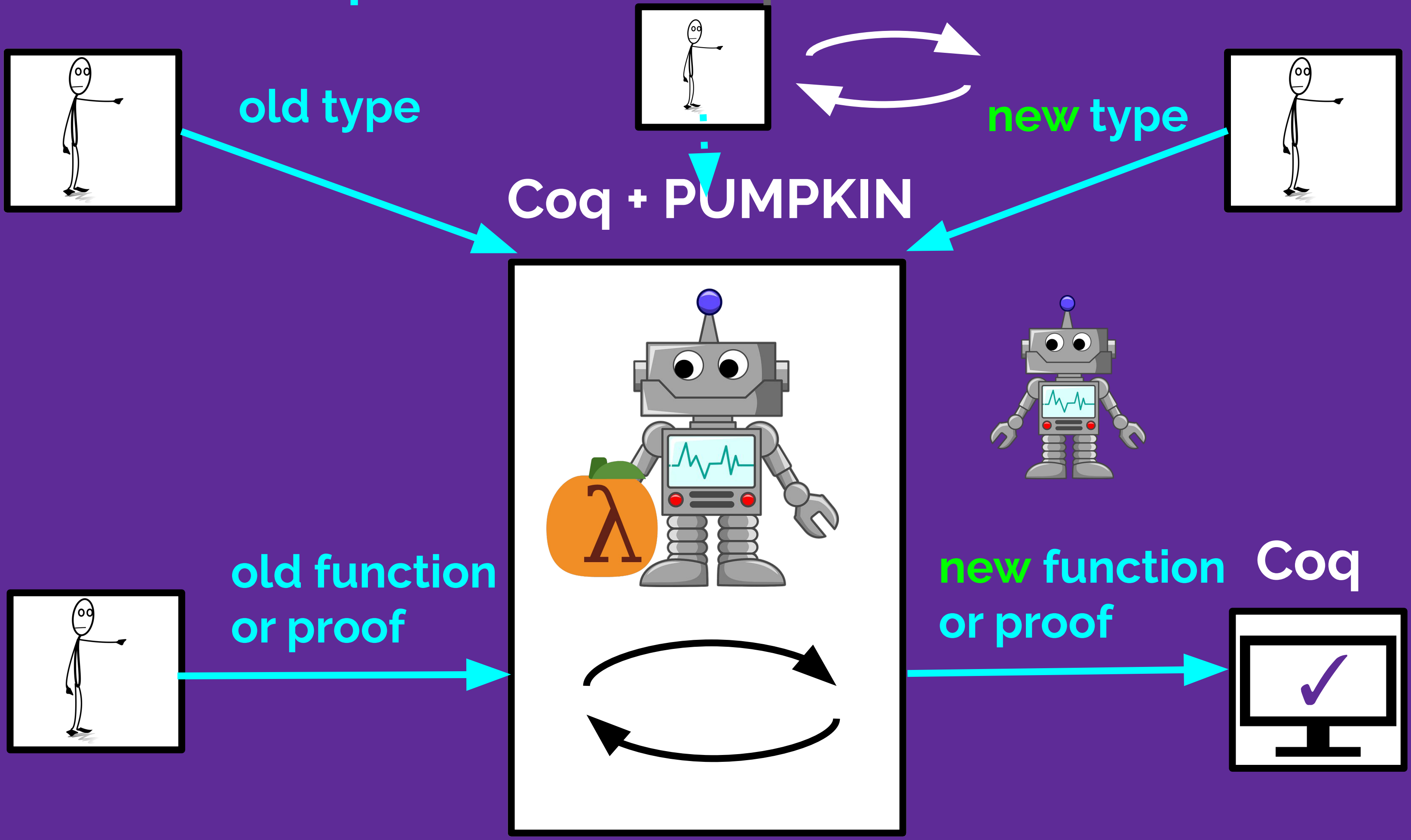
new function or proof

Coq

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable



old type

new type

Coq + PUMPKIN

old function or proof

new function or proof

Coq

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable

PUMPKIN Pi is
flexible & **useful**
for **real scenarios.**

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Predictable (Equivalences), Dependable

	1	Unary to binary (classic benchmark)	🤖
☺	2	Modifying PL (user study)	🤖 🤖
	3	Extending PL (user study)	🤖
☺	4	Adding indices (ornaments)	🤖 🤖
☺	5	Factoring constructors (reviewer)	🤖
	6	Permute hypotheses (type theorist)	🤖
☺	7	Vector to finite set (type theorist)	🤖
☺	8	Industrial use (mixed methods)	🤖 🤖

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Symbolic proof repair:

+ predictable
+ dependable
+ understandable
- limited in scope
- takes expertise to extend

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

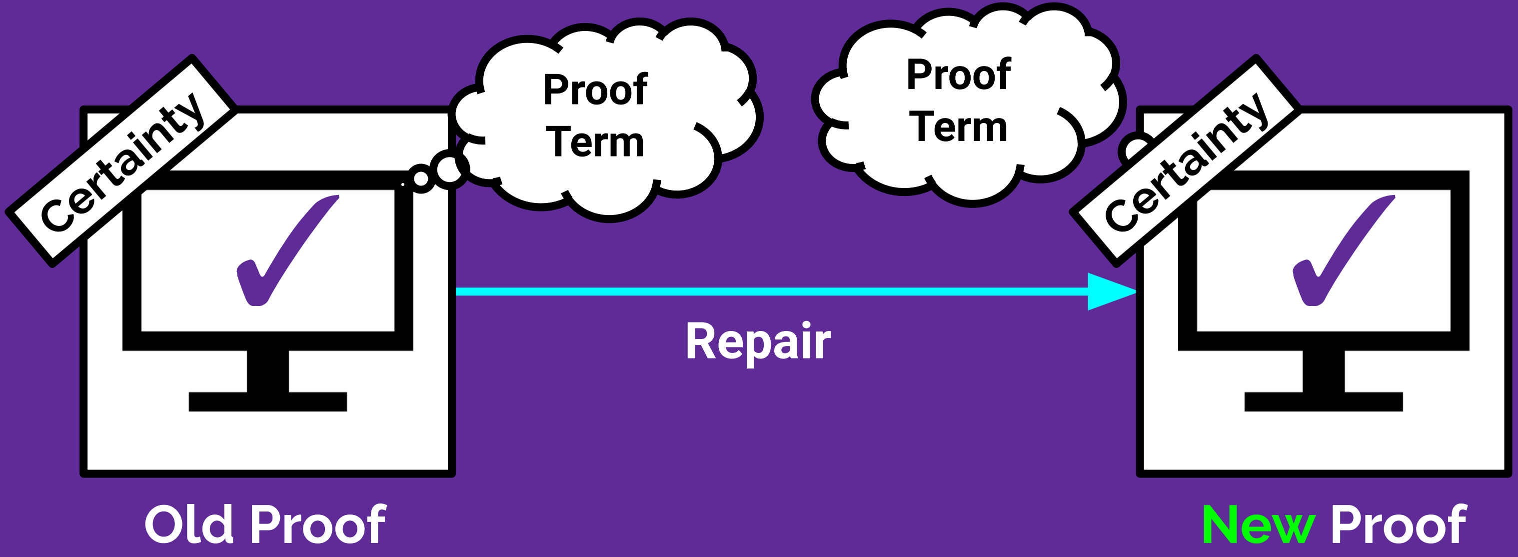
Symbolic proof repair:

+ predictable
+ dependable
+ understandable* (for type nerds)
- limited in scope
- takes expertise to extend

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

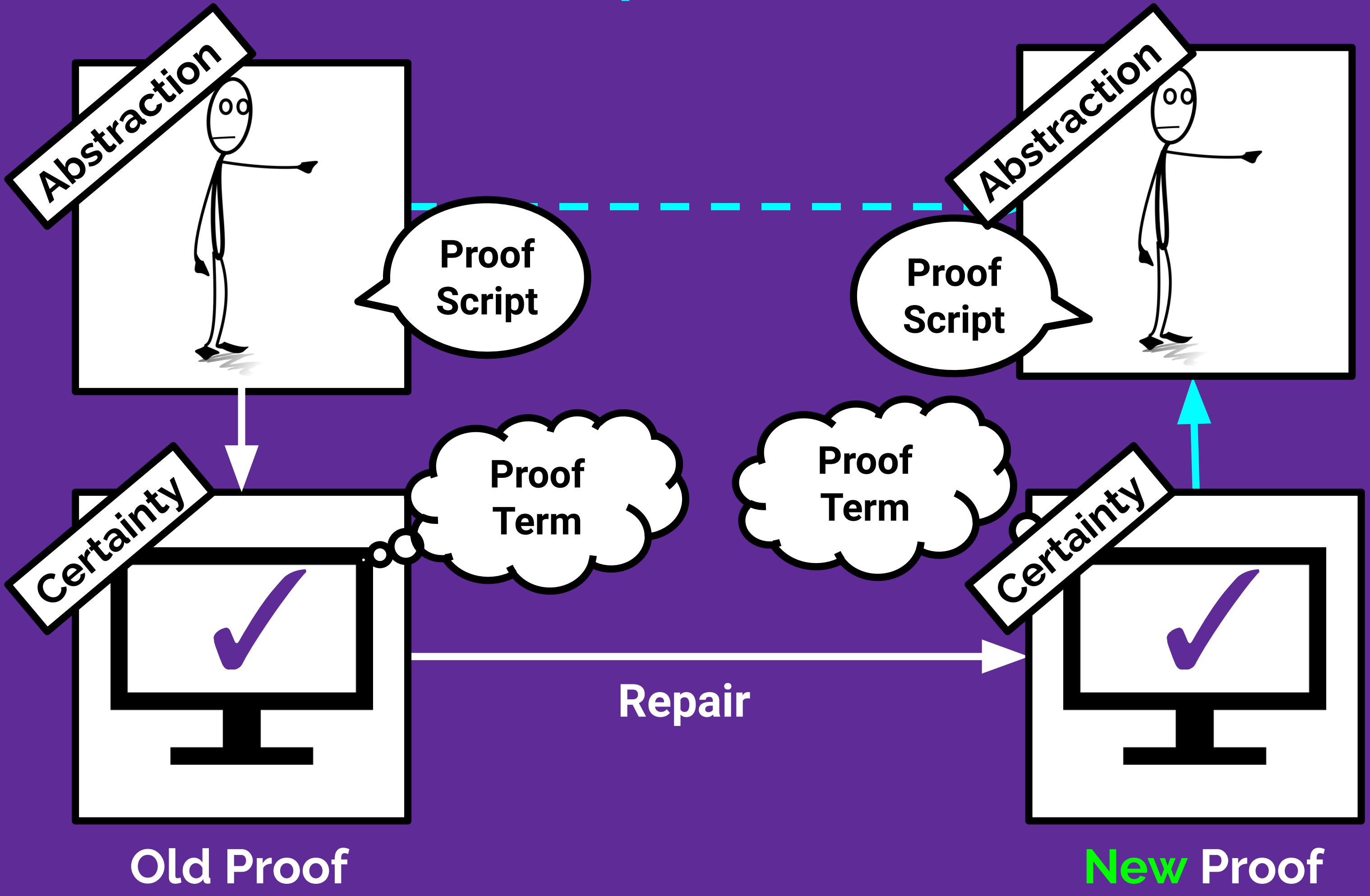
Understandable* (Transport as a Transformation)



Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

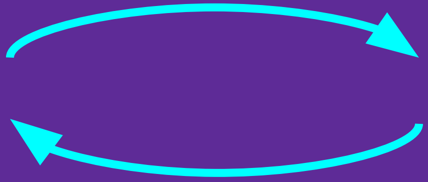
Understandable* (Transport as a Transformation)



Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Understandable* (Transport as a Transformation)



Transport: Rewriting across Equivalences

The Univalent Foundations Program. 2013. **Homotopy Type Theory: Univalent Foundations of Mathematics.** Institute for Advanced Study.

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Understandable* (Transport as a Transformation)

Transport as a Proof Term Transformation

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Understandable* (Transport as a Transformation)

For type nerds:
Deconstruct Equivalence
(Lambek's Theorem)

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

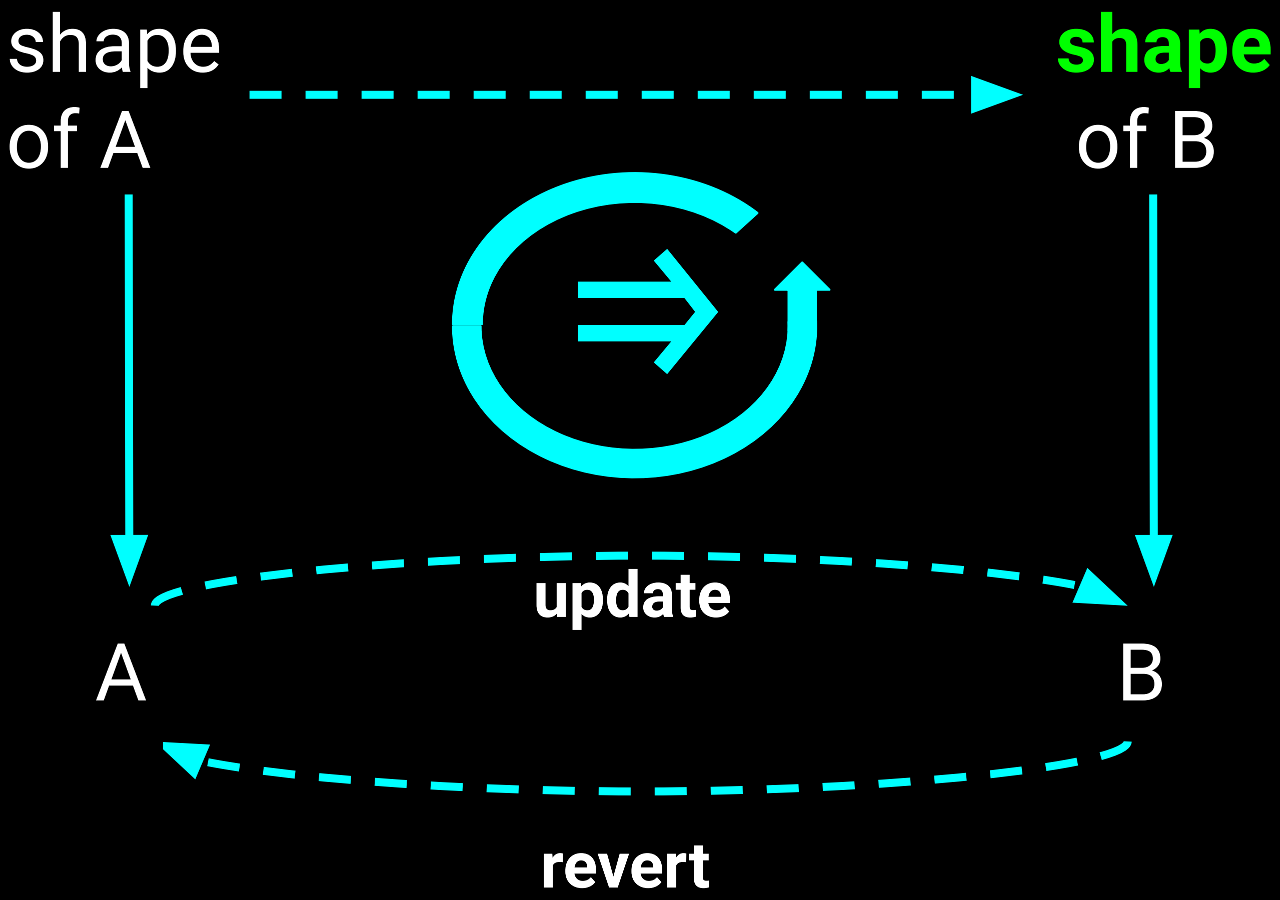
Understandable* (Transport as a Transformation)

$$\begin{tikzcd}
\text{shape of A} \arrow[r, dashed] \arrow[d] \arrow[dr, phantom, "\Rightarrow"] & \textbf{shape}\text{ of B} \arrow[d] \\
A \arrow[r, dashed, bend left, "\textbf{update}"] & B \arrow[l, dashed, bend left, "\textbf{revert}"]
\end{tikzcd}$$

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Understandable* (Transport as a Transformation)



shape of A

shape of B

update

A

B

revert

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Symbolic proof repair:

\+ predictable

\+ dependable

\+ understandable* (for type nerds)

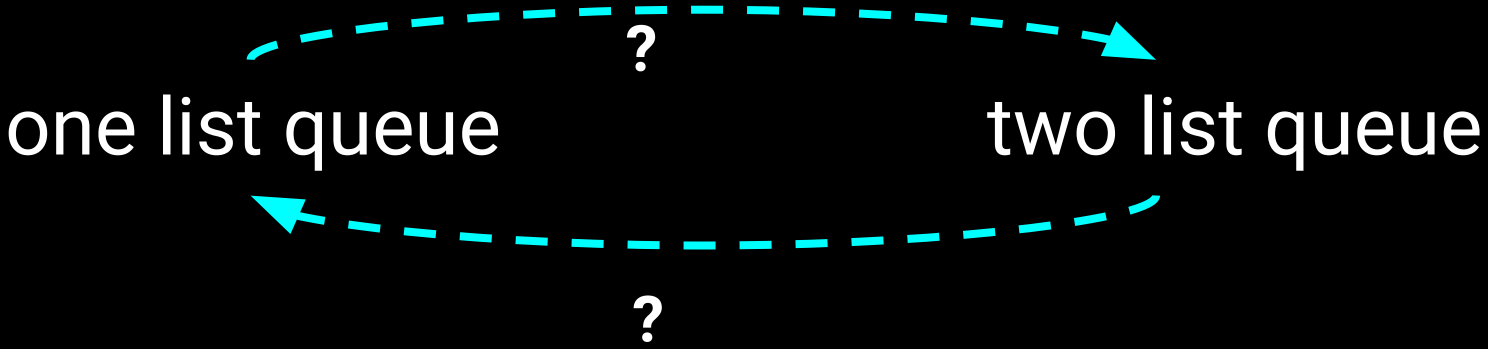
\- limited in scope

\- takes expertise to extend

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

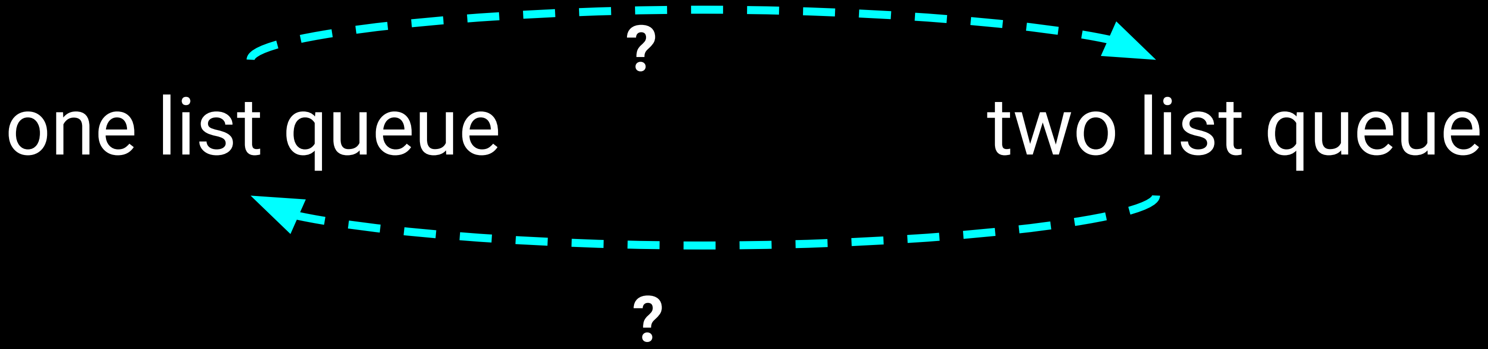
Limited Scope (Quotient Equivalences), Hard to Extend



Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. **Internalizing Representation Independence with Univalence.** POPL 2021.

