Introduction to Proof System Interoperability

Frédéric Blanqui

Deducl-eam

September 2022
Outline

Introduction

Lambda-Pi-calculus modulo rewriting
  Lambda-calculus
  Simple types
  Dependent types
  Pure Type Systems
  Rewriting

Dedukti language

Lambdapi proof assistant

Encoding logics in $\lambda\Pi/\beta$

Automated Theorem Provers
  Instrumenting provers for Dedukti proof production
  Reconstructing proofs
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, . . .

LOC

0

5000000

10000000

15000000

20000000

25000000

30000000

35000000

40000000

45000000

50000000

55000000

60000000

65000000

70000000

75000000

80000000

85000000

90000000

95000000

100000000

105000000

110000000

115000000

120000000

125000000

130000000

135000000

140000000

145000000

150000000

155000000

160000000

165000000

170000000

175000000

180000000

185000000

190000000

195000000

200000000
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).
**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?

\[
\begin{array}{c}
\text{1} \\
\text{2} \\
\text{3} \\
\text{4} \\
n(n - 1)
\end{array}
\quad
\begin{array}{c}
\text{1} \\
\text{2} \\
\text{3} \\
\text{4} \\
2n
\end{array}
\]
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: $D = \text{predicate calculus}$
allows one to represent $S=\text{geometry}, S=\text{arithmetic}, S=\text{set theory}, \ldots$
not well suited for functional computations and dependent types

Better: $D = \lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/R$)
allows one to represent also:
$S=\text{HOL}, S=\text{Coq}, S=\text{Agda}, S=\text{PVS}, \ldots$
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

<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 (AFP soon?)</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 $\rightarrow$ OpenTheory, Coq, PVS, Lean, Agda

http://logipedia.inria.fr
Outline

Introduction

Lambda-Pi-calculus modulo rewriting
  Lambda-calculus
  Simple types
  Dependent types
  Pure Type Systems
  Rewriting

Dedukti language

Lambdapi proof assistant

Encoding logics in $\lambda\Pi/\mathcal{R}$

Automated Theorem Provers
  Intrumenting provers for Dedukti proof production
  Reconstructing proofs
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 $l \leftrightarrow r$
What is $\lambda$-calculus?

introduced by Alonzo Church in 1932

the (untyped or pure) $\lambda$-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

$$\lambda x, (+2x)1$$
\textit{\(\alpha\)-equivalence}

the names of abstracted variables are theoretically not significant:

\[ \lambda x, +(\ast 2x)1 \quad \text{denotes the same function as} \quad \lambda y, +(\ast 2y)1 \]

terms equivalent modulo valid renamings are said \(\alpha\)-equivalent

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

\[ \Rightarrow \text{but, then, one has to be careful that functions and relations are actually invariant by } \alpha\text{-equivalence!} \ldots \]

in practice, dealing with \(\alpha\)-equivalence is not trivial

\[ \Rightarrow \text{this gave raise to a lot of research and tools (still nowdays)!} \]
Example: the set of free variables

A variable is free if it is not abstracted.

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

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

One can check that $\text{FV}$ is invariant by $\alpha$-equivalence:

If $t =_\alpha u$ then $\text{FV}(t) = \text{FV}(u)$
**Substitution**

a substitution is a finite map from variables to terms

\[ \sigma = \{(x_1, t_1), \ldots, (x_n, t_n)\} \]

the domain of a substitution \( \sigma \) is

\[ \text{dom}(\sigma) = \{x \in V \mid \sigma(x) \neq x\} \]

how to define the result of applying a substitution \( \sigma \) on a term \( t \)?

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

a substitution is a finite map from variables to terms

\[ \sigma = \{(x_1, t_1), \ldots, (x_n, t_n)\} \]

the domain of a substitution \( \sigma \) is

\[ \text{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\} \]

how to define the result of applying a substitution \( \sigma \) on a term \( t \)?

