Introduction to Proof System Interoperability

Frédéric Blanqui

Dedukti

INRIA

September 2022
Summary of first lecture

Introduction to:
• logical frameworks
• \(\lambda\)-calculus
• simple types
• dependent types
• rewriting
• \(\lambda\Pi\)-calculus modulo rewriting (\(\lambda\Pi/R\))
• Dedukti language
• Lambdapi proof assistant
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
  Instrumenting provers for Dedukti proof production
  Reconstructing proofs
Encoding logics in $\lambda\Pi/R$

we have seen what is a theory in the $\lambda\Pi$-calculus modulo rewriting
we are now going to see how to encode logics as $\lambda\Pi/R$ theories
First-order logic

- the set of terms
  - built from a set of function symbols equipped with an arity
- the set of propositions
  - built from a set of predicate symbols equipped with an arity
  - and the logical connectives $\top, \bot, \neg, \Rightarrow, \land, \lor, \leftrightarrow, \forall, \exists$
- the set of axioms (the actual theory)
- the subset of provable propositions
  - using deduction rules (e.g. natural deduction)
Natural deduction

provability, \( \vdash \), is a relation between a sequence of propositions \( \Gamma \) (the assumptions) and a proposition \( B \) (the conclusion) inductively defined from introduction and elimination rules for each connective:

\[
\begin{align*}
(\implies\text{-intro}) & \quad \frac{\Gamma, A \vdash B \quad \Gamma \vdash A}{\Gamma \vdash A \implies B} \\
(\implies\text{-elim}) & \quad \frac{\Gamma \vdash A \implies B \quad \Gamma \vdash A}{\Gamma \vdash B} \\
(\forall\text{-intro}) & \quad \frac{\Gamma \vdash A \quad x \notin \Gamma}{\Gamma \vdash \forall x, A} \\
(\forall\text{-elim}) & \quad \frac{\Gamma \vdash \forall x, A \quad \Gamma \vdash \{(x, u)\}}{\Gamma \vdash A}
\end{align*}
\]

\[
\ldots
\]
Encoding of first-order logic

• the set of terms \( I : \text{TYPE} \)
  \[ I \text{ built from a set of function symbols equipped with an arity function symbol: } I \rightarrow \ldots \rightarrow I \rightarrow I \]
Encoding of first-order logic

- the set of terms \( I : \text{TYPE} \)
  - built from a set of function symbols equipped with an arity
    function symbol: \( I \rightarrow \ldots \rightarrow I \rightarrow I \)

- the set of propositions \( Prop : \text{TYPE} \)
  - built from a set of predicate symbols equipped with an arity
    predicate symbol: \( I \rightarrow \ldots \rightarrow I \rightarrow Prop \)
Encoding of first-order logic

- the set of terms \( I : \text{TYPE} \)
  - built from a set of function symbols equipped with an arity
    function symbol: \( I \rightarrow \ldots \rightarrow I \rightarrow I \)

- the set of propositions \( \text{Prop} : \text{TYPE} \)
  - built from a set of predicate symbols equipped with an arity
    predicate symbol: \( I \rightarrow \ldots \rightarrow I \rightarrow \text{Prop} \)

- and the logical connectives \( \top, \bot, \neg, \rightarrow, \land, \lor, \leftrightarrow, \forall, \exists \)
  \( \top : \text{Prop}, \neg : \text{Prop} \rightarrow \text{Prop}, \forall : (I \rightarrow \text{Prop}) \rightarrow \text{Prop}, \ldots \)
  we use \( \lambda \)-calculus to encode quantifiers:
    we encode \( \forall x, A \) as \( \forall (\lambda x : I, A) \)
Encoding of first-order logic

- the set of terms \( I : \text{TYPE} \)
  - built from a set of function symbols equipped with an arity
    \( \text{function symbol: } I \rightarrow \ldots \rightarrow I \rightarrow I \)

- the set of propositions \( \text{Prop} : \text{TYPE} \)
  - built from a set of predicate symbols equipped with an arity
    \( \text{predicate symbol: } I \rightarrow \ldots \rightarrow I \rightarrow \text{Prop} \)
  - and the logical connectives \( \top, \bot, \neg, \Rightarrow, \land, \lor, \leftrightarrow, \forall, \exists \)
    \[ \top : \text{Prop}, \neg : \text{Prop} \rightarrow \text{Prop}, \forall : (I \rightarrow \text{Prop}) \rightarrow \text{Prop}, \ldots \]
    we use \( \lambda \)-calculus to encode quantifiers:
    we encode \( \forall x, A \) as \( \forall (\lambda x : I, A) \)

- the set of axioms (the actual theory)

- the subset of provable propositions
  - using deduction rules (e.g. natural deduction)
    \[ \text{but how to encode proofs?} \]
Using $\lambda$-terms to represent proofs
(Curry-de Bruijn-Howard isomorphism)

<table>
<thead>
<tr>
<th>logic</th>
<th>$\lambda$-calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposition</td>
<td>type</td>
</tr>
<tr>
<td>proof</td>
<td>$\lambda$-term</td>
</tr>
<tr>
<td>assumption</td>
<td>variable</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>$\rightarrow$</td>
</tr>
<tr>
<td>$\Rightarrow$-intro</td>
<td>abstraction</td>
</tr>
<tr>
<td>$\Rightarrow$-elim</td>
<td>application</td>
</tr>
<tr>
<td>$\forall$</td>
<td>$\Pi$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

the Curry-de Bruijn-Howard isomorphism reduces:

