Matching μ-Logic: Foundation of A Unifying Programming Language Framework

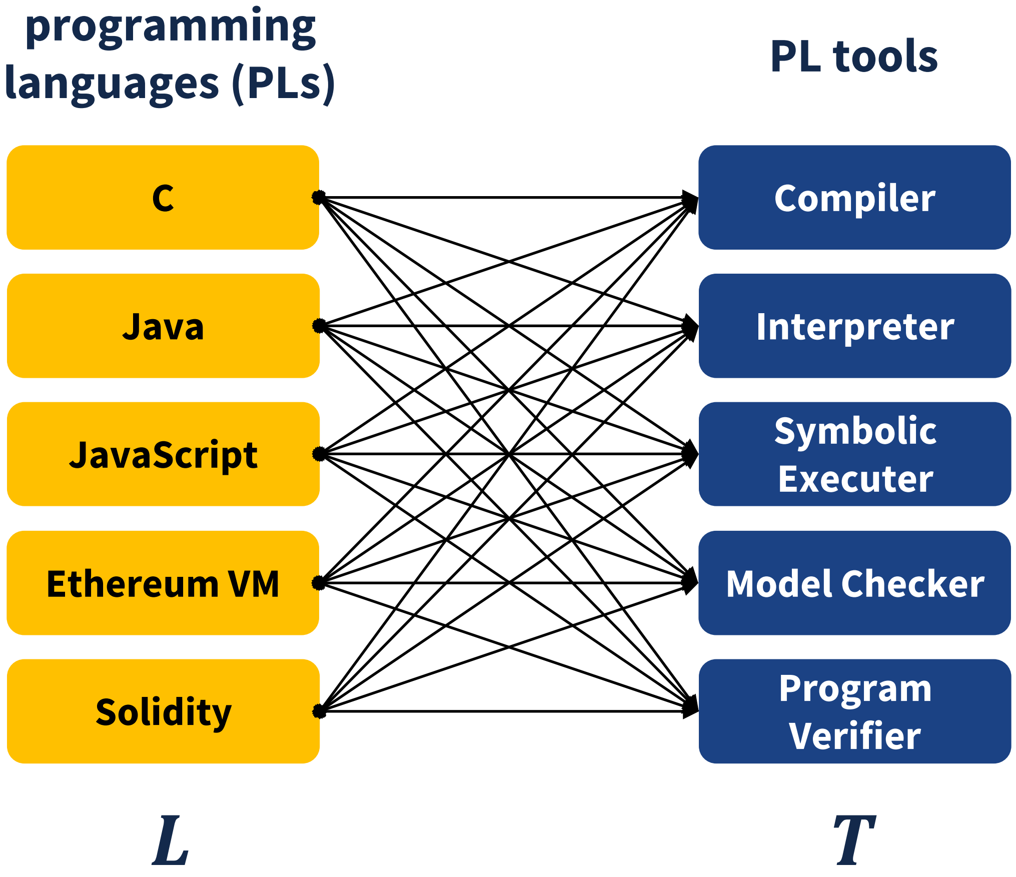
Xiaohong Chen
PhD Final Exam
University of Illinois Urbana-Champaign
Department of Computer Science

May 3, 2023

Overview

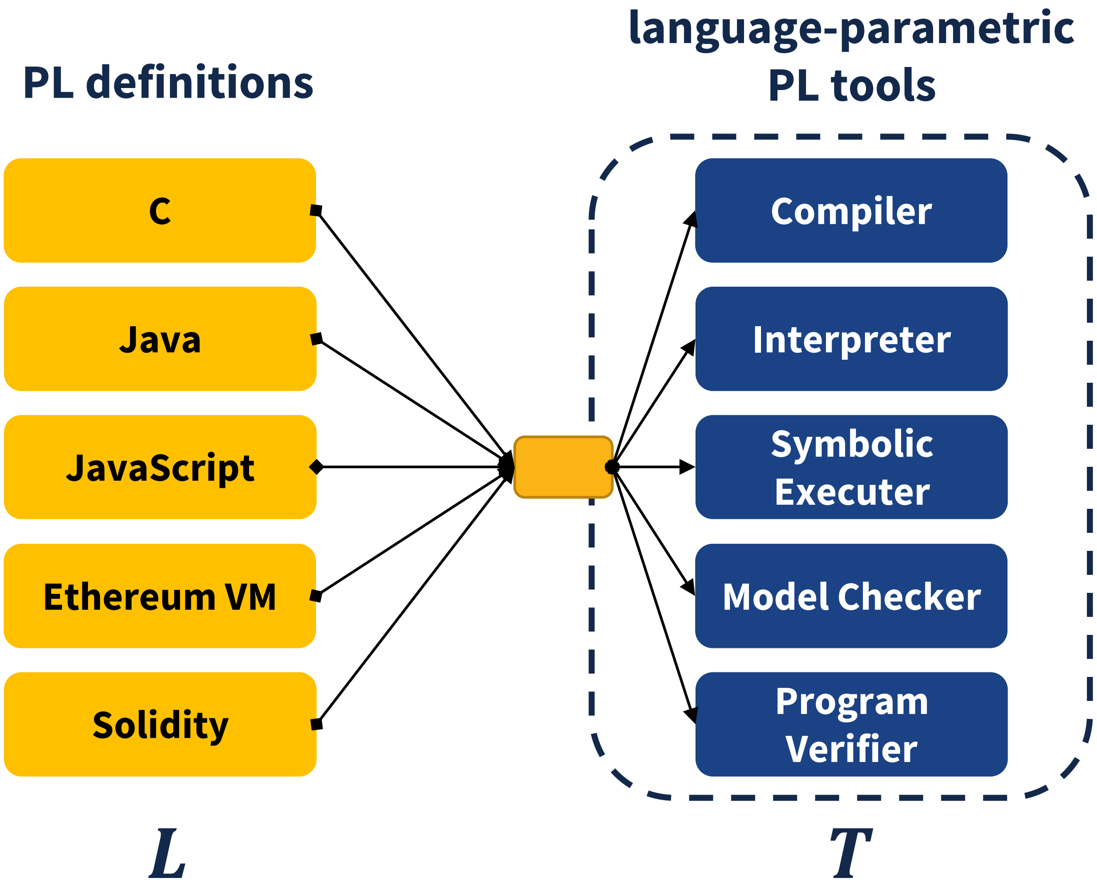
- **Introduction to a Unifying Programming Language Framework**
  - Motivating Example: The K Semantic Framework
  - Research Challenge: Proving the Correctness of K
- **Main Contribution: Matching μ-Logic**
  - Basic Definitions
  - Expressive Power
  - Proof System and Proof Checker
  - Automatic Theorem Prover
- **Using Matching μ-Logic to Prove the Correctness of K**
- **Concluding Remarks**

Programming Language Design & Implementation: State-of-the-Art



$L \times T$ systems
to develop and maintain

A Unifying Programming Language Framework



$L + T$ systems

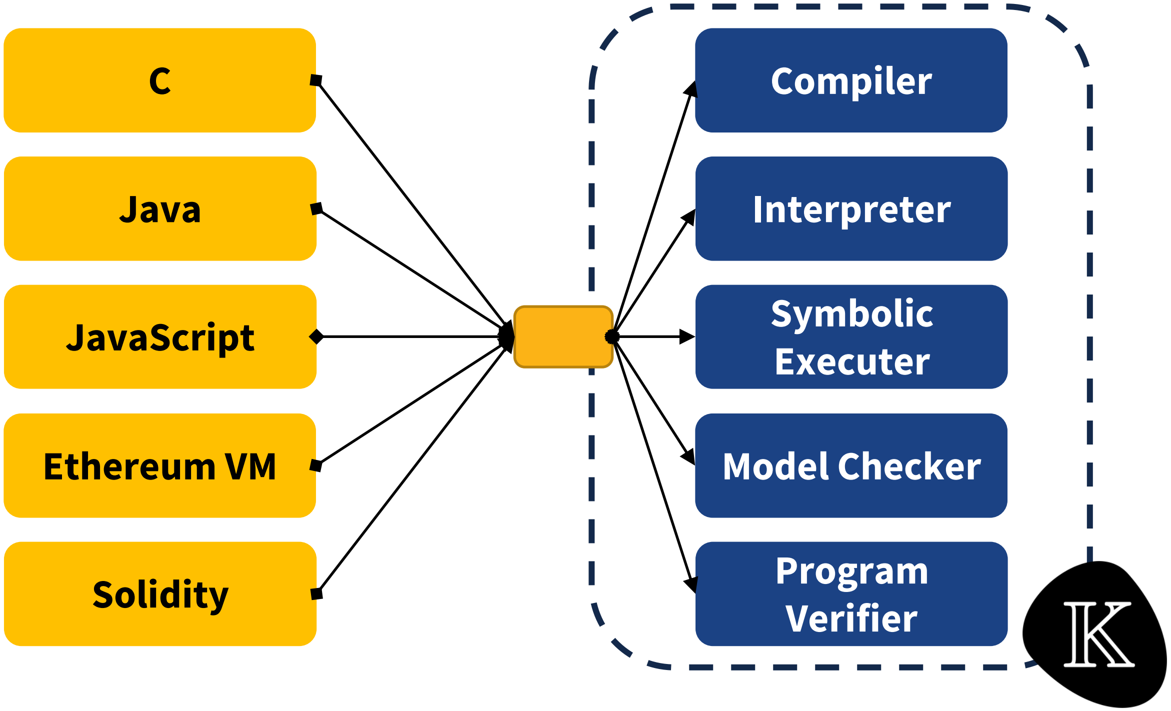
to develop and maintain

K Semantic Framework https://kframework.org/

formal systems laboratory

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

runtime verification



K has wide applications

RV-Match

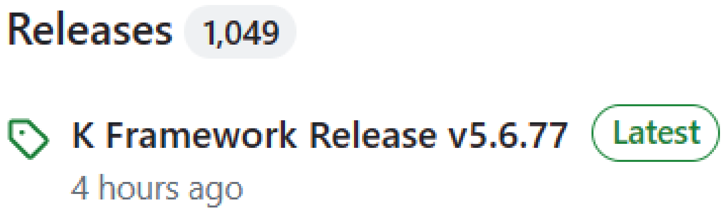
NASA

BOEING

ethereum

Research Challenge: Proving the Correctness of K

- **K has a large code base**
  - >500k LOC in 4 programming languages
  - complex data structures, algorithms, and optimizations
- **K is constantly evolving**



Releases 1,049

K Framework Release v5.6.77 Latest

4 hours ago

- **It's not practical to thoroughly verify the entire K.**
- **Main Idea: Translation Validation**

Main Idea: Translation Validation

K	**Matching μ-Logic: Foundation of K**
A PL definition **Ethereum VM**	A *logical theory* Γ^{EVM}
Any PL task • program execution **Interpreter** • formal verification **Program Verifier**	A *logical theorem* proved by a *proof system* • $\Gamma^{\mathrm{EVM}} \vdash t_{\mathrm{init}} \Rightarrow_{\mathrm{exec}} t_{\mathrm{final}}$ • $\Gamma^{\mathrm{EVM}} \vdash \varphi_{\mathrm{pre}} \rightsquigarrow \varphi_{\mathrm{post}}$
Correctness of the task	Generating the proof and checking it using a *200-LOC proof checker*

correctness of any task done by any tool of any PL in K ⟹ **correctness of 1 task (proof checking) done by 1 program (proof checker)**

Why Matching μ-Logic?

- **We tried many logics/calculi/foundations**

  First-order logic; Second/higher-order logic; Least fixpoint logic; Modal logics; Temporal logics (LTL, CTL, CTL*, …), λ-calculus; Type systems (parametric, dependent, inductive, …); μ-calculus; Hoare logics; Separation logics; Dynamic logics; Rewriting logic; Reachability logic; Equational logic; Small-/big-step SOS; Evaluation contexts; Abstract machines (CC, CK, CEK, SECD, …); Chemical abstract machine; Axiomatic; Continuations; Denotational; Initial Algebras; …

- **… but each of the above had limitations**
  - Some only handle certain aspects of K (e.g., operational semantics)
  - Some are "design patterns" (e.g., Hoare logics)
  - Some are domain-specific (e.g., separation logic)
  - Some require complex encodings/translations

- **Matching μ-logic: Expressive and Small**
  - PLs defined as theories; PL tools specified by theorems
  - Logics defined as theories; logical proof rules proved as theorems
  - A 14-rule proof system and a 200-LOC proof checker: small trust base

Overview

- **Introduction to a Unifying Programming Language Framework**
    - Motivating Example: The K Semantic Framework
    - Research Challenge: Proving the Correctness of K
- **Main Contribution: Matching μ-Logic**
    - Basic Definitions
    - Expressive Power
    - Proof System and Proof Checker
    - Automatic Theorem Prover
- **Using Matching μ-Logic to Prove the Correctness of K**
- **Concluding Remarks**

Matching μ-Logic Syntax

Matching μ-logic formulas, called *patterns*:

$$\varphi ::= \underbrace{\boxed{x \mid \sigma(\varphi_1, \dots, \varphi_n)}}_{\textbf{structures}} \mid \underbrace{\boxed{\varphi_1 \wedge \varphi_2 \ \mid \neg\varphi}}_{\textbf{logical constraints}} \mid \underbrace{\boxed{\exists x.\, \varphi}}_{\substack{\textbf{first-order} \\ \textbf{quantification}}} \mid \underbrace{\boxed{X \mid \mu X.\, \varphi}}_{\substack{\textbf{fixpoints} \\ \textbf{(in this talk)}}}$$

- X a *set variable*, ranging over sets
- $\mu X.\, \varphi$ the *least fixpoint* of φ, where X occurs positively in φ
- $\nu X.\, \varphi \equiv \neg\mu X.\, \neg\varphi[\neg X / X]$ the *greatest fixpoint* of φ

Matching μ-Logic Semantics

A matching μ-logic *model* has:

- a carrier set M
- a function $\sigma_M : M \times \cdots \times M \to \mathcal{P}(M)$ for each symbol σ

Given a model M and a variable valuation ρ:

$$\boldsymbol{\varphi} \xrightarrow{\textbf{pattern matching}} |\boldsymbol{\varphi}|_{\boldsymbol{M,\rho}} \subseteq \boldsymbol{M}$$

- $|x|_{M,\rho} = \{\rho(x)\}$
- $|\sigma(\varphi_1, \dots, \varphi_n)|_{M,\rho} = \bigcup\{\sigma_M(a_1, \dots, a_n) \mid a_i \in |\varphi_i|_{M,\rho}\}$
- $|\varphi_1 \wedge \varphi_2|_{M,\rho} = |\varphi_1|_{M,\rho} \cap |\varphi_2|_{M,\rho}$
- $|\neg\varphi|_{M,\rho} = M \setminus |\varphi|_{M,\rho}$
- $|\exists x.\, \varphi|_{M,\rho} = \bigcup\{|\varphi|_{M,\rho[a/x]} \mid a \in M\}$
- $|X|_{M,\rho} = \rho(X)$
- $|\mu X.\, \varphi|_{M,\rho} = \mathbf{lfp}\left(A \mapsto |\varphi|_{M,\rho[A/X]}\right)$

Examples of Fixpoint Patterns

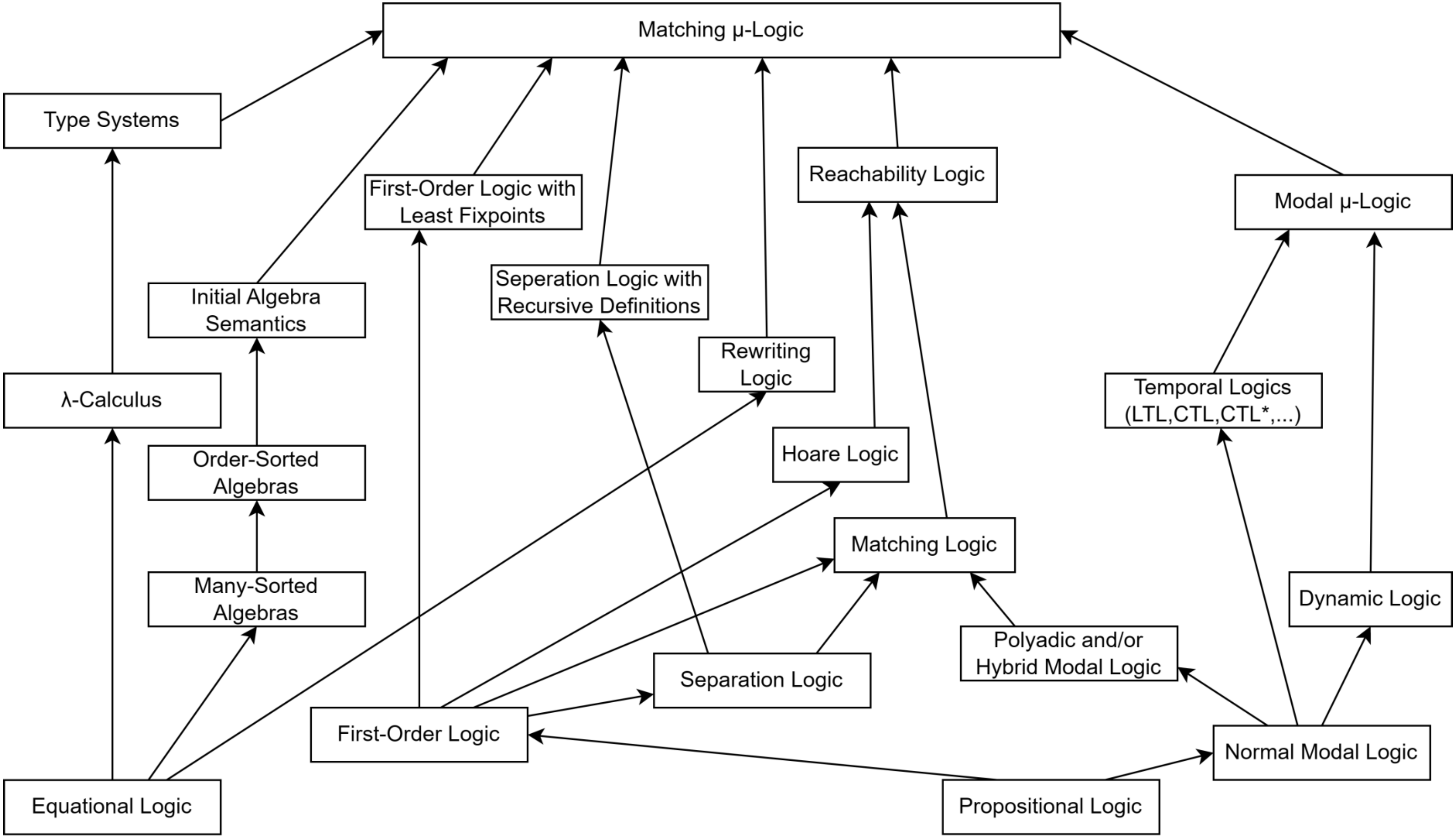
- **inductive datatypes** [JLAMP'21]
  - `type nat = Zero | Succ of nat`
  - $\top_{\mathbf{nat}} = \mu N.\, \mathbf{0} \vee \mathbf{Succ}(N)$
  - `type list = Nil | Cons of nat * list`
  - $\top_{\mathbf{list}} = \mu L.\, \mathbf{Nil} \vee \mathbf{Cons}(\top_{\mathbf{nat}}, L)$
- **program execution** [LICS'19, CAV'21]
  - $t_1 \Rightarrow_{\mathrm{exec}} t_2 \quad \equiv \quad t_1 \to \underbrace{\mathbf{eventually}\ t_2}_{\mu S.\, t_2 \vee (\mathbf{next}\ S)}$
- **formal verification** [LICS'19, OOPSLA'23]
  - $\varphi_{\mathrm{pre}} \rightsquigarrow \varphi_{\mathrm{post}} \quad \equiv \quad \varphi_{\mathrm{pre}} \to \underbrace{\mathbf{weak\text{-}eventually}\ \varphi_{\mathrm{post}}}_{\nu S.\, \varphi_{\mathrm{post}} \vee (\mathbf{next}\ S)}$

    (if φ_{pre} holds when P starts, then φ_{post} holds when P terminates)

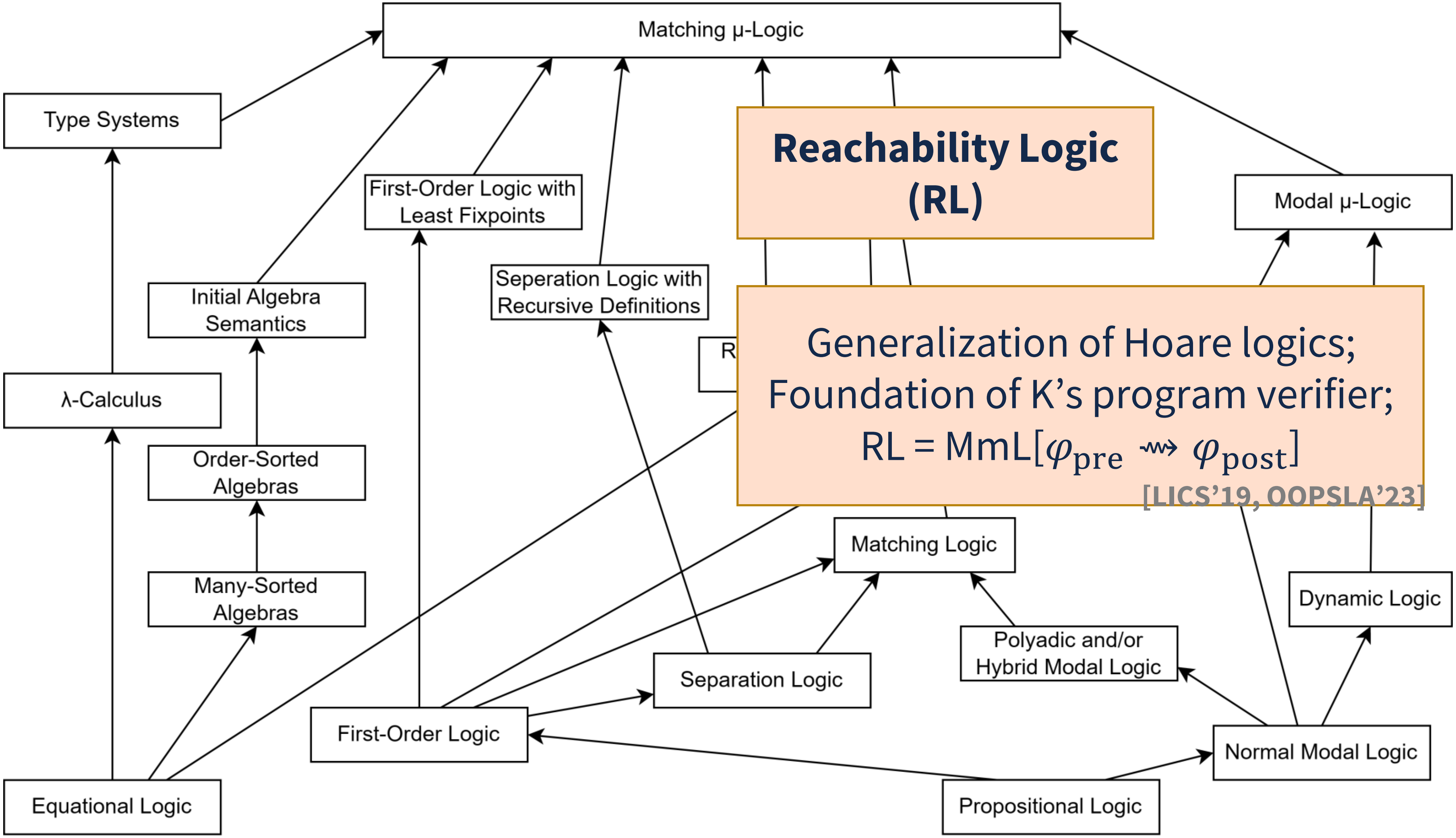
Various forms/instances of fixpoints are definable by patterns.

