Introduction to proof system interoperability, the Dedukti language and the Lambdapi tool

Frédéric Blanqui

Dedukti team

24 June 2022
Thank you

Nicolas Tabareau, Matthieu Sozeau and their colleagues
for the local organization in Nantes of
Women in EuroProofnet and the 1st Dedukti school!
Outline

Introduction to proof system interoperability

\(\lambda\Pi\)-calculus modulo rewriting

Dedukti language

Lambdapi tool
### Libraries of formal proofs today

<table>
<thead>
<tr>
<th>Library</th>
<th>Nb files</th>
<th>Nb objects*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq Opam</td>
<td>16,000</td>
<td>473,000</td>
</tr>
<tr>
<td>Isabelle AFP</td>
<td>7,000</td>
<td>90,000</td>
</tr>
<tr>
<td>Lean Mathlib</td>
<td>2,000</td>
<td>81,000</td>
</tr>
<tr>
<td>Mizar Mathlib</td>
<td>1,400</td>
<td>77,000</td>
</tr>
<tr>
<td>HOL-Light</td>
<td>500</td>
<td>35,000</td>
</tr>
</tbody>
</table>

* type, definition, theorem, …

![LOC](chart.png)

Every system has basic libraries on integers, lists, … Some definitions/theorems are available in one system only ⇒ Can't we translate a proof between two systems automatically?
Libraries of formal proofs today

<table>
<thead>
<tr>
<th>Library</th>
<th>Nb files</th>
<th>Nb objects*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq Opam</td>
<td>16,000</td>
<td>473,000</td>
</tr>
<tr>
<td>Isabelle AFP</td>
<td>7,000</td>
<td>90,000</td>
</tr>
<tr>
<td>Lean Mathlib</td>
<td>2,000</td>
<td>81,000</td>
</tr>
<tr>
<td>Mizar Mathlib</td>
<td>1,400</td>
<td>77,000</td>
</tr>
<tr>
<td>HOL-Light</td>
<td>500</td>
<td>35,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

* type, definition, theorem, ... 

▶ Every system has basic libraries on integers, lists, ...
▶ Some definitions/theorems are available in one system only
## Libraries of formal proofs today

<table>
<thead>
<tr>
<th>Library</th>
<th>Nb files</th>
<th>Nb objects*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coq Opam</td>
<td>16,000</td>
<td>473,000</td>
</tr>
<tr>
<td>Isabelle AFP</td>
<td>7,000</td>
<td>90,000</td>
</tr>
<tr>
<td>Lean Mathlib</td>
<td>2,000</td>
<td>81,000</td>
</tr>
<tr>
<td>Mizar Mathlib</td>
<td>1,400</td>
<td>77,000</td>
</tr>
<tr>
<td>HOL-Light</td>
<td>500</td>
<td>35,000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

* type, definition, theorem, ...

- Every system has basic libraries on integers, lists, ...
- Some definitions/theorems are available in one system only

⇒ Can’t we translate a proof between two systems automatically?
Interest of proof interoperability

- Avoid duplicating developments and losing time
- Facilitate development of new proof systems
- Increase reliability of formal proofs (cross-checking)
- Facilitate validation by certification authorities
- Relativize the choice of a system (school, industry)
- Provide multi-system data to machine learning
Difficulties of interoperability

- Each system is based on different axioms and deduction rules.
- It is usually non-trivial and sometimes impossible to translate a proof from one system to the other (e.g., a classical proof in an intuitionistic system).
- Is it reasonable to have \( n(n - 1) \) translators for \( n \) systems?
Difficulties of interoperability

- Each system is based on different axioms and deduction rules.
- It is usually non-trivial and sometimes impossible to translate a proof from one system to the other (e.g., a classical proof in an intuitionistic system).
- Is it reasonable to have $n(n - 1)$ translators for $n$ systems?
A common language for proof systems?

Logical framework $D$

language for describing axioms, deduction rules and proofs of a system $S$ as a theory $D(S)$ in $D$

Example: $D = \text{predicate calculus}$

allows one to represent $S=\text{geometry}$, $S=\text{arithmetic}$, $S=\text{set theory}$, . . .

not well suited for functional computations and dependent types
A common language for proof systems?