- proof-checking to type-checking
- provability to type inhabitation
Using \( \lambda \)-terms to represent proofs
(Curry-de Bruijn-Howard isomorphism)

take the rules of natural deduction

\[
\begin{align*}
\Rightarrow\text{-intro} & \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \\
\Rightarrow\text{-elim} & \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \\
\forall\text{-intro} & \quad \frac{\Gamma \vdash A \quad x \notin \Gamma}{\Gamma \vdash \forall x, A} \\
\forall\text{-elim} & \quad \frac{\Gamma \vdash \forall x, A}{\Gamma \vdash A\{(x, u)\}}
\end{align*}
\]
Using $\lambda$-terms to represent proofs
(Curry-de Bruijn-Howard isomorphism)

take the rules of natural deduction
by giving a name to every assumption, we get a typing environment

\[ A_1, \ldots, A_n \leadsto x_1 : A_1, \ldots, x_n : A_n \]

\[
\begin{align*}
(\Rightarrow\text{-intro}) & \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \\
(\Rightarrow\text{-elim}) & \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \\
(\forall\text{-intro}) & \quad \frac{\Gamma \vdash A \quad x \notin \Gamma}{\Gamma \vdash \forall x, A} \\
(\forall\text{-elim}) & \quad \frac{\Gamma \vdash \forall x, A}{\Gamma \vdash A\{x, u\}}
\end{align*}
\]
Using \( \lambda \)-terms to represent proofs
(Curry-de Bruijn-Howard isomorphism)

take the rules of natural deduction
by giving a name to every assumption, we get a typing environment

\[
A_1, \ldots, A_n \quad \leadsto \quad x_1 : A_1, \ldots, x_n : A_n
\]

by mapping every deduction rule to a \( \lambda \)-term construction
the typing rules of \( \lambda \Pi \) correspond to natural deduction rules!

\[
\begin{align*}
(\Rightarrow\text{-intro}) & \quad \Gamma, x : A \vdash t : B \\
& \quad \Gamma \vdash \lambda x : A, t : A \Rightarrow B \\
(\Rightarrow\text{-elim}) & \quad \Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash u : A \\
& \quad \Gamma \vdash tu : B \\
(\forall\text{-intro}) & \quad \Gamma \vdash t : A \\
& \quad x \notin \Gamma \\
& \quad \Gamma \vdash \lambda x, t : \forall x, A \\
(\forall\text{-elim}) & \quad \Gamma \vdash t : \forall x, A \\
& \quad \Gamma \vdash tu : A\{(x, u)\}
\end{align*}
\]
Encoding the Curry-de Bruijn-Howard isomorphism

terms of type $\textit{Prop}$ are not types...

but we can interpret a proposition as a type by taking:

\[
\textit{Prf} : \textit{Prop} \rightarrow \text{TYPE}
\]

$\textit{Prf}$ $A$ is the type of proofs of proposition $A$
Encoding the Curry-de Bruijn-Howard isomorphism

terms of type \( \text{Prop} \) are not types...

but we can interpret a proposition as a type by taking:

\[
\text{Prf} : \text{Prop} \rightarrow \text{TYPE}
\]

\( \text{Prf} \ A \) is the type of proofs of proposition \( A \)

but

\[
\lambda x : \text{Prf} \ A, x : \text{Prf} \ A \rightarrow \text{Prf} \ A
\]

and

\[
\lambda x : \text{Prf} \ A, x \neq \text{Prf} (A \Rightarrow A)
\]
Encoding the Curry-de Bruijn-Howard isomorphism

terms of type \( \text{Prop} \) are not types... 

but we can interpret a proposition as a type by taking:

\[
\text{Prf} : \text{Prop} \rightarrow \text{TYPE}
\]

\( \text{Prf} A \) is the type of proofs of proposition \( A \)

but

\[
\lambda x : \text{Prf} A, x : \text{Prf} A \rightarrow \text{Prf} A
\]

and

\[
\lambda x : \text{Prf} A, x \neq \text{Prf} (A \Rightarrow A)
\]

unless we add the rewrite rule

\[
\text{Prf} (A \Rightarrow B) \leftrightarrow \text{Prf} A \rightarrow \text{Prf} B
\]
Encoding ∀

we can do something similar for $\forall : (I \to Prop) \to Prop$ by taking:

$$Prf(\forall A) \leftrightarrow \Pi x : I, Prf(A x)$$
Encoding the other connectives

The other connectives can be defined by using a meta-level quantification on propositions:

\[ \text{Prf}(A \land B) \iff \Pi \cdot \text{Prop}, (\text{Prf} A \to \text{Prf} B) \to \text{Prf} \Downarrow. \]

Note that introduction and elimination rules can be derived:

(\&\text{-intro}): 
\[ \lambda a : \text{Prf} A, \lambda b : \text{Prf} B, \lambda \Downarrow : \text{Prop}, \lambda h : \text{Prf} A \to \text{Prf} B \to \text{Prf} \Downarrow, h a b \]
is of type 
\[ \text{Prf} A \to \text{Prf} B \to \text{Prf}(A \land B) \]

(\&\text{-elim1}): 
\[ \lambda c : \text{Prf}(A \land B), c A(\lambda a : \text{Prf} A, \lambda b : \text{Prf} B, a) \]
is of type 
\[ \text{Prf}(A \land B) \to \text{Prf} A \]
To summarize: \(\lambda\Pi/R\)-theory \(FOL\) for first-order logic

signature \(\Sigma_{FOL}\):

