1st EuroProofNet Dedukti School

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

Frédéric Blanqui

Deduc⊢eam

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

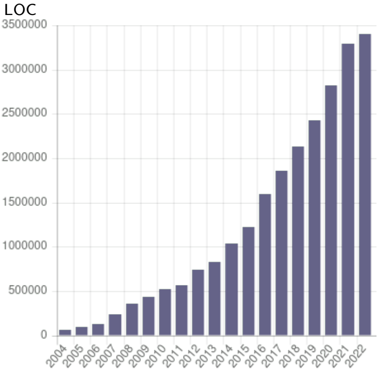
Dedukti language

Lambdapi tool

Libraries of formal proofs today

Library	Nb files	Nb objects*
Coq Opam	16,000	473,000
Isabelle AFP	7,000	90,000
Lean Mathlib	2,000	81,000
Mizar Mathlib	1,400	77,000
HOL-Light	500	35,000
...	...	...

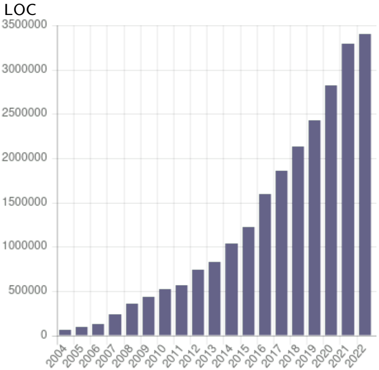
* type, definition, theorem, . . .



Libraries of formal proofs today

Library	Nb files	Nb objects*
Coq Opam	16,000	473,000
Isabelle AFP	7,000	90,000
Lean Mathlib	2,000	81,000
Mizar Mathlib	1,400	77,000
HOL-Light	500	35,000
...	...	...

* type, definition, theorem, ...

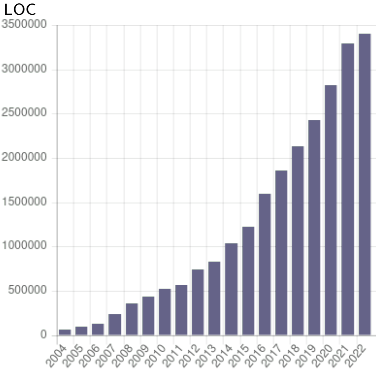


- Every system has basic libraries on integers, lists, ...
- Some definitions/theorems are available in one system only

Libraries of formal proofs today

Library	Nb files	Nb objects*
Coq Opam	16,000	473,000
Isabelle AFP	7,000	90,000
Lean Mathlib	2,000	81,000
Mizar Mathlib	1,400	77,000
HOL-Light	500	35,000
...	...	...

* type, definition, theorem, ...



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

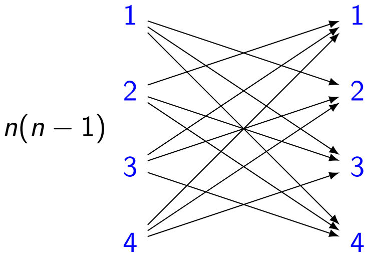
⇒ Can't we translate a proof between two systems automatically?

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

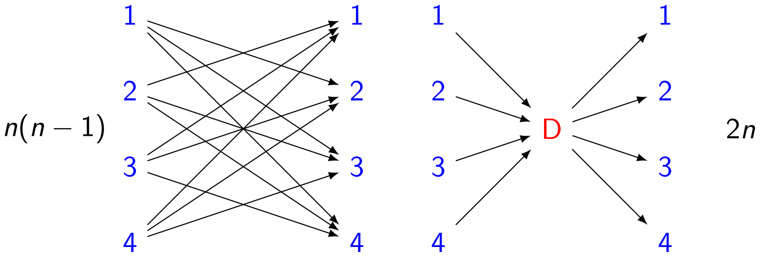
Difficulties of interoperability

- 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 = predicate calculus

allows one to represent S=geometry, S=arithmetic, S=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: $D =$ predicate calculus

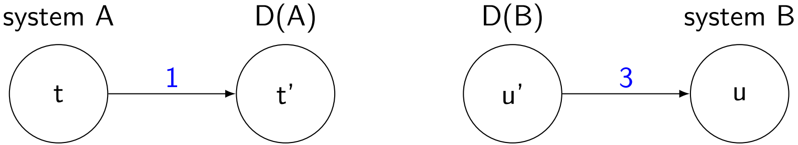
allows one to represent S=geometry, S=arithmetic, S=set theory, ...
not well suited for functional computations and dependent types

Better: $D = \lambda\Pi$-calculus modulo rewriting

allows one to represent also:
S=HOL, S=Coq, S=Agda, S=PVS, ...

