EuroProofNet

Introduction to Proof System Interoperability

Frédéric Blanqui

Deduc⊢eam

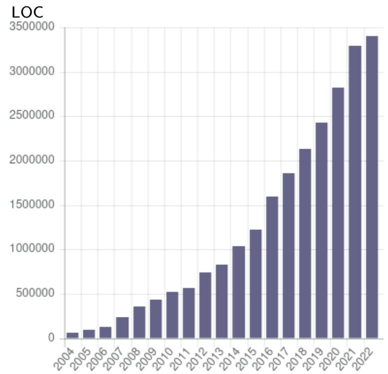
September 2022

Outline

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
...	...	...

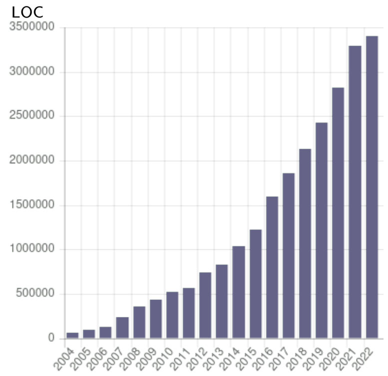
* type, definition, theorem, ...



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, ...

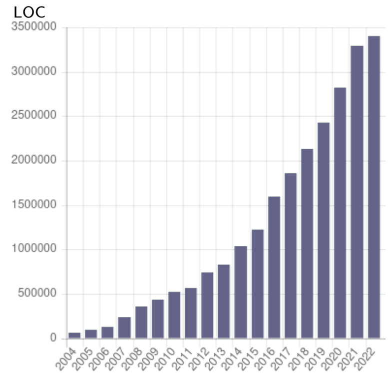


- Every system has basic libraries on integers, lists, ...
- Some definitions/theorems are available in one system only

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

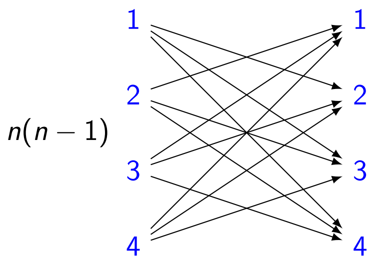
- Avoid duplicating developments and losing time
- Facilitate development of new proof systems
- Increase reliability of formal proofs (cross-checking)
- Facilitate validation by certification authorities
- Relativize the choice of a system (school, industry)
- Provide multi-system data to machine learning

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)

Difficulties of interoperability

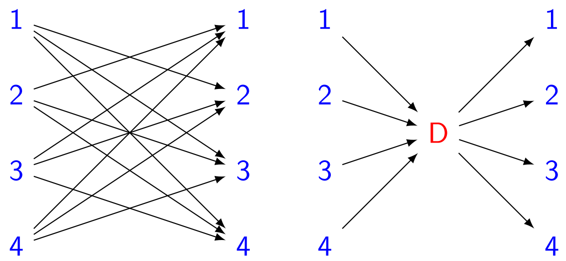
- Each system is based on different axioms and deduction rules
- It is usually non trivial and sometimes impossible to translate a proof from one system to the other (e.g. a classical proof in an intuitionistic system)
- Is it reasonable to have $n(n-1)$ translators for n systems?



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?

$n(n-1)$



$2n$

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

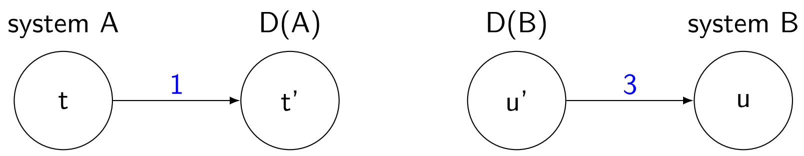
allows one to represent S=geometry, S=arithmetic, S=set theory, ...
not well suited for functional computations and dependent types

Better: D = $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$)

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:

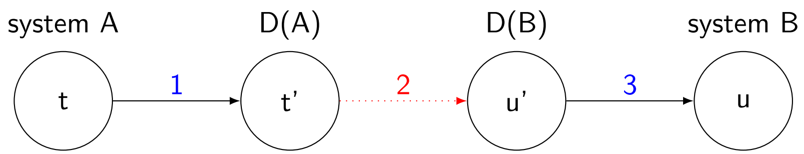


1. translate $t \in A$ in $t' \in D(A)$

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

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

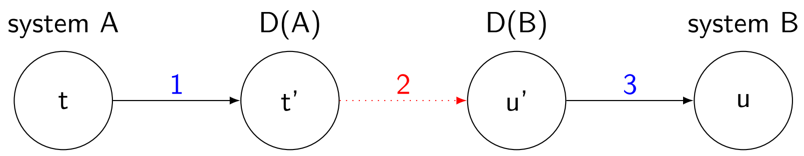
In a logical framework D:



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:

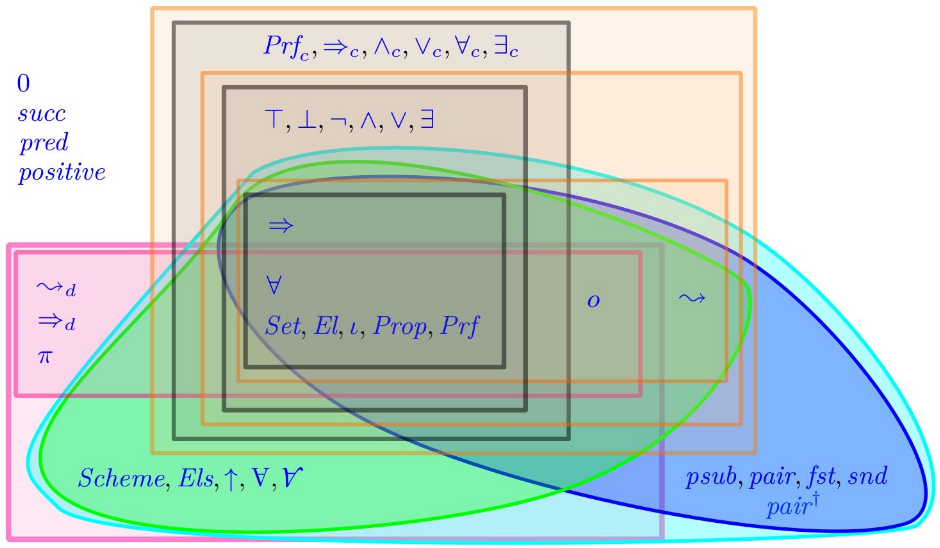


1. translate $t \in A$ in $t' \in D(A)$
2. identify the axioms and deduction rules of A used in t'
   translate $t' \in D(A)$ in $u' \in D(B)$ if possible
3. translate $u' \in D(B)$ in $u \in B$

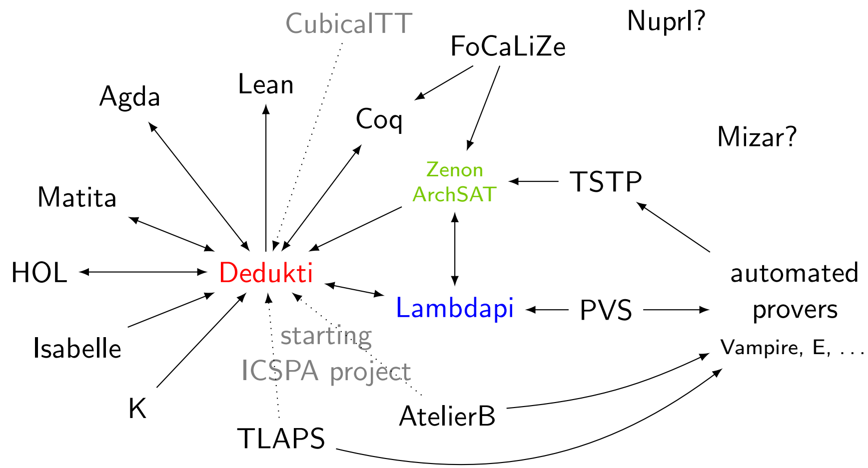
$\Rightarrow$ represent in the same way functionalities common to A and B

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

38 symbols, 28 rules, 13 sub-theories



Dedukti, an assembly language for proof systems implementing $\lambda\Pi/\mathcal{R}$



Libraries currently available in Dedukti

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

Case study:

Matita/Arith $\longrightarrow$ OpenTheory, Coq, PVS, Lean, Agda

http://logipedia.inria.fr

Outline

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 λ-calculus?

introduced by Alonzo Church in 1932

the (untyped or pure) λ-calculus is a general framework for defining functional terms (objects or propositions)

initially thought as a possible foundation for logic
but turned out to be inconsistent

it however provided a foundation for computability theory
and functional programming !