- \(I : TYPE\)
- \(f : I \to \ldots \to I \to I\) for each function symbol \(f\) of arity \(n\)
- \(Prop : TYPE\)
- \(P : I \to \ldots \to I \to Prop\) for each predicate symbol \(P\) of arity \(n\)
- \(\top : Prop, \bot : Prop, \forall : (I \to Prop) \to Prop, \ldots\)
- \(Prf : Prop \to TYPE\)
- \(a : Prf A\) for each axiom \(A\)

rules \(R_{FOL}\):

- \(Prf (A \Rightarrow B) \leftrightarrow Prf A \Rightarrow Prf B\)
- \(Prf (\forall A) \leftrightarrow \Pi x : I, Prf (A x)\)
- \(Prf (A \land B) \leftrightarrow \Pi \flat : Prop, (Prf A \Rightarrow Prf B \Rightarrow Prf \flat) \Rightarrow Prf \flat\)
  - \(Prf \bot \leftrightarrow \Pi \flat : Prop, Prf \flat\)
- \(Prf (\lnot A) \leftrightarrow Prf A \Rightarrow Prf \bot\)

...
Encoding of first-order logic in λΠ/FOL

encoding of terms:
- $|x| = x$
- $|t_1 \ldots t_n| = f|t_1| \ldots |t_n|

encoding of propositions:
- $|P t_1 \ldots t_n| = P|t_1| \ldots |t_n|
- $|T| = T$
- $|A \land B| = |A| \land |B|
- $|\forall x, A| = \forall(\lambda x : I, |A|)$
- $\ldots$
- $|\Gamma, A| = |\Gamma|, x_{|\Gamma|+1} : A$

encoding of proofs:
- $\frac{\pi_{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B} (\Rightarrow_i) = \lambda x_{|\Gamma|+1} : Prf |A|, |\pi_{\Gamma, A \vdash B}|$
- $\frac{\pi_{\Gamma \vdash A \Rightarrow B} \quad \pi_{\Gamma \vdash A}}{\Gamma \vdash B} (\Rightarrow_e) = |\pi_{\Gamma \vdash A \Rightarrow B}|, |\pi_{\Gamma \vdash A}|$

$\ldots$
Properties of the encoding in $\lambda\Pi/FOL$

- a term is mapped to a term of type $I$
- a proposition is mapped to a term of type $Prop$
- a proof of $A$ is mapped to a term of type $Prf \mid A$
Properties of the encoding in $\lambda\Pi/FOL$

- a term is mapped to a term of type $I$
- a proposition is mapped to a term of type $Prop$
- a proof of $A$ is mapped to a term of type $Prf \ |A|$

but, if we find a term $t$ of type $Prf \ |A|$, can we deduce that $A$ is provable?
Properties of the encoding in $\lambda\Pi/FOL$

- a term is mapped to a term of type $I$
- a proposition is mapped to a term of type $Prop$
- a proof of $A$ is mapped to a term of type $Prf\mid A$

but, if we find a term $t$ of type $Prf\mid A$, can we deduce that $A$ is provable?

- yes, the encoding is conservative: if $Prf\mid A$ is inhabited then $A$ is provable

proof sketch: because $\leftrightarrow_\beta$ terminates and is confluent, $t$ has a normal form, and terms in normal form can be easily translated back in first-order logic and natural deduction
Multi-sorted first-order logic

for each sort $I_k$ (e.g. point, line, circle), add:

$I_k : \text{TYPE}$

$\forall_k : (I_k \rightarrow \text{Prop}) \rightarrow \text{Prop}$

$\text{Prf}(\forall_k A) \leftrightarrow \Pi x : I_k, \text{Prf}(Ax)$
Polymorphic first-order logic

same trick as Curry-de Bruijn-Howard

\[
\begin{align*}
Set &: \text{TYPE} \\
El &: Set \to \text{TYPE} \\
i &: Set \\
\forall &: \Pi a : Set, (El a \to Prop) \to Prop \\
Prf (\forall a p) &: \Pi x : El a, Prf (p x)
\end{align*}
\]
Higher-order logic

<table>
<thead>
<tr>
<th>order</th>
<th>quantification on</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>elements</td>
</tr>
<tr>
<td>2</td>
<td>sets of elements</td>
</tr>
<tr>
<td>3</td>
<td>sets of sets of elements</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\omega$</td>
<td>any set</td>
</tr>
</tbody>
</table>
Higher-order logic

<table>
<thead>
<tr>
<th>order</th>
<th>quantification on</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>elements</td>
</tr>
<tr>
<td>2</td>
<td>sets of elements</td>
</tr>
<tr>
<td>3</td>
<td>sets of sets of elements</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ω</td>
<td>any set</td>
</tr>
</tbody>
</table>

quantification on functions:

\[ \sim : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set} \]

\[ El(a \sim b) \leftrightarrow El \ a \rightarrow El \ b \]
Higher-order logic

<table>
<thead>
<tr>
<th>order</th>
<th>quantification on</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>elements</td>
</tr>
<tr>
<td>2</td>
<td>sets of elements</td>
</tr>
<tr>
<td>3</td>
<td>sets of sets of elements</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$\omega$</td>
<td>any set</td>
</tr>
</tbody>
</table>

quantification on functions:

$\leadsto : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$

$ El(a \leadsto b) \leftrightarrow El a \rightarrow El b $

quantification on propositions/impredicativity (e.g. $\forall p, p \Rightarrow p$):