How to translate a proof $t \in A$ in a proof $u \in B$?

In a logical framework D:

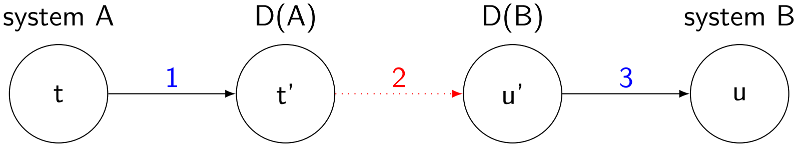


1. translate $t \in A$ in $t' \in D(A)$

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

How to translate a proof $t \in A$ in a proof $u \in B$?

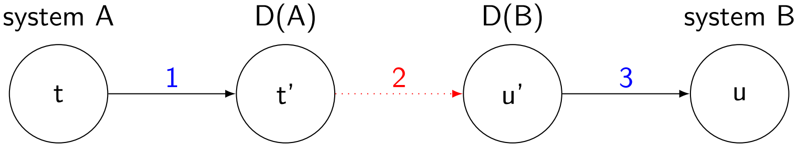
In a logical framework D:



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:

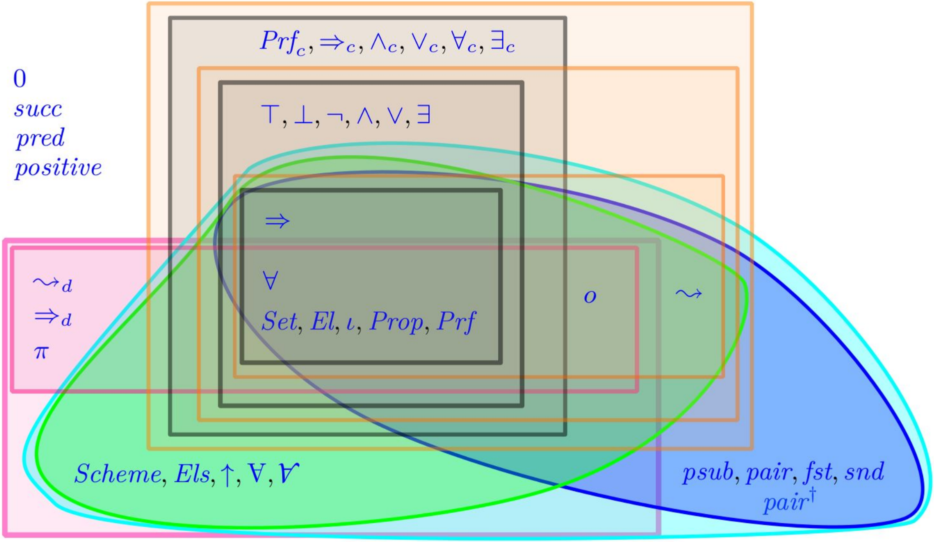


1. translate $t \in A$ in $t' \in D(A)$
2. identify the axioms and deduction rules of A used in t'
   translate $t' \in D(A)$ in $u' \in D(B)$ if possible
3. translate $u' \in D(B)$ in $u \in B$

