Proofs in Dedukti

EuroProofNet WG2 meeting/PAAR Workshop

Guillaume Burel

Friday August 12th, 2022

Samovar, ENSIIE

Challenges in Automated Theorem Proving

Cooperation

- between provers/solvers
- with proof assistants

Trust

- checkable proofs
- reproducibility

Dedukti as a solution

Proof interoperability

- bridge between proof systems

Proof (re)checking

- from various sources

Outline

- Introduction
- What is Dedukti
- Intrumenting provers for Dedukti proof production
- Reconstructing proofs
- Conclusion

Dedukti

A logical framework

- a tool in which logical systems can be expressed
  - logics
  - calculi
  - proofs

Theoretical foundations

$\lambda\Pi$-calculus modulo theory

based on the Curry-Howard-De Bruijn correspondence

- λ-calculus with dependent types

enhanced with rewriting

- equate terms/formulas that are congruent

(At least) Two implementations

Dedukti	Core system proof checker `https://github.com/Deducteam/Dedukti`
lambdapi	more interactive (proof assistant) tactics, new friendlier syntax `https://github.com/Deducteam/lambdapi`

Why Dedukti?

Universal

- can embed a wide range of logics
  FOL, HOL, CoC, ...
- inputs from various tools
  ATPs, HOL Light, Isabelle/HOL, Matita, Coq, ...
- outputs to various tools
  Coq, Lean, PVS, Matita, OpenTheory (see `logipedia.science`)

Simple

- small kernel
- can be reimplemented independently

Efficient

- can check proofs > 50GB

Heart of Dedukti

Declaration of symbols

- with their type

```
symbol name : type;
```

Declaration of rewrite rules

```
rule left_hand side ↪ right_hand side;
```

Example

```
symbol nat : TYPE;
symbol 0 : nat;
symbol S : nat → nat;
symbol + : nat → nat → nat;
notation + infix left 10;

rule 0 + $x ↪ $x;
rule S $y + $x ↪ S ($y + $x);
```

Proof checking?

When a rule is declared

- subject reduction is checked:
  However the left-hand side can be typed,
  the right-hand side can be typed with the same type
  (modulo previous rewrite rules)

Example of checking

```
symbol nat : TYPE;
symbol 0 : nat;
symbol S : nat → nat;
symbol + : nat → nat → nat;
notation + infix left 10;

rule 0 + $x ↪ $x;
rule S $y + $x ↪ S ($y + $x);

symbol P : nat → TYPE;
symbol f : Π x : nat, P x;
rule f (0 + S $x) ↪ f (S 0 + $x);
```

Embedding a logic into Dedukti

Type for propositions:

```
symbol Prop : TYPE;
```

Deep embedding of connectives:

```
symbol ⊥ : Prop;
symbol ⇒ : Prop → Prop → Prop;
notation ⇒ infix right 10;
```

$\Pi\, \mathtt{p} : \mathtt{Prop},\ ((\mathtt{p} \Rightarrow \bot) \Rightarrow \bot) \Rightarrow \mathtt{p}$

First order

Type for term sorts:

```
symbol Set : TYPE;
symbol ι : Set;
```

Embedding `Set` terms into Dedukti terms:

```
symbol El : Set → TYPE;
```

Deep embedding of quantifiers:

```
symbol ∀ : Π a : Set, (El a → Prop) → Prop;
```

$(\forall x, p\ x) \Rightarrow p\ c$ embedded as `∀ ι (λ x, p x) ⇒ p c`

Proofs

Embedding `Prop` into Dedukti level:

```
symbol Prf : Prop → TYPE;
```

Making connectives more shallow:

```
rule Prf ($x ⇒ $y) ↪ Prf $x → Prf $y;
rule Prf ⊥ ↪ Π r, Prf r;
rule Prf (∀ $s $p) ↪ Π x : El $s, Prf ($p x);
```

```
Prf (∀ι (λ x, p x) ⇒ p c)
≡ Prf (∀ι (λ x, p x)) → Prf (p c)
≡ (Π x : El ι, Prf (p x)) → Prf (p c)
```

Dedukti terms as proofs

Proving $(\forall x, p\ x) \Rightarrow p\ c$

```
symbol my_theorem :
  Π c : El ι,
  Π p : El ι → Prop,
  Prf (∀ ι (λ x, p x) ⇒ p c);

rule my_theorem $c $p ↪ λ pp, pp $c;
```

Alternative syntax:

```
symbol my_theorem_alt (c : El ι) (p : El ι → Prop) :
  Prf (∀ ι (λ x, p x) ⇒ p c)
≔ λ pp, pp c;
```

Outline

Trusting automated theorem provers

Automated theorem provers:

- 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 at least proof traces (*i.e.* big steps)

Trusting ATPs

To increase confidence:

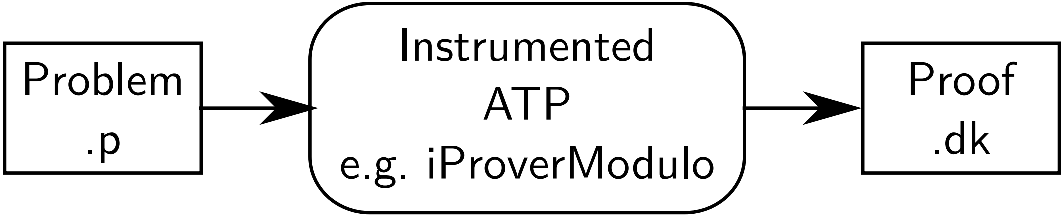
- either build a certified proof checker for proof traces
  - e.g. Coq certified proof checker for DRAT proof traces of SAT solvers
- or directly produce a proof checkable by your favorite assistant

Trusting ATPs

To increase confidence:

- either build a certified proof checker for proof traces
  - e.g. Coq certified proof checker for DRAT proof traces of SAT solvers
- or directly produce a proof checkable by your favorite assistant

Instrumenting a prover to produce a proof



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 on the complexity of the code/the proof calculus
- easier if a proof output was designed from the start (e.g. in Zenon)

Can only be done for a few provers

Provers outputting Dedukti proofs

iProverModulo: extension of iProver to handle Deduction Modulo Theory
`https://github.com/gburel/iProverModulo.git`

Zenon Modulo: extension of Zenon to handle Deduction Modulo Theory and 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. D[FOL]

Then, two approaches:

- Directly translating proofs into Dedukti
  - iProverModulo
- Embedding the proof calculus into Dedukti
  - Zenon Modulo

Zenon Modulo

[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
$\simeq$ bottom-up sequent-calculus with metavariables

$$\frac{P, \neg P}{\odot}\odot \qquad \frac{\neg(A \Rightarrow B)}{\neg A, B}\alpha_{\neg\Rightarrow} \qquad \frac{\neg(A \land B)}{\neg A \quad \mid \quad \neg B}\beta_{\neg\land}$$

Example, proof by refutation of $P \Rightarrow (P \land P)$:

$$\cfrac{\neg(P \Rightarrow (P \land P))}{\cfrac{P}{\cfrac{\neg(P \land P)}{\cfrac{\neg P}{\odot}\odot \qquad \cfrac{\neg P}{\odot}\odot}\beta_{\neg\land}}}\alpha_{\neg\Rightarrow}$$

Deep embedding of proof calculus

$$\frac{P, \neg P}{\odot} \odot :$$

```
symbol Rax p : Prf p → Prf (¬p) → Prf ⊥;
```

$$\frac{\neg(A \Rightarrow B)}{\neg A, B} \alpha_{\neg\Rightarrow} :$$

```
symbol R¬⇒ a b : (Prf a → Prf (¬b) → Prf ⊥) → Prf (¬(a ⇒ b)) → Prf ⊥;
```

$$\frac{\neg(A \wedge B)}{\neg A \quad | \quad \neg B} \beta_{\neg\wedge} :$$

```
symbol R¬∧ a b : (Prf (¬ a) → Prf ⊥) → (Prf (¬ b) → Prf ⊥) →
                  Prf (¬ (a ∧ b)) → Prf ⊥;
```

Deep translation of the example

(after η-reduction to make it more readable)

```
opaque symbol goal : Prf ᶜ (p ⇒ (p ∧ p)) ≔
  R¬⇒ p (p ∧ p)
    (λ π, R¬∧ p p (Rax p π) (Rax p π));
```

Making the embedding more shallow

Defines Tableaux rules as Dedukti terms,
prove that they are derivable in FOL:

```
rule Rax ↪ λ p h π, π h;
rule R¬∧ ↪ λ p q h1 h2 h3,
  h1 (λ h5, h2 (λ h6, h3 (λ r π, π h5 h6)));
rule R⇒ ↪ λ p q h1 h2,
  h2 (λ h3, h1 h3 (λ h4, h2 (λ _, h4)) q);
```

Shallow proof from the example

```
assert ⊢ goal : Prf ᶜ (p ⇒ (p ∧ p));
assert ⊢ goal ≡
  λ h2, h2 (λ h3, h2 (λ _ _ π, π h3 h3) (p ∧ p));
```

Outline

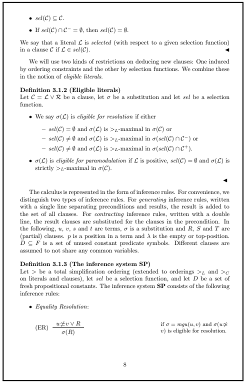
- Introduction
- What is Dedukti
- Intrumenting provers for Dedukti proof production
- Reconstructing proofs
- Conclusion

Limits of instrumentation

Provers can be hard to instrument to produce exact 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



- $sel(\mathcal{C}) \subseteq \mathcal{C}$.
- If $sel(\mathcal{C}) \cap \mathcal{C}^- = \emptyset$, then $sel(\mathcal{C}) = \emptyset$.

