A Foundationally Verified Intermediate Verification Language

Joshua M. Cohen
Princeton University

Final Public Oral

May 2, 2025

How to verify functional correctness?

```
int minimum (int a[ ], int n) {
    int min = a[0];
    for(int i = 0; i < n; i++) {
        int j = a[i];
        if (j < min) min = j;
    }
    return min;
}
```

Dafny

```
method min (a: array<int>) returns (m: int)
requires a.Length > 0
ensures forall i : int :: 0 <= i < a.Length ==> m <= a[i]
ensures m in a[..]
{
    m := a[0];
    var i := 1;
    while (i < a.Length)
    invariant 1 <= i <= a.Length
    invariant forall j : int :: 0 <= j < i ==> m <= a[j]
    invariant m in a[0..i]
    {
        if (a[i] < m) {
            m := a[i];
        }
        i := i + 1;
    }
    return m;
}
```

Semi-Automated Verifiers

```
method min (a: array<int>) returns (m: int)
requires a.Length > 0
ensures forall i : int :: 0 <= i < a.Length ==> m <= a[i]
{
    m := a[0];
    var i := 1;
    while (i < a.Length)
    invariant 1 <= i <= a.Length
    invariant forall j : int :: 0 <= j < i ==> m <= a[j]
    {
        if (a[i] < m) {
            m := a[i];
        }
        i := i + 1;
    }
    return m;
}
```

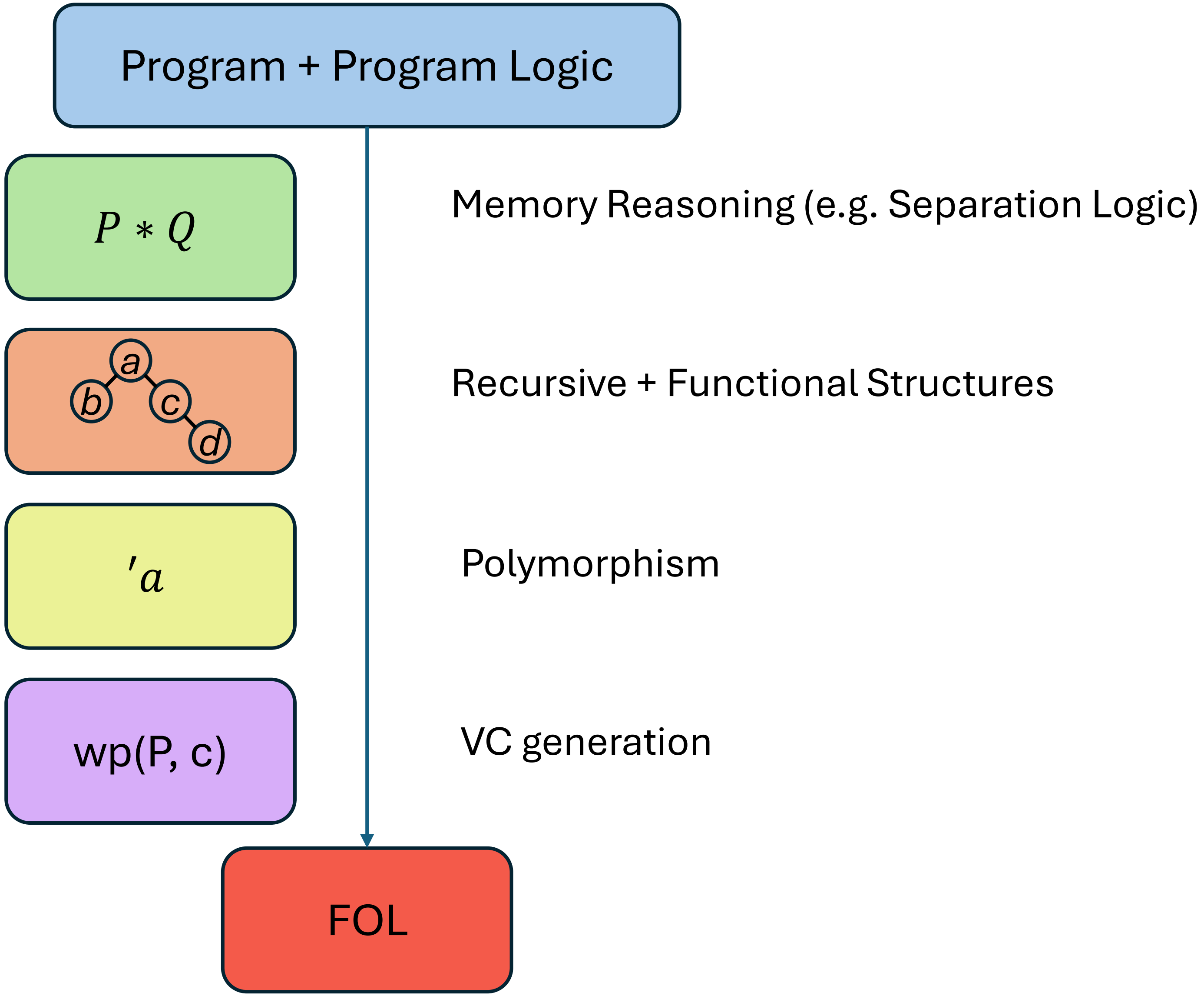
Verification Condition (VC) generator

$$\big((\mathrm{a[i]} < m) \rightarrow m' = a[i]\big) \wedge (\neg\,(a[i] < m) \rightarrow m' = m) \wedge \ldots$$

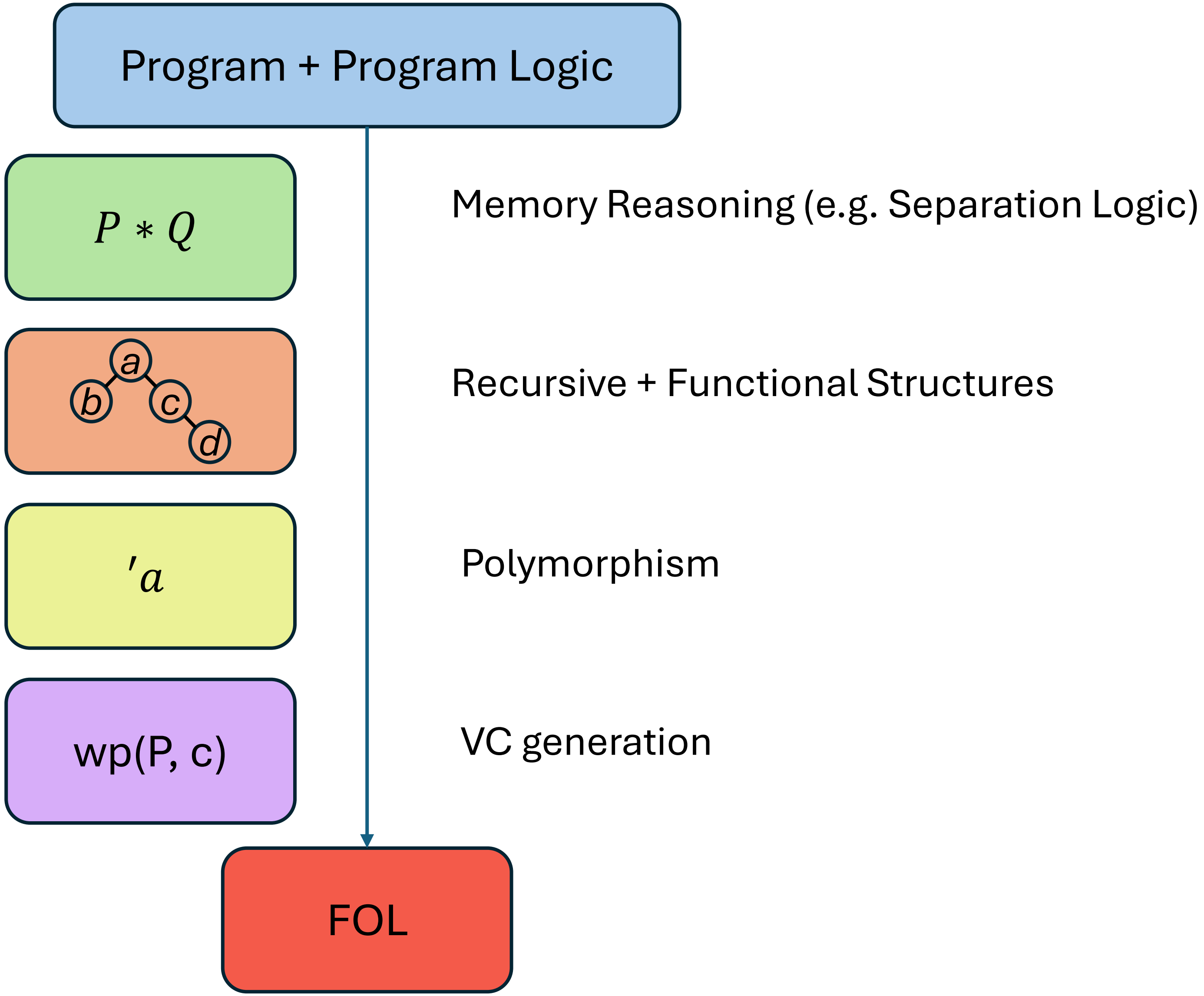
(SMT) Solver

Valid?

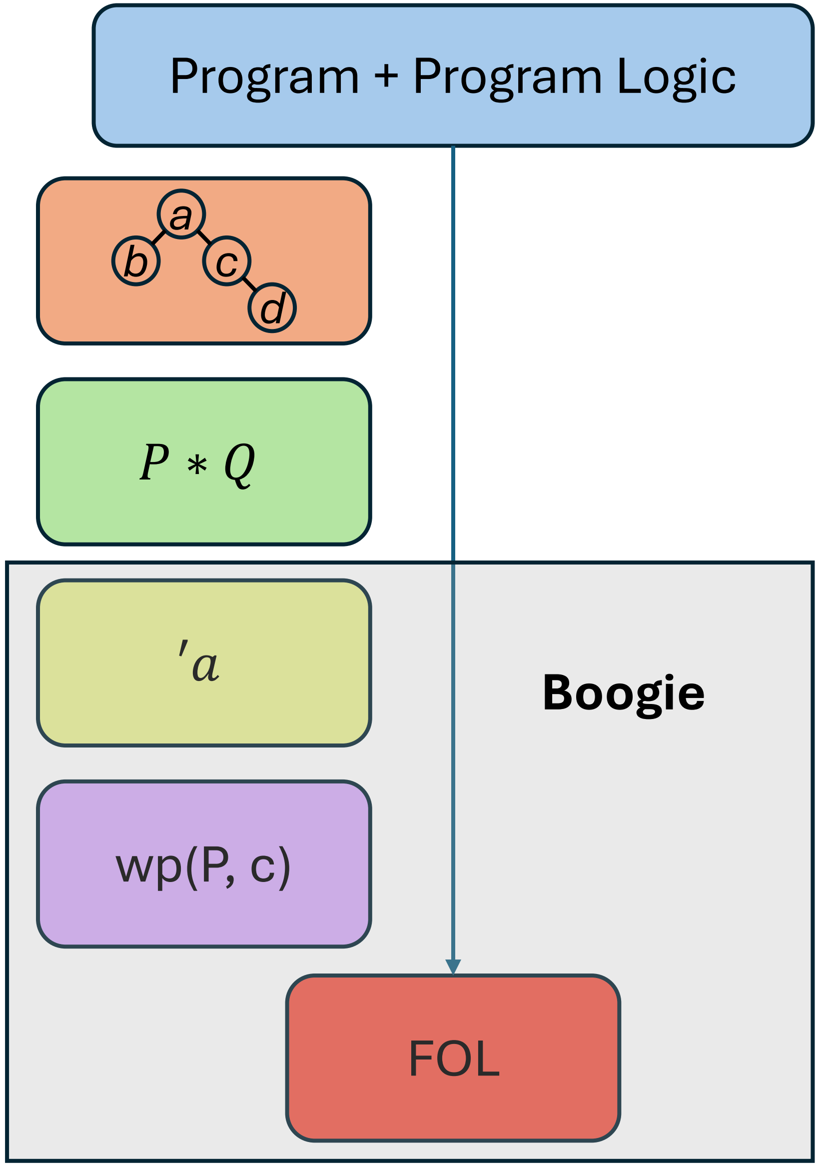
From Programs to FOL



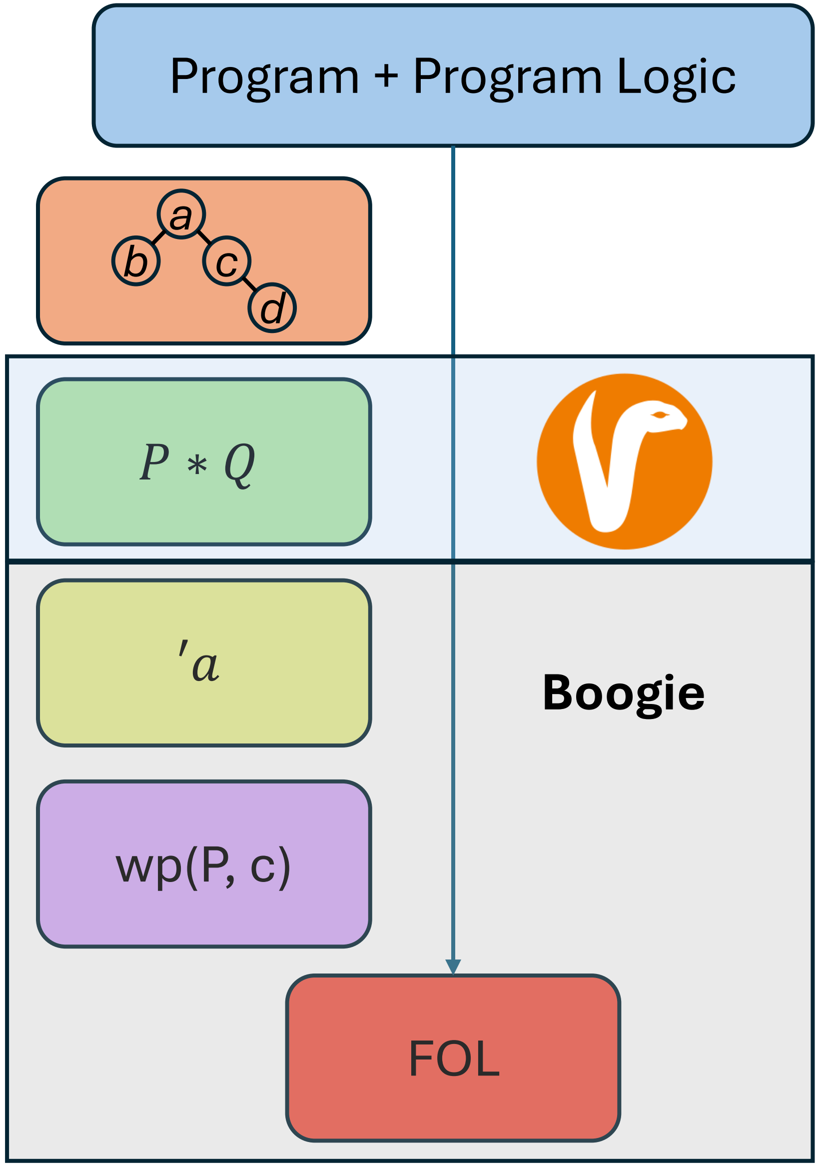
From Programs to FOL



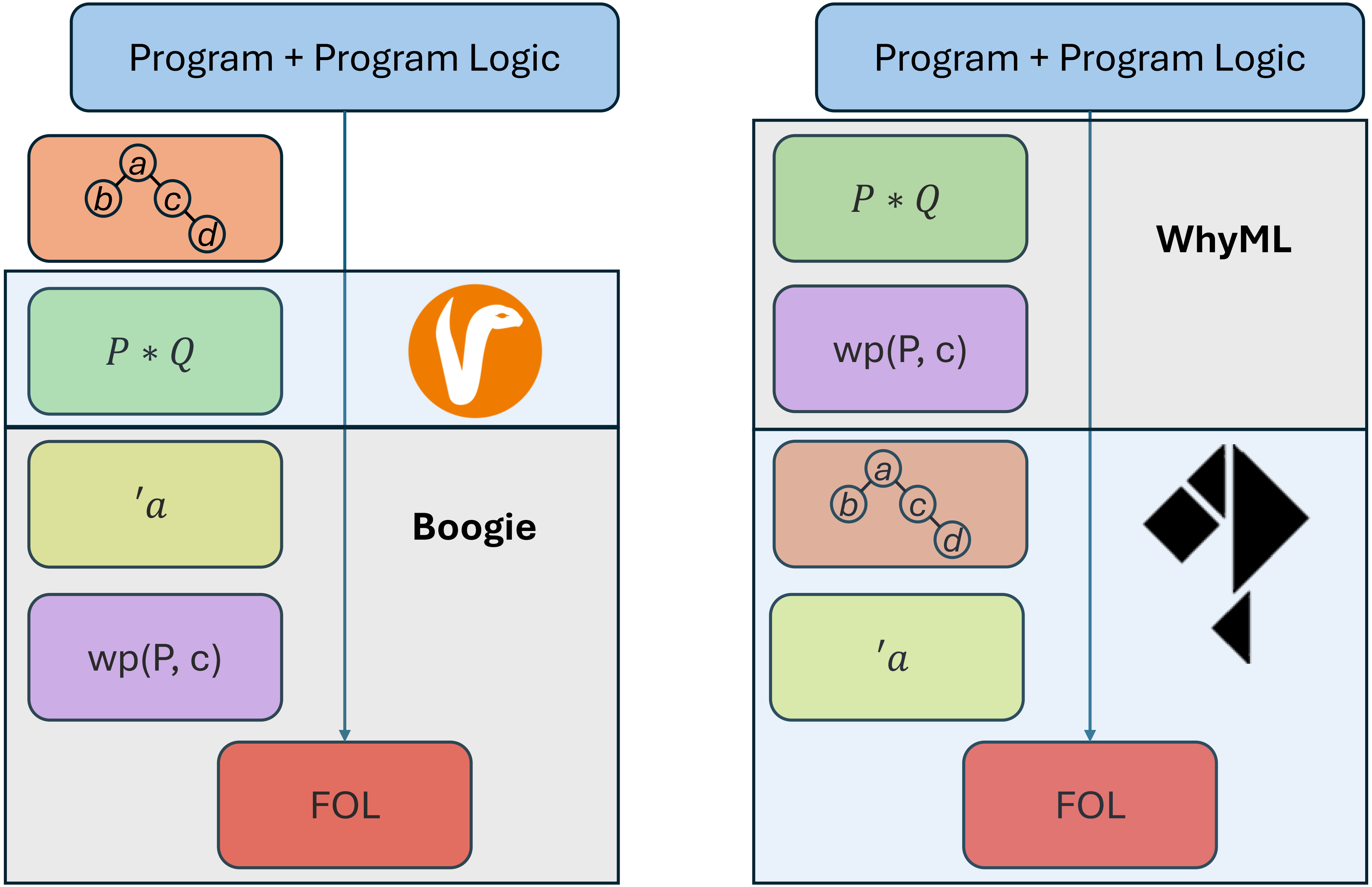
Intermediate Verification Languages (IVLs)



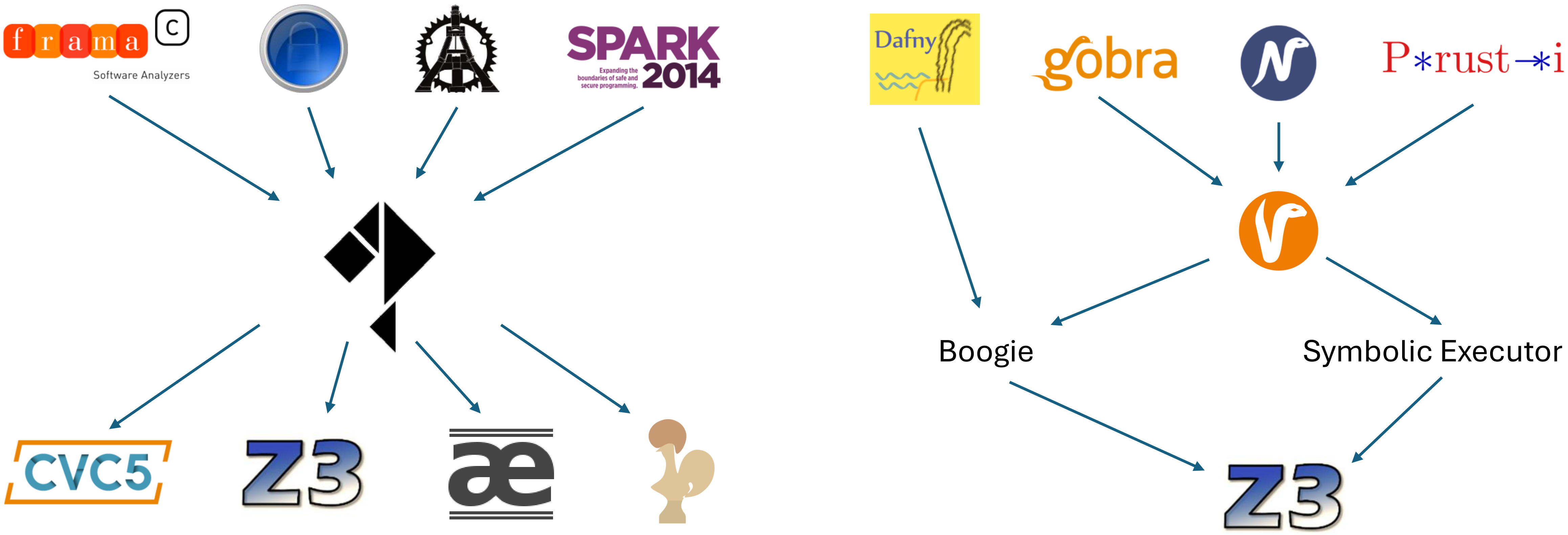
Intermediate Verification Languages (IVLs)



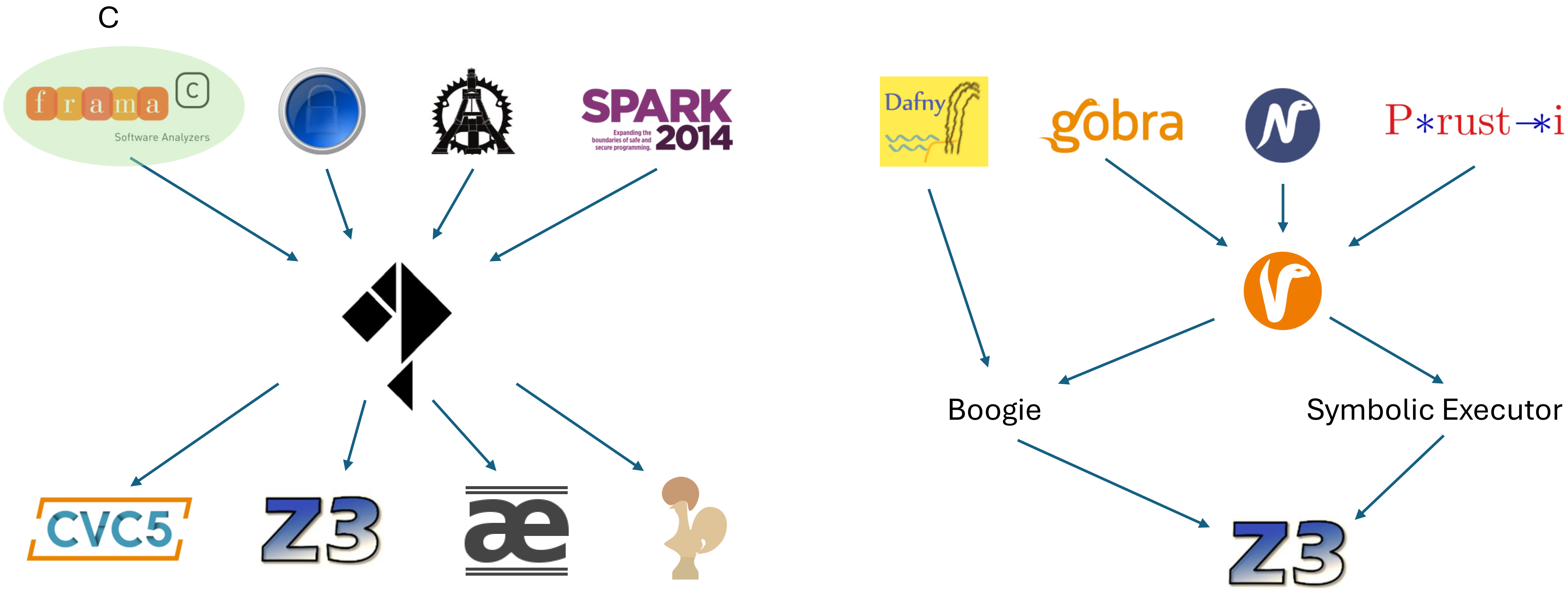
Intermediate Verification Languages (IVLs)



