Programming and proving with FoCaLiZe:

from computer algebra to proofs interoperability

Catherine Dubois, **François Pessaux**

joined work with the **FoCaLiZe development team**
Sylvain Boulmé, Matthieu Carlier, Raphaël Cauderlier, David Delahaye, Damien Doligez, Catherine Dubois, Jean-Frédéric Etienne, Stéphane Fechter, Pierre Halmagrand, **Thérèse Hardin**, Mathieu Jaume, **Renaud Rioboo**, Virgile Prevosto, ...

(Slides were gently stolen from Catherine Dubois ☺)

FoCaLiZe: in a nutshell

- Environment for stepwise and incremental development of formally verified programs
- First-order specifications
- ML as an implementation language
- Automated proofs: proofs found from user hints by Zenon (resp. Zenon modulo) and checked by Coq (resp. Dedukti)
- Modularity / Reuse (multiple inheritance and parametrization)
- Compilation to OCaml, Coq, Dedukti

`http://focalize.inria.fr/`

Some *historical* points

1999 start of the project with the following objective (Hardin, Rioboo): develop an environment for certified computer algebra (or effective mathematics) relying on OCaml (for execution) et Coq (for verification) while avoiding the use of some complex features of these languages

2003 first version FOC (for Formal, OCaml, Coq) (Prevosto Phd)

2004 Integration of Zenon for doing proofs (translated and verified by Coq)

2006 Certifying Airport Security Regulations using the Focal Environment (Edemoi project)
$\longrightarrow$ general-purpose environment

2008 Property-based testing: FocalTest

2009 Complete redesign of the compiler: FoCaLiZe

2015 New backend for Dedukti: proofs done by Zenon can be verified by Dedukti

2016 FoCaliZe to the rescue for proof interoperability - MathTransfer

Roadmap of the talk

1. Presentation of FoCaLiZe
   - Design principles
   - Ground concepts
   - Examples
2. Inside the compiler
3. FoCaLiZe as a platform for proofs interoperability

Design principles/Requirements

- Structuration, abstraction and reusability
  $\longrightarrow$ inheritance, early-binding, redefinition, parametrization, encapsulation
- Development process based on refinement
  $\longrightarrow$ signatures/definitions, properties/theorems
- Clean and confident semantic approach
  $\longrightarrow$ functional approach, types, module based, dependencies calculus, proof verification
- First order like approach for simplicity (with a back-door to more if necessary)

Species

- Species contain *methods* acting on the *representation* type of the species
- The representation is written `Self` inside the species
- Methods are either *computational* or *logical*
- Methods are either *declared* or *defined*
  - Declared computational = signature
  - Defined computational = function
  - Declared logical = property
  - Defined logical = theorem

Method declaration/definition: Setoids (1)

```
species Setoid =
  signature equal:  Self -> Self -> bool;

  property equal_reflexive: all x: Self, equal(x,x);
  property equal_symmetric: all x y: Self,
    equal(x,y) -> equal(y,x);
  property equal_transitive: all x y z: Self,
    equal(x,y) -> equal(y,z) -> equal(x,z);

  let diff (x,y) = ~~(equal(x,y));

  theorem eq_diff: all x y: Self, ~equal(x,y) <-> diff(x,y)
    proof = by definition of diff;
  theorem diff_irreflexive: all x: Self, ~(diff (x, x))
    proof = by property equal_reflexive, eq_diff;
  ...
end;;
```

Modularity

- Species are extended by (multiple) inheritance
- During inheritance, we can
  - add new methods
  - (re) define declared methods
  - define the representation

Develop with FoCaLiZe = build a hierarchy of species

Inheritance: partial and total orders

```
species Partial_order = inherit Setoid;
  signature leq:  Self -> Self -> bool;
  property leq_equal: all x y: Self, equal(x,y) -> leq(x,y)
  theorem leq_reflexive: all x: Self, leq (x, x)
    proof = by property leq_equal, equal_reflexive;
  ...
end;;

species Total_order = inherit Partial_order;
  property leq_total: all x y: Self, leq(x,y) || leq(y,x);
  let equal(x,y) = leq(x,y) && leq(y,x);
  proof of equal_reflexive =
    by definition of equal property leq_total;
  ...
end;;
```

Complete species and collection

Complete species: every signature (resp. property) completed with a definition (resp. proof)

```
species I_integers = inherit Setoid;
  representation = int;
  let equal =  ( = );
  proof of equal_transitive  = assumed;
  proof of equal_symmetric  = assumed;
  proof of equal_reflexive  = assumed;
end;;

collection Integers = implement I_integers; end;;
```

Collection:

- abstract data type/encapsulation
- built from a complete species
- used as parameters of species

Parametrized species: Cartesian product of setoids

```
species Cartesian (S1 is Setoid, S2 is Setoid) =
  inherit Setoid ;
  representation = S1 * S2 ;

  let equal (x, y) = S1!equal (fst(x), fst(y)) &&
                     S2!equal (snd(x), snd(y));

  proof of equal_reflexive =
    by property S1!equal_reflexive, S2!equal_reflexive
       definition of equal;
  ...
end;;
```

Focus on properties and proofs

Proofs independent from any particular proof checker
$\longrightarrow$ FoCaLiZe declarative proof language (inspired from Lamport's style)

Proof

- hierarchical decomposition into intermediate steps introducing hypotheses and subgoals
- leaf: subgoal which can be automatically handled by Zenon (first order automated prover) using facts (by definition/property/type) given by the programmer

Focus on properties and proofs

```
theorem zero_is_unique: all o : Self,
   (all x: Self, equal(x, plus(x, o))) -> equal(o, zero)
proof =
<1>1 assume o: Self,
     hypothesis H1: all x: Self, equal(x, plus (x, o)),
     prove equal(o, zero)
     <2>1 prove equal(zero, plus (zero, o))
        by hypothesis H1
     <2>2 prove equal(o, zero)
        by step <2>1
        property zero_is_neutral, equal_transitive,
        equal_symmetric
     <2>f qed by step <2>2
<1>f conclude ;
```

A shorter proof :

```
by property zero_is_neutral, equal_transitive, equal_symmetric
```

Focus on properties and proofs

Each subgoal is sent to Zenon that has to automatically find a proof.
The problem sent to Zenon:

- conclusion = goal of the corresponding step
- facts = hypotheses, properties, steps, definitions (unfolded by Zenon) used in the corresponding step

4 answers from Zenon: *proof found*, *out of memory*, *time out*, *no proof found*

Then Zenon returns a term (in the language of the proof checker) plugged in the context by the compiler.