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

definition not invariant by \( \alpha \)-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 $\sigma' = \sigma|_V \cup \{(x, y)\}$, $V = \text{FV}(\lambda x, t)$ and $y \notin V$.
Operational semantics: $\beta$-reduction

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

this is the top $\beta$-rewrite relation:

$$(\lambda x, t)u \rightarrow^\varepsilon_{\beta} t\{(x, u)\}$$

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

$$
\begin{array}{cccc}
  t \rightarrow_{\beta}^{\varepsilon} u & t \rightarrow_{\beta} u & t \rightarrow_{\beta} u & t \rightarrow_{\beta} u \\
  t \rightarrow_{\beta} u & tv \rightarrow_{\beta} uv & vt \rightarrow_{\beta} vu & \lambda x, t \rightarrow_{\beta} \lambda x, u \\
\end{array}
$$

let $\simeq_{\beta}$ be the smallest equivalence relation containing $\rightarrow_{\beta}$
Properties of $\beta$-reduction in pure $\lambda$-calculus

$\rightarrow_{\beta}$ is confluent:

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

this means that the order of reduction steps does not matter

and every term has at most one normal form
Properties of $\beta$-reduction in pure $\lambda$-calculus

$\rightarrow_\beta$ does not terminate:

$$(\lambda x, xx)(\lambda x, xx) \rightarrow_\beta (\lambda x, xx)(\lambda x, xx)$$
Properties of $\beta$-reduction in pure $\lambda$-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 $Y_t := (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \rightarrow_\beta tY_t$$
Properties of $\beta$-reduction in pure $\lambda$-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 $Y_t := (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \rightarrow_\beta tY_t$$

$\lambda$-calculus is Turing-complete/can encode any recursive function
Properties of $\beta$-reduction in pure $\lambda$-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 $Y_t := (\lambda x, t(xx))(\lambda x, t(xx))$:

$$Y_t \rightarrow_\beta tY_t$$

$\lambda$-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 $\lambda$-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 \( \lambda \)-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 \( R \notin R \)
\( \lambda \)-calculus: with \( R := \lambda x, \neg(xx) \) we have \( RR \rightarrow_\beta \neg(RR) \)
On the origin of type theory

like in unrestricted set theory where every term is a set
in pure $\lambda$-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 $R \notin R$

$\lambda$-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 \(\lambda\)-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 \(R \notin R\)
\(\lambda\)-calculus: with \(R := \lambda x. \neg (xx)\) we have \(RR \rightarrow_{\beta} \neg (RR)\)

proposals to overcome this problem:

• restrict comprehension axiom to already defined sets
  use \(\{x \in A \mid P\}\) instead of \(\{x \mid P\}\)
  \(\sim\) modern set theory

• organize terms into a hierarchy
  - natural numbers are of type \(\iota\) and propositions of type \(o\)
  - unary predicates/sets of natural numbers are of type \(\iota \rightarrow o\)
  - sets of sets of natural numbers are of type \((\iota \rightarrow o) \rightarrow o\)
  - ...
  \(\sim\) modern type theory
Church simply-typed $\lambda$-calculus

simple types:

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

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

raw terms:

$$t, u ::= x \in \mathcal{V}_{obj} \mid tu \mid \lambda x : A, t$$
Well-typed terms

A typing environment $\Gamma$ is a finite map from variables to types.

Typing rules for terms:

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

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

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

- $xx$ is not typable anymore
- $\rightarrow_\beta$ terminates on well-typed terms
- $\rightarrow_\beta$ preserves typing: if $\Gamma \vdash t : A$ and $t \rightarrow_\beta u$, then $\Gamma \vdash u : A$
Dependent types / λΠ-calculus

A dependent type is a type that depends on terms

e.g. type (Array n) of arrays of size n

First introduced by de Bruijn in the Automath system in the 60’s

types:

\[ A, B ::= X \ t_1 \ldots t_n \mid \Pi x : A, B \]

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

e.g. concatenation function on arrays

\[ \text{concat} : \Pi p : \mathbb{N}, \text{Array} p \to \Pi q : \mathbb{N}, \text{Array} q \to \text{Array}(p + q) \]
Dependent types / \( \lambda \Pi \)-calculus

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

<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>KIND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>kinds ( K )</td>
<td>( \text{TYPE}</td>
<td>\Pi ) ( x : A, K )</td>
</tr>
<tr>
<td>families ( A )</td>
<td>( X</td>
<td>\text{At}</td>
</tr>
<tr>
<td>objects ( t )</td>
<td>( x</td>
<td>t t</td>
</tr>
</tbody>
</table>

this can be summarized as follows:

"\( t : A : K : \text{KIND} \)"