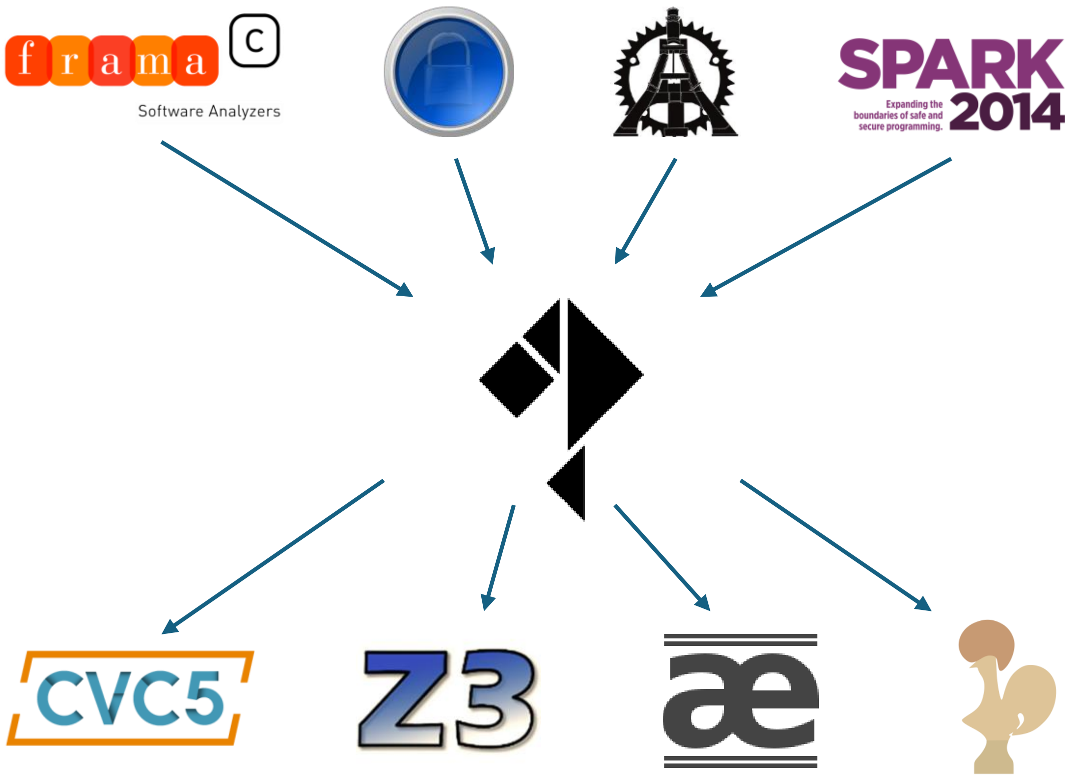
Semi-Automated Verifiers

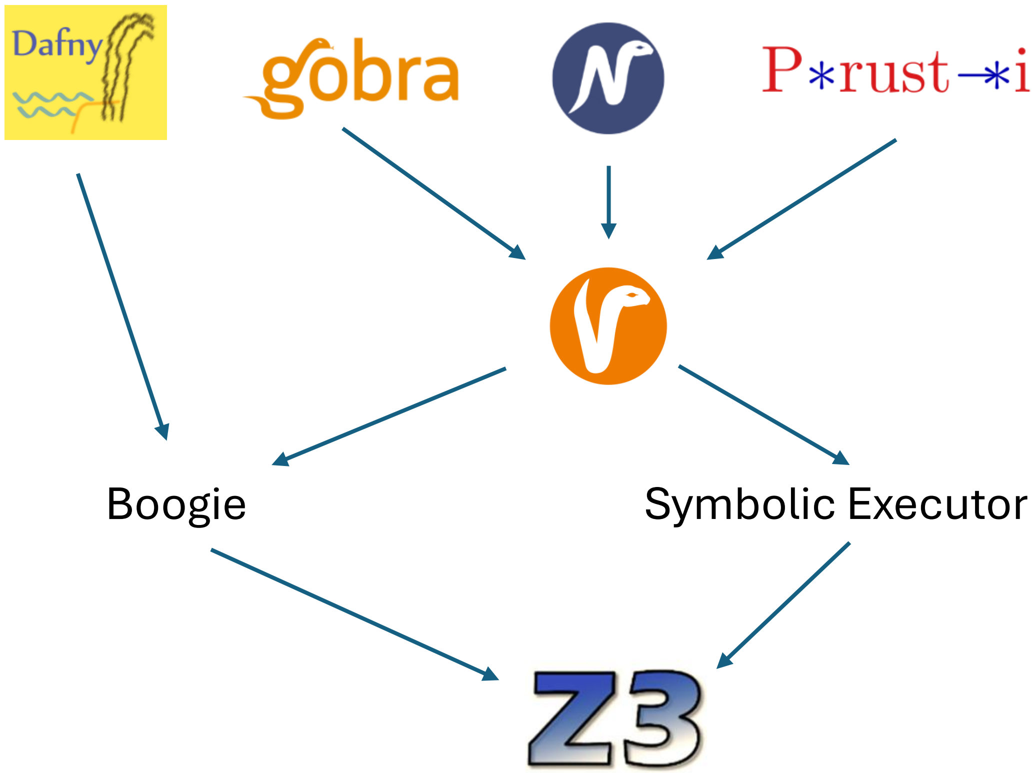


Semi-Automated Verifiers

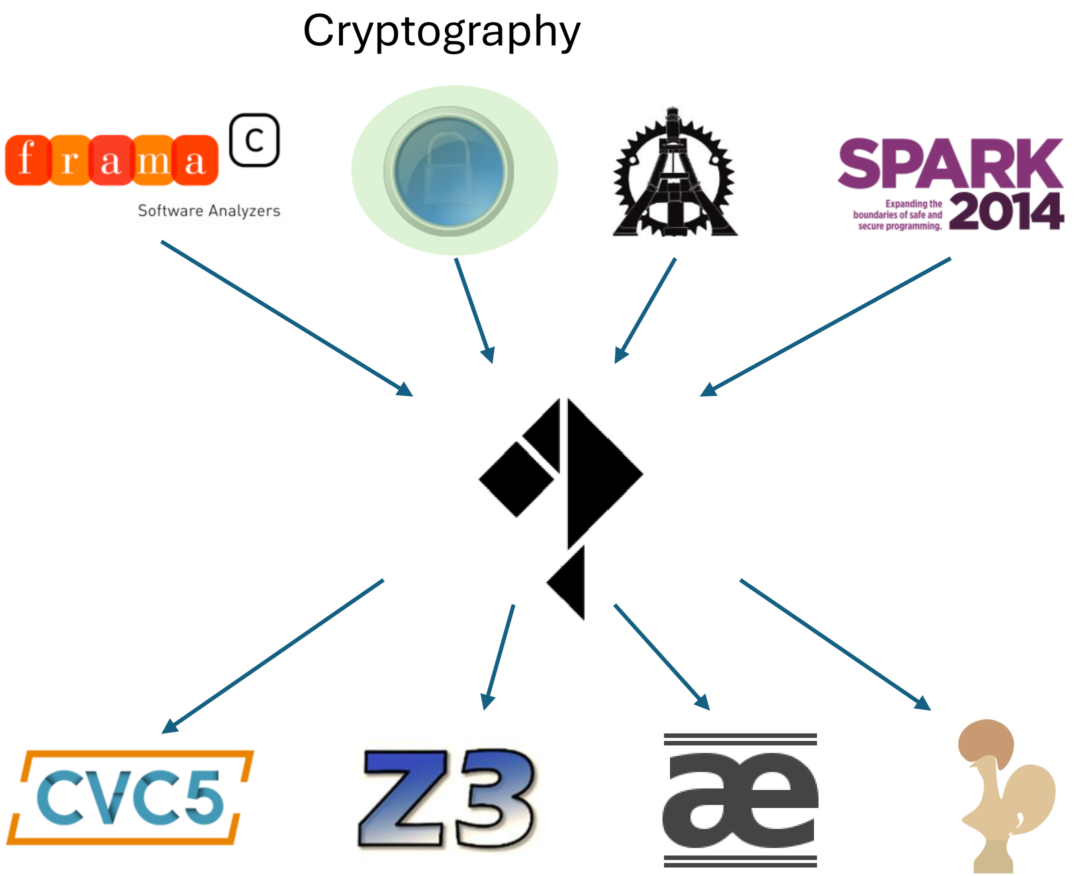


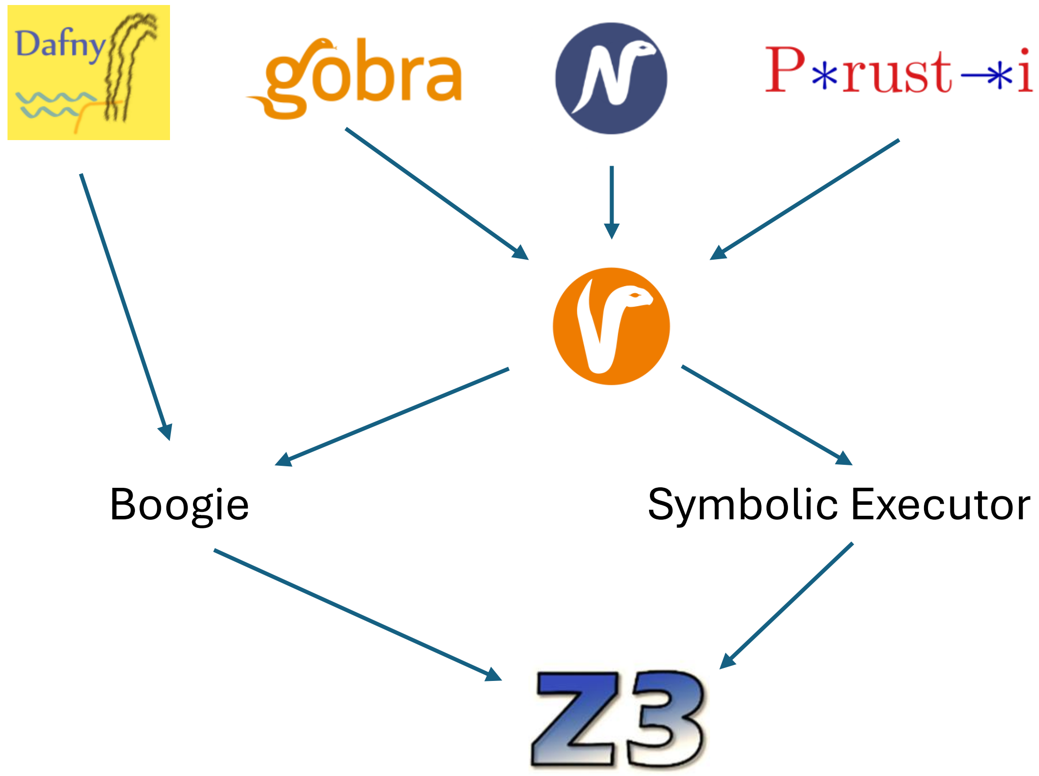
Semi-Automated Verifiers



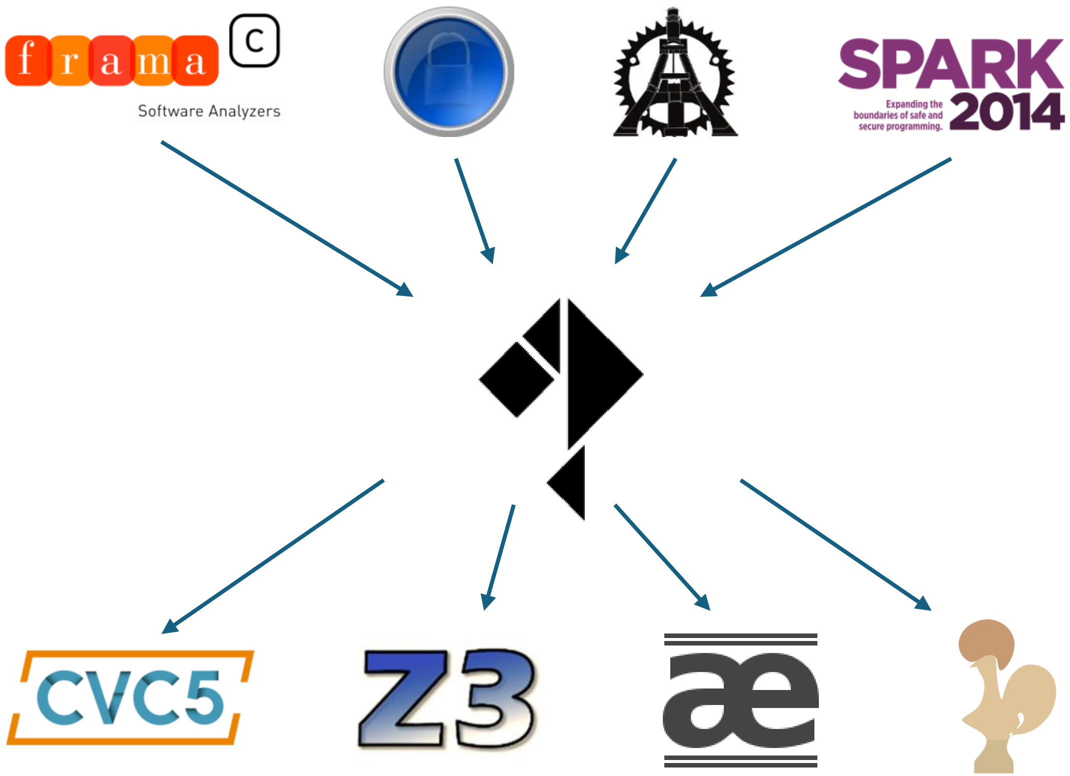


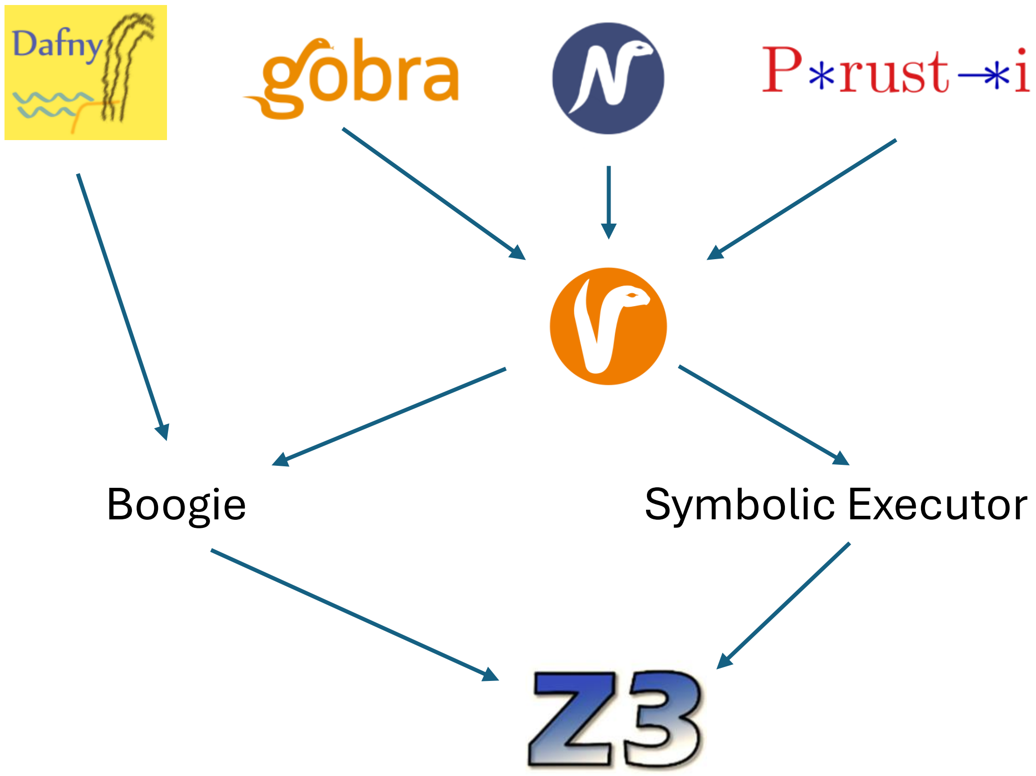
Semi-Automated Verifiers



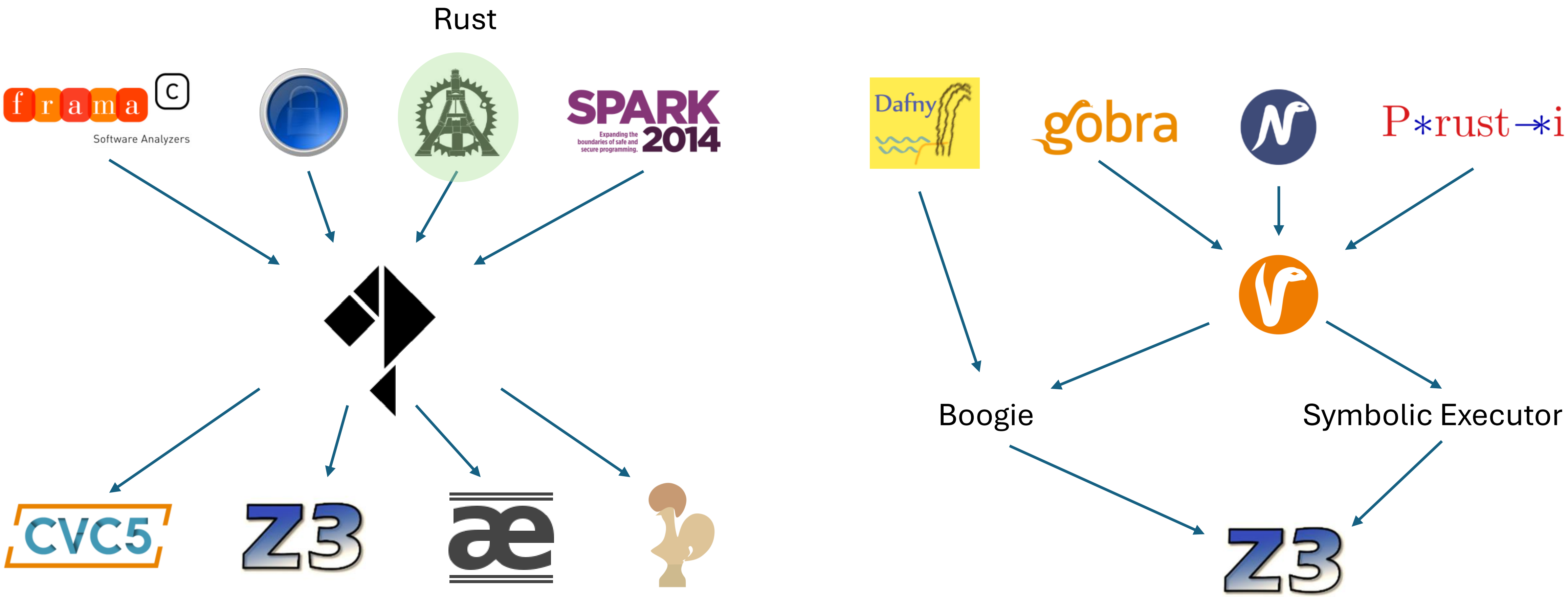


Semi-Automated Verifiers

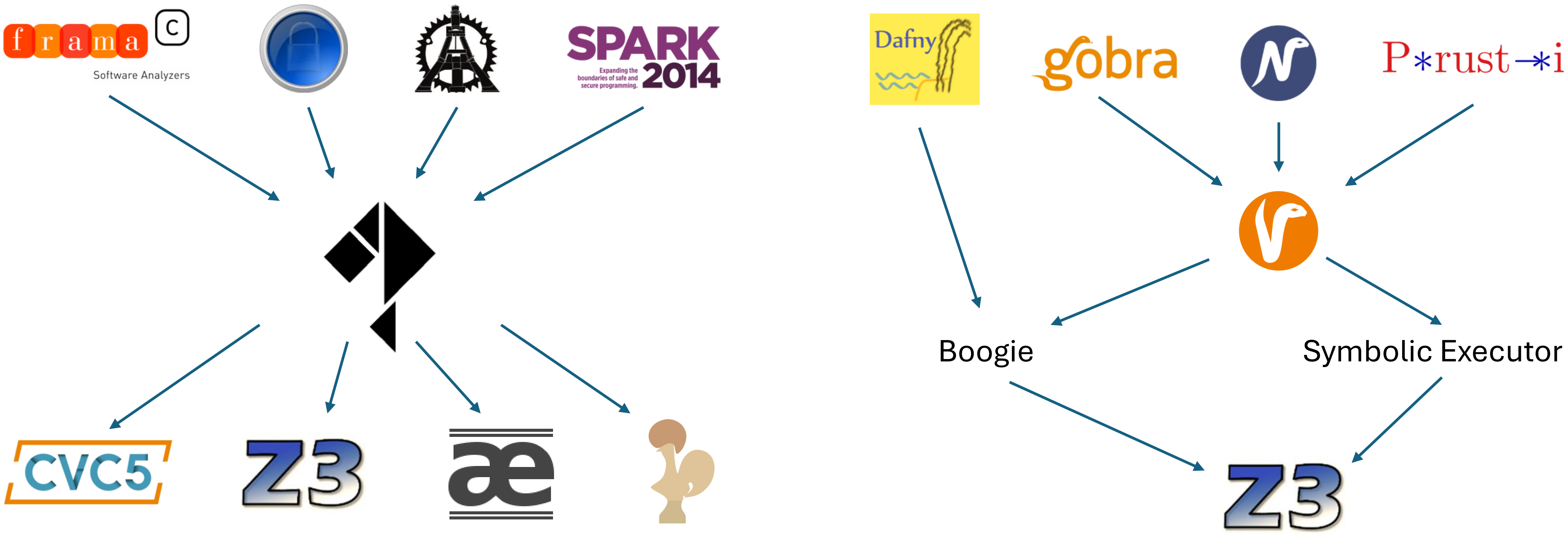




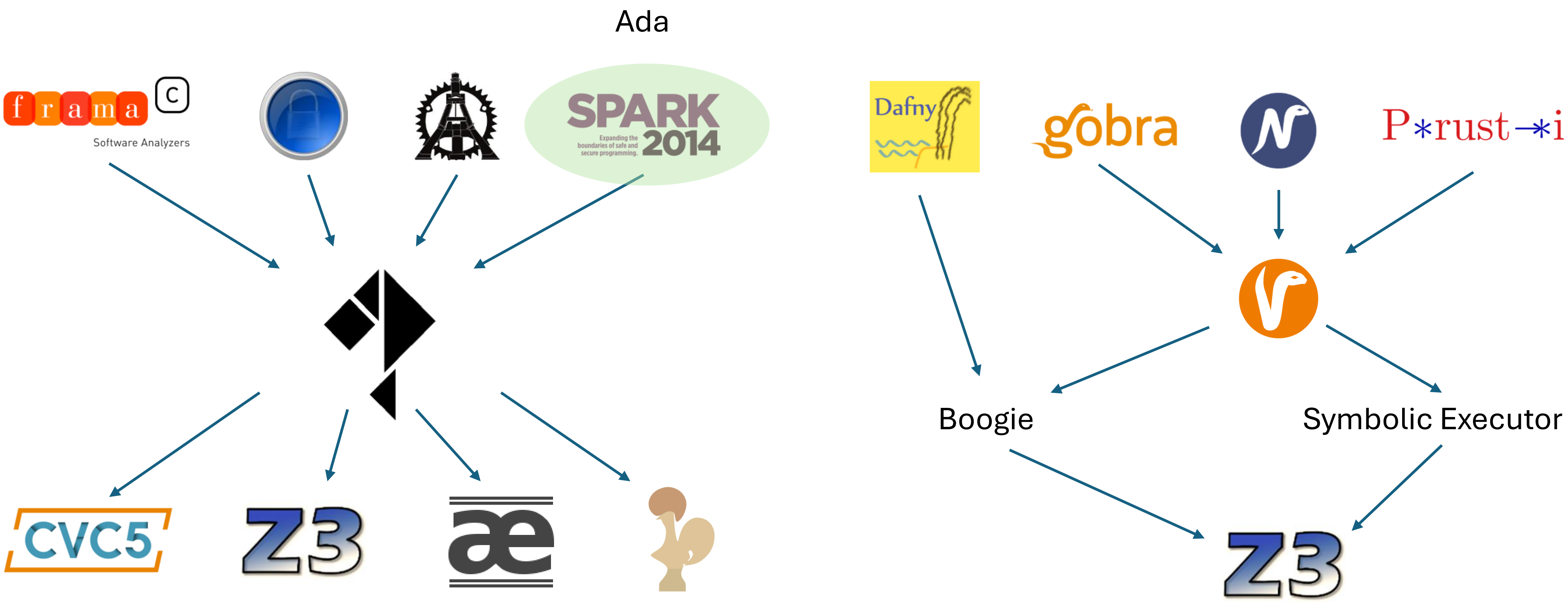
Semi-Automated Verifiers



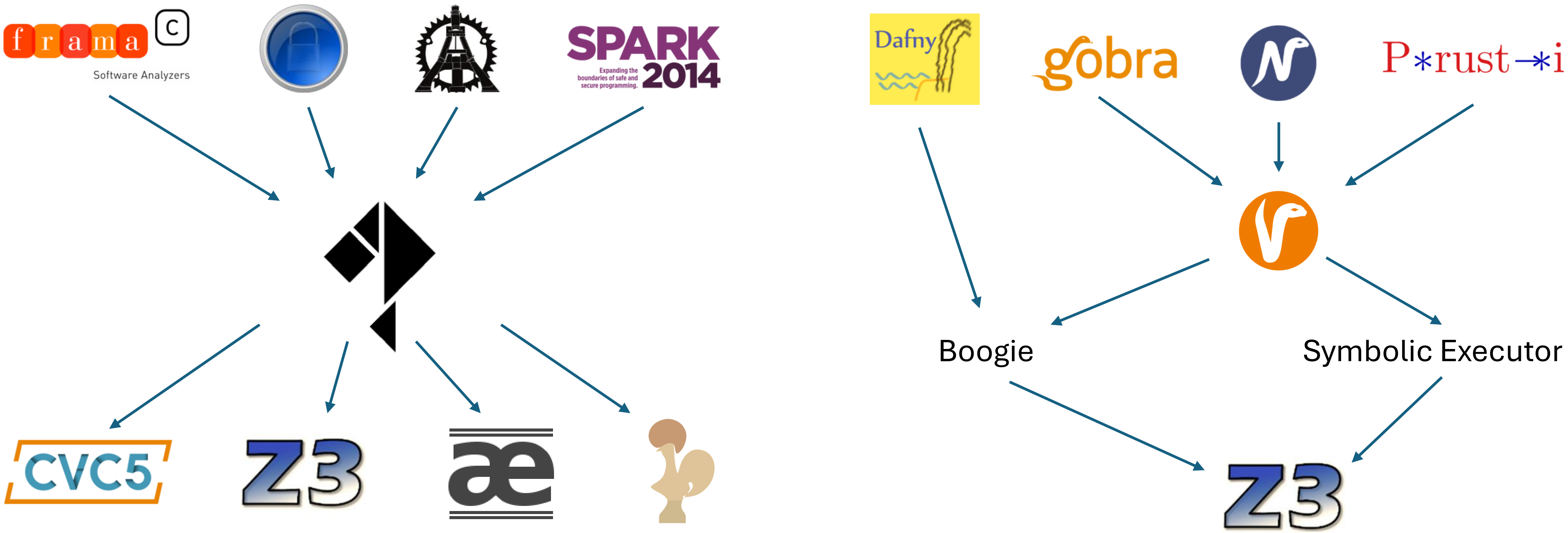