Logical framework $D$

language for describing axioms, deduction rules and proofs of a system $S$ as a theory $D(S)$ in $D$

Example: $D = \text{predicate calculus}$
allows one to represent $S=\text{geometry}$, $S=\text{arithmetic}$, $S=\text{set theory}$, …
not well suited for functional computations and dependent types

Better: $D = \lambda\Pi$-calculus modulo rewriting
allows one to represent also:
$S=\text{HOL}$, $S=\text{Coq}$, $S=\text{Agda}$, $S=\text{PVS}$, …
How to translate a proof $t \in A$ in a proof $u \in B$?

In a logical framework $D$:

1. translate $t \in A$ in $t' \in D(A)$

2. identify the axioms and deduction rules of $A$ used in $t'$

3. translate $u' \in D(B)$ in $u \in B$
How to translate a proof \( t \in A \) in a proof \( u \in B \)?

In a logical framework \( D \):

1. translate \( t \in A \) in \( t' \in D(A) \)

2. identify the axioms and deduction rules of \( A \) used in \( t' \)
   translate \( t' \in D(A) \) in \( u' \in D(B) \) if possible

3. translate \( u' \in D(B) \) in \( u \in B \)
How to translate a proof $t \in A$ in a proof $u \in B$?

In a logical framework $D$:

1. translate $t \in A$ in $t' \in D(A)$

2. identify the axioms and deduction rules of $A$ used in $t'$
   translate $t' \in D(A)$ in $u' \in D(B)$ if possible

3. translate $u' \in D(B)$ in $u \in B$

$\Rightarrow$ represent in the same way functionalities common to $A$ and $B$
The modular $\lambda\Pi/R$ theory $U$ and its sub-theories

38 symbols, 28 rules, 13 sub-theories
Dedukti, an assembly language for proof systems

Michael

AtelierB

starting
ICSPA project

Amélie

K

Isabelle

HOL

Bruno

Cubical

Lean

Coq

FoCaLiZe

Guillaume

Catherine

Frédéric

Gabriel

Guillaume

Theory U

Jesper & Thiago

Agda

Matita

HOL

Isabelle

Vampire, E, ...

automated provers

Mizar?

Nuprl?

TSTP

PVS

Lambdapi

ArchSAT

Zenon

Cooked

Guillaume

Lambdapi

AtelierB

TLAPS

Dedukti

starting
ICSPA project

Vampire, E, ...

automated provers

Mizar?

Nuprl?
## Libraries currently available in Dedukti

<table>
<thead>
<tr>
<th>System</th>
<th>Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOL-Light</td>
<td>OpenTheory</td>
</tr>
<tr>
<td>Matita</td>
<td>Arith</td>
</tr>
<tr>
<td>Coq</td>
<td>Stdlib parts, GeoCoq</td>
</tr>
<tr>
<td>Isabelle</td>
<td>HOL.Complex_Main <strong>NEW!</strong> <em>(AFP soon?)</em></td>
</tr>
<tr>
<td>Agda</td>
<td>Stdlib parts (± 25%)</td>
</tr>
<tr>
<td>PVS</td>
<td>Stdlib parts</td>
</tr>
<tr>
<td>TPTP</td>
<td>E 69%, Vampire 83%</td>
</tr>
</tbody>
</table>

**Case study:**

Matita/Arith → OpenTheory, Coq, PVS, Lean, Agda

[http://logipedia.inria.fr](http://logipedia.inria.fr)
Outline

Introduction to proof system interoperability

\(\lambda\Pi\)-calculus modulo rewriting

Dedukti language

Lambdapi tool
What is the $\lambda\Pi$-calculus modulo rewriting?

$\lambda\Pi/\mathcal{R} =$

- $\lambda$ simply-typed $\lambda$-calculus
- $\Pi$ dependent types, e.g. array($n$)
- $\mathcal{R}$ identification of types modulo rewrites rules $l \leftrightarrow r$
What is the $\lambda\Pi$-calculus modulo rewriting?

\[ \lambda\Pi/\mathcal{R} = \]

- $\lambda$ simply-typed $\lambda$-calculus
- $\Pi$ dependent types, e.g. $\text{array}(n)$
- $\mathcal{R}$ identification of types modulo rewrites rules $l \rightarrow r$