$\circ : \text{Set}$

$El \circ \rightarrow \text{Prop}$
Encoding dependent types

dependent implication:
\[ \Rightarrow_d : \Pi a : Prop, (\text{Prf } a \rightarrow Prop) \rightarrow Prop \]
\[ \text{Prf} (a \Rightarrow_d b) \leftrightarrow \Pi x : \text{Prf } a, \text{Prf } (b x) \]
Encoding dependent types

dependent implication:
\[ \Rightarrow^d : \Pi a : Prop, (Prf a \to Prop) \to Prop \]
\[ Prf(a \Rightarrow^d b) \leftrightarrow \Pi x : Prf a, Prf(b x) \]
dependent types:
\[ \sim^d : \Pi a : Set, (El a \to Set) \to Set \]
\[ El(a \sim^d b) \leftrightarrow \Pi x : El a, El(b x) \]
Encoding dependent types

dependent implication:
\(\Rightarrow_d : \Pi a : \text{Prop}, (\text{Prf} \ a \rightarrow \text{Prop}) \rightarrow \text{Prop} \)
\(\text{Prf} (a \Rightarrow_d b) \iff \Pi x : \text{Prf} \ a, \text{Prf} (b \ x)\)

dependent types:
\(\sim_d : \Pi a : \text{Set}, (\text{El} \ a \rightarrow \text{Set}) \rightarrow \text{Set} \)
\(\text{El} (a \sim_d b) \iff \Pi x : \text{El} \ a, \text{El} (b \ x)\)

proofs in object-terms:
\(\pi : \Pi p : \text{Prop}, (\text{Prf} \ p \rightarrow \text{Set}) \rightarrow \text{Set} \)
\(\text{El} (\pi \ p \ a) \iff \Pi x : \text{Prf} \ p, \text{El} (a \ x)\)

example: \(\text{div} : \text{El} (i \sim i \sim_d \lambda y : \text{El} \ i, \pi (y > 0)(\lambda_\_ , i))\)
takes 3 arguments: \(x : \text{El} \ i, \ y : \text{El} \ i, \ p : \text{Prf} (y \ > \ 0)\)
and returns a term of type \(\text{El} \ i\)
Encoding the calculus of constructions

we now have all the ingredients to encode
the calculus of constructions:

<table>
<thead>
<tr>
<th>system</th>
<th>PTS rule</th>
<th>λΠ/Ω rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple types</td>
<td>TYPE, TYPE</td>
<td>( \text{Prf}(a \Rightarrow_d b) \leftrightarrow \Pi x : \text{Prf}(a), \text{Prf}(b x) )</td>
</tr>
<tr>
<td>polymorphic types</td>
<td>KIND, TYPE</td>
<td>( \text{Prf}(\forall a b) \leftrightarrow \Pi x : \text{El}(a, \text{Prf}(b x)) )</td>
</tr>
<tr>
<td>dependent types</td>
<td>TYPE, KIND</td>
<td>( \text{El}(\pi a b) \leftrightarrow \Pi x : \text{Prf}(a, \text{El}(b x)) )</td>
</tr>
<tr>
<td>type constructors</td>
<td>KIND, KIND</td>
<td>( \text{El}(a \sim_d b) \leftrightarrow \Pi x : \text{El}(a), \text{El}(b x) )</td>
</tr>
</tbody>
</table>
Encoding Functional Pure Type Systems

terms and types:

\[ t ::= x \mid tt \mid \lambda x : t, t \mid \Pi x : t, t \mid s \in S \]

typing rules:

\[
\begin{align*}
\emptyset \vdash & \quad \Gamma \vdash A : s & & \quad \Gamma \vdash (x, A) \in \Gamma \\
\Gamma, x : A \vdash & \quad \Gamma \vdash x : A \\
\text{(sort)} & \quad \Gamma \vdash (s_1, s_2) \in A \\
& \quad \Gamma \vdash s_1 : s_2 \\
\text{(prod)} & \quad \Gamma \vdash A : s_1 \\
& \quad \Gamma, x : A \vdash B : s_2 \\
& \quad ((s_1, s_2), s_3) \in P \\
& \quad \Gamma \vdash \Pi x : A, B : s_3 \\
\Gamma, x : A \vdash & \quad t : B \\
\Gamma \vdash & \quad \Pi x : A, B \\
\Gamma \vdash & \quad u : A \\
\Gamma \vdash & \quad \lambda x : A, t : \Pi x : A, B \\
\Gamma \vdash & \quad tu : B \{(x, u)\} \\
\Gamma \vdash & \quad t : A \\
\Gamma \vdash A \approx_\beta A' & \quad \Gamma \vdash A' : s \\
\Gamma \vdash & \quad t : A'
\end{align*}
\]
Encoding Functional Pure Type Systems