Example: Proof Repair (PUMPKIN Pi)

Limited Scope (Setoid Equivalences), Hard to Extend



one list queue

two list queue

Example: Proof Repair (PUMPKIN Pi)

Limited Scope (Setoid Equivalences), Hard to Extend

One PhD student,
one undergrad,
one advisor,
2.5 years.
Is this sustainable?

2.5 years later...

Symbolic Automation (Part 2 of 5)

Example: Proof Repair (PUMPKIN Pi)

Limited Scope (Setoid Equivalences), Hard to Extend

2.5 years later...

"While the reviewers agree that this article tackles an interesting problem, its contributions with respect to pre-existing and related work appear too incremental and limited in scope."

Symbolic Automation (Part 2 of 5)

1. Proof Assistants
2. Symbolic Automation
3. Neural Automation
4. Building Bridges
5. Opportunities

Neural automation:

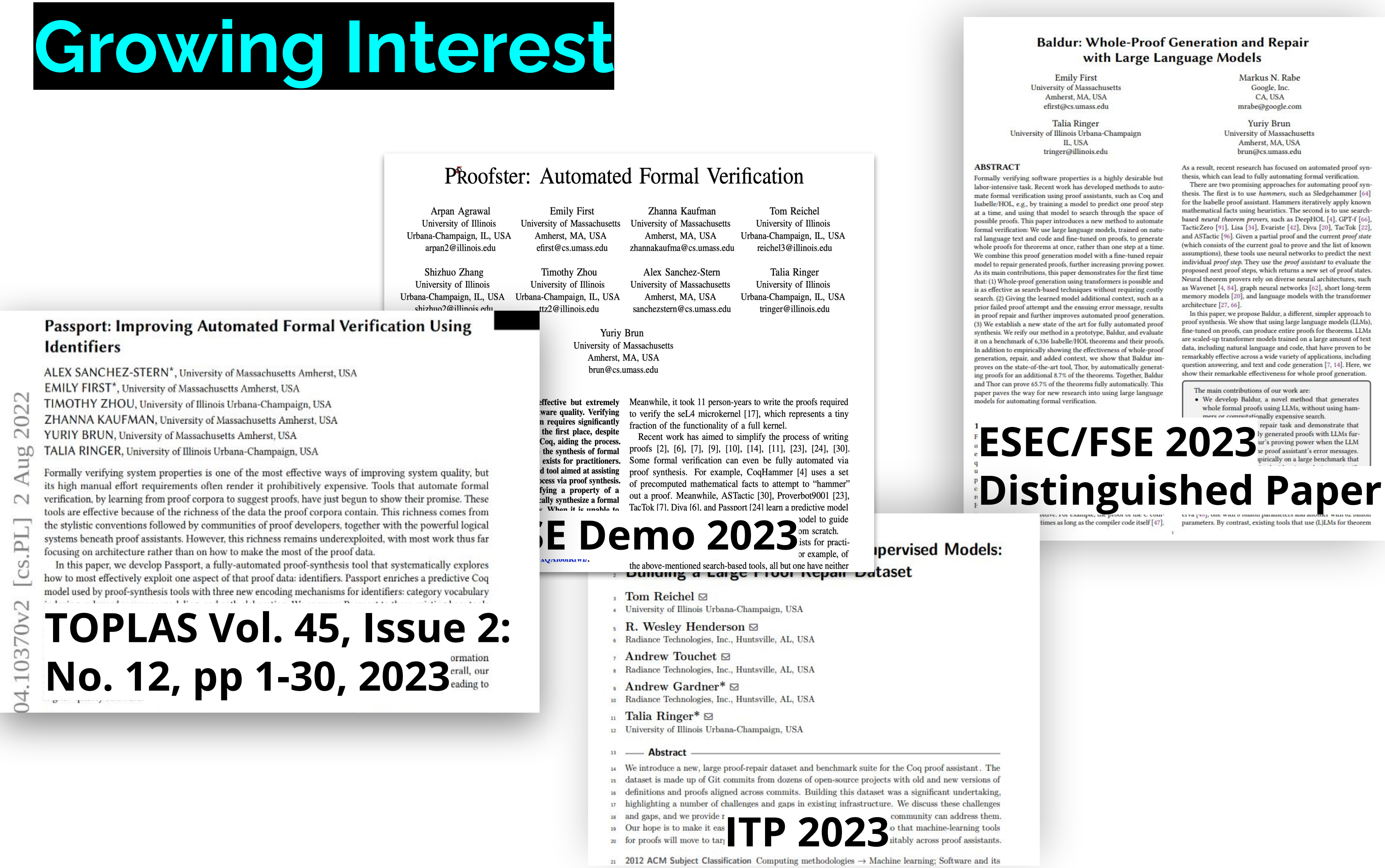
- *un*predictable
- *not* dependable
- *not* understandable
+ *not very* limited in scope
+ takes *little* expertise to extend

Neural Automation (Part 3 of 5)

Important note:
Neural proof automation is not brand new! It is just growing in popularity.

Neural Automation (Part 3 of 5)

Growing Interest



Passport: Improving Automated Formal Verification Using Identifiers

ALEX SANCHEZ-STERN*, University of Massachusetts Amherst, USA
EMILY FIRST*, University of Massachusetts Amherst, USA
TIMOTHY ZHOU, University of Illinois Urbana-Champaign, USA
ZHANNA KAUFMAN, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

04.10370v2 [cs.PL] 2 Aug 2022

Formally verifying system properties is one of the most effective ways of improving system quality, but its high manual effort requirements often render it prohibitively expensive. Tools that automate formal verification, by learning from proof corpora to suggest proofs, have just begun to show their promise. These tools are effective because of the richness of the data the proof corpora contain. This richness comes from the stylistic conventions followed by communities of proof developers, together with the powerful logical systems beneath proof assistants. However, this richness remains underexploited, with most work thus far focusing on architecture rather than on how to make the most of the proof data.

In this paper, we develop Passport, a fully-automated proof-synthesis tool that systematically explores how to most effectively exploit one aspect of that proof data: identifiers. Passport enriches a predictive Coq model used by proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary

TOPLAS Vol. 45, Issue 2: No. 12, pp 1-30, 2023

Proofster: Automated Formal Verification

Arpan Agrawal
University of Illinois
Urbana-Champaign, IL, USA
arpan2@illinois.edu

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Zhanna Kaufman
University of Massachusetts
Amherst, MA, USA
zhannakaufma@cs.umass.edu

Tom Reichel
University of Illinois
Urbana-Champaign, IL, USA
reichel3@illinois.edu

Shizhuo Zhang
University of Illinois
Urbana-Champaign, IL, USA

Timothy Zhou
University of Illinois
Urbana-Champaign, IL, USA
ttz2@illinois.edu

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Talia Ringer
University of Illinois
Urbana-Champaign, IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

Meanwhile, it took 11 person-years to write the proofs required to verify the seL4 microkernel [17], which represents a tiny fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30]. Some formal verification can even be fully automated via proof synthesis. For example, CoqHammer [4] uses a set of precomputed mathematical facts to attempt to "hammer" out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23], TacTok [7], Diva [6], and Passport [24] learn a predictive model

the above-mentioned search-based tools, all but one have neither

SE Demo 2023

---

pervised Models:
Building a Large Proof Repair Dataset

Tom Reichel ✉
University of Illinois Urbana-Champaign, USA

R. Wesley Henderson ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Touchet ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Gardner* ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Talia Ringer* ✉
University of Illinois Urbana-Champaign, USA

Abstract

We introduce a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from dozens of open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset was a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide r … community can address them. Our hope is to make it eas … o that machine-learning tools for proofs will move to tar … uitably across proof assistants.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its

ITP 2023

Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Markus N. Rabe
Google, Inc.
CA, USA
mrabe@google.com

Talia Ringer
University of Illinois Urbana-Champaign
IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Formally verifying software properties is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time, and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language text and code and fine-tuned on proofs, to generate whole proofs for theorems at once, rather than one step at a time. We combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. As its main contributions, this paper demonstrates for the first time that: (1) Whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. (2) Giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation. (3) We establish a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs. In addition to empirically showing the effectiveness of whole-proof generation, repair, and added context, we show that Baldur improves on the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [64] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [4], GPT-f [66], TacticZero [91], Lisa [34], Evariste [42], Diva [20], TacTok [22], and ASTactic [96]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [4, 84], graph neural networks [62], short long-term memory models [20], and language models with the transformer architecture [27, 66].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [7, 14]. Here, we show their remarkable effectiveness for whole proof generation.

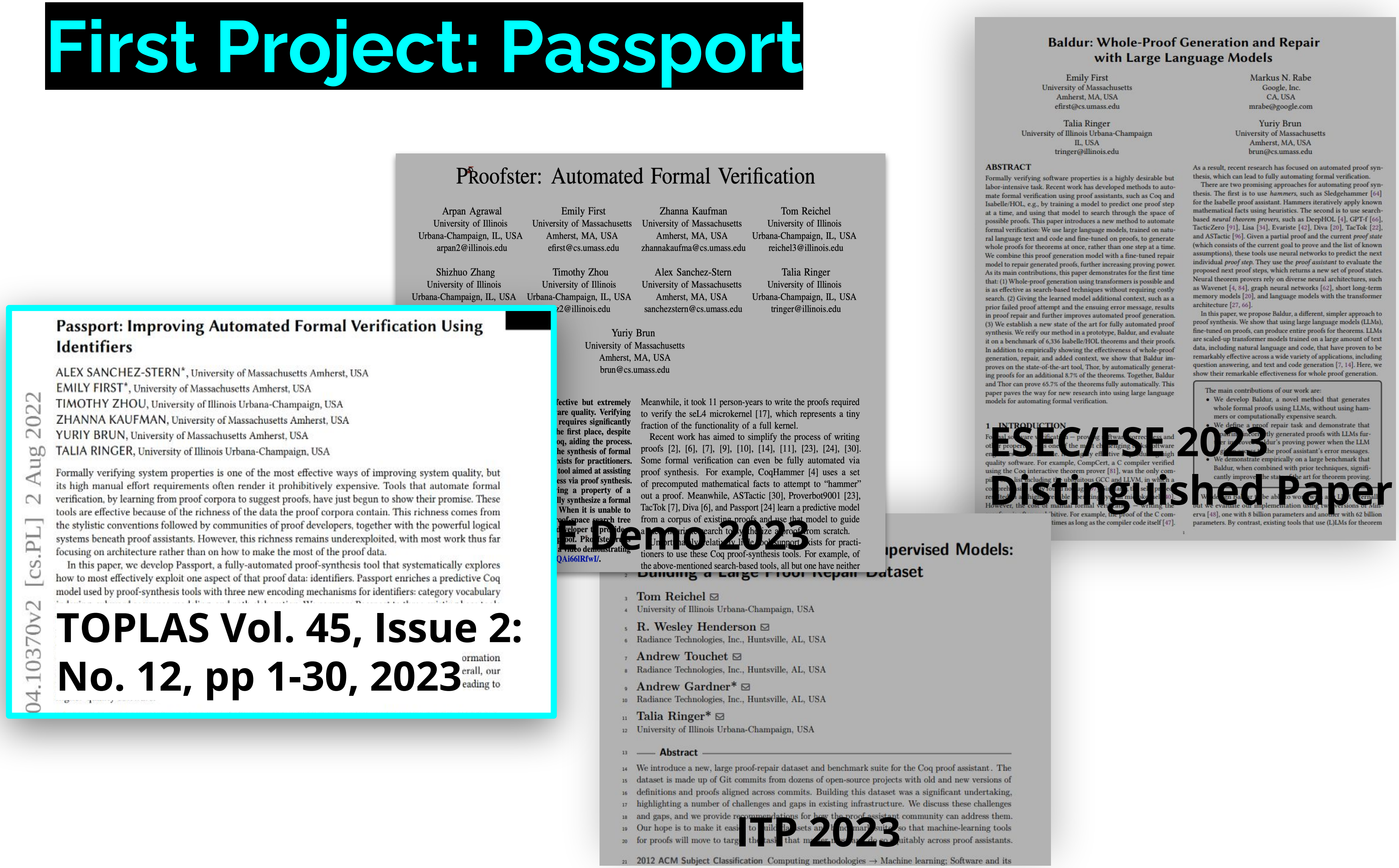
The main contributions of our work are:
- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.

parameters. By contrast, existing tools that use (L)LMs for theorem

ESEC/FSE 2023
Distinguished Paper

Neural Automation (Part 3 of 5)

First Project: Passport



Passport: Improving Automated Formal Verification Using Identifiers

ALEX SANCHEZ-STERN*, University of Massachusetts Amherst, USA
EMILY FIRST*, University of Massachusetts Amherst, USA
TIMOTHY ZHOU, University of Illinois Urbana-Champaign, USA
ZHANNA KAUFMAN, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

Formally verifying system properties is one of the most effective ways of improving system quality, but its high manual effort requirements often render it prohibitively expensive. Tools that automate formal verification, by learning from proof corpora to suggest proofs, have just begun to show their promise. These tools are effective because of the richness of the data the proof corpora contain. This richness comes from the stylistic conventions followed by communities of proof developers, together with the powerful logical systems beneath proof assistants. However, this richness remains underexploited, with most work thus far focusing on architecture rather than on how to make the most of the proof data.

In this paper, we develop Passport, a fully-automated proof-synthesis tool that systematically explores how to most effectively exploit one aspect of that proof data: identifiers. Passport enriches a predictive Coq model used by proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary

04.10370v2 [cs.PL] 2 Aug 2022

TOPLAS Vol. 45, Issue 2: No. 12, pp 1-30, 2023

PRoofster: Automated Formal Verification

Arpan Agrawal, University of Illinois Urbana-Champaign, IL, USA, arpan2@illinois.edu
Emily First, University of Massachusetts Amherst, MA, USA, efirst@cs.umass.edu
Zhanna Kaufman, University of Massachusetts Amherst, MA, USA, zhannakaufma@cs.umass.edu
Tom Reichel, University of Illinois Urbana-Champaign, IL, USA, reichel3@illinois.edu
Shizhuo Zhang, University of Illinois Urbana-Champaign, IL, USA
Timothy Zhou, University of Illinois Urbana-Champaign, IL, USA
Alex Sanchez-Stern, University of Massachusetts Amherst, MA, USA, sanchezstern@cs.umass.edu
Talia Ringer, University of Illinois Urbana-Champaign, IL, USA, tringer@illinois.edu
Yuriy Brun, University of Massachusetts Amherst, MA, USA, brun@cs.umass.edu

Meanwhile, it took 11 person-years to write the proofs required to verify the seL4 microkernel [17], which represents a tiny fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30]. Some formal verification can even be fully automated via proof synthesis. For example, CoqHammer [4] uses a set of precomputed mathematical facts to attempt to "hammer" out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23], TacTok [7], Diva [6], and Passport [24] learn a predictive model from a corpus of existing proofs and use that model to guide

tioners to use these Coq proof-synthesis tools. For example, of the above-mentioned search-based tools, all but one have neither

E Demo 2023

…pervised Models: Building a Large Proof Repair Dataset

Tom Reichel ✉
University of Illinois Urbana-Champaign, USA

R. Wesley Henderson ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Touchet ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Gardner* ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Talia Ringer* ✉
University of Illinois Urbana-Champaign, USA

Abstract

We introduce a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from dozens of open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset was a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide re… Our hope is to make it easi… for proofs will move to targ…

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its

ITP 2023

Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First, University of Massachusetts, Amherst, MA, USA, efirst@cs.umass.edu
Markus N. Rabe, Google, Inc., CA, USA, mrabe@google.com
Talia Ringer, University of Illinois Urbana-Champaign, IL, USA, tringer@illinois.edu
Yuriy Brun, University of Massachusetts, Amherst, MA, USA, brun@cs.umass.edu

ABSTRACT

Formally verifying software properties is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time, and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language text and code and fine-tuned on proofs, to generate whole proofs for theorems at once, rather than one step at a time. We combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. As its main contributions, this paper demonstrates for the first time that: (1) Whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. (2) Giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation. (3) We establish a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs. In addition to empirically showing the effectiveness of whole-proof generation, repair, and added context, we show that Baldur improves on the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

1 INTRODUCTION

As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [64] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [4], GPT-f [66], TacticZero [91], Lisa [34], Evariste [42], Diva [20], TacTok [22], and ASTactic [96]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [4, 84], graph neural networks [62], short long-term memory models [20], and language models with the transformer architecture [27, 66].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [7, 14]. Here, we show their remarkable effectiveness for whole proof generation.

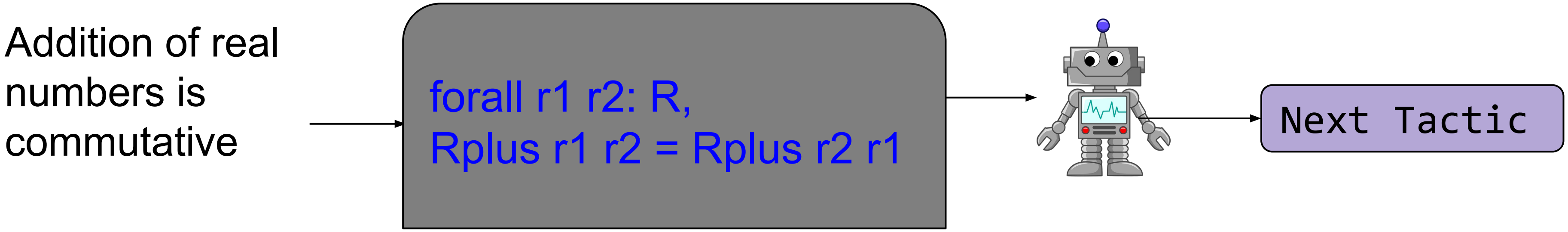
The main contributions of our work are:
- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.
- We define a proof repair task and demonstrate that …the proof assistant's error messages.
- We demonstrate empirically on a large benchmark that Baldur, when combined with prior techniques, significantly improve… the art for theorem proving.