**terms $t, u =$**

- $\text{TYPE}$ sort of types
- $f$ global constant
- $x$ local variable
- $tu$ application
- $\lambda x : t, u$ abstraction
- $\Pi x : t, u$ dependent product
- $t \rightarrow u$ abbreviation for $\Pi x : t, u$ when $x \notin u$
What is the $\lambda\Pi$-calculus modulo rewriting?

theory =

$\Sigma$ sequence of type declarations for global constants

$+ \mathcal{R}$ set of rewrite rules $l \leftrightarrow r$

including rules on types!
What is the $\lambda\Pi$-calculus modulo rewriting?

theory =

$\Sigma$ sequence of type declarations for global constants

$+ \mathcal{R}$ set of rewrite rules $l \rightarrow r$

including rules on types $!$

typing = ... +

\[
\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A, B : \text{TYPE}}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B}
\]

$\Gamma: \text{types of local variables}$

\[
\frac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B\{x \mapsto u}\}
\]

$\Gamma \vdash t : A \quad A \equiv_{\beta\mathcal{R}} B$

$\frac{}{\Gamma \vdash t : B}$ $\equiv_{\beta\mathcal{R}}: \text{equational theory}$

generated by $\beta$ and $\mathcal{R}$

\text{ex: } \text{concat} : \Pi p : \mathbb{N}, \text{array } p \rightarrow \Pi q : \mathbb{N}, \text{array } q \rightarrow \text{array}(p + q)$

$\text{concat } 2 : \text{array } 2 \rightarrow \Pi q : \mathbb{N}, \text{array } q \rightarrow \text{array}(2 + q)$
Properties of the $\lambda\Pi$-calculus modulo rewriting

$\lambda\Pi/\mathcal{R}$ enjoys all the properties of $\lambda\Pi$:

- unicity of types modulo $\equiv_{\beta\mathcal{R}}$
- decidability of $\equiv_{\beta\mathcal{R}}$ and type-checking

assuming that $\hookrightarrow_{\beta\mathcal{R}}$:

- terminates: there is no infinite $\hookrightarrow_{\beta\mathcal{R}}$ sequences
- is confluent: the order of $\hookrightarrow_{\beta\mathcal{R}}$ steps does not matter
- $\mathcal{R}$ preserves typing: if $l\theta : A$ and $l \hookrightarrow r \in \mathcal{R}$ then $r\theta : A$

There exists (certified) tools for checking those properties
Outline

Introduction to proof system interoperability

λΠ-calculus modulo rewriting

Dedukti language

Lambdapi tool
What is Dedukti?

Dedukti is a concrete language for defining $\lambda\Pi/R$ theories

There are several tools to check the correctness of Dedukti files:

- Kocheck  https://github.com/01mf02/kontroli-rs
- Dkcheck  https://github.com/Deducteam/dedukti
- Lambdapi https://github.com/Deducteam/lambdapi

Efficiency: Kocheck $>$ Dkcheck $>$ Lambdapi
Features: Kocheck $<$ Dkcheck $<$ Lambdapi

Dkcheck and Lambdapi can export $\lambda\Pi/R$ theories to:
- the HRS format of the confluence competition
- the XTC format of the termination competition
  extended with dependent types
How to install and use Kocheck?

**Installation:**
```
cargo install --git https://github.com/01mf02/kontroli-rs
```

**Use:**
```
kocheck file.dk
```
How to install and use Dkcheck?

Installation:

Using Opam:

```
opam install dedukti
```

Compilation from the sources:

```
git clone https://github.com/Deducteam/dedukti.git
cd dedukti
make
make install
```

Use:

```
dk check file.dk
```
BNF grammar:

file extension: .dk

comments: (; ... (; ... ;) ... ;)

identifiers:
(a-z|A-Z|0-9|_)+ and {1 arbitrary string 1}
Terms

Type

id

id.id

term term ... term

id [ : term ] => term

[id :] term -> term

( term )

sort for types

variable or constant

constant from another file

application

abstraction

[dependent] product
Command for declaring/defining a symbol

\[ \text{modifier}* \ id \ param* : \ term \ [\text{:=} \ term] \ . \]

\[ \text{param} := ( \ id : \ term ) \]

modifier's:

- **def**: definable
- **thm**: never reduced
- **AC**: associative and commutative
- **private**: exported but usable in rule left-hand sides only
- **injective**: used in subject reduction

\[
\begin{align*}
\text{N} & : \text{Type}. \\
\text{0} & : \text{N}. \\
\text{s} & : \text{N} \rightarrow \text{N}. \\
\text{def add} & : \text{N} \rightarrow \text{N} \rightarrow \text{N}. \\
\text{thm add_com} & : \\
& \text{x:} \text{N} \rightarrow \text{y:} \text{N} \rightarrow \text{Eq (add x y) (add y x)} := \ldots
\end{align*}
\]
Command for declaring rewrite rules

\[
[ id^* ] (term \rightarrow term)^+. 
\]

\[
[ x \ y ] \\
x + 0 \rightarrow x \\
x + s\ y \rightarrow s\ (x + y) .
\]

Dkcheck tries to automatically check:

**preservation of typing by rewrite rules** (aka subject reduction)
Queries and assertions

# INFER term .
# EVAL term .
(# ASSERT | # ASSERTNOT) term (==) term .
(# CHECK | # CHECKNOT) term (==) term .

# INFER 0.
# EVAL add 2 2.

# ASSERT 0 : N.
# ASSERTNOT 0 : N \rightarrow N.

# ASSERT add 2 2 == 4.
# ASSERTNOT add 2 2 == 5.
Importing the declarations of other files

file1.dk:

A : Type.

file2.dk:

#REQUIRE file1.
a : file1.A.
Outline

Introduction to proof system interoperability

\( \lambda \Pi \)-calculus modulo rewriting

Dedukti language

Lambdapi tool
What is Lambdapi?

**Lambdapi is an interactive proof assistant for** $\lambda\Pi/\mathcal{R}$

- has its own syntax and file extension `.lp`
- can read and output `.dk` files
- symbols can have implicit arguments
- symbol declaration/definition generates typing/unification goals
- goals can be solved by structured proof scripts (tactic trees)
- ...
Where to find Lambdapi?

Webpage: https://github.com/Deducteam/lambdapi

User manual: https://lambdapi.readthedocs.io/

Libraries:
https://github.com/Deducteam/opam-lambdapi-repository
How to install Lambdapi?

2 possibilities:

1. Using Opam:

   opam install lambdapi

2. Compilation from the sources:

   git clone https://github.com/Deducteam/lambdapi.git
   cd lambdapi
   make
   make install
How to use Lambdapi?

2 possibilities:

1. Command line (batch mode):
   
   `lambdapi check file.lp`

2. Through an editor (interactive mode):
   
   ▶ Emacs
   ▶ VSCode

   Lambdapi automatically (re)compiles dependencies if necessary
How to install the Emacs interface?

3 possibilities:

1. Nothing to do when installing Lambdapi with opam

2. From Emacs using MELPA:

   \texttt{M-x \text{package\text{-}install} \ RET \ \text{lambdapi\text{-}mode}}

3. From sources:

   \texttt{make install_emacs}

   \texttt{+ add in ~/.emacs:}

   \texttt{(load \text{"lambdapi\text{-}site\text{-}file"})}
Emacs interface

edition buffer

goals

messages

window layout can be customized

checked part

How to install the VSCode interface?

From the VSCode Marketplace
VSCode interface

checked part

goals

edition buffer

messages
developments must have a file `lambdapi.pkg` describing where to install the files relatively to the root of all installed libraries

```
package_name = my_lib
root_path = logical.path.from.root.to.my_lib
```
Importing the declarations of other files

**lambdapi.pkg:**

```plaintext
package_name = unary
root_path = nat.unary
```

**file1.lp:**

```plaintext
symbol A : TYPE;
```

**file2.lp:**

```plaintext
require nat.unary.file1;
symbol a : nat.unary.file1.A;
open nat.unary.file1;
symbol a' : A;
```

**file3.lp:**

```plaintext
require open nat.unary.file1 nat.unary.file2;
symbol b := a;
```
Lambdapi syntax

BNF grammar:

file extension: .lp

comments: /* ... */ or // ...