Semi-Automated Verifiers



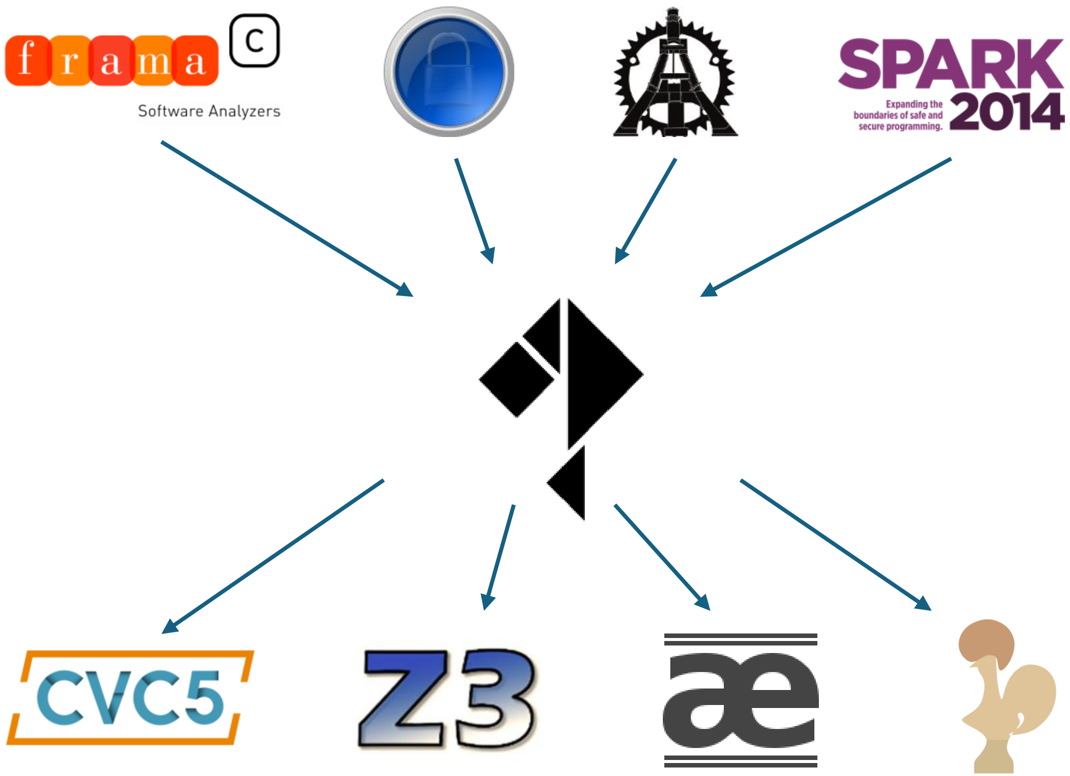
Semi-Automated Verifiers

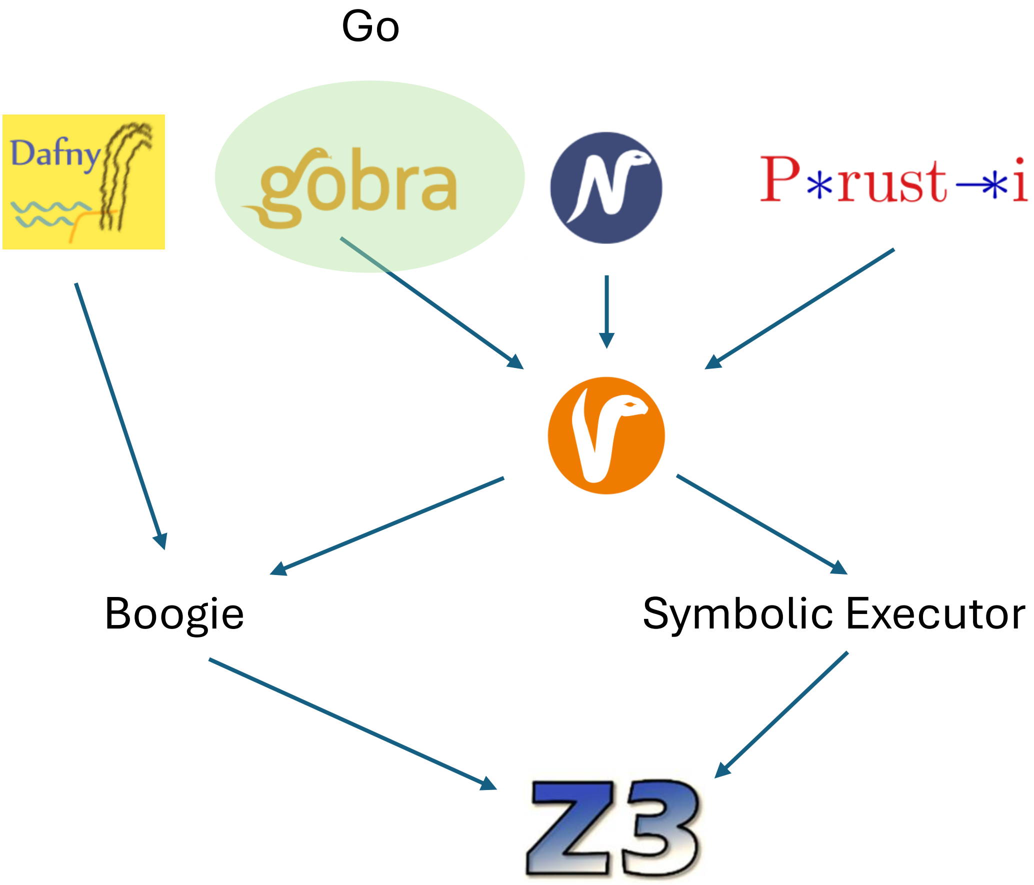


Semi-Automated Verifiers

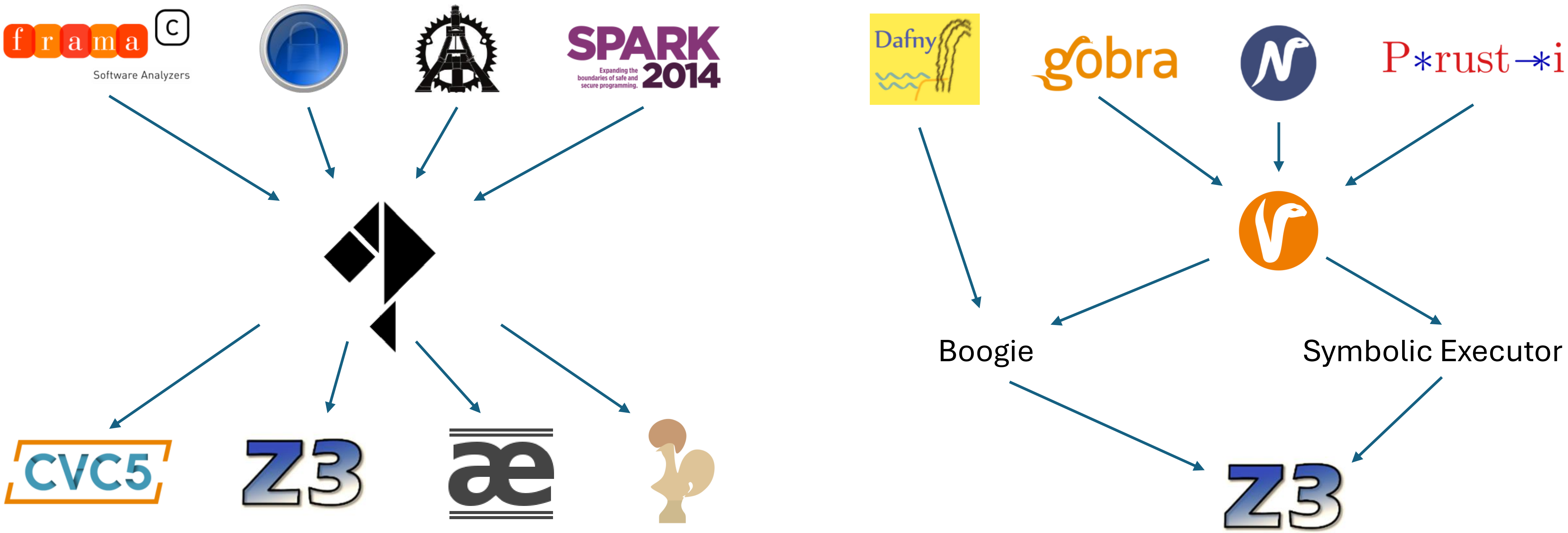


Semi-Automated Verifiers

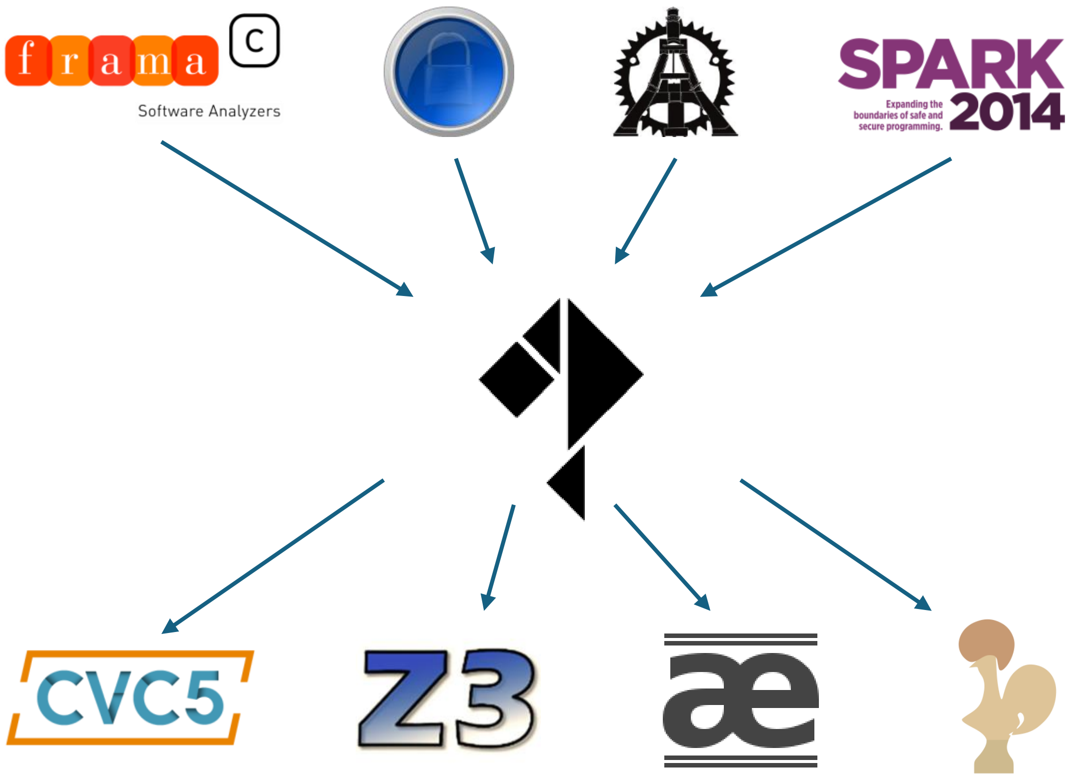


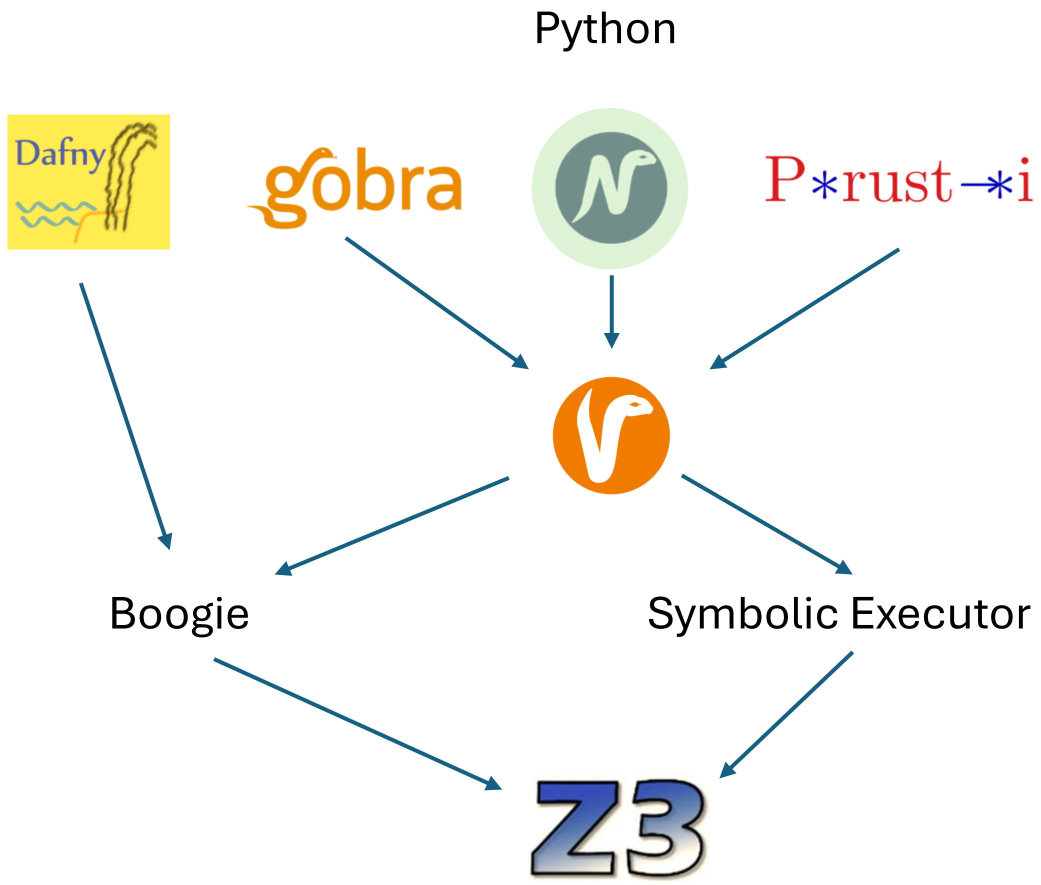


Semi-Automated Verifiers

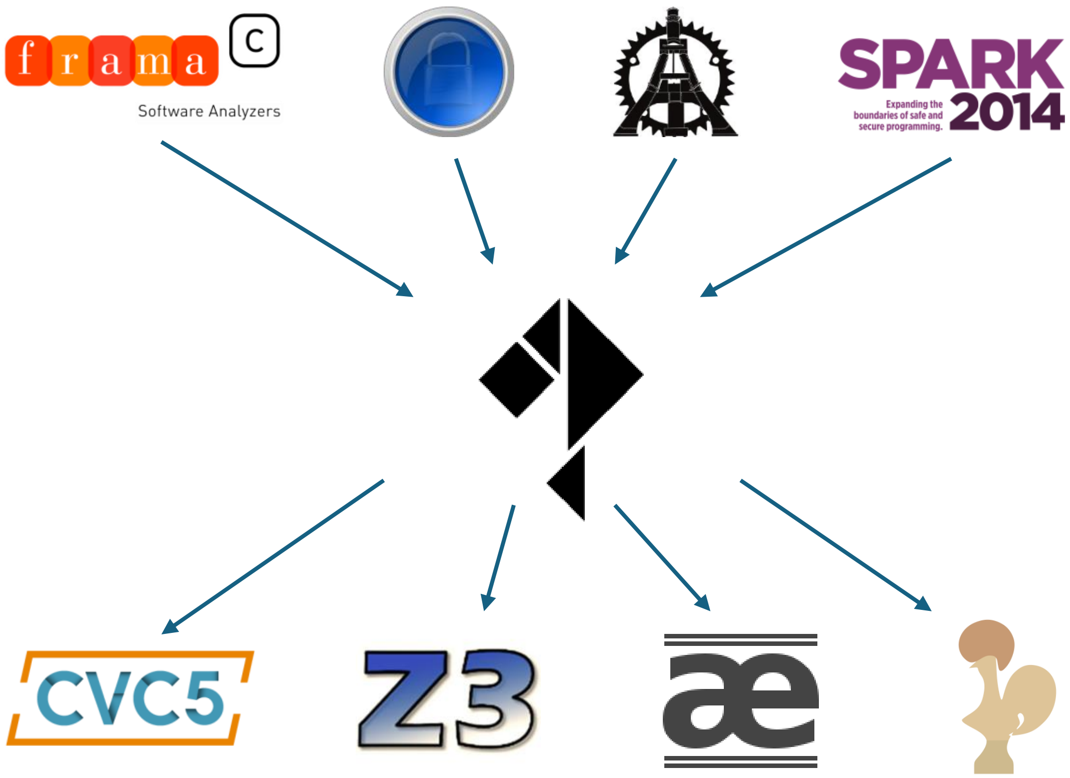


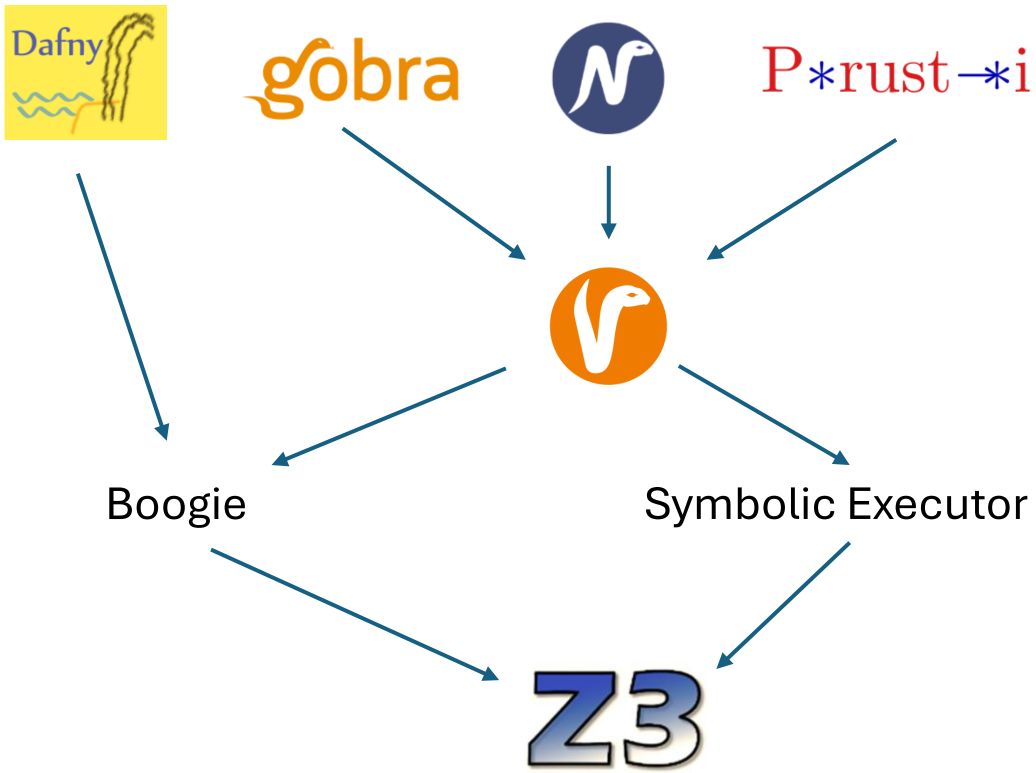
Semi-Automated Verifiers



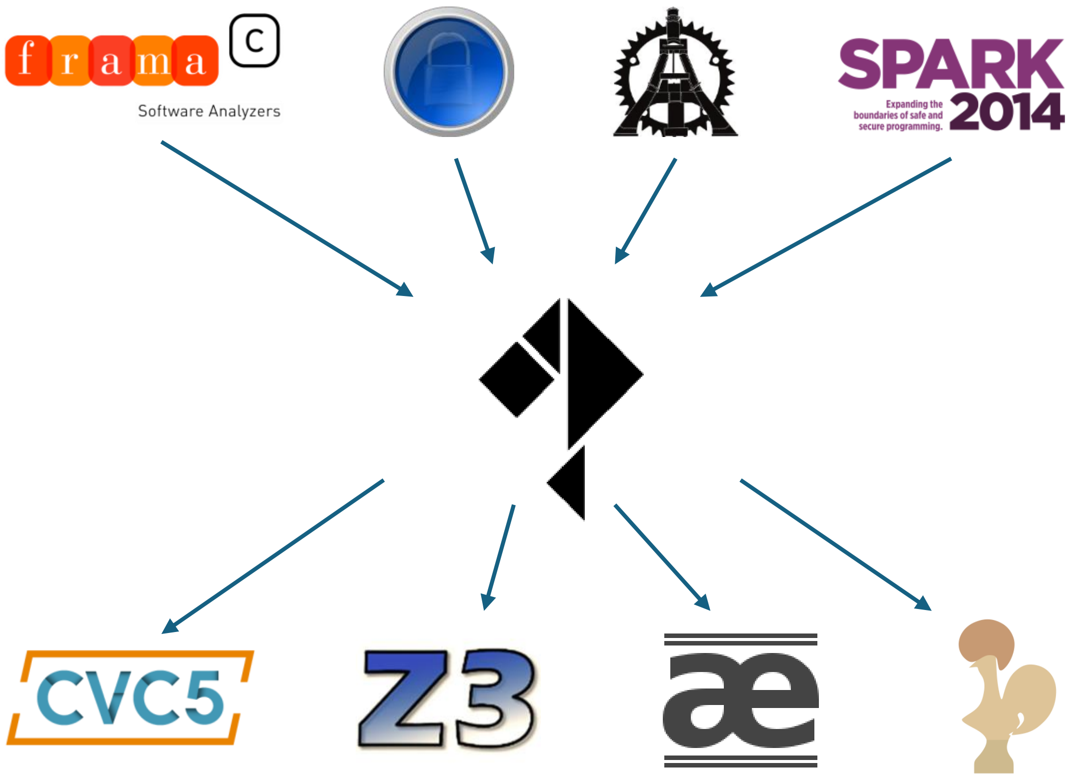


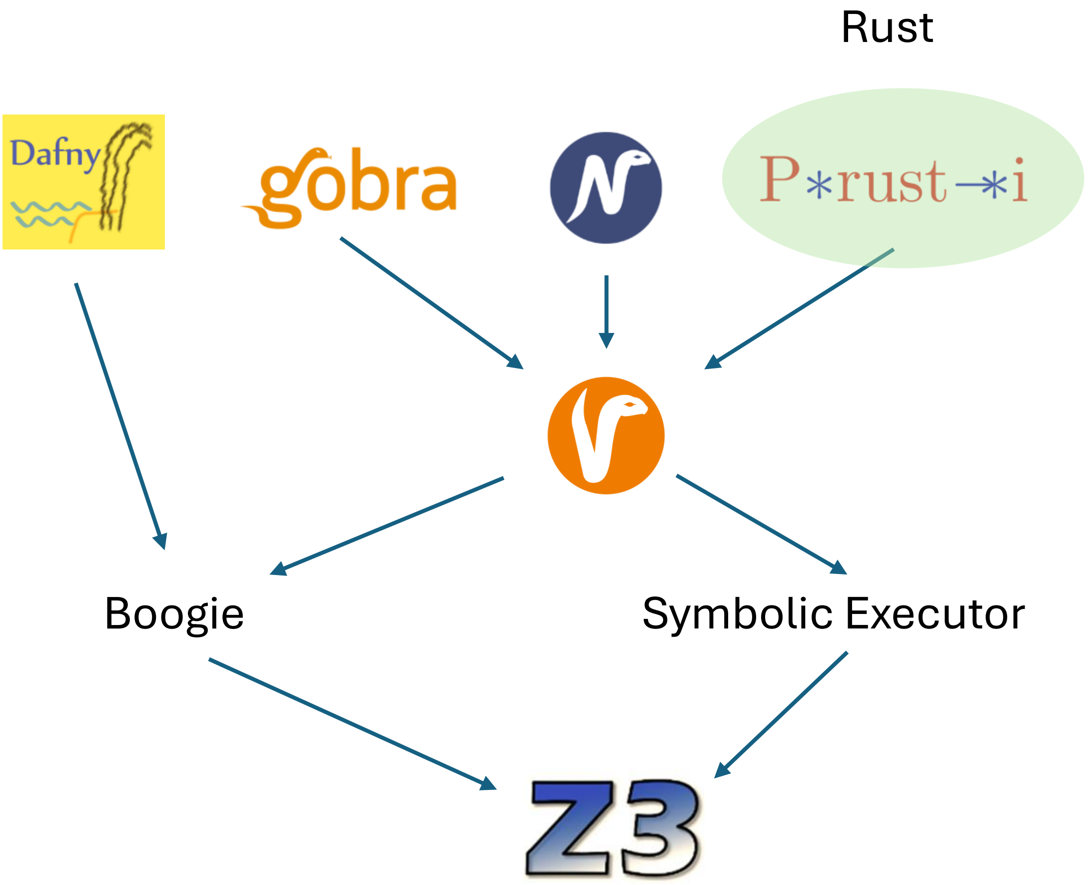