(Cousineau & Dowek, 2007)

signature:

\[ U_s : \text{TYPE} \quad \text{for each sort } s \in S \]

\[ El_s : U_s \rightarrow \text{TYPE} \quad \text{for every } (s_1, s_2) \in A \]

\[ s_1 : U_{s_2} \quad \text{for every } (s_1, s_2) \in \mathcal{A} \]

\[ \pi_{s_1, s_2} : \Pi a : U_{s_1}, (El_{s_1} a \rightarrow U_{s_2}) \rightarrow U_{s_3} \quad \text{for every } (s_1, s_2, s_3) \in \mathcal{P} \]

rules:

\[ El_{s_2} s_1 \leftrightarrow U_{s_1} \quad \text{for every } (s_1, s_2) \in A \]

\[ El_{s_3}(\pi_{s_1, s_2} a b) \leftrightarrow \Pi x : El_{s_1} a, El_{s_2}(b x) \quad \text{for every } (s_1, s_2, s_3) \in \mathcal{P} \]

encoding:

|\lambda x : A, t|_\Gamma = \lambda x : El_{s_1}|_\Gamma, |t|_\Gamma, x:A | \Gamma, \ A : s \quad \text{if } \Gamma \vdash A : s

|tu|_\Gamma = |t|_\Gamma |u|_\Gamma

|\Pi x : A, B|_\Gamma = \pi_{s_1, s_2}|A|_\Gamma (\lambda x : El_{s_1}|_\Gamma, |B|_\Gamma, x:A) | \Gamma, \ A : s_1 \text{ and } \Gamma, x : A \vdash B : s_2
Encoding other features

- recursive functions (Assaf 2015, Cauderlier 2016, Férey 2021)
  - different approaches, no general theory
  - encoding in recursors (ongoing work by Felicissimo & Cockx)
- universe polymorphism (Genestier 2020)
  - requires rewriting with matching modulo AC
    or rewriting on AC canonical forms
- $\eta$-conversion on function types (Genestier 2020)
- predicate subtyping with proof irrelevance (Hondet 2020)
- co-inductive objects and co-recursion (Felicissimo 2021)
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
  Instrumenting provers for Dedukti proof production
  Reconstructing proofs
from slides by Guillaume Burel at the Dedukti school (June 2022)
ITP vs ATP

Limitations of interactive theorem provers (ITP):
• lack of automation
• need for specially trained experts
• bottleneck for widespread use

Limitations of automated theorem provers (ATP):
• lack of confidence
• highly optimized tools
• code too complex to be certified
Cooperation

ITP:
• use ATPs to discharge some proof obligations
e.g. Sledgehammer, SMTCoq

ATP:
• Export proofs that can be independently checked
• Ideally, checkable by a well known tool
Ideal goal

- ITP
- ATP
- Formula
- Call
- Proof
- Reconstruction
- Translation
- Output
From Lambdapi to ATPs

Why3:
- platform for deductive program verification
- able to delegate proofs to many provers
- https://why3.lri.fr/

Calling provers within Lambdapi:
- Tactic why3
Current why3 tactic

- Lambdapi: goal
- abstract
- FOL formula: goal admitted as an axiom
- Why3: return
- Vampire: yes
- AltErgo
- CVC4
Trusting ATPs

ATP:
- quite big piece of software
- complex proof calculi
- finely tuned, optimization hacks

Trust?
- Originally, only answer "yes"/"no" (more often, "maybe")
- More and more, produce proof traces/big steps proofs
To increase confidence:

- either build a certified proof checker for proof traces
e.g. Coq certified checker for DRAT proof traces of SAT solvers

- or directly produce a proof checkable by your favorite assistant
Instrumenting a prover to produce proofs

Pros:
- Access to all needed informations

Cons:
- Needs to embed the calculus of the prover into Dedukti
- Needs to know precisely the code of the prover

more or less easy depending complexity of code/proof calculus
easier if proof output designed from the start (e.g. Zenon)

⇒ can only be done for a few provers
Provers outputing Dedukti proofs

- **iProverModulo:**
  extension of iProver for Deduction Modulo Theory
  https://github.com/gburel/iProverModulo.git

- **ZenonModulo:**
  extension of Zenon for Deduction Modulo Theory + Arithmetic
  https://github.com/Deducteam/zenon_modulo.git

- **ArchSAT:**
  SMT solver
  https://github.com/Gbury/archsat
Translating proofs

First, need to carefully choose in which theory we are working e.g. FOL

Then, two approaches:
- Directly translate proofs into Dedukti, e.g. iProverModulo
- Embedding the proof calculus into Dedukti, e.g. ZenonModulo
iProverModulo (Burel 2011)

Patch to iProver (Korovin 2008)
iProver: Combination of two proof procedures:
• Inst-Gen
• Ordered resolution
iProverModulo: add support for Deduction Modulo Theory
Resolution Calculus

Literal: atom $A$ or negation of atom $\neg A$
Clause: set/disjunction of literals $L_1 \lor \ldots \lor L_m \ (m \geq 0)$
Problem: set/conjunction of clauses $C_1 \land \ldots \land C_k$

Derive new clauses using

$$
\frac{A, C \quad \neg B, D}{C \sigma, D \sigma} \quad \sigma = mgu(A, B)
$$

until the empty clause is produced
we want to prove \((C_1 \land \ldots \land C_k) \Rightarrow \bot\)

\((C_1 \land \ldots \land C_k) \Rightarrow \bot\) is equivalent to \((C_1 \Rightarrow \bot) \lor \ldots \lor (C_k \Rightarrow \bot)\)

\((L_1 \lor \ldots \lor L_m) \Rightarrow \bot\) is equivalent to \((L_1 \Rightarrow \bot) \land \ldots \land (L_m \Rightarrow \bot)\)