What is λ-calculus?

only 3 constructions:

- **variables** $x, y, \ldots$
- **application** of a term t to another term u, written tu
- **abstraction** over a variable x in a term t, written $\lambda x, t$

example: the function mapping x to $2x + 1$ is written

$$\lambda x, +(*2x)1$$

α-equivalence

the names of abstracted variables are theoretically not significant:

$$\lambda x, +(*2x)1 \quad \text{denotes the same function as} \quad \lambda y, +(*2y)1$$

terms equivalent modulo valid renamings are said α-equivalent

in theory, one usually works modulo α-equivalence, that is, on α-equivalence classes of terms (hence, one can always rename some abstracted variables if it is more convenient)

$\Rightarrow$ but, then, one has to be careful that functions and relations are actually invariant by α-equivalence!...

in practice, dealing with α-equivalence is not trivial

$\Rightarrow$ this gave raise to a lot of research and tools (still nowdays)!

Example: the set of free variables

a variable is free if it is not abstracted

the set $\mathrm{FV}(t)$ of free variables of a term t is defined as follows:

- $\mathrm{FV}(x) = \{x\}$
- $\mathrm{FV}(tu) = \mathrm{FV}(t) \cup \mathrm{FV}(u)$
- $\mathrm{FV}(\lambda x, t) = \mathrm{FV}(t) - \{x\}$

one can check that FV is invariant by α-equivalence:

$$\text{if } t =_\alpha u \text{ then } \mathrm{FV}(t) = \mathrm{FV}(u)$$

Substitution

a substitution is a finite map from variables to terms

$$\sigma = \{(x_1, t_1), \dots, (x_n, t_n)\}$$

the domain of a substitution σ is

$$\mathrm{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$$

how to define the result of applying a substitution σ on a term t?

- $x\sigma = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$
- $x\sigma = x$ if $x \notin \mathrm{dom}(\sigma)$
- $(tu)\sigma = (t\sigma)(u\sigma)$
- $(\lambda x, t)\sigma = \lambda x, (t\sigma)$? example: $(\lambda x, y)\{(y, x)\} = \lambda x, x$?

Substitution

a substitution is a finite map from variables to terms

$$\sigma = \{(x_1, t_1), \dots, (x_n, t_n)\}$$

the domain of a substitution σ is

$$\mathrm{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$$

how to define the result of applying a substitution σ on a term t?

- $x\sigma = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$
- $x\sigma = x$ if $x \notin \mathrm{dom}(\sigma)$
- $(tu)\sigma = (t\sigma)(u\sigma)$
- $(\lambda x, t)\sigma = \lambda x, (t\sigma)$? example: $(\lambda x, y)\{(y, x)\} = \lambda x, x$?

definition not invariant by α-equivalence ! $\lambda x, y =_\alpha \lambda z, y$

Substitution

in λ-calculus, substitution is not trivial!

we must rename abstracted variables to avoid name clashes:

$$(\lambda x, t)\sigma = \lambda y, (t\sigma')$$

where $\sigma' = \sigma|_V \cup \{(x, y)\}$, $V = \mathrm{FV}(\lambda x, t)$ and $y \notin V$

Operational semantics: β-reduction

applying the term $\lambda x, +(*2x)1$ to 3 should return 7

this is the top β-rewrite relation:

$$(\lambda x, t)u \rightarrow_{\beta}^{\varepsilon} t\{(x, u)\}$$

the β-rewrite relation $\rightarrow_{\beta}$ is the closure by context of $\rightarrow_{\beta}^{\varepsilon}$:

$$\frac{t \rightarrow_{\beta}^{\varepsilon} u}{t \rightarrow_{\beta} u} \quad \frac{t \rightarrow_{\beta} u}{tv \rightarrow_{\beta} uv} \quad \frac{t \rightarrow_{\beta} u}{vt \rightarrow_{\beta} vu} \quad \frac{t \rightarrow_{\beta} u}{\lambda x, t \rightarrow_{\beta} \lambda x, u}$$

let $\simeq_{\beta}$ be the smallest equivalence relation containing $\rightarrow_{\beta}$