When out of scope of Zenon, you may open the back door and write proofs in Coq or Dedukti.
$\longrightarrow$ need to be aware of the compilation process of FoCaLiZe ☹

And more features

Inductive type definitions and pattern matching *à la ML*, recursion (`let rec` and `let rec ... and`)

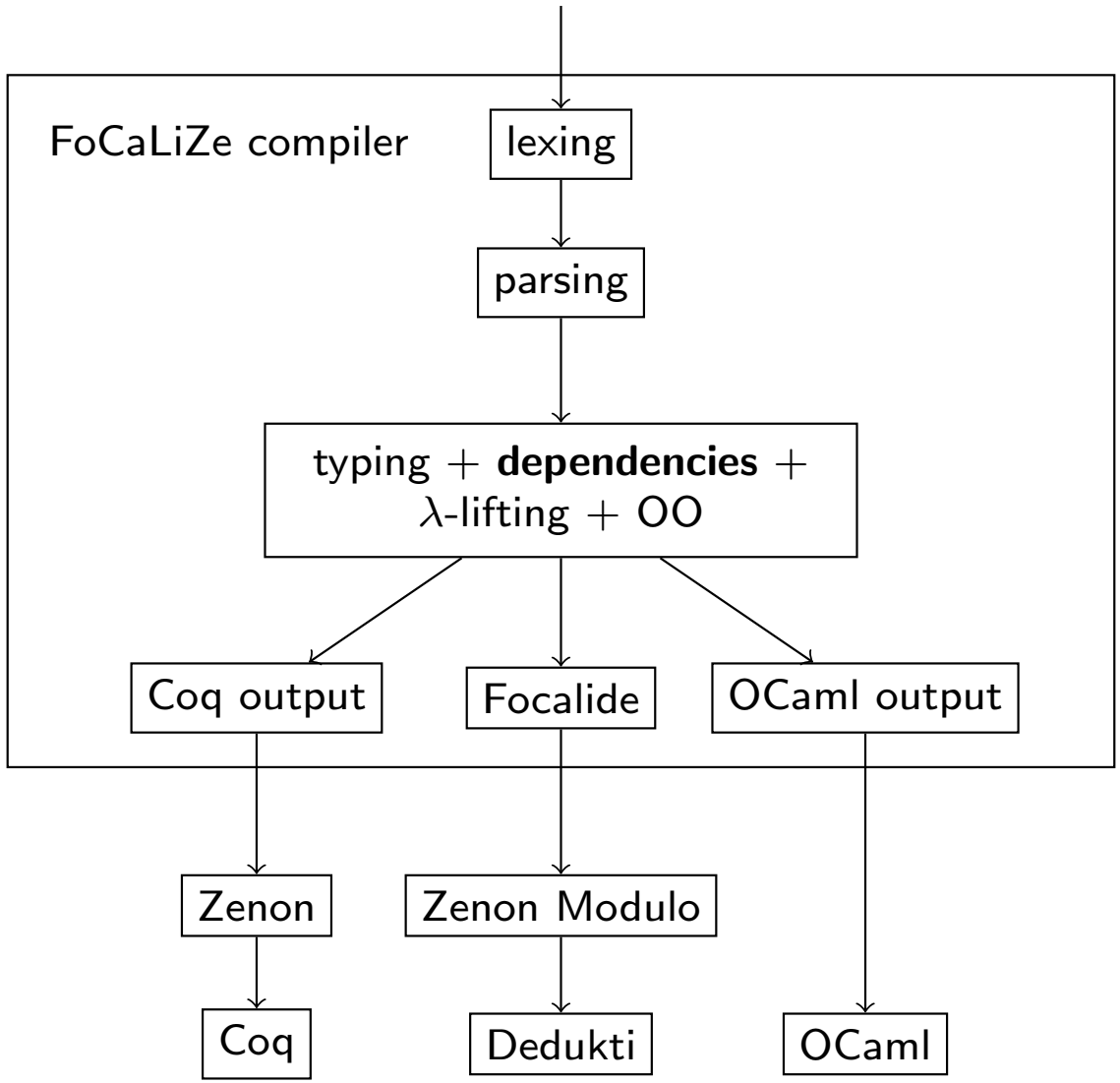
Termination proof (`structural, order`), induction, `by type`

Global definitions, expressions and theorems (at top-level)

External definitions

```
let ( + ) =
  internal int -> int -> int
  external
  | caml -> {* Ml_builtins.bi__int_plus *}
  | coq -> {* coq_builtins.bi__int_plus *}
  | dedukti -> {* dk_int.plus *}
;;
```

Overview of the compilation scheme



Dependencies

A method def-depends on m when it depends on the definition of m.

+ proof with `'by definition of m'`

```
theorem eq_diff: all x y: Self, ~equal(x,y) <-> diff(x,y)
proof = by definition of diff;
```

  → if m redefined, proof must be invalidated (warning emitted)

+ Functions and proofs may def-depends on the representation type.
  - Syntax forbids a function definition to def-depend on a proof
  - By encapsulation no def-dependency on a method of a species parameter
  - Analysis required to prevent def-dependencies on the representation type in properties and theorems statements.

A method decl-depends on m if it depends on the declaration of m

```
let diff (x,y) = ~~(equal(x,y));
```

Dependencies computation

Complex process: tracking them syntactically is not sufficient, process of completion is required

Computation of the *visible universe* of a method m = (minimal) set of methods of Self needed to analyze m

No cyclic dependencies (except for mutual recursive functions)

Computation done on a species in normal form (after processing of inheritance)

Code generation

- After typing, resolution of inheritance and early-binding and dependency analysis
- For traceability, common code generation model OCaml/Coq and Dedukti
- When a method m is defined, generation of its *method generator*
  - compiled version of m's body
  - methods on which m decl-depends are λ-lifted
  - methods on which m def-depends are not λ-lifted (use of their method generator)
  - similar mechanism for the representation
- Method generator shared along inheritance and between collections of a same species
- Code generation for collections create computational runnable code and checkable logical terms.

Code generation for Dedukti

What is Dedukti?

Proof checker, based on the $\lambda\Pi$-calculus modulo = extension of $\lambda\Pi$-calculus, the simplest calculus with dependent types, extended with user-defined rewrite rules on terms and/or propositions.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv_{\beta \mathbf{R}} B}{\Gamma \vdash t : B}\ (\text{Conv})$$

`http://dedukti.gforge.inria.fr/`

Code generation for Dedukti

- Same as for Coq except that Dedukti has no module, no pattern-matching *à la ML*
- Compilation of pattern-matching expressions using destructors
- Compilation of recursive functions using rewrite rules and combinator CBV (for Call By Value) for preserving termination
- Definition of a Dedukti standard library

Roadmap of the talk

1. Presentation of FoCaLiZe
2. Inside the compiler
3. FoCaLiZe as a platform for proofs interoperability
   - **Problem statement and solutions**
   - **MathTransfer**
   - **Combined proof of Eratosthenes Sieve**

Interoperability

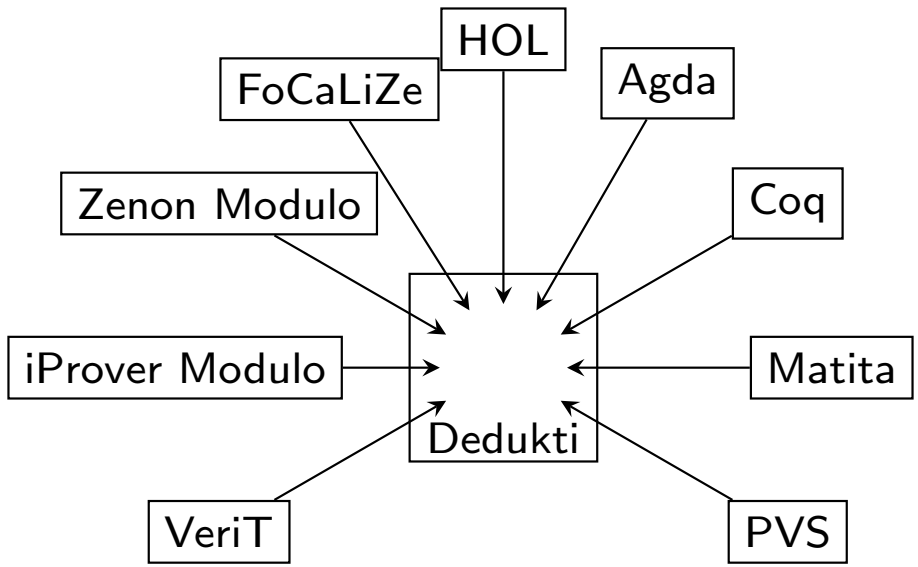
Motivation

- Proof development is *expensive*
  - 4-colors theorem, Kepler conjecture, Feit-Thomson theorem
- Proof assistants are *specialized*
  - Counterexamples, proof by reflection, decision procedures, ...