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

\( \Pi x_1 : A_1, \ldots, \Pi x_n : A_n : \text{TYPE} \)

a family is like a function returning a type:

\( (\lambda n : \mathbb{N}, \text{Array} n) 2 \rightarrow_\beta \text{Array} 2 \)
Typing rules for typing environments

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

a typing environment is now a sequence of type declarations

\[ \Gamma := \emptyset \mid \Gamma, x : A \mid \Gamma, X : K \]

“\( \Gamma \vdash \)” means that \( \Gamma \) is a well-typed environment:

\[
\text{\( \emptyset \vdash \)} \quad \frac{\Gamma \vdash A : \text{TYPE} \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash} \quad \frac{\Gamma \vdash K : \text{KIND} \quad X \notin \text{dom}(\Gamma)}{\Gamma, X : K \vdash}
\]
Signatures $\Sigma$

A typing environment can be split in two parts:
1. A fixed part $\Sigma$ representing global constants
2. A variable part $\Gamma$ for local variables
Typing rules for kinds and families

kinds:

\[
\begin{align*}
\Gamma \vdash & \quad \Gamma, x: A \vdash K : \text{KIND} \\
\Gamma \vdash \text{TYPE} : \text{KIND} & \quad \Gamma \vdash \Pi x : A, K : \text{KIND}
\end{align*}
\]

families:

\[
\begin{align*}
\Gamma \vdash (X, K) & \in \Gamma \\
\Gamma \vdash X : K & \\
\Gamma, x : A \vdash B : \text{TYPE} & \\
\Gamma \vdash \Pi x : A, B : \text{TYPE}
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : A \vdash B : K & \\
\Gamma \vdash \lambda x : A, B : \Pi x : A, K & \\
\Gamma, x : A \vdash \Pi x : B, K & \\
\Gamma \vdash t : B & \\
\Gamma \vdash A : K & \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash A : K \quad K \simeq_{\beta} K' & \\
\Gamma \vdash K' : \text{KIND} & \\
\Gamma \vdash A : K'
\end{align*}
\]
Typing rules for objects

\[ \Gamma \vdash (x, A) \in \Gamma \]
\[ \Gamma \vdash x : A \]
\[ \Gamma, x : A \vdash t : B \]
\[ \Gamma \vdash \lambda x : A, t : \Pi x : A, B \]
\[ \Gamma \vdash t : \Pi x : A, B \quad \Gamma \vdash u : A \]
\[ \Gamma \vdash tu : B\{(x, t)\} \]
\[ \Gamma \vdash t : A \quad A \equiv_{\beta} A' \quad \Gamma \vdash A' : \text{TYPE} \]
\[ \Gamma \vdash t : A' \]
Properties of the $\lambda\Pi$-calculus

- types are equivalent: if $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $A \betaeq B$
- $\leftrightarrow_\beta$ terminates on well-typed terms
- $\leftrightarrow_\beta$ 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:

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

typing rules:

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

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

$$\frac{\Gamma \vdash \lambda x : A, t : \Pi x : A, B \vdash \Gamma \vdash t : A \quad A \simeq_\beta A'}{\Gamma \vdash t : A'} \quad \frac{\Gamma \vdash t : A' \vdash \Gamma \vdash s}{\Gamma \vdash t : A'}$$
Pure Type Systems (PTS)

\[
\begin{align*}
\frac{}{\Gamma \vdash \text{Type : Kind}} & \quad \frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A, B : s}
\end{align*}
\]

the rules \((\text{sort})\) and \((\text{prod})\) can be generalized as follows:

\[
\begin{align*}
\frac{}{\Gamma \vdash (s_1, s_2) \in \mathcal{A}} & \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A, B : s_3}
\end{align*}
\]

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 $\mathcal{S} = \{\text{TYPE,KIND}\}$ and $\mathcal{A} = \{(\text{TYPE,KIND})\}$:

<table>
<thead>
<tr>
<th>feature</th>
<th>product rule in $\mathcal{P}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple types</td>
<td>TYPE, TYPE, TYPE</td>
</tr>
<tr>
<td>polymorphic types</td>
<td>KIND, TYPE, TYPE</td>
</tr>
<tr>
<td>dependent types</td>
<td>TYPE, KIND, KIND</td>
</tr>
<tr>
<td>type constructors</td>
<td>KIND, KIND, KIND</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
A : U & \quad B : U \\
\hline
A \to B : U
\end{align*}
\]

example: the sort TYPE of the simple types $\tau, \tau \to \tau$, \ldots

- 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 $U_0$ => they need to inhabit a new universe $U_1$ containing $U_0$