⇒ represent in the same way functionalities common to A and B

The modular $\lambda\Pi/\mathcal{R}$ theory U and its sub-theories

38 symbols, 28 rules, 13 sub-theories



$Prf_c, \Rightarrow_c, \wedge_c, \vee_c, \forall_c, \exists_c$

$\top, \bot, \neg, \wedge, \vee, \exists$

$\Rightarrow$

$\forall$

$Set, El, \iota, Prop, Prf$

o

$\leadsto$

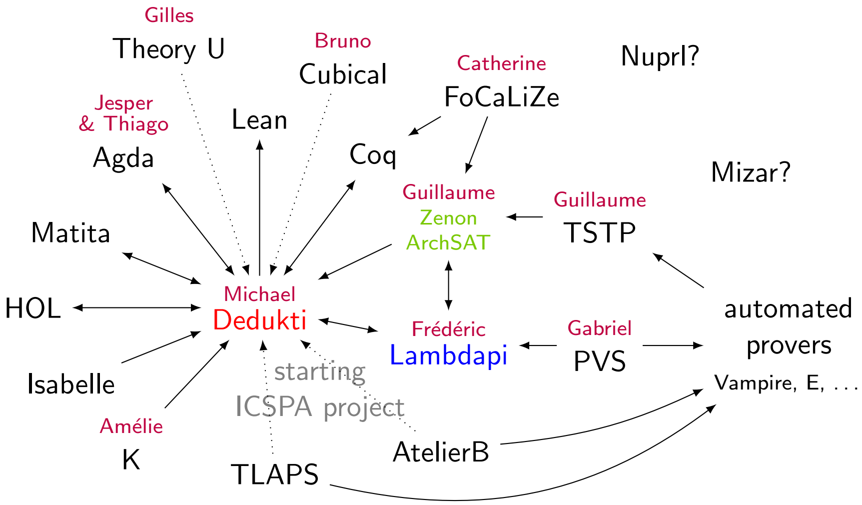
0
$succ$
$pred$
$positive$

$\leadsto_d$
$\Rightarrow_d$
π

$Scheme, Els, \uparrow, \forall, \rotatebox[origin=c]{180}{$\mathcal{A}$}$

$psub, pair, fst, snd$
$pair^\dagger$

Dedukti, an assembly language for proof systems



Libraries currently available in Dedukti

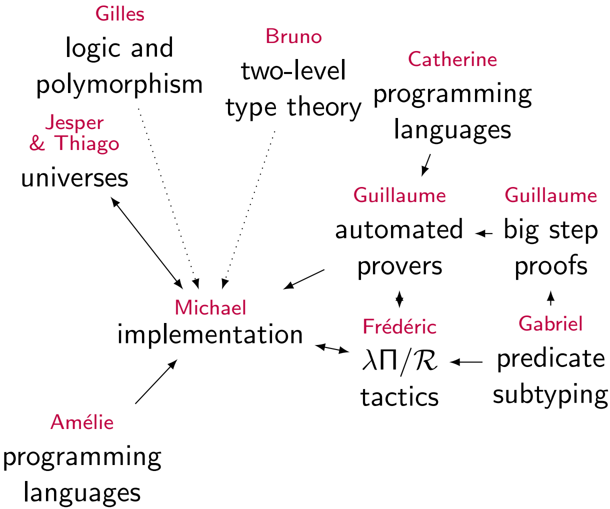
System	Libraries
HOL-Light	OpenTheory
Matita	Arith
Coq	Stdlib parts, GeoCoq
Isabelle	HOL.Complex_Main NEW! (*AFP soon?*)
Agda	Stdlib parts (± 25%)
PVS	Stdlib parts
TPTP	E 69%, Vampire 83%

Case study:

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

`http://logipedia.inria.fr`

Functionalities



Gilles

logic and polymorphism

Bruno

two-level type theory

Catherine

programming languages

Jesper & Thiago

universes

Guillaume

automated provers

Guillaume

big step proofs

Michael

implementation

Frédéric

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

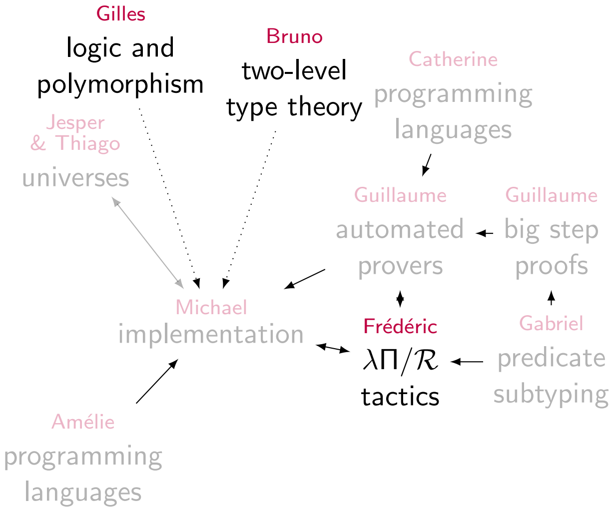
Gabriel

predicate subtyping

Amélie

programming languages

Today



Gilles
logic and polymorphism

Bruno
two-level type theory

Catherine
programming languages

Jesper & Thiago
universes

Guillaume
automated provers

Guillaume
big step proofs

Michael
implementation

Frédéric
$\lambda\Pi/\mathcal{R}$ tactics

Gabriel
predicate subtyping

Amélie
programming languages

Outline

Introduction to proof system interoperability

$\lambda\Pi$-calculus modulo rewriting

Dedukti language

Lambdapi tool

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

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

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

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

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

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

terms $t, u =$

TYPE	sort of types
f	global constant
x	local variable
tu	application
$\lambda x : t, u$	abstraction
$\Pi x : t, u$	dependent product
$t \to u$	abbreviation for $\Pi x : t, u$ when $x \notin u$

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

theory $=$

Σ	sequence of type declarations for global constants
$+ \mathcal{R}$	set of rewrite rules $l \hookrightarrow r$ including rules on types !

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

theory $=$

Σ sequence of type declarations for global constants

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

including rules on types !

typing $= \ldots +$

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A, B : \textsc{type}}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B}$$

Γ: types of local variables

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

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

$\equiv_{\beta\mathcal{R}}$: equational theory generated by β and $\mathcal{R}$

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

concat 2 : $\text{array } 2 \to \Pi q : \mathbb{N}, \text{array } q \to \text{array}(2+q)$

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

$\lambda\Pi$-calculus modulo rewriting

Dedukti language

Lambdapi tool

What is Dedukti?

Dedukti is a concrete language for defining $\lambda\Pi/\mathcal{R}$ 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 $\lambda\Pi/\mathcal{R}$ 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
```

Dedukti syntax

BNF grammar:

`https://github.com/Deducteam/Dedukti/blob/master/syntax.bnf`

file extension: `.dk`

comments: *`(; ... (; ... ;) ... ;)`*

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

Terms

`Type`	sort for types
id	variable or constant
id`.`*id*	constant from another file
term term . . . *term*	application
id [: *term*] => *term*	abstraction
[*id* :] *term* -> *term*	[dependent] product
(*term*)	

Command for declaring/defining a symbol

_modifier_* _id_ _param_* : _term_ [:= _term_] .

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

```
N : Type.
0 : N.
s : N -> N.
def add : N -> N -> N.

thm add_com :
  x:N -> y:N -> Eq (add x y) (add y x) := ...
```

Command for declaring rewrite rules

`[` *id* * `]` (*term* --> *term*)$^+$.

```
[x y]
x + 0 --> x
x + s y --> 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 → 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:

```
M-x package-install RET lambdapi-mode
```

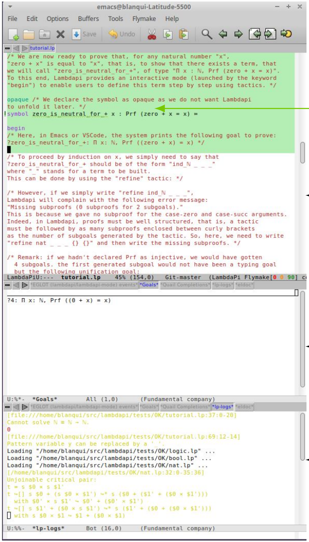
3. From sources:

```
make install_emacs
```

