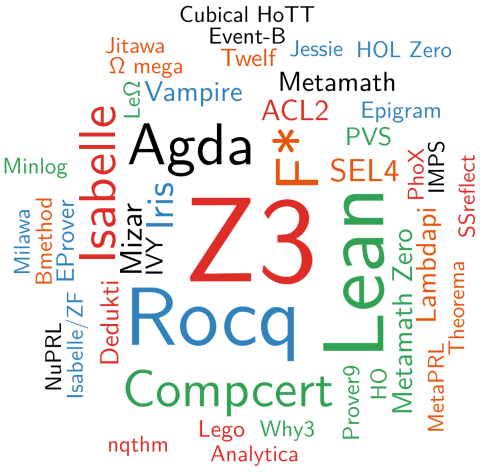
Dedukti and Lambdapi

Mikhail Ushakov

Radboud University

June 4, 2025

The diverse world of Proof Systems

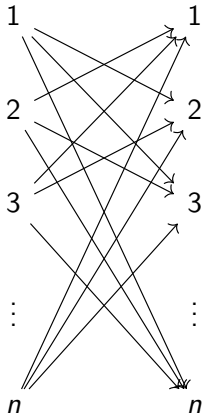


- There are **many** proof systems (more than 40 mentioned in this course)
- Different systems use different logical theories (DTT, HOL)
- Not all theorems and definitions are available in every proof system

Why do we need interoperability?

- Preventing work duplication
- Improving the reliability of formal proofs through cross-checking
- Providing multi-system data to machine learning

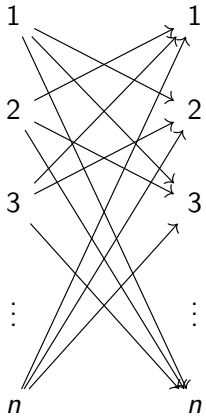
Why interoperability is difficult?



Theoretical issues:

- Each system is based on different axioms
- A theorem proved in one logic may be not provable in another proof system
- Different systems use different tactics

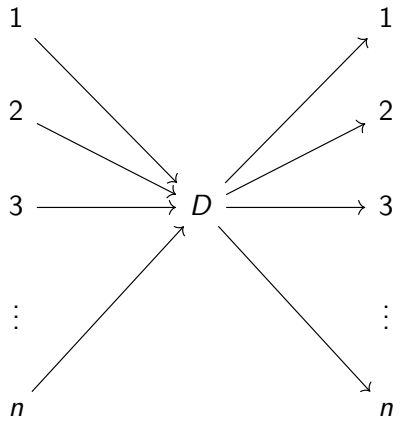
Why interoperability is difficult?



Practical issues:

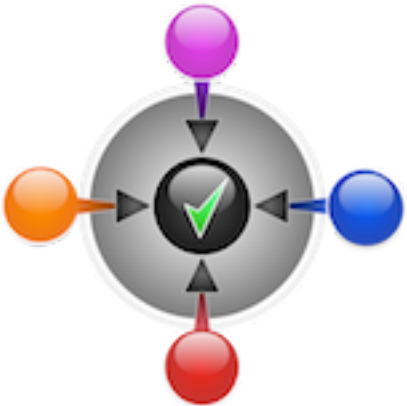
- The number of translations is quadratic: $n \times (n-1)$ for n systems
- How can the translations be maintained and implemented?

The desirable solution



- A logical framework that allows embedding different logics
- Allows for a linear number of cross-system translations: $2n$ for n systems
- Allows for expressive embeddings from other systems that scale with real proofs

Dedukti: Overview



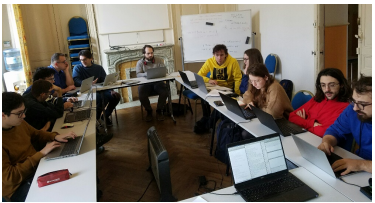
- A logical framework based on $\lambda\Pi$-calculus modulo theory
- It can express different kinds of logic (Constructive logic, CIC, STT, etc.)
- Allows for checking various libraries of proofs developed in different systems

Dedukti: History

- Developed in 2011 by Mathieu Boespflug in Haskell as a PhD thesis in École Polytechnique
- In late 2011, the Deducteam research group in Inria was created
- In 2012, the tool was re-implemented in OCaml