Properties of β-reduction in pure λ-calculus

$\rightarrow_\beta$ is confluent:

if $t \hookrightarrow^*_\beta u$ and $t \hookrightarrow^*_\beta v$,
then there is w s.t.
$u \hookrightarrow^*_\beta w$ and $v \hookrightarrow^*_\beta w$

$$\begin{tikzcd}
& \forall t \arrow[dl, dotted, twoheadrightarrow] \arrow[dr, dotted, twoheadrightarrow] & \\
\forall u \arrow[dr, dotted, twoheadrightarrow] & & \forall v \arrow[dl, dotted, twoheadrightarrow] \\
& \exists w &
\end{tikzcd}$$

this means that the order of reduction steps does not matter

and every term has <u>at most</u> one normal form

Properties of β-reduction in pure λ-calculus

$\rightarrow_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \rightarrow_\beta (\lambda x, xx)(\lambda x, xx)$$

Properties of β-reduction in pure λ-calculus

$\rightarrow_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \rightarrow_\beta (\lambda x, xx)(\lambda x, xx)$$

every term t has a fixpoint $Y_t \coloneqq (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \rightarrow_\beta tY_t$$

Properties of β-reduction in pure λ-calculus

$\to_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \to_\beta (\lambda x, xx)(\lambda x, xx)$$

every term t has a fixpoint $Y_t \coloneqq (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \to_\beta tY_t$$

λ-calculus is Turing-complete/can encode any recursive function

Properties of β-reduction in pure λ-calculus

$\to_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \to_\beta (\lambda x, xx)(\lambda x, xx)$$

every term t has a fixpoint $Y_t := (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \to_\beta tY_t$$

λ-calculus is Turing-complete/can encode any recursive function

a natural number n can be encoded as

$$\lambda f, \lambda x, f^n x$$

where $f^0 x = x$ and $f^{n+1} x = f(f^n x)$

On the origin of type theory

like in unrestricted set theory where every term is a set
in pure λ-calculus, every term is a function
$\Rightarrow$ every term can be applied to another term, including itself!

On the origin of type theory

like in unrestricted set theory where every term is a set
in pure λ-calculus, every term is a function
$\Rightarrow$ every term can be applied to another term, including itself!

Russell's paradox: with $R := \{x \mid x \notin x\}$ we have $R \in R$ and $R \notin R$
λ-calculus: with $R := \lambda x, \neg(xx)$ we have $RR \to_\beta \neg(RR)$

On the origin of type theory

like in unrestricted set theory where every term is a set
in pure λ-calculus, every term is a function
$\Rightarrow$ every term can be applied to another term, including itself!

Russell's paradox: with $R := \{x \mid x \notin x\}$ we have $R \in R$ and $R \notin R$
λ-calculus: with $R := \lambda x, \neg(xx)$ we have $RR \to_\beta \neg(RR)$

proposals to overcome this problem:

- restrict comprehension axiom to already defined sets
  use $\{x \in A \mid P\}$ instead of $\{x \mid P\}$

  $\leadsto$ modern set theory

On the origin of type theory

like in unrestricted set theory where every term is a set
in pure λ-calculus, every term is a function
$\Rightarrow$ every term can be applied to another term, including itself!

Russell's paradox: with $R := \{x \mid x \notin x\}$ we have $R \in R$ and $R \notin R$
λ-calculus: with $R := \lambda x, \neg(xx)$ we have $RR \to_\beta \neg(RR)$

proposals to overcome this problem:

- restrict comprehension axiom to already defined sets
  use $\{x \in A \mid P\}$ instead of $\{x \mid P\}$

  $\rightsquigarrow$ modern set theory

- organize terms into a hierarchy
  - natural numbers are of type ι and propositions of type o
  - unary predicates/sets of natural numbers are of type $\iota \to o$
  - sets of sets of natural numbers are of type $(\iota \to o) \to o$
  - ...

  $\rightsquigarrow$ modern type theory

Church simply-typed λ-calculus

simple types:

$$A, B \coloneqq X \in \mathcal{V}_{typ} \mid A \to B$$

- X is a user-defined type variable
- $A \to B$ is the type of functions from A to B

raw terms:

$$t, u \coloneqq x \in \mathcal{V}_{obj} \mid tu \mid \lambda x : A, t$$

Well-typed terms

a typing environment Γ is a finite map from variables to types

typing rules for terms:

$$\frac{(x, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \cup \{(x, A)\} \vdash t : B \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x : A, t : A \to B}$$

- xx is not typable anymore
- $\to_\beta$ terminates on well-typed terms
- $\to_\beta$ preserves typing: if $\Gamma \vdash t : A$ and $t \to_\beta u$, then $\Gamma \vdash u : A$