Matching μ-Logic (MmL) Expressive Power

[Chap 5 of Thesis, also in LICS’19, OOPSLA’20, ICFP’20, CAV’21, JLAMP’21, JLAMP’22, OOPSLA’23]



Matching μ-Logic (MmL) Expressive Power

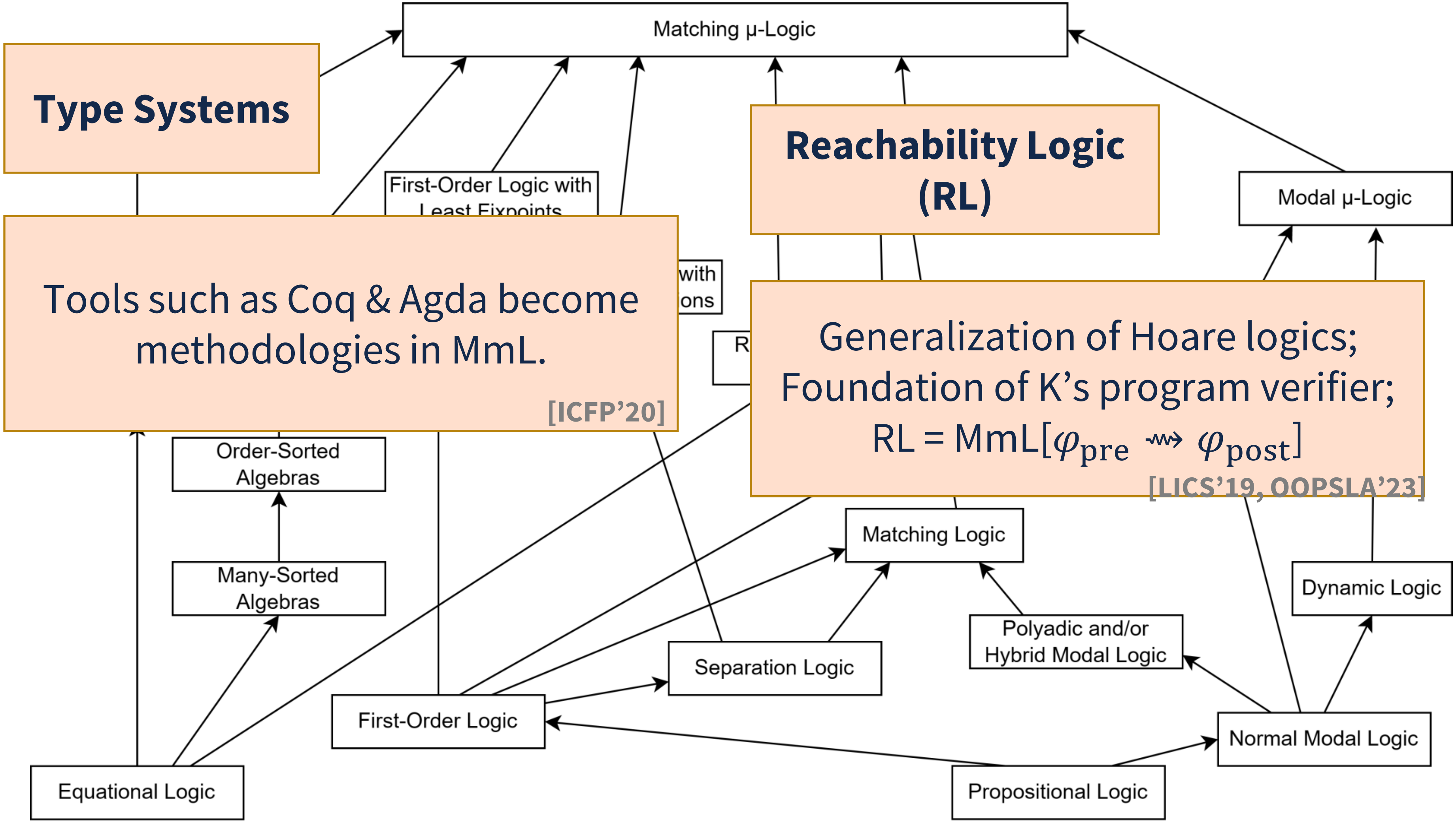


Reachability Logic (RL)

Generalization of Hoare logics;
Foundation of K's program verifier;
RL = MmL[$\varphi_{\mathrm{pre}} \rightsquigarrow \varphi_{\mathrm{post}}$]

[LICS'19, OOPSLA'23]

Matching μ-Logic (MmL) Expressive Power



Type Systems

Tools such as Coq & Agda become methodologies in MmL.

[ICFP’20]

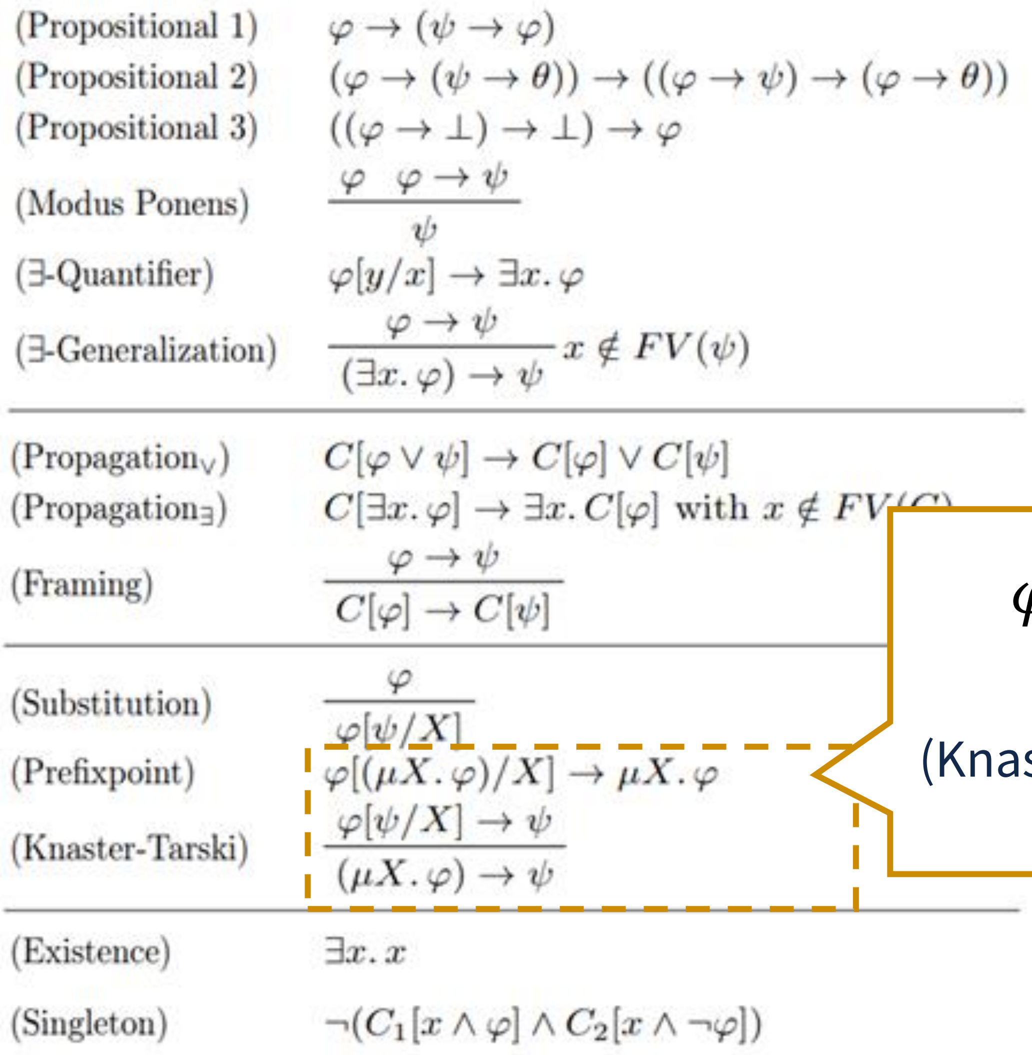
Reachability Logic (RL)

Generalization of Hoare logics;
Foundation of K’s program verifier;
RL = MmL[$\varphi_{\text{pre}} \rightsquigarrow \varphi_{\text{post}}$]

[LICS’19, OOPSLA’23]

Matching μ-Logic Proof System

(only 14 proof rules)



(Propositional 1)	$\varphi \to (\psi \to \varphi)$
(Propositional 2)	$(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))$
(Propositional 3)	$((\varphi \to \bot) \to \bot) \to \varphi$
(Modus Ponens)	$\dfrac{\varphi \quad \varphi \to \psi}{\psi}$
($\exists$-Quantifier)	$\varphi[y/x] \to \exists x.\, \varphi$
($\exists$-Generalization)	$\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}\ x \notin FV(\psi)$
(Propagation$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
(Propagation$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ with $x \notin FV(C)$
(Framing)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
(Prefixpoint)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$
(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
(Existence)	$\exists x.\, x$
(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

Defines provability relation

$$\Gamma \vdash \varphi$$

theory theorem

$$\varphi[(\mu X.\, \varphi)/X] \leftrightarrow \mu X.\, \varphi$$

(Knaster-Tarski) $$\frac{\varphi[\psi/X] \leftrightarrow \psi}{(\mu X.\, \varphi) \to \psi}$$

proof rules for fixpoints

Deriving Mathematical Induction in Matching μ-Logic

Mathematical Induction: To show a property P holds for all naturals, prove:
(**basis**). The number 0 satisfies P
(**step**). If n satisfies P then $n+1$ also satisfies P.

Step 1. Note that $\top_{\mathbf{nat}} = \mu N.\, 0 \vee \mathbf{succ}(N)$ captures all natural numbers.

Step 2. Set the proof goal $\vdash \big(\mu N.\, 0 \vee \mathbf{succ}(N)\big) \to \psi_P$

Step 3. Apply (**Knaster Tarski**) and get

$$\vdash \big(0 \vee \mathbf{succ}(\psi_P)\big) \to \psi_P$$

i.e.,

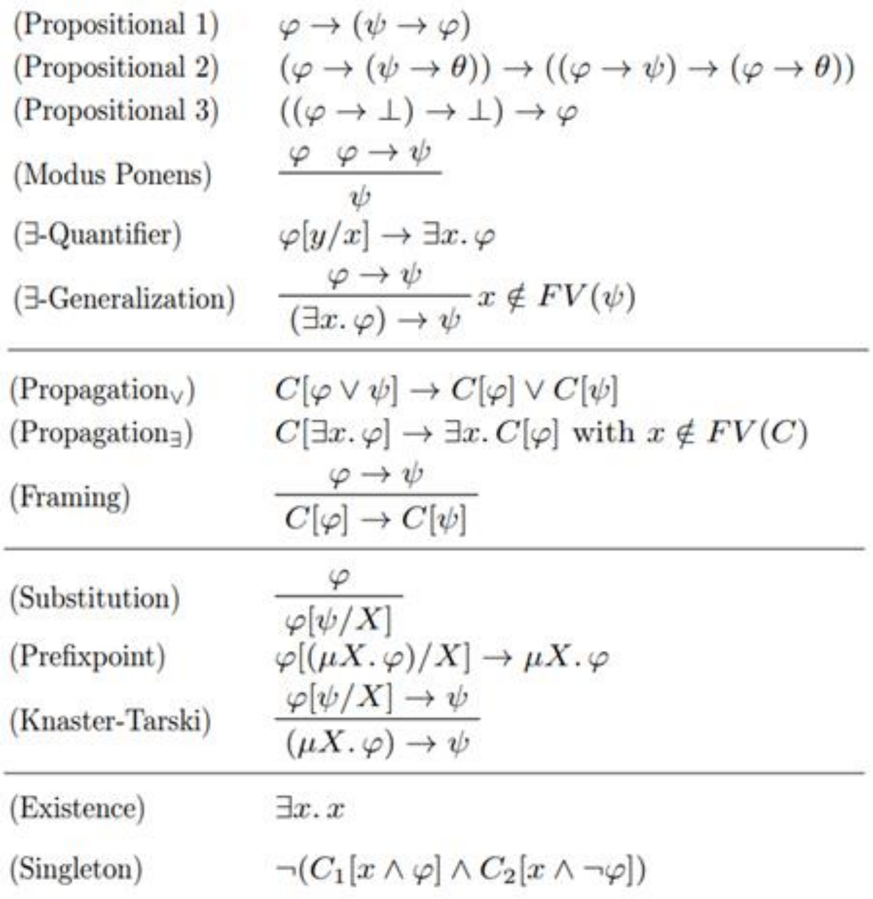
Sub-Goal-1 $0 \to \psi_P$ .... (**basis**)
Sub-Goal-2 $\mathbf{succ}(\psi_P) \to \psi_P$ .... (**step**)

(**Knaster Tarski**)

$$\frac{\varphi[\psi/X] \to \psi}{\mu X.\, \varphi \to \psi}$$

Various forms/instances of fixpoints reasoning are supported by (Knaster Tarski)

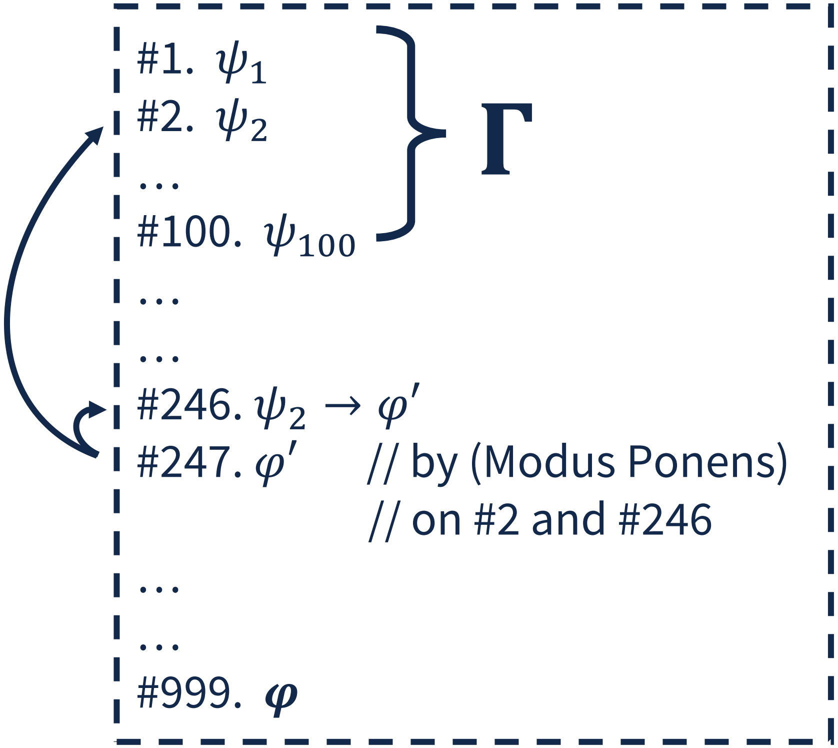
Matching μ-Logic Proof Object



(Propositional 1)	$\varphi \to (\psi \to \varphi)$
(Propositional 2)	$(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))$
(Propositional 3)	$((\varphi \to \bot) \to \bot) \to \varphi$
(Modus Ponens)	$\frac{\varphi \quad \varphi \to \psi}{\psi}$
($\exists$-Quantifier)	$\varphi[y/x] \to \exists x.\, \varphi$
($\exists$-Generalization)	$\frac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}\ x \notin FV(\psi)$
(Propagation$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
(Propagation$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ with $x \notin FV(C)$
(Framing)	$\frac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
(Substitution)	$\frac{\varphi}{\varphi[\psi/X]}$
(Prefixpoint)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$
(Knaster-Tarski)	$\frac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
(Existence)	$\exists x.\, x$
(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

$$\boldsymbol{\Gamma \vdash \varphi}$$

theory theorem



#1. ψ_1

#2. ψ_2

…

#100. ψ_{100}

Γ

…

…

#246. $\psi_2 \to \varphi'$

#247. φ' // by (Modus Ponens) // on #2 and #246

…

…

#999. $\boldsymbol{\varphi}$

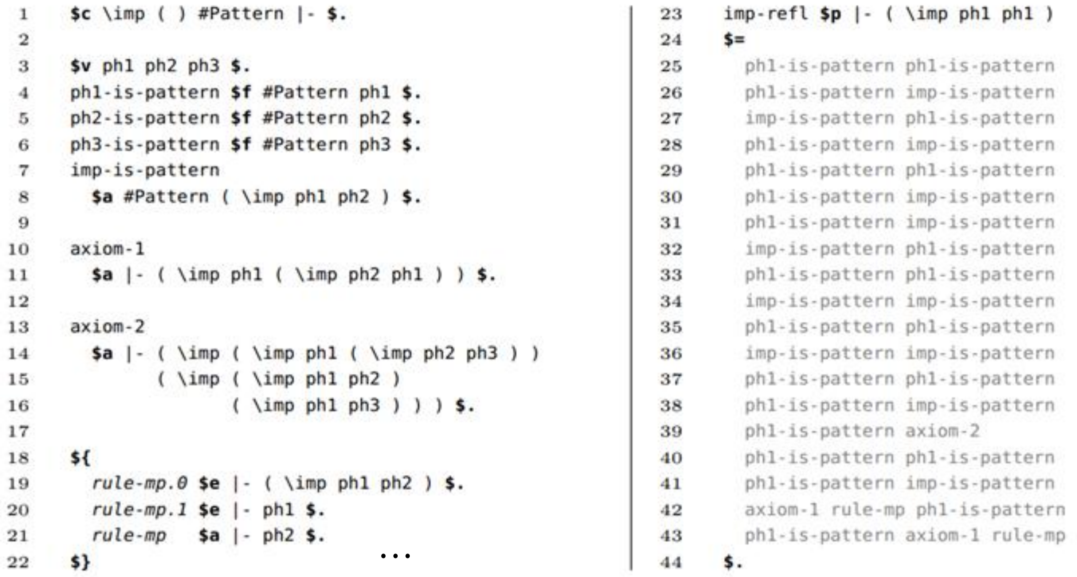
a proof object;

very easy & fast to check;

embarrassingly parallelable

Matching μ-Logic Proof Checker

- We use Metamath [Megill & Wheeler] http://metamath.org
    - to encode proof objects &
    - check them automatically
    - embarrassingly parallelable
- Very small trust base
    - Matching μ-logic: 200 LOC
    - Metamath itself:
        - 350 LOC in Python
        - 400 LOC in Haskell
        - 550 LOC in C#
        - …



```
$c \imp ( ) #Pattern |- $.

$v ph1 ph2 ph3 $.
ph1-is-pattern $f #Pattern ph1 $.
ph2-is-pattern $f #Pattern ph2 $.
ph3-is-pattern $f #Pattern ph3 $.
imp-is-pattern
  $a #Pattern ( \imp ph1 ph2 ) $.

axiom-1
  $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.

axiom-2
  $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
        ( \imp ( \imp ph1 ph2 )
               ( \imp ph1 ph3 ) ) ) $.

${
  rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
  rule-mp.1 $e |- ph1 $.
  rule-mp   $a |- ph2 $.
$}                          …
```

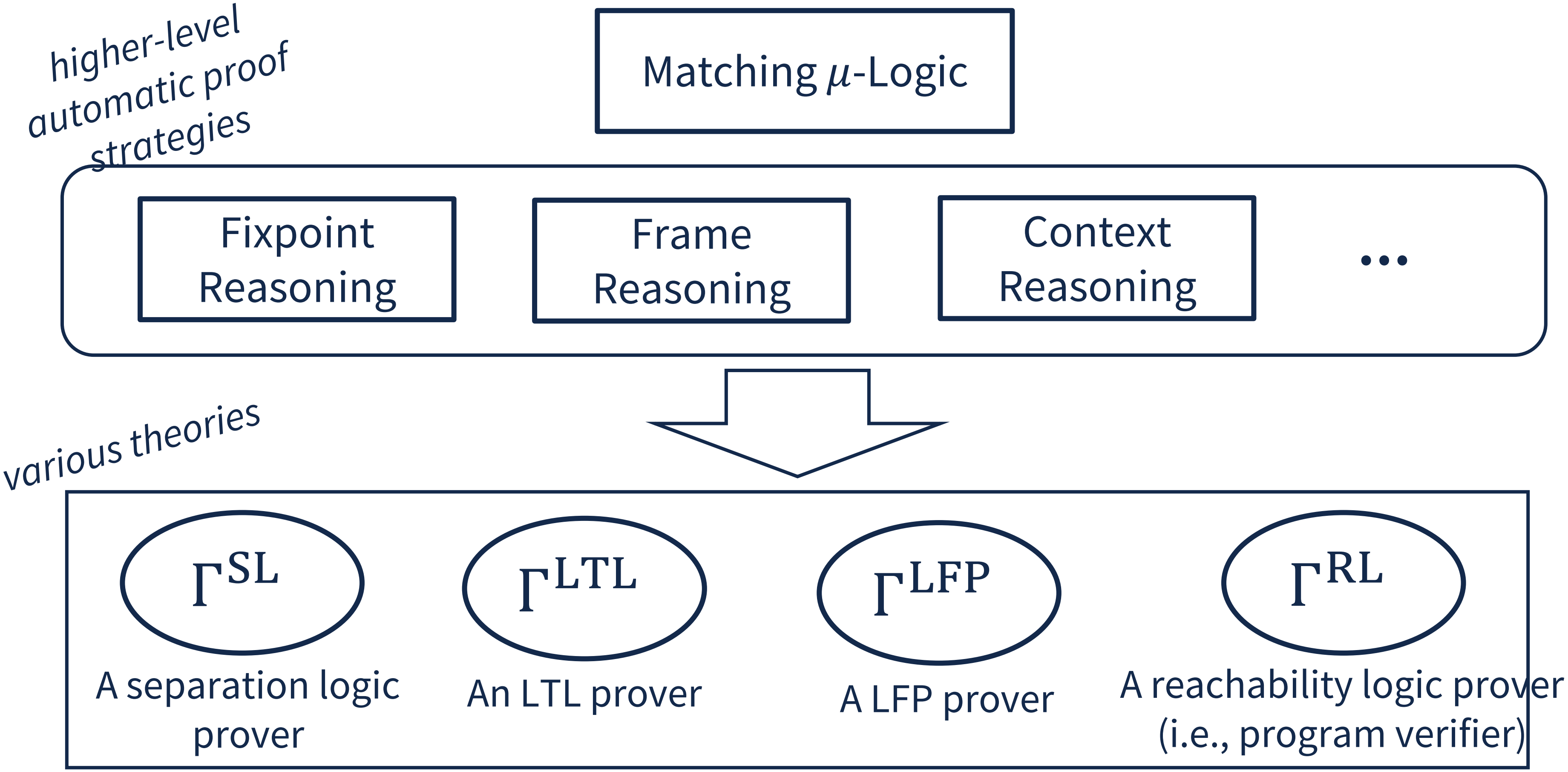
```
imp-refl $p |- ( \imp ph1 ph1 )
$=
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern axiom-2
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  axiom-1 rule-mp ph1-is-pattern
  ph1-is-pattern axiom-1 rule-mp
$.
```

Matching μ-logic syntax & proof rules; Defined in 200 LOC

Proof objects (automatically checked)

Checking proof objects is fast and trustworthy.

Automatic Theorem Prover for Matching μ-Logic



higher-level automatic proof strategies

Matching μ-Logic

Fixpoint Reasoning

Frame Reasoning

Context Reasoning

…

various theories

Γ^{SL}

A separation logic prover

Γ^{LTL}

An LTL prover

Γ^{LFP}

A LFP prover

Γ^{RL}

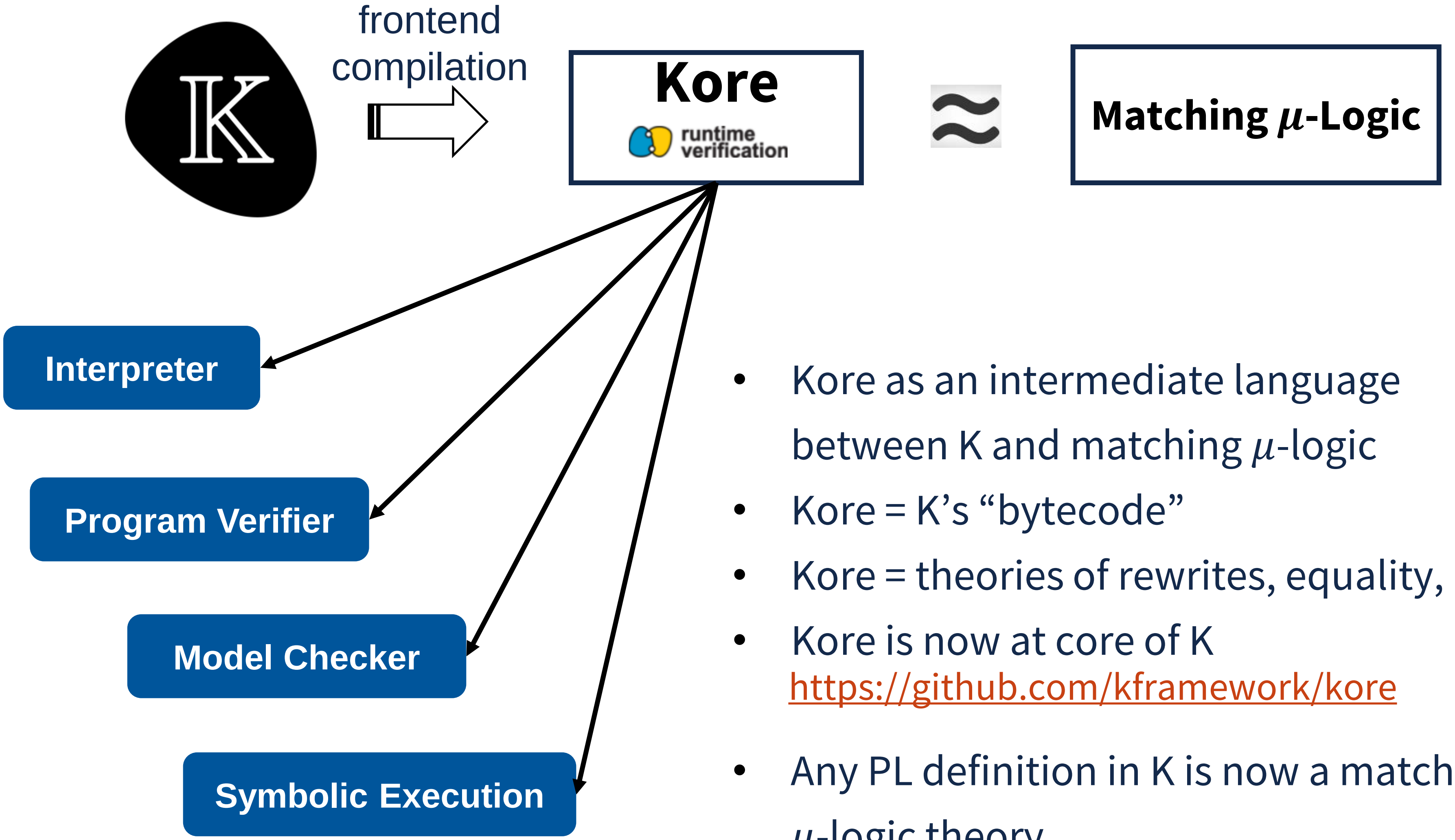
A reachability logic prover (i.e., program verifier)

- Separation logic: Proved 265/280 benchmark tests in SL-COMP’19
  - (latest WIP even reached 280/280!)