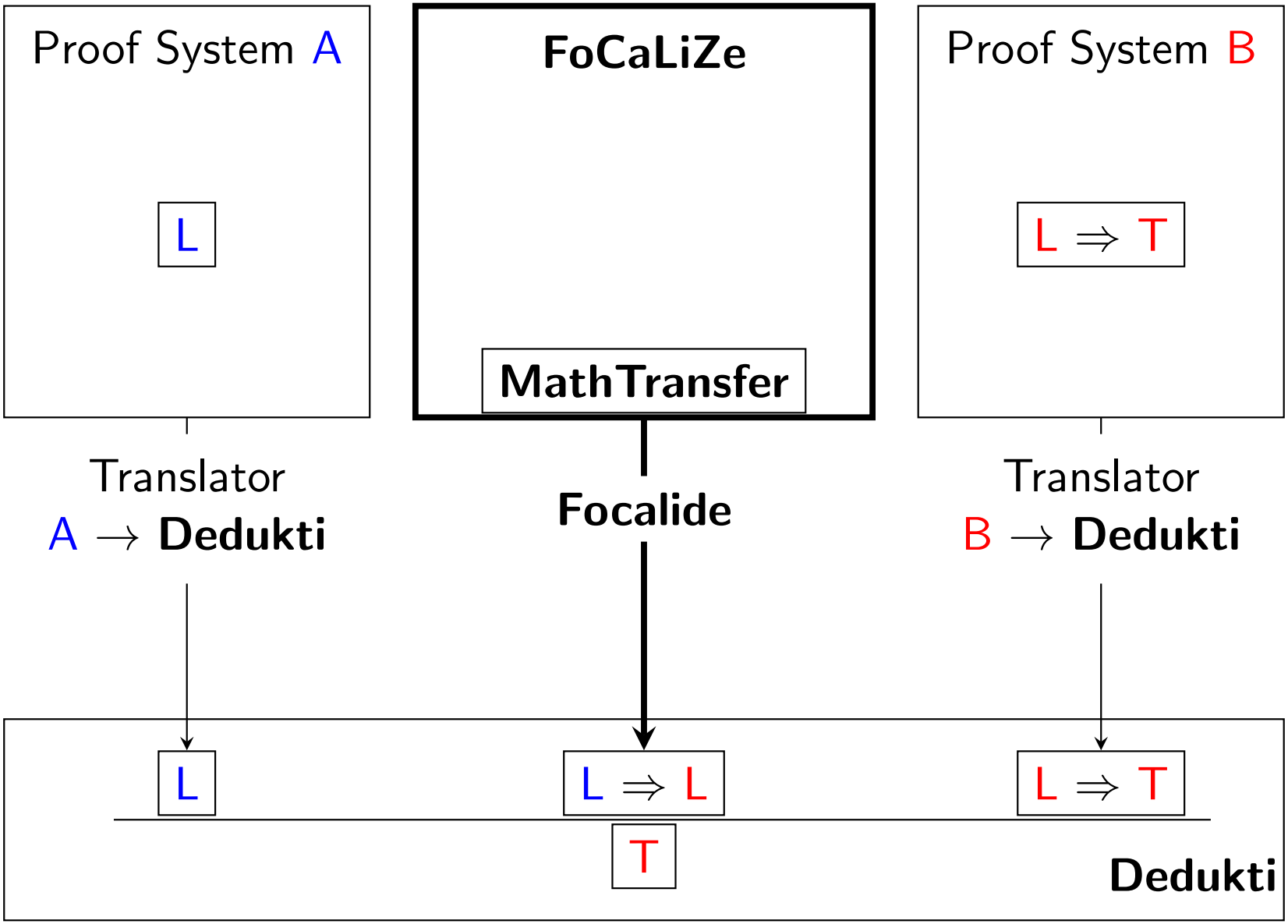
Obstacles

- Logical problem: combine logics of systems A and B in a consistent way.
- Mathematical problem: formulae L and L are not identical
  Theories such as arithmetic are independently defined in systems A and B.
  We need to identify similar concepts.

Our solution: Dedukti & Theorem Transfer



Our solution: Dedukti & Theorem Transfer



The MathTransfer Library

MathTransfer is a FoCaLiZe library of transfer theorems about natural number arithmetic containing :

- mathematical structures (add common operations on naturals),
- morphisms between abstract representations of natural numbers,
- transfer theorems.

Hierarchy of species to instanciate for particular logical systems.

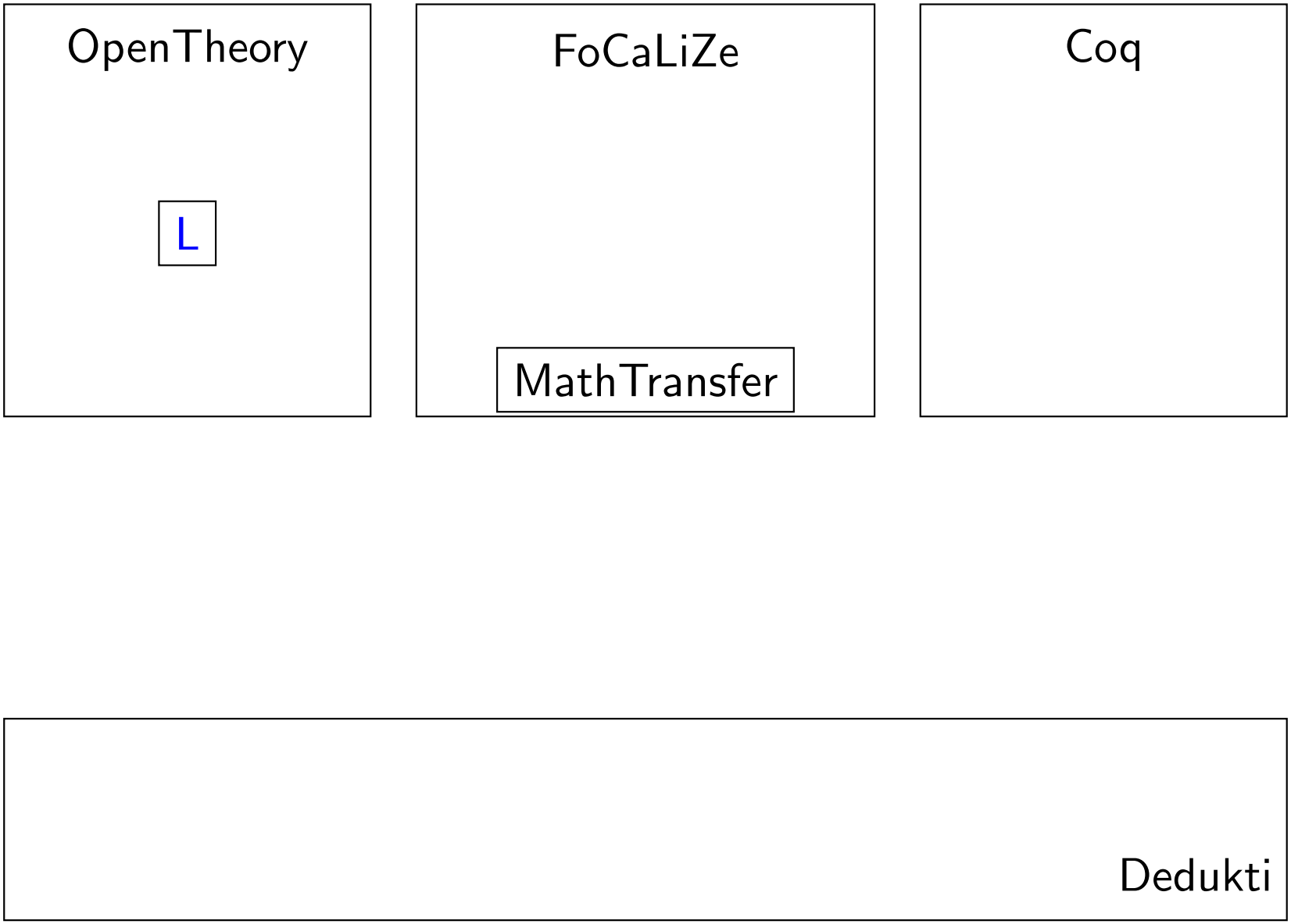
`https://gitlab.math.univ-paris-diderot.fr/cauderlier/math_transfer`