Semi-Automated Verifiers



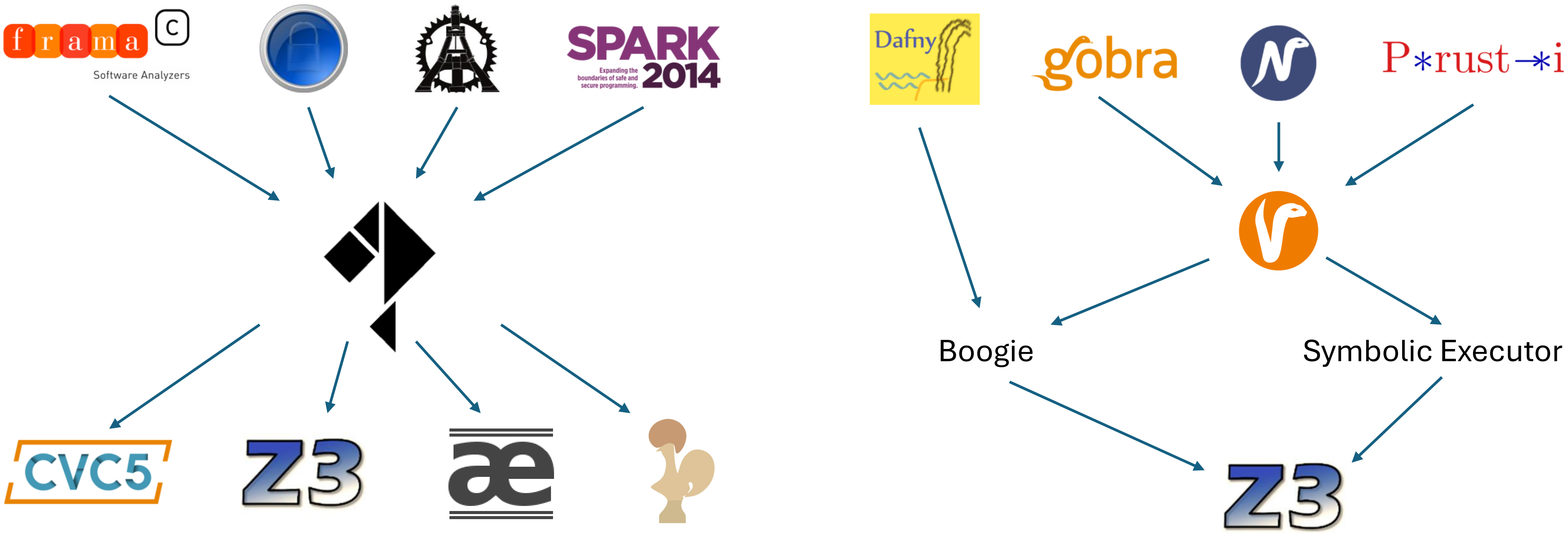


Semi-Automated Verifiers





Semi-Automated Verifiers

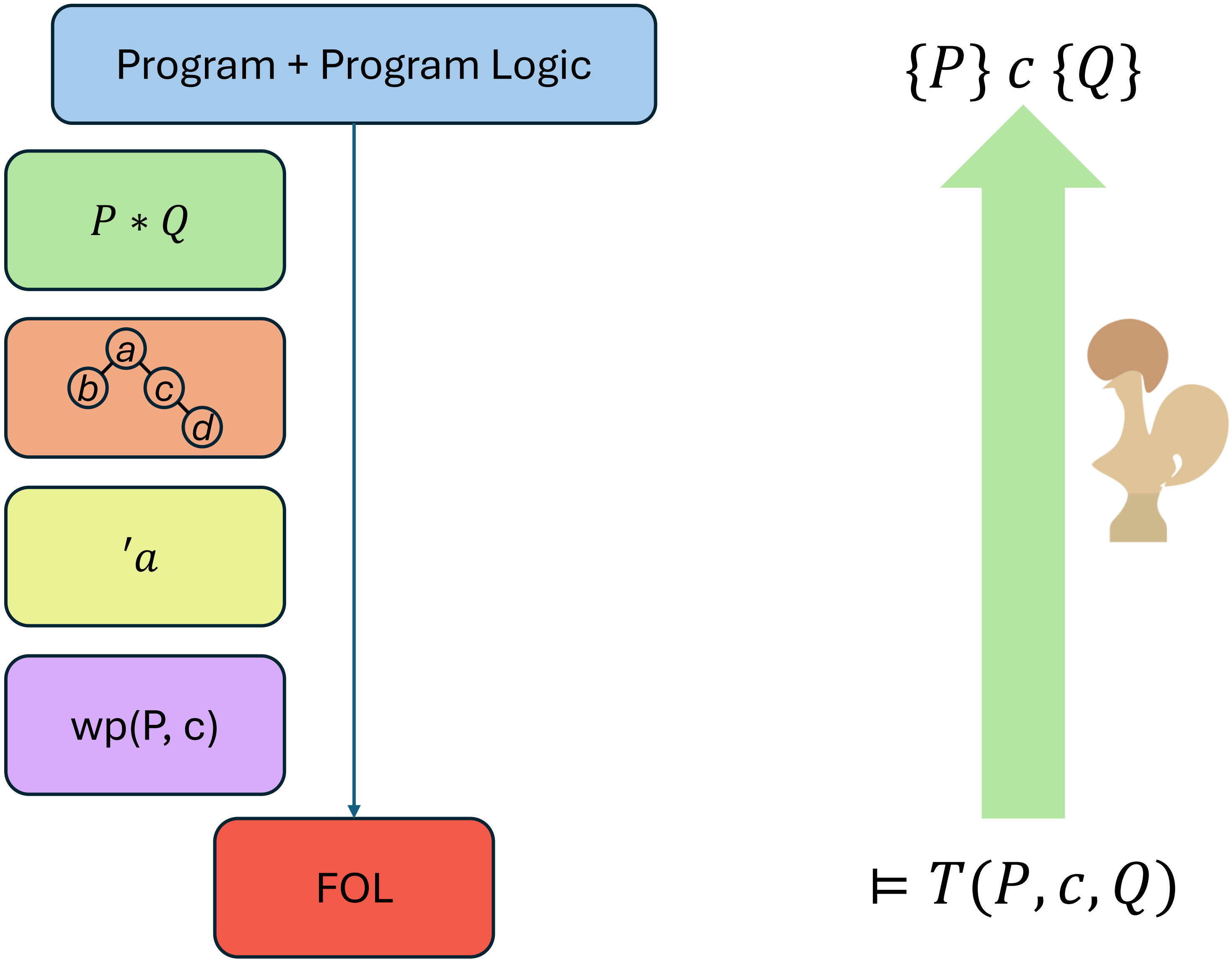


Semi-Automated Verifiers

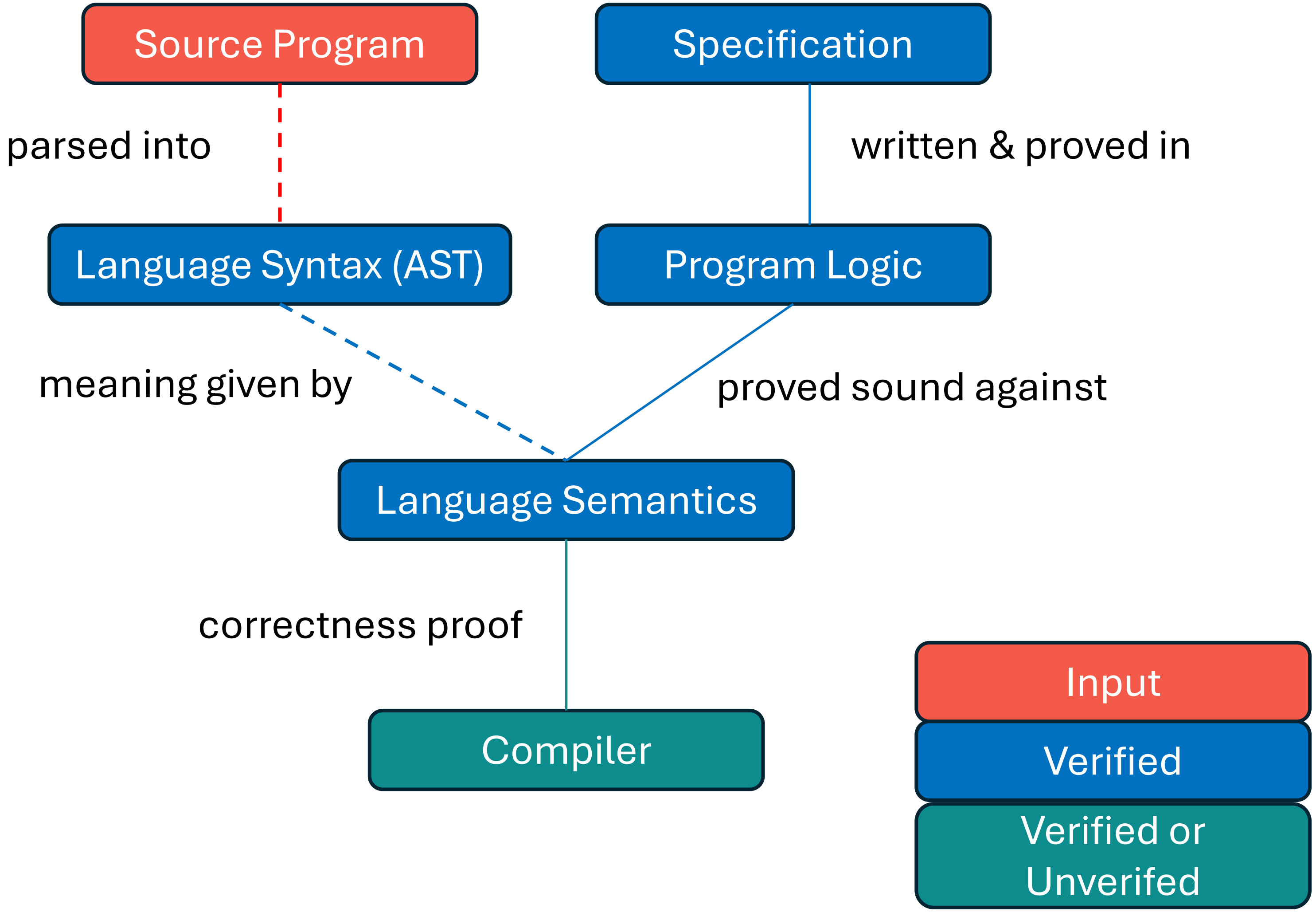
How do we know these tools are correct?

Symbolic Executor

Soundness



Foundational Verifiers



Verified Software Toolchain

CAKEML
A Verified Implementation of ML

Iris

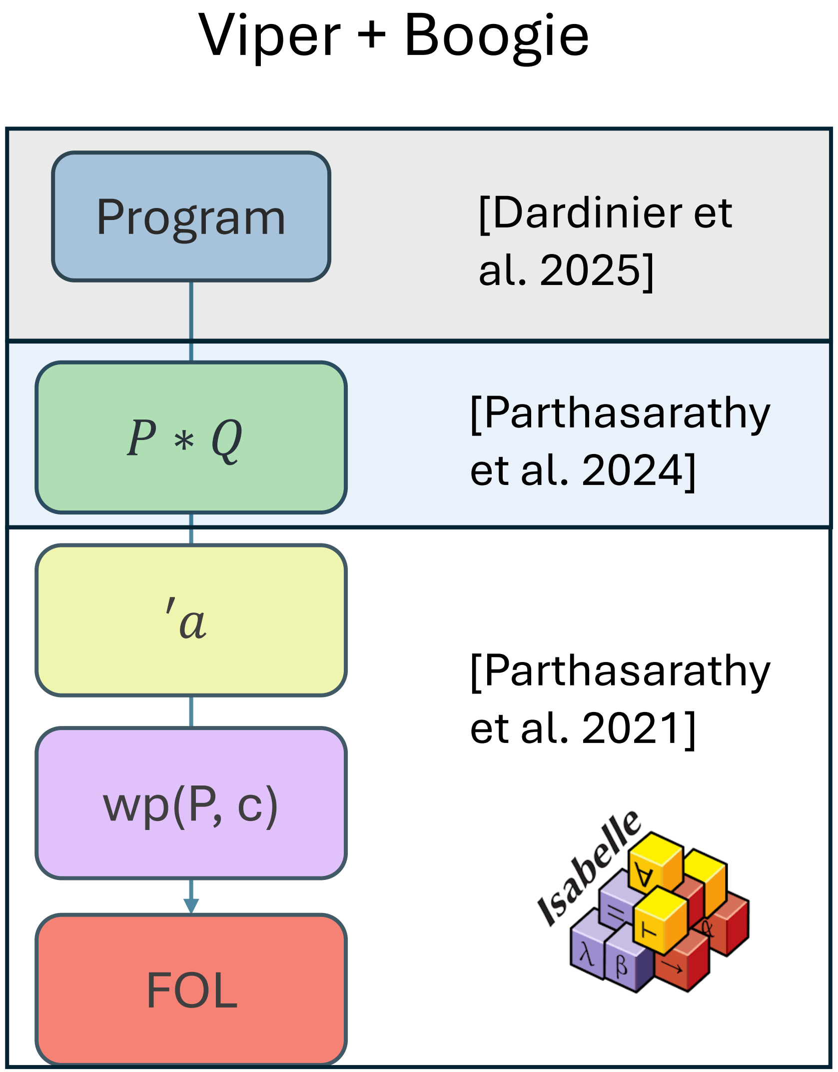
Bedrock2

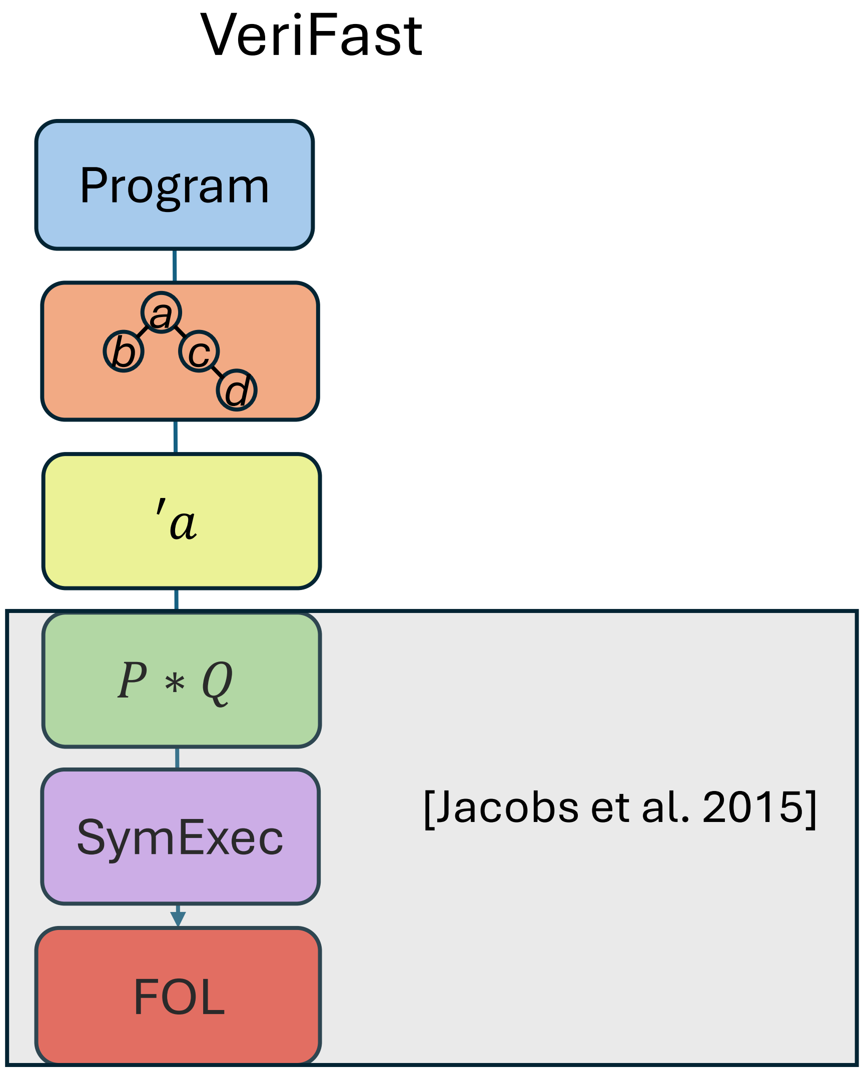
How can we achieve these soundness guarantees for IVL and SMT-based tools?