…erva [48], one with 8 billion parameters and another with 62 billion parameters. By contrast, existing tools that use (L)LMs for theorem

ESEC/FSE 2023 Distinguished Paper

Neural Automation (Part 3 of 5)

First Project: Passport



Addition of real numbers is commutative

```
forall r1 r2: R,
Rplus r1 r2 = Rplus r2 r1
```

Next Tactic

First Project: Passport

```
forall r1 r2: R,
Rplus r1 r2 = Rplus r2 r1
```

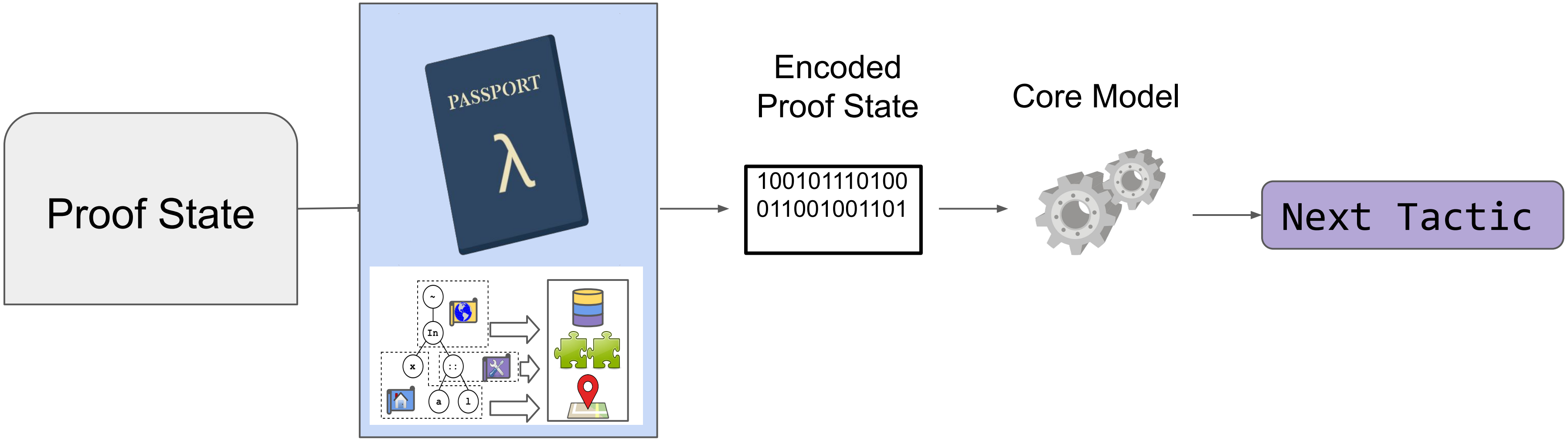
Category Vocabulary Indexing

Subword Sequence Modeling

Path Elaboration

Neural Automation (Part 3 of 5)

First Project: Passport



Proof State

Encoded
Proof State

100101110100
011001001101

Core Model

Next Tactic

Neural Automation (Part 3 of 5)

First Project: Passport

Neural automation:

- *un*predictable
- *not* dependable
- *not* understandable

\+ *not very* limited in scope
\+ takes *little* expertise to extend

Neural Automation (Part 3 of 5)

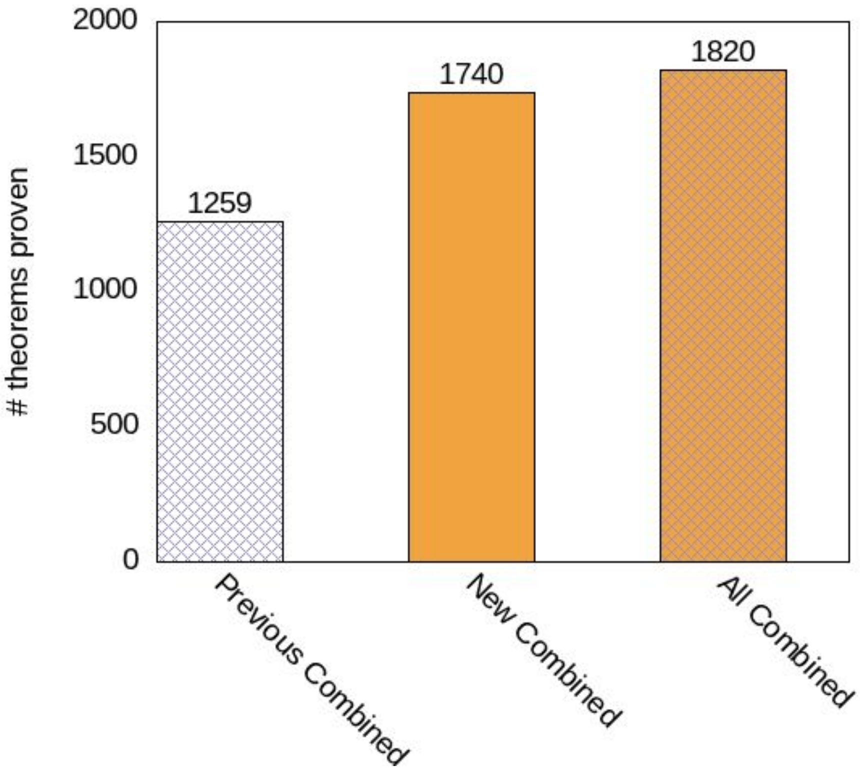
First Project: Passport – Big Scope

- Yang and Deng 2019
- **Math**ematical formalizations, proven correct **programs**, and Coq automation **libraries**
- 123 open-source Coq projects
- Trained on **97 projects** (**57,719 theorems**)
- Tested on **26 projects** (**10,782 theorems**)

CoqGym

First Project: Passport – Big Scope

We can prove **45% more** theorems than before!

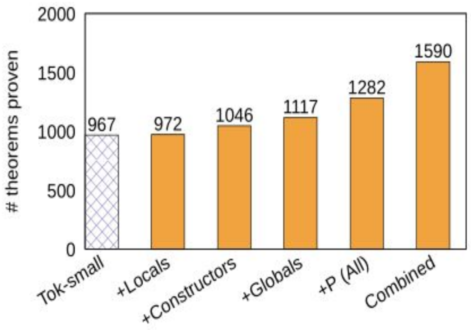


Neural Automation (Part 3 of 5)

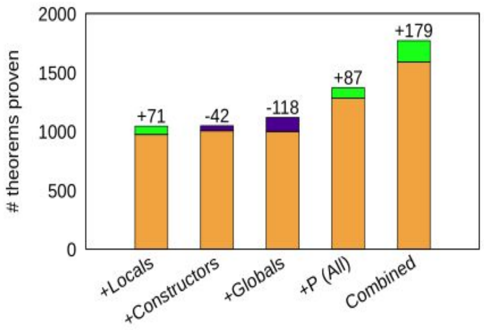
First Project: Passport – Big Scope

Diversity brings even higher returns!

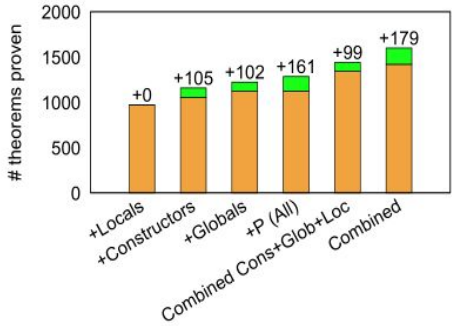
64% more theorems than the baseline!



(a) The impact of category vocabulary indexing on three identifier categories (without subwords or paths): local variables, type constructors, and global definitions.



(b) The impact of subword encoding on each of the categories of identifiers (with category vocabulary indexing but without paths).



(c) The impact of fully-qualified path encoding of type constructors and global definitions (with category vocabulary indexing but without subwords).

First Project: Passport – Easy to Extend

- Some **easy Python scripts** on top of someone else's **existing project**
- **Parallelized work** for different extensions between me and five other authors
- **Undergraduate** implemented most challenging extension in an order of **weeks**
- Scripts were **simple and fun** enough that I got excited when writing one in between drafting thesis chapters, ran into a couch, and broke my big toe

Neural Automation (Part 3 of 5)

First Project: Passport

Language models:

- *un*predictable
- *not* dependable
- *not* understandable

\+ *not very* limited in scope
\+ takes *little* expertise to extend

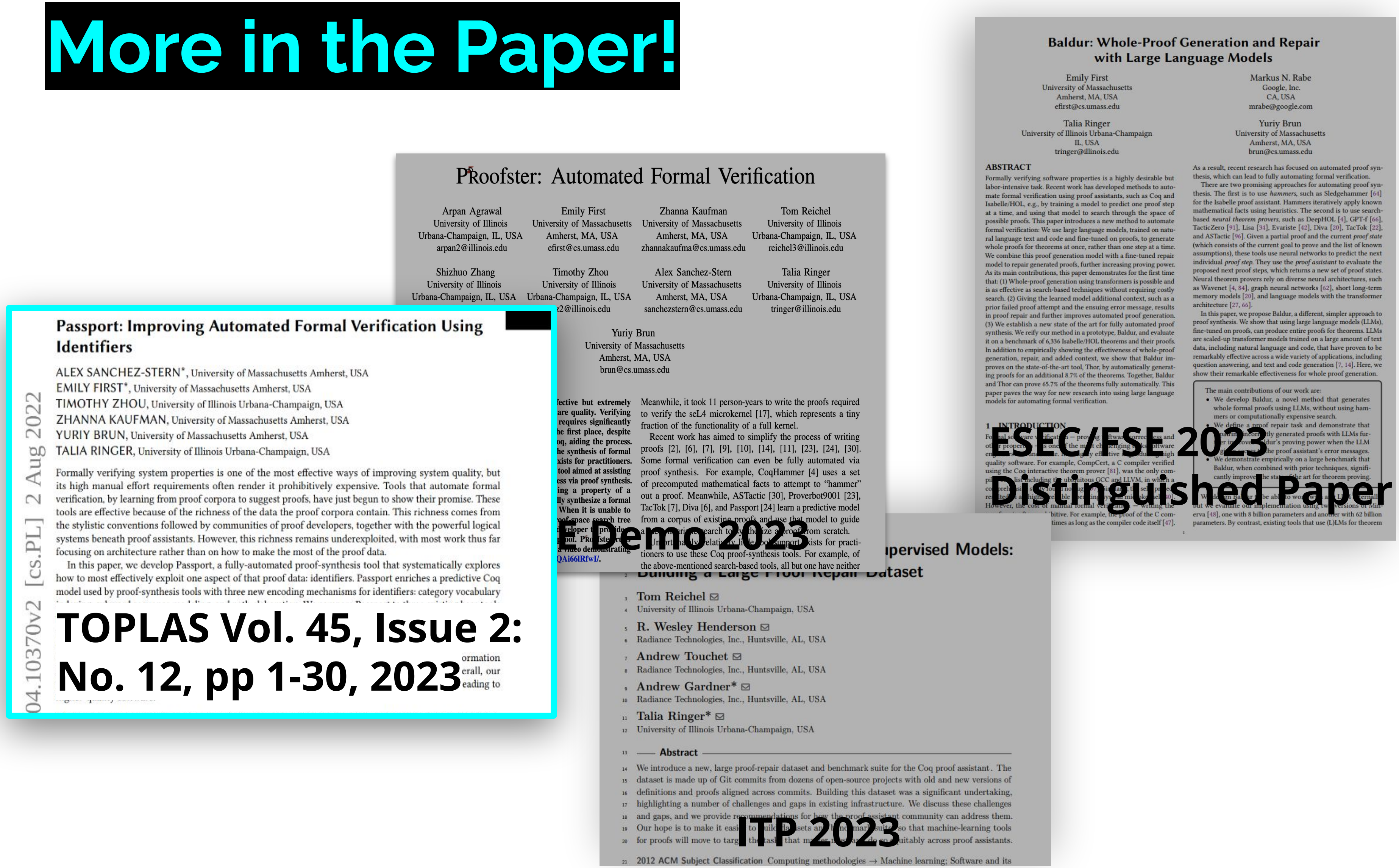
Neural Automation (Part 3 of 5)

First Project: Passport – Confusion

- Somehow, the ***name*** of the user running the training script impacted the **file order**, which impacted the **results** of training a model on **identical data** in an **identical way**
- We found a **nondeterminism bug in Pytorch**
- **Some combinations** of extensions worked **mysteriously poorly**, even though all together they helped
- Apparently **this is just life** with even small **LMs**? Is this life now? **Help?**

Neural Automation (Part 3 of 5)

More in the Paper!



ESEC/FSE 2023
Distinguished Paper

E Demo 2023

TOPLAS Vol. 45, Issue 2: No. 12, pp 1-30, 2023

ITP 2023

Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Markus N. Rabe
Google, Inc.
CA, USA
mrabe@google.com

Talia Ringer
University of Illinois Urbana-Champaign
IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Formally verifying software properties is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time, and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language text and code and fine-tuned on proofs, to generate whole proofs for theorems at once, rather than one step at a time. We combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. As its main contributions, this paper demonstrates for the first time that: (1) Whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. (2) Giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation. (3) We establish a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs. In addition to empirically showing the effectiveness of whole-proof generation, repair, and added context, we show that Baldur improves on the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

1 INTRODUCTION

quality software. For example, CompCert, a C compiler verified using the Coq interactive theorem prover [81], was the only com-

However, the cost of manual formal verification

times as long as the compiler code itself [47].

As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [64] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [4], GPT-f [66], TacticZero [91], Lisa [34], Evariste [42], Diva [20], TacTok [22], and ASTactic [96]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [4, 84], graph neural networks [62], short long-term memory models [20], and language models with the transformer architecture [27, 66].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [7, 14]. Here, we show their remarkable effectiveness for whole proof generation.

The main contributions of our work are:

- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.
- We define a proof repair task and demonstrate that
- We demonstrate empirically on a large benchmark that Baldur, when combined with prior techniques, significantly improve

erva [48], one with 8 billion parameters and another with 62 billion parameters. By contrast, existing tools that use (L)LMs for theorem

1

PRoofster: Automated Formal Verification

Arpan Agrawal
University of Illinois
Urbana-Champaign, IL, USA
arpan2@illinois.edu

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Zhanna Kaufman
University of Massachusetts
Amherst, MA, USA
zhannakaufma@cs.umass.edu

Tom Reichel
University of Illinois
Urbana-Champaign, IL, USA
reichel3@illinois.edu

Shizhuo Zhang
University of Illinois
Urbana-Champaign, IL, USA

Timothy Zhou
University of Illinois
Urbana-Champaign, IL, USA

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Talia Ringer
University of Illinois
Urbana-Champaign, IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

Meanwhile, it took 11 person-years to write the proofs required to verify the seL4 microkernel [17], which represents a tiny fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30]. Some formal verification can even be fully automated via proof synthesis. For example, CoqHammer [4] uses a set of precomputed mathematical facts to attempt to "hammer" out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23], TacTok [7], Diva [6], and Passport [24] learn a predictive model from a corpus of existing proofs and use that model to guide

tioners to use these Coq proof-synthesis tools. For example, of the above-mentioned search-based tools, all but one have neither

---

Passport: Improving Automated Formal Verification Using Identifiers

ALEX SANCHEZ-STERN*, University of Massachusetts Amherst, USA
EMILY FIRST*, University of Massachusetts Amherst, USA
TIMOTHY ZHOU, University of Illinois Urbana-Champaign, USA
ZHANNA KAUFMAN, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

Formally verifying system properties is one of the most effective ways of improving system quality, but its high manual effort requirements often render it prohibitively expensive. Tools that automate formal verification, by learning from proof corpora to suggest proofs, have just begun to show their promise. These tools are effective because of the richness of the data the proof corpora contain. This richness comes from the stylistic conventions followed by communities of proof developers, together with the powerful logical systems beneath proof assistants. However, this richness remains underexploited, with most work thus far focusing on architecture rather than on how to make the most of the proof data.