Overview

- **Introduction to a Unifying Programming Language Framework**
  - Motivating Example: The K Semantic Framework
  - Research Challenge: Proving the Correctness of K
- **Main Contribution: Matching μ-Logic**
  - Basic Definitions
  - Expressive Power
  - Proof System and Proof Checker
  - Automatic Theorem Prover
- **Using Matching μ-Logic to Prove the Correctness of K (in the translation validation style)**
  - Translating PL Definitions in K to Matching μ-Logic Theories
  - Generating Proof Objects for K’s Program Verifier
- **Concluding Remarks**

Translating K to Matching μ-Logic



frontend compilation

Kore

runtime verification

$\approx$

Matching μ-Logic

Interpreter

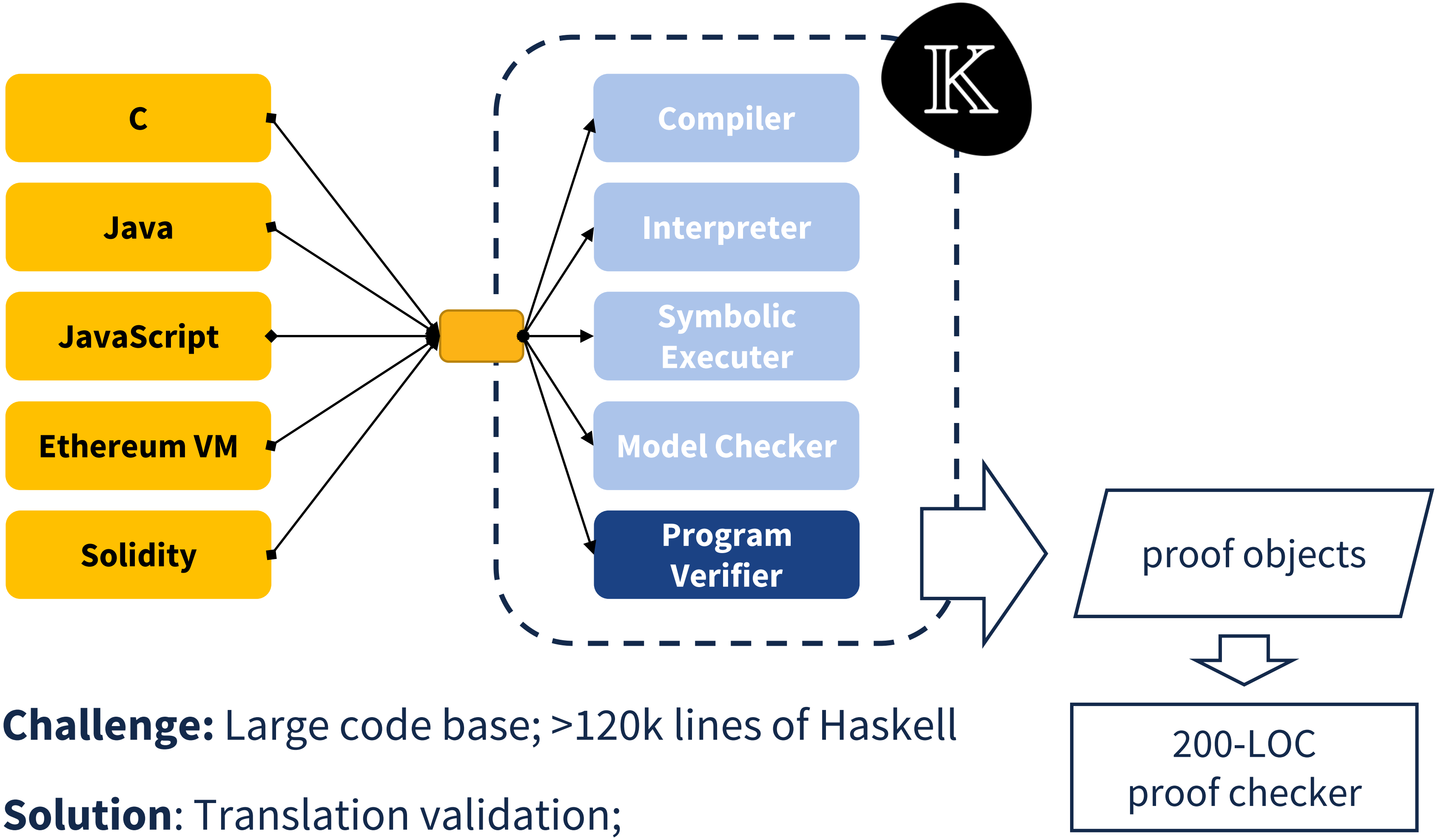
Program Verifier

Model Checker

Symbolic Execution

- Kore as an intermediate language between K and matching μ-logic
- Kore = K's "bytecode"
- Kore = theories of rewrites, equality, …
- Kore is now at core of K
  https://github.com/kframework/kore
- Any PL definition in K is now a matching μ-logic theory

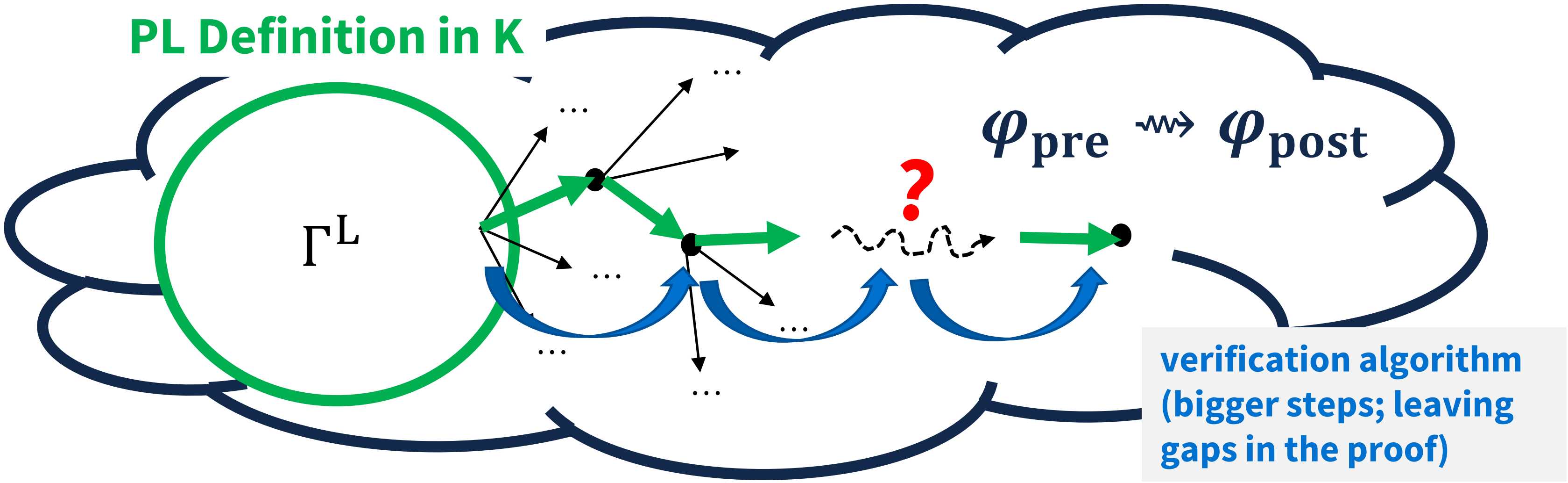
Proving the Correctness of K's Program Verifier



Challenge: Large code base; >120k lines of Haskell

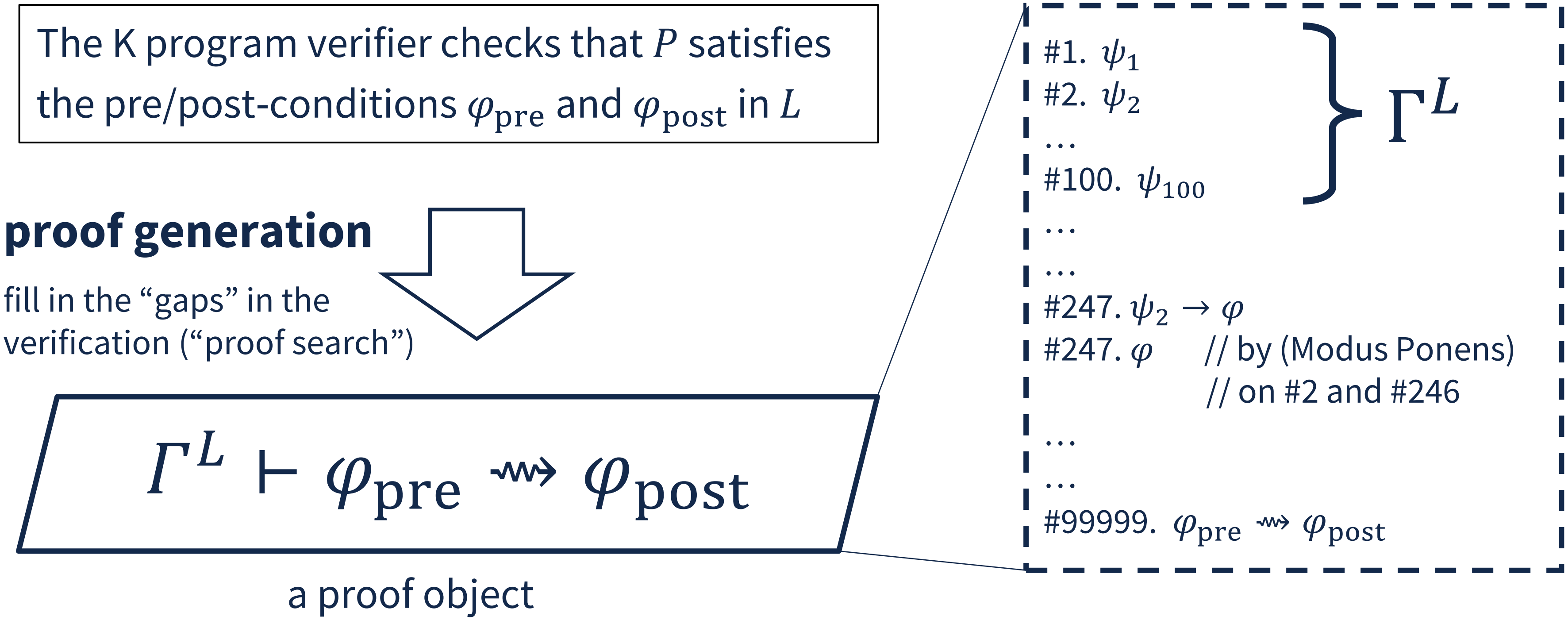
Solution: Translation validation;
Generating proof objects and
checking them automatically

Program Verification is Actually Proof Search


PL Definition in K
Γ^{L}
$\varphi_{\text{pre}} \rightsquigarrow \varphi_{\text{post}}$
?
...
verification algorithm (bigger steps; leaving gaps in the proof)

A program verifier is a specialized, optimized, proof searcher.

Proof Generation for Program Verification



The K program verifier checks that P satisfies the pre/post-conditions φ_{pre} and φ_{post} in L

proof generation

fill in the “gaps” in the verification (“proof search”)

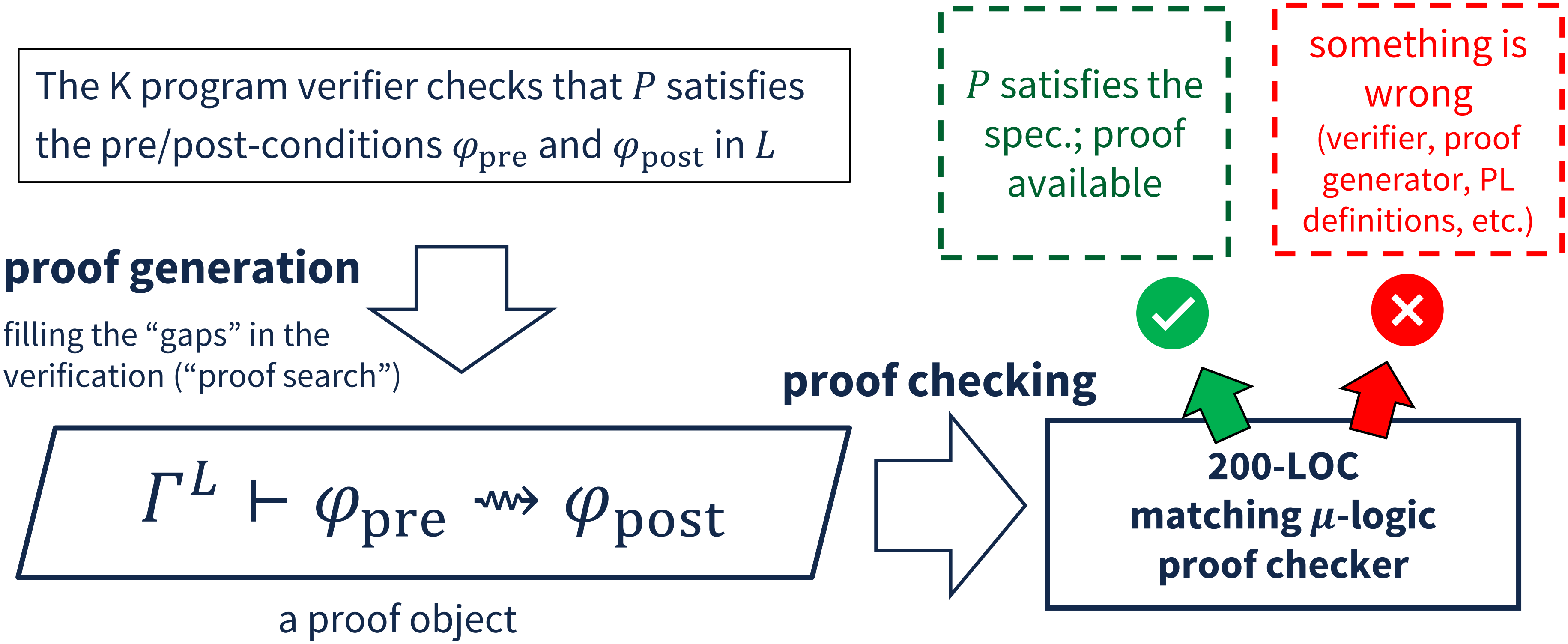
$$\Gamma^L \vdash \varphi_{\text{pre}} \rightsquigarrow \varphi_{\text{post}}$$

a proof object

#1. ψ_1
#2. ψ_2
…
#100. ψ_{100}
…
…
#247. $\psi_2 \to \varphi$
#247. φ // by (Modus Ponens)
// on #2 and #246
…
…
#99999. $\varphi_{\text{pre}} \rightsquigarrow \varphi_{\text{post}}$

Γ^L

Proof Generation for Program Verification



The K program verifier checks that P satisfies the pre/post-conditions φ_{pre} and φ_{post} in L

proof generation

filling the "gaps" in the verification ("proof search")

$$\Gamma^L \vdash \varphi_{\mathrm{pre}} \rightsquigarrow \varphi_{\mathrm{post}}$$

a proof object

proof checking

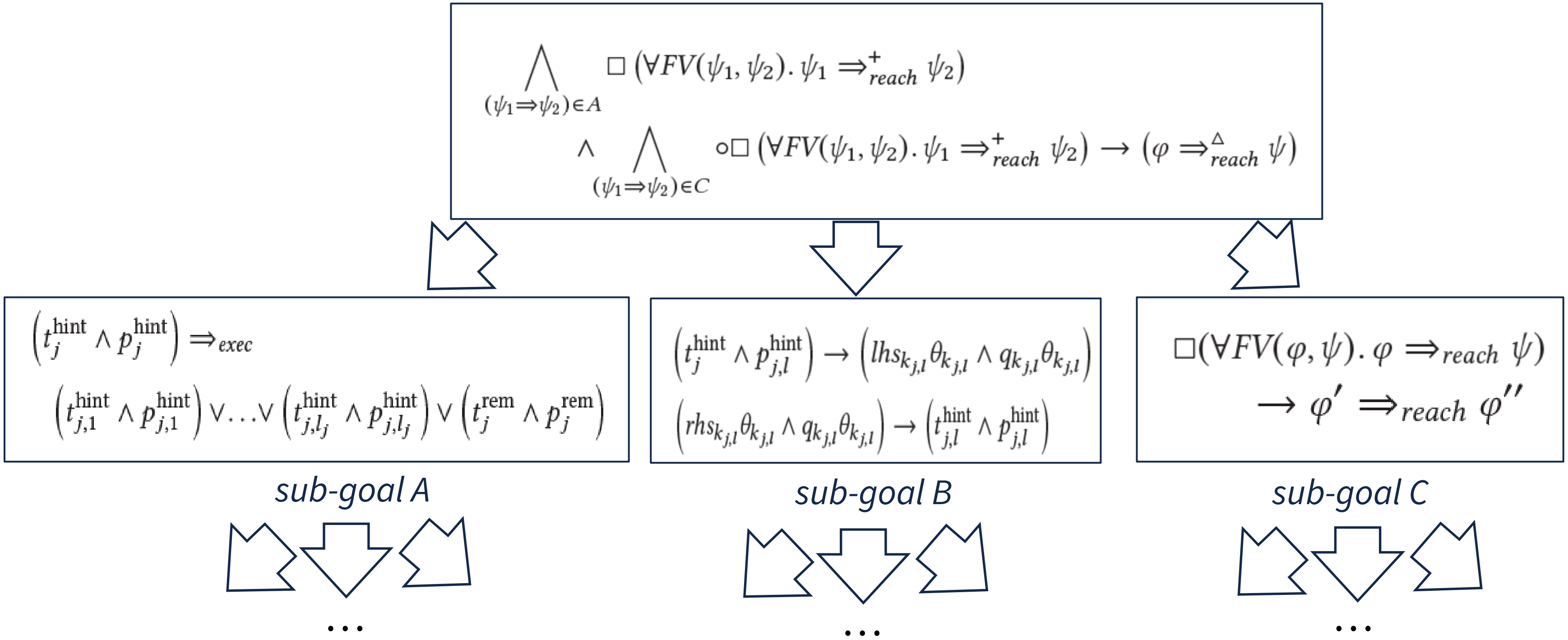
200-LOC matching μ-logic proof checker

✓ P satisfies the spec.; proof available

✗ something is wrong (verifier, proof generator, PL definitions, etc.)

Proof Generation: Complicated …

top-level proof goal $\Gamma^{L} \vdash \varphi_{\text{pre}} \rightsquigarrow \varphi_{\text{post}}$



$$\bigwedge_{(\psi_1 \Rightarrow \psi_2) \in A} \Box \left(\forall FV(\psi_1, \psi_2).\, \psi_1 \Rightarrow^{+}_{reach} \psi_2\right)$$
$$\wedge \bigwedge_{(\psi_1 \Rightarrow \psi_2) \in C} \circ\Box \left(\forall FV(\psi_1, \psi_2).\, \psi_1 \Rightarrow^{+}_{reach} \psi_2\right) \rightarrow \left(\varphi \Rightarrow^{\triangle}_{reach} \psi\right)$$

$$\left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \Rightarrow_{exec}$$
$$\left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right)$$

sub-goal A

$$\left(t_j^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \rightarrow \left(lhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right)$$
$$\left(rhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right) \rightarrow \left(t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right)$$

sub-goal B

$$\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi)$$
$$\rightarrow \varphi' \Rightarrow_{reach} \varphi''$$

sub-goal C

…

…

…

… but none of the above needs to be trusted.

Evaluation

We tested on 3 PL paradigms:

- imperative
- register-based
- functional

Reduced K trust base
(~120k lines of Haskell)

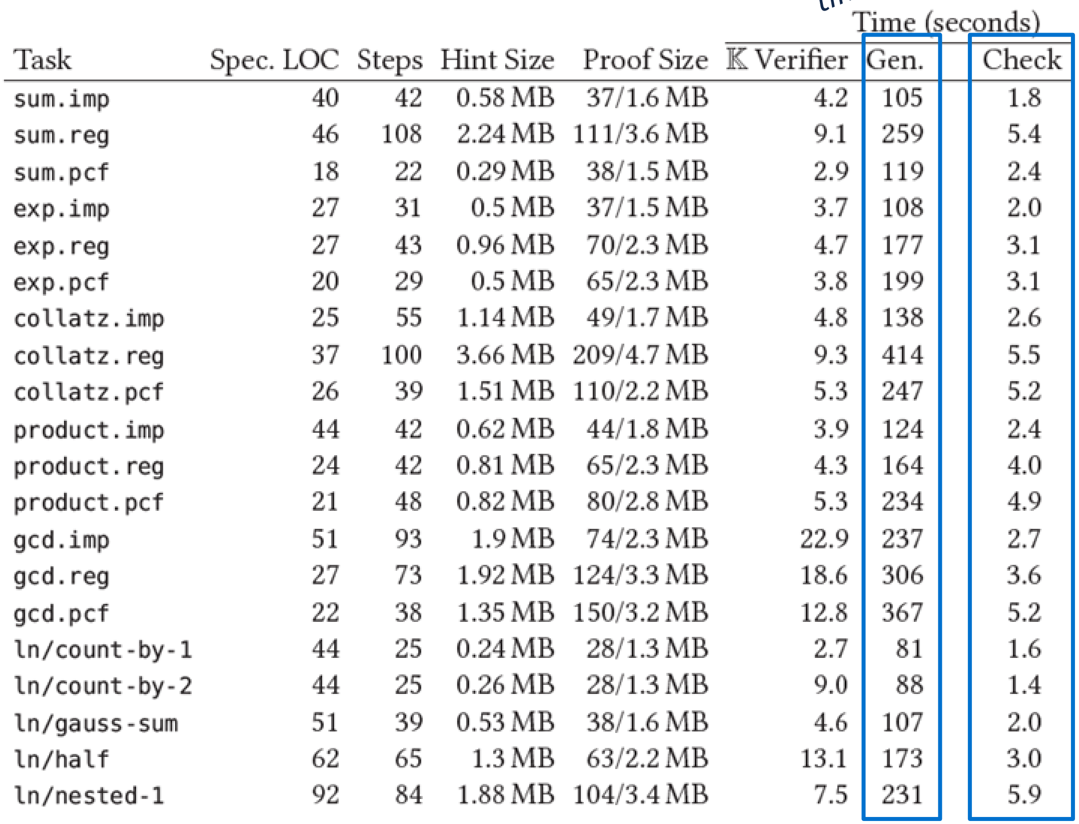
Found issues in K
(missing axioms etc.)