Dependent types / $\lambda\Pi$-calculus

a dependent type is a type that depends on terms

example: type ($\texttt{Array}\ n$) of arrays of size n

first introduced by de Bruijn in the Automath system in the 60's

types:

$$A, B \coloneqq X\ t_1 \dots t_n \mid \Pi x : A, B$$

$A \to B$ is an abbreviation for $\Pi x : A, B$ when $x \notin \mathrm{FV}(B)$

example: concatenation function on arrays

$$\texttt{concat} : \Pi p : \mathbb{N}, \texttt{Array}\ p \to \Pi q : \mathbb{N}, \texttt{Array}\ q \to \texttt{Array}(p + q)$$

Dependent types / $\lambda\Pi$-calculus

Harper,Honsell&Plotkin distinguish 4 syntactic classes for terms:

name	definition	type
	$\texttt{KIND}$	
kinds K	$\texttt{TYPE} \mid \Pi x : A, K$	$\texttt{KIND}$
families A	$X \mid At \mid \Pi x : A, A \mid \lambda x : A, A$	kinds
objects t	$x \mid tt \mid \lambda x : A, t$	families

this can be summarized as follows:

$$''t : A : K : \texttt{KIND}''$$

kinds describe the types of families; they are of the form:

$$\Pi x_1 : A_1, \ldots, \Pi x_n : A_n, \texttt{TYPE}$$

a family is like a function returning a type:

$$(\lambda n : \mathbb{N}, \texttt{Array}\, n)\, 2 \hookrightarrow_\beta \texttt{Array}\, 2$$

Typing rules for typing environments

because types depend on terms, we now need typing rules for types!

a typing environnment is now a *sequence* of type declarations

$$\Gamma := \emptyset \mid \Gamma, x : A \mid \Gamma, X : K$$

"$\Gamma \vdash$" means that Γ is a well-typed environment:

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A : \texttt{TYPE} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash K : \texttt{KIND} \quad X \notin \mathrm{dom}(\Gamma)}{\Gamma, X : K \vdash}$$

Signatures Σ

a typing environment can be split in two parts:

1. a fixed part Σ representing global constants
2. a variable part Γ for local variables

Typing rules for kinds and families

kinds:

$$\frac{\Gamma \vdash}{\Gamma \vdash \texttt{TYPE} : \texttt{KIND}} \quad \frac{\Gamma, x : A \vdash K : \texttt{KIND}}{\Gamma \vdash \Pi x : A, K : \texttt{KIND}}$$

families:

$$\frac{\Gamma \vdash \quad (X, K) \in \Gamma}{\Gamma \vdash X : K} \quad \frac{\Gamma, x : A \vdash B : \texttt{TYPE}}{\Gamma \vdash \Pi x : A, B : \texttt{TYPE}}$$

$$\frac{\Gamma, x : A \vdash B : K}{\Gamma \vdash \lambda x : A, B : \Pi x : A, K} \quad \frac{\Gamma \vdash A : \Pi x : B, K \quad \Gamma \vdash t : B}{\Gamma \vdash At : K\{(x, t)\}}$$

$$\frac{\Gamma \vdash A : K \quad K \simeq_\beta K' \quad \Gamma \vdash K' : \texttt{KIND}}{\Gamma \vdash A : K'}$$

Typing rules for objects

$$\frac{\Gamma \vdash \quad (x, A) \in \Gamma}{\Gamma \vdash x : A}$$

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

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

$$\frac{\Gamma \vdash t : A \quad A \simeq_\beta A' \quad \Gamma \vdash A' : \mathtt{TYPE}}{\Gamma \vdash t : A'}$$

Properties of the $\lambda\Pi$-calculus

- types are equivalent: if $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $A \simeq_\beta B$
- $\hookrightarrow_\beta$ terminates on well-typed terms
- $\hookrightarrow_\beta$ preserves typing
- type-inference $\exists A, \Gamma \vdash t : A$? is decidable
- type-checking $\Gamma \vdash t : A$? is decidable

PTS presentation of $\lambda\Pi$ (Barendregt)

terms and types:

$$t := x \mid tt \mid \lambda x : t, t \mid \Pi x : t, t \mid s \in \mathcal{S} = \{\texttt{TYPE}, \texttt{KIND}\}$$

typing rules:

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash \quad (x, A) \in \Gamma}{\Gamma \vdash x : A}$$

$$(sort)\ \frac{\Gamma \vdash}{\Gamma \vdash \texttt{TYPE} : \texttt{KIND}} \quad (prod)\ \frac{\Gamma \vdash A : \texttt{TYPE} \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A, B : s}$$

$$\frac{\Gamma, x : A \vdash t : B \quad \Gamma \vdash \Pi x : A, B : s}{\Gamma \vdash \lambda x : A, t : \Pi x : A, B} \quad \frac{\Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B\{(x, u)\}}$$

$$\frac{\Gamma \vdash t : A \quad A \simeq_\beta A' \quad \Gamma \vdash A' : s}{\Gamma \vdash t : A'}$$

Pure Type Systems (PTS)

$$(sort)\ \frac{\Gamma \vdash}{\Gamma \vdash \mathtt{TYPE} : \mathtt{KIND}} \quad (prod)\ \frac{\Gamma \vdash A : \mathtt{TYPE} \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A, B : s}$$

the rules (*sort*) and (*prod*) can be generalized as follows:

$$(sort)\ \frac{\Gamma \vdash \quad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}$$

$$(prod)\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad ((s_1, s_2), s_3) \in \mathcal{P}}{\Gamma \vdash \Pi x : A, B : s_3}$$

where:

- $\mathcal{S}$ is an arbitrary set of sorts
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ describes the types of sorts
- $\mathcal{P} \subseteq \mathcal{S}^2 \times \mathcal{S}$ describes the allowed products

Pure Type Systems (PTS)

many well-known type systems can be described as PTSs

examples with $\mathcal{S} = \{\mathtt{TYPE}, \mathtt{KIND}\}$ and $\mathcal{A} = \{(\mathtt{TYPE}, \mathtt{KIND})\}$:

feature	product rule in $\mathcal{P}$
simple types	$\mathtt{TYPE}, \mathtt{TYPE}, \mathtt{TYPE}$
polymorphic types	$\mathtt{KIND}, \mathtt{TYPE}, \mathtt{TYPE}$
dependent types	$\mathtt{TYPE}, \mathtt{KIND}, \mathtt{KIND}$
type constructors	$\mathtt{KIND}, \mathtt{KIND}, \mathtt{KIND}$

the combination of all these rules is the calculus of constructions

remark: a PTS is functional if $\mathcal{A}$ and $\mathcal{P}$ are functions (e.g. CoC)
then types are unique modulo $\simeq_\beta$

Universes

- a universe U is a type closed by exponentiation

$$\frac{A : U \quad B : U}{A \to B : U}$$

  example: the sort $\mathtt{TYPE}$ of the simple types ι, $\iota \to o$, ...

- universes are like inaccessible cardinals in set theory:
  - an inaccessible cardinal is closed by set exponentiation
  - a universe is closed by type exponentiation

More universes

- some math. constructions quantifies over the elements of U_0
  $\Rightarrow$ they need to inhabit a new universe U_1 containing U_0

- by iteration we get an infinite sequence of nested universes

$$U_0 : U_1 : \ldots U_i : U_{i+1} \ldots \qquad \frac{A : U_i \quad B : U_j}{A \to B : U_{\mathsf{max}(i,j)}}$$

  available in some proof assistants like Coq, Agda, Lean

- PTS representation:
  $\mathcal{S} = \{\mathrm{TYPE}_i \mid i \in \mathbb{N}\}$
  $\mathcal{A} = \{(\mathrm{TYPE}_i, \mathrm{TYPE}_{i+1}) \mid i \in \mathbb{N}\}$
  $\mathcal{P} = \{(\mathrm{TYPE}_i, \mathrm{TYPE}_j, \mathrm{TYPE}_{\mathsf{max}(i,j)}) \mid i, j \in \mathbb{N}\}$

What is rewriting?

introduced at the end of the 60's (Knuth)

a rewrite rule $l \hookrightarrow r$ is an equation $l = r$ used from left-to-right

rewriting simply consists in repeatedly replacing a subterm $l\sigma$ by $r\sigma$
(rewriting is Turing-complete)

it can be used to decide equational theories:

> given a set $\mathcal{E}$ of equations, $\simeq_{\mathcal{E}}$ is decidable
> if there is a rewrite system $\mathcal{R}$ such that:
>
> - $\hookrightarrow_{\mathcal{R}}$ terminates
> - $\hookrightarrow_{\mathcal{R}}$ is confluent
> - $\simeq_{\mathcal{R}} = \simeq_{\mathcal{E}}$
>
> where $\hookrightarrow_{\mathcal{R}}$ is the closure by context of $\mathcal{R}$