In this paper, we develop Passport, a fully-automated proof-synthesis tool that systematically explores how to most effectively exploit one aspect of that proof data: identifiers. Passport enriches a predictive Coq model used by proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary

04.10370v2 [cs.PL] 2 Aug 2022

Building a Large Proof Repair Dataset

Tom Reichel ✉
University of Illinois Urbana-Champaign, USA

R. Wesley Henderson ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Touchet ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Gardner* ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Talia Ringer* ✉
University of Illinois Urbana-Champaign, USA

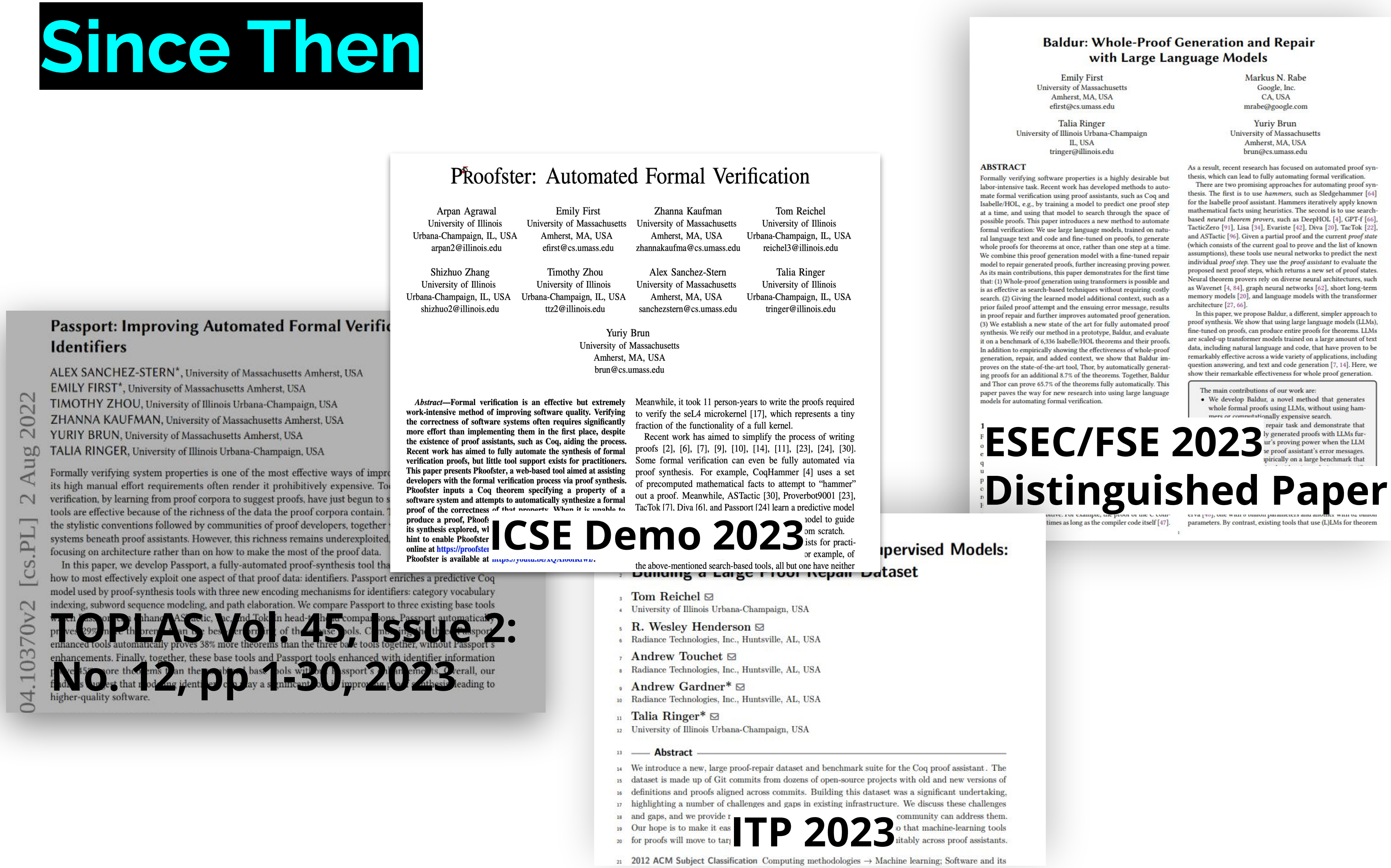
Abstract

We introduce a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from dozens of open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset was a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide recommendations for how the proof assistant community can address them.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its

Neural Automation (Part 3 of 5)

Since Then



04.10370v2 [cs.PL] 2 Aug 2022

Passport: Improving Automated Formal Verific
Identifiers

ALEX SANCHEZ-STERN*, University of Massachusetts Amherst, USA
EMILY FIRST*, University of Massachusetts Amherst, USA
TIMOTHY ZHOU, University of Illinois Urbana-Champaign, USA
ZHANNA KAUFMAN, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

Formally verifying system properties is one of the most effective ways of impro
its high manual effort requirements often render it prohibitively expensive. To
verification, by learning from proof corpora to suggest proofs, have just begun to s
tools are effective because of the richness of the data the proof corpora contain. T
the stylistic conventions followed by communities of proof developers, together
systems beneath proof assistants. However, this richness remains underexploited,
focusing on architecture rather than on how to make the most of the proof data.

In this paper, we develop Passport, a fully-automated proof-synthesis tool tha
how to most effectively exploit one aspect of that proof data: identifiers. Passport enriches a predictive Coq model used by proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary indexing, subword sequence modeling, and path elaboration. We compare Passport to three existing base tools
enhanced tools automatically proves 38% more theorems than the three base tools together, without Passport's enhancements. Finally, together, these base tools and Passport tools enhanced with identifier information
higher-quality software.

TOPLAS Vol. 45, Issue 2: No. 12, pp 1-30, 2023

PRoofster: Automated Formal Verification

Arpan Agrawal
University of Illinois
Urbana-Champaign, IL, USA
arpan2@illinois.edu

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Zhanna Kaufman
University of Massachusetts
Amherst, MA, USA
zhannakaufma@cs.umass.edu

Tom Reichel
University of Illinois
Urbana-Champaign, IL, USA
reichel3@illinois.edu

Shizhuo Zhang
University of Illinois
Urbana-Champaign, IL, USA
shizhuo2@illinois.edu

Timothy Zhou
University of Illinois
Urbana-Champaign, IL, USA
ttz2@illinois.edu

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Talia Ringer
University of Illinois
Urbana-Champaign, IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

***Abstract*—Formal verification is an effective but extremely work-intensive method of improving software quality. Verifying the correctness of software systems often requires significantly more effort than implementing them in the first place, despite the existence of proof assistants, such as Coq, aiding the process. Recent work has aimed to fully automate the synthesis of formal verification proofs, but little tool support exists for practitioners. This paper presents PRoofster, a web-based tool aimed at assisting developers with the formal verification process via proof synthesis. PRoofster inputs a Coq theorem specifying a property of a software system and attempts to automatically synthesize a formal proof of the correctness of that property. When it is unable to produce a proof, PRoofs its synthesis explored, wh hint to enable PRoofster online at https://proofster PRoofster is available at https://youtu.be/xQAi00iKIwI/.**

Meanwhile, it took 11 person-years to write the proofs required to verify the seL4 microkernel [17], which represents a tiny fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30]. Some formal verification can even be fully automated via proof synthesis. For example, CoqHammer [4] uses a set of precomputed mathematical facts to attempt to "hammer" out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23], TacTok [7], Diva [6], and Passport [24] learn a predictive model
nodel to guide
om scratch.
ists for practi-
or example, of
the above-mentioned search-based tools, all but one have neither

ICSE Demo 2023

Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Markus N. Rabe
Google, Inc.
CA, USA
mrabe@google.com

Talia Ringer
University of Illinois Urbana-Champaign
IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Formally verifying software properties is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time, and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language text and code and fine-tuned on proofs, to generate whole proofs for theorems at once, rather than one step at a time. We combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. As its main contributions, this paper demonstrates for the first time that: (1) Whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. (2) Giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation. (3) We establish a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs. In addition to empirically showing the effectiveness of whole-proof generation, repair, and added context, we show that Baldur improves on the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

times as long as the compiler code itself [47].

As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [64] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [4], GPT-f [66], TacticZero [91], Lisa [34], Evariste [42], Diva [20], TacTok [22], and ASTactic [96]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [4, 84], graph neural networks [62], short long-term memory models [20], and language models with the transformer architecture [27, 66].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [7, 14]. Here, we show their remarkable effectiveness for whole proof generation.

The main contributions of our work are:
- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.
- repair task and demonstrate that ly generated proofs with LLMs fur- ur's proving power when the LLM he proof assistant's error messages. npirically on a large benchmark that

parameters. By contrast, existing tools that use (L)LMs for theorem

ESEC/FSE 2023
Distinguished Paper

…pervised Models: Building a Large Proof Repair Dataset

Tom Reichel ✉
University of Illinois Urbana-Champaign, USA
R. Wesley Henderson ✉
Radiance Technologies, Inc., Huntsville, AL, USA
Andrew Touchet ✉
Radiance Technologies, Inc., Huntsville, AL, USA
Andrew Gardner* ✉
Radiance Technologies, Inc., Huntsville, AL, USA
Talia Ringer* ✉
University of Illinois Urbana-Champaign, USA

Abstract

We introduce a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from dozens of open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset was a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide r community can address them. Our hope is to make it eas o that machine-learning tools for proofs will move to tar uitably across proof assistants.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its

ITP 2023

Neural Automation (Part 3 of 5)

Second Project: Proofster

P̊roofster: Automated Formal Verification

Arpan Agrawal
University of Illinois
Urbana-Champaign, IL, USA
arpan2@illinois.edu

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Zhanna Kaufman
University of Massachusetts
Amherst, MA, USA
zhannakaufma@cs.umass.edu

Tom Reichel
University of Illinois
Urbana-Champaign, IL, USA
reichel3@illinois.edu

Shizhuo Zhang
University of Illinois
Urbana-Champaign, IL, USA
shizhuo2@illinois.edu

Timothy Zhou
University of Illinois
Urbana-Champaign, IL, USA
ttz2@illinois.edu

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Talia Ringer
University of Illinois
Urbana-Champaign, IL, USA
tringer@illinois.edu

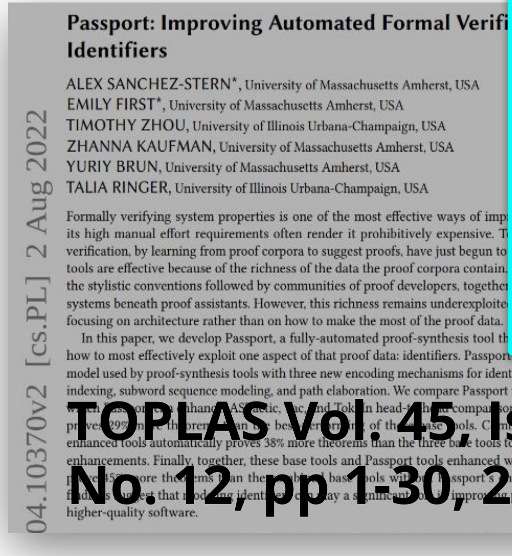
Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

Abstract—**Formal verification is an effective but extremely work-intensive method of improving software quality. Verifying the correctness of software systems often requires significantly more effort than implementing them in the first place, despite the existence of proof assistants, such as Coq, aiding the process. Recent work has aimed to fully automate the synthesis of formal verification proofs, but little tool support exists for practitioners. This paper presents P̊roofster, a web-based tool aimed at assisting developers with the formal verification process via proof synthesis. P̊roofster inputs a Coq theorem specifying a property of a software system and attempts to automatically synthesize a formal proof of the correctness**
produce a proof, P̊roofs
its synthesis explored, wl
hint to enable P̊roofster
online at https://proofster
P̊roofster is available at

Meanwhile, it took 11 person-years to write the proofs required to verify the seL4 microkernel [17], which represents a tiny fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30]. Some formal verification can even be fully automated via proof synthesis. For example, CoqHammer [4] uses a set of precomputed mathematical facts to attempt to "hammer" out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23], TacTok [7], Diva [6], and Passport [24] learn a predictive model
odel to guide
om scratch.
ists for practi-
or example, of
the above-mentioned search-based tools, all but one have neither

ICSE Demo 2023



Passport: Improving Automated Formal Verifi
Identifiers

ALEX SANCHEZ-STERN*, University of Massachusetts Amherst, USA
EMILY FIRST*, University of Massachusetts Amherst, USA
TIMOTHY ZHOU, University of Illinois Urbana-Champaign, USA
ZHANNA KAUFMAN, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

Formally verifying system properties is one of the most effective ways of impr
its high manual effort requirements often render it prohibitively expensive. T
verification, by learning from proof corpora to suggest proofs, have just begun to
tools are effective because of the richness of the data the proof corpora contain.
the stylistic conventions followed by communities of proof developers, together
systems beneath proof assistants. However, this richness remains underexploite
focusing on architecture rather than on how to make the most of the proof data.

In this paper, we develop Passport, a fully-automated proof-synthesis tool th
how to most effectively exploit one aspect of that proof data: identifiers. Passport
model used by proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary indexing, subword sequence modeling, and path elaboration. We compare Passport to three existing base tools
enhanced tools automatically proves 38% more theorems than the three ba e tools together, without Passport's enhancements. Finally, together, these base tools and Passport tools enhanced with identifier information
higher-quality software.

TOPLAS Vol. 45, Issue 2: No. 12, pp 1-30, 2023

Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Markus N. Rabe
Google, Inc.
CA, USA
mrabe@google.com

Talia Ringer
University of Illinois Urbana-Champaign
IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Formally verifying software properties is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time, and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language text and code and fine-tuned on proofs, to generate whole proofs for theorems at once, rather than one step at a time. We combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. As its main contributions, this paper demonstrates for the first time that: (1) Whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. (2) Giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation. (3) We establish a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs. In addition to empirically showing the effectiveness of whole-proof generation, repair, and added context, we show that Baldur improves on the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

times as long as the compiler code itself [47].

As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

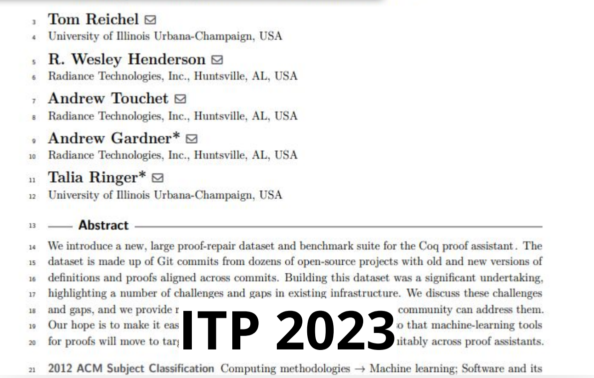
There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [64] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [4], GPT-f [66], TacticZero [91], Lisa [34], Evariste [42], Diva [20], TacTok [22], and ASTactic [96]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [4, 84], graph neural networks [62], short long-term memory models [20], and language models with the transformer architecture [27, 66].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [7, 14]. Here, we show their remarkable effectiveness for whole proof generation.

The main contributions of our work are:
- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.
repair task and demonstrate that
ly generated proofs with LLMs fur-
ur's proving power when the LLM
e proof assistant's error messages.
pirically on a large benchmark that

parameters. By contrast, existing tools that use (L)LMs for theorem

ESEC/FSE 2023
Distinguished Paper



pervised Models:
aset

Tom Reichel ✉
University of Illinois Urbana-Champaign, USA
R. Wesley Henderson ✉
Radiance Technologies, Inc., Huntsville, AL, USA
Andrew Touchet ✉
Radiance Technologies, Inc., Huntsville, AL, USA
Andrew Gardner* ✉
Radiance Technologies, Inc., Huntsville, AL, USA
Talia Ringer* ✉
University of Illinois Urbana-Champaign, USA

Abstract

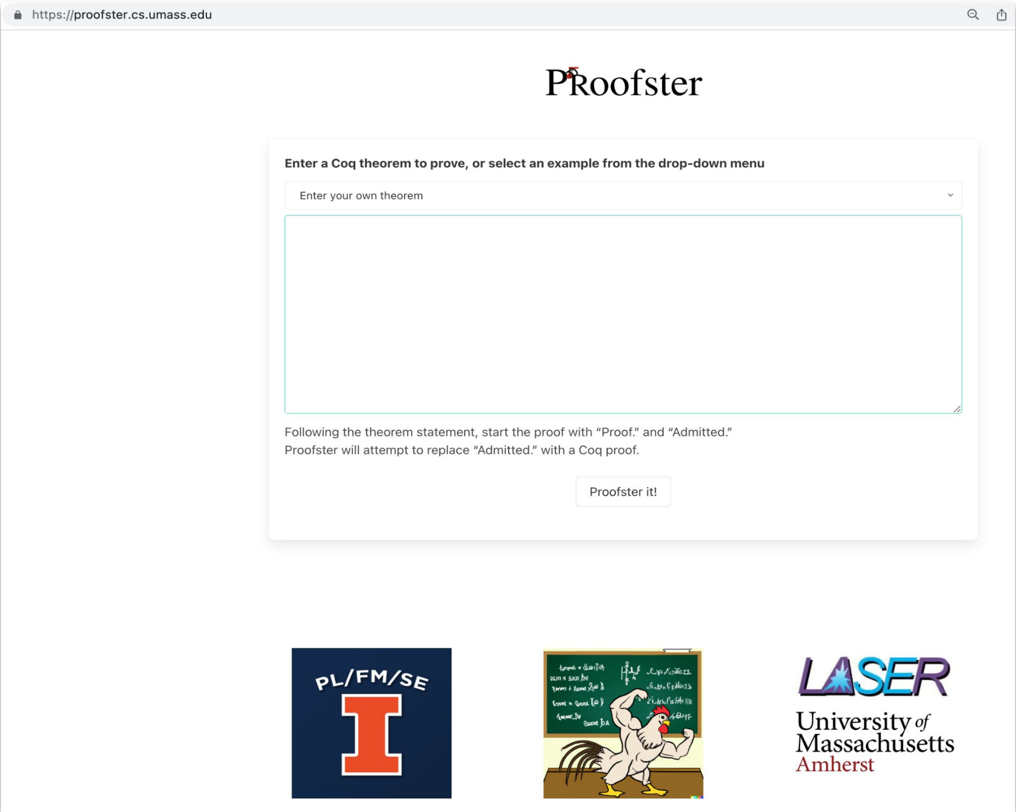
We introduce a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from dozens of open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset was a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide r
community can address them.
Our hope is to make it eas
o that machine-learning tools
for proofs will move to tar
itably across proof assistants.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its

ITP 2023

Neural Automation (Part 3 of 5)

Second Project: Proofster



https://proofster.cs.umass.edu

PRoofster

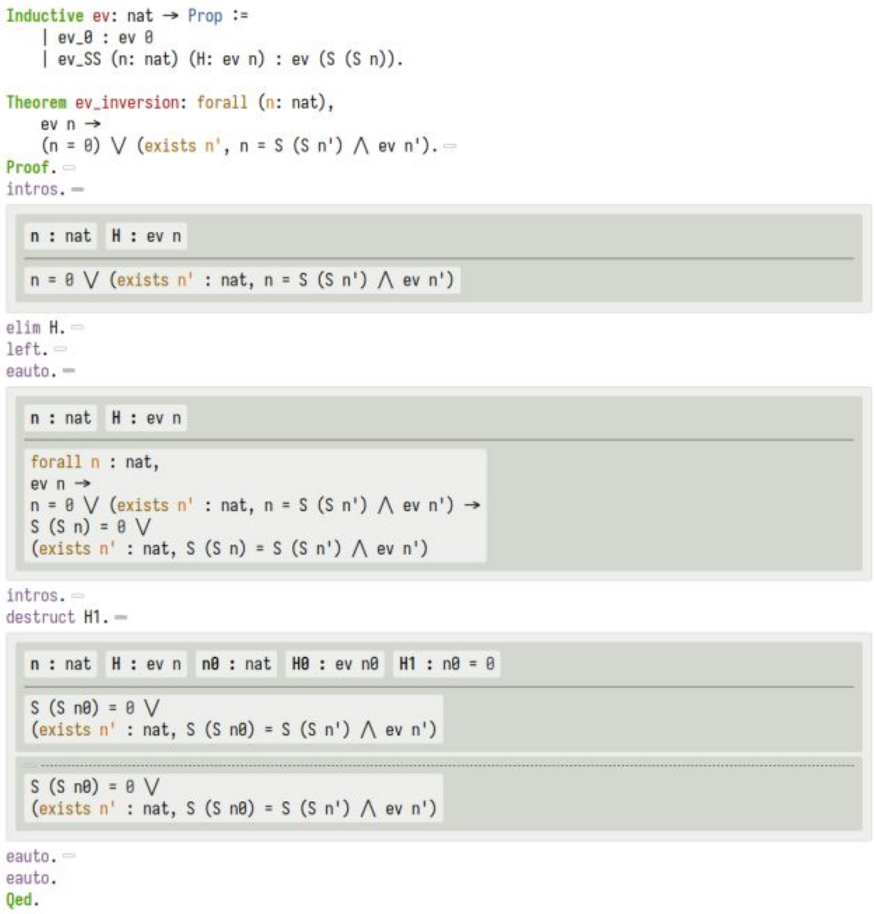
Enter a Coq theorem to prove, or select an example from the drop-down menu