Not-Quite-Solutions

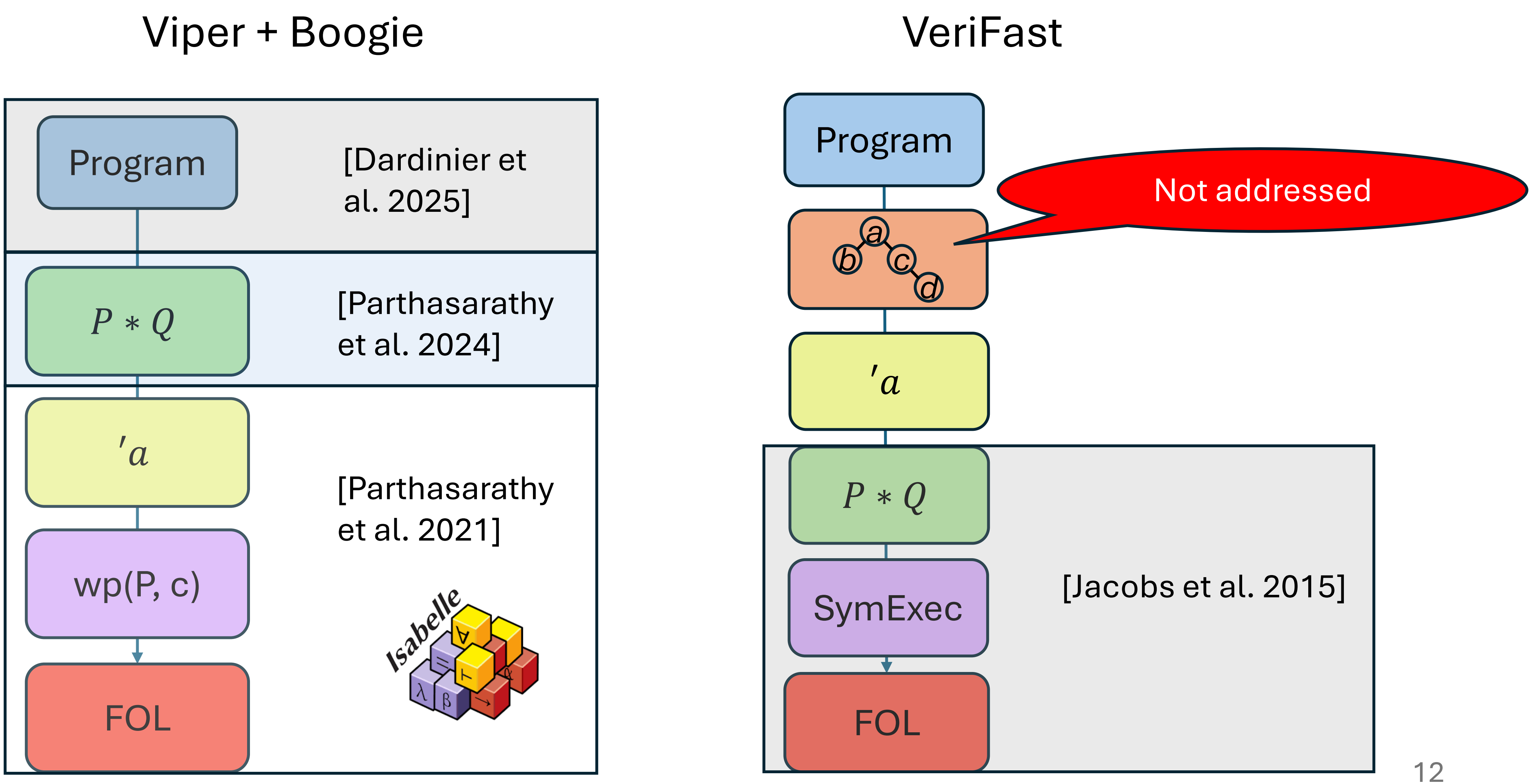
- Ways to use automated solvers in proof assistants
  - e.g. Sledgehammer [Blanchette et al. 2011], SMTCoq [Ekici et al. 2017], Sniper [Blot et al. 2023], Itauto [Besson 2021], CoqHammer [Czajka and Kaliszyk 2018]
- Improve automation of foundational tools
  - e.g. VST-A [Zhou et al. 2024], RefinedC [Sammler et al. 2021], RefinedRust [Gäher et al. 2024]

Verifying & Validating IVLs

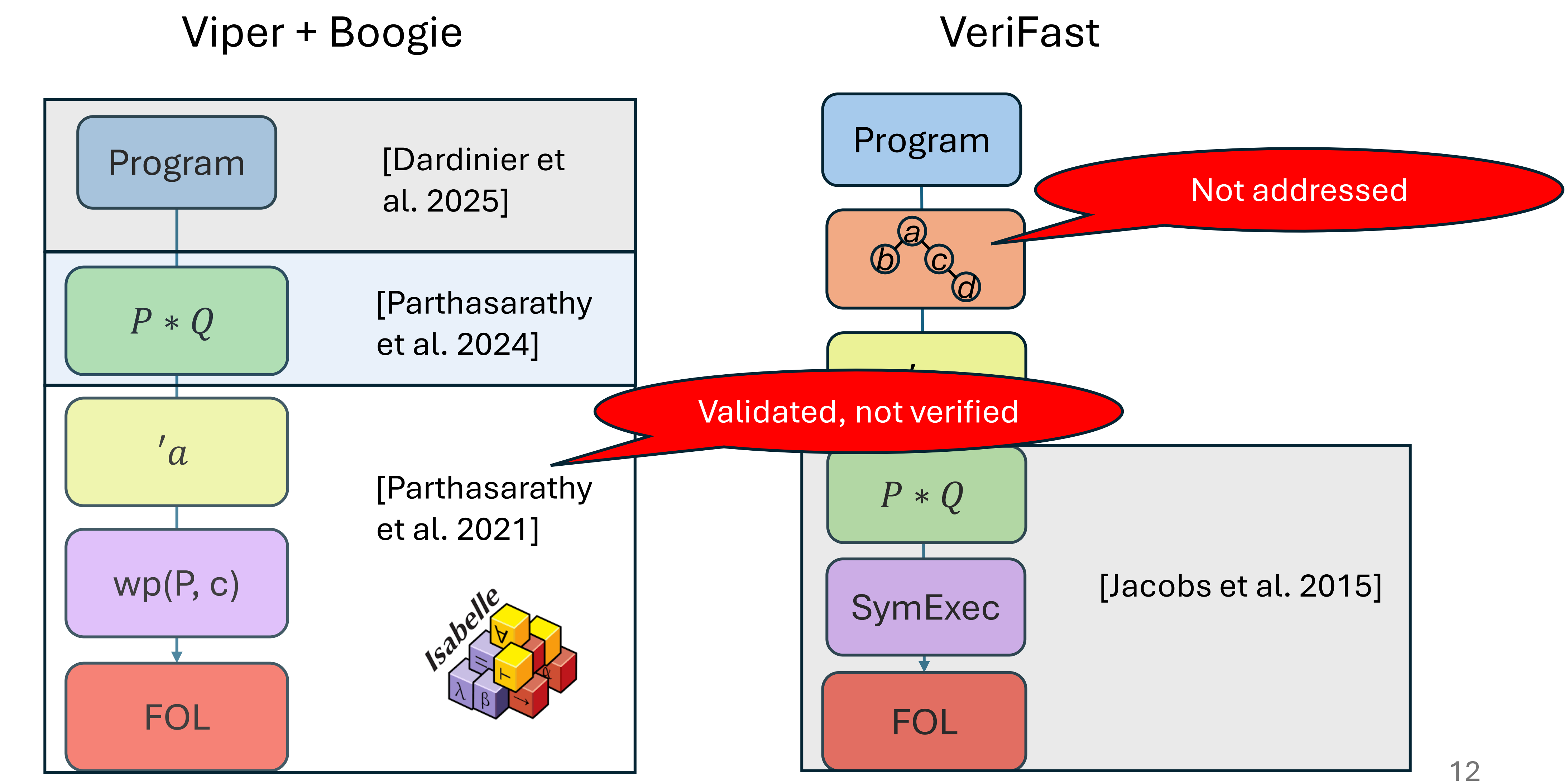




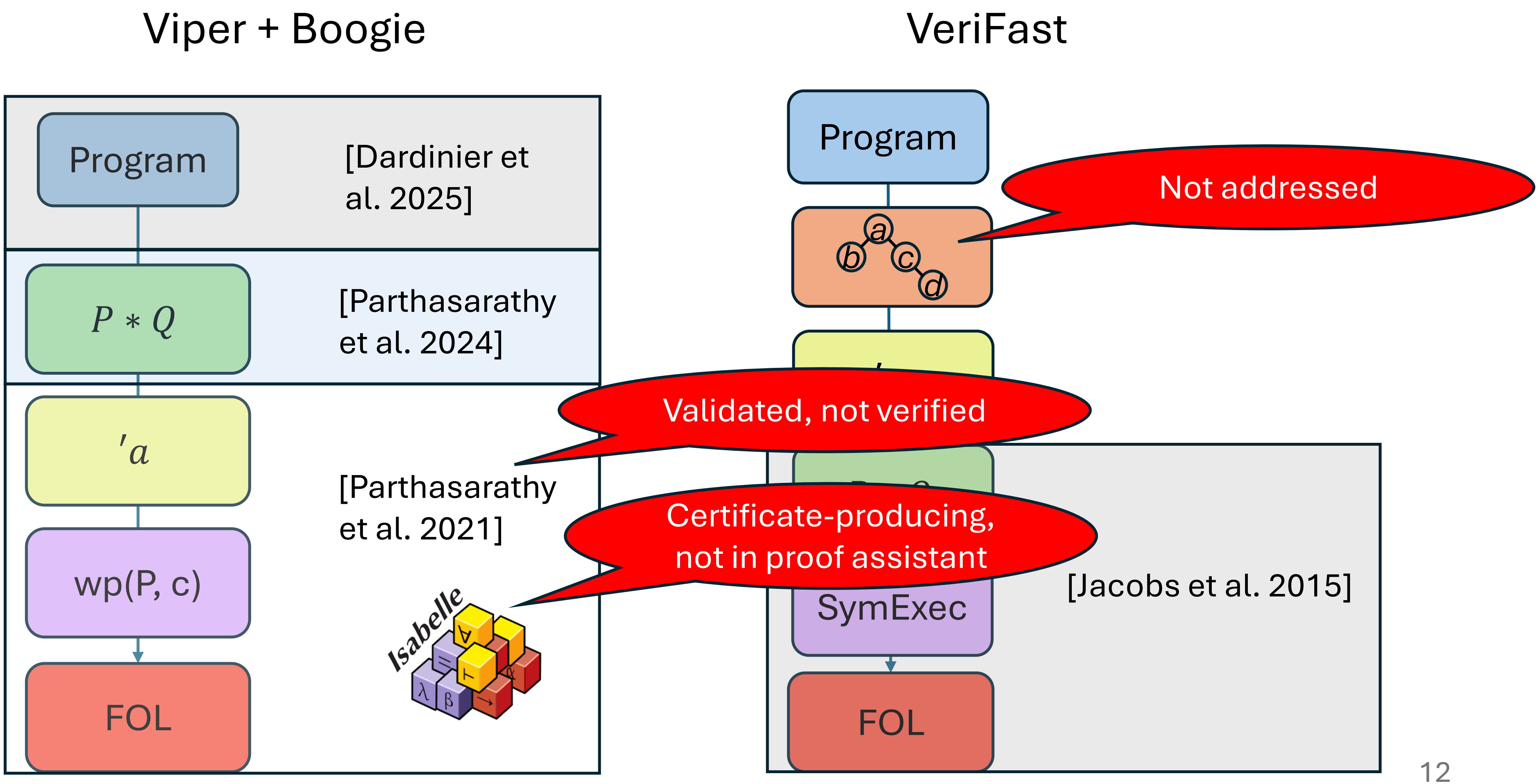
Verifying & Validating IVLs



Verifying & Validating IVLs



Verifying & Validating IVLs



This thesis: a verified implementation of Why3 IVL in Coq

Thesis Contributions

1. A novel formal semantics for Why3's logic
2. A proved-sound compiler from Why3 to (polymorphic) FOL, including
   - Pattern matching compilation
   - Algebraic Data Type axiomatization
3. Why3 API implementation in Coq
   - Method to implement stateful OCaml APIs in Coq
   - Resulting implementation executable both in Coq and OCaml
   - Run existing Why3 + EasyCrypt tests against our tool

Foundational Why3

Coq (Core) | External OCaml

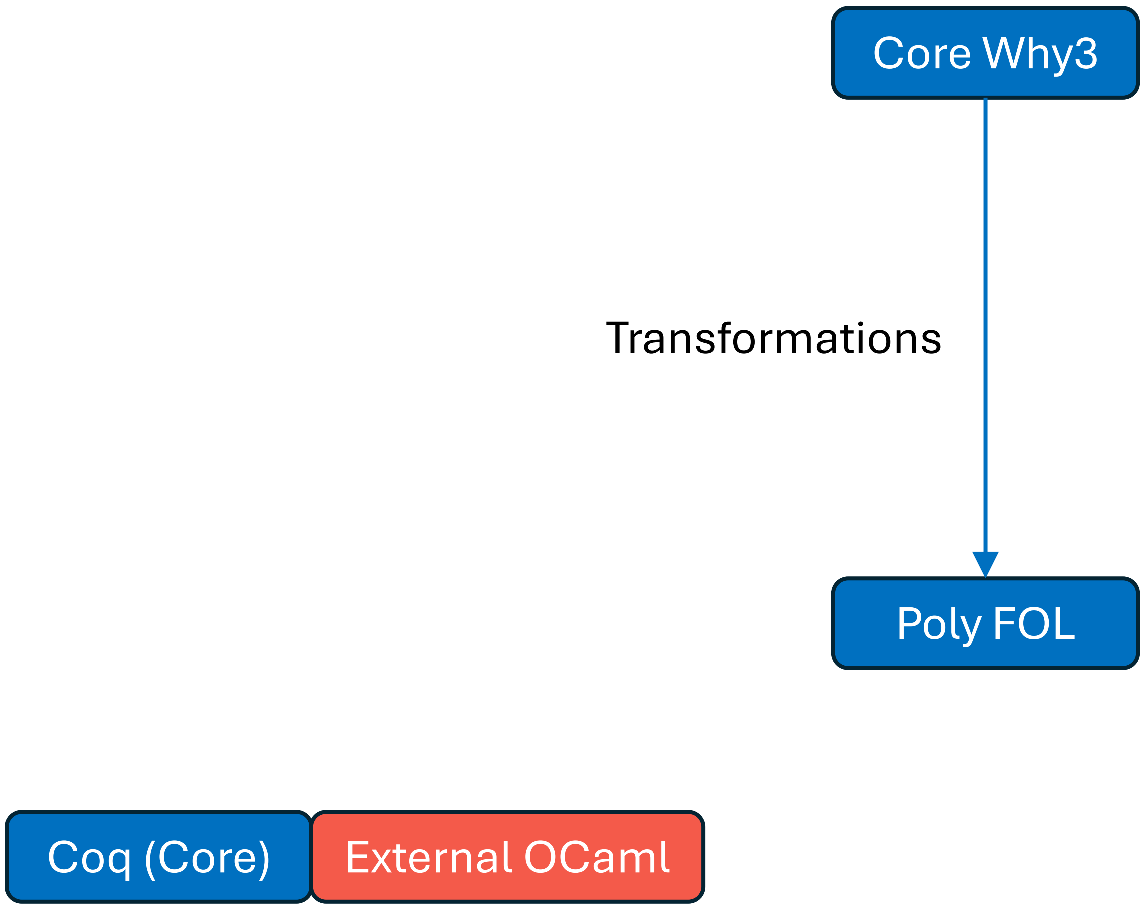
Foundational Why3

Core Why3

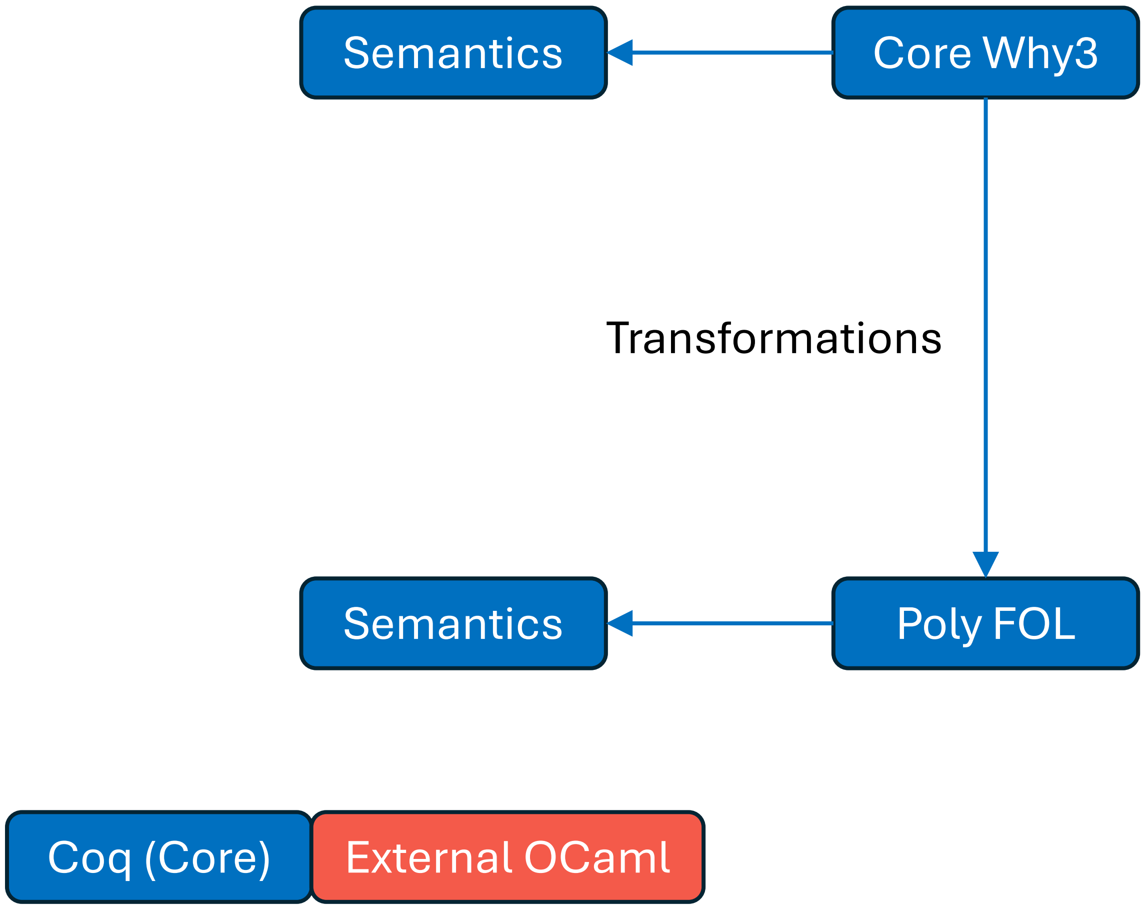
Coq (Core)