\(C = \{L_1, \ldots, L_m\}\) which corresponds to \(\forall x_1, \ldots, \forall x_p, L_1 \lor \ldots \lor L_m\),

where \(x_1, \ldots, x_p\) are the free variables of \(L_1, \ldots, L_m\), is translated as:

\[
\Pi x_1 : I, \ldots \Pi x_p : I, \Pi \vdash: \text{Prop}, \mid L_1 \mid \vdash \ldots \mid L_m \mid \vdash \text{Prf} \vdash
\]

with \(\mid A \mid = \text{Prf} A \rightarrow \text{Prf} \vdash\) and \(\mid \neg A \mid = (\text{Prf} A \rightarrow \text{Prf} \vdash) \rightarrow \text{Prf} \vdash\)

(remember that \(\text{Prf} \bot \leftrightarrow \Pi \vdash: \text{Prop}, \text{Prf} \vdash\))
Translation of propositional resolution

\[
\frac{A, L_1, \ldots, L_m, \neg A, L_{m+1}, \ldots, L_n}{L_1, \ldots, L_n}
\]

given \( c : |A, L_1, \ldots, L_m| = \Pi \vDash \text{Prop}, |A|_b \rightarrow |L_1|_b \rightarrow \ldots \rightarrow |L_m|_b \rightarrow \text{Prf} \)

and \( d : |\neg A, L_{m+1}, \ldots, L_n| = \Pi \vDash \text{Prop}, (|A|_b \rightarrow \text{Prf} _b) \rightarrow |L_{m+1}|_b \rightarrow \ldots \rightarrow |L_n|_b \rightarrow \text{Prf} _b \)

we obtain

\( e : |L_1, \ldots, L_n| = \Pi \vDash \text{Prop}, |L_1|_b \rightarrow \ldots \rightarrow |L_n|_b \rightarrow \text{Prf} _b \)

by taking

\( e = \lambda b, \lambda \overline{1}, \ldots, \lambda \overline{n}, c \vDash (\lambda a, d \vDash (\lambda \overline{a}, \exists a) \overline{L}_{m+1} \ldots \overline{L}_n) \overline{1} \ldots \overline{m} \)
Limits

Can handle various simplification rules, rewriting
Can be extended to superposition (E, Vampire, ...) 

But:
• works if the proof uses resolution only (i.e. no Inst-Gen)
• no translation of the transformation into clauses
ZenonModulo

(Delahaye, Doligez, Gilbert, Halmagrand, and Hermant, 2013)

- extension of Zenon to Deduction Modulo Theory
- tableau-based
- polymorphic first-order logic with equality
Tableau proofs

- proofs by contradiction
- roughly bottom-up sequent-calculus with metavariables

\[
\frac{P, \neg P}{\circ} \quad \frac{\neg (A \Rightarrow B)}{A, \neg B} \quad \frac{\neg (A \land B)}{\neg A} \quad \frac{\neg (A \land B)}{\neg B} \quad \frac{\neg (P \Rightarrow (P \land P))}{P} \quad \frac{\neg (P \land P)}{\neg P} \quad \frac{\neg P}{\circ} \quad \frac{\neg P}{\circ}
\]

Example of proof:
Deep embedding of proof calculus

\[
\begin{align*}
P, \neg P \quad \odot : \\
\text{symbol } Rax p & : \text{Prf } p \rightarrow \text{Prf } (\neg p) \rightarrow \text{Prf } \bot; \\
\neg (A \Rightarrow B) & \quad \alpha_{\Rightarrow} : \\
A, \neg B & \rightarrow : \\
\text{symbol } R_{\Rightarrow} a b & : (\text{Prf } a \rightarrow \text{Prf } (\neg b) \rightarrow \text{Prf } \bot) \rightarrow \text{Prf } (\neg (a \Rightarrow b)) \rightarrow \text{Prf } \bot; \\
\neg (A \land B) & \quad \beta_{\land} : \\
\neg A & \quad \neg B : \\
\text{symbol } R_{\land} a b & : (\text{Prf } (\neg a) \rightarrow \text{Prf } \bot) \rightarrow (\text{Prf } (\neg b) \rightarrow \text{Prf } \bot) \rightarrow \text{Prf } (\neg (a \land b)) \rightarrow \text{Prf } \bot;
\end{align*}
\]
Deep translation of the example

\[ \frac{\neg (P \Rightarrow (P \land P))}{P} \alpha \Rightarrow \] 
\[ \frac{\neg (P \land P)}{P} \beta \land \] 
\[ \frac{\neg P}{\circ} \circ \quad \frac{\neg P}{\circ} \circ \] 

opaque symbol goal : Prf \( \neg (p \Rightarrow (p \land p)) \Rightarrow \text{Prf} \bot \) :=
\[ R \equiv p (p \land p) (\lambda \pi, R \land p p (\text{Rax } p \pi)) (\text{Rax } p \pi); \]
Making the embedding more shallow

by reducing it to Natural Deduction:

\[
(\land I) \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \land B} \quad (\land El) \quad \frac{\Gamma \vdash A \land B}{\Gamma \vdash A} \quad (\land Er) \quad \frac{\Gamma \vdash A \land B}{\Gamma \vdash B}
\]

\[
(\Rightarrow I) \quad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad (\Rightarrow E) \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}
\]