Following the theorem statement, start the proof with "Proof." and "Admitted."
Proofster will attempt to replace "Admitted." with a Coq proof.

https://proofster.cs.umass.edu/ (web)
https://github.com/agrarpan/coq-synthesis (plugin)

Neural Automation (Part 3 of 5)

Second Project: Proofster

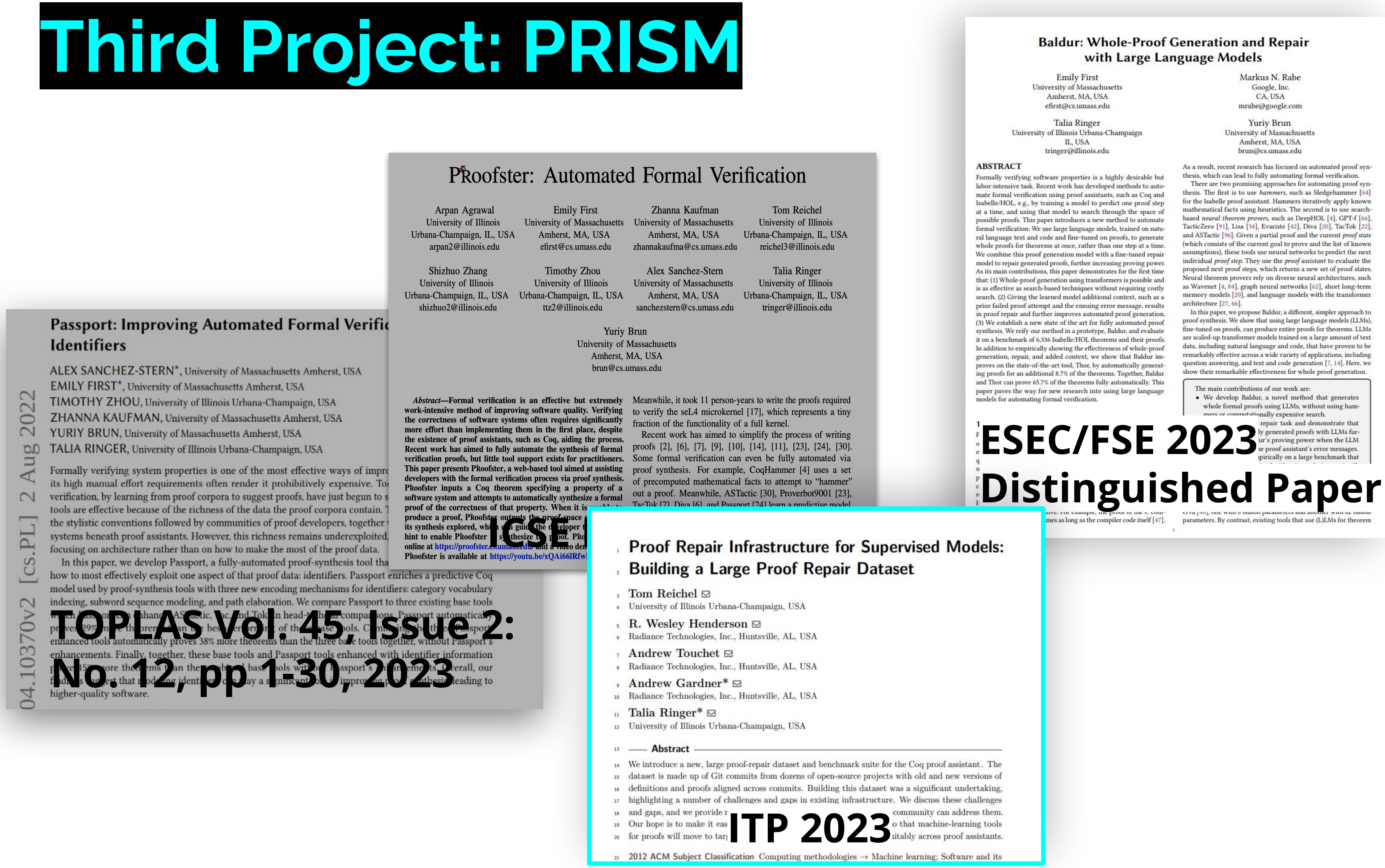


```
Inductive ev: nat → Prop :=
    | ev_0 : ev 0
    | ev_SS (n: nat) (H: ev n) : ev (S (S n)).

Theorem ev_inversion: forall (n: nat),
    ev n →
    (n = 0) ∨ (exists n', n = S (S n') ∧ ev n').
Proof.
intros.

  n : nat   H : ev n
  ─────────────────────────────────────────────
  n = 0 ∨ (exists n' : nat, n = S (S n') ∧ ev n')

elim H.
left.
eauto.

  n : nat   H : ev n
  ─────────────────────────────────────────────
  forall n : nat,
  ev n →
  n = 0 ∨ (exists n' : nat, n = S (S n') ∧ ev n') →
  S (S n) = 0 ∨
  (exists n' : nat, S (S n) = S (S n') ∧ ev n')

intros.
destruct H1.

  n : nat   H : ev n   n0 : nat   H0 : ev n0   H1 : n0 = 0
  ─────────────────────────────────────────────
  S (S n0) = 0 ∨
  (exists n' : nat, S (S n0) = S (S n') ∧ ev n')
  ·············································
  S (S n0) = 0 ∨
  (exists n' : nat, S (S n0) = S (S n') ∧ ev n')

eauto.
eauto.
Qed.
```

Uses https://github.com/cpitclaudel/alectryon

Neural Automation (Part 3 of 5)

Third Project: PRISM



Passport: Improving Automated Formal Verific
Identifiers

ALEX SANCHEZ-STERN*, University of Massachusetts Amherst, USA
EMILY FIRST*, University of Massachusetts Amherst, USA
TIMOTHY ZHOU, University of Illinois Urbana-Champaign, USA
ZHANNA KAUFMAN, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

Formally verifying system properties is one of the most effective ways of impro
its high manual effort requirements often render it prohibitively expensive. To
verification, by learning from proof corpora to suggest proofs, have just begun to s
tools are effective because of the richness of the data the proof corpora contain. T
the stylistic conventions followed by communities of proof developers, together w
systems beneath proof assistants. However, this richness remains underexploited,
focusing on architecture rather than on how to make the most of the proof data.

In this paper, we develop Passport, a fully-automated proof-synthesis tool tha
how to most effectively exploit one aspect of that proof data: identifiers. Passport enriches a predictive Coq model used by proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary indexing, subword sequence modeling, and path elaboration. We compare Passport to three existing base tools
enhanced tools automatically proves 38% more theorems than the three base tools together, without Passport's enhancements. Finally, together, these base tools and Passport tools enhanced with identifier information
higher-quality software.

04.10370v2 [cs.PL] 2 Aug 2022

TOPLAS Vol. 45, Issue 2: No.12, pp 1-30, 2023

Proofster: Automated Formal Verification

Arpan Agrawal
University of Illinois
Urbana-Champaign, IL, USA
arpan2@illinois.edu

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Zhanna Kaufman
University of Massachusetts
Amherst, MA, USA
zhannakaufma@cs.umass.edu

Tom Reichel
University of Illinois
Urbana-Champaign, IL, USA
reichel3@illinois.edu

Shizhuo Zhang
University of Illinois
Urbana-Champaign, IL, USA
shizhuo2@illinois.edu

Timothy Zhou
University of Illinois
Urbana-Champaign, IL, USA
ttz2@illinois.edu

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Talia Ringer
University of Illinois
Urbana-Champaign, IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

Abstract—**Formal verification is an effective but extremely work-intensive method of improving software quality. Verifying the correctness of software systems often requires significantly more effort than implementing them in the first place, despite the existence of proof assistants, such as Coq, aiding the process. Recent work has aimed to fully automate the synthesis of formal verification proofs, but little tool support exists for practitioners. This paper presents Proofster, a web-based tool aimed at assisting developers with the formal verification process via proof synthesis. Proofster inputs a Coq theorem specifying a property of a software system and attempts to automatically synthesize a formal proof of the correctness of that property. When it is**
produce a proof, Proofster outputs the proof-space s
its synthesis explored, whi
hint to enable Proofster
online at https://proofster.cs.umass.edu/ and a video de
Proofster is available at https://youtu.be/xQAi66lRfw

Meanwhile, it took 11 person-years to write the proofs required to verify the seL4 microkernel [17], which represents a tiny fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30]. Some formal verification can even be fully automated via proof synthesis. For example, CoqHammer [4] uses a set of precomputed mathematical facts to attempt to "hammer" out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23], TacTok [7], Diva [6], and Passport [24] learn a predictive model

ICSE

Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First
University of Massachusetts
Amherst, MA, USA
efirst@cs.umass.edu

Markus N. Rabe
Google, Inc.
CA, USA
mrabe@google.com

Talia Ringer
University of Illinois Urbana-Champaign
IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

ABSTRACT

Formally verifying software properties is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time, and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language text and code and fine-tuned on proofs, to generate whole proofs for theorems at once, rather than one step at a time. We combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. As its main contributions, this paper demonstrates for the first time that: (1) Whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. (2) Giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation. (3) We establish a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs. In addition to empirically showing the effectiveness of whole-proof generation, repair, and added context, we show that Baldur improves on the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

mes as long as the compiler code itself [47].

As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [64] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [4], GPT-f [66], TacticZero [91], Lisa [34], Evariste [42], Diva [20], TacTok [22], and ASTactic [96]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [4, 84], graph neural networks [62], short long-term memory models [20], and language models with the transformer architecture [27, 66].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [7, 14]. Here, we show their remarkable effectiveness for whole proof generation.

The main contributions of our work are:
- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.

repair task and demonstrate that
ly generated proofs with LLMs fur-
ur's proving power when the LLM
e proof assistant's error messages.
pirically on a large benchmark that

parameters. By contrast, existing tools that use (L)LMs for theorem

ESEC/FSE 2023 Distinguished Paper

Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset

Tom Reichel ✉
University of Illinois Urbana-Champaign, USA

R. Wesley Henderson ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Touchet ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Gardner* ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Talia Ringer* ✉
University of Illinois Urbana-Champaign, USA

Abstract

We introduce a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from dozens of open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset was a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide r
community can address them.
Our hope is to make it eas
o that machine-learning tools
for proofs will move to tar
itably across proof assistants.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its

ITP 2023

Neural Automation (Part 3 of 5)

Third Project: PRISM

- **Dataset** for **proof repair** models for Coq
- **Actual proof repairs** by proof engineers
- **Collaboration with Radiance**
- **Massive infrastructure undertaking**
  - Building many different projects
  - … with many different Coq versions
  - … for many different commits
  - … and aligning data across commit pairs
- **First repair model trained**
- **Evaluation WIP**

Neural Automation (Part 3 of 5)

Fourth Project: Baldur

Passport: Improving Automated Formal Verific… Identifiers

ALEX SANCHEZ-STERN*, University of Massachusetts Amherst, USA
EMILY FIRST*, University of Massachusetts Amherst, USA
TIMOTHY ZHOU, University of Illinois Urbana-Champaign, USA
ZHANNA KAUFMAN, University of Massachusetts Amherst, USA
YURIY BRUN, University of Massachusetts Amherst, USA
TALIA RINGER, University of Illinois Urbana-Champaign, USA

Formally verifying system properties is one of the most effective ways of impro… its high manual effort requirements often render it prohibitively expensive. To… verification, by learning from proof corpora to suggest proofs, have just begun to s… tools are effective because of the richness of the data the proof corpora contain. T… the stylistic conventions followed by communities of proof developers, together … systems beneath proof assistants. However, this richness remains underexploited, … focusing on architecture rather than on how to make the most of the proof data.

In this paper, we develop Passport, a fully-automated proof-synthesis tool tha… how to most effectively exploit one aspect of that proof data: identifiers. Passport enriches a predictive Coq model used by proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary indexing, subword sequence modeling, and path elaboration. We compare Passport to three existing base tools which Passport can enhance: ASTactic, Tac, and Tok. In head-to-head comparisons, Passport automatically proves 29% more theorems than the best-performing of the base tools. Combining the three Passport-enhanced tools automatically proves 38% more theorems than the three base tools together, without Passport's enhancements. Finally, together, these base tools and Passport tools enhanced with identifier information prove 45% more theorems than the combined base tools without Passport's enhancements. Overall, our findings suggest that modeling identifiers can play a significant role in improving proof synthesis, leading to higher-quality software.

arXiv:2204.10370v2 [cs.PL] 2 Aug 2022

TOPLAS Vol. 45, Issue 2: No. 12, pp 1-30, 2023

PRoofster: Automated Formal Verification

Arpan Agrawal University of Illinois Urbana-Champaign, IL, USA arpan2@illinois.edu	Emily First University of Massachusetts Amherst, MA, USA efirst@cs.umass.edu	Zhanna Kaufman University of Massachusetts Amherst, MA, USA zhannakaufma@cs.umass.edu	Tom Reichel University of Illinois Urbana-Champaign, IL, USA reichel3@illinois.edu
Shizhuo Zhang University of Illinois Urbana-Champaign, IL, USA shizhuo2@illinois.edu	Timothy Zhou University of Illinois Urbana-Champaign, IL, USA ttz2@illinois.edu	Alex Sanchez-Stern University of Massachusetts Amherst, MA, USA sanchezstern@cs.umass.edu	Talia Ringer University of Illinois Urbana-Champaign, IL, USA tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

***Abstract*—Formal verification is an effective but extremely work-intensive method of improving software quality. Verifying the correctness of software systems often requires significantly more effort than implementing them in the first place, despite the existence of proof assistants, such as Coq, aiding the process. Recent work has aimed to fully automate the synthesis of formal verification proofs, but little tool support exists for practitioners. This paper presents PRoofster, a web-based tool aimed at assisting developers with the formal verification process via proof synthesis. PRoofster inputs a Coq theorem specifying a property of a software system and attempts to automatically synthesize a formal proof of the correctness of that property. When it is unable to produce a proof, PRoofster outputs the proof-space se… its synthesis explored, wh… guide the developer t… hint to enable PRoofster … synthesize the proof. PRo… online at https://proofster.cs.umass.edu/ and a video demo… PRoofster is available at https://youtu.be/xQAi66lRfwI/…**

Meanwhile, it took 11 person-years to write the proofs required to verify the seL4 microkernel [17], which represents a tiny fraction of the functionality of a full kernel.

Recent work has aimed to simplify the process of writing proofs [2], [6], [7], [9], [10], [14], [11], [23], [24], [30]. Some formal verification can even be fully automated via proof synthesis. For example, CoqHammer [4] uses a set of precomputed mathematical facts to attempt to "hammer" out a proof. Meanwhile, ASTactic [30], Proverbot9001 [23], TacTok [7], Diva [6], and Passport [24] learn a predictive model

ICSE

Proof Repair Infrastructure for Supervised Models: Building a Large Proof Repair Dataset

Tom Reichel ✉
University of Illinois Urbana-Champaign, USA

R. Wesley Henderson ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Touchet ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Andrew Gardner* ✉
Radiance Technologies, Inc., Huntsville, AL, USA

Talia Ringer* ✉
University of Illinois Urbana-Champaign, USA

Abstract

We introduce a new, large proof-repair dataset and benchmark suite for the Coq proof assistant. The dataset is made up of Git commits from dozens of open-source projects with old and new versions of definitions and proofs aligned across commits. Building this dataset was a significant undertaking, highlighting a number of challenges and gaps in existing infrastructure. We discuss these challenges and gaps, and we provide recommendations for how the proof assistant community can address them. Our hope is to make it easi… to build datasets and benchmark suites so that machine-learning tools for proofs will move to targ… the tasks that matter … and do so equitably across proof assistants.

2012 ACM Subject Classification Computing methodologies → Machine learning; Software and its

ITP 2023

Baldur: Whole-Proof Generation and Repair with Large Language Models

Emily First University of Massachusetts Amherst, MA, USA efirst@cs.umass.edu	Markus N. Rabe Google, Inc. CA, USA mrabe@google.com
Talia Ringer University of Illinois Urbana-Champaign IL, USA tringer@illinois.edu	Yuriy Brun University of Massachusetts Amherst, MA, USA brun@cs.umass.edu

ABSTRACT

Formally verifying software properties is a highly desirable but labor-intensive task. Recent work has developed methods to automate formal verification using proof assistants, such as Coq and Isabelle/HOL, e.g., by training a model to predict one proof step at a time, and using that model to search through the space of possible proofs. This paper introduces a new method to automate formal verification: We use large language models, trained on natural language text and code and fine-tuned on proofs, to generate whole proofs for theorems at once, rather than one step at a time. We combine this proof generation model with a fine-tuned repair model to repair generated proofs, further increasing proving power. As its main contributions, this paper demonstrates for the first time that: (1) Whole-proof generation using transformers is possible and is as effective as search-based techniques without requiring costly search. (2) Giving the learned model additional context, such as a prior failed proof attempt and the ensuing error message, results in proof repair and further improves automated proof generation. (3) We establish a new state of the art for fully automated proof synthesis. We reify our method in a prototype, Baldur, and evaluate it on a benchmark of 6,336 Isabelle/HOL theorems and their proofs. In addition to empirically showing the effectiveness of whole-proof generation, repair, and added context, we show that Baldur improves on the state-of-the-art tool, Thor, by automatically generating proofs for an additional 8.7% of the theorems. Together, Baldur and Thor can prove 65.7% of the theorems fully automatically. This paper paves the way for new research into using large language models for automating formal verification.

piler is more than three times as long as the compiler code itself [47].

As a result, recent research has focused on automated proof synthesis, which can lead to fully automating formal verification.