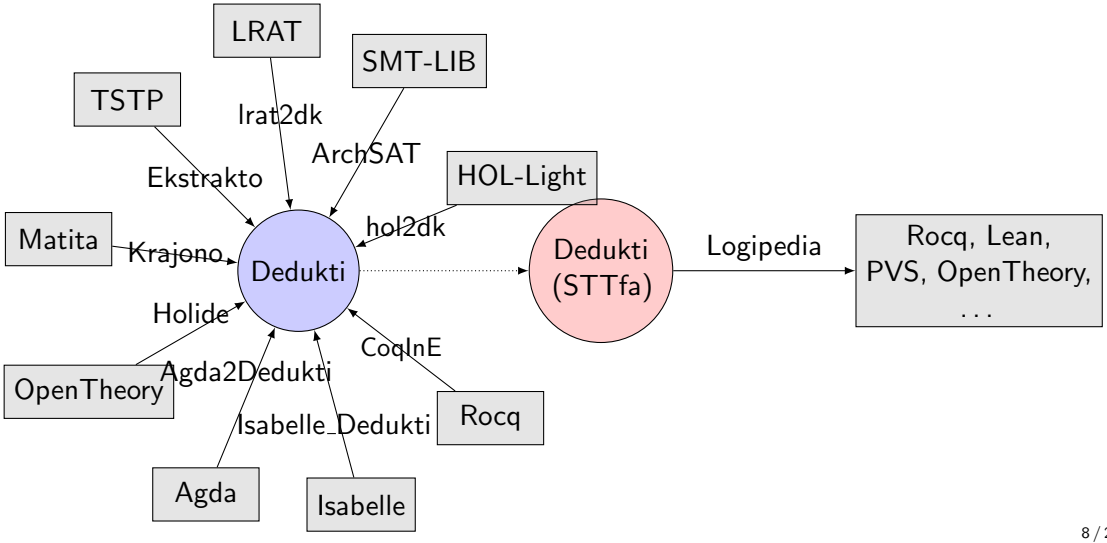
DEDUCTEAM

Dedukti: History



- From 2012 to 2015: new system versions were developed, several logical theories were expressed in Dedukti (Matita, HOL Light, FoCaLiZe, etc.)
- From 2015, Deducteam focused on translating proofs between different logical theories to allow export to different proof systems (Coq, Lean, PVS, etc.)

Dedukti: Projects



Dedukti: Logical Foundations

$\lambda\Pi$-Calculus modulo theory = $\lambda\Pi$-Calculus[1] + Rewriting rules

[1] Extension of the Simply Typed Lambda Calculus with Dependent types

λΠ-Calculus: Typing Rules

Well-formedness of a context

$$\frac{}{[\,] \text{ well-formed}} \qquad \frac{\Gamma \vdash A : \mathit{Kind}}{\Gamma, x : A \text{ well-formed}} \qquad \frac{\Gamma \vdash A : \mathit{Type}}{\Gamma, x : A \text{ well-formed}} \qquad \frac{\Gamma \text{ well-formed}}{\Gamma \vdash \mathit{Type} : \mathit{Kind}}$$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A}(x : A \in \Gamma)$$

λΠ-Calculus: Typing Rules

Product rules

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Kind}}{\Gamma \vdash \Pi x : A \,.\, B : \mathit{Kind}} \qquad \frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type}}{\Gamma \vdash \Pi x : A \,.\, B : \mathit{Type}}$$

Abstraction rules

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Kind} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A \; t : \Pi x : A \,.\, B}$$

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma, x : A \vdash B : \mathit{Type} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A \; t : \Pi x : A \,.\, B}$$

λΠ-Calculus: Typing Rules

Application rule

$$\frac{\Gamma \vdash t : \Pi x : A\,.\,B \qquad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : (t'/x)B}$$

Conversion rules

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type}}{\Gamma \vdash t : B}(A \equiv_\beta B)$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : \mathit{Kind} \qquad \Gamma \vdash B : \mathit{Kind}}{\Gamma \vdash t : B}(A \equiv_\beta B)$$

λΠ-Calculus Modulo Theory

Context is split into local and global ones:

- Locality of the empty context

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash [\,] \text{ local}}$$

- Declaration of an object variable in a local context

$$\frac{\Gamma \vdash \Delta \text{ local} \qquad \Gamma, \Delta \vdash A : \mathit{Type}}{\Gamma \vdash \Delta, x : A \text{ local}}$$

Well-formed global contexts are allowed to store rewrite rules:

$$\frac{\Gamma \text{ well-formed} \qquad \Gamma \vdash \Delta \text{ local}}{\Gamma, l \to^{\Delta} r \text{ well-formed}}$$

$\lambda\Pi$-Calculus Modulo Theory

Conversion rules are extended to account for the rewrite rules

$$\frac{\Gamma \vdash A : \mathit{Type} \qquad \Gamma \vdash B : \mathit{Type} \qquad \Gamma \vdash t : A}{\Gamma \vdash t : B}(A \equiv_{\beta\Gamma} {}^{1}B)$$

$$\frac{\Gamma \vdash A : \mathit{Kind} \qquad \Gamma \vdash B : \mathit{Kind} \qquad \Gamma \vdash t : A}{\Gamma \vdash t : B}(A \equiv_{\beta\Gamma} B)$$