$\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$)

a theory in the $\lambda\Pi$-calculus modulo rewriting is given by

- a signature Σ
- a set $\mathcal{R}$ of rewrite rules on Σ

such that:

- $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ terminates
- $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ is confluent
- every rule $l \hookrightarrow r$ preserves typing: if $\Gamma \vdash l\sigma : A$ then $\Gamma \vdash r\sigma : A$

Outline

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 algorithm

```
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

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?

Using Opam:

```
opam install lambdapi
```

Compilation from the sources:

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

How to use Lambdapi?

Command line (batch mode):

```
lambdapi check file.lp
```

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
```

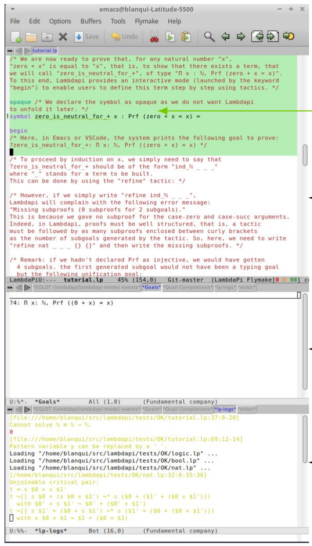
3. From sources:

```
make install_emacs
```

+ add in `~/.emacs`:

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

Emacs interface



checked part

edition buffer

goals

messages

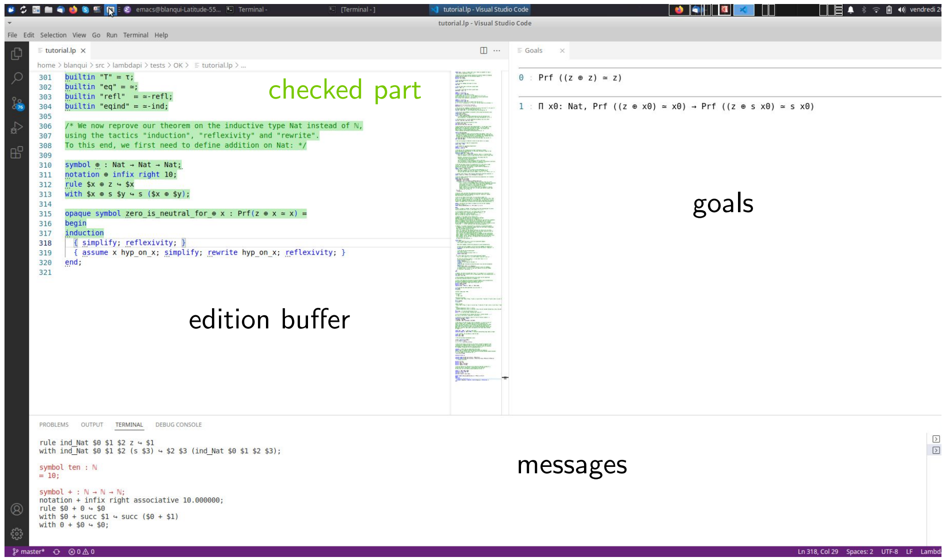
window layout
can be customized

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

How to install the VSCode interface?

From the VSCode Marketplace

VSCode interface



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 $\to$ *term*	non-dependent product
(*term*)	
_	unknown term
`let` *id* [: *term*] $:=$ *term* `in` *term*	

Command for declaring/defining a symbol

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

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

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

$$\texttt{rule}\ \textit{term} \hookrightarrow \textit{term}\ (\texttt{with}\ \textit{term} \hookrightarrow \textit{term}\ )^*\ ;$$

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;
```

Queries and assertions

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

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

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-mon 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:

$$\begin{array}{r}
\mathit{proof} ::= (\mathit{proof_step}\ \texttt{;})^* \\
\mathit{proof_step} ::= \mathit{tactic}\ (\texttt{\{}\ \mathit{proof}\ \texttt{\}})^*
\end{array}$$

- 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
    {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 prover

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

(currently for strictly positive parametric inductive types only)

```
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)
```

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;
```

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

Exercise for next lecture

- install `https://github.com/Deducteam/lambdapi`
- have a look at `https://lambdapi.readthedocs.io/`
- and the tutorial `tests/OK/tutorial.lp`