There are two promising approaches for automating proof synthesis. The first is to use *hammers*, such as Sledgehammer [64] for the Isabelle proof assistant. Hammers iteratively apply known mathematical facts using heuristics. The second is to use search-based *neural theorem provers*, such as DeepHOL [4], GPT-f [66], TacticZero [91], Lisa [34], Evariste [42], Diva [20], TacTok [22], and ASTactic [96]. Given a partial proof and the current *proof state* (which consists of the current goal to prove and the list of known assumptions), these tools use neural networks to predict the next individual *proof step*. They use the *proof assistant* to evaluate the proposed next proof steps, which returns a new set of proof states. Neural theorem provers rely on diverse neural architectures, such as Wavenet [4, 84], graph neural networks [62], short long-term memory models [20], and language models with the transformer architecture [27, 66].

In this paper, we propose Baldur, a different, simpler approach to proof synthesis. We show that using large language models (LLMs), fine-tuned on proofs, can produce entire proofs for theorems. LLMs are scaled-up transformer models trained on a large amount of text data, including natural language and code, that have proven to be remarkably effective across a wide variety of applications, including question answering, and text and code generation [7, 14]. Here, we show their remarkable effectiveness for whole proof generation.

The main contributions of our work are:
- We develop Baldur, a novel method that generates whole formal proofs using LLMs, without using hammers or computationally expensive search.

parameters. By contrast, existing tools that use (L)LMs for theorem

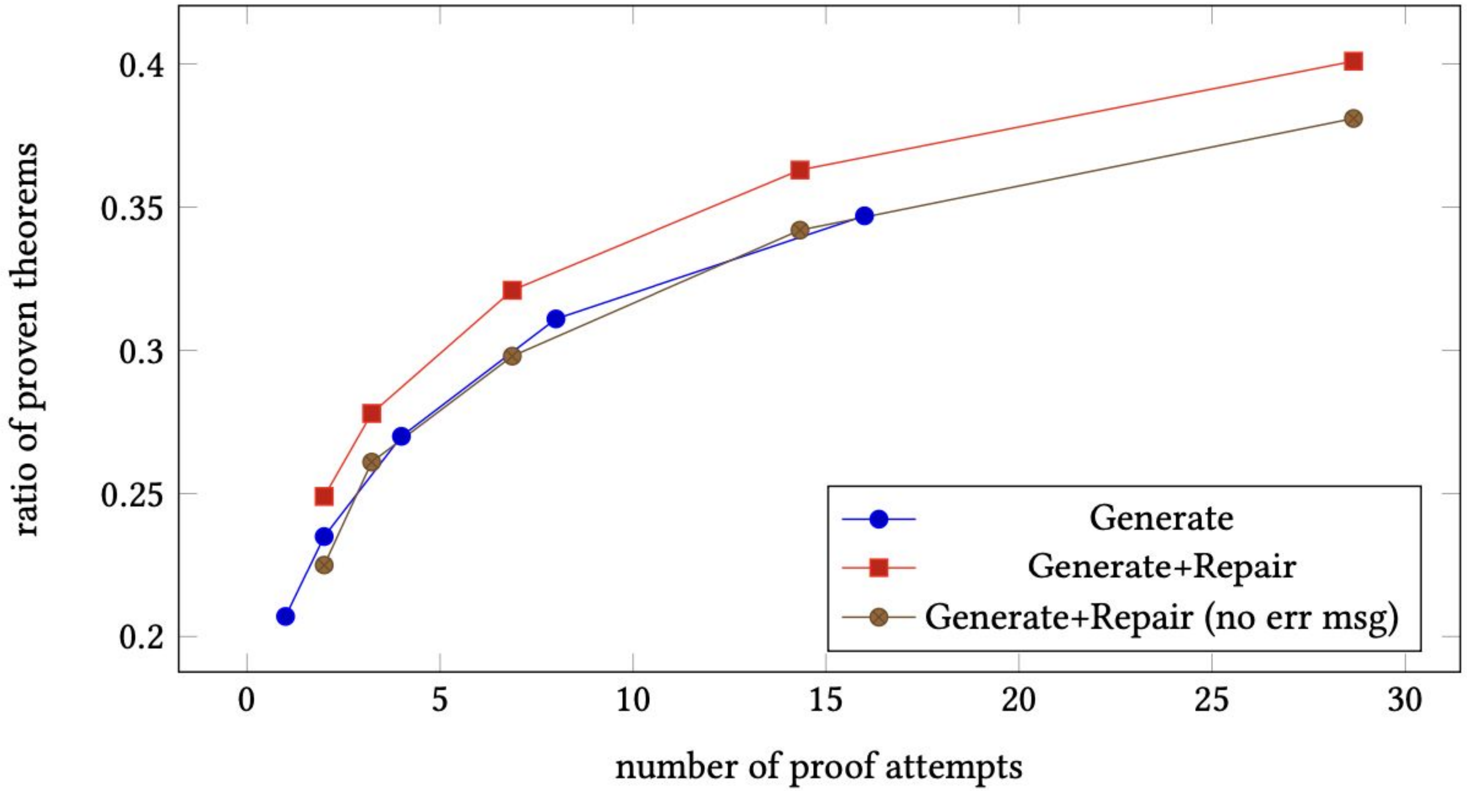
ESEC/FSE 2023 Distinguished Paper

Neural Automation (Part 3 of 5)

Fourth Project: Baldur

- Using an **LLM**, one could, conceivably, synthesize **entire proofs at once**.
- Collaborating with Google, we fine-tuned the Minerva model to synthesize proofs in Isabelle/HOL
- Evaluated on PISA dataset (theorems in Isabelle/HOL)

Fourth Project: Baldur



Neural Automation (Part 3 of 5)

Fourth Project: Baldur

- Baldur (without repair) can **synthesize whole proofs** for **47.9%** of the theorems, whereas search-based approaches prove **39.0%**.
- Baldur can **repair its own erroneous proof attempts** using the error message from the proof assistant, proving another **1.5%**.
- **Diversity continues to help**. Together with Thor, a tool that combines a model, search, and a hammer, Baldur can prove **65.7%**.

Neural Automation (Part 3 of 5)

Neural automation:

- *un*predictable
- *not* dependable
- *not* understandable
+ *not very* limited in scope
+ takes *little* expertise to extend

Neural Automation (Part 3 of 5)

Checking the Proof

Small Logical Kernel

Tactics

Domain-Specific Heuristics

Proof Transformations

Producing the Proof

Scary Programs

Neural Automation (Part 3 of 5)

Checking the Proof

Small Logical Kernel

Tactics

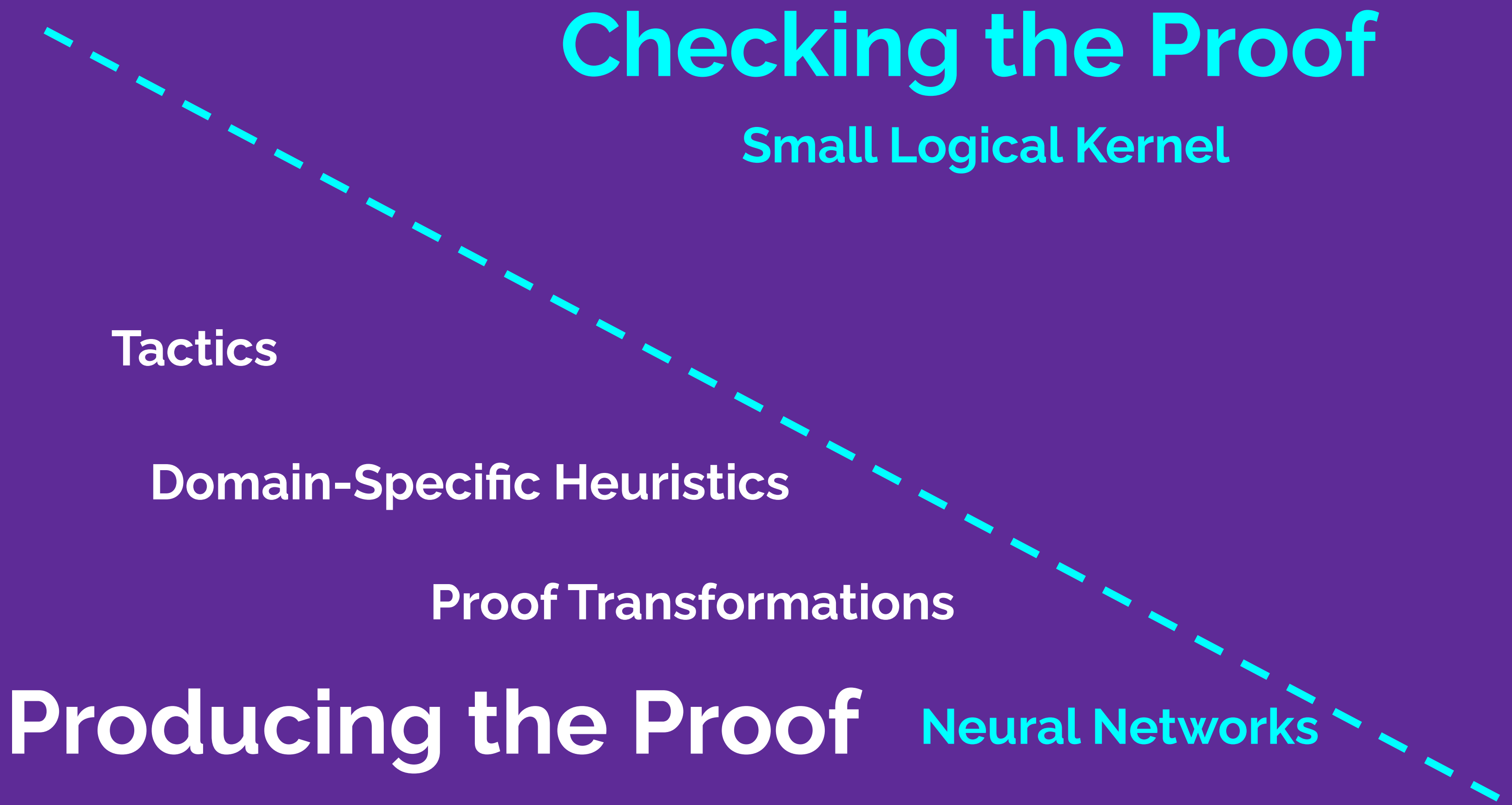
Domain-Specific Heuristics

Proof Transformations

Producing the Proof ChatGPT

Neural Automation (Part 3 of 5)

Already Neurosymbolic



Checking the Proof

Small Logical Kernel

Tactics

Domain-Specific Heuristics

Proof Transformations

Producing the Proof

Neural Networks

But we want even more of the benefits of both kinds of automation.

1. Proof Assistants
2. Symbolic Automation
3. Neural Automation
4. **Building Bridges**
5. Opportunities

Observation 1: We can do fairly well sometimes without **search**. Maybe we can use search at a **higher level** than before and get further returns?

Building Bridges (Part 4 of 5)

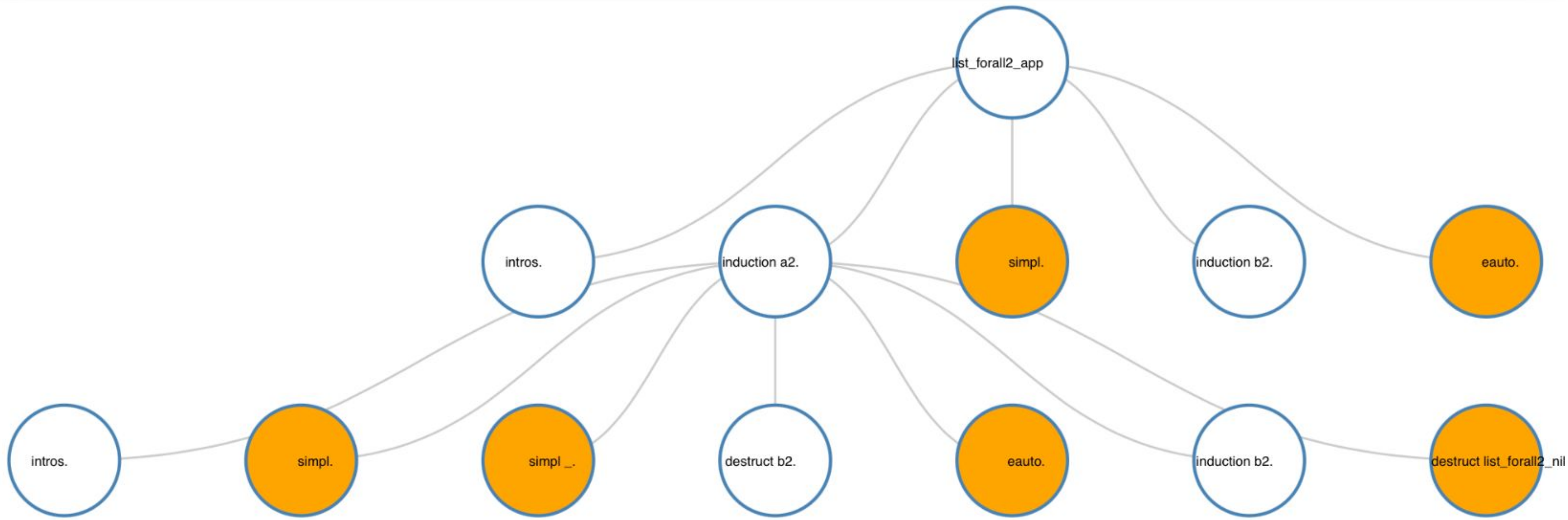
One idea: **Move the search process *up* in abstraction.**

Building Bridges (Part 4 of 5)

One idea: Move the search process *up* in abstraction.

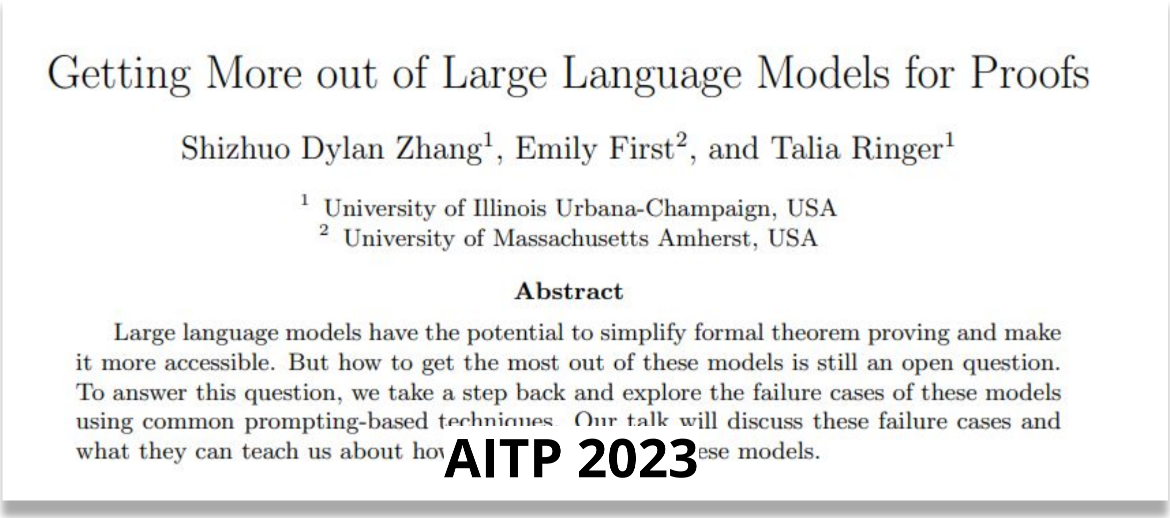
Building Bridges (Part 4 of 5)

Proof Search



Building Bridges (Part 4 of 5)

Conversational Action Search



Getting More out of Large Language Models for Proofs

Shizhuo Dylan Zhang[1], Emily First[2], and Talia Ringer[1]

[1] University of Illinois Urbana-Champaign, USA
[2] University of Massachusetts Amherst, USA

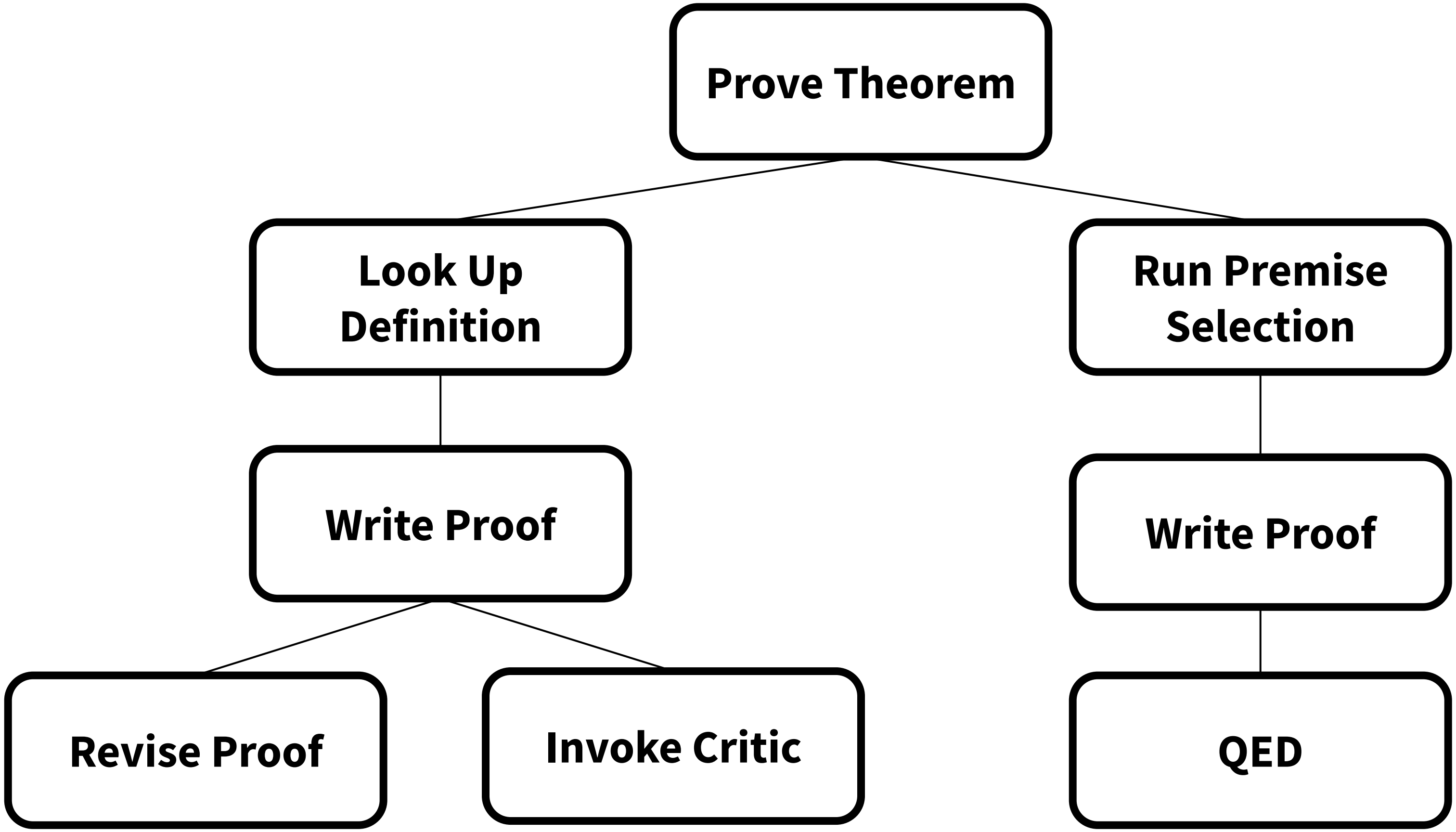
Abstract

Large language models have the potential to simplify formal theorem proving and make it more accessible. But how to get the most out of these models is still an open question. To answer this question, we take a step back and explore the failure cases of these models using common prompting-based techniques. Our talk will discuss these failure cases and what they can teach us about ho [illegible] ese models.