External OCaml

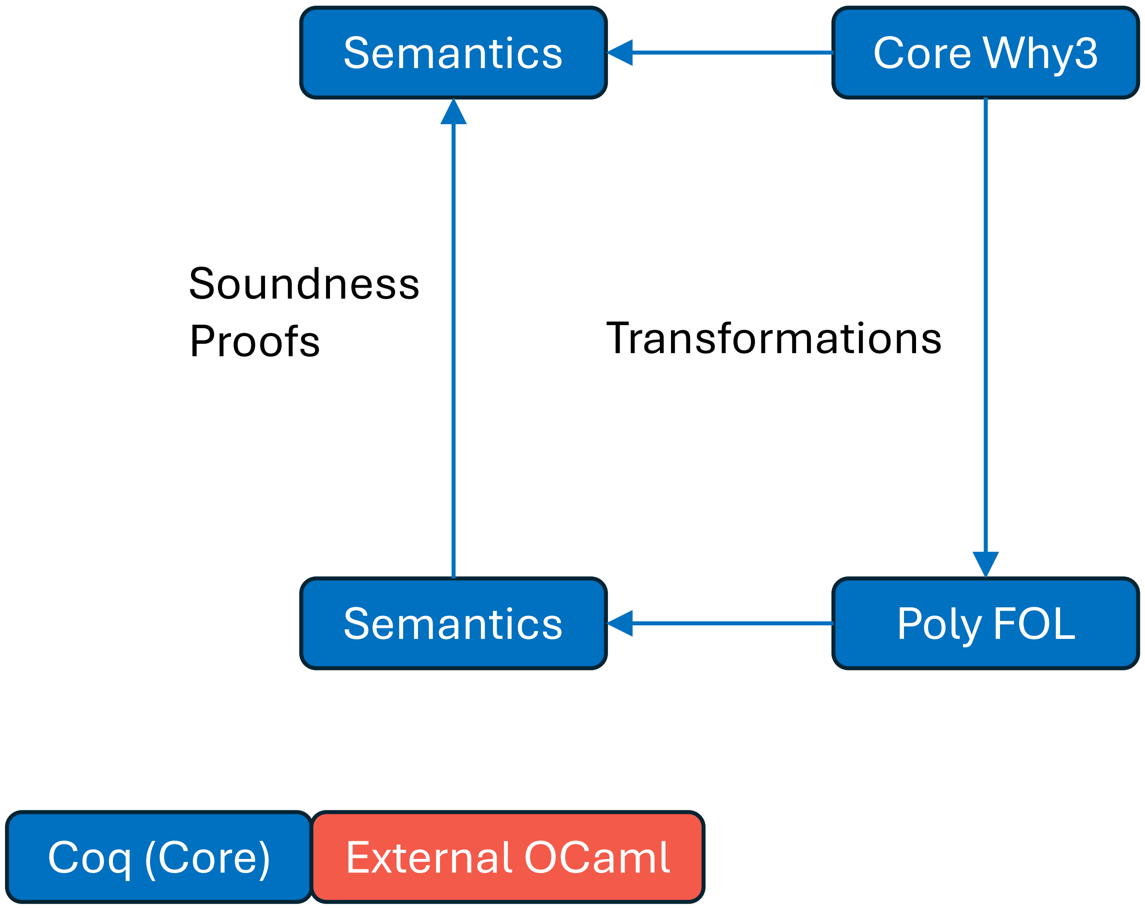
Foundational Why3



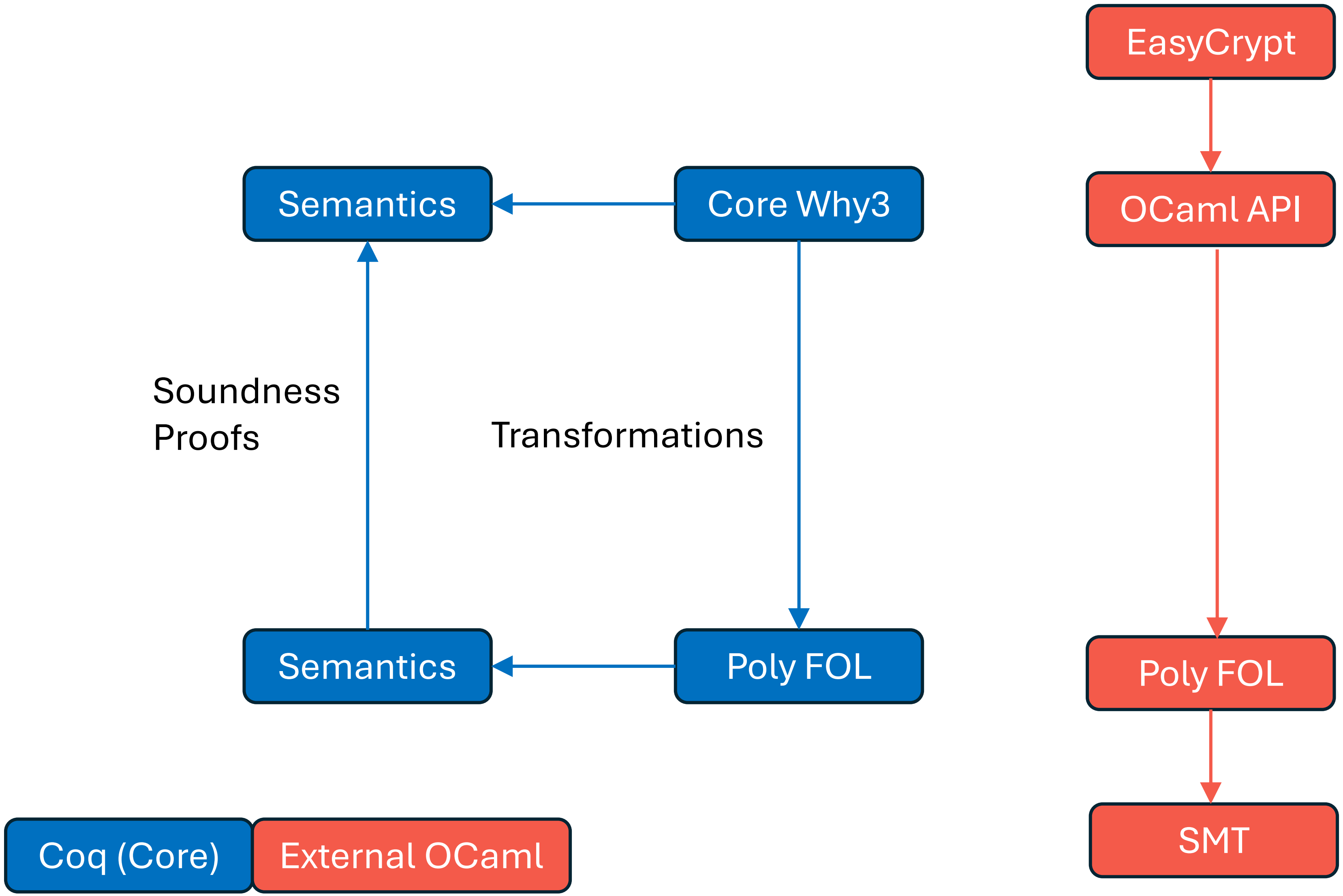
Foundational Why3



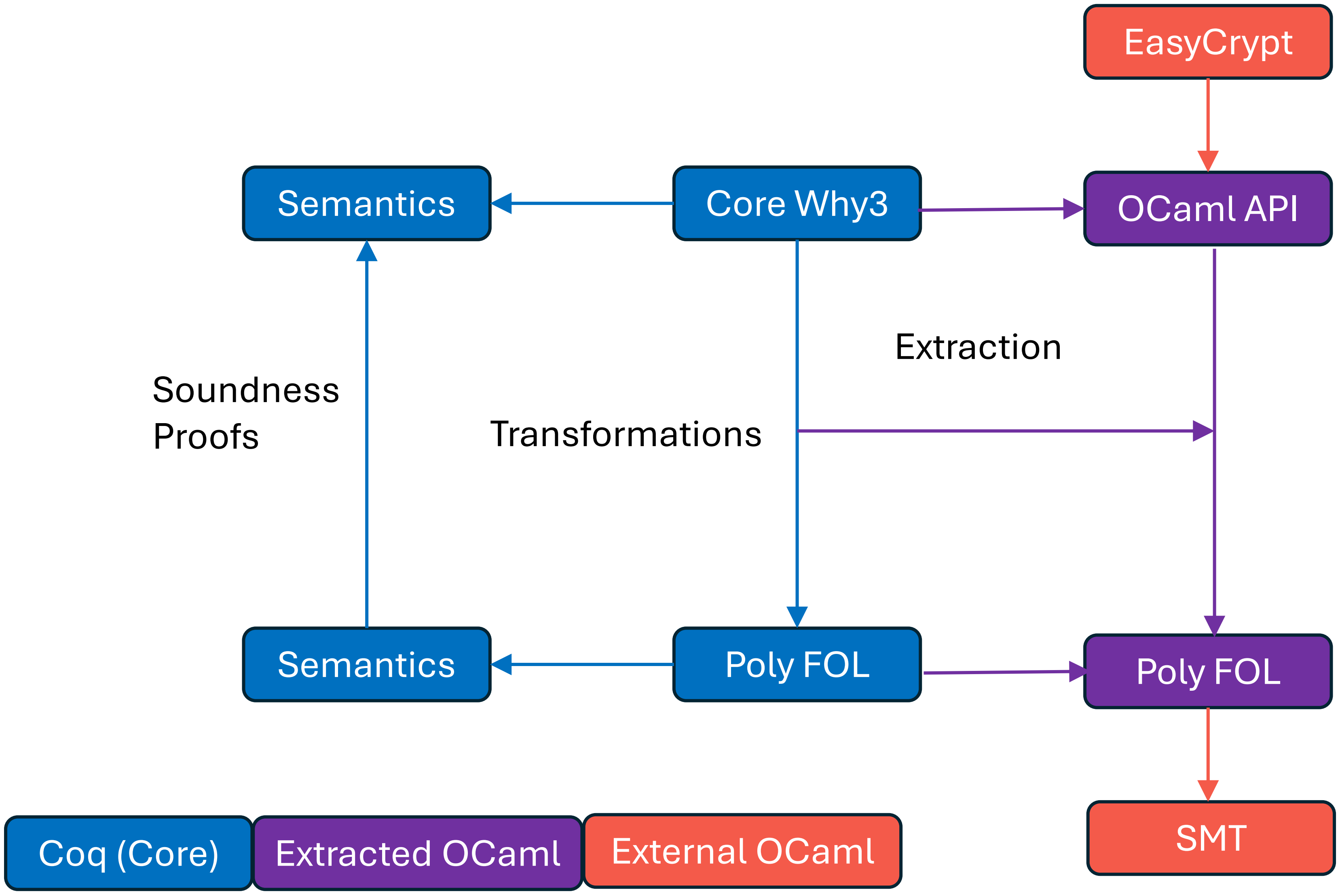
Foundational Why3



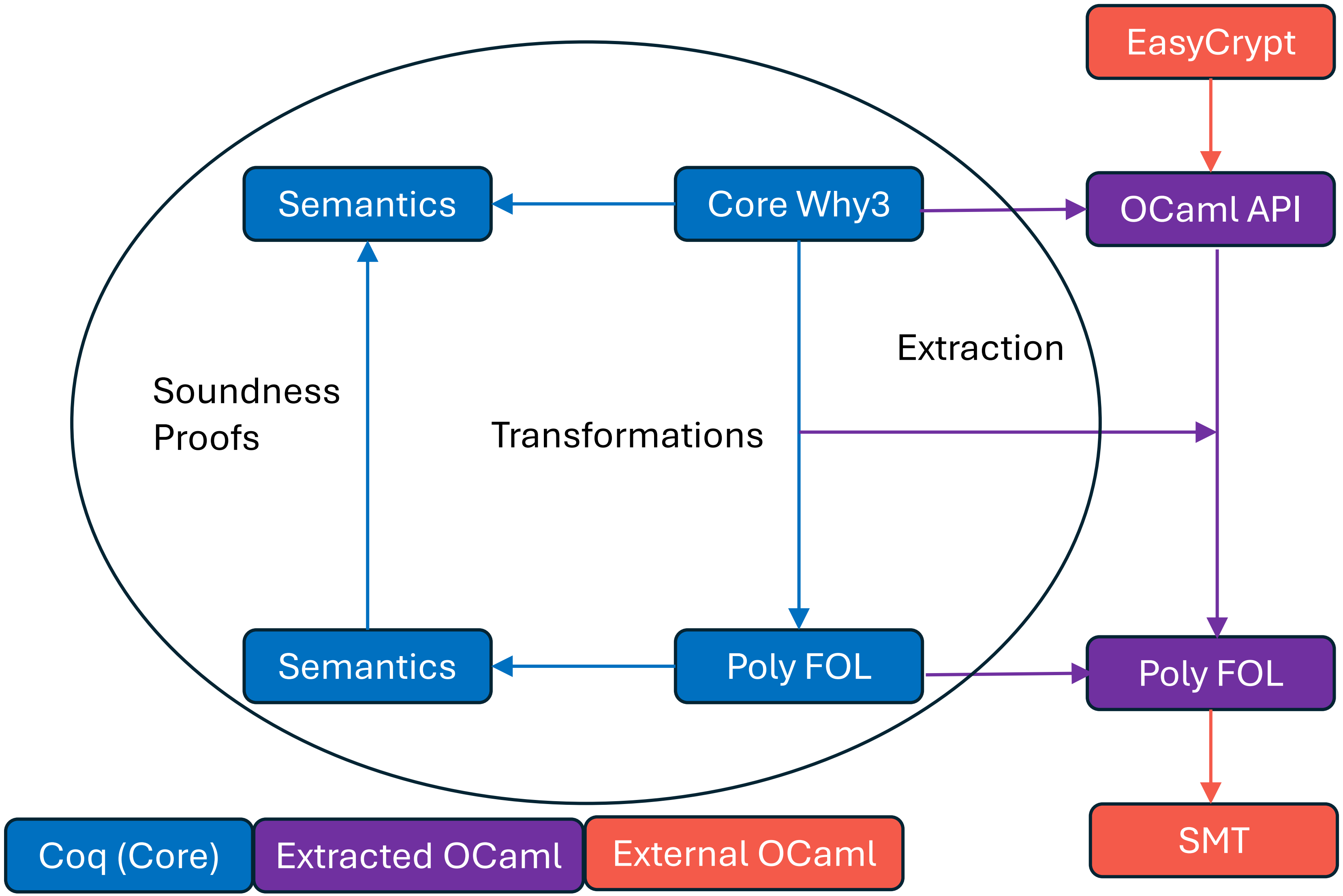
Foundational Why3



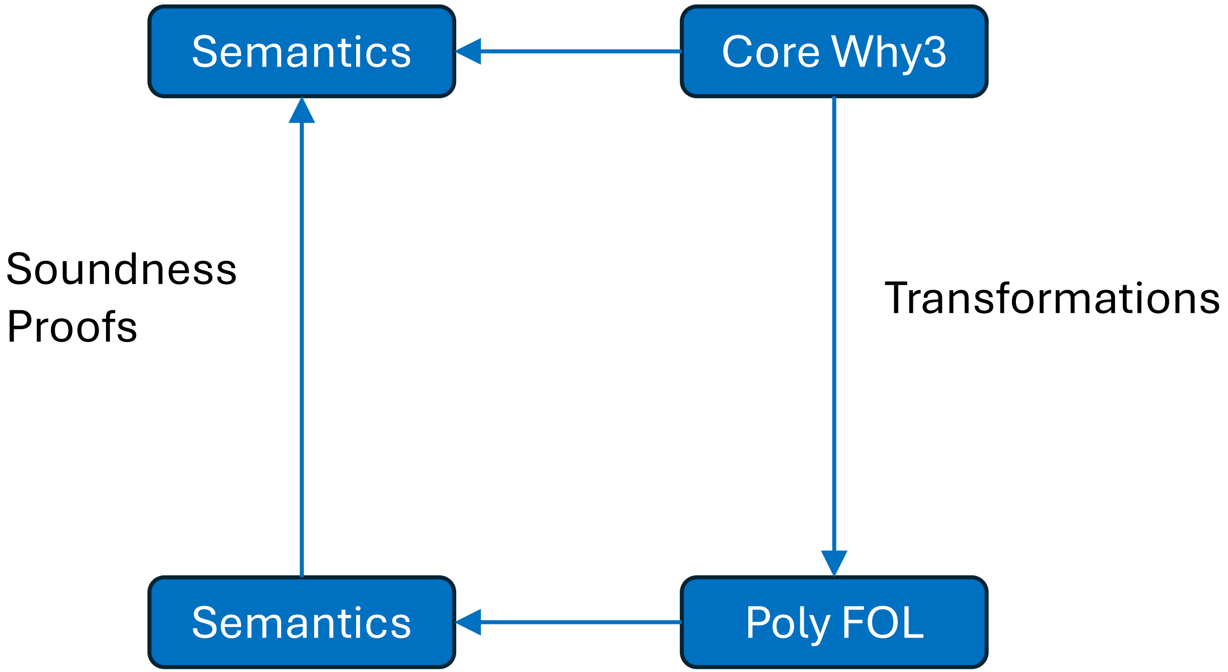
Foundational Why3



Foundational Why3



Core Why3


Semantics
Core Why3
Soundness
Proofs
Transformations
Semantics
Poly FOL

Proof Tasks and Entailment

$$(\underbrace{\Gamma}_{\text{Context}}, \underbrace{\Delta}_{\text{Hypotheses}}, \underbrace{g}_{\text{Goal}})$$

Proof Tasks and Entailment

$$(\Gamma, \Delta, g)$$

Context Hypotheses Goal

$$\Gamma, \Delta \vDash g$$

"(Γ, Δ, g) is entailed"

$\coloneqq$*entailed* (Γ, Δ, g)

Transformations

$$T : task \rightarrow list\ task$$

$$\forall\, t, typed(t) \rightarrow \left(\forall\, r\ \in T(t), entailed(r)\right) \rightarrow entailed(t)$$

If all outputs are entailed,

so was the input.

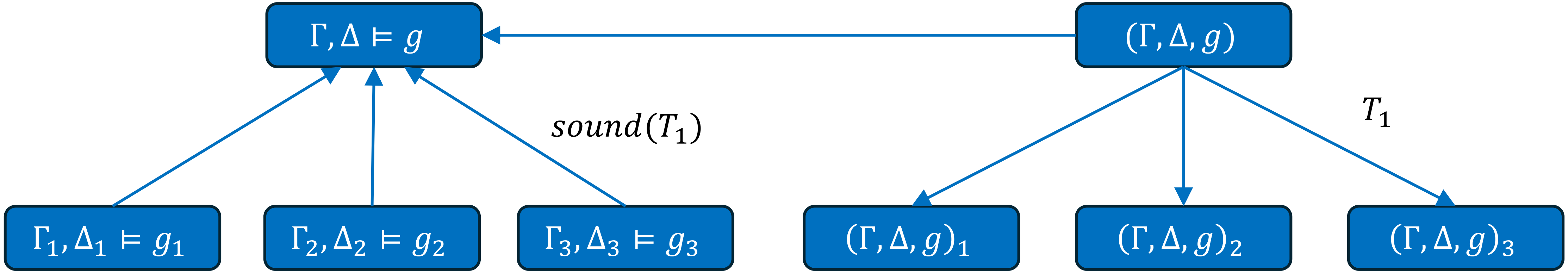
Example

$$T : task \rightarrow list\ task$$

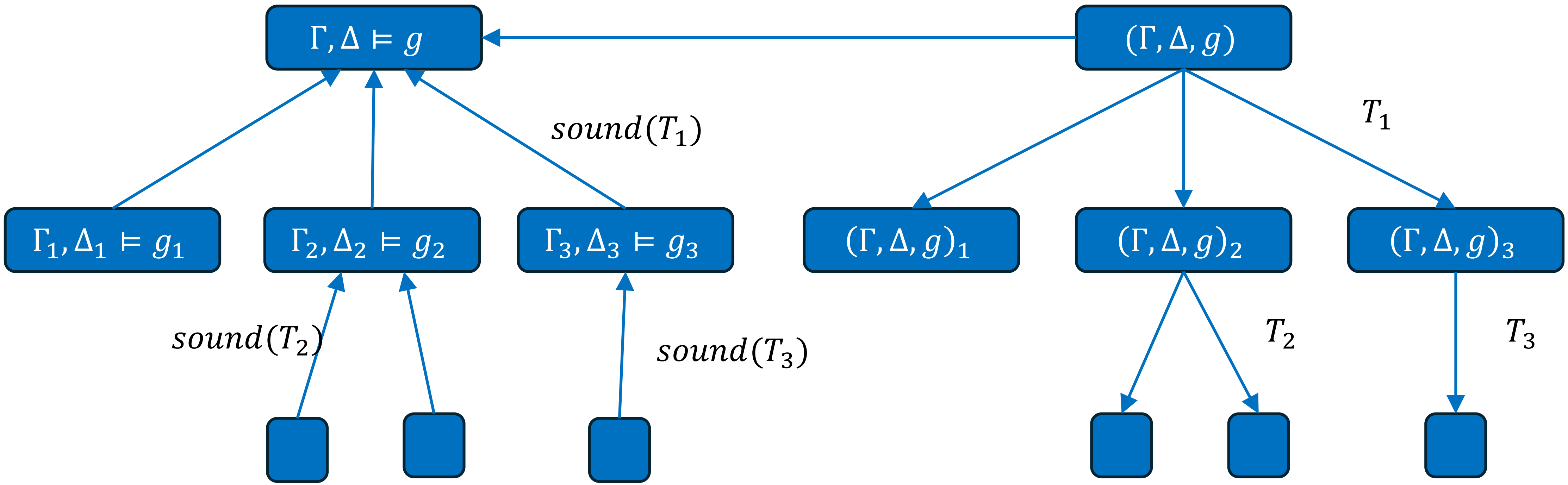
$$\forall\ t, typed(t) \rightarrow \left(\forall\ r\ \in T(t), entailed(r)\right) \rightarrow entailed(t)$$

$$T(\Gamma, \Delta, g_1 \wedge g_2) = \{(\Gamma, \Delta, g_1), (\Gamma, \Delta, g_2)\}$$

Back to Core Why3



Back to Core Why3



Thesis Contributions

1. **A novel formal semantics for Why3's logic**
2. A proved-sound compiler from Why3 to (polymorphic) FOL, including
   - Pattern matching compilation
   - Algebraic Data Type axiomatization
3. Why3 API implementation in Coq
   - Method to implement stateful OCaml APIs in Coq
   - Resulting implementation executable both in Coq and OCaml

Cohen, J. M., and Johnson-Freyd, P. A Formalization of Core Why3 in Coq. Proceedings of the ACM on Programming Languages 8, POPL (Jan. 2024), 60:1789–60:1818.

The Logic of Why3

Classical first-order logic with

1. Polymorphism
2. Let- and if-expressions
3. Algebraic data types*
4. Pattern matching
5. Recursive functions and predicates*
6. Inductive predicates*
7. Hilbert's epsilon choice operator

P-FOLDR (**P**olymorphic **F**irst-**O**rder **L**ogic with **D**atatypes and **R**ecursion

```
theory TreeForest
type list 'a = Nil | Cons  'a (list 'a)
type tree 'a = Leaf 'a | Node (tree 'a) (forest 'a)
with forest 'a = list (tree 'a)

use int.Int

function count_forest (f: forest int) : int =
  match f with
  | Nil -> 0
  | Cons t' f' -> count_tree t' + count_forest f'
  end
with count_tree (t: tree int) : int =
  match t with
  | Leaf i -> i
  | Node t' f' -> count_tree t' + count_tree f'
  end
```

Formalizing Why3

- Deep embedding of Why3 in Coq
- Formalize type system
  - Terms and formulas straightforward
  - (Mutual) well-foundedness checks:
    - Recursive function termination
    - Inductive predicate positivity
    - ADT non-emptiness
    - Pattern matching exhaustiveness
- Give verified typechecker

Semantics – Main Ideas

- Hilbert-style (denotational) semantics

$$[g_1 \wedge g_2] = [g_1] \wedge [g_2]$$

Why3 "and"

Meta-logic (Coq) "and"

- Give *model* of Why3 in Coq

Semantics – Main Ideas

- Hilbert-style (denotational) semantics

Interpretation of type, function, predicate symbols

Valuation of type and term variables

$$[g_1 \wedge g_2]_{I,v} = [g_1]_{I,v} \wedge [g_2]_{I,v}$$

Why3 “and”

Meta-logic (Coq) “and”

- *Truth* given by fixed interpretation and valuation
- *Validity* – true under all interpretations

Semantics: Types

$[f]_v : Prop$ for formula f

$[t]_v : [v(\tau)]$ for term t of Why3 type τ

Return type *depends* on Why3 type, interpretation, and valuation!

Semantics - Examples

$$[x]_v = v(x)$$

$$[\forall x, g]_v = \forall d, [g]_{v[x \to d]}$$

$$[f(\tau_1, \tau_2, \dots, \tau_m)(t_1, t_2, \dots t_n)]_v$$
$$= \left[f\left(v(\tau_1), v(\tau_2), \dots, v(\tau_m)\right)\right]_I([t_1]_v, \dots, [t_n]_v)$$

Function interpretation with type substitution

Heterogenous list of arguments

Semantics – Recursive Structures

Prior pen-and-paper description [Filliâtre 2013] merely imposes conditions on interpretation:

An *interpretation* is a pre-interpretation that is consistent with recursive and inductive definitions, that is:

- For any recursive definition `function` $\mathsf{f}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{x}) : \tau = t$ and any $\boldsymbol{s}$, we require $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!]$ to be such that, for all $\boldsymbol{t}$, $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t}) = [\![t]\!]_v$ where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$ and the $\boldsymbol{x}$ to the $\boldsymbol{t}$ (and similarly for a predicate definition).
- For any inductive definition $\mathsf{p}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{\tau}) = f_1 \mid \ldots \mid f_l$ and any $\boldsymbol{s}$, we require $[\![\mathsf{p}\langle\boldsymbol{s}\rangle]\!]$ to be the least predicate such that $[\![f_1]\!]_v, \ldots, [\![f_l]\!]_v$ hold where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$.

Semantics – Recursive Structures

Prior pen-and-paper description [Filliâtre 2013] merely imposes conditions on interpretation:

An *interpretation* is a pre-interpretation that is consistent with recursive and inductive definitions, that is:

- For any recursive definition `function` $\mathsf{f}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{x}) : \tau = t$ and any $\boldsymbol{s}$, we require $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!]$ to be such that, for all $\boldsymbol{t}$, $[\![\mathsf{f}\langle\boldsymbol{s}\rangle]\!](\boldsymbol{t}) = [\![t]\!]_v$ where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$ and the $\boldsymbol{x}$ to the $\boldsymbol{t}$ (and similarly for a predicate definition).
- For any inductive definition $\mathsf{p}\langle\boldsymbol{\alpha}\rangle(\boldsymbol{\tau}) = f_1 \mid \ldots \mid f_l$ and any $\boldsymbol{s}$, we require $[\![\mathsf{p}\langle\boldsymbol{s}\rangle]\!]$ to be the least predicate such that $[\![f_1]\!]_v, \ldots, [\![f_l]\!]_v$ hold where v maps the $\boldsymbol{\alpha}$ to the $\boldsymbol{s}$.

How do we know that such interpretations (models) exist?
If not, every formula is valid!

Recursive Structures

We give explicit, generic constructions satisfying these properties:

- ADTs → **W-types**
- Inductive predicates → **impredicative encoding**
- Recursive functions → **well-founded recursion**

Users of semantics need only properties, not complex encodings

Algebraic Data Types

1. Constructors are injective: if $[c](t_1) = [c](t_2)$, then $t_1 = t_2$
2. Constructors are disjoint: if $[c_1](t_1) = [c_2](t_2)$, then $c_1 = c_2$
3. There is a (computable) function $\boldsymbol{find}$ that gives the constructor c and arguments t for any element x of ADT type such that
$$x = [c](t)$$
4. A generalized induction principle holds

Implement ADTs using W-types, prove these properties satisfied

Pattern matching: describe new bound variables, use $\boldsymbol{find}$ for constructors

Recursive Functions

- Relies on many parts of typing and semantics:

1. Define well-founded relation on W-type encoding denoting structural inclusion
2. Prove (via induction on ADT interpretation) that pattern matching produces “smaller” variables
3. Use termination check to show recursion occurs on variables from pattern match

Entailment Revisited

Entailment Revisited

$$\Gamma, \Delta \vDash g$$

Entailment Revisited

$$\Gamma, \Delta \vDash g$$

$$\forall I, full(I) \rightarrow \left(\forall d\ v, d \in \Delta \rightarrow [d]_{I,v}\right) \rightarrow (\forall v, [g]_{I,v})$$

Entailment Revisited

$$\Gamma, \Delta \vDash g$$

$$\forall I, full(I) \to \left(\forall d\ v, d \in \Delta \to [d]_{I,v}\right) \to (\forall v, [g]_{I,v})$$

For every interpretation I consistent with defined types, functions, and predicates,

Entailment Revisited

$$\Gamma, \Delta \vDash g$$

$$\forall I, full(I) \to \left(\forall d\ v, d \in \Delta \to [d]_{I,v}\right) \to (\forall v, [g]_{I,v})$$

For every interpretation I consistent with defined types, functions, and predicates,

If every formula in Δ is satisfied by I,

Entailment Revisited

$$\Gamma, \Delta \vDash g$$

$$\forall I, full(I) \rightarrow \left(\forall d\ v, d \in \Delta \rightarrow [d]_{I,v}\right) \rightarrow (\forall v, [g]_{I,v})$$

For every interpretation I consistent with defined types, functions, and predicates,

If every formula in Δ is satisfied by I,

Then g is satisfied by I.

P-FOLDR as a Logic

Consistency:

```
Theorem consistent (pd: pi_dom) (pdf: pi_dom_full gamma pd)
(pf: pi_funpred gamma_valid pd pdf)
(pf_full: full_interp gamma_valid pd pf) (f: formula)
(f_typed: formula_typed gamma f):
~ (satisfies pd pdf pf pf_full f f_typed /\
  satisfies pd pdf pf pf_full (Fnot f) (F_Not f_typed)).
```

Existence of models:

```
Theorem full_interp_exists: forall funi predi,
  {pf: pi_funpred gamma_valid pd pdf |
    full_interp gamma_valid pd pf /\
    (forall f srts a, In (abs_fun f) gamma ->
      (funs gamma_valid pd pf ) f srts a = funi f srts a) /\
    (forall p srts a, In (abs_pred p) gamma ->
      (preds gamma_valid pd pf) p srts a = predi p srts a)}.
```

A Sound Proof System for P-FOLDR

- Build sound-by-construction, natural-deduction-style proof system and tactic system
- Prove all introduction and elimination rules for connectives
- Prove rules for induction, unfolding recursive functions, type substitution, rewriting, etc
- Prove lemmas from Why3's standard library about lists and trees, e.g.

```
lemma inorder_length: ∀ t : tree 'a, length (inorder t) = size t
```

- Gives confidence that our semantics matches the intended one

Thesis Contributions

1. A novel formal semantics for Why3's logic
2. **A proved-sound compiler from Why3 to (polymorphic) FOL, including**
    - **Pattern matching compilation**
    - Algebraic Data Type axiomatization
3. Why3 API implementation in Coq
    - Method to implement stateful OCaml APIs in Coq
    - Resulting implementation executable both in Coq and OCaml

Pattern Matching in P-FOLDR

$$
\begin{aligned}
p \coloneqq \quad & \mid \_ \\
& \mid x \\
& \mid c(p_1, \ldots, p_n) \\
& \mid p_1 | p_2 \\
& \mid p \; as \; x
\end{aligned}
$$

Pattern Matching in P-FOLDR

$$
\begin{aligned}
p \coloneqq \quad & \mid \_ \\
& \mid x \\
& \mid c(p_1, \ldots, p_n) \\
& \mid p_1 | p_2 \\
& \mid p \ as \ x
\end{aligned}
$$

Complicated!

- Nested matching
- Simultaneous matching
- Interactions with termination checking

Pattern Matching in P-FOLDR

$$
\begin{aligned}
p \coloneqq \quad & \mid \_ \\
& \mid x \\
& \mid c(p_1, \dots, p_n) \\
& \mid p_1 | p_2 \\
& \mid p \ as \ x
\end{aligned}
$$

Complicated!