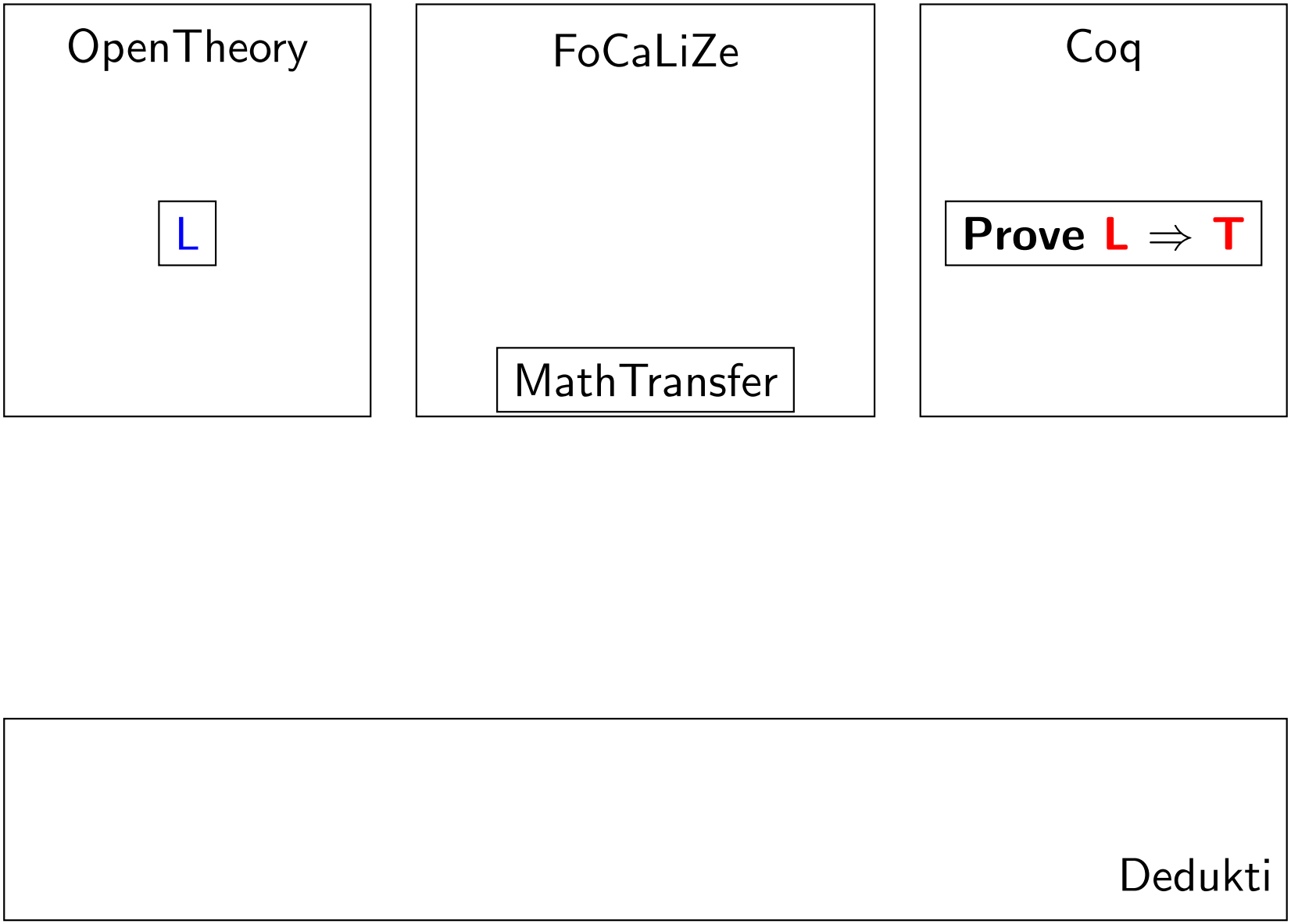
Case Study: a combined proof of correctness of Eratosthenes Sieve

- A = HOL (OpenTheory)
- B = Coq
- T = correctness of the Sieve of Eratosthenes
- L = prime divisor lemma

$$\mathsf{L} := \forall n \neq 1.\ \exists p.\ \mathsf{prime}(p) \wedge p \mid n$$