Natural Deduction in Lambdapi:

```
symbol ∧I p q : Prf p → Prf q → Prf (p ∧ q);
symbol ∧El p q : Prf (p ∧ q) → Prf p;
symbol ∧Er p q : Prf (p ∧ q) → Prf q;
symbol ⇒I p q : (Prf p → Prf q) → Prf (p ⇒ q);
symbol ⇒E p q : Prf (p ⇒ q) → Prf p → Prf q;
```
Defining Tableau rules in ND

rule Rax \leftrightarrow \lambda \ p \ h \ \pi, \ \neg E \ p \ \pi \ h;

rule R\land \leftrightarrow \lambda \ p \ q \ h1 \ h2 \ h3,
\ h1 (\neg I \ p \ (\lambda \ h5, \ h2 (\neg I \ q \ (\lambda \ h6,
\ \neg E \ (p \ \land \ q) \ h3 (\land I \ p \ q \ h5 \ h6)))));

rule R\Rightarrow \leftrightarrow \lambda \ p \ q \ h1 \ h2,
\ \neg E \ (p \Rightarrow q) \ h2 (\Rightarrow I \ p \ q \ (\lambda \ h3, \ \neg E \ (h1 \ h3
\ (\neg I \ q \ (\lambda \ h4, \ \neg E \ (p \Rightarrow q) \ h2 (\Rightarrow I \ p \ q \ (\lambda \ _ \ h4)))))) \ q));

correctness follows from subject reduction
which is checked automatically by Lambdapi!

compute goal;
assert \vdash \ goal \equiv \lambda \ h2, \ \neg E \ (p \Rightarrow (p \ \land \ p)) \ h2 (\Rightarrow I \ p \ (p \ \land \ p)
\ (\lambda \ h3, \ \neg E \ (\neg E \ (p \Rightarrow (p \ \land \ p)) \ h2
\ (\Rightarrow I \ p \ (p \ \land \ p) (\lambda \ _ \ (\land I \ p \ p \ h3 \ h3))) \ (p \ \land \ p)));
Making it even more shallow

Reduce Natural Deduction thanks to the shallow encoding of FOL

\[
\begin{align*}
\text{rule } \Rightarrow & \leftarrow \lambda p \ q \ \pi, \ \pi; \\
\text{rule } \Rightarrow & \leftarrow \lambda p \ q \ \pi, \ \pi; \\
\text{rule } \wedge & \leftarrow \lambda p \ q \ \pi \ p, \ \pi; \\
\text{rule } \Rightarrow & \leftarrow \lambda p \ q \ \pi \ p, \ \pi; \\
\text{rule } \wedge & \leftarrow \lambda p \ q \ \pi \ p, \ \pi; \\
\text{rule } \Rightarrow & \leftarrow \lambda p \ q \ \pi \ p, \ \pi; \\
\text{compute } \text{goal}; \\
\text{assert } \vdash \text{goal } \equiv \\
\lambda h2, h2 \ (\lambda h3, h2 \ (\lambda \_ \_ \pi, \pi \ h3 \ h3) \ (p \ p));
\end{align*}
\]
Limits of instrumentation

Provers can be hard to instrument to produce Dedukti proofs
- large piece of software
- developers not expert in $\lambda\Pi$-calculus modulo theory
- non stable and quite big proof calculus
Proof calculus of \( E \)
But often, provers produce at least a proof trace:
• list of formulas that were derived to obtain the proof
• sometimes with more information
  – premises
  – name of the inference rules
  – theory
  – ...
Example of trace: TSTP format

Output format of E, Vampire, Zipperposition, ...

- list of formulas
- annotated by an inference tree whose leaves are other formulas

\[
\text{cnf(c}_{-}0_{-}60,\text{plain}, \nonumber
\begin{array}{l}
\quad \text{ ( join(X1,join(X2,X3)) = join(X2,join(X1,X3)) )},
\quad \text{inference(rw,[status(thm)]},
\quad \text{ [inference(spm,[status(thm)],[c}_{-}0_{-}30,c}_{-}0_{-}18]!,}
\quad \text{ c}_{-}0_{-}30]!)
onumber\end{array}
\]

Example of trace: TSTP format

Output format of E, Vampire, Zipperposition, ...

- list of formulas
- annotated by an inference tree whose leaves are other formulas

\[
\text{cnf}(c_0, \text{plain}, \\
( \text{join}(X_1, \text{join}(X_2, X_3)) = \text{join}(X_2, \text{join}(X_1, X_3)) ), \\
\text{inference}(\text{rw}, [\text{status(thm)}], \\
[\text{inference}(\text{spm}, [\text{status(thm)}], [c_{0,30}, c_{0,18}]), \\
c_{0,30}])).
\]

Independent of the proof calculus
Proof reconstruction

Use the content of the proof trace to reconstruct a Dedukti proof

Idea:
• Prove each step using a Dedukti producing tool
• Combine those proofs to get a proof of the original formula

Try to be agnostic:
• w.r.t. the prover that produces the trace
• w.r.t. the prover that reproves the steps
Ekstrakto (El Haddad 2021)

• Input: TSTP proof trace
• Output: Reconstructed LambdaPi proof

https://github.com/Deducteam/ekstrakto
Ekstrakto architecture
Experimental evaluation

Benchmark:
• CNF problems of TPTP v7.4.0 (8118 files)

Trace producers:
• E and Vampire

Step provers:
• ZenonModulo and ArchSat
### Results

#### Percentage of reconstructed proof steps

<table>
<thead>
<tr>
<th>Prover</th>
<th>% E</th>
<th>% Vampire</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZenonModulo</td>
<td>87%</td>
<td>60%</td>
</tr>
<tr>
<td>ArchSAT</td>
<td>92%</td>
<td>81%</td>
</tr>
<tr>
<td>ZenonModulo $\cup$ ArchSAT</td>
<td>95%</td>
<td>85%</td>
</tr>
</tbody>
</table>