- Nested matching
- Simultaneous matching
- Interactions with termination checking

We defined pattern/matching

- Syntax
- Typing
- Semantics

Semantics based on describing valuations of newly bound variables

Pattern Matching Compilation

```
match l1, l2 with
| [], [] -> x1
| [_], _ -> x2
| _ :: _, _ -> x3
| [], _ :: _ -> x4
end
```

```
match l1 with
| [] -> match l2 with
        | [] -> x1
        | y3 :: y4 -> x4
        end
| y1 :: y2 ->
        match y2 with
        | [] -> x2
        | y5 :: y6 -> x3
        end
end
```

Pattern Matching Compilation

```
match l1, l2 with
| [], [] -> x1
| [_], _ -> x2
| _ :: _, _ -> x3
| [], _ :: _ -> x4
end
```

```
match l1 with
| [] -> match l2 with
        | [] -> x1
        | y3 :: y4 -> x4
        end
| y1 :: y2 ->
        match y2 with
        | [] -> x2
        | y5 :: y6 -> x3
        end
end
```

```
match l1, l2 with
| [], [] -> x1
| [_], _ -> x2
| _ :: _, _ -> x3
end
```

Non-exhaustive!

Pattern Matching Compilation

- Widely applicable and well-studied problem: e.g. OCaml, Haskell, Coq

[Augustsson 1985], [Baudinet and MacQueen 1985], [Laville 1988], [Puel and Suarez 1990], [Maranget 1992], [Pettersson 1992], [Sekar et al. 1995], [Sestoft 1996], [Scott and Ramsey 2000], [Le Fessant and Maranget 2001], [Maranget 2007], [Maranget 2008], [Karachalias 2015], [Tuerk et al. 2015]

Pattern Matching Compilation

- Widely applicable and well-studied problem: e.g. OCaml, Haskell, Coq

[Augustsson 1985], [Baudinet and MacQueen 1985], [Laville 1988], [Puel and Suarez 1990], [Maranget 1992], [Pettersson 1992], [Sekar et al. 1995], [Sestoft 1996], [Scott and Ramsey 2000], [Le Fessant and Maranget 2001], [Maranget 2007], [Maranget 2008], [Karachalias 2015], [Tuerk et al. 2015]

Pattern Matching Compilation

- Widely applicable and well-studied problem: e.g. OCaml, Haskell, Coq

[Augustsson 1985], [Baudinet and MacQueen 1985], [Laville 1988], [Puel and Suarez 1990], [Maranget 1992], [Pettersson 1992], [Sekar et al. 1995], [Sestoft 1996], [Scott and Ramsey 2000], [Le Fessant and Maranget 2001], [Maranget 2007], [Maranget 2008], [Karachalias 2015], [Tuerk et al. 2015]

Pattern Matching Compilation [Le Fassant and Maranget 2001, Maranget 2008]

```
match l1, l2 with
| [], [] -> x1
| [_], _ -> x2
| _ :: _, _ -> x3
| [], _ :: _ -> x4
end
```

$$\begin{pmatrix} nil & nil & x_1 \\ cons(\_, nil) & nil & x_2 \\ cons(\_, \_) & \_ & x_3 \\ nil & cons(\_, \_) & x_4 \end{pmatrix}$$

Pattern Matching Compilation [Le Fassant and Maranget 2001, Maranget 2008]

```
match l1, l2 with
| [], [] -> x1
| [_], _ -> x2
| _ :: _, _ -> x3
| [], _ :: _ -> x4
end
```

$$\begin{pmatrix} nil & nil & x_1 \\ cons(\_, nil) & nil & x_2 \\ cons(\_,\_) & \_ & x_3 \\ nil & cons(\_,\_) & x_4 \end{pmatrix}$$

$S(c, P)$: rest of match, assuming the first term matches constructor c

$$S(nil, P) = \begin{pmatrix} nil & x_1 \\ cons(\_,\_) & x_4 \end{pmatrix} \qquad S(cons, P) = \begin{pmatrix} \_ & nil & nil & x_2 \\ \_ & \_ & \_ & x_3 \end{pmatrix}$$

1. Termination

- Recurse on decompositions, not subterms
- “or” patterns and added wildcards make matrix larger
- Prove generic termination results for any matrix-based algorithm

2. Semantics

If $compile(P, ts) = Some\ t$, then $\left[[ts]_v, P\right]_v = Some\ [t]_v$

- Relies on semantics of pattern matrix matching
- Purely *semantic* reasoning – need to reason about ADTs, $\boldsymbol{find}(x)$, interpretation
- Existing proofs in literature – based on *syntactic* match relation on values

$$c(v_1, \dots, v_n) \preccurlyeq c(p_1, \dots, p_n) \leftrightarrow \forall i, v_i \preccurlyeq p_i$$

- Different than our setting: match semantics depends on interpretation!

3. Exhaustiveness

Corollary of semantic correctness:

$$\text{If } \left[[ts]_v, P\right]_v = None, \text{ then } compile(P, ts) = None$$

- Different than proofs in the literature [Maranget 2007]
- Adapt proofs for purely logical setting vs call-by-value or lazy evaluation

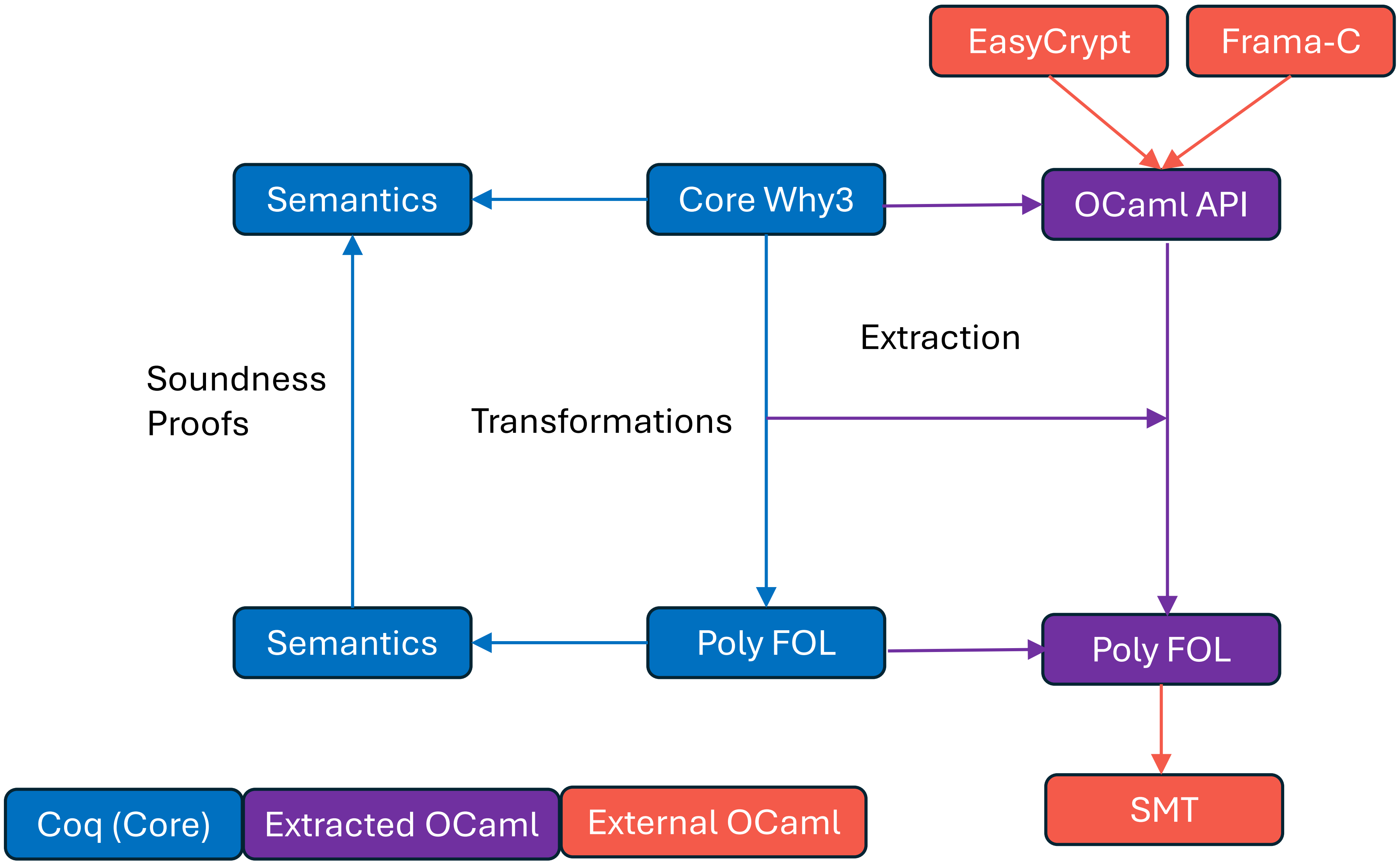
4. Robustness

- Exhaustiveness check succeeds under reasonable changes to types, terms, patterns, etc
  - E.g. substitution, alpha-conversion, rewriting, etc
  - Not true in Why3! Rewriting can cause exhaustiveness check to fail
  - Found and reported bug to Why3 developers
  - Fixed with provably stronger exhaustiveness check

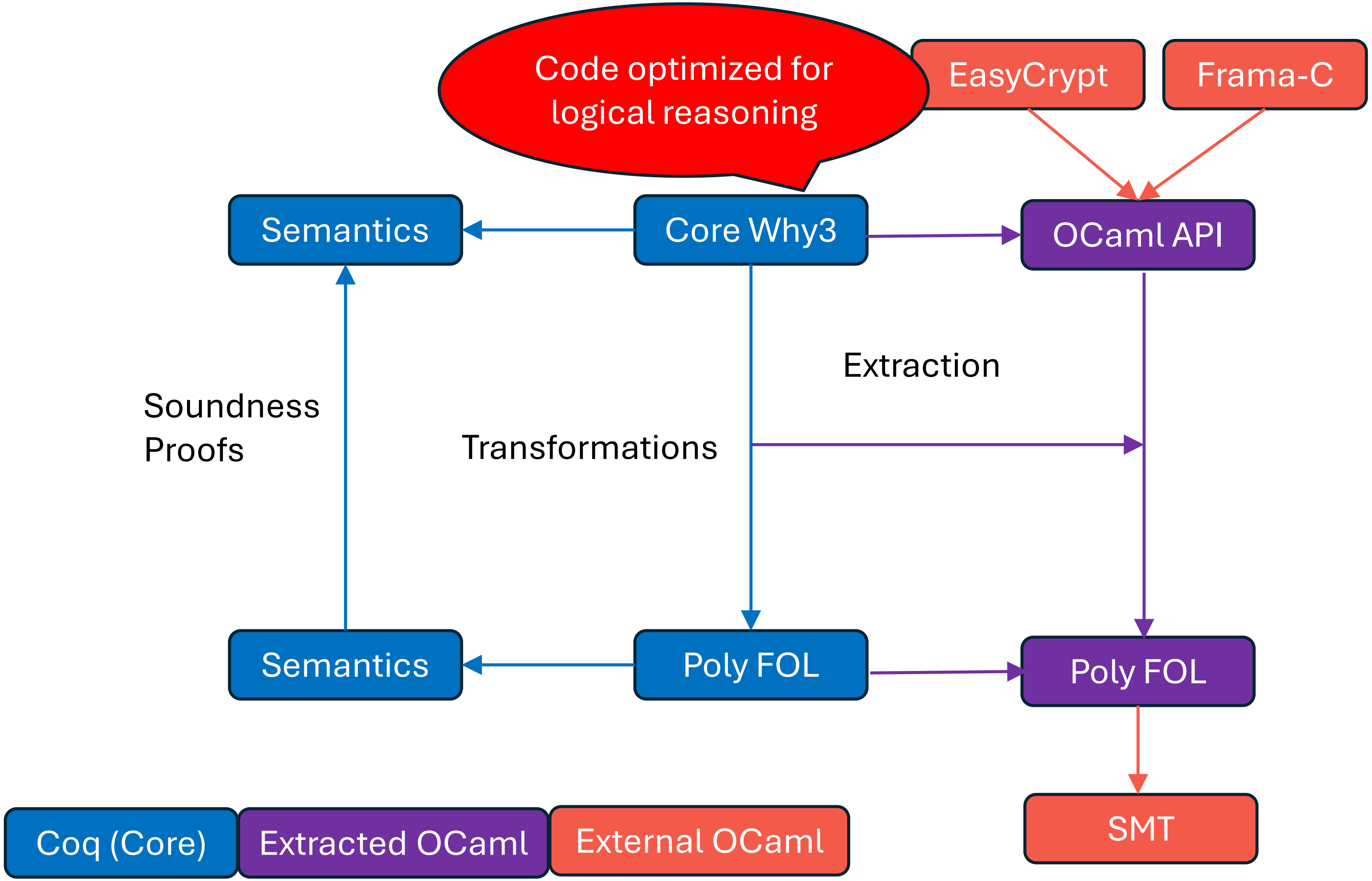
Thesis Contributions

1. A novel formal semantics for Why3’s logic
2. A proved-sound compiler from Why3 to (polymorphic) FOL, including
   - Pattern matching compilation
   - Algebraic Data Type axiomatization
3. **Why3 API implementation in Coq**
   - Method to implement stateful OCaml APIs in Coq
   - Resulting implementation executable both in Coq and OCaml