Already proved in HOL/OpenTheory `natural-prime` library ☺

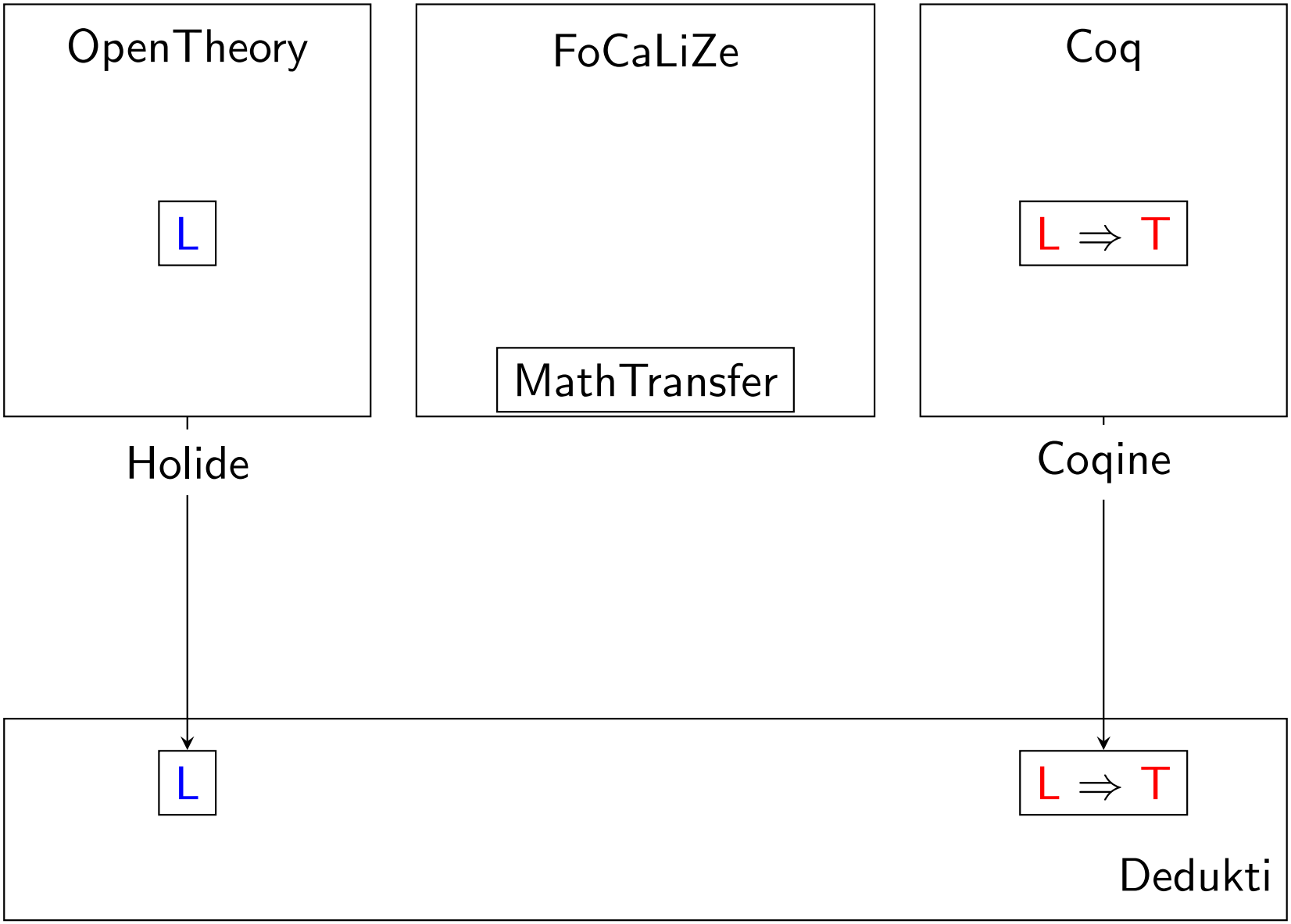

OpenTheory
L
FoCaLiZe
MathTransfer
Coq
Dedukti

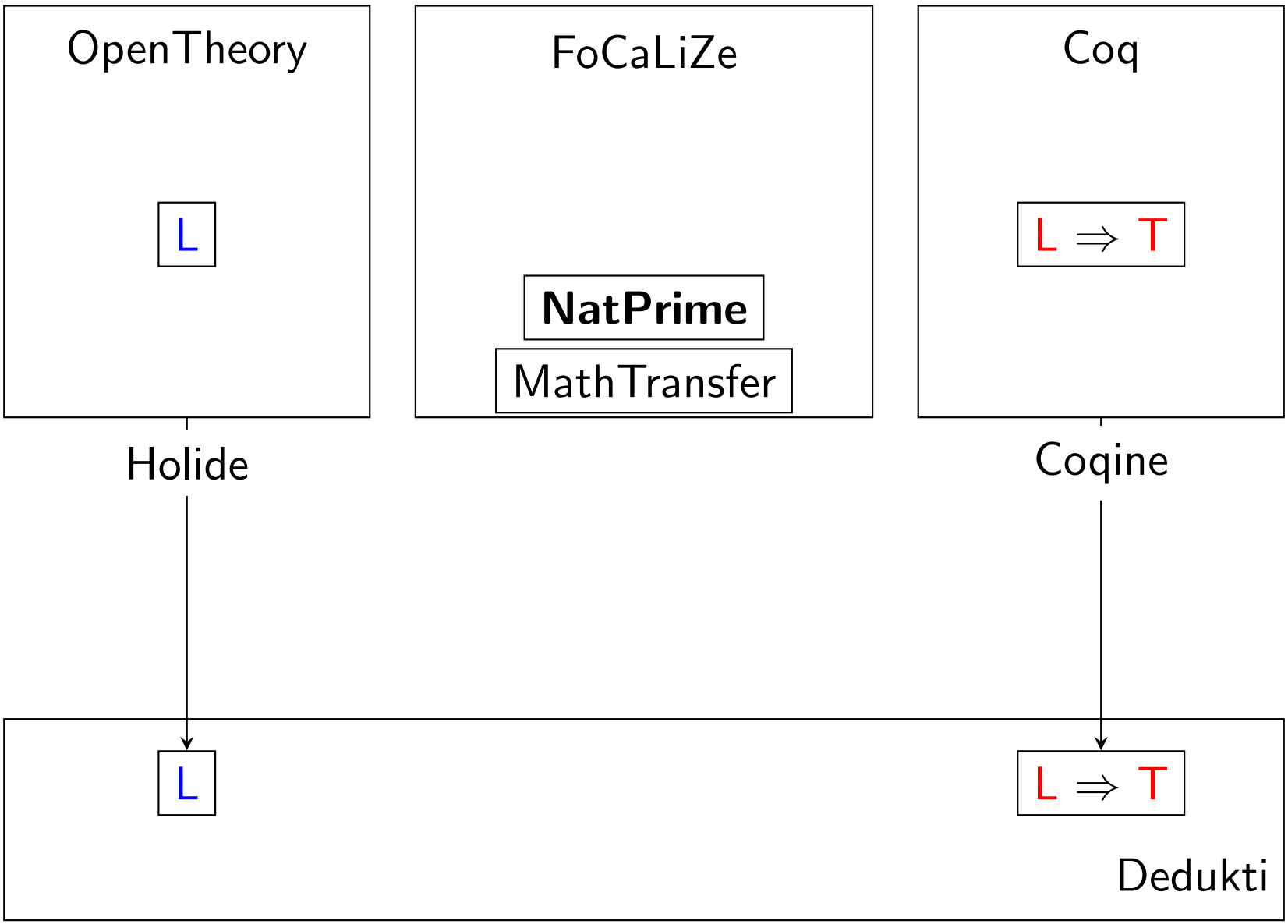

OpenTheory
L
FoCaLiZe
MathTransfer
Coq
Prove L ⇒ T
Dedukti

Prove L → T in Coq

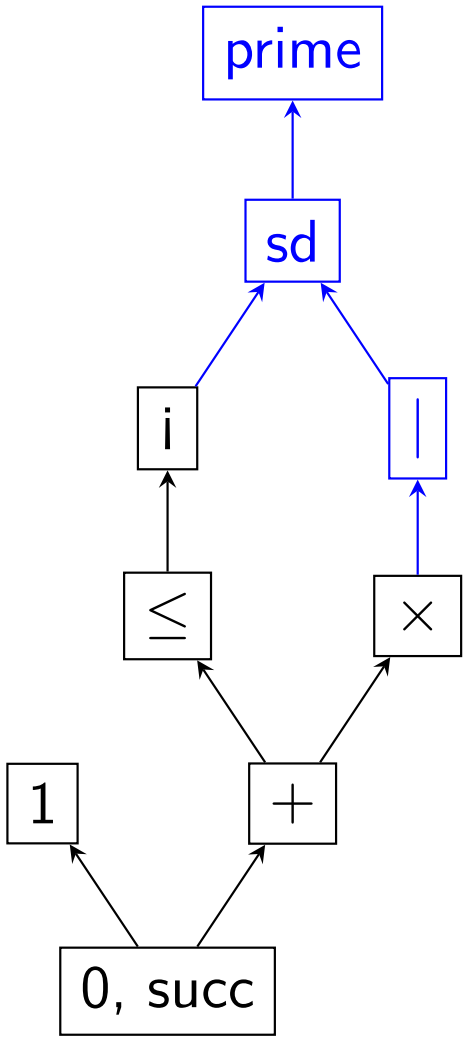
```
Definition eratosthenes n := ...

Section correctness_proof.

Hypothesis prime_divisor :
  forall n : nat, n <> 1 ->
    exists p : nat, prime p /\ divides p n.

Theorem correctness p n :
  In p (eratosthenes n) <-> (p <= 2 + n /\ prime p).
Proof.
...
Qed.

End correctness_proof.
```