[1] $\equiv_{\beta\Gamma}$ is a reflexive-symmetric transitive closure of the rewrite relation $\rightarrow_{\beta\Gamma}$

λΠ-Calculus Modulo Theory

Subject reduction expresses that the reduction preserves the type of the term

$$\frac{}{[\,] \text{ strongly well-formed}^1} \qquad \frac{\Gamma \text{ strongly well-formed} \qquad \Gamma \vdash A : \mathit{Type}/\mathit{Kind}}{\Gamma, x : A \text{ strongly well-formed}}$$

$$\frac{\Gamma \text{ strongly well-formed} \qquad \Gamma \vdash l \to^{\Delta} r}{\Gamma, l \to^{\Delta} r \text{ strongly well-formed}}$$

Rewrite rule typing

$$\frac{l \text{ is a pattern} \qquad \Gamma, \Delta \vdash \tau l : T \qquad \Gamma, \Delta \vdash \tau r : T \qquad \mathit{dom}(\Delta) \subseteq \mathit{FV}(l)}{\Gamma \vdash l \to^{\Delta} r}$$

[1] All the rules in Γ are well-typed

Dedukti: Syntax

```
nat : Type.

0 : nat.
S : nat -> nat.

def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).

def two := S (S 0).

def K2 := x:nat => two.
```

Dedukti: Syntax

```
nat : Type.
```

⟵ Declaration of a type: $\dfrac{\Gamma \vdash \mathit{Type} : \mathit{Kind}}{\Gamma \vdash \mathit{nat} : \mathit{Type} \text{ well-formed}}$

```
0 : nat.
S : nat -> nat.

def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).

def two := S (S 0).

def K2 := x:nat => two.
```

Dedukti: Syntax

```
nat : Type.

0 : nat.
S : nat -> nat.

def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).

def two := S (S 0).

def K2 := x:nat => two.
```

⟵ Declaration of an object variable: $\dfrac{\Gamma \vdash \mathit{nat} : \mathit{Type}}{\Gamma \vdash \mathit{O} : \mathit{nat} \text{ well-formed}}$

Dedukti: Syntax

```
nat : Type.

0 : nat.
S : nat -> nat.

def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).

def two := S (S 0).

def K2 := x:nat => two.
```

⟵ Types product: $\dfrac{\Gamma \vdash nat : Type \qquad \Gamma, x : nat \vdash nat : Type}{\Gamma \vdash \Pi x : nat\,.\,nat : Type}$

Dedukti: Syntax

```
nat : Type.

0 : nat.
S : nat -> nat.

def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).

def two := S (S 0).

def K2 := x:nat => two.
```

Rewrite rules with local contexts:

$$\frac{\Gamma, \{n\} \vdash \tau\, plus(O, n) : nat \qquad \Gamma, \{n\} \vdash \tau\, n : nat \qquad \{n\} \subseteq FV(plus(O, n))}{\Gamma \vdash plus(O, n) \to^{\{n\}} n}$$

$plus(O, n)$ is a pattern

Dedukti: Syntax

```
nat : Type.

0 : nat.
S : nat -> nat.

def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).

def two := S (S 0).

def K2 := x:nat => two.
```

⟵ Objects application:

$$\frac{\Gamma \vdash S : \Pi x : nat\,.\,nat \qquad \Gamma \vdash S\,O : nat}{\Gamma \vdash S\,(S\,O) : nat}$$

Dedukti: Syntax

```
nat : Type.

0 : nat.
S : nat -> nat.

def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).

def two := S (S 0).

def K2 := x:nat => two.
```

Lambda abstraction:

$$\frac{\Gamma \vdash \mathit{nat} : \mathit{Type} \qquad \Gamma, x : \mathit{nat} \vdash \mathit{nat} : \mathit{Type} \qquad \Gamma, x : \mathit{nat} \vdash \mathit{two} : \mathit{nat}}{\Gamma \vdash \lambda x : \mathit{nat} \,.\, \mathit{two} : \Pi x : \mathit{nat} \,.\, \mathit{nat}}$$

Dedukti: Embedding Minimal Predicate Logic

Minimal Predicate Logic

$$t = x \mid f(t, \ldots, t)$$

$$A = P(t, \ldots, t) \mid (A \to A) \mid \forall x \,.\, A$$

Embedding in Dedukti

$$[\![x]\!] = \mathtt{x}, \qquad [\![f(t_1, \ldots, t_n)]\!] = (\mathtt{f}\ [\![t_1]\!]\ \ldots\ [\![t_n]\!]), \qquad [\![P(t_1, \ldots, t_n)]\!] = (\mathtt{P}\ [\![t_1]\!]\ \ldots\ [\![t_n]\!]),$$

$$[\![A \implies B]\!] = [\![A]\!]\ \mathtt{->}\ [\![B]\!], \qquad [\![\forall_s x \,.\, A]\!] = \mathtt{x:s}\ \mathtt{->}\ [\![A]\!]$$