identifiers: UTF16 characters and { | arbitrary string |}
Terms

TYPE
(id .)*id

term term ... term

\( \lambda \) id [ : term ] , term

\( \Pi \) id [ : term ] , term

term \( \rightarrow \) term

-

let id [ : term ] := term in term

( term )

sort for types

variable or constant

application

abstraction

dependent product

non-dependent product

unknown term
Command for declaring/defining a symbol

```latex
modifier* symbol id param* [: term ] [= term ] [begin proof end] ;

\text{param} = id \mid _ \mid ( id ^ : \text{term} ) \mid [ id ^ : \text{term} ]
```

**modifier's:**

- **constant:** not definable
- **opaque:** never reduced
- **associative**
- **commutative**
- **private:** not exported
- **protected:** exported but usable in rule left-hand sides only
- **sequential:** reduction strategy
- **injective:** used in unification
Examples of symbol declarations

symbol $N$ : TYPE;
symbol 0 : $N$;
symbol $s$ : $N \rightarrow N$;

symbol $+$ : $N \rightarrow N \rightarrow N$; notation $+$ infix right 10;

symbol $\times$ : $N \rightarrow N \rightarrow N$; notation $\times$ infix right 20;
Command for declaring rewrite rules

```
rule term ↦ term (with term ↦ term )* ;
```

Pattern variables must be prefixed by $:

```
rule $x + 0 ↦ $x
with $x + s $y ↦ s ($x + $y);
```

Lambdapi tries to automatically check:

**Preservation of typing by rewrite rules** (aka subject reduction)
Command for adding rewrite rules

Lambdapi supports:

**overlapping rules**

```latex
\text{rule} \ x + 0 \leftrightarrow x
\text{with} \ x + s \ y \leftrightarrow s (x + y)
\text{with} \ 0 + x \leftrightarrow x
\text{with} \ s \ x + y \leftrightarrow s (x + y);
```

**matching on defined symbols**

```latex
\text{rule} \ (x + y) + z \leftrightarrow x + (y + z);
```

**non-linear patterns**

```latex
\text{rule} \ x - x \leftrightarrow 0;
```

Lambdapi tries to automatically check:

**local confluence** (AC symbols/HO patterns not handled yet)
symbol R: TYPE;

symbol 0: R;
symbol sin: R → R;
symbol cos: R → R;
symbol D: (R → R) → (R → R);

rule D (λ x, sin $F.[x])
  ↦ λ x, D $F.[x] × cos $F.[x];
rule D (λ x, $V.[[]])
  ↦ λ x, 0;
Non-linear matching

Example: decision procedure for group theory

symbol G : TYPE;
symbol 1 : G;
symbol · : G → G → G; notation · infix 10;
symbol inv : G → G;

rule ($x \cdot $y) · $z ↦ $x · ($y · $z)
with 1 · $x ↦ $x
with $x · 1 ↦ $x
with inv $x · $x ↦ 1
with $x · inv $x ↦ 1
with inv $x · ($x · $y) ↦ $y
with $x · (inv $x · $y) ↦ $y
with inv 1 ↦ 1
with inv (inv $x) ↦ $x
with inv ($x · $y) ↦ inv $y · inv $x;
Defining inductive-recursive types

because symbol and rule declarations are separated, one can easily define inductive-recursive types in Dedukti or Lambdapi:

// lists without duplicated elements

constant symbol L : TYPE;

symbol $\notin : \mathbb{N} \rightarrow L \rightarrow \text{Prop};$ notation $\notin$ infix 20;

constant symbol nil : L;
constant symbol cons x l : Prf($x \notin l$) → L;

rule _ $\notin$ nil $\rightarrow \top$
with $x \notin$ cons $y$ $l$ _ $\rightarrow$ $x \neq y \land x \notin y$;
Command for generating induction principles

```plaintext
inductive N : TYPE := 0 : N | s : N → N;
```

is equivalent to:

```plaintext
symbol N : TYPE;
symbol 0 : N;
symbol s : N → N;
symbol ind_N (p : N → Prop)
  (case_0: Prf(p 0))
  (case_s: Π x : N, Prf(p x) → Prf(p(s x)))
  (n : N) : Prf(p n);
rule ind_N $p $c0 $cs 0 → $c0
with ind_N $p $c0 $cs (s $x)
  → $cs $x (ind_N $p $c0 $cs $x)
```