- by iteration we get an infinite sequence of nested universes

$$U_0 : U_1 : \ldots : U_i : U_{i+1} \ldots$$

$$\begin{array}{c}
A : U_i \quad B : U_j \\
\overrightarrow{A \to B : U_{\max(i,j)}}
\end{array}$$

available in some proof assistants like Coq, Agda, Lean

- PTS representation:

$$\begin{align*}
S &= \{ \text{TYPE}_i \mid i \in \mathbb{N} \} \\
A &= \{ (\text{TYPE}_i, \text{TYPE}_{i+1}) \mid i \in \mathbb{N} \} \\
P &= \{ (\text{TYPE}_i, \text{TYPE}_j, \text{TYPE}_{\max(i,j)}) \mid i, j \in \mathbb{N} \}
\end{align*}$$
What is rewriting?

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

a rewrite rule $l \overset{r}{\rightarrow} r$ 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:

<table>
<thead>
<tr>
<th>given a set $\mathcal{E}$ of equations, $\simeq_\mathcal{E}$ is decidable if there is a rewrite system $\mathcal{R}$ such that:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leftarrow_\mathcal{R}$ terminates</td>
</tr>
<tr>
<td>$\leftarrow_\mathcal{R}$ is confluent</td>
</tr>
<tr>
<td>$\simeq_\mathcal{R} = \simeq_\mathcal{E}$</td>
</tr>
<tr>
<td>where $\leftarrow_\mathcal{R}$ is the closure by context of $\mathcal{R}$</td>
</tr>
</tbody>
</table>
\( \lambda \Pi \)-calculus modulo rewriting (\( \lambda \Pi / R \) )

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

- a signature \( \Sigma \)
- a set \( R \) of rewrite rules on \( \Sigma \)

such that:

- \( \rightsquigarrow_\beta \cup \rightsquigarrow_R \) terminates
- \( \rightsquigarrow_\beta \cup \rightsquigarrow_R \) is confluent
- every rule \( l \rightsquigarrow r \) preserves typing: if \( \Gamma \vdash l \sigma : A \) then \( \Gamma \vdash r \sigma : A \)
Outline

Introduction

Lambda-Pi-calculus modulo rewriting
  Lambda-calculus
  Simple types
  Dependent types
  Pure Type Systems
  Rewriting

Dedukti language

Lambdapi proof assistant

Encoding logics in $\lambda\Pi/R$

Automated Theorem Provers
  Intrumenting provers for Dedukti proof production
  Reconstructing proofs
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](https://github.com/01mf02/kontroli-rs)
- Dkcheck [https://github.com/Deducteam/dedukti](https://github.com/Deducteam/dedukti)
- Lambdapi [https://github.com/Deducteam/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:

```bash
opam install dedukti
```

Compilation from the sources:

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

**Use:**

```bash
dk check file.dk
```
Dedukti syntax

BNF grammar:

file extension: .dk

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

identifiers:
(a-z|A-Z|0-9|_)+ [arbitrary string]
Terms

Type

<table>
<thead>
<tr>
<th>id</th>
<th>variable or constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>id.id</td>
<td>constant from another file</td>
</tr>
<tr>
<td>term term ... term</td>
<td>application</td>
</tr>
<tr>
<td>id [: term ] =&gt; term</td>
<td>abstraction</td>
</tr>
<tr>
<td>[id :] term -&gt; term</td>
<td>[dependent] product</td>
</tr>
<tr>
<td>( term )</td>
<td></td>
</tr>
</tbody>
</table>
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

\[ [ id \ast ] (term \to term)^+ . \]

\[ \begin{array}{l}
    x + 0 \to x \\
    x + s\,y \to s\,(x + y).
\end{array} \]

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

Lambda-Pi-calculus modulo rewriting
  Lambda-calculus
  Simple types
  Dependent types
  Pure Type Systems
  Rewriting

Dedukti language

Lambdapi proof assistant

Encoding logics in $\lambda\Pi/\mathcal{R}$

Automated Theorem Provers
  Intrumenting provers for Dedukti proof production
  Reconstructing proofs
**Lambdapi**

Lambdapi is an *interactive proof assistant* for $\lambda \Pi / R$.