+ add in `~/.emacs`:

```
(load "lambdapi-site-file")
```

Emacs interface



```
/* We are now ready to prove that, for any natural number "x",
"zero + x" is equal to "x", that is, to show that there exists a term, that
we will call "zero_is_neutral_for_+", of type "Π x : ℕ, Prf (zero + x = x)".
To this end, Lambdapi provides an interactive mode (launched by the keyword
"begin") to enable users to define this term step by step using tactics. */

opaque /* We declare the symbol as opaque as we do not want Lambdapi
to unfold it later. */
!symbol zero_is_neutral_for_+ x : Prf (zero + x = x) ≔

begin
/* Here, in Emacs or VSCode, the system prints the following goal to prove:
?zero_is_neutral_for_+: Π x: ℕ, Prf ((zero + x) = x) */

/* To proceed by induction on x, we simply need to say that
?zero_is_neutral_for_+ should be of the form "ind_ℕ _ _ _"
where "_" stands for a term to be built.
This can be done by using the "refine" tactic: */

/* However, if we simply write "refine ind_ℕ _ _ _",
Lambdapi will complain with the following error message:
"Missing subproofs (0 subproofs for 2 subgoals)."
This is because we gave no subproof for the case-zero and case-succ arguments.
Indeed, in Lambdapi, proofs must be well structured, that is, a tactic
must be followed by as many subproofs enclosed between curly brackets
as the number of subgoals generated by the tactic. So, here, we need to write
"refine nat _ _ _ {} {}" and then write the missing subproofs. */

/* Remark: if we hadn't declared Prf as injective, we would have gotten
  4 subgoals. the first generated subgoal would not have been a typing goal
  but the following unification goal:
LambdaPiU:---  tutorial.lp    45% (154,0)   Git-master  (LambdaPi Flymake[0 0 90] c
```

```
?4: Π x: ℕ, Prf ((0 + x) = x)
U:%*-  *Goals*       All (1,0)       (Fundamental company)
```

```
[file:///home/blanqui/src/lambdapi/tests/OK/tutorial.lp:37:0-20]
Cannot solve ℕ ≡ ℕ → ℕ.
0
[file:///home/blanqui/src/lambdapi/tests/OK/tutorial.lp:69:12-14]
Pattern variable y can be replaced by a '_'.
Loading "/home/blanqui/src/lambdapi/tests/OK/logic.lp" ...
Loading "/home/blanqui/src/lambdapi/tests/OK/bool.lp" ...
Loading "/home/blanqui/src/lambdapi/tests/OK/nat.lp" ...
[/home/blanqui/src/lambdapi/tests/OK/nat.lp:32:0-35:36]
Unjoinable critical pair:
t ≔ s $0 × s $1'
t ↪[] s $0 + (s $0 × $1') ↪* s ($0 + ($1' + ($0 × $1')))
  with $0' × s $1' ↪ $0' + ($0' × $1')
t ↪[] s $1' + ($0 × s $1') ↪* s ($1' + ($0 + ($0 × $1')))
  with s $0 × $1 ↪ $1 + ($0 × $1)
U:%%-  *lp-logs*     Bot (16,0)      (Fundamental company)
```

checked part

edition buffer

goals

messages

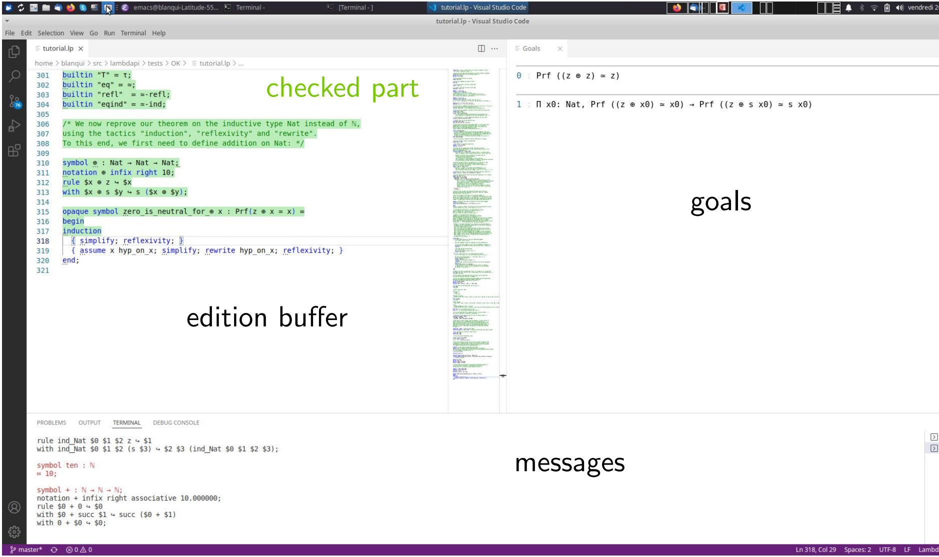
window layout
can be customized

shortcuts: `https://lambdapi.readthedocs.io/en/latest/emacs.html`

How to install the VSCode interface?