≈ 1300 lines of Coq code

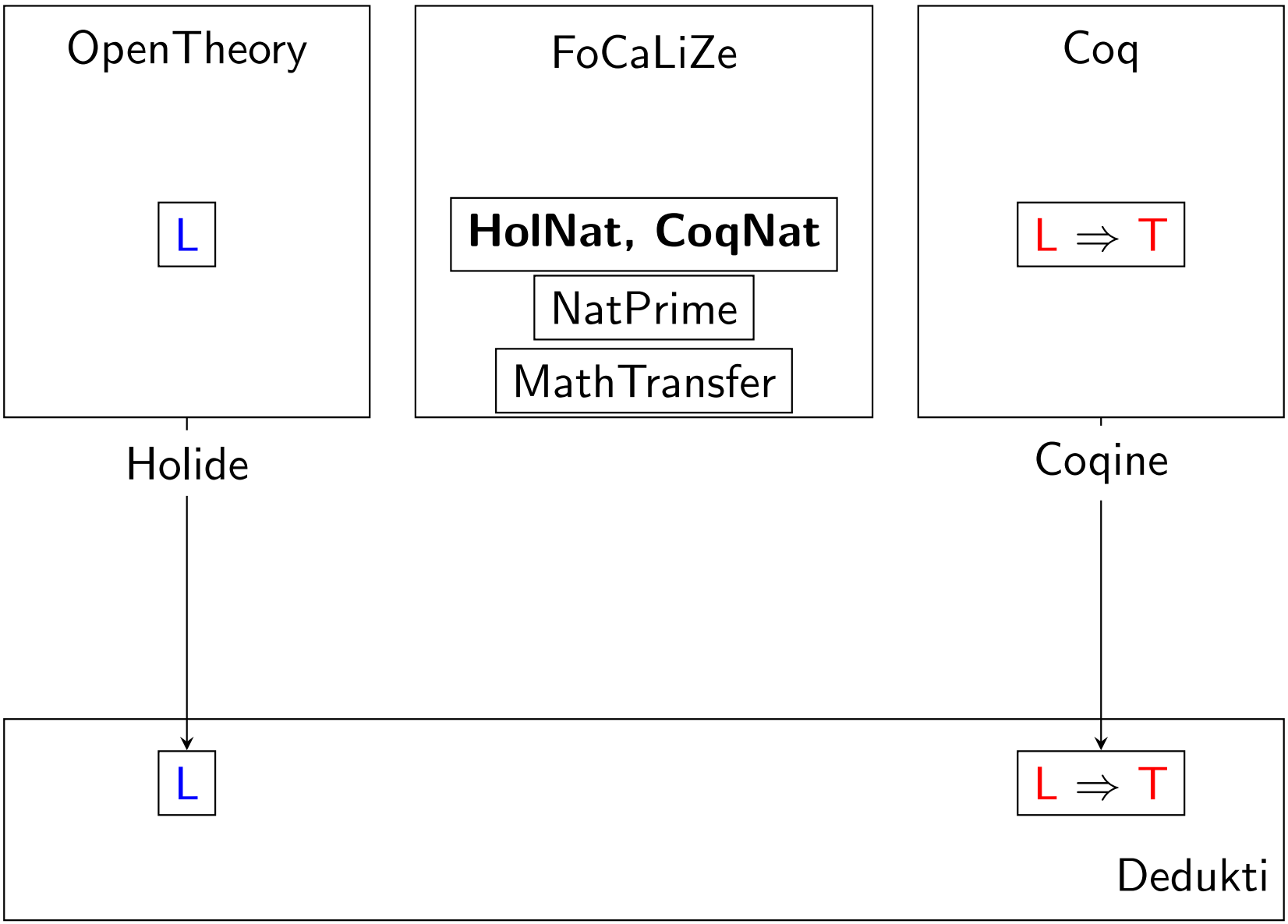

OpenTheory
L
FoCaLiZe
MathTransfer
Coq
L ⇒ T
Holide
Coqine
L
L ⇒ T
Dedukti


OpenTheory
L
FoCaLiZe
NatPrime
MathTransfer
Coq
L ⇒ T
Holide
Coqine
L
L ⇒ T
Dedukti