Future work

- Apply it to more PLs

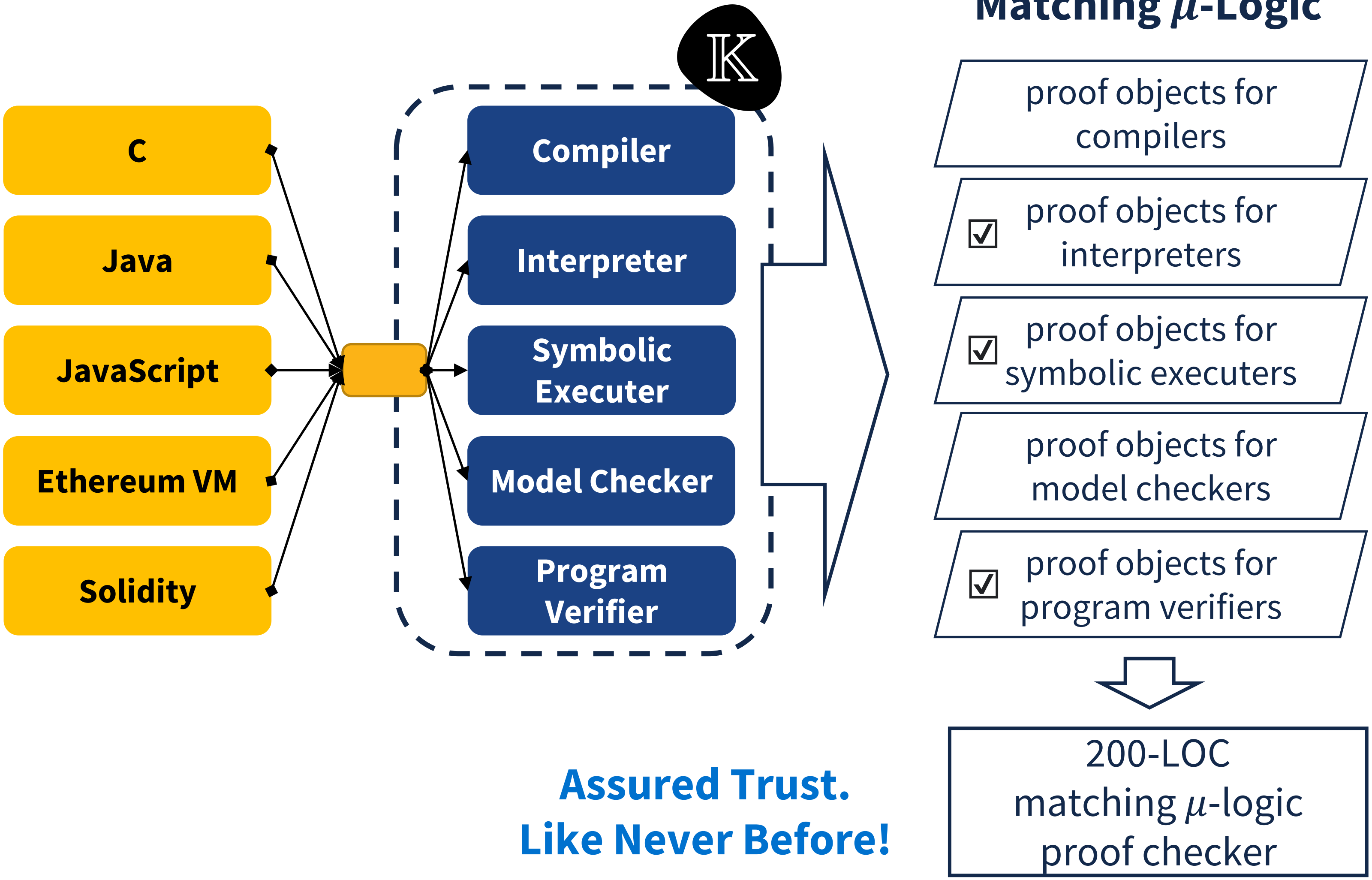
proof generation time

proof checking time



Task	Spec. LOC	Steps	Hint Size	Proof Size	Time (seconds)		
					𝕂 Verifier	Gen.	Check
sum.imp	40	42	0.58 MB	37/1.6 MB	4.2	105	1.8
sum.reg	46	108	2.24 MB	111/3.6 MB	9.1	259	5.4
sum.pcf	18	22	0.29 MB	38/1.5 MB	2.9	119	2.4
exp.imp	27	31	0.5 MB	37/1.5 MB	3.7	108	2.0
exp.reg	27	43	0.96 MB	70/2.3 MB	4.7	177	3.1
exp.pcf	20	29	0.5 MB	65/2.3 MB	3.8	199	3.1
collatz.imp	25	55	1.14 MB	49/1.7 MB	4.8	138	2.6
collatz.reg	37	100	3.66 MB	209/4.7 MB	9.3	414	5.5
collatz.pcf	26	39	1.51 MB	110/2.2 MB	5.3	247	5.2
product.imp	44	42	0.62 MB	44/1.8 MB	3.9	124	2.4
product.reg	24	42	0.81 MB	65/2.3 MB	4.3	164	4.0
product.pcf	21	48	0.82 MB	80/2.8 MB	5.3	234	4.9
gcd.imp	51	93	1.9 MB	74/2.3 MB	22.9	237	2.7
gcd.reg	27	73	1.92 MB	124/3.3 MB	18.6	306	3.6
gcd.pcf	22	38	1.35 MB	150/3.2 MB	12.8	367	5.2
ln/count-by-1	44	25	0.24 MB	28/1.3 MB	2.7	81	1.6
ln/count-by-2	44	25	0.26 MB	28/1.3 MB	9.0	88	1.4
ln/gauss-sum	51	39	0.53 MB	38/1.6 MB	4.6	107	2.0
ln/half	62	65	1.3 MB	63/2.2 MB	13.1	173	3.0
ln/nested-1	92	84	1.88 MB	104/3.4 MB	7.5	231	5.9

Conclusion: Matching μ-Logic as A Unifying Foundation for Programming


C
Java
JavaScript
Ethereum VM
Solidity
𝕂
Compiler
Interpreter
Symbolic Executer
Model Checker
Program Verifier
Matching μ-Logic
proof objects for compilers
☑ proof objects for interpreters
☑ proof objects for symbolic executers
proof objects for model checkers
☑ proof objects for program verifiers
200-LOC matching μ-logic proof checker

Assured Trust.
Like Never Before!

Thank you

Xiaohong Chen
https://xchen.page
xc3@illinois.edu

Completeness of Matching Logic (without X or μ)

$$\begin{tikzcd}
\begin{array}{c} \Gamma \vdash_{\mathcal{H}} \varphi \\ (\text{Definition } \boxed{3.2}) \end{array} \arrow[r, Rightarrow, shift left=2, "\text{Soundness}"] \arrow[dr, phantom, "\text{diagram commutes if } \Gamma = \emptyset"] & \begin{array}{c} \Gamma \vDash \varphi \\ (\text{Definition } \boxed{2.48}) \end{array} \arrow[l, Rightarrow, shift left=2, "\text{if } \Gamma \text{ includes definedness}", "\text{Definedness Completeness}"'] \\
\begin{array}{c} \Gamma \vdash^{loc}_{\mathcal{H}} \varphi \\ (\text{Definition } \boxed{3.3}) \end{array} \arrow[u, Rightarrow] \arrow[r, Rightarrow, shift left=2, "\text{Local Soundness}"] & \begin{array}{c} \Gamma \vDash^{loc} \varphi \\ (\text{Definition } \boxed{3.3}) \end{array} \arrow[u, Rightarrow] \arrow[l, Rightarrow, shift left=2, "\text{Local Completeness}"']
\end{tikzcd}$$

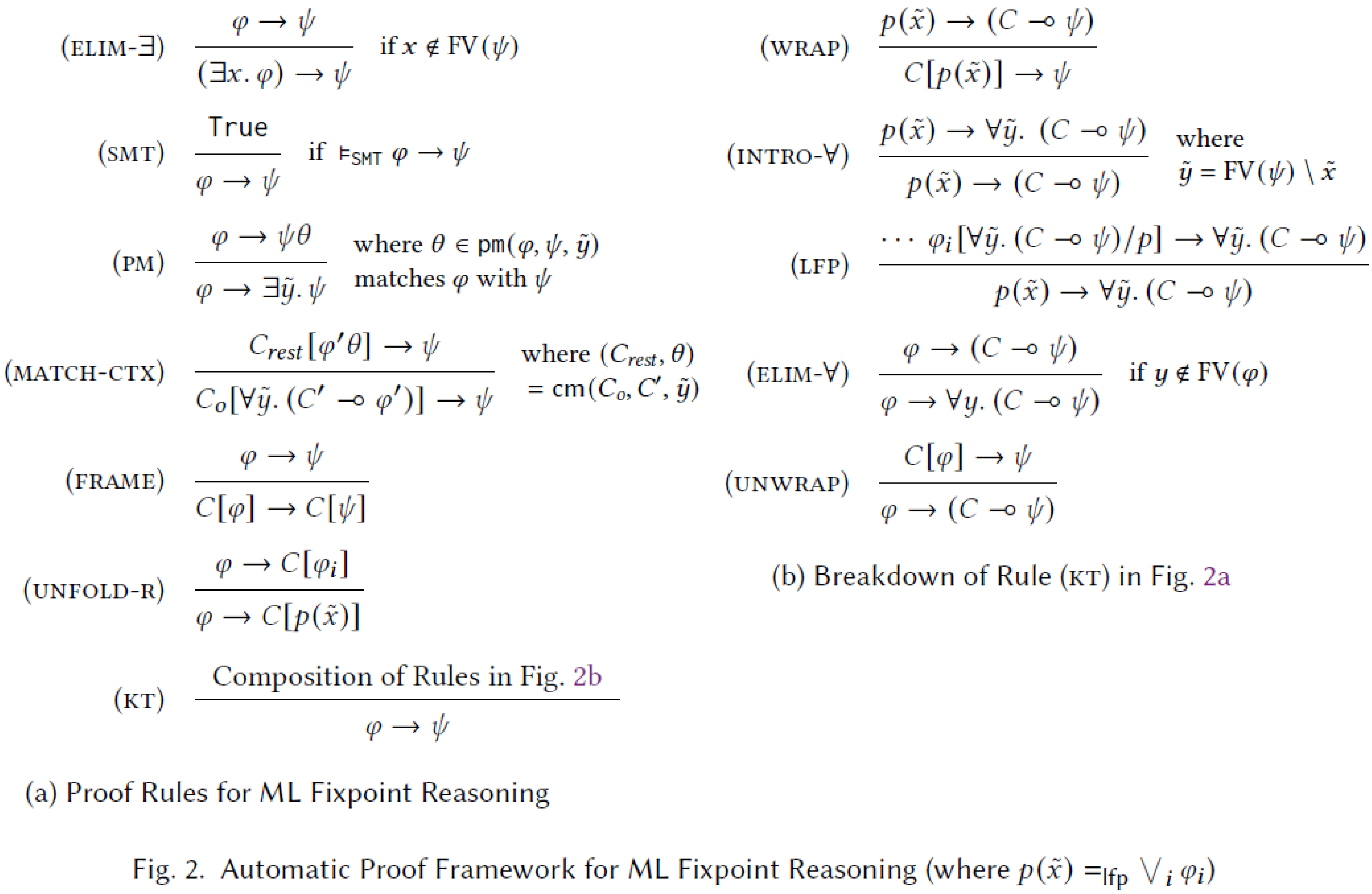
Figure 3.1: Known Relation among $\vDash$, $\vDash^{loc}$, $\vdash_{\mathcal{H}}$, and $\vdash^{loc}_{\mathcal{H}}$

Definition 3.3. Let Γ be a theory and φ be a pattern. The *local provability relation* $\Gamma \vdash^{loc}_{\mathcal{H}} \varphi$ holds iff there exists a finite subset $\Delta \subseteq \Gamma$ such that $\emptyset \vdash_{\mathcal{H}} \bigwedge \Delta \to \varphi$, where $\bigwedge \Delta$ is the conjunction of all patterns in Δ. We let $\bigwedge \emptyset$ be $\top$. The *local validity relation* $\Gamma \vDash^{loc} \varphi$ holds iff for any model M, any valuation ρ, and any element $a \in M$, $a \in |\psi|_{M,\rho}$ for all $\psi \in \Gamma$ implies $a \in |\varphi|_{M,\rho}$.

Reasoning Fixpoints within Contexts

- Proof Goal: $ll(x,y) * list(y) \to list(x)$ consists of
  - A **fixpoint** $ll(x,y)$
  - A **context** $C[\square] \equiv \square * list(y)$
- We (WRAP) the context and move it to the RHS:
  - $ll(x,y) \to \underbrace{\exists h{:}\, Heap.\left(h \land \left(h * list(y) \to list(x)\right)\right)}_{\text{The set of all heaps h such that h*list(y)->list(x).}}$
- We call the above RHS a **contextual implication**, abbreviated:
  - $ll(x,y) \to \left(C \multimap list(x)\right)$
- Now, LHS is a fixpoint and we can apply (LFP) in the usual way.

Automatic Proof Strategies



(ELIM-∃) $$\frac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi} \quad \text{if } x \notin FV(\psi)$$

(SMT) $$\frac{\mathsf{True}}{\varphi \to \psi} \quad \text{if } \models_{\mathsf{SMT}} \varphi \to \psi$$

(PM) $$\frac{\varphi \to \psi\theta}{\varphi \to \exists \tilde{y}.\, \psi} \quad \text{where } \theta \in \mathsf{pm}(\varphi, \psi, \tilde{y}) \text{ matches } \varphi \text{ with } \psi$$

(MATCH-CTX) $$\frac{C_{rest}[\varphi'\theta] \to \psi}{C_o[\forall \tilde{y}.\, (C' \multimap \varphi')] \to \psi} \quad \text{where } (C_{rest}, \theta) = \mathsf{cm}(C_o, C', \tilde{y})$$

(FRAME) $$\frac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$$

(UNFOLD-R) $$\frac{\varphi \to C[\varphi_i]}{\varphi \to C[p(\tilde{x})]}$$

(KT) $$\frac{\text{Composition of Rules in Fig. 2b}}{\varphi \to \psi}$$

(a) Proof Rules for ML Fixpoint Reasoning

(WRAP) $$\frac{p(\tilde{x}) \to (C \multimap \psi)}{C[p(\tilde{x})] \to \psi}$$

(INTRO-∀) $$\frac{p(\tilde{x}) \to \forall \tilde{y}.\ (C \multimap \psi)}{p(\tilde{x}) \to (C \multimap \psi)} \quad \text{where } \tilde{y} = FV(\psi) \setminus \tilde{x}$$

(LFP) $$\frac{\cdots\ \varphi_i[\forall \tilde{y}.\, (C \multimap \psi)/p] \to \forall \tilde{y}.\, (C \multimap \psi)}{p(\tilde{x}) \to \forall \tilde{y}.\, (C \multimap \psi)}$$

(ELIM-∀) $$\frac{\varphi \to (C \multimap \psi)}{\varphi \to \forall y.\, (C \multimap \psi)} \quad \text{if } y \notin FV(\varphi)$$

(UNWRAP) $$\frac{C[\varphi] \to \psi}{\varphi \to (C \multimap \psi)}$$

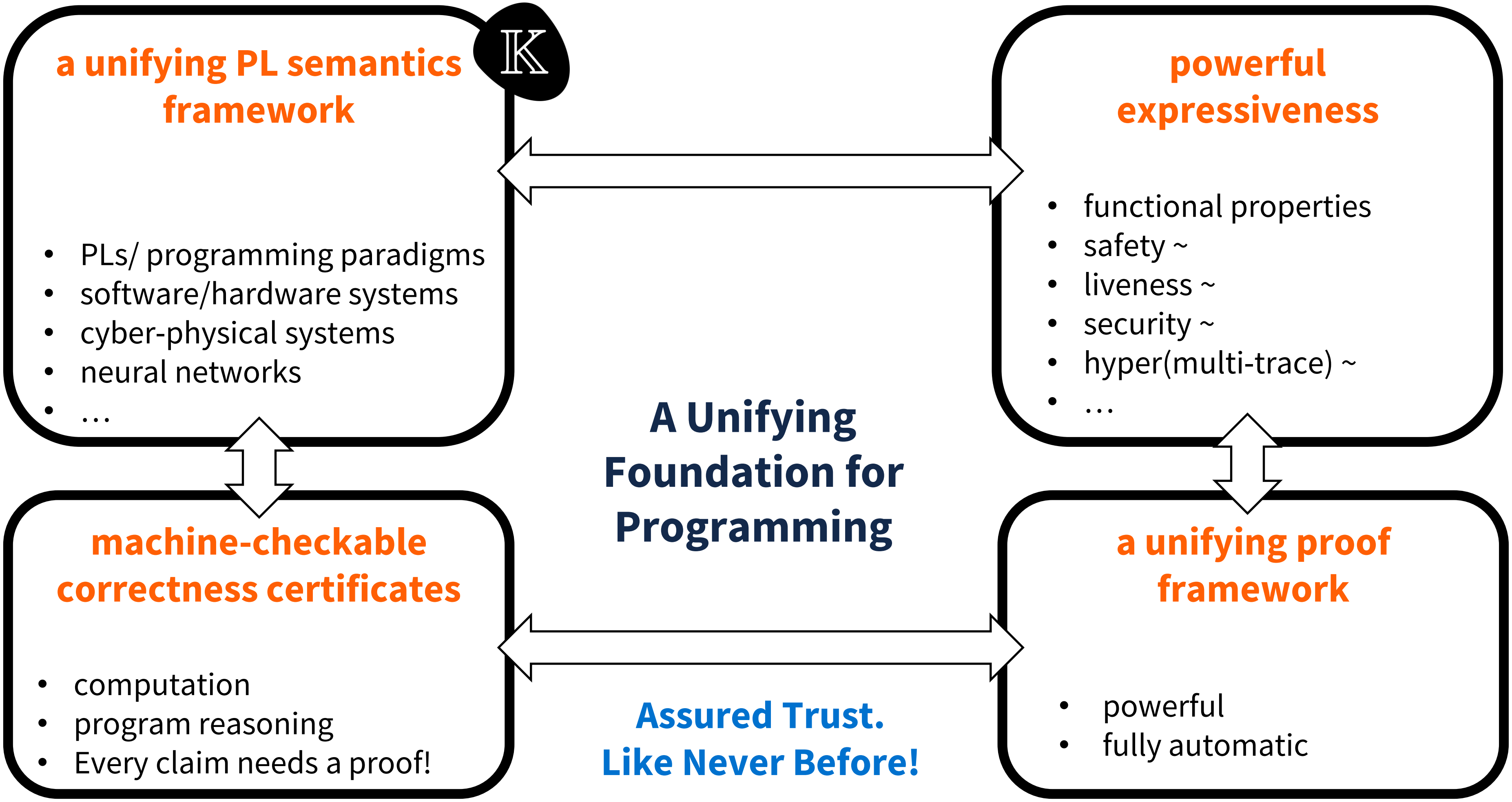
(b) Breakdown of Rule (KT) in Fig. 2a

Fig. 2. Automatic Proof Framework for ML Fixpoint Reasoning (where $p(\tilde{x}) =_{\mathsf{lfp}} \bigvee_i \varphi_i$)

Reduction to MSO

$$\begin{aligned}
MSO(\varphi) &= \forall r \,.\, MSO_2(\varphi, r) \\
MSO_2(x, r) &= x = r \\
MSO_2(\sigma(\varphi_1, \ldots, \varphi_n), r) &= \exists r_1 \ldots \exists r_n \,.\, MSO_2(\varphi_i, r_i) \wedge \pi_\sigma(r_1, \ldots, r_n, r) \\
MSO_2(\neg\varphi, r) &= \neg MSO_2(\varphi, r) \\
MSO_2(\varphi_1 \wedge \varphi_2, r) &= MSO_2(\varphi_1, r) \wedge MSO_2(\varphi_2, r) \\
MSO_2(\exists x \,.\, \varphi, r) &= \exists x \,.\, MSO_2(\varphi, r) \\
MSO_2(X, r) &= X(r) \\
MSO_2(\mu X \,.\, \varphi, r) &= \forall X \,.\, (\forall r' \,.\, MSO_2(\varphi, r') \rightarrow X(r')) \rightarrow X(r)
\end{aligned}$$

Our Vision



a unifying PL semantics framework

- PLs/ programming paradigms
- software/hardware systems
- cyber-physical systems
- neural networks
- …

powerful expressiveness

- functional properties
- safety ~
- liveness ~
- security ~
- hyper(multi-trace) ~
- …

A Unifying Foundation for Programming

machine-checkable correctness certificates

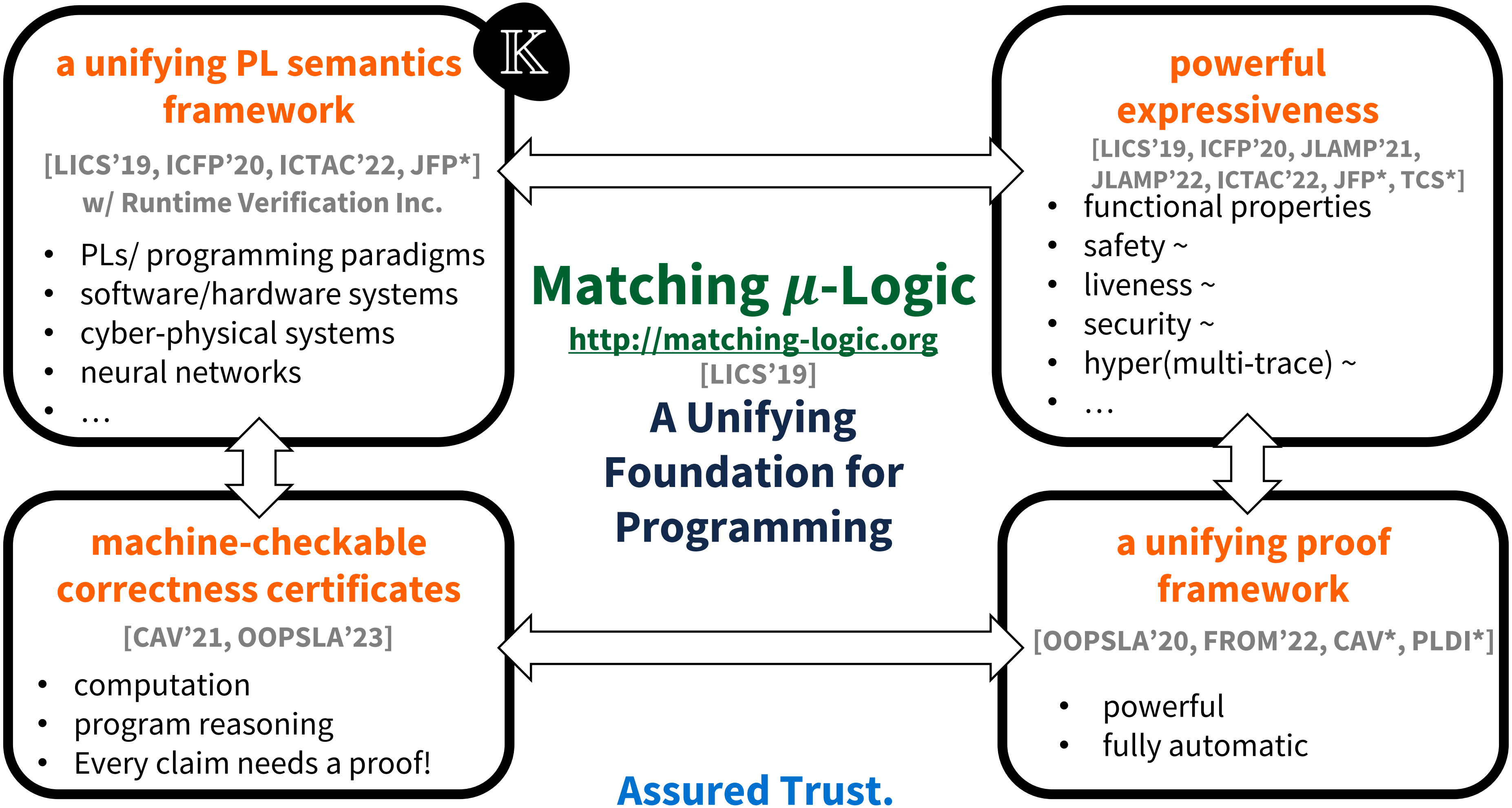
- computation
- program reasoning
- Every claim needs a proof!

Assured Trust.
Like Never Before!

a unifying proof framework

- powerful
- fully automatic

My PhD Work



a unifying PL semantics framework

[LICS'19, ICFP'20, ICTAC'22, JFP*]
w/ Runtime Verification Inc.

- PLs/ programming paradigms
- software/hardware systems
- cyber-physical systems
- neural networks
- …

powerful expressiveness

[LICS'19, ICFP'20, JLAMP'21, JLAMP'22, ICTAC'22, JFP*, TCS*]

- functional properties
- safety ~
- liveness ~
- security ~
- hyper(multi-trace) ~
- …

Matching μ-Logic

http://matching-logic.org

[LICS'19]

A Unifying Foundation for Programming

machine-checkable correctness certificates

[CAV'21, OOPSLA'23]

- computation
- program reasoning
- Every claim needs a proof!