Lambdapi handles strictly positive parametric inductive types
Example of inductive-inductive type

```haskell
/* contexts and types in dependent type theory Forsberg’s 2013 PhD thesis */

// contexts
inductive Ctx : TYPE :=
| □ : Ctx
| · Γ : Ty Γ → Ctx

// types
with Ty : Ctx → TYPE :=
| U Γ : Ty Γ
| P Γ a : Ty (· Γ a) → Ty Γ;
```
Queries and assertions

print \textit{id} ; 
\textbf{type} \textit{term} ; 
\textbf{compute} \textit{term} ; 
(\textbf{assert} | \textbf{assertnot}) \textit{id} \ast \vdash \textit{term} (: | \equiv) \textit{term} ;

print \textit{N}; \textit{// constructors and induction principle} 
print +; \textit{// type and rules} 

\textbf{type} \textit{\times} ; 
\textbf{compute} 2 \times 5 ;

\textbf{assert} 0 : \textit{N} ; 
\textbf{assertnot} 0 : \textit{N} \rightarrow \textit{N} ; 

\textbf{assert} \textit{x y z} \vdash \textit{x + y \times z} \equiv \textit{x + (y \times z)} ; 
\textbf{assertnot} \textit{x y z} \vdash \textit{x + y \times z} \equiv (\textit{x + y}) \times \textit{z} ;
Reducing proof checking to type checking
(aka the Curry-Howard isomorphism)

// type of propositions
symbol Prop : TYPE;
symbol = : N → N → Prop; notation = infix 1;

// interpretation of propositions as types
// (Curry-Howard isomorphism)
symbol Prf : Prop → TYPE;

// examples of axioms
symbol =-refl x : Prf(x = x);
symbol =-s x y : Prf(x = y) → Prf(s x = s y);
symbol ind_N (p : N → Prop)
  (case_0 : Prf(p 0))
  (case_s : Π x : N, Prf(p x) → Prf(p(s x)))
  (n : N) : Prf(p n);
Stating an axiom vs Proving a theorem

Stating an axiom:

opaque symbol 0_is_neutral_for_+ x : 
    Prf (0 + x = x);
// no definition given now
// one can still be given later with a rule

Proving a theorem:

opaque symbol 0_is_neutral_for_+ x : 
    Prf (0 + x = x) :=
// generates the typing goal Prf (0 + x = x)
// a proof must be given now
begin
    ...
end;
Goals and proofs

symbol declarations/definitions can generate:

- typing goals
  \[ x_1 : A_1, \ldots, x_n : A_n \vdash ? : B \]
- unification goals
  \[ x_1 : A_1, \ldots, x_n : A_n \vdash t \equiv u \]

these goals can be solved by writing proof’s:

\[
\text{proof ::= (proof_step ;)*} \\
\text{proof_step ::= tactic \{ proof \})*}
\]

- a proof is a ;-separated sequence of proof_step’s
- a proof_step is a tactic followed by as many proof’s enclosed in curly braces as the number of goals generated by the tactic

tactic’s for unification goals:
- solve (applied automatically)
Example of proof


opaque symbol 0_is_neutral_for_ + x :
  Prf(0 + x = x)
:= begin
  induction
  {simplify; reflexivity;}
  {assume x h; simplify; rewrite h; reflexivity;}
end;
Tactics for typing goals

▶ simplify \([id]\)
▶ refine \(term\)
▶ assume \(id^+\)
▶ generalize \(id\)
▶ apply \(term\)
▶ induction
▶ have \(id : term\)
▶ reflexivity
▶ symmetry
▶ rewrite \([right][pattern] term\)

▶ why3

like Coq SSReflect

calls external provers
Lambdapi’s additional features wrt Dkcheck/Kocheck

**Lambdapi is an interactive proof assistant for** \(\lambda\Pi/\mathcal{R}\)

- has its own syntax and file extension \(lp\)
- can read and output \(dk\) files
- supports Unicode characters and infix operators
- symbols can have implicit arguments
- symbol declaration/definition generates typing/unification goals
- goals can be solved by structured proof scripts (tactic trees)
- provides a rewrite tactic similar to Coq/SSReflect
- can call external (first-order) theorem provers
- provides a command for generating induction principles
- provides a local confluence checker
- handles associative-commutative symbols differently
- supports user-defined unification rules