#### Percentage of completely reconstructed proofs

<table>
<thead>
<tr>
<th>Prover</th>
<th>% E TSTP</th>
<th>% Vampire TSTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZenonModulo</td>
<td>45%</td>
<td>54%</td>
</tr>
<tr>
<td>ArchSAT</td>
<td>56%</td>
<td>74%</td>
</tr>
<tr>
<td>ZenonModulo $\cup$ ArchSAT</td>
<td>69%</td>
<td>83%</td>
</tr>
</tbody>
</table>
Non provable steps

Problem:
- some steps are not provable
  their conclusion is not a logical consequence of their premises
- OK because they preserve provability
- but Ekstrakto cannot work for them
**Non provable steps**

Problem:
- some steps are not provable
  their conclusion is not a logical consequence of their premises
- OK because they preserve provability
- but Ekstrako cannot work for them

Main instance: Skolemization

\[
\Gamma, \forall x, \exists y, A[x, y] \vdash B \text{ iff } \Gamma, \forall x, A[x, f(x)] \vdash B \text{ for a fresh } f
\]

Present in the CNF transformation used by almost all ATPs
Skonvert (El Haddad 2021)

Inputs:
- an axiom and its Skolemized version
- a Lambdapi proof using the latter

Output:
- a Lambdapi proof using the non-Skolemized axiom
Implementation of Dowek & Werner's constructive proof of Skolem theorem (2005) in the context of first-order natural deduction

Problem:
- the proof has to be in normal form
- also w.r.t. so-called commuting cuts
Commuting cuts

\[
\begin{align*}
\Gamma &\vdash A \lor B & \Gamma, A \vdash C \land D & \Gamma, B \vdash C \land D \\
\hline
\Gamma &\vdash C \land D & \Gamma &\vdash C & \land E \\
\hline
\Gamma &\vdash C & \land E
\end{align*}
\]

\[
\begin{align*}
\Gamma &\vdash A \lor B \\
\hline
\Gamma, A \vdash C \land D & \Gamma, B \vdash C \land D \\
\hline
\Gamma &\vdash C & \land E
\end{align*}
\]
Reducing commuting cuts

If we work on shallow proofs, these cuts are no longer visible

⇒ we need to stay at the ND level
and add rules to reduce commuting cuts:

\[
\begin{align*}
\text{rule } \& \text{El } c \; d \; (\vee \; a \; b \; \text{paorb } (c \; \& \; d) \; \text{pac } \; \text{pbc}) \\
\leftrightarrow \vee \; a \; b \; \text{paorb } c \; (\lambda \; \text{pa}, \; \& \; \text{El } c \; d \; (\text{pac } \; \text{pa})) \\
(\lambda \; \text{pb}, \; \& \; \text{El } c \; d \; (\text{pbc } \; \text{pb}));
\end{align*}
\]
Example proof with Skolem symbol

```
symbol_goal
(ax_tran : Prf (\ (\ X1 , \ (\ X2 , \ (\ X3 ,
        (p X1 X2) \rightarrow ((p X2 X3) \rightarrow (p X1 X3)))))

// skolemized version of
// (ax_step : Prf (\ (\ X, \ (\ Y, (p X (s Y)))))
(ax_step : Prf (\ (\ X, (p X (s (f X)))))
(ax_congr : Prf (\ (\ X1 , \ (\ X2 ,
        (p X1 X2) \rightarrow (p (s X1) (s X2)))))
(ax_goal : Prf (\ (\ X, ((p a (s (s X)))))
: Prf ⊥
:= ax_goal (\I (\ X, p a (s (s X))) (f (f a))
(ax_tran a (s (f a)) (s (s (f (f a))))
(ax_step a)
(ax_congr (f a) (s (f (f a))) (ax_step (f a)))));
```
Example proof without Skolem symbol

generated by Skonverto

\[\text{symbol goal}\\(\text{ax\_tran} : \text{Prf} ( \forall (\lambda X_1, \forall (\lambda X_2, \forall (\lambda X_3, (p X_1 X_2) \Rightarrow ((p X_2 X_3) \Rightarrow (p X_1 X_3))))) )\]\n
\(\text{ax\_step} : \text{Prf} ( \forall (\lambda X, \exists (\lambda Y, (p X (s Y)))) )\)

\(\text{ax\_congr} : \text{Prf} ( \forall (\lambda X_1, \forall (\lambda X_2, (p X_1 X_2) \Rightarrow (p (s X_1) (s X_2)))) )\)

\(\text{ax\_goal} : \text{Prf} ( \neg (\exists (\lambda X_4, ((p a (s (s X_4))))))) )\)

\( : \text{Prf} \bot \)

\[\text{ax\_goal} (\lambda r h, \exists E (\lambda z, p a (s z)) (\text{ax\_step} a) r (\lambda z a_1, \exists E (\lambda z_0, p z (s z_0)) (\text{ax\_step} z) r (\lambda z_0 a_2, h z_0 (\text{ax\_tran} a (s z) (s (s z_0)) a_1 (\text{ax\_congr} z (s z_0) a_2)))));\]
Conclusion

Instrumenting a prover to produce Dedukti proofs
• good if you start your prover from scratch

Reconstructing proofs
• more adapted for existing provers
• cannot reconstruct all proofs
• useful for proof assistants using provers internally
  e.g. PVS, Atelier B
Putting everything together

Dedukti \rightarrow \text{Formula} \rightarrow \text{ATP} \rightarrow \text{Ekstrakto + Skonverta}