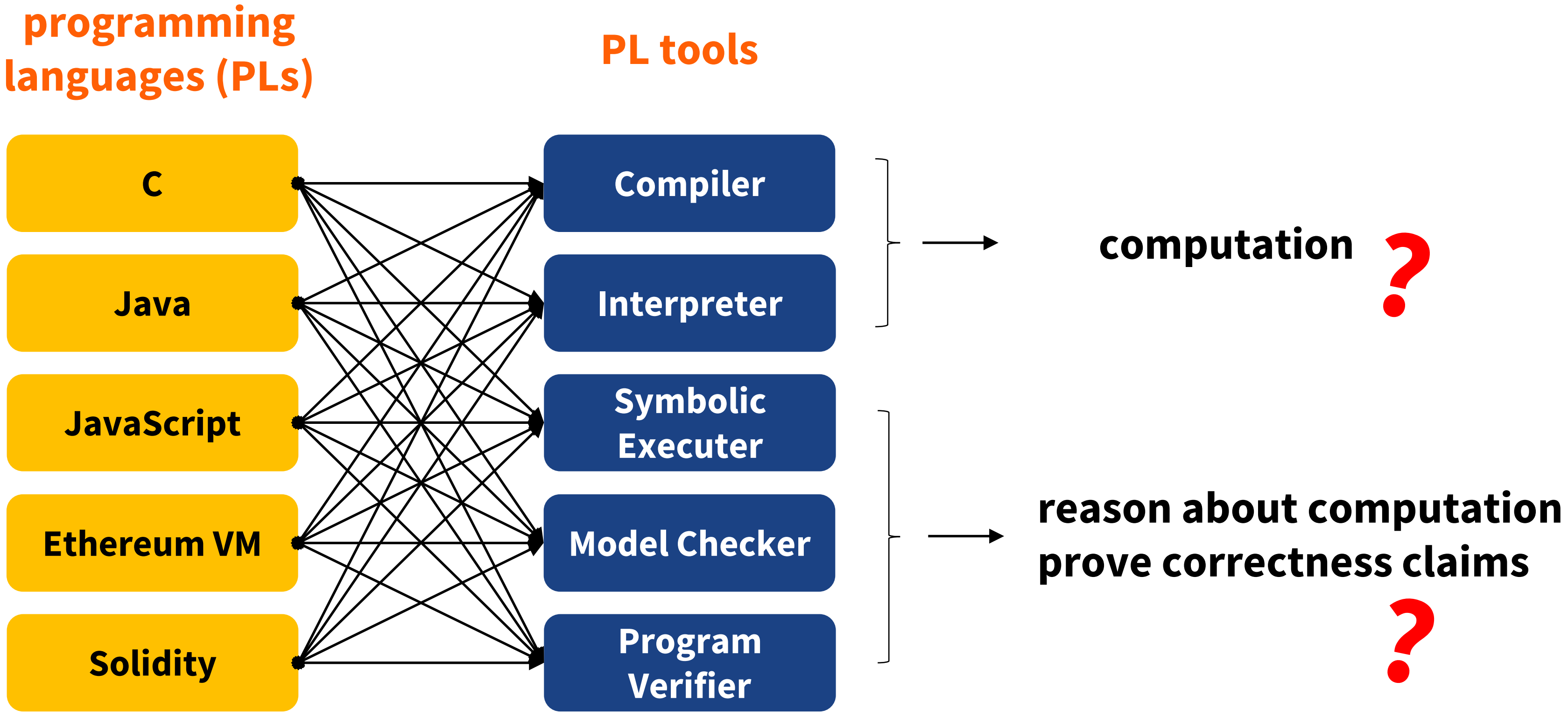
a unifying proof framework

[OOPSLA'20, FROM'22, CAV*, PLDI*]

- powerful
- fully automatic

**Assured Trust.
Like Never Before!**

State-of-the-Art: A Lot of Complexity



programming languages (PLs)

PL tools

C

Java

JavaScript

Ethereum VM

Solidity

Compiler

Interpreter

Symbolic Executer

Model Checker

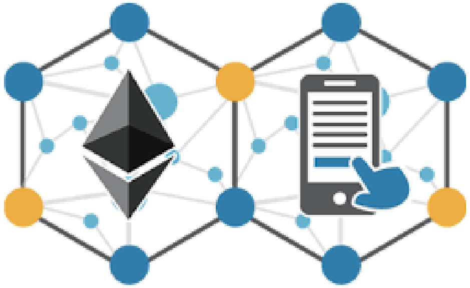
Program Verifier

computation ?

reason about computation
prove correctness claims
?

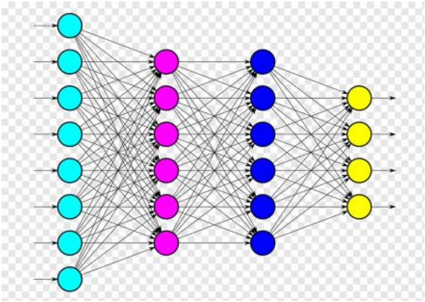
Safety/Mission-Critical Computer Programs

autopilot airplanes



smart contracts

banking systems



AI systems

State-of-the-Art: A Lot of Complexity

Example: executing smart contracts

```
function _transfer(from, to, amount) {
  uint256 fromBalance = _balances[from];
  require(fromBalance >= amount);
  _balances[from] = fromBalance - amount;
  _balances[to] += amount;
  emit Transfer(from, to, amount);
}
```

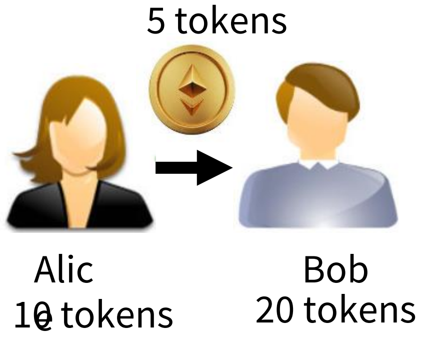
"_transfer" function in a Solidity smart contract

solc

```
tag 8
  JUMPDEST          PUSH 40
  PUSH F•40         SWAP1
  CALLER            SHA3
  AND               DUP1
  PUSH 0            SLOAD
  SWAP1             CALLVALUE
  DUP2              ADD
  MSTORE            SWAP1
                    SSTORE
```

EVM opcodes

geth



"Alice and Bob had 10 tokens and 20 tokens resp.
Now they have 5 tokens and 25 tokens resp." ---- solc & geth

Need to trust solc (300k LOC) & geth (500k LOC).

State-of-the-Art: A Lot of Complexity

Example: verifying smart contracts

$$\boldsymbol{\Psi}_{\text{ERC20}} \equiv \begin{array}{c} \forall \boldsymbol{A}. \forall \boldsymbol{B}. \forall \boldsymbol{T}. \boldsymbol{A} \geq \boldsymbol{T} \rightarrow \\ \texttt{\_transfer(Alice, Bob, } \boldsymbol{T}\texttt{)} \Rightarrow \ . \\ \boldsymbol{balance}_{\boldsymbol{Alice}} \mapsto \boldsymbol{A} \Rightarrow (\boldsymbol{A}-\boldsymbol{T}) \\ \boldsymbol{balance}_{\boldsymbol{Bob}} \mapsto \boldsymbol{B} \Rightarrow (\boldsymbol{B}+\boldsymbol{T}) \end{array}$$

kframework
Verified

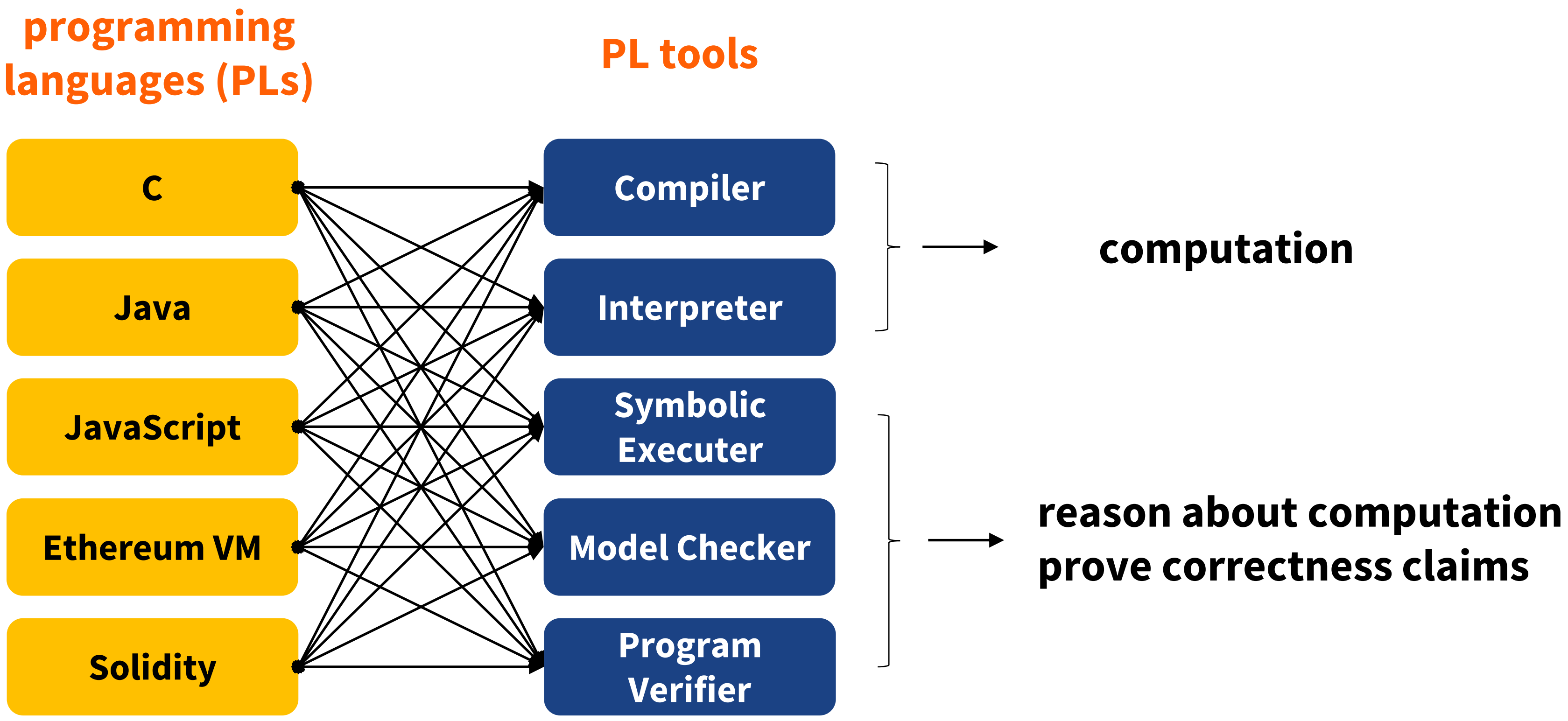
formal specification of `_transfer`

> “ For any values A, B, T,
> if $A \geq T$ and Alice and Bob have A and B tokens resp.
> then they will have $(A - T)$ and $(B + T)$ tokens resp. ” ---- kframework

☹ Need to trust kframework (500k LOC).

State-of-the-Art: A Lot of Complexity

Claims to Trust



computation

reason about computation
prove correctness claims

100% Correctness; How?

- **sources of compromised correctness**
  - complex programming languages (unspecified behavior, implementation-defined behavior, compiler optimization, …)

    → **need a unifying specification framework**
  - cloud computing; outsourced computation; BYOD (Bring Your Own Device) scenarios; untrusted devices (mobiles, tablets; …)

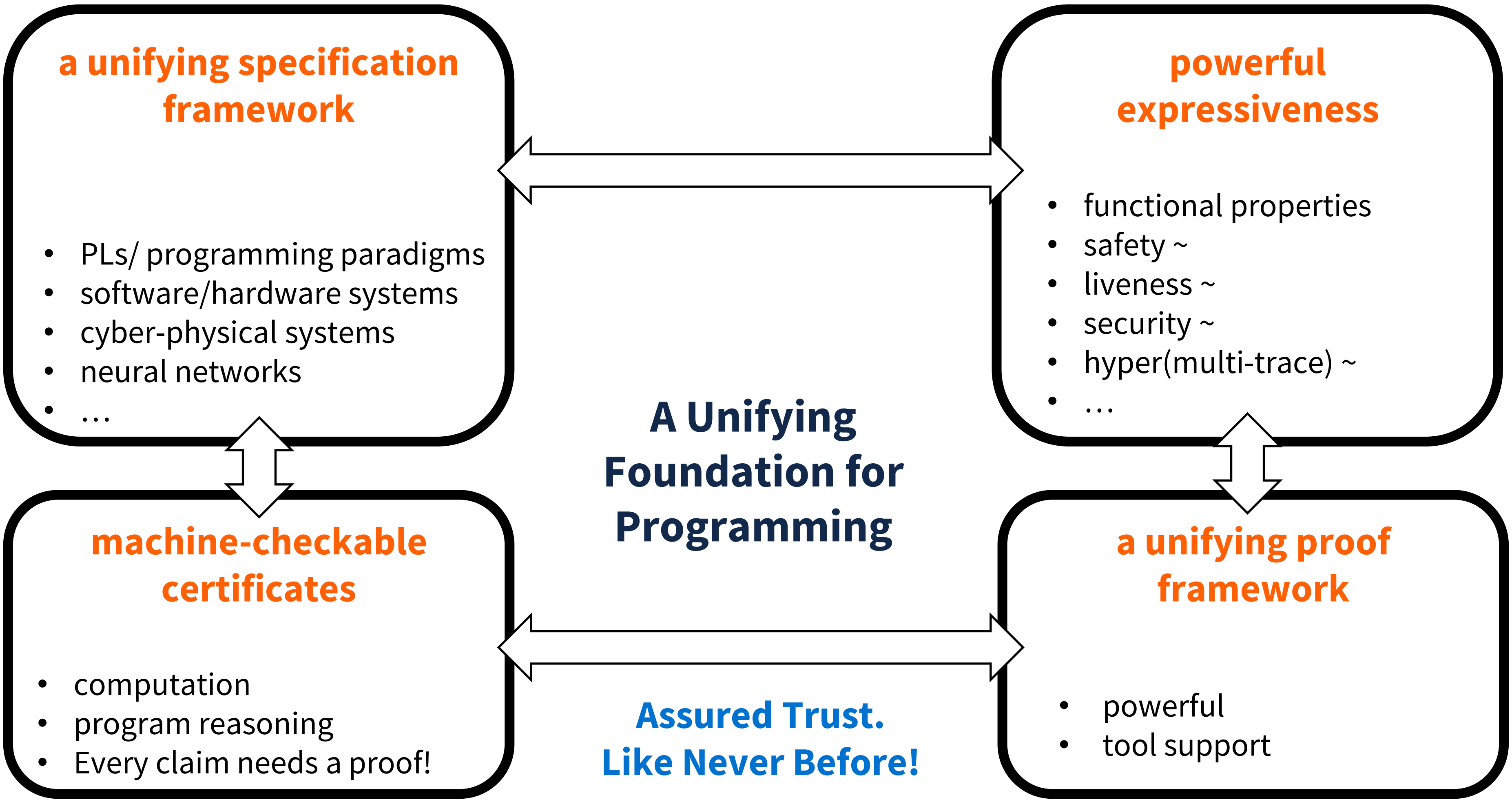
    → **need machine-checkable certificates for all computation results**
  - various "correctness" (functional, safety, liveness, security, termination, …)

    → **need a formal language with powerful expressiveness**
  - specialized program reasoning tools ("Prove Property X for Programs in L")

    → **need a unifying proof framework**

We need a unifying foundation for programming.

Vision



a unifying specification framework

- PLs/ programming paradigms
- software/hardware systems
- cyber-physical systems
- neural networks
- …

powerful expressiveness

- functional properties
- safety ~
- liveness ~
- security ~
- hyper(multi-trace) ~
- …

A Unifying Foundation for Programming

machine-checkable certificates

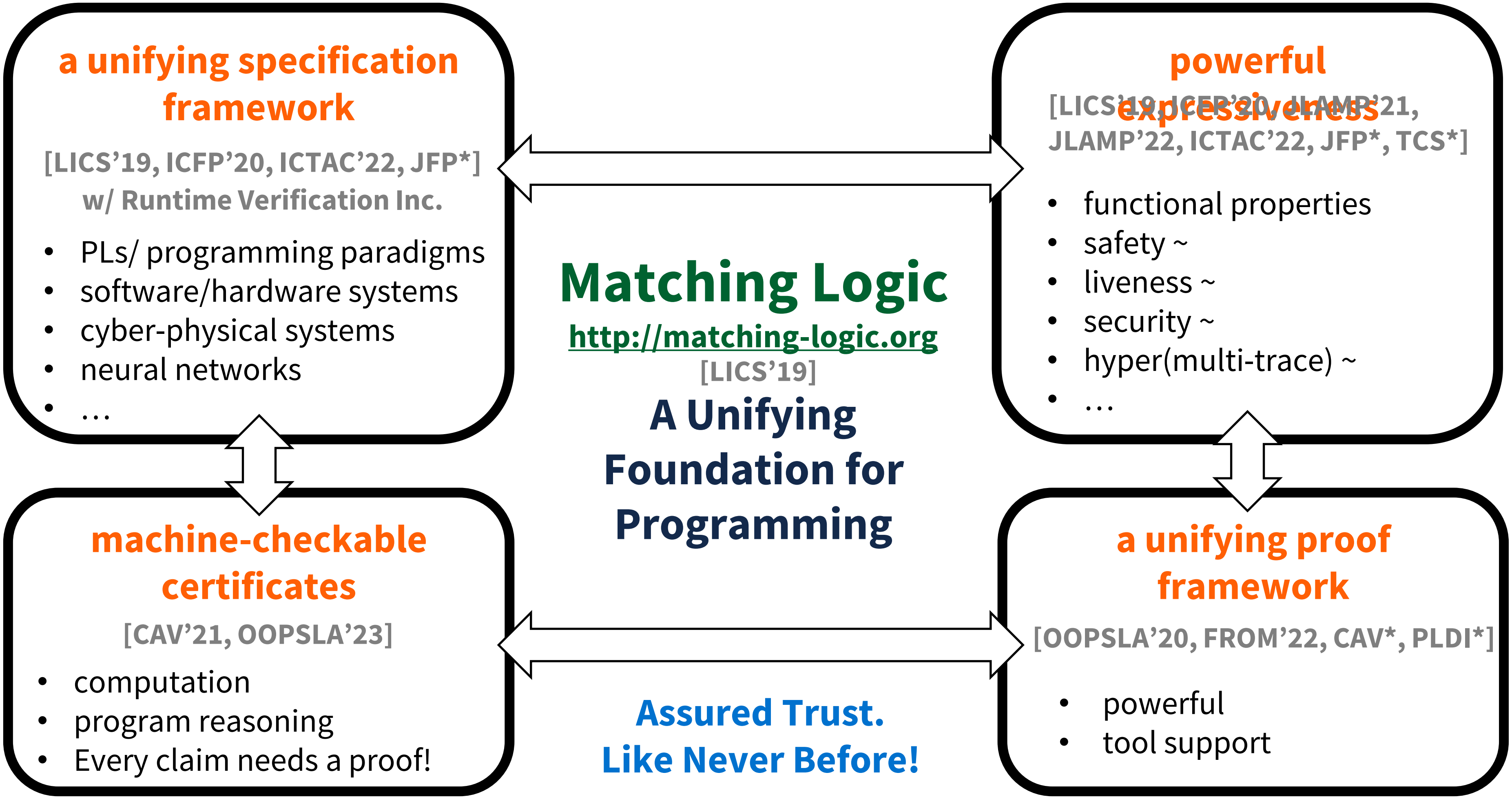
- computation
- program reasoning
- Every claim needs a proof!

a unifying proof framework

- powerful
- tool support

Assured Trust.
Like Never Before!

My PhD Work



a unifying specification framework

[LICS'19, ICFP'20, ICTAC'22, JFP*]
w/ Runtime Verification Inc.

- PLs/ programming paradigms
- software/hardware systems
- cyber-physical systems
- neural networks
- …

powerful expressiveness

[LICS'19, ICFP'20, JLAMP'21, JLAMP'22, ICTAC'22, JFP*, TCS*]

- functional properties
- safety ~
- liveness ~
- security ~
- hyper(multi-trace) ~
- …

Matching Logic

http://matching-logic.org

[LICS'19]

A Unifying Foundation for Programming

machine-checkable certificates

[CAV'21, OOPSLA'23]

- computation
- program reasoning
- Every claim needs a proof!

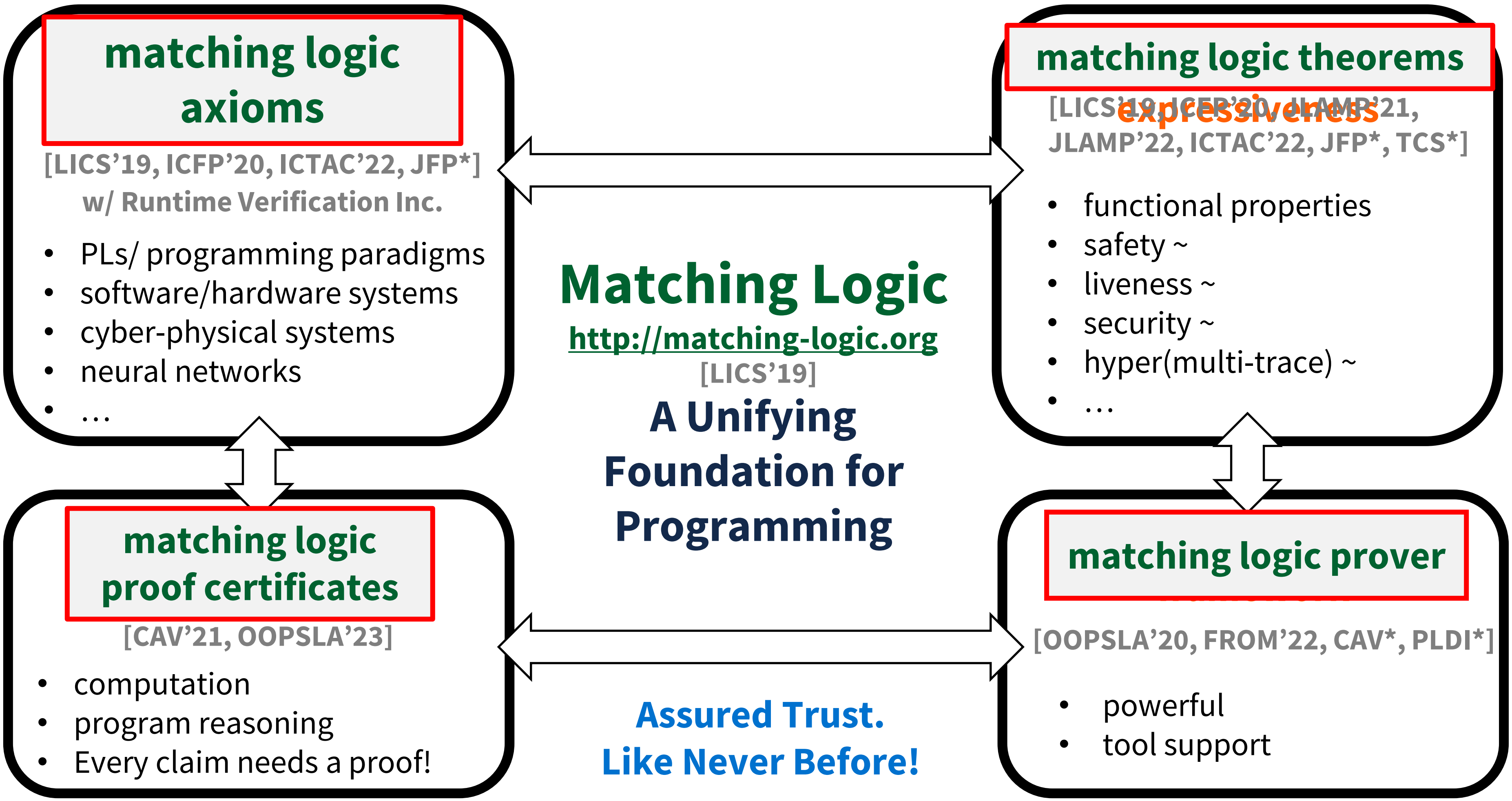
a unifying proof framework

[OOPSLA'20, FROM'22, CAV*, PLDI*]

- powerful
- tool support

Assured Trust. Like Never Before!

My PhD Work



matching logic axioms

[LICS'19, ICFP'20, ICTAC'22, JFP*]
w/ Runtime Verification Inc.

- PLs/ programming paradigms
- software/hardware systems
- cyber-physical systems
- neural networks
- …

matching logic theorems

[LICS'19, ICFP'20, JLAMP'21, JLAMP'22, ICTAC'22, JFP*, TCS*]

expressiveness

- functional properties
- safety ~
- liveness ~
- security ~
- hyper(multi-trace) ~
- …

Matching Logic

http://matching-logic.org

[LICS'19]

A Unifying Foundation for Programming

matching logic proof certificates

[CAV'21, OOPSLA'23]

- computation
- program reasoning
- Every claim needs a proof!