Dedukti: Demo

Checking a proof for:

$$\forall_T x\,.\,(P(x) \implies P(x))$$

Lambdapi: Overview

- An interactive proof assistant based on $\lambda\Pi$-calculus modulo rewriting
- A new implementation of the Dedukti logical framework
- Allows developing proofs inside the system
- Comes with a repository of pre-defined logics
- Compatible with Dedukti (through the `export` command)
- Sources can be exported to Rocq
- There exist plugins for Emacs, Vim, and VSCode

Lambdapi: History

- The system was proposed by Rodolphe Lepigre
- The first version was released in 2018

Commit `6aed610`

Rodolphe Lepigre committed on Sep 1, 2017

```
First version, few tests, no file format.
```

`master` · lambdapi-1.0 ••• 0.0.8.9

First commit

Lambdapi: Syntax

```
constant symbol nat : TYPE;

constant symbol 0 : nat;
constant symbol S : nat → nat;

symbol plus : nat → nat → nat;
rule plus 0 $n ↪ $n;
rule plus (S $n1) $n2 ↪ S (plus $n1 $n2);

symbol two ≔ S (S 0);
symbol K2 ≔ λ (x : nat), two;
```

Lambdapi: Tactics

The `symbol` command followed by `begin` allows to enter an interactive proof mode.
The `end` exits from the proof mode.

Proof tactics:

abort admit apply assume fail generalize have induction

refine remove set simplify solve why3

Proof tactics on equality:

reflexivity rewrite symmetry

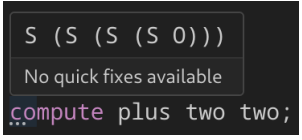
Lambdapi: Tactics

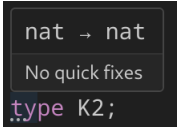
The `symbol` command followed by `begin` allows to enter an interactive proof mode.
The `end` exits from the proof mode.

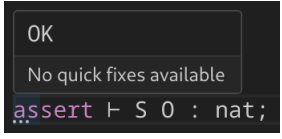
Queries

assert assertnot compute debug flag print proofterm prover

prover_timeout search type verbose







Lambdapi: Demo

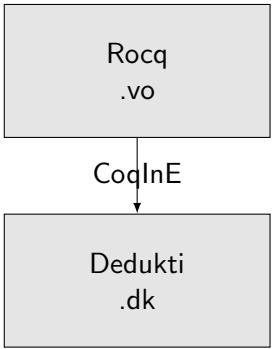
$$\forall_T x\,.\,(P(x) \implies P(x))$$

$$(A \implies B \implies C) \implies (A \implies B) \implies A \implies C$$

$$\forall_T x\,.\,(\forall_T y\,.\,P(y)) \implies P(x)$$

CoqInE

A Rocq plugin to translate Rocq proofs into Dedukti terms.



Overview:

- Is capable of translating a sufficiently large subset of Rocq theory in Dedukti (84% of StdLib)
- Current development is mainly focused on supporting universe polymorphism

Translation details:

- Fixpoints and inductive types are translated using new declarations and rewrite rules
- Since Dedukti only supports namespaces, all structure definitions include module parameters

CoqInE: Demo

Inductive types and inductive predicates.

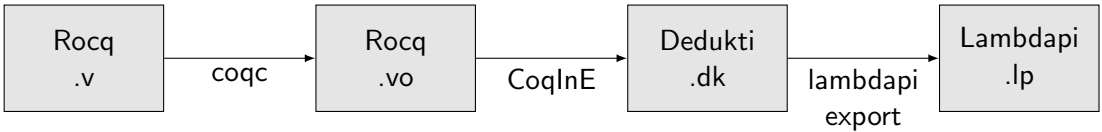
```
Inductive MyNat :=
    | MyO : MyNat
    | MyS : MyNat -> MyNat.

Inductive MyEven : MyNat -> Prop :=
    | O_even : MyEven MyO
    | S_even : forall n : MyNat, MyEven n -> MyEven (MyS (MyS n)).

Lemma two_is_even : MyEven (MyS (MyS MyO)).
Proof.
    apply S_even, O_even.
Qed.
```

CoqInE: Demo

Translation pipeline



CoqInE: Demo

Let's prove:

```
Lemma four_is_even : MyEven (MyS (MyS (MyS (MyS MyO)))).
```

. . . but in Lambdapi

Thank you!

Do you have any questions?