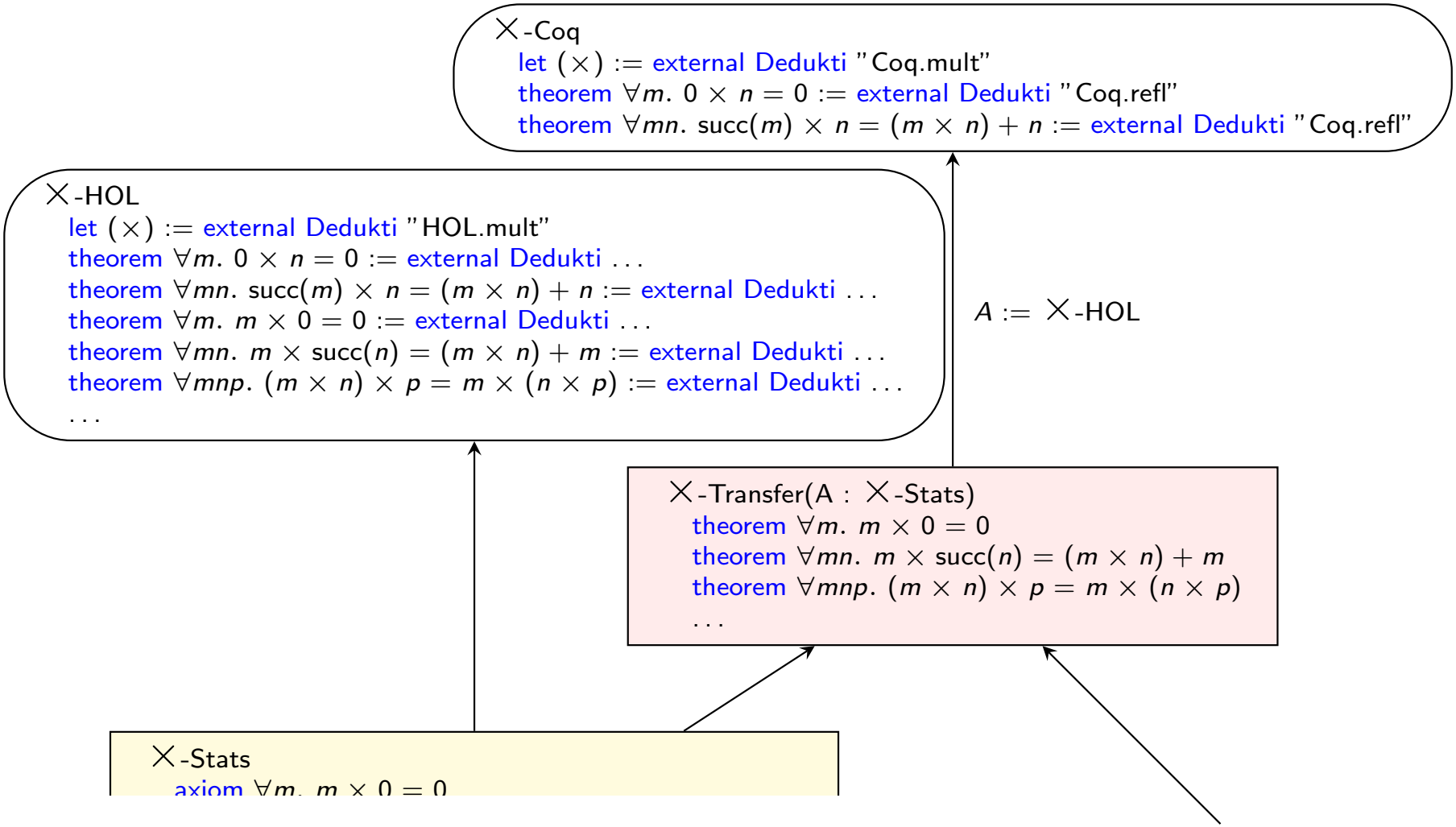
Extend the MathTransfer hierarchies

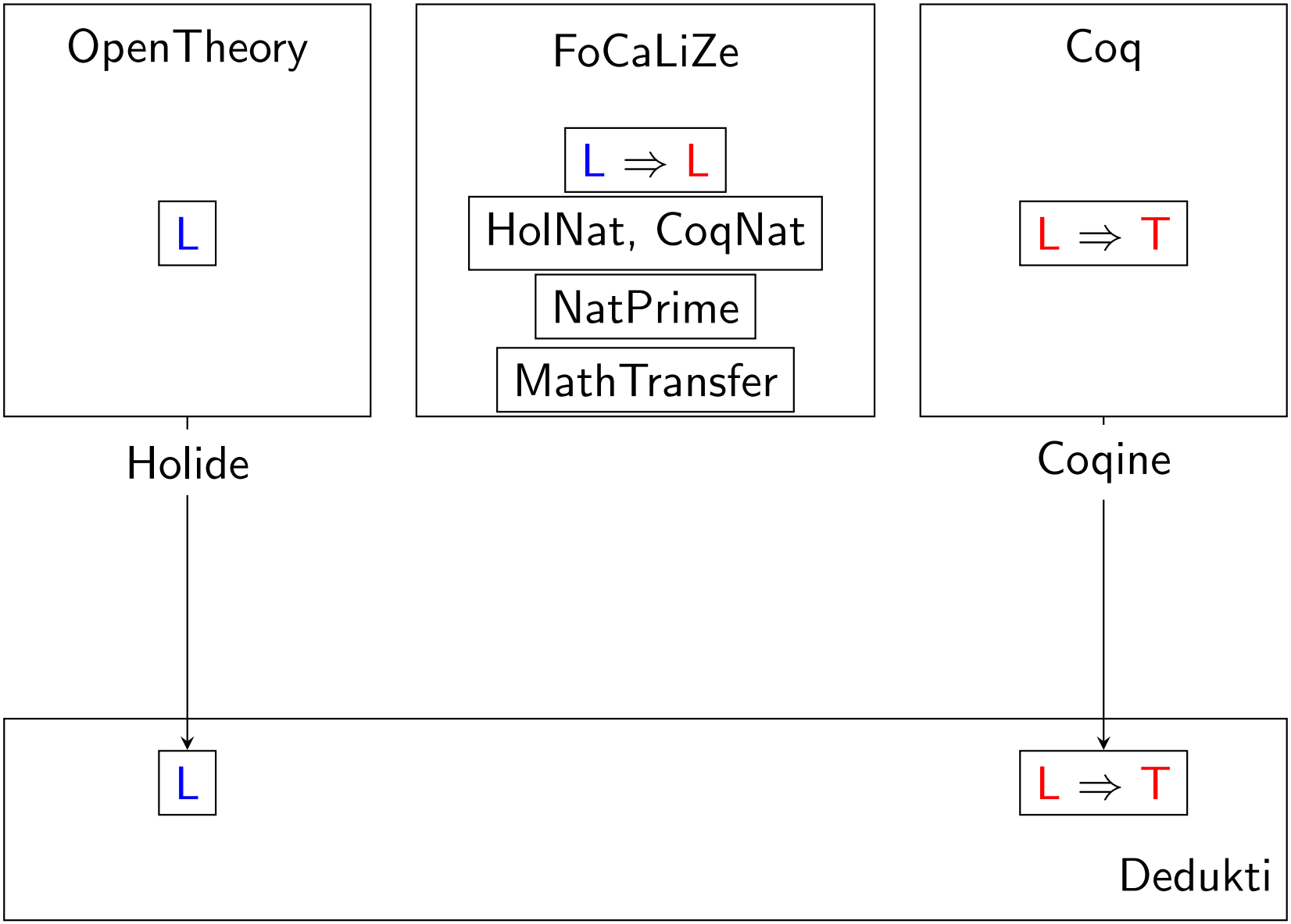


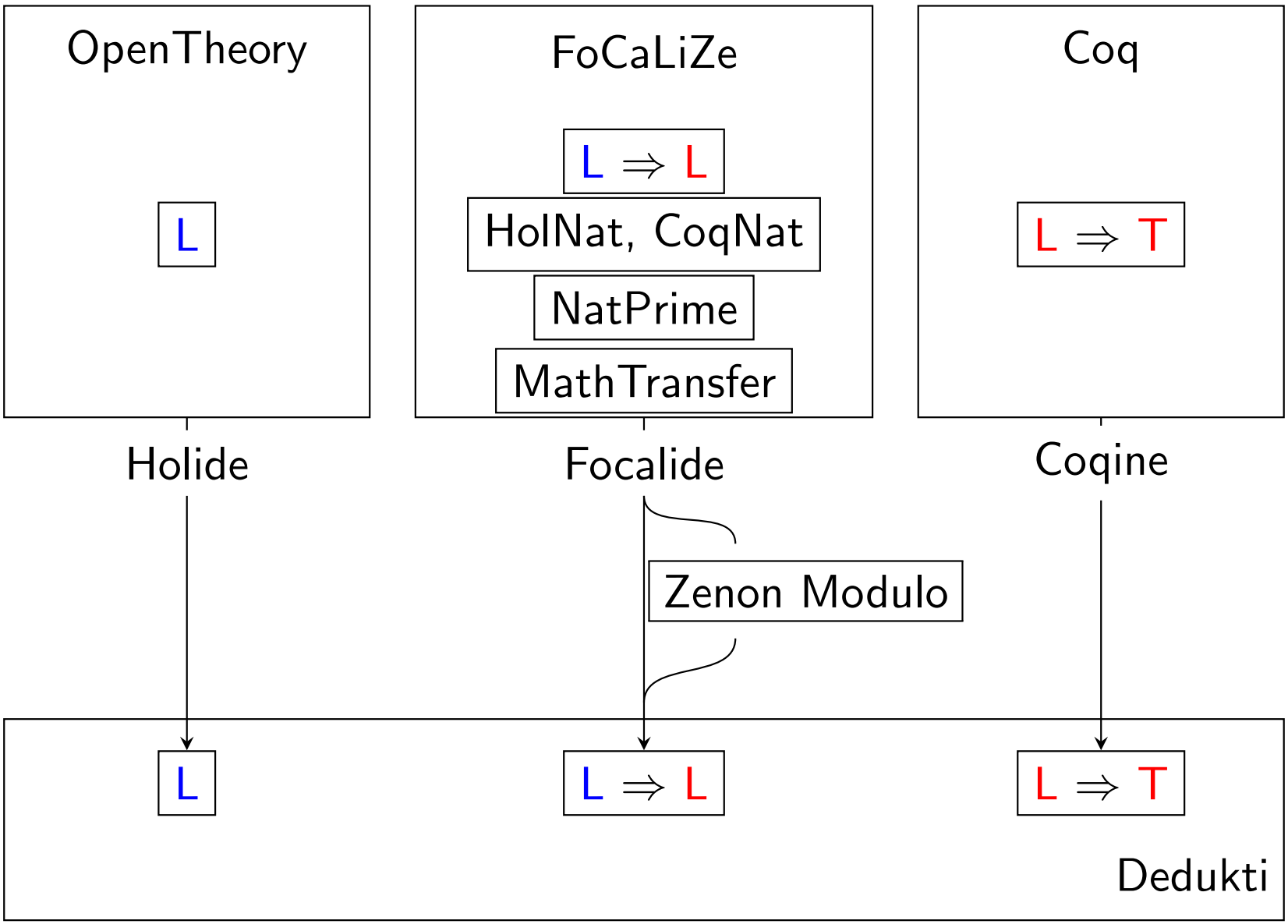
- 3 new operations: divisibility, strict divisibility, and primality
- The morphism hierarchy is also extended