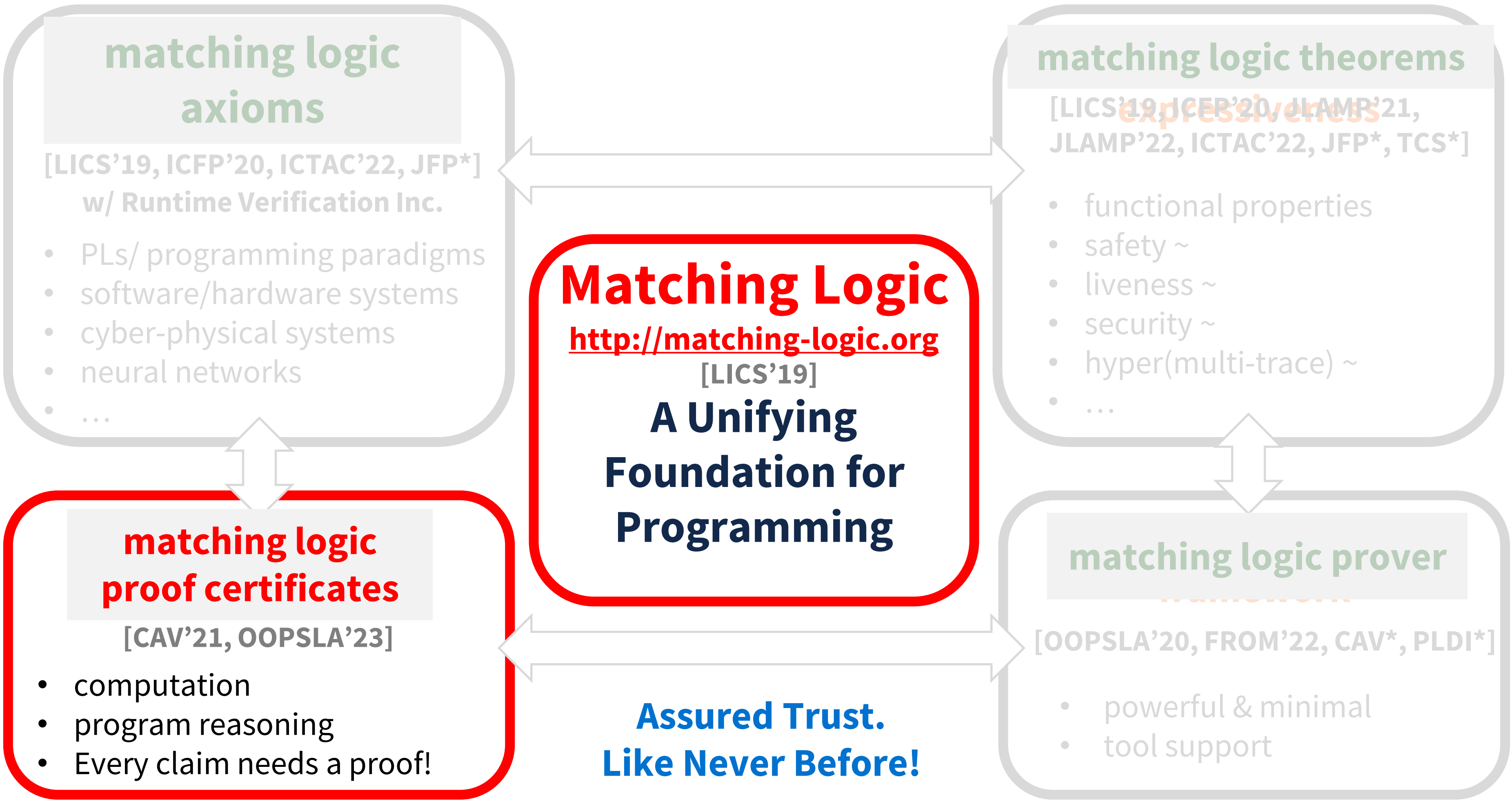
matching logic prover

[OOPSLA'20, FROM'22, CAV*, PLDI*]

- powerful
- tool support

**Assured Trust.
Like Never Before!**

In This Talk



matching logic axioms

[LICS'19, ICFP'20, ICTAC'22, JFP*]
w/ Runtime Verification Inc.

- PLs/ programming paradigms
- software/hardware systems
- cyber-physical systems
- neural networks
- …

matching logic theorems

expressiveness

[LICS'19, ICFP'20, JLAMP'21, JLAMP'22, ICTAC'22, JFP*, TCS*]

- functional properties
- safety ~
- liveness ~
- security ~
- hyper(multi-trace) ~
- …

Matching Logic

http://matching-logic.org

[LICS'19]

A Unifying Foundation for Programming

matching logic proof certificates

[CAV'21, OOPSLA'23]

- computation
- program reasoning
- Every claim needs a proof!

Assured Trust. Like Never Before!

matching logic prover framework

[OOPSLA'20, FROM'22, CAV*, PLDI*]

- powerful & minimal
- tool support

Why Matching Logic?

- **I studied existing logics, calculi, foundations, and semantics styles.**
  - first-order logic; second/higher-order logic; least fixpoint logic; modal logics; temporal logics (LTL, CTL, CTL*, …), λ-calculus; type systems (parametric, dependent, inductive, …); μ-calculus; Hoare logics; separation logics; dynamic logics; rewriting logic; reachability logic; equational logic; …
  - operational semantics (small-step, big-step, …); evaluation contexts; abstract machines (CC, CK, CEK, SECD, …); chemical abstract machines; axiomatic; algebraic (initial, final, …); continuations; denotational; …
- **But each of the above had limitations.**
  - Some only handle certain aspects of computation (e.g., execution only).
  - Some are "design patterns" (e.g., Hoare logics); re-design new logics for new PLs.
  - modularity, notation
- **Matching logic: keep advantages and avoid limitations**

What is Matching Logic?

A logic with first-order variables and quantifiers, polyadic modalities and function symbols, fixpoint operations, and top-level second-order universal quantifiers. [LICS'19]

matching logic formulas called **patterns:**

Matching Logic Syntax
(minimal; only 7 constructs)

$$\varphi ::= \underbrace{\boxed{x \mid \sigma(\varphi_1, \ldots, \varphi_n)}}_{\text{structures}} \mid \underbrace{\boxed{\varphi_1 \land \varphi_2 \mid \neg\varphi}}_{\text{logical constraints}} \mid \underbrace{\boxed{\exists x.\, \varphi}}_{\substack{\text{abstraction} \\ \text{FO quantifiers}}} \mid \underbrace{\boxed{X \mid \mu X.\, \varphi}}_{\substack{\text{fixpoints} \\ \text{(in this talk)}}}$$

Matching Logic Fixpoints

- **inductive datatypes** [JLAMP'21, TCS*]
  - `type list = Nil | Cons of element * list`
  - $\mu L.\, \mathbf{Nil} \vee \exists x.\, \mathbf{Cons}(x, L)$
- **program execution** [CAV'21]
  - finite execution trace from t_{init} to t_{final}
  - $t_{\text{init}} \rightarrow \underbrace{\textbf{eventually } t_{\text{final}}}_{\mu S.\, t_{\text{final}} \vee (\textbf{next } S)}$
- **formal verification** [OOPSLA'23]
  - if φ_{pre} holds then φ_{post} holds on termination
  - $\varphi_{\text{pre}} \rightarrow \underbrace{\textbf{weak-eventually } \varphi_{\text{post}}}_{\nu S.\, \varphi_{\text{post}} \vee (\textbf{next } S)}$

"partial correctness" $\nu S.\, \varphi_{\text{post}} \vee (\textbf{next } S)$

- **algebraic specification (datatypes + equations)**

(Bergstra & Tucker 1982)
Any computable domain has a finite algebraic specification

Any computable domain has a finite **matching logic** axiomatization.

Further, we can design automatic proof strategies to reason about all these fixpoints. [OOPSLA'20, PLDI*]

Matching Logic Expressive Power

[LICS’19, OOPSLA’20, ICFP’20, JLAMP’21, JLAMP’22, JFP*, TCS*]

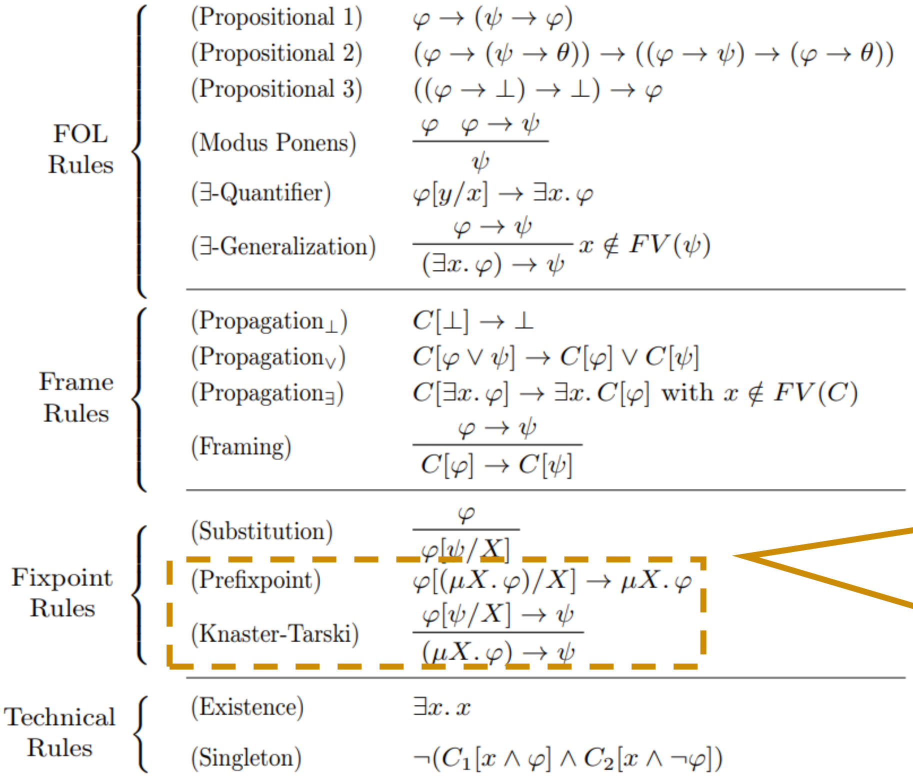
Important logics for program reasoning are all definable in matching logic.

- first-order logic
  - equality, membership, partial functions, definedness
- λ calculus
- dependent types (Coq & Agda)
- higher-order logic
- modal logic & temporal logics (LTL, CTL, CTL*)
- Hoare logics
- dynamic logics
- rewriting logic
- reachability logic
- separation logic
- μ-calculus
- inductive/co-inductive datatypes
- …

Proof assistants such as Coq & Agda become **methodologies** in matching logic.

[ICFP’20, extended ver. invited to JFP]

Matching Logic Proof System



FOL Rules	(Propositional 1)	$\varphi \to (\psi \to \varphi)$
	(Propositional 2)	$(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))$
	(Propositional 3)	$((\varphi \to \bot) \to \bot) \to \varphi$
	(Modus Ponens)	$\dfrac{\varphi \quad \varphi \to \psi}{\psi}$
	(∃-Quantifier)	$\varphi[y/x] \to \exists x.\, \varphi$
	(∃-Generalization)	$\dfrac{\varphi \to \psi}{(\exists x.\, \varphi) \to \psi}\ x \notin FV(\psi)$
Frame Rules	(Propagation$_\bot$)	$C[\bot] \to \bot$
	(Propagation$_\vee$)	$C[\varphi \vee \psi] \to C[\varphi] \vee C[\psi]$
	(Propagation$_\exists$)	$C[\exists x.\, \varphi] \to \exists x.\, C[\varphi]$ with $x \notin FV(C)$
	(Framing)	$\dfrac{\varphi \to \psi}{C[\varphi] \to C[\psi]}$
Fixpoint Rules	(Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
	(Prefixpoint)	$\varphi[(\mu X.\, \varphi)/X] \to \mu X.\, \varphi$
	(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{(\mu X.\, \varphi) \to \psi}$
Technical Rules	(Existence)	$\exists x.\, x$
	(Singleton)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$

$$\Gamma \vdash \varphi$$

specification (axioms)

theorem

very simple (15 proof rules)

$$\varphi[(\mu X.\varphi)/X] \leftrightarrow \mu X.\varphi$$

$$\frac{\varphi[\psi/X] \leftrightarrow \psi}{(\mu X.\varphi) \to \psi}$$

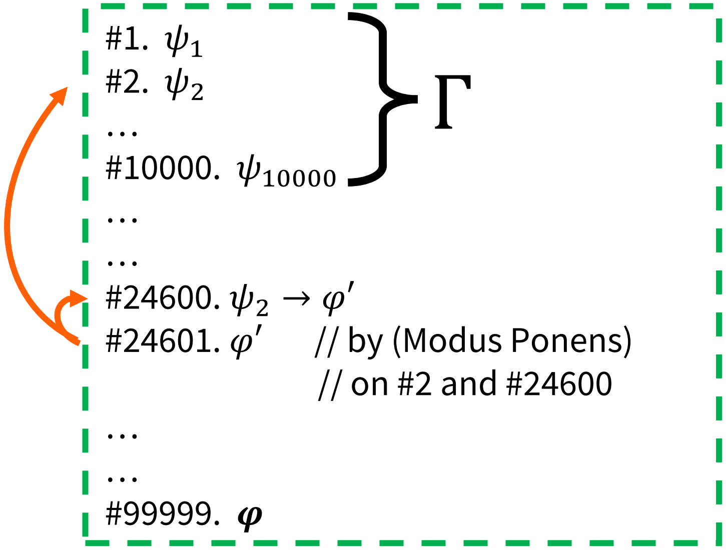
Knaster-Tarski Fixpoint Theorem incarnated in matching logic

Matching Logic Proof System

$$\boldsymbol{\Gamma \vdash \varphi}$$

specification
(axioms)

theorem



#1. ψ_1
#2. ψ_2
…
#10000. ψ_{10000} $\Bigg\}\,\Gamma$
…
…
#24600. $\psi_2 \to \varphi'$
#24601. φ' // by (Modus Ponens)
// on #2 and #24600
…
…
#99999. $\boldsymbol{\varphi}$

machine-checkable proof certificate

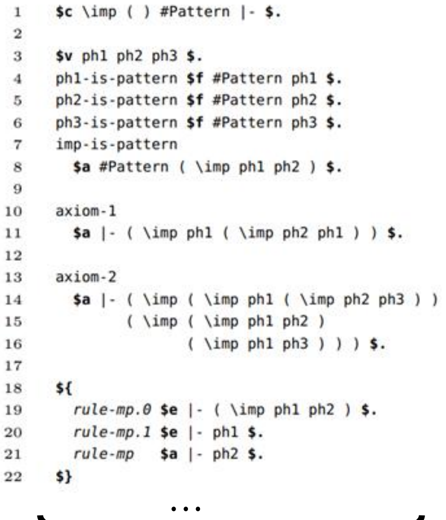
Matching Logic Proof Checker

- We use **metamath**
  - http://metamath.org [Megill & Wheeler]
  - encode matching logic proofs
  - mechanize proof checking

Small trust base to check proofs!

$$\underset{\text{axioms}}{\Gamma} \vdash \underset{\text{theorem}}{\varphi}$$

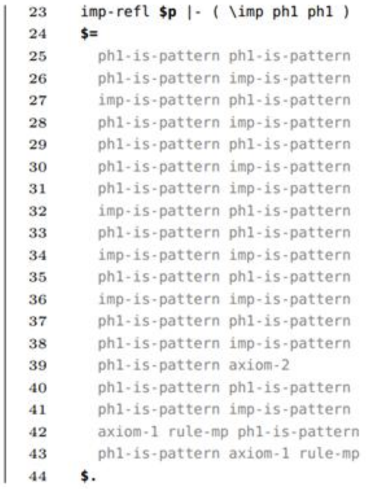
Reducing correctness to proof checking



```
$c \imp ( ) #Pattern |- $.

$v ph1 ph2 ph3 $.
ph1-is-pattern $f #Pattern ph1 $.
ph2-is-pattern $f #Pattern ph2 $.
ph3-is-pattern $f #Pattern ph3 $.
imp-is-pattern
  $a #Pattern ( \imp ph1 ph2 ) $.

axiom-1
  $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.

axiom-2
  $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
        ( \imp ( \imp ph1 ph2 )
               ( \imp ph1 ph3 ) ) ) $.

${
  rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
  rule-mp.1 $e |- ph1 $.
  rule-mp   $a |- ph2 $.
$}
```

…

definitions of
syntax & proof rules
(240 LOC in total)



```
imp-refl $p |- ( \imp ph1 ph1 )
$=
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern axiom-2
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  axiom-1 rule-mp ph1-is-pattern
  ph1-is-pattern axiom-1 rule-mp
$.
```

theorems and proofs
(machine checked)

Formal Verification: Two Approaches

A **traditional, language-specific** verifier for language L takes

- a program P in L and its formal spec φ_P

and checks whether P satisfies φ_P

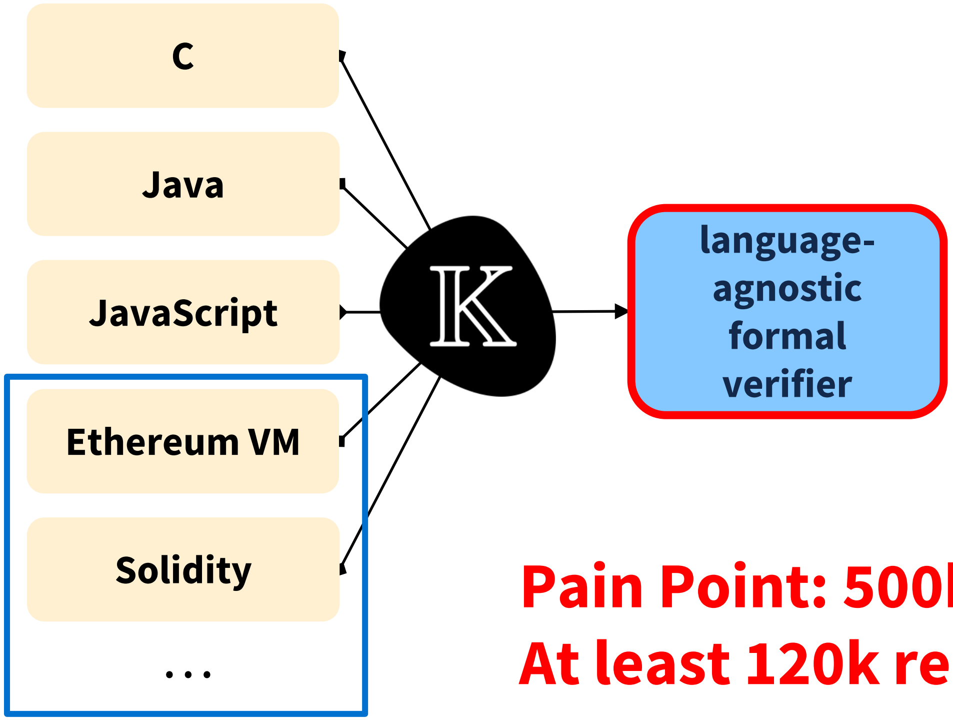
A **language-agnostic** formal verifier takes

- a program P in L and its formal spec φ_P
- the **formal specification of L** +1 abstraction level

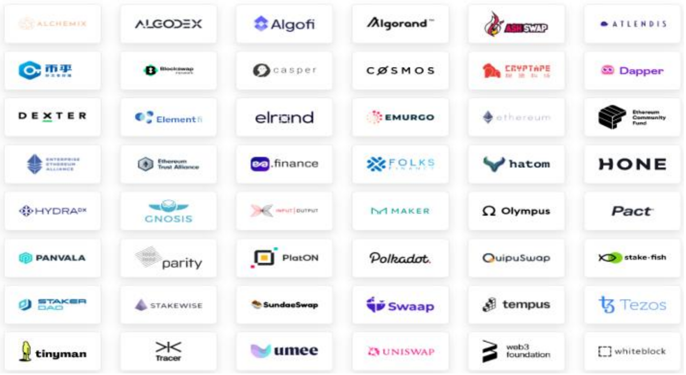
and checks whether P satisfies φ_P

+ more principled
+ highly re-usable

Example: K Framework (https://kframework.org)



50+ smart contracts, consensus protocols, VMs, etc.

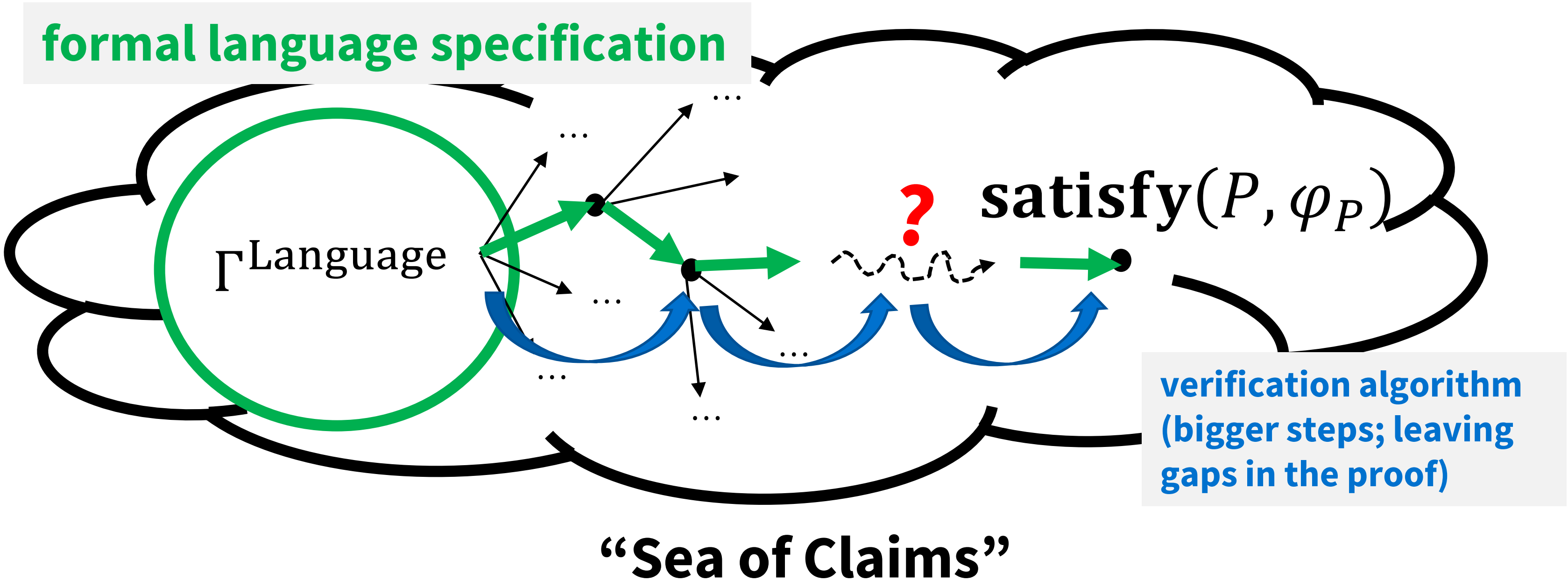


Pain Point: 500k-line code base to trust.
At least 120k related to verification.