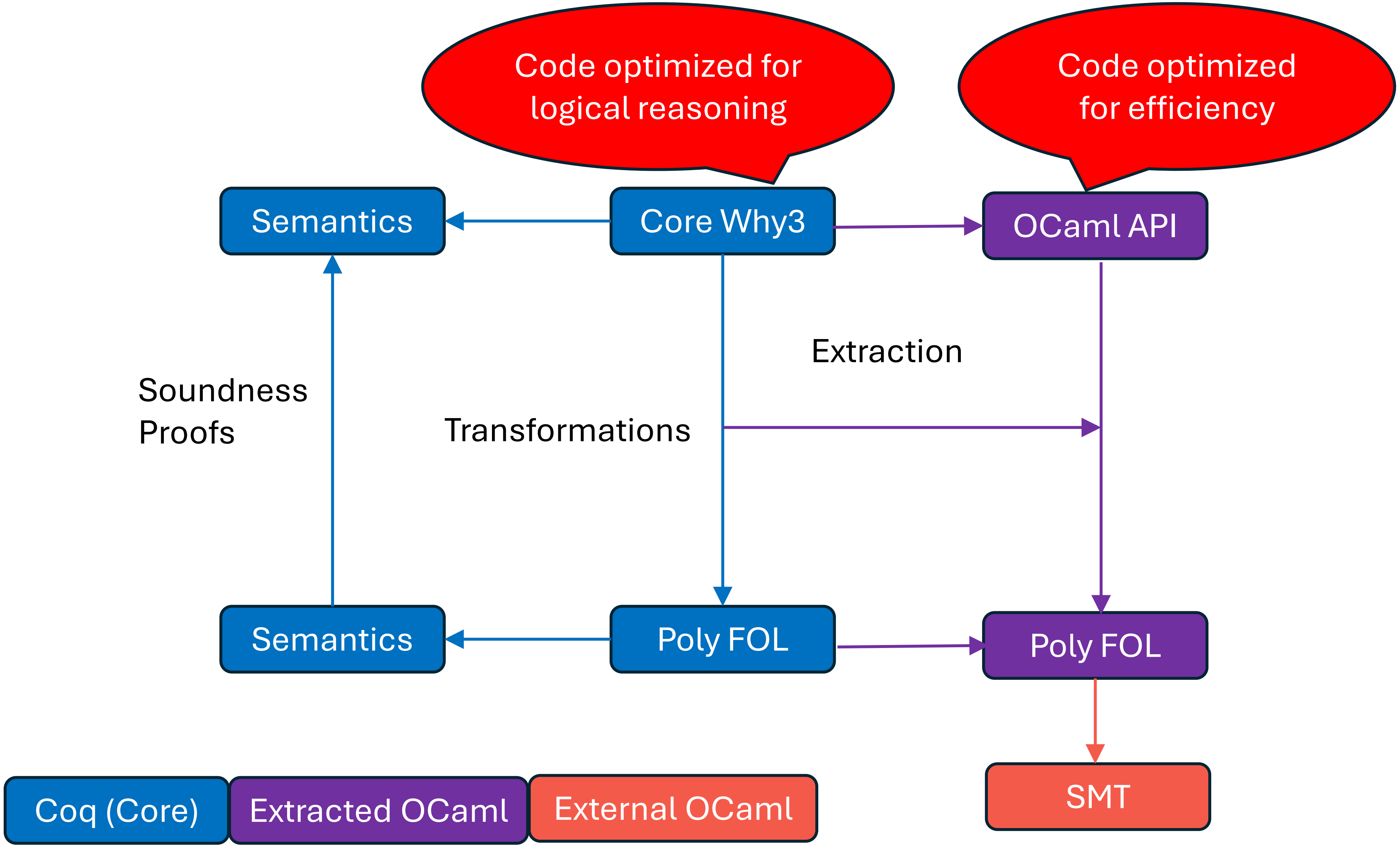
Foundational Why3



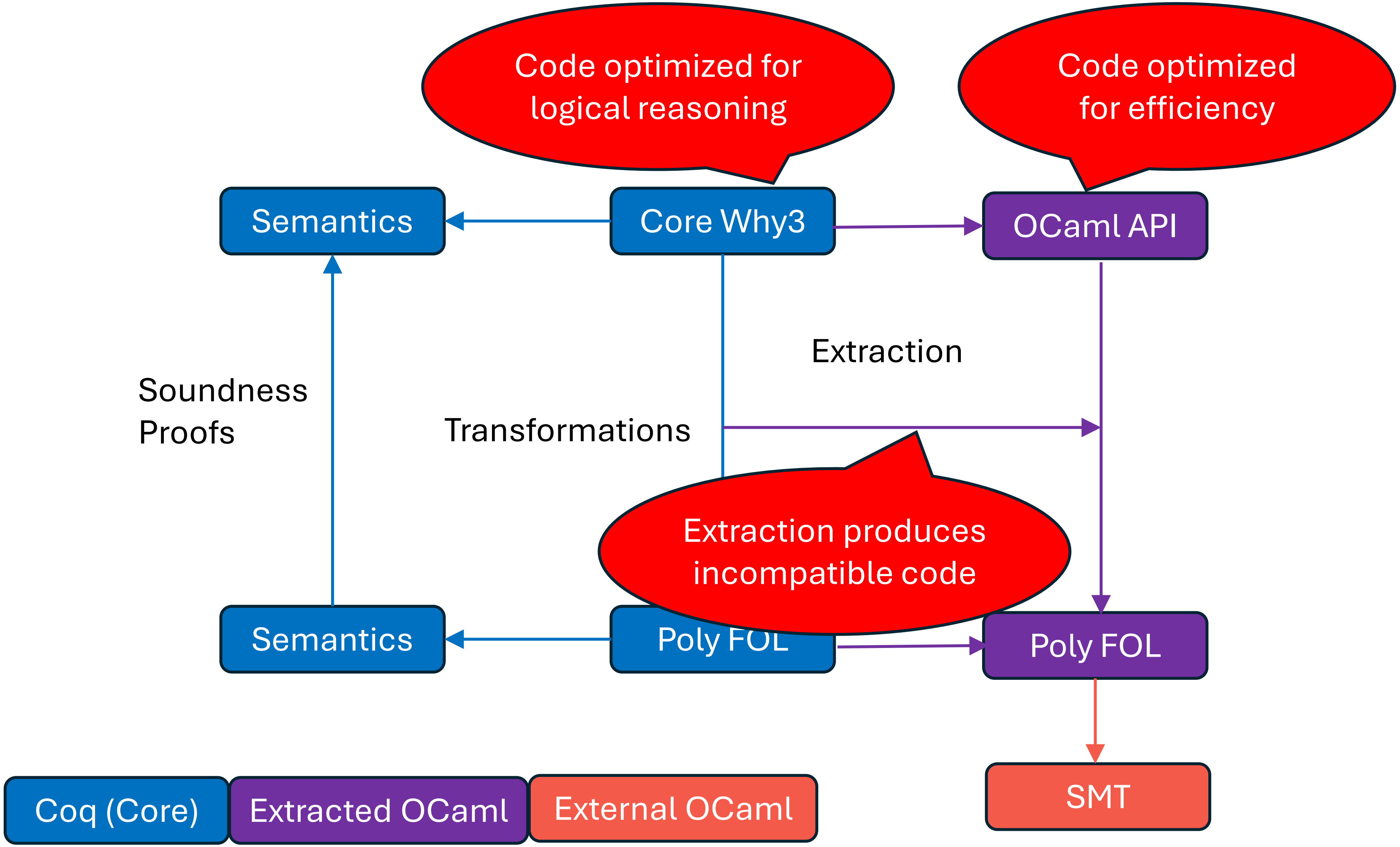
Foundational Why3



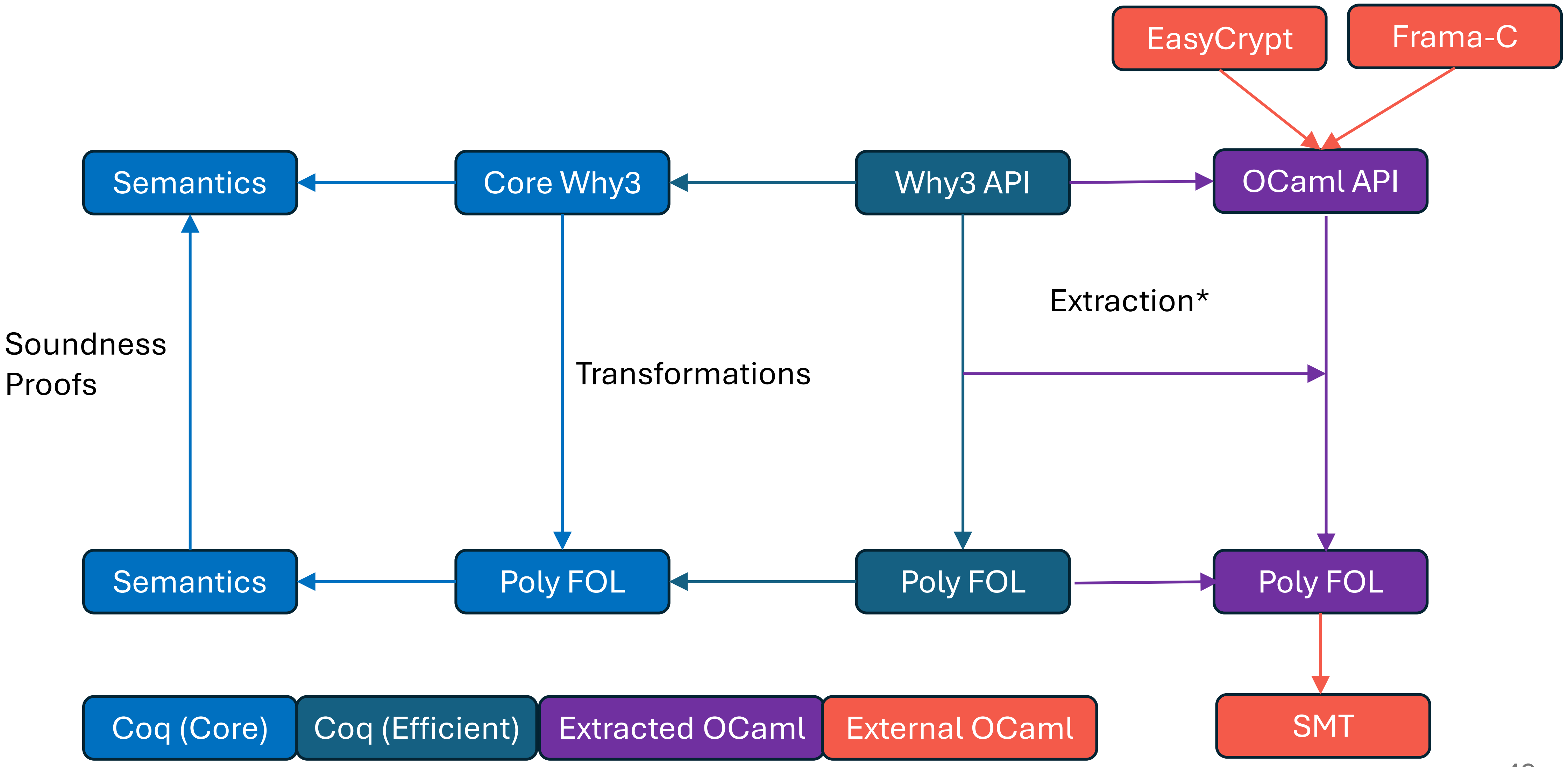
Foundational Why3



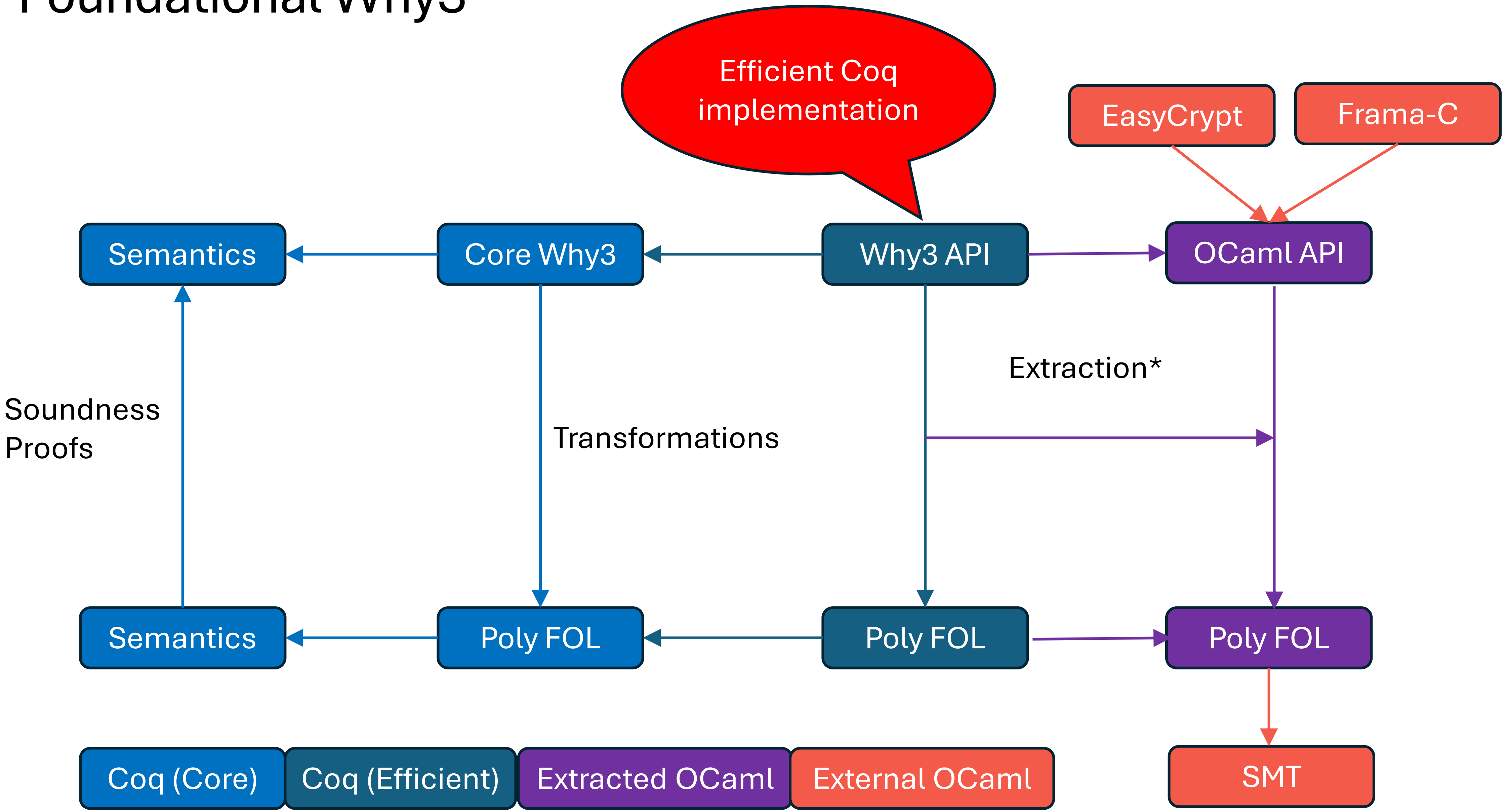
Foundational Why3



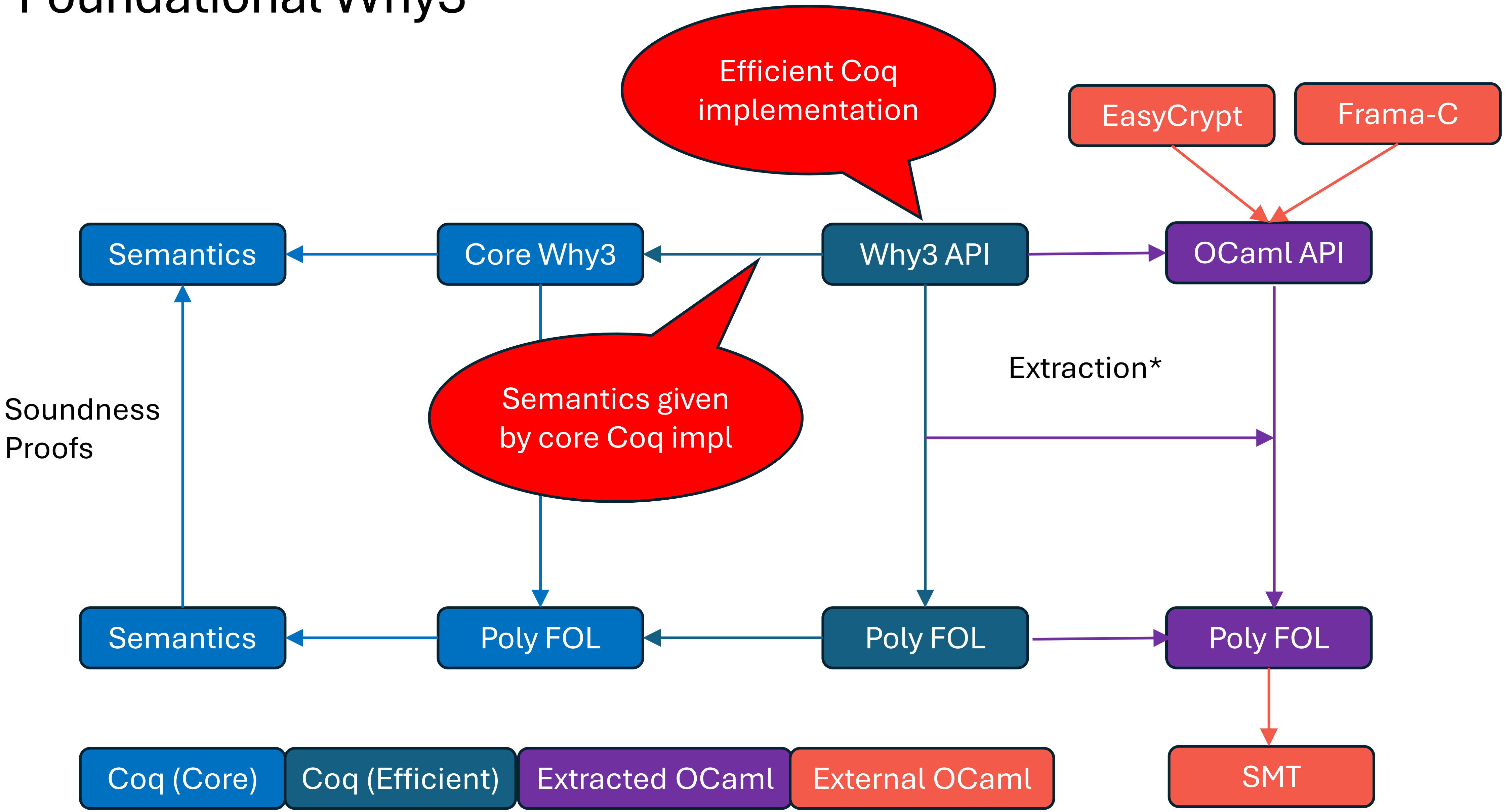
Foundational Why3



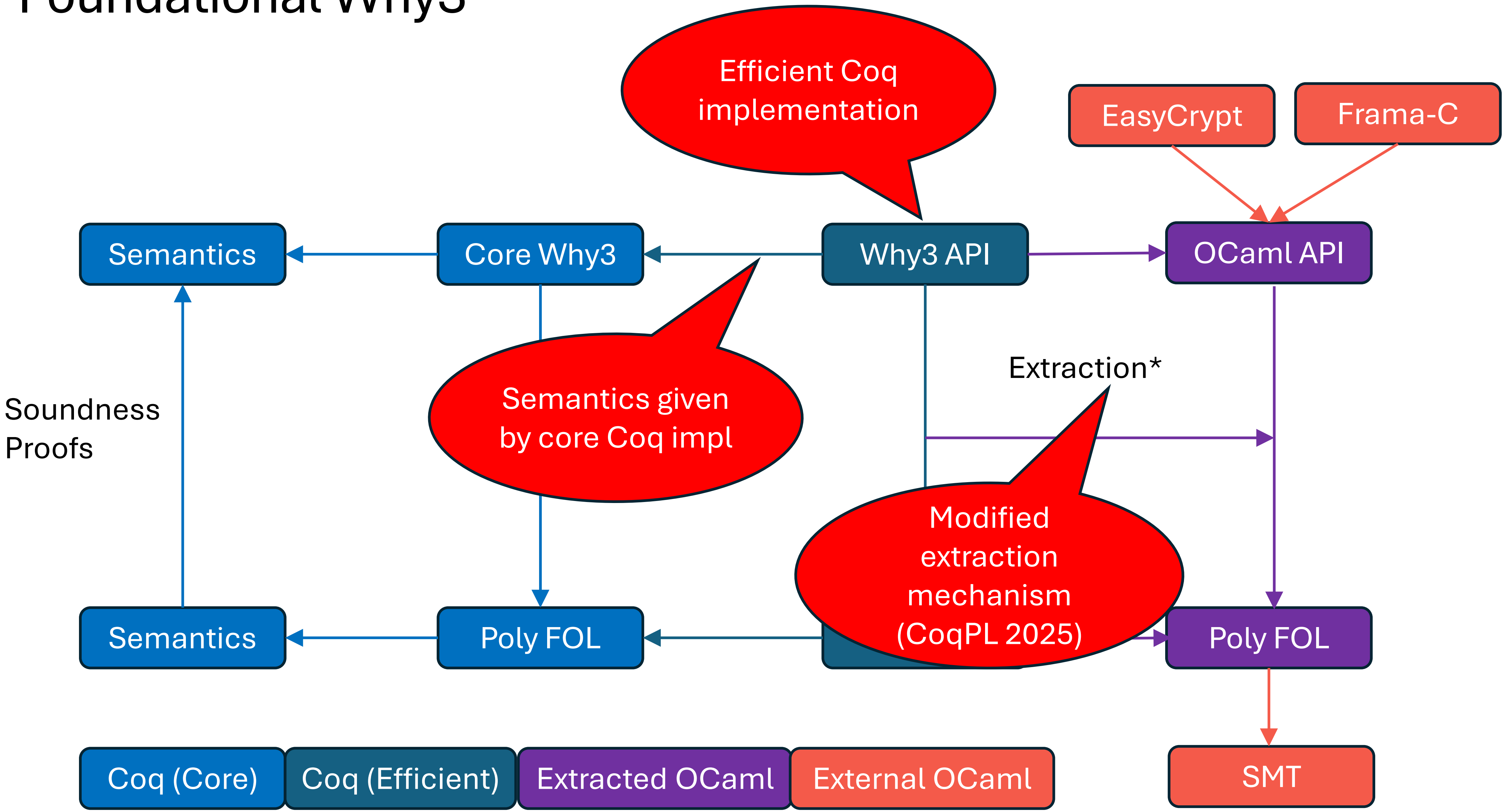
Foundational Why3



Foundational Why3



Foundational Why3



Implementing OCaml APIs in Coq - Example

```
val hd : 'a list -> 'a

let hd = function
    [] -> failwith "hd"
  | a::_ -> a
```

Similar ideas: [Nanevski et al. 2008] [Abrahamsson et al. 2020] [Sakaguchi 2020] [Boulmé 2021] [Korkut et al. 2025]

Implementing OCaml APIs in Coq - Example

```
val hd : 'a list -> 'a

let hd = function
    [] -> failwith "hd"
  | a::_ -> a
```

```
Definition hd {A: Type} (l: list A) : errorM A :=
    match l with
    | [] => throw (Failure "hd")
    | x :: _ => err_ret x
    end.
```

```
let hd = function
| [] -> raise (Failure "hd")
| x :: _ -> x
```

Similar ideas: [Nanevski et al. 2008] [Abrahamsson et al. 2020] [Sakaguchi 2020] [Boulmé 2021] [Korkut et al. 2025]

Implementing OCaml APIs in Coq - Example

```
val hd : 'a list -> 'a

let hd = function
    [] -> failwith "hd"
  | a::_ -> a
```

Implement in Coq in error monad

```
Definition hd {A: Type} (l: list A) : errorM A :=
    match l with
    | [] => throw (Failure "hd")
    | x :: _ => err_ret x
    end.
```

```
let hd = function
| [] -> raise (Failure "hd")
| x :: _ -> x
```

Similar ideas: [Nanevski et al. 2008] [Abrahamsson et al. 2020] [Sakaguchi 2020] [Boulmé 2021] [Korkut et al. 2025]

Implementing OCaml APIs in Coq - Example

```
val hd : 'a list -> 'a

let hd = function
    [] -> failwith "hd"
  | a::_ -> a
```

Implement in Coq in error monad

```
Definition hd {A: Type} (l: list A) : errorM A :=
    match l with
    | [] => throw (Failure "hd")
    | x :: _ => err_ret x
    end.
```

Modified extraction replaces monad with exceptions

```
let hd = function
| [] -> raise (Failure "hd")
| x :: _ -> x
```

Similar ideas: [Nanevski et al. 2008] [Abrahamsson et al. 2020] [Sakaguchi 2020] [Boulmé 2021] [Korkut et al. 2025]

Implementing OCaml APIs in Coq - Example

```
val hd : 'a list -> 'a

let hd = function
    [] -> failwith "hd"
  | a::_ -> a
```

Implement in Coq in error monad

```
Definition hd {A: Type} (l: list A) : errorM A :=
    match l with
    | [] => throw (Failure "hd")
    | x :: _ => err_ret x
    end.
```

Modified extraction replaces monad with exceptions

Identical type and behavior as original!

```
let hd = function
| [] -> raise (Failure "hd")
| x :: _ -> x
```

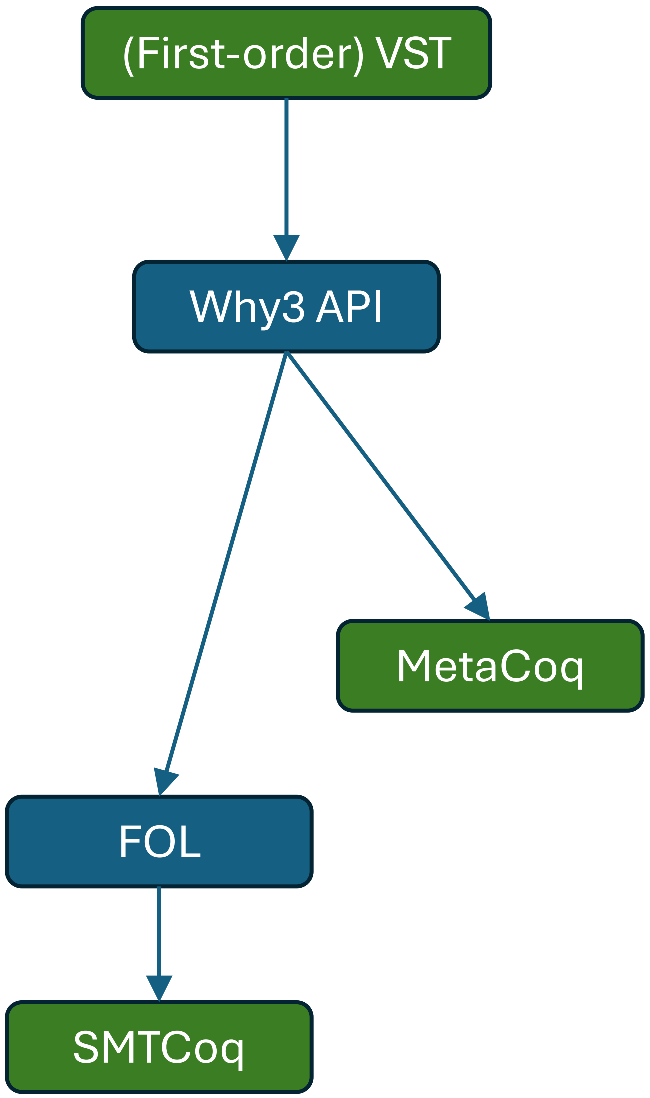
Similar ideas: [Nanevski et al. 2008] [Abrahamsson et al. 2020] [Sakaguchi 2020] [Boulmé 2021] [Korkut et al. 2025]

A New Why3 Implementation

- Handles large subset of Why3, not everything
  - Missing: lexicographic termination, function types, nested + nonuniform ADTs, interfacing with pure WhyML code
- Run Why3 and EasyCrypt test suites against Foundational Why3
- EasyCrypt good test: uses lots of features for real-world reasoning
- Pass all 183 EasyCrypt tests, on average 1.8x slowdown (~8 min vs 15 min), still practical!
- Main performance bottlenecks: functional hash tables, eager substitution, arbitrary-length integers
- Found several bugs in Why3

Conclusion

- Gave first:
  - Verified real-world IVL implementation
  - IVL implemented in a proof assistant
  - Formal semantics and proofs of soundness for recursive structures
- Future work:
  - Verify rest of (simpler) transformations for real-world use
  - Connect to other tools in front-end and back-end
- Implementation and proofs available at https://github.com/joscoh/why3-semantics
- Thanks for listening!



References

Abrahamsson, O., Ho, S., Kanabar, H., Kumar, R., Myreen, M. O., Norrish, M. and Tan, Y. K. Proof Producing Synthesis of CakeML from Monadic HOL Functions. Journal of Automated Reasoning 64.7 (Oct 2020), 1287-1306.

Alain Laville. Implementation of lazy pattern matching algorithms. In Proceedings of the 2nd European Symposium on Programming, ESOP ’88, page 298–316, Berlin, Heidelberg, 1988. Springer-Verlag. ISBN 3540190279.

Besson, F. Itauto: An Extensible Intuitionistic SAT Solver. In 12th International Conference on Interactive Theorem Proving (ITP 2021) (Wroc law, Poland, 2021), L. Cohen and C. Kaliszyk, Eds., vol. 193 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 9:1–9:18.

Blanchette, J. C., Böhme, S., and Paulson, L. C. Extending Sledgehammer with SMT Solvers. In Automated Deduction – CADE-23 (Wroc law, Poland, 2011), N. Bjørner and V. Sofronie-Stokkermans, Eds., Springer, pp. 116–130.

Blot, V., Cousineau, D., Crance, E., de Prisque, L. D., Keller, C., Mahboubi, A., and Vial, P. Compositional Pre-processing for Automated Reasoning in Dependent Type Theory. In Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (Boston, USA, Jan. 2023), CPP 2023, Association for Computing Machinery, pp. 63–77

Boulmé, S. Formally Verified Defensive Programming (Efficient Coq-verified Computations from Untrusted ML Oracles). PhD thesis, Université Grenoble-Alpes, Sept 2021.

Cohen, J. M., and Johnson-Freyd, P. A Formalization of Core Why3 in Coq. Proceedings of the ACM on Programming Languages 8, POPL (Jan. 2024), 60:1789–60:1818.

Czajka, L., and Kaliszyk, C. Hammer for Coq: Automation for Dependent Type Theory. Journal of Automated Reasoning 61, 1 (June 2018), 423–453.

Dardinier, T., Sammler, M., Parthasarathy, G., Summers, A. J., and Müller, P. Formal Foundations for Translational Separation Logic Verifiers. Proceedings of the ACM on Programming Languages 9, POPL (Jan. 2025), 20:569–20:599.

Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., and Barrett, C. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In Computer Aided Verification (Heidelberg, Germany, 2017), R. Majumdar and V. Kunčak, Eds., Springer International Publishing, pp. 126–133.

References

Filliâtre, J.-C. One Logic to Use Them All. In Automated Deduction – CADE-24 (Lake Placid, New York, 2013), M. P. Bonacina, Ed., Springer, pp. 1–20.

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. Gadts meet their match: pattern-matching warnings that account for gadts, guards, and laziness. SIGPLAN Not., 50(9):424–436, aug 2015. ISSN 0362-1340. doi: 10.1145/2858949.2784748.

Gäher, L., Sammler, M., Jung, R., Krebbers, R., and Dreyer, D. RefinedRust: A Type System for High-Assurance Verification of Rust Programs. Proceedings of the ACM on Programming Languages 8, PLDI (June 2024), 192:1115–192:1139.

Jacobs, B., Vogels, F., and Piessens, F. Featherweight VeriFast. Logical Methods in Computer Science Volume 11, Issue 3 (Sept. 2015).

Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical report, University of Virginia, USA, 2000.

Korkut, J., Stark, K., and Appel A. W. A Verified Foreign Function Interface Between Coq and C. Proceedings of the ACM on Programming Languages 9, POPL (Jan 2025), 24:687-24:717.

Laurence Puel and Ascander Suarez. Compiling pattern matching by term decomposition. In Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP '90, page 273–281, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 089791368X. doi: 10.1145/91556.91670.

Le Fessant, F., and Maranget, L. Optimizing pattern matching. In Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (Florence, Italy, Oct. 2001), Association for Computing Machinery, pp. 26–37.

Lennart Augustsson. Compiling pattern matching. In Proc. of a Conference on Functional Programming Languages and Computer Architecture, page 368–381, Berlin, Heidelberg, 1985. Springer-Verlag. ISBN 3387159754.

Luc Maranget. Compiling lazy pattern matching. SIGPLAN Lisp Pointers, V(1):21–31, jan 1992. ISSN 1045-3563. doi: 10.1145/141478.141499.

References

Maranget, L. Compiling pattern matching to good decision trees. In Proceedings of the 2008 ACM SIGPLAN Workshop on ML (Victoria, British Columbia, Canada, Sept. 2008), ML ’08, Association for Computing Machinery, pp. 35–46.

Maranget, L. Warnings for pattern matching. Journal of Functional Programming 17, 3 (May 2007), 387–421.

Marianne Baudinet and David B. MacQueen. Tree pattern matching for ml., 1985. URL http://www.smlnj.org/compiler-notes/85-note-baudinet.ps.

Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In Proceedings of the 4th International Conference on Compiler Construction, CC ’92, page 258–270, Berlin, Heidelberg, 1992. Springer-Verlag. ISBN 3540559841.

Nanevski,A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. Ynot: Dependent types for imperative programs. In Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (Victoria, British Columbia, Canada, Sept 2008). ICFP '08, Association for Computing Machinery, pp 229-240.

Parthasarathy, G., Dardinier, T., Bonneau, B., Müller, P., and Summers, A. J. Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language. Proceedings of the ACM on Programming Languages 8, PLDI (June 2024), 208:1510–208:1534.

Parthasarathy, G., Müller, P., and Summers, A. J. Formally Validating a Practical Verification Condition Generator. In Computer Aided Verification (2021), A. Silva and K. R. M. Leino, Eds., Springer International Publishing, pp. 704–727.

Peter Sestoft. Ml pattern match compilation and partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, Partial Evaluation, pages 446–464, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70589-5.

R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. SIAM J. Comput., 24(6):1207–1234, dec 1995. ISSN 0097-5397. doi: 10.1137/S0097539793246252.

Sakaguchi, K. Program extraction for mutable arrays. Science of Computer Programming 191 (June 2020), 102372.

References

Sammler, M., Lepigre, R., Krebbers, R., Memarian, K., Dreyer, D., and Garg, D. RefinedC: Automating the foundational verification of C code with refined ownership types. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (June 2021), Association for Computing Machinery, pp. 158–174.

Sozeau, M., Forster, Y., Lennon-Bertrand, M., Nielsen, J., Tabareau, N., and Winterhalter, T. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. Journal of the ACM 72, 1 (Jan. 2025), 8:1–8:74.

Thomas Tuerk, Magnus O. Myreen, and Ramana Kumar. Pattern matches in HOL:. In Christian Urban and Xingyuan Zhang, editors, Interactive Theorem Proving, pages 453–468, Cham, 2015. Springer International Publishing. ISBN 978-3-319-22102-1.

Zhou, L., Qin, J., Wang, Q., Appel, A. W., and Cao, Q. VST-A: A Foundationally Sound Annotation Verifier. Proceedings of the ACM on Programming Languages 8, POPL (Jan. 2024), 69:2069–69:2098.