From the VSCode Marketplace

VSCode interface



```
builtin "T" ≔ τ;
builtin "eq" ≔ ≃;
builtin "refl"  ≔ ≃-refl;
builtin "eqind" ≔ ≃-ind;

/* We now reprove our theorem on the inductive type Nat instead of ℕ,
using the tactics "induction", "reflexivity" and "rewrite".
To this end, we first need to define addition on Nat: */

symbol ⊕ : Nat → Nat → Nat;
notation ⊕ infix right 10;
rule $x ⊕ z ↪ $x
with $x ⊕ s $y ↪ s ($x ⊕ $y);

opaque symbol zero_is_neutral_for_⊕ x : Prf(z ⊕ x ≃ x) ≔
begin
induction
  { simplify; reflexivity; }
  { assume x hyp_on_x; simplify; rewrite hyp_on_x; reflexivity; }
end;
```

```
0 : Prf ((z ⊕ z) ≃ z)

1 : Π x0: Nat, Prf ((z ⊕ x0) ≃ x0) → Prf ((z ⊕ s x0) ≃ s x0)
```

```
rule ind_Nat $0 $1 $2 z ↪ $1
with ind_Nat $0 $1 $2 (s $3) ↪ $2 $3 (ind_Nat $0 $1 $2 $3);

symbol ten : ℕ
≔ 10;

symbol + : ℕ → ℕ → ℕ;
notation + infix right associative 10.000000;
rule $0 + 0 ↪ $0
with $0 + succ $1 ↪ succ ($0 + $1)
with 0 + $0 ↪ $0;
```

File `lambdapi.pkg`

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`:

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

`file1.lp`:

```
symbol A : TYPE;
```

`file2.lp`:

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

`file3.lp`:

```
require open nat.unary.file1 nat.unary.file2;
symbol b ≔ a;
```

Lambdapi syntax

BNF grammar:

`https://raw.githubusercontent.com/Deducteam/lambdapi/master/doc/lambdapi.bnf`

file extension: `.lp`

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

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

Terms

`TYPE`	sort for types
(_id_ .)*_id_	variable or constant
term *term* . . . *term*	application
λ *id* [: *term*] , *term*	abstraction
Π *id* [: *term*] , *term*	dependent product
term → *term*	non-dependent product
`_`	unknown term
`let` *id* [: *term*] ≔ *term* `in` *term*	
(*term*)	

Command for declaring/defining a symbol

$$\mathit{modifier}^*\ \texttt{symbol}\ \mathit{id}\ \mathit{param}^*\ [:\ \mathit{term}\ ]\ [\coloneqq\ \mathit{term}\ ]\ [\texttt{begin}\ \mathit{proof}\ \texttt{end}]\ ;$$

$$\mathit{param} = \mathit{id} \mid \_ \mid (\ \mathit{id}^{+} : \mathit{term}\ ) \mid \underset{\textit{implicit parameters}}{[\ \mathit{id}^{+} : \mathit{term}\ ]}$$

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 → N;

symbol + : N → N → N; notation + infix right 10;

symbol × : N → N → N; notation × 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

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

matching on defined symbols

```
rule ($x + $y) + $z ↪ $x + ($y + $z);
```

non-linear patterns

```
rule $x - $x ↪ 0;
```

Lambdapi tries to automatically check:

local confluence (AC symbols/HO patterns not handled yet)

Higher-order pattern-matching

```
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 · $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 ∉ : N → L → Prop; notation ∉ infix 20;

constant symbol nil : L;
constant symbol cons x l : Prf(x ∉ l) → L;

rule _ ∉ nil ↪ ⊤
with $x ∉ cons $y $l _ ↪ $x ≠ $y ∧ $x ∉ $l;
```

Command for generating induction principles

```
inductive ℕ : TYPE ≔ 0 : ℕ | s : ℕ → ℕ;
```

is equivalent to:

```
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

```
/* 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 *id* ;

type *term* ;

compute *term* ;

(assert | assertnot) *id* * ⊢ *term* (: | ≡) *term* ;

```
print N; // constructors and induction principle
print +; // type and rules

type ×;
compute 2 × 5;

assert 0 : N;
assertnot 0 : N → N;

assert x y z ⊢ x + y × z ≡ x + (y × z);
assertnot x y z ⊢ x + y × z ≡ (x + y) × 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
  ... // proof script
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:**

$$\begin{aligned} \mathit{proof} &::= (\mathit{proof\_step}\ ;)^* \\ \mathit{proof\_step} &::= \mathit{tactic}\ (\texttt{\{}\ \mathit{proof}\ \texttt{\}})^* \end{aligned}$$

- 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

`https://raw.githubusercontent.com/Deducteam/lambdapi/master/tests/OK/tutorial.lp`

```
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* like Coq SSReflect
- `why3` 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