- 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:

```bash
opam install lambdapi
```

Compilation from the sources:

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

Command line (batch mode):
```
lambda 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:
   \texttt{M-x \texttt{package-install RET lambdapi-mode}}

3. From sources:
   \texttt{make install_emacs}

+ add in `~/.emacs`:
   \texttt{(load "lambdapi-site-file")}
Emacs interface

- checked part
- edition buffer
- goals
- messages
- window layout can be customized

How to install the VSCode interface?

From the VSCode Marketplace
VSCode interface

- Checked part
- Edition buffer
- Goals
- Messages
developments must have a file `lambdapi.pkg` describing where to install the files relatively to the root of all installed libraries

```plaintext
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:

file extension: .lp

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

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

TYPE
(id .)*id
(term term . . term)
\( \lambda \text{ id } [\vdash \text{ term }], \text{ term} \)
\( \Pi \text{ id } [\vdash \text{ term }], \text{ term} \)
(term → term)
( ( term )
-
let id [\vdash \text{ term } ] := term in term
Command for declaring/defining a symbol

```
modifier* symbol id param* [ : term ] [= term ] [ begin proof end ] ;
param = id | _ | ( id + : term ) | [ id + : 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

```plaintext
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 \rightarrow term \ (with \ term \rightarrow \ term \ ) ^ * ;
```

pattern variables must be prefixed by $:
```
rule $x + 0 \rightarrow x$
with $x + s \; y \rightarrow s \; (x + y) ;$
```

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 \rightarrow $x
with $x + s \ y \rightarrow s \ ($x + $y)
with 0 + $x \rightarrow $x
with s \ $x + \ $y \rightarrow s \ ($x + $y);
```

matching on defined symbols

```
rule ($x + $y) + $z \rightarrow $x + ($y + $z);
```

non-linear patterns

```
rule $x - $x \rightarrow 0;
```

Lambdapi tries to automatically check:

local confluence (AC symbols/HO patterns not handled yet)
Higher-order pattern-matching

```
symbol  R: TYPE;
symbol  O: 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

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

rule ($x \cdot $y) \cdot $z \leftrightarrow $x \cdot ($y \cdot $z)
with 1 \cdot $x \leftrightarrow $x
with $x \cdot 1 \leftrightarrow $x
with inv $x \cdot $x \leftrightarrow 1
with $x \cdot inv $x \leftrightarrow 1
with inv $x \cdot ($x \cdot $y) \leftrightarrow $y
with $x \cdot (inv $x \cdot $y) \leftrightarrow $y
with inv 1 \leftrightarrow 1
with inv (inv $x) \leftrightarrow $x
with inv ($x \cdot $y) \leftrightarrow inv $y \cdot 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:

```plaintext
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:

```plaintext
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; // proof script
```
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:

\[
\begin{align*}
\text{proof} & ::= (\text{proof\_step} ;)^* \\
\text{proof\_step} & ::= \text{tactic} (\{ \text{proof} \})^*
\end{align*}
\]

• 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
  {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 Deduki or Lambdapi:

```plaintext
// 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 $x / cons$ $y$ $l$ _ ← $x$ $\neq$ $y$ $\land$ $x$ / $l$;
```
Command for generating induction principles
(currently for strictly positive parametric inductive types only)

**inductive** \( N : \text{TYPE} \) := \( 0 : N \mid s : N \to N \);

is equivalent to:

symbol \( N : \text{TYPE}; \)
symbol \( 0 : N; \)
symbol \( s : N \to N; \)
symbol \( \text{ind}_N \) (\( p : N \to \text{Prop} \))
  (case_0 : \text{Prf}(p \ 0))
  (case_s : \Pi \ x : N, \text{Prf}(p \ x) \to \text{Prf}(p(s \ x)))
  (n : N) : \text{Prf}(p \ n);
rule \( \text{ind}_N \) \( p \ $c0 \ $cs \ 0 \to $c0 \)
with \( \text{ind}_N \) \( p \ $c0 \ $cs \ (s \ $x) \)
  \( \to $cs \ $x \ (\text{ind}_N \ $p \ $c0 \ $cs \ $x) \)
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