… and it's growing

Viewpoint:

A formal verifier is a specialized, optimized, proof searcher.



formal language specification

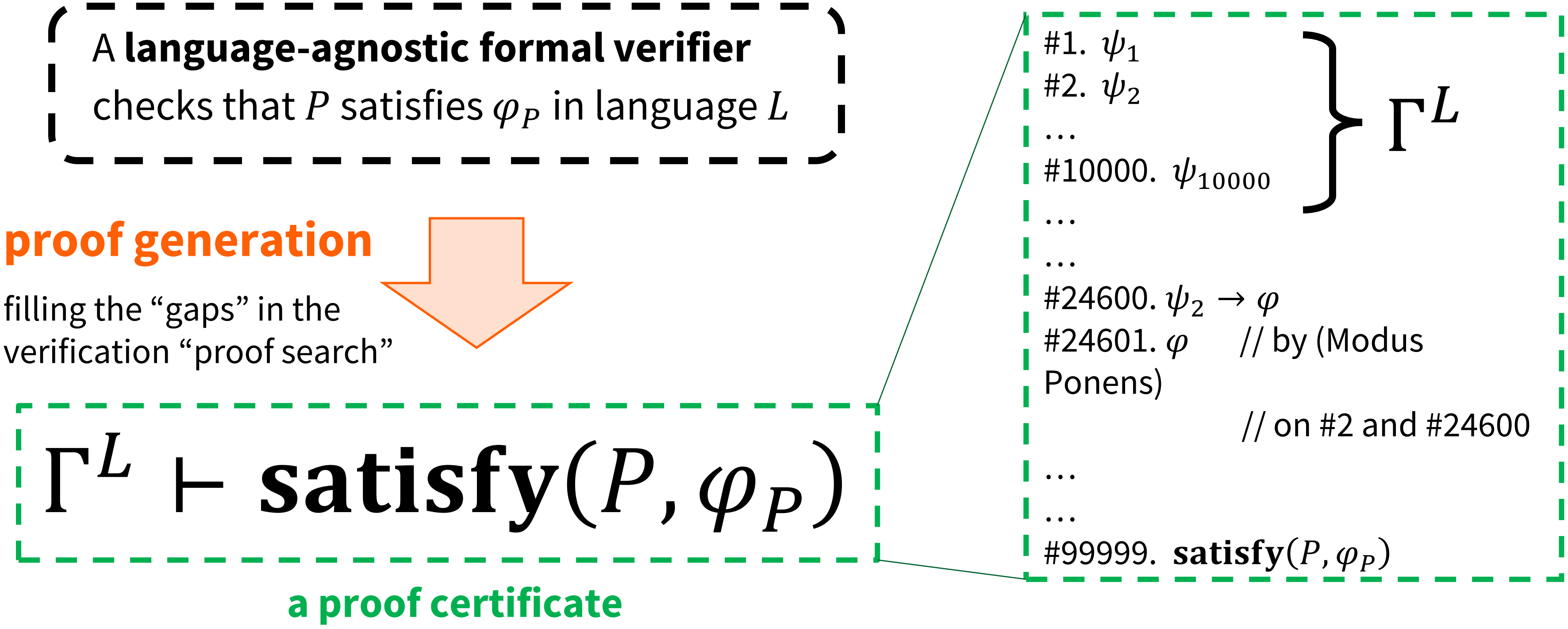
Γ^{Language}

$\textbf{satisfy}(P, \varphi_P)$

verification algorithm (bigger steps; leaving gaps in the proof)

“Sea of Claims”

Proof-Certifying Language-Agnostic Formal Verification



A **language-agnostic formal verifier** checks that P satisfies φ_P in language L

proof generation

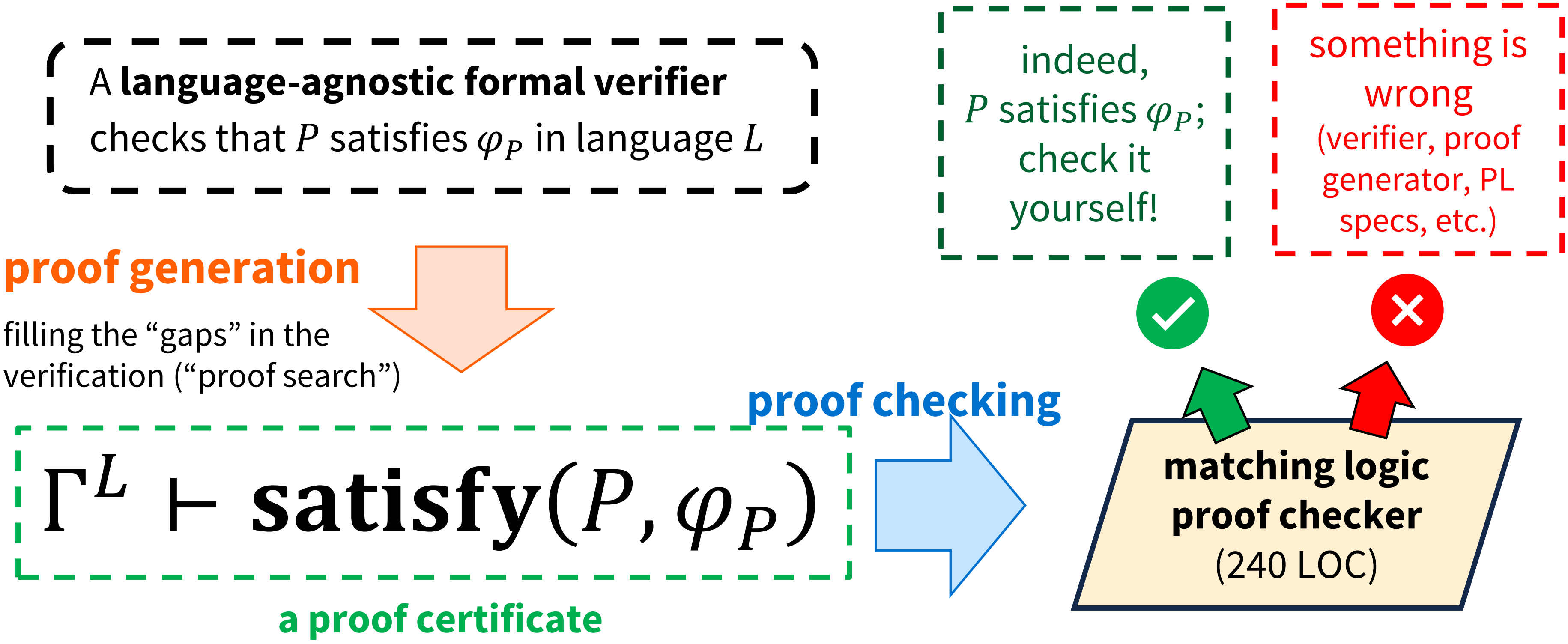
filling the "gaps" in the verification "proof search"

$$\Gamma^L \vdash \mathbf{satisfy}(P, \varphi_P)$$

a proof certificate

#1. ψ_1
#2. ψ_2
…
#10000. ψ_{10000}
$\Big\}\ \Gamma^L$
…
…
#24600. $\psi_2 \to \varphi$
#24601. φ // by (Modus Ponens)
// on #2 and #24600
…
…
#99999. $\mathbf{satisfy}(P, \varphi_P)$

Proof-Certifying Language-Agnostic Formal Verification



A **language-agnostic formal verifier** checks that P satisfies φ_P in language L

proof generation

filling the "gaps" in the verification ("proof search")

$$\Gamma^L \vdash \textbf{satisfy}(P, \varphi_P)$$

a proof certificate

proof checking

matching logic proof checker
(240 LOC)

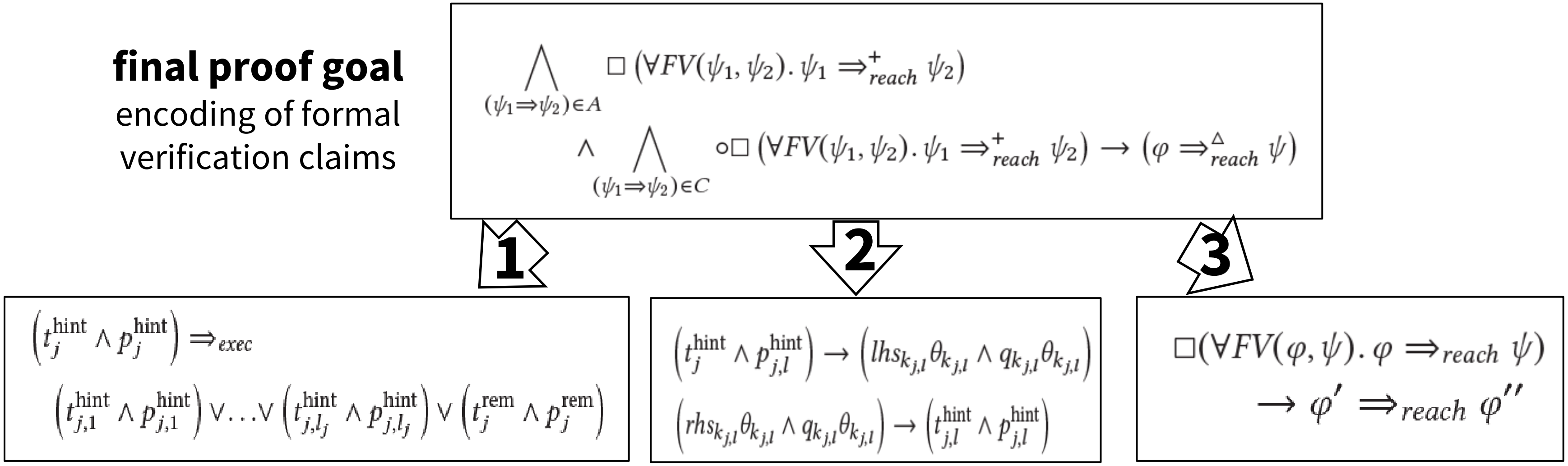
✓

indeed, P satisfies φ_P; check it yourself!

✗

something is wrong (verifier, proof generator, PL specs, etc.)

Proof Generation Procedures (technical details)



final proof goal
encoding of formal
verification claims

$$\bigwedge_{(\psi_1 \Rightarrow \psi_2) \in A} \Box \left(\forall FV(\psi_1, \psi_2).\, \psi_1 \Rightarrow^{+}_{reach} \psi_2\right)$$
$$\wedge \bigwedge_{(\psi_1 \Rightarrow \psi_2) \in C} \circ\Box \left(\forall FV(\psi_1, \psi_2).\, \psi_1 \Rightarrow^{+}_{reach} \psi_2\right) \to \left(\varphi \Rightarrow^{\triangle}_{reach} \psi\right)$$

$$\left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \Rightarrow_{exec}$$
$$\left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right)$$

sub-goals for
symbolic execution
(dynamic)

$$\left(t_j^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \to \left(lhs_{k_{j,l}} \theta_{k_{j,l}} \wedge q_{k_{j,l}} \theta_{k_{j,l}}\right)$$
$$\left(rhs_{k_{j,l}} \theta_{k_{j,l}} \wedge q_{k_{j,l}} \theta_{k_{j,l}}\right) \to \left(t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right)$$

sub-goals for
subsumptions/implications
(static)

$$\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi)$$
$$\to \varphi' \Rightarrow_{reach} \varphi''$$

sub-goals for
circularity/coinduction
(loops; recursion; etc.)

But, none of the above need to be trusted.

Evaluation

We tested on 3 PL paradigms:

- imperative
- register-based
- functional

Main Takeaways:

- large **Proof Size**
- fast **Proof Checking** (seconds)
- OK **Proof Generation** (minutes)

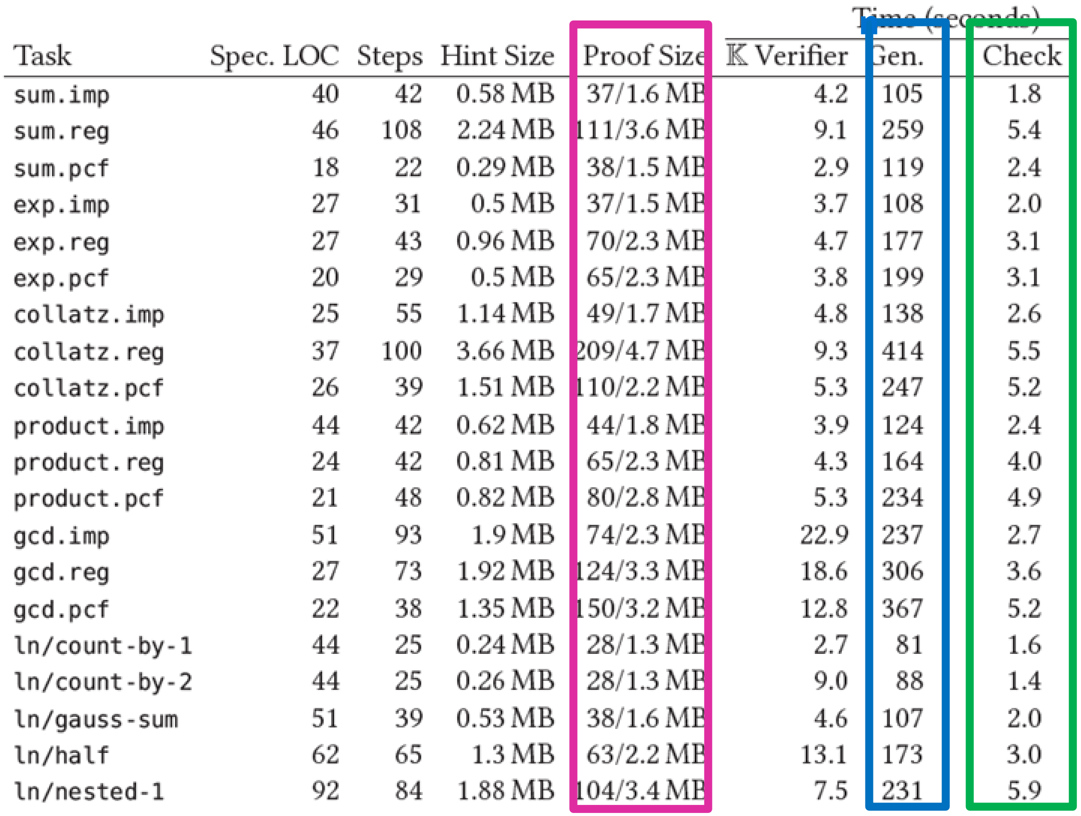
Found issues in K

(missing axioms/assumptions etc.)

Proof Size

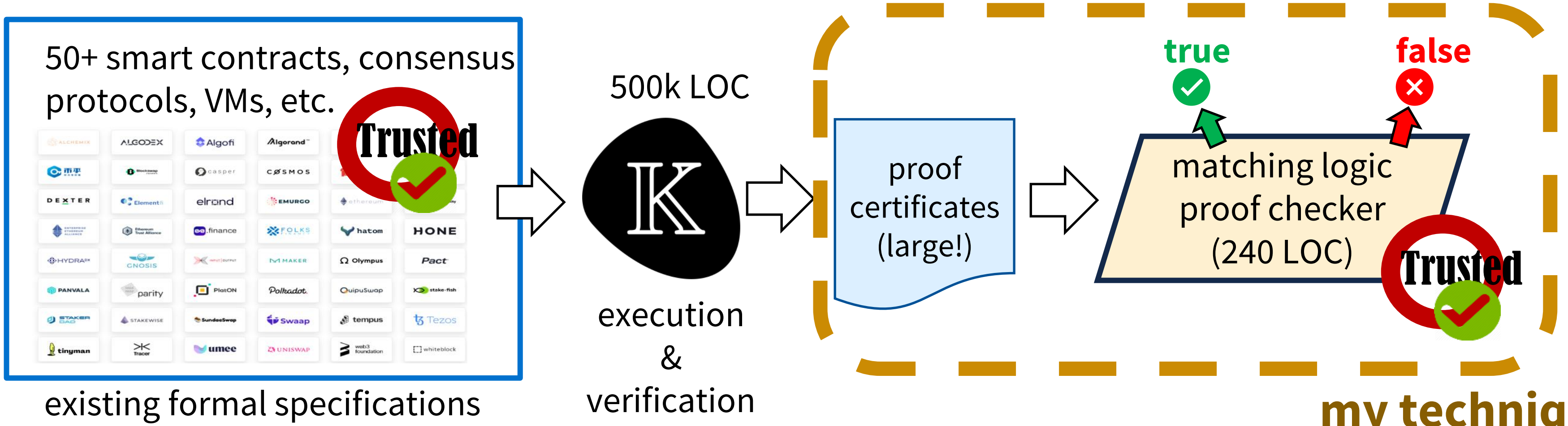
Proof Generatio

Proof Checking



					Time (seconds)		
Task	Spec. LOC	Steps	Hint Size	Proof Size	𝕂 Verifier	Gen.	Check
sum.imp	40	42	0.58 MB	37/1.6 MB	4.2	105	1.8
sum.reg	46	108	2.24 MB	111/3.6 MB	9.1	259	5.4
sum.pcf	18	22	0.29 MB	38/1.5 MB	2.9	119	2.4
exp.imp	27	31	0.5 MB	37/1.5 MB	3.7	108	2.0
exp.reg	27	43	0.96 MB	70/2.3 MB	4.7	177	3.1
exp.pcf	20	29	0.5 MB	65/2.3 MB	3.8	199	3.1
collatz.imp	25	55	1.14 MB	49/1.7 MB	4.8	138	2.6
collatz.reg	37	100	3.66 MB	209/4.7 MB	9.3	414	5.5
collatz.pcf	26	39	1.51 MB	110/2.2 MB	5.3	247	5.2
product.imp	44	42	0.62 MB	44/1.8 MB	3.9	124	2.4
product.reg	24	42	0.81 MB	65/2.3 MB	4.3	164	4.0
product.pcf	21	48	0.82 MB	80/2.8 MB	5.3	234	4.9
gcd.imp	51	93	1.9 MB	74/2.3 MB	22.9	237	2.7
gcd.reg	27	73	1.92 MB	124/3.3 MB	18.6	306	3.6
gcd.pcf	22	38	1.35 MB	150/3.2 MB	12.8	367	5.2
ln/count-by-1	44	25	0.24 MB	28/1.3 MB	2.7	81	1.6
ln/count-by-2	44	25	0.26 MB	28/1.3 MB	9.0	88	1.4
ln/gauss-sum	51	39	0.53 MB	38/1.6 MB	4.6	107	2.0
ln/half	62	65	1.3 MB	63/2.2 MB	13.1	173	3.0
ln/nested-1	92	84	1.88 MB	104/3.4 MB	7.5	231	5.9

Future Direction: Proof-Certifying Smart Contracts



50+ smart contracts, consensus protocols, VMs, etc.

Trusted

existing formal specifications

500k LOC

execution & verification

proof certificates (large!)

matching logic proof checker (240 LOC)

true

false

Trusted

my techniques

[CAV'21, OOPSLA'23]

proof-certifying smart contracts

+ more trustworthiness and transparency (specs + proof checker)

+ better scalability (off-chain computation + proofs; in a proof-carrying code style)

+ proof certificates can be stored off-chain or "STARK"-ed (discussed next).

Future Direction:
STARK Proof-Certifying Smart Contracts

$\textbf{ProofChecker}(\boldsymbol{\Gamma}^{\mathbf{EVM}}, \boldsymbol{\Psi}_{\mathbf{ERC20}}, \boldsymbol{\pi}_{\mathbf{ML}}) = \textbf{true}$ ✓

a fixed program

language spec

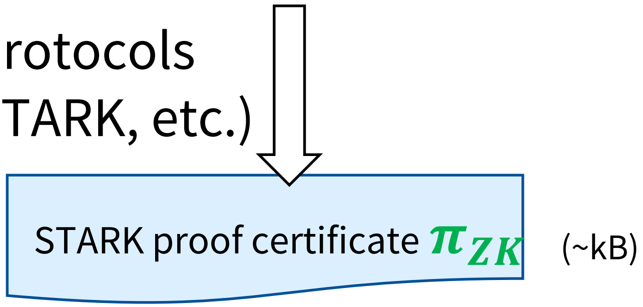
execution & verification

matching logic proof certificate (all “proof generation” techniques are here)

Future Direction:
STARK Proof-Certifying Smart Contracts

ProofChecker$(\Gamma^{\mathbf{EVM}}, \Psi_{\mathbf{ERC20}}, \pi_{\mathbf{ML}})$ = **true** ✓

STARK or other related protocols
(SNARK, ZK-STARK, ZK-STARK, etc.)



STARK proof certificate π_{ZK} (~kB)

Currently working with RiscZero Inc. to implement the checker in the RISC Zero zkVM.

$\exists \pi_{\mathbf{ML}}$. **ProofChecker**$(\Gamma^{\mathbf{EVM}}, \Psi_{\mathbf{ERC20}}, \pi_{\mathbf{ML}})$ = **true** ✓

STARK proof-certifying smart contracts

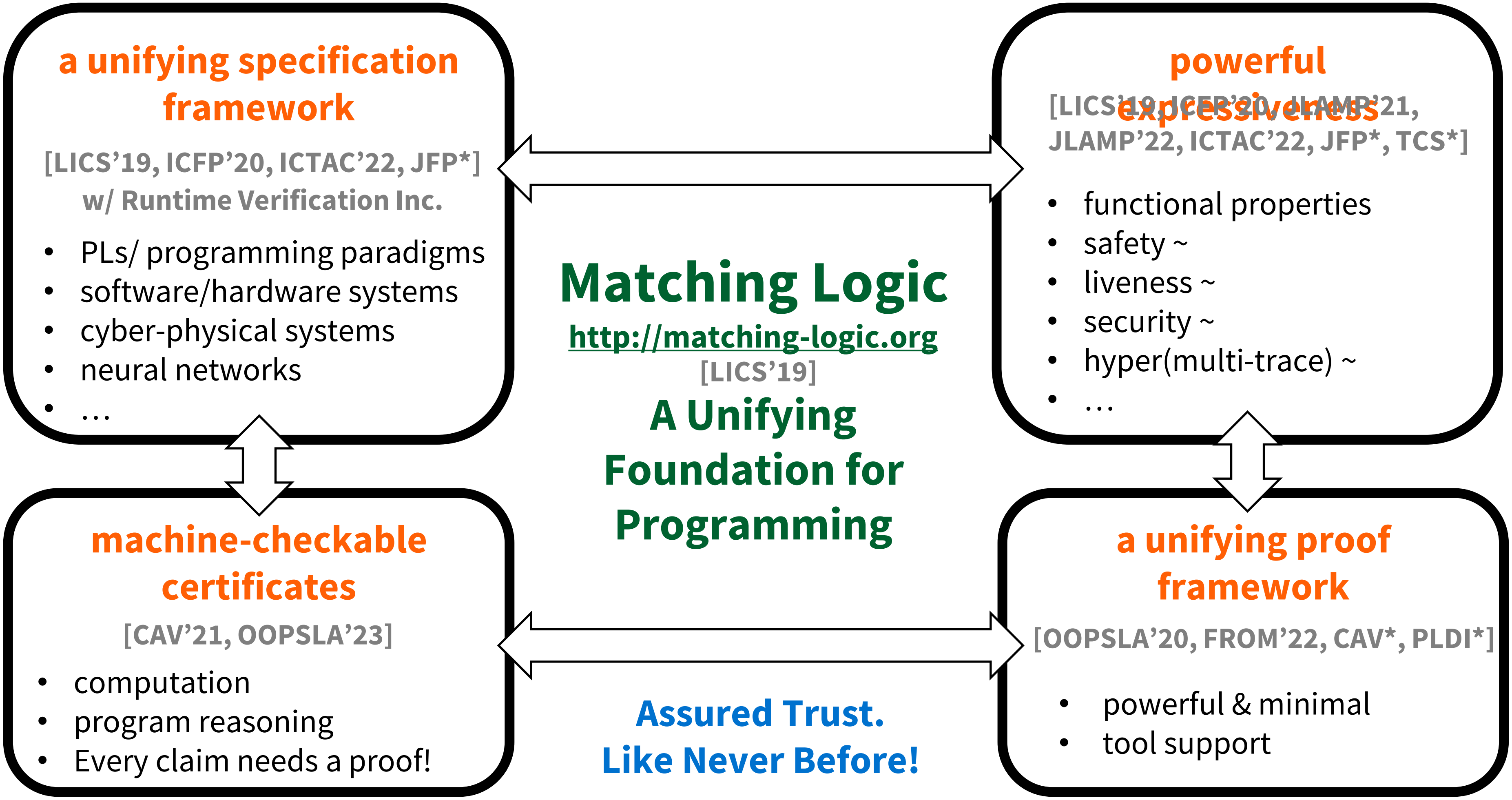
+ much smaller proof certificates; stored on-chain

+ producing STARK certificates on-the-fly; never pay the cost

+ great compatibility (dealing with one fixed checker; works for all PLs/platforms)

+ **Computation = Proof**; built-in checker ensures valid computations and correctness claims on the blockchain

Conclusion



a unifying specification framework

[LICS'19, ICFP'20, ICTAC'22, JFP*]
w/ Runtime Verification Inc.

- PLs/ programming paradigms
- software/hardware systems
- cyber-physical systems
- neural networks
- …

powerful expressiveness

[LICS'19, ICFP'20, JLAMP'21, JLAMP'22, ICTAC'22, JFP*, TCS*]

- functional properties
- safety ~
- liveness ~
- security ~
- hyper(multi-trace) ~
- …

Matching Logic

http://matching-logic.org

[LICS'19]

A Unifying Foundation for Programming

machine-checkable certificates

[CAV'21, OOPSLA'23]

- computation
- program reasoning
- Every claim needs a proof!

a unifying proof framework

[OOPSLA'20, FROM'22, CAV*, PLDI*]

- powerful & minimal
- tool support

Assured Trust.
Like Never Before!