AITP 2023

Building Bridges (Part 4 of 5)

Conversational Action Search



Prove Theorem

Look Up Definition

Run Premise Selection

Write Proof

Write Proof

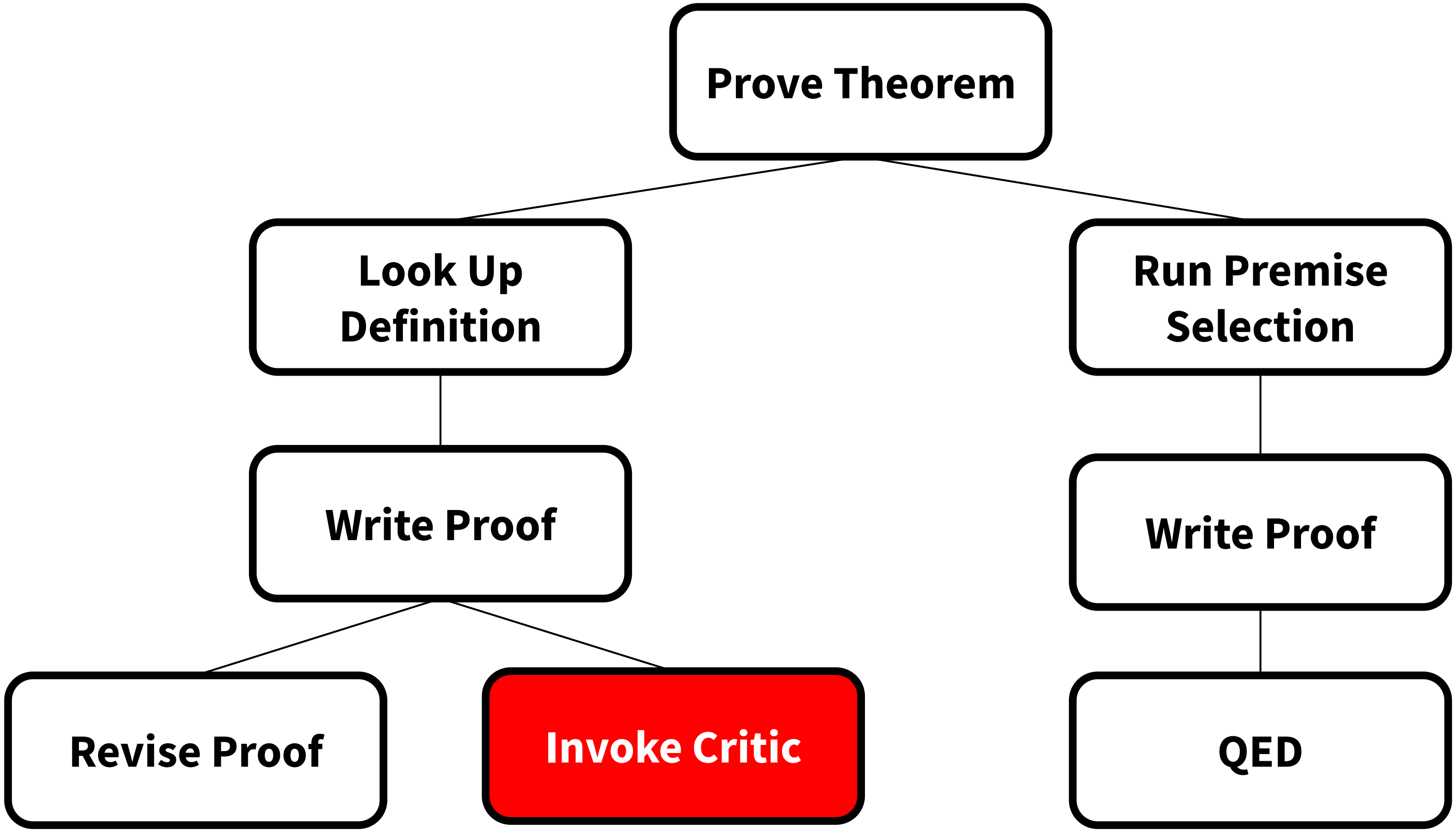
Revise Proof

Invoke Critic

QED

Building Bridges (Part 4 of 5)

Conversational Action Search



Prove Theorem

Look Up Definition

Run Premise Selection

Write Proof

Write Proof

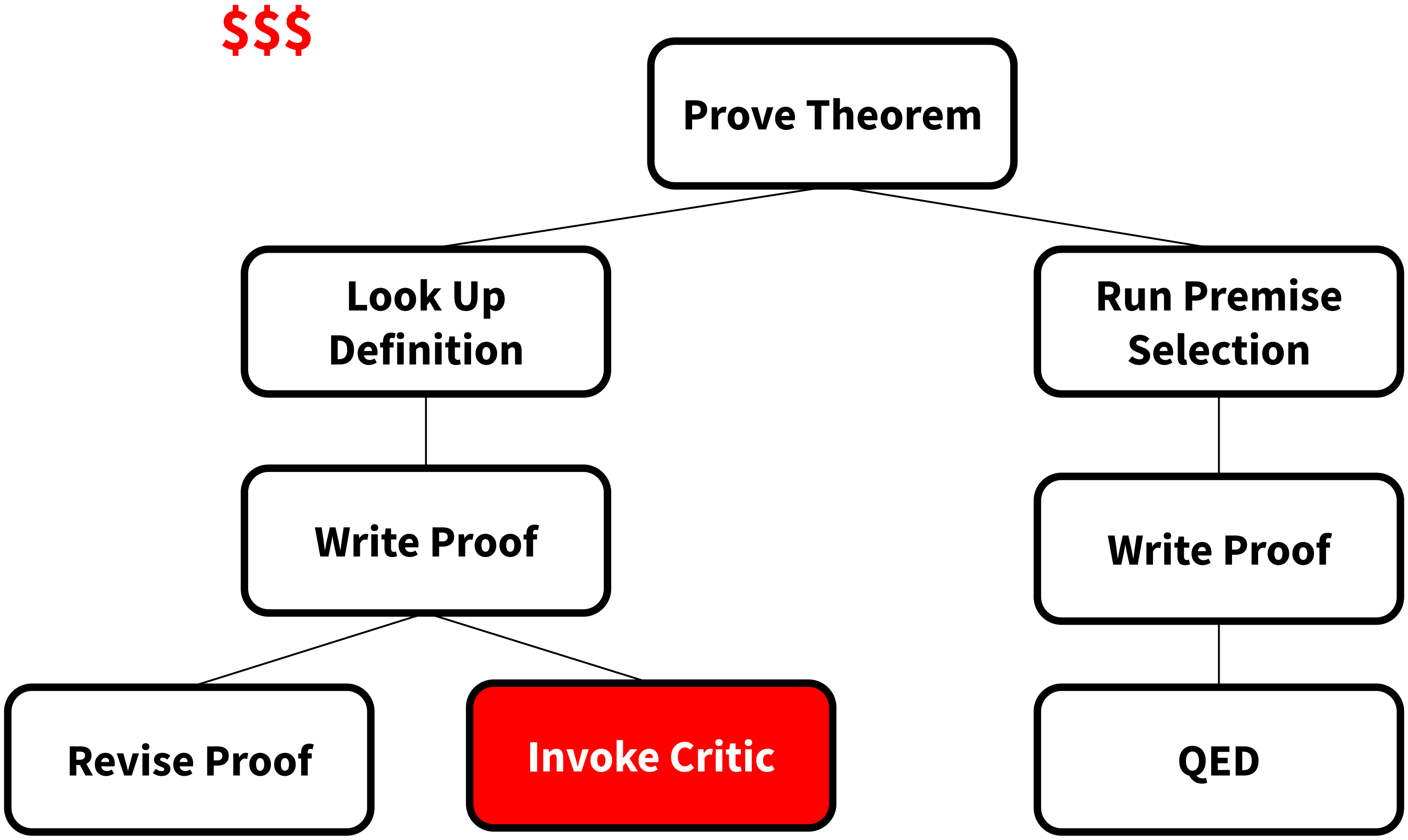
Revise Proof

Invoke Critic

QED

Building Bridges (Part 4 of 5)

Conversational Action Search



$$$

Prove Theorem

Look Up Definition

Run Premise Selection

Write Proof

Write Proof

Revise Proof

Invoke Critic

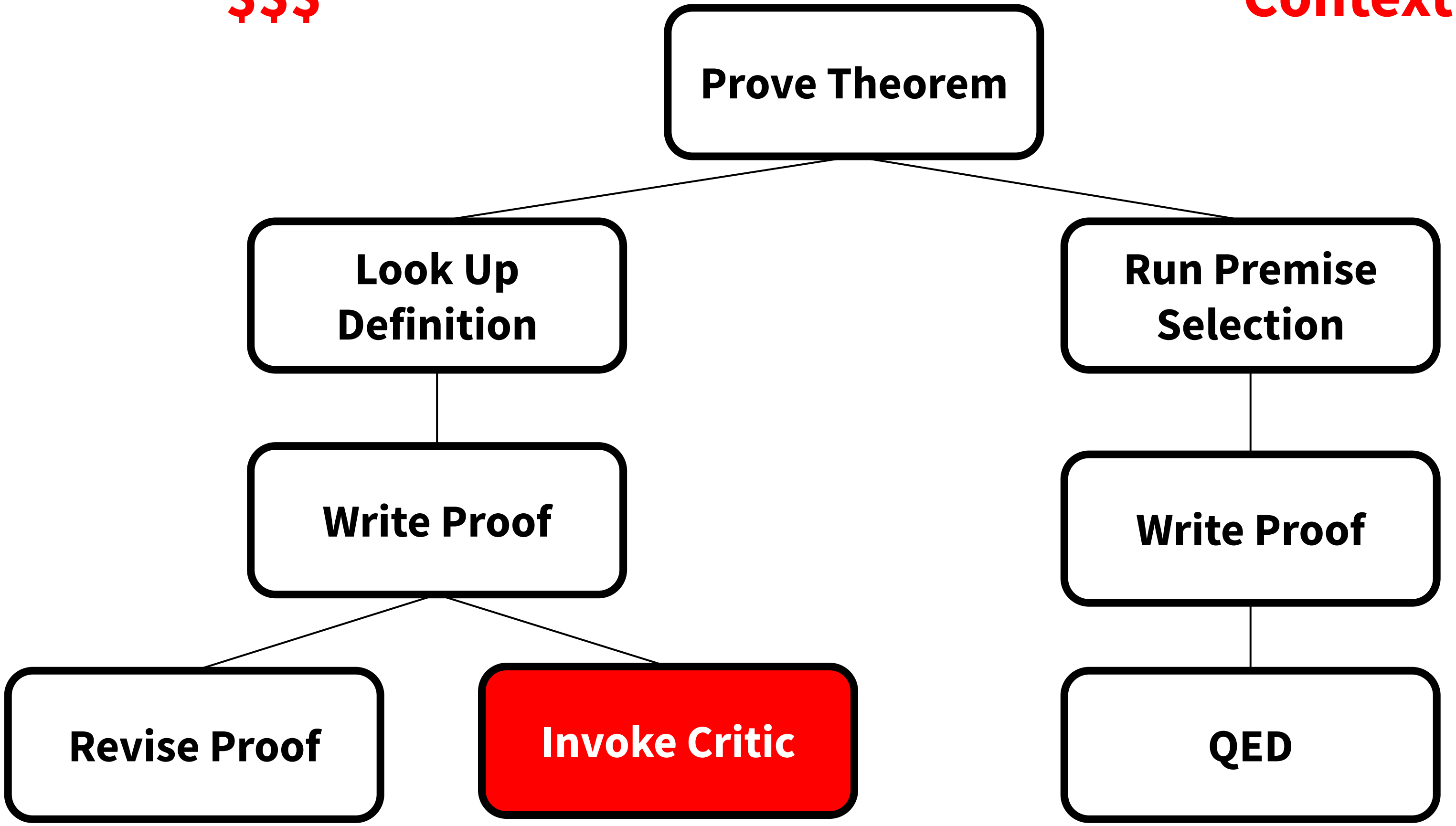
QED

Building Bridges (Part 4 of 5)

Conversational Action Search

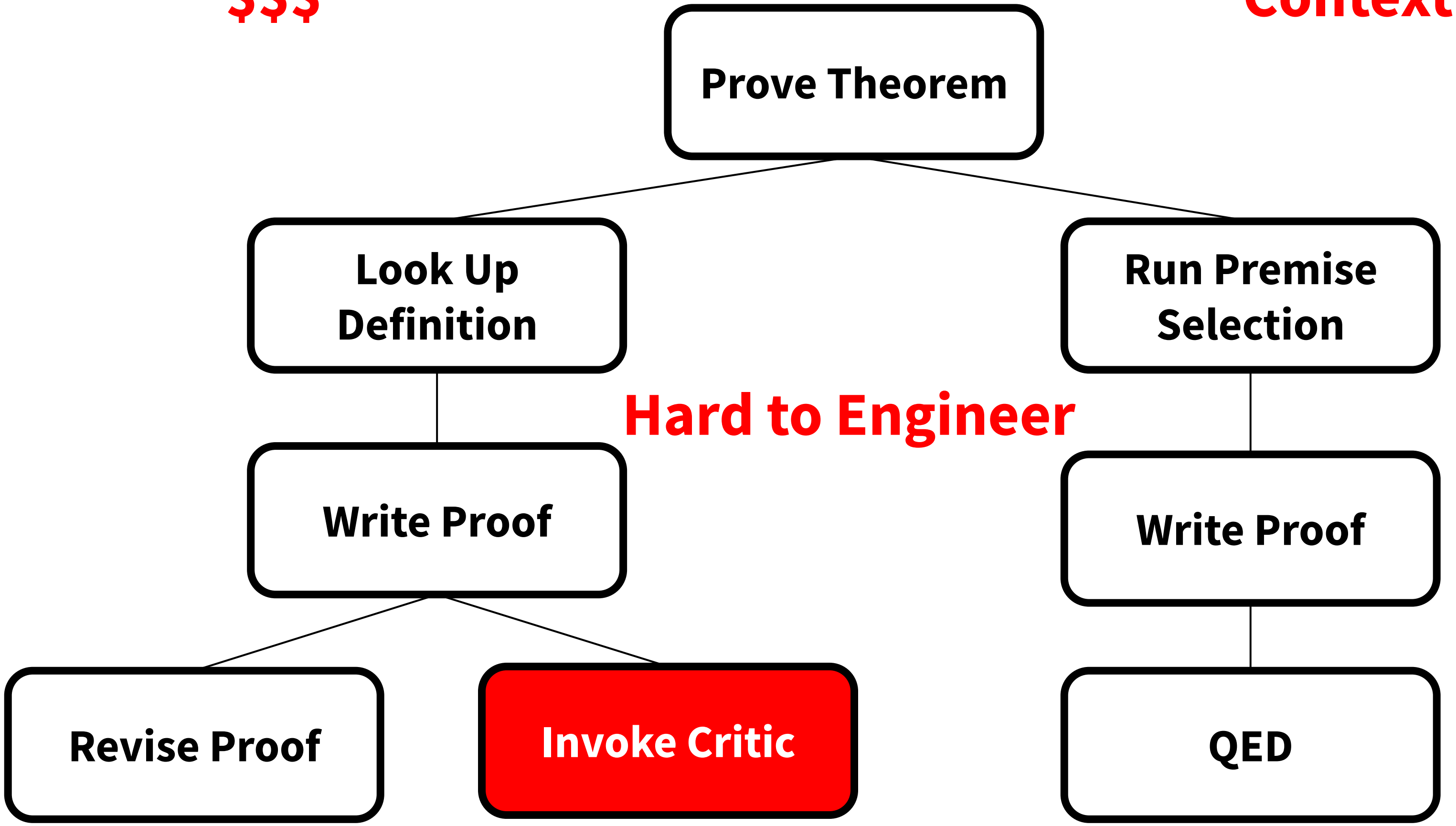
$$$

Context Size



Building Bridges (Part 4 of 5)

Conversational Action Search



$$$

Context Size

Hard to Engineer

Building Bridges (Part 4 of 5)

Conversational Action Search

Promising Results

Observation 2: Diversity in **models** helps, and **diversity** in **techniques** appears to help, too. Let’s keep taking advantage of that.

Ongoing: Best of both worlds for proof repair, too.

Building Bridges (Part 4 of 5)

Neural proof repair: good for **large**, **repetitive**, mostly **syntactic** changes at the **tactic** level, like from updating Coq versions

Symbolic proof repair: good for well-scoped semantic changes at the term level, like those described by equivalences

Building Bridges (Part 4 of 5)

Neural proof repair: good for **large**, **repetitive**, mostly **syntactic** changes at the **tactic** level, like from updating Coq versions

Symbolic proof repair: good for **well-scoped semantic** changes at the **term** level, like those described by equivalences

Building Bridges (Part 4 of 5)

Neurosymbolic proof repair:

good for large, repetitive, mostly syntactic changes at the tactic level, like from updating Coq versions? And also, good for well-scoped semantic changes at the term level, like those described by equivalences? Better than the sum of its parts?