We say that a literal $\mathcal{L}$ is *selected* (with respect to a given selection function) in a clause $\mathcal{C}$ if $\mathcal{L} \in sel(\mathcal{C})$. ◀

We will use two kinds of restrictions on deducing new clauses: One induced by ordering constraints and the other by selection functions. We combine these in the notion of *eligible literals*.

Definition 3.1.2 (Eligible literals)
Let $\mathcal{C} = \mathcal{L} \vee \mathcal{R}$ be a clause, let σ be a substitution and let *sel* be a selection function.

- We say $\sigma(\mathcal{L})$ is *eligible for resolution* if either
  - $sel(\mathcal{C}) = \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(\mathcal{C})$ or
  - $sel(\mathcal{C}) \neq \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(sel(\mathcal{C}) \cap \mathcal{C}^-)$ or
  - $sel(\mathcal{C}) \neq \emptyset$ and $\sigma(\mathcal{L})$ is $>_L$-maximal in $\sigma(sel(\mathcal{C}) \cap \mathcal{C}^+)$.
- $\sigma(\mathcal{L})$ is *eligible for paramodulation* if $\mathcal{L}$ is positive, $sel(\mathcal{C}) = \emptyset$ and $\sigma(\mathcal{L})$ is strictly $>_L$-maximal in $\sigma(\mathcal{C})$.

◀

The calculus is represented in the form of inference rules. For convenience, we distinguish two types of inference rules. For *generating* inference rules, written with a single line separating preconditions and results, the result is added to the set of all clauses. For *contracting* inference rules, written with a double line, the result clauses are substituted for the clauses in the precondition. In the following, u, v, s and t are terms, σ is a substitution and R, S and T are (partial) clauses. p is a position in a term and λ is the empty or top-position. $D \subseteq F$ is a set of unused constant predicate symbols. Different clauses are assumed to not share any common variables.

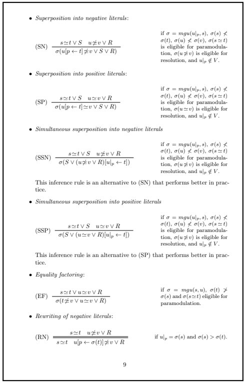
Definition 3.1.3 (The inference system SP)
Let $>$ be a total simplification ordering (extended to orderings $>_L$ and $>_C$ on literals and clauses), let *sel* be a selection function, and let D be a set of fresh propositional constants. The inference system **SP** consists of the following inference rules:

- *Equality Resolution*:

$$\text{(ER)} \quad \frac{u \not\simeq v \vee R}{\sigma(R)}$$

if $\sigma = mgu(u, v)$ and $\sigma(u \not\simeq v)$ is eligible for resolution.

8



- *Superposition into negative literals*:

$$\text{(SN)} \quad \frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(u[p \leftarrow t] \not\simeq v \vee S \vee R)}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

- *Superposition into positive literals*:

$$\text{(SP)} \quad \frac{s \simeq t \vee S \quad u \simeq v \vee R}{\sigma(u[p \leftarrow t] \simeq v \vee S \vee R)}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \simeq v)$ is eligible for resolution, and $u|_p \notin V$.

- *Simultaneous superposition into negative literals*

$$\text{(SSN)} \quad \frac{s \simeq t \vee S \quad u \not\simeq v \vee R}{\sigma(S \vee (u \not\simeq v \vee R)[u|_p \leftarrow t])}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

This inference rule is an alternative to (SN) that performs better in practice.

- *Simultaneous superposition into positive literals*

$$\text{(SSP)} \quad \frac{s \simeq t \vee S \quad u \simeq v \vee R}{\sigma(S \vee (u \simeq v \vee R)[u|_p \leftarrow t])}$$

if $\sigma = mgu(u|_p, s)$, $\sigma(s) \not< \sigma(t)$, $\sigma(u) \not< \sigma(v)$, $\sigma(s \simeq t)$ is eligible for paramodulation, $\sigma(u \not\simeq v)$ is eligible for resolution, and $u|_p \notin V$.

This inference rule