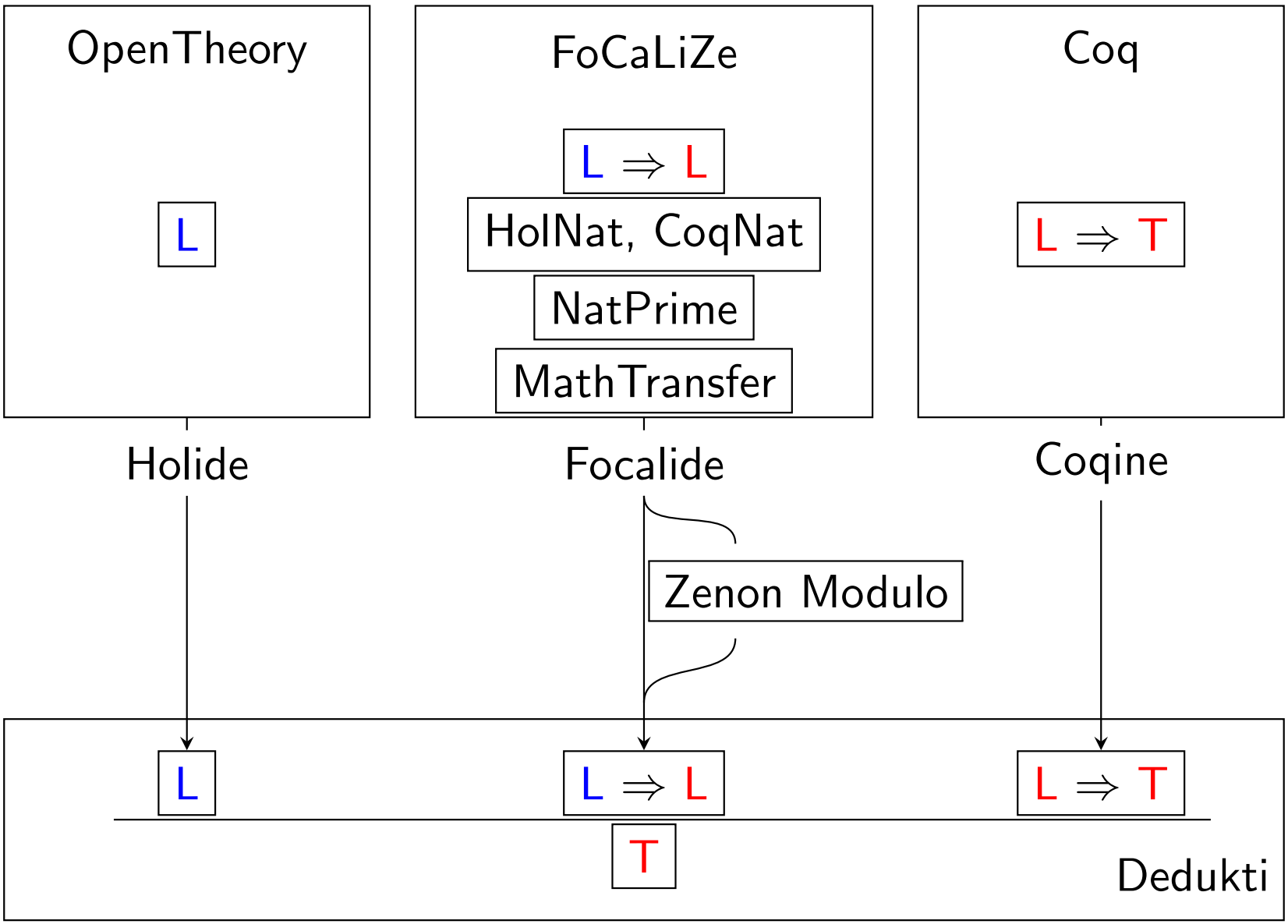

OpenTheory
L
FoCaLiZe
HolNat, CoqNat
NatPrime
MathTransfer
Coq
L ⇒ T
Holide
Coqine
L
L ⇒ T
Dedukti

Instantiate the hierarchy




OpenTheory
L
FoCaLiZe
L ⇒ L
HolNat, CoqNat
NatPrime
MathTransfer
Coq
L ⇒ T
Holide
Coqine
L
L ⇒ T
Dedukti


OpenTheory
L
FoCaLiZe
L ⇒ L
HolNat, CoqNat
NatPrime
MathTransfer
Coq
L ⇒ T
Holide
Focalide
Coqine
Zenon Modulo
L
L ⇒ L
L ⇒ T
Dedukti


OpenTheory
L
FoCaLiZe
L ⇒ L
HolNat, CoqNat
NatPrime
MathTransfer
Coq
L ⇒ T
Holide
Focalide
Coqine
Zenon Modulo
L
L ⇒ L
T
L ⇒ T
Dedukti

Case Study in numbers

	Coq	FoCaLiZe	Zenon Modulo	Dedukti
Source Code	31K	55K		9K
Generated Dedukti	828K	495K	886K	

Conclusion and Perspectives (interoperability)

Contributions

- Working interoperability proof of concept and proposition of Interoperability Methodology (in 8 steps)
- The MathTransfer Library

Generic in the proof systems as long as they have:

- Dedukti translators
- Merged logics (!!!! consistency of $A \cup B$)

What next?

- Extend the library ($\mathbb{Z}$, $\mathbb{R}$, algebra, data structures)
- Plug other logics/translators
- Improve proof automation
- Develop backward translators (ongoing work by Thiré)

Future work FoCaLiZe

- Improvement of FoCaliZe and Zenon/Zenon for shorter proofs
- Opening to other automatic provers while keeping verification by Coq or Dedukti
- Addition of invariants
- Development of applications
- Modern gui

Some references

- S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. *On the way to certify computer algebra systems*, Calculemus, 1999
- V. Prevosto, D. Doligez. *Algorithms and Proofs Inheritancey in the FOC Language*. J. Autom. Reasoning, 2002
- C. Dubois, T. Hardin and V. Viguié Donzeau Gouge. *Building certified components within FOCAL*, Trends in Functional Programming, 2006.
F. Pessaux. *FoCaLiZe: Inside an F-IDE*. F-IDE 2014,
- R. Cauderlier and C. Dubois.*ML pattern-matching, recursion, and rewriting: from FoCaLiZe to Dedukti*. ICTAC 2016.
- A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. *Expressing Theories in the λΠ-Calculus Modulo Theory and in the Dedukti System*. 2016 Draft available online at
- R. Cauderlier and C. Dubois. *FoCaLiZe and Dedukti to the Rescue for Proof Interoperability*. ITP 2017.

Thank you for your attention!