Building Bridges (Part 4 of 5)

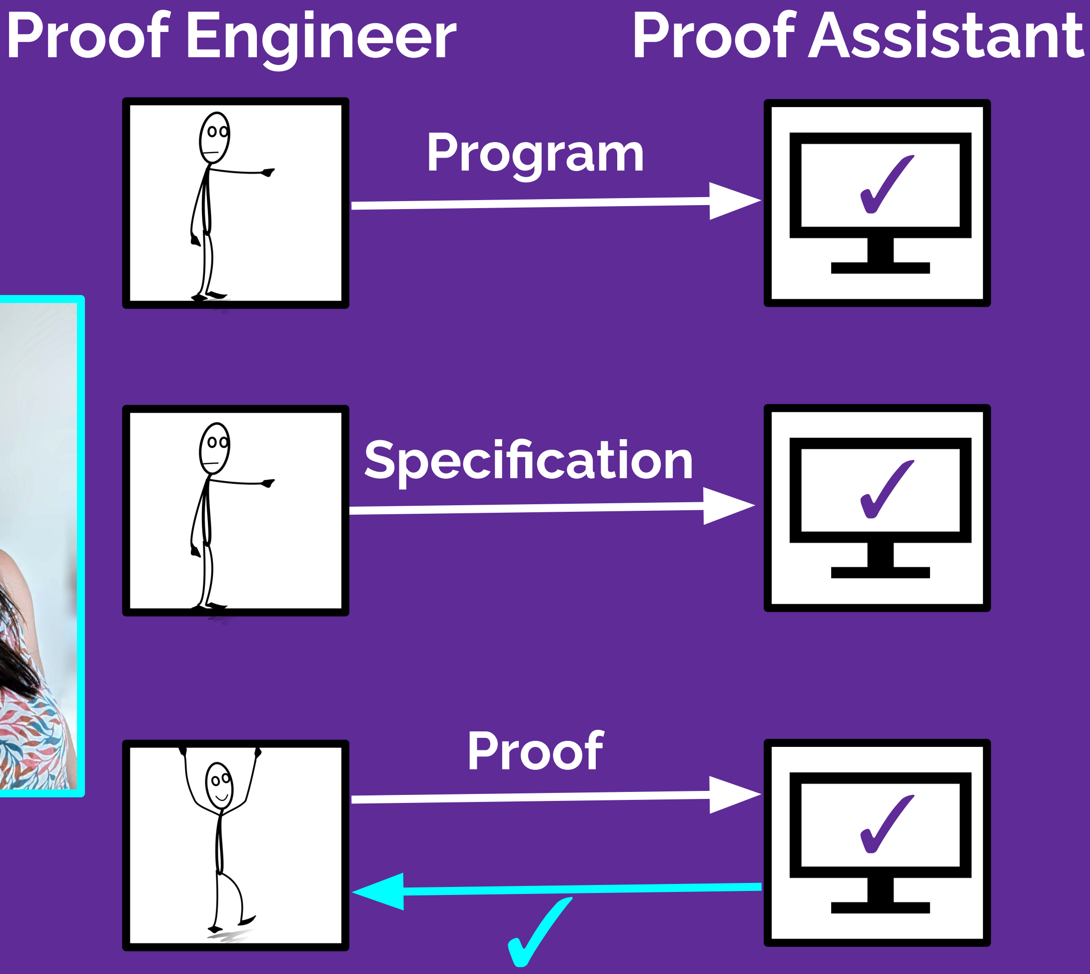
Neurosymbolic proof repair: good for large, repetitive, mostly syntactic changes at the tactic level, like from updating Coq versions? And also, good for well-scoped semantic changes at the term level, like those described by equivalences? Better than the sum of its parts?

Building Bridges (Part 4 of 5)

1. Proof Assistants
2. Symbolic Automation
3. Neural Automation
4. Building Bridges
5. **Opportunities**

So far I've assumed the specification already exists.

Tree Proofs for Free?



Proof Engineer

Proof Assistant

Program

Specification

Proof

✓

Opportunities (Part 5 of 5)

Tree Proofs for Free?

```
Inductive isLeft {A} : @tree A -> @tree A -> Prop :=
| LeafLeaf : ∀ x, isLeft (Leaf x) (Leaf x)
| NodeLeft : ∀ x l r, isLeft (Leaf x) l -> isLeft (Leaf x) (Node l r)
| NodeRight : ∀ x l r,
    r <> Leaf x -> isLeft (Leaf x) r -> isLeft (Leaf x) (Node l r).

Definition forall_Left {A} (P : @tree A -> Prop) (t : @tree A) :=
  ∀ l, isLeft l t -> P l.

Definition lift_to_tree_prop {A} (P : A -> bool) : @tree A -> Prop :=
  fun l => exists x, l = Leaf x /\ P x = true.

Theorem forall_left_leaves_correct {A} : ∀ pred (t : @tree A),
   (forall_Left (lift_to_tree_prop pred) t) <->
   (@forall_left_leaves A pred t = true).
```

Opportunities (Part 5 of 5)

What can we do to help people *specify* software, or *conjecture* in mathematics?

This is risky, but promising.

Opportunities (Part 5 of 5)

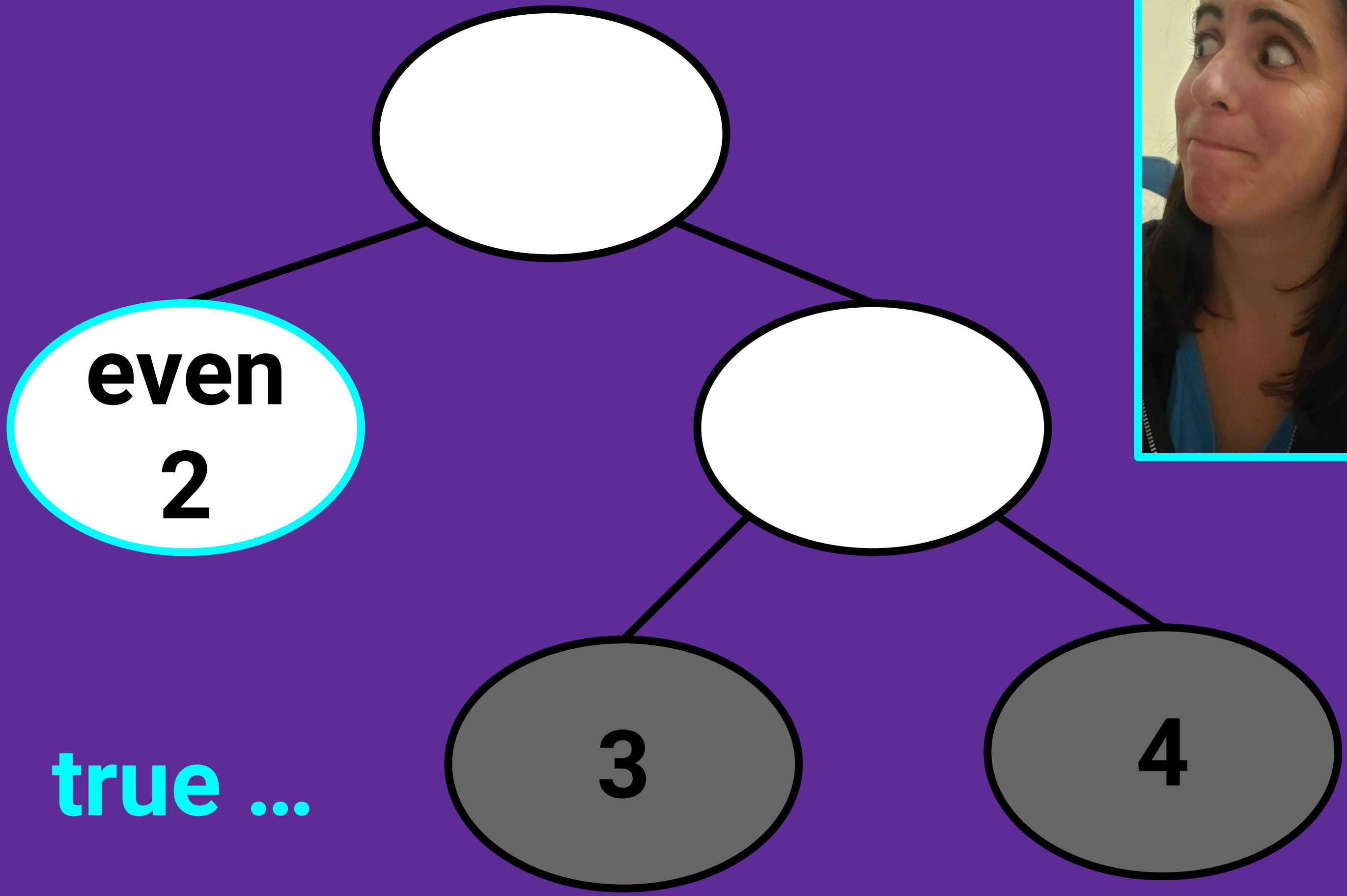
What can we do to help people *specify* software, or *conjecture* in mathematics? This is risky, but promising.

Opportunities (Part 5 of 5)

Key Challenge: There is no oracle for a specification!

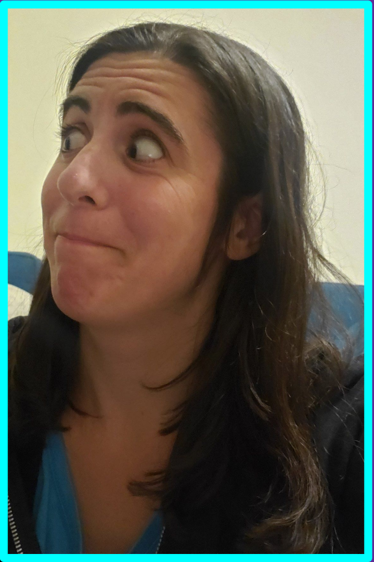
Opportunities (Part 5 of 5)

No Oracle for Specification



Opportunities (Part 5 of 5)

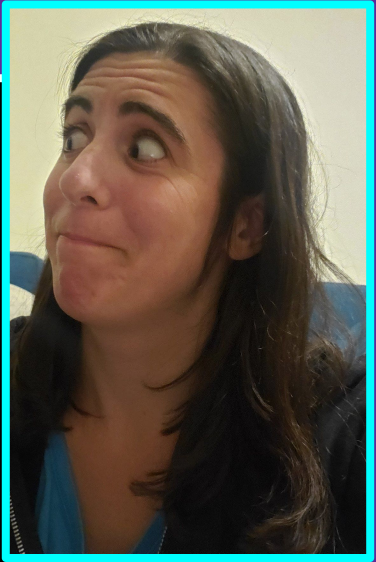
No Oracle for Specification



```
Inductive isLeft {A} : @tree A -> @tree A -> Prop :=
| LeafLeaf : ∀ x, isLeft (Leaf x) (Leaf x)
| NodeLeft : ∀ x l r, isLeft (Leaf x) l -> isLeft (Leaf x) (Node l r)
| NodeRight : ∀ x l r,
    r <> Leaf x -> isLeft (Leaf x) r -> isLeft (Leaf x) (Node l r).

Definition forall_Left {A} (P : @tree A -> Prop) (t : @tree A) :=
   ∀ l, isLeft l t -> P l.

Definition lift_to_tree_prop {A} (P : A -> bool) : @tree A -> Prop :=
  fun l => exists x, l = Leaf x /\ P x = true.

Theorem forall_left_leaves_correct {A} : ∀ pred (t : @tree A),
    (forall_Left (lift_to_tree_prop pred) t) <->
    (@forall_left_leaves A pred t = true).
```

Opportunities (Part 5 of 5)

No Oracle for Specification



```
Inductive isLeft {A} : @tree A -> @tree A -> Prop :=
| LeafLeaf : ∀ x, isLeft (Leaf x) (Leaf x)
| NodeLeft : ∀ x l r, isLeft (Leaf x) l -> isLeft (Leaf x) (Node l r).



Definition forall_Left {A} (P : @tree A -> Prop) (t : @tree A) :=
   ∀ l, isLeft l t -> P l.

Definition lift_to_tree_prop {A} (P : A -> bool) : @tree A -> Prop :=
  fun l => exists x, l = Leaf x /\ P x = true.

Theorem forall_left_leaves_correct {A} : ∀ pred (t : @tree A),
    (forall_Left (lift_to_tree_prop pred) t) <->
    (@forall_left_leaves A pred t = true).
```

Opportunities (Part 5 of 5)

No Oracle for Specification

Here are some example input/output pairs that satisfy the specification I have generated for you:

```
let tree = Node (Leaf 2, Node (Leaf 4, Leaf 3))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree) (* true *)

let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree) (* true *)

let tree = Node (Leaf 3, Node (Leaf 2, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree) (* false *)
```

Opportunities (Part 5 of 5)

No Oracle for Specification

One of these examples looks wrong. The following code should print false since 3 is not even:

```
let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree) (* false *)
```

Opportunities (Part 5 of 5)

No Oracle for Specification

I see, so you want to consider nodes that occur in right branches too. How is this?

```
Inductive isLeft {A} : @tree A -> @tree A -> Prop :=
| LeafLeaf : ∀ x, isLeft (Leaf x) (Leaf x)
| NodeLeft : ∀ x l r, isLeft (Leaf x) l -> isLeft (Leaf x) (Node l r)
| NodeRight : ∀ x l r, isLeft (Leaf x) r -> isLeft (Leaf x) (Node l r).
```

Opportunities (Part 5 of 5)

No Oracle for Specification

Close! But I don’t want that NodeRight case to hold when r is a leaf, otherwise it will just include all leaves, not just the left ones. For example, this test case you generated should still print true, but now it is false:

```
let tree = Node (Leaf 2, Node (Leaf 4, Leaf 3))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree) (* false *)
```

Opportunities (Part 5 of 5)

No Oracle for Specification

Understood. How is this?

```
Inductive isLeft {A} : @tree A -> @tree A -> Prop :=
| LeafLeaf : ∀ x, isLeft (Leaf x) (Leaf x)
| NodeLeft : ∀ x l r, isLeft (Leaf x) l -> isLeft (Leaf x) (Node l r)
| NodeRight : ∀ x l r,
    r <> Leaf x -> isLeft (Leaf x) r -> isLeft (Leaf x) (Node l r).
```

Opportunities (Part 5 of 5)

No Oracle for Specification

LGTM! Passes all tests.

```
Inductive isLeft {A} : @tree A -> @tree A -> Prop :=
| LeafLeaf : ∀ x, isLeft (Leaf x) (Leaf x)
| NodeLeft : ∀ x l r, isLeft (Leaf x) l -> isLeft (Leaf x) (Node l r)
| NodeRight : ∀ x l r,
   r <> Leaf x -> isLeft (Leaf x) r -> isLeft (Leaf x) (Node l r).
```

Opportunities (Part 5 of 5)

No Oracle for Specification

OK, then the generated code is wrong. Here is updated code that satisfies the specification:

```
let rec check_predicate_left_leaves predicate tree =
  let rec aux is_left = function
  | Leaf x -> if is_left then predicate x else true
  | Node (left, right) -> (aux true left) && (aux false right)
  in aux true tree

let tree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
let () = Printf.printf "%b\n"
  (check_predicate_left_leaves is_even tree)
```

Opportunities (Part 5 of 5)

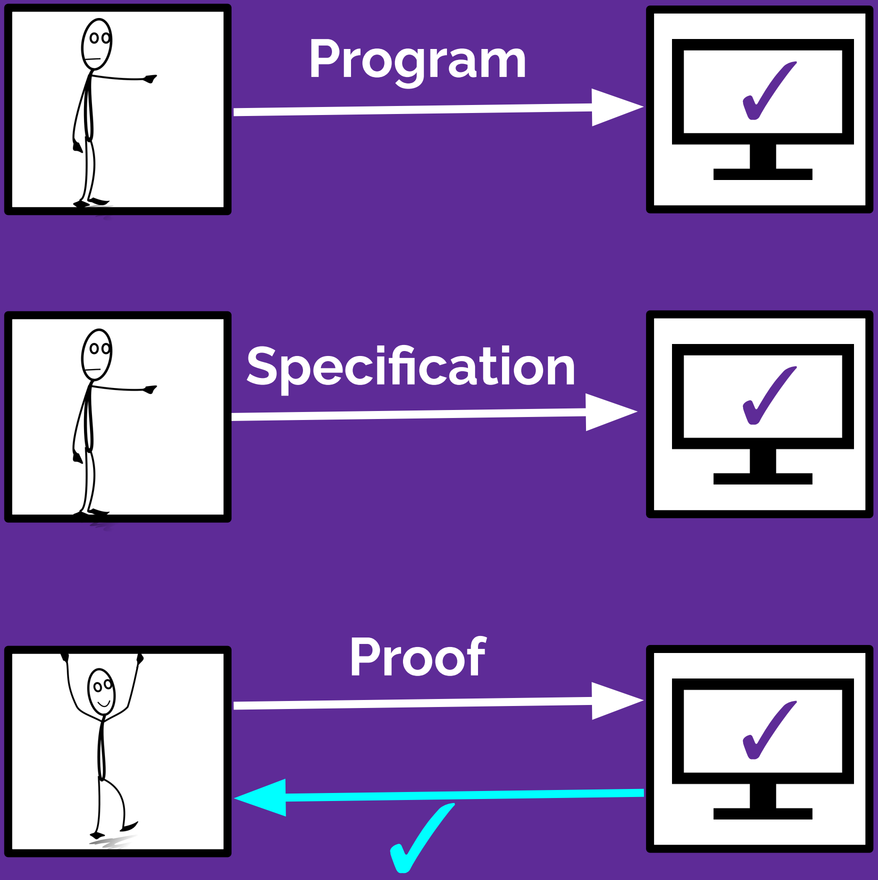
No Oracle for Specification

I’ve proven it satisfies the specification in Coq for you.

```
Theorem forall_left_leaves_correct {A} : ∀ pred (t : @tree A),
    (forall_Left (lift_to_tree_prop pred) t) <->
    (@forall_left_leaves A pred t = true).
Proof.
  unfold forall_left_leaves, forall_Left, lift_to_tree_prop. split.
  - intros. apply aux_correct. auto.
  - induction t; intros.
    + edestruct H. inversion H0. subst. exists a. split; auto.
    + simpl in H. inversion H0; subst.
      * apply IHt1; auto. destruct (aux pred true t1) eqn:b1; auto.
      * apply IHt2; auto. destruct (aux pred false t2) eqn:b2.
        -- destruct t2; auto. inversion H5. subst. destruct H4. auto.
        -- destruct (aux pred true t1) eqn:b1; discriminate.
Qed.
```

Opportunities (Part 5 of 5)

Tree Proofs for Free!



Proof Engineer

Proof Assistant

Program

Specification

Proof

✓

Opportunities (Part 5 of 5)

Open Question:

What tools can best help users make sense of generated theorems? What information presented in what ways best helps users ensure that they match their intentions?

Opportunities (Part 5 of 5)