Matching Logic Impact

runtime verification

K framework

NASA

BOEING

DENSO
Crafting the Core

TOYOTA
InfoTechnology
Center Co., Ltd.

ethereum

Tezos

INPUT | OUTPUT

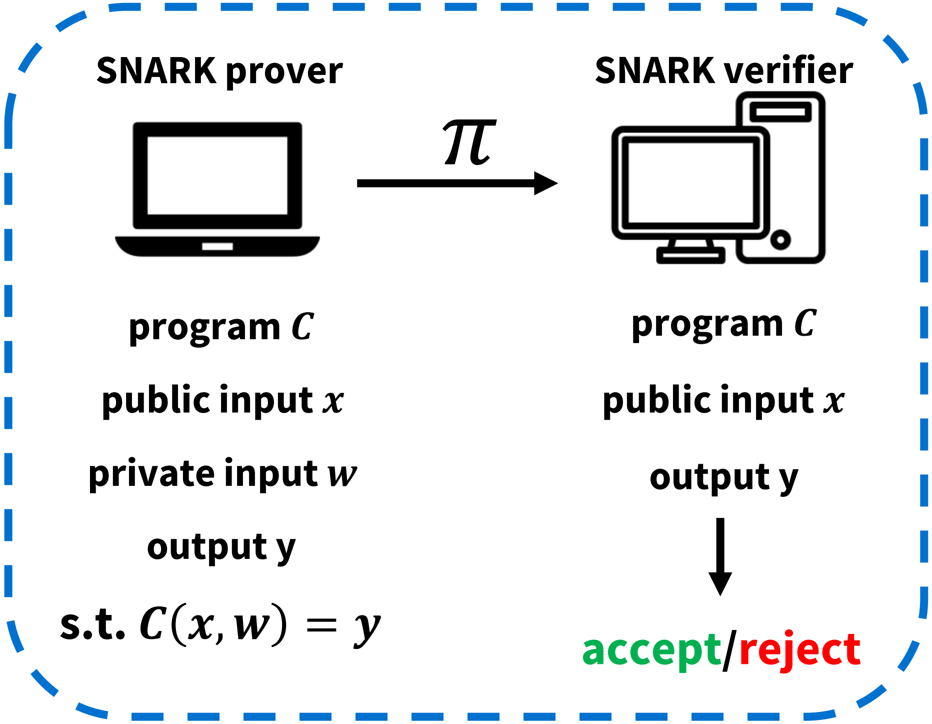
Algorand™

Graduate College Dissertation Completion Fellowship
“Matching Logic: Unifying Foundation of Programming”

Ethereum Foundation Funding
“Trustworthy Formal Verification for Ethereum Smart Contracts via Machine-Checkable Proof Certificates”

Future Direction: Generating SNARK-Proofs

SNARK = Succinct Non-Interactive ARgument of Knowledge*



SNARK prover $\xrightarrow{\pi}$ **SNARK verifier**

program C

public input x

private input w

output y

s.t. $C(x,w) = y$

program C

public input x

output y

↓

accept/reject

Knowledge Soundness:

$\Pr(\exists w.\, C(x,w) = y \mid \textbf{accept}) > 1 - \epsilon$

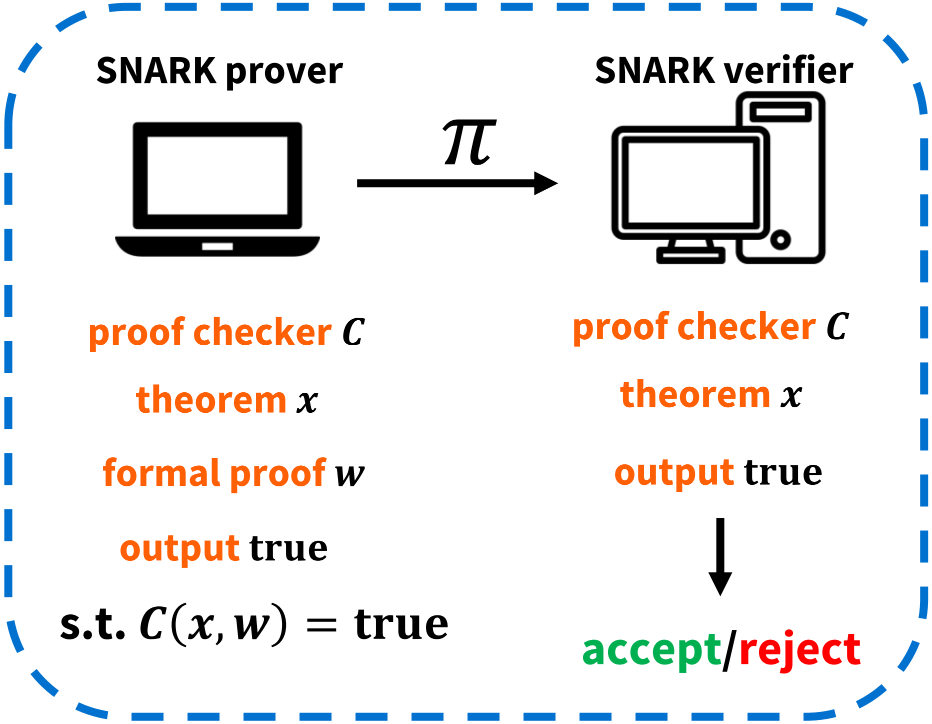
SNARK Proof Sizes:

small (~kB), even constant

Applications:

smart contracts (zcash, …)

Future Direction: SNARK-Proof Generation



Knowledge Soundness:

$$\Pr(\exists w. C(x, w) = \text{true} \mid \textbf{accept}) > 1 - \epsilon$$

“there exists a formal proof of theorem x”

Matching Logic Proof Checker + SNARK

- proof-carrying code

Vision

PLs = logical axioms (PL semantics)

- **C**
- **Java**
- **JavaScript**
- **Python**
- **Rust**

Matching Logic

A unifying foundation for programming

claims = formulas

program execution

`sum(100) = 5050`

formal verification

```
∀n.n ≥ 0
  → sum(n)= n(n+1)/2
```

...

$$\Gamma^{\mathrm{Lang}} \vdash \varphi_{\mathrm{Claim}}$$

machine-checkable correctness certificates for all PLs and all PL tools

Similar Pain Point in Mathematics

- many mathematical domains/fields
  - algebras, geometry, calculus, …
- many claims
  - theorems, lemmas, propositions, …
- mathematical proofs
  - written by humans, checked by humans
  - error-prone
- **Idea: formalizing mathematics & mechanizing proofs**
  - a unifying foundation for mathematics (e.g., set theory)
  - proofs on paper ⇒ mechanized, formal proofs ⇒ proof checker (small)
  - examples: metamath, unimath

Can We Do the Same for Programming?

Matching Logic 101: Semantics (slide 2/3)

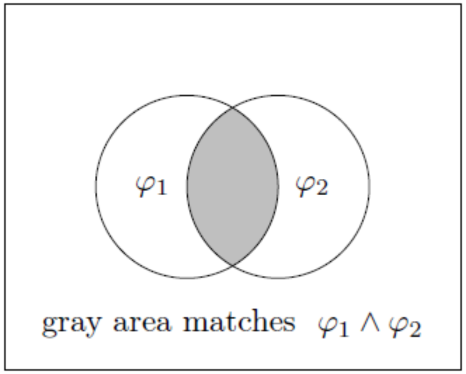
A matching logic ***model*** has

- a carrier set M
- an interpretation $\sigma_M : M \times \cdots \times M \to \mathcal{P}(M)$ for each symbol σ

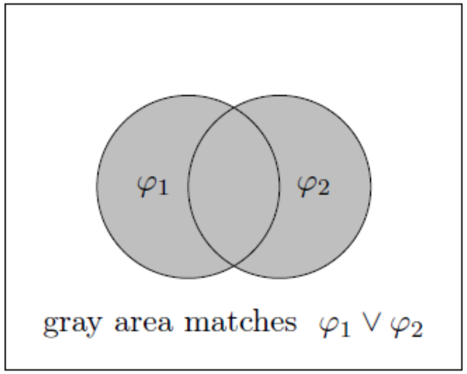
$$\boldsymbol{\varphi} \Longrightarrow |\boldsymbol{\varphi}|_{\boldsymbol{M},\boldsymbol{\rho}} \subseteq \boldsymbol{M}$$

pattern

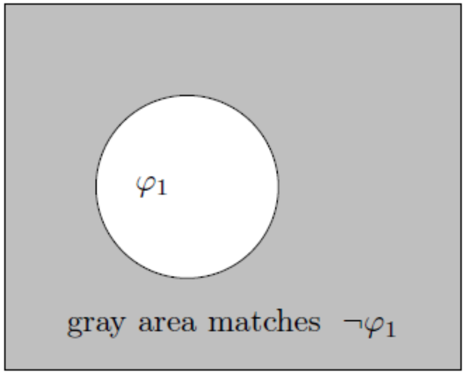
the set of elements that ***match*** φ



gray area matches $\varphi_1 \wedge \varphi_2$



gray area matches $\varphi_1 \vee \varphi_2$



gray area matches $\neg\varphi_1$

Similar Pain Point in Mathematics

- many mathematical domains/fields
  - algebras, geometry, calculus, …
- many claims
  - theorems, lemmas, propositions, …
- mathematical proofs
  - written by humans, checked by humans
  - error-prone
- **Idea: formalizing mathematics & mechanizing proofs**
  - a unifying foundation for mathematics (e.g., set theory)
  - proofs on paper ⇒ mechanized, formal proofs ⇒ proof checker (small)
  - examples: metamath, unimath

Can We Do the Same for Programming?

Matching Logic

http://matching-logic.org

- **We studied various logics, calculi, foundations, and semantics styles.**
  - First-order logic; Second/higher-order logic; Least fixpoint logic; Modal logics; Temporal logics (LTL, CTL, CTL*, …), λ-calculus; Type systems (parametric, dependent, inductive, …); μ-calculus; Hoare logics; Separation logics; Dynamic logics; Rewriting logic; Reachability logic; Equational logic; …
  - Small-/big-step SOS; Evaluation contexts; Abstract machines (CC, CK, CEK, SECD, …); Chemical abstract machine; Axiomatic; Continuations; Denotational; Initial; …
- **But each of the above had limitations.**
  - Some only handle certain aspects of computation (e.g., execution only).
  - Some are "design patterns"; re-design a new logic for a new PL/domain.
  - simplicity, modularity, notation
- **Matching logic: keep advantages and avoid limitations**

What is Matching Logic?

(http://matching-logic.org)

[…, TechRep’09, AMAST’10, UV’10, ICALP’12, OOPSLA’12, LMCS’17, LICS’19]

- one logic to specify and reason about any property of any program in any programming language
- one proof checker to automatically check proofs
- correctness of PL tools ⇒ proof checking
- absolute correctness guarantee: No Compromise!

+ v

+ v

+ p

- embedded in the K framework (https://kframework.org)
- proof-certifying interpreter [CAV’21] and formal verifier [OOPSLA’23]

PhD Research

Matching Logic [LICS'19]

- syntax → **rules for writing formulas**
- semantics → **models;**
  giving meaning to formulas
- proof system → **proof rules;**
  proving formulas/theorems

Expressive Power

[LICS'19, OOPSLA'20, ICFP'20, JLAMP'21, JLAMP'22, JFP*]

- defining various program properties
- defining various PL semantics methods

Principles of Formal Reasoning

- automated theorem proving
  [OOPSLA'20, PLDI*]
- interactive theorem proving [FROM'22]
- completeness [LICS'19 → **truth ⇒ proofs?**
- decidability] [LICS*] → **decision procedures**

Proof-Certifying PL Tools

- proof-certifying program execution
  [CAV'21]
- proof-certifying formal verification
  [OOPSLA'23]

In this Talk

Matching Logic [LICS’19]

- syntax
- semantics

- proof system

Expressive Power (Summary)

[LICS’19, OOPSLA’20, ICFP’20, JLAMP’21, JLAMP’22]

- defining existing logics & calculi

- supporting existing formal PL semantics

Principles of Formal Reasoning

- automated theorem proving [OOPSLA’20, PLDI*]
- interactive theorem proving [FROM’22]
- completeness [LICS’19
- decidability] [LICS*]

Proof-Certifying PL Tools

- proof-certifying program execution [CAV’21]
- proof-certifying formal verification [OOPSLA’23]

Matching Logic Proof Checker

- We use **metamath**
  - http://metamath.org
  - encode matching logic proofs
  - mechanize proof checking (very fast; million steps per sec)
- **small trust base to check proofs!**

$$\underbrace{\Gamma}_{\text{axioms}} \vdash \underbrace{\varphi}_{\text{theorem}}$$

Matching Logic

http://matching-logic.org

A Unifying Foundation for Programming

Reducing the correctness of **any computation** & **program reasoning** to **proof checking**.

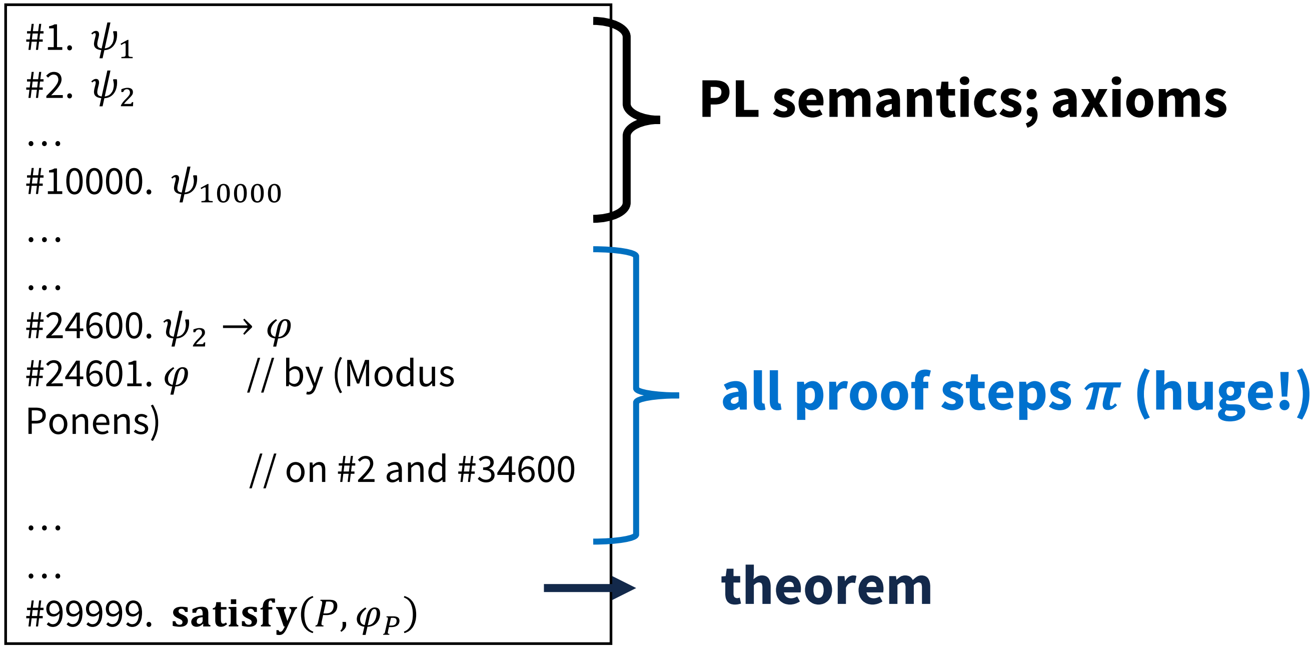
What is Matching Logic?

A logic with first-order variables and quantifiers, polyadic modalities and function symbols, fixpoint operations, and top-level second-order universal quantifiers. [**LICS'19**]

+ **very simple**
  - the matching logic proof checker has only 240 lines (Coq has 8000 lines)
+ **very expressive** [**LICS'19**, **OOPSLA'20**, **ICFP'20**, **JLAMP'21**, **JLAMP'22**, **CAV'21**, **OOPSLA'23**]
  - **Program Properties**: functional, safety, liveness, security, hyper, …
  - **PL Semantics Methods**: operational, Hoare logics, denotational, continuations, initial algebras, evaluation contexts, rewriting, abstract machines, …
+ **practical** [with Runtime Verification Inc.]
  - embedded in the **K framework** (https://kframework.org)
  - proof-certifying interpreter [**CAV'21**] and formal verifier [**OOPSLA'23**]

Future Direction 2: Matching Logic Proof Checker + SNARK

Currently, proof certificates are huge (1 MB per exec. step)



#1. ψ_1
#2. ψ_2
…
#10000. ψ_{10000}
…
…
#24600. $\psi_2 \rightarrow \varphi$
#24601. φ // by (Modus Ponens)
// on #2 and #34600
…
…
#99999. $\mathbf{satisfy}(P, \varphi_P)$

PL semantics; axioms

all proof steps π (huge!)

theorem

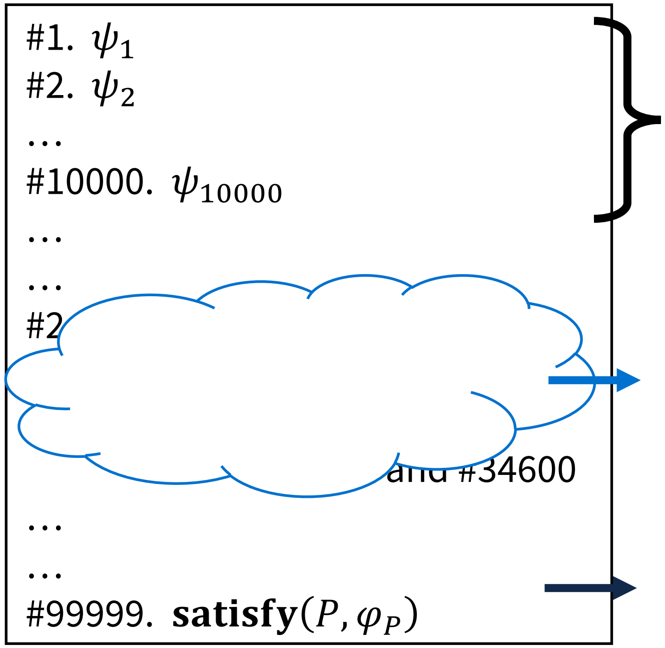
Future Direction 2: More Succinct Proof Certificates

Use succinct (zero-knowledge/ZK) cryptographic proofs

proof certificates
(several MB; with π)

cryptographic certificates
(256 bits; without π)

RISC Zero



#1. ψ_1
#2. ψ_2
…
#10000. ψ_{10000}
…
…
#2
and #34600
…
…
#99999. $\textbf{satisfy}(P, \varphi_P)$

PL semantics; axioms

there exist proof steps π

theorem

K Framework and Matching Logic

https://kframework.org/ http://matching-logic.org/

Matching logic formulas, called *patterns*:

$$\varphi ::= \underbrace{\boxed{x \mid \sigma(\varphi_1, \ldots, \varphi_n)}}_{\textbf{structures}} \mid \underbrace{\boxed{\varphi_1 \land \varphi_2 \mid \neg\varphi}}_{\textbf{logical constraints}} \mid \underbrace{\boxed{\exists x. \varphi}}_{\substack{\textbf{first-order} \\ \textbf{quantification}}}$$

But, matching logic has a serious limitation.

Matching Logic Lacks Support for Fixpoints

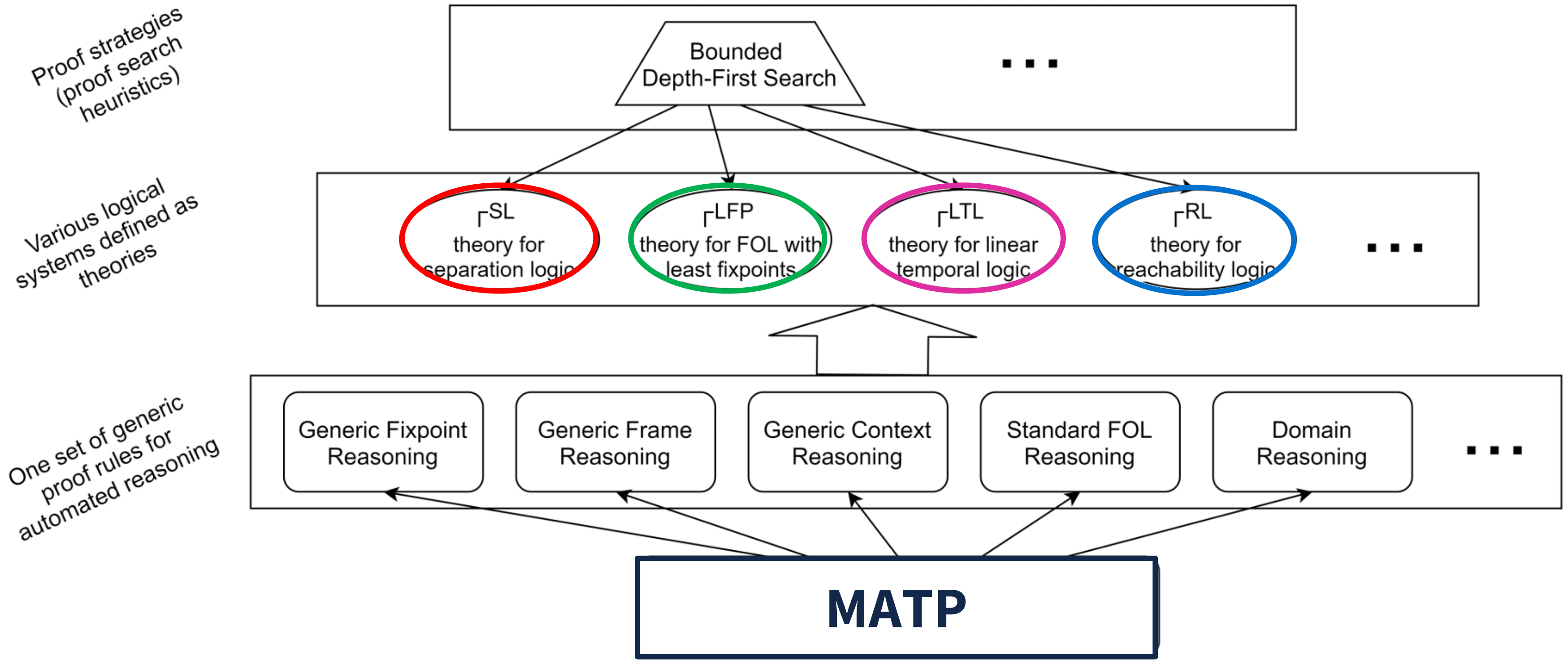
Fixpoints are ubiquitous in computer science.

- inductive datatypes
- induction principles
- recursive functions and loops in PLs
- formal verification
- …

Matching logic itself is insufficient.

- Outsource to other tools (e.g., Coq).
- Extend it for specific purposes (e.g., formal verification).

Automatic Theorem Prover for Matching μ-Logic: Architecture


Proof strategies (proof search heuristics)
Bounded Depth-First Search
. . .
Various logical systems defined as theories
Γ^{SL} theory for separation logic
Γ^{LFP} theory for FOL with least fixpoints
Γ^{LTL} theory for linear temporal logic
Γ^{RL} theory for reachability logic
. . .
One set of generic proof rules for automated reasoning
Generic Fixpoint Reasoning
Generic Frame Reasoning
Generic Context Reasoning
Standard FOL Reasoning
Domain Reasoning
. . .
MATP

- Separation logic: Proved 265/280 benchmark tests in SL-COMP’19
- LTL: Proved all the axioms in the complete LTL proof system
- LTP & RL: Proved the correctness of the SUM program

Where do Proof Objects Come From?

Q1: Is there always a proof object for a true statement?

- Completeness of matching μ-logic (briefly)

Q2: Can we find proof objects automatically?

- Automatic theorem prover for matching μ-logic (briefly)

Q3: Can we generate proof objects from K?

- Proving the correctness of K in the translation validation style.

Where do Proof Objects Come From?

Q1: Is there always a proof object for a true statement?

- Completeness of matching μ-logic (briefly)

Q2: Can we find proof objects automatically?

- Automatic theorem prover for matching μ-logic (briefly)

Q3: Can we generate proof objects from K?

- Proving the correctness of K in the translation validation style.

Where do Proof Objects Come From?

Q1: Is there always a proof object for a true statement?

- Completeness of matching μ-logic (briefly)

Q2: Can we find proof objects automatically?

- Automatic theorem prover for matching μ-logic (briefly)

Q3: Can we generate proof objects from K?

- Proving the correctness of K in the translation validation style.

Translating K to Matching μ-Logic

K	**Matching μ-Logic**
A PL definition **Ethereum VM**	A *logical theory* Γ^{EVM}
Any PL task • program execution **Interpreter** • formal verification **Program Verifier**	A *theorem* proved by the 15-rule *proof system* • $\Gamma^{\mathrm{EVM}} \vdash t_{\mathrm{init}} \Rightarrow_{\mathrm{exec}} t_{\mathrm{final}}$ • $\Gamma^{\mathrm{EVM}} \vdash \varphi_{\mathrm{pre}} \rightsquigarrow \varphi_{\mathrm{post}}$
Correctness of the task	Generating the proof and checking it using the 200-LOC *proof checker*

- **Task 1**: Generating the logical theory (e.g., Γ^{EVM})
- **Task 2**: Generating the proof for a given PL task (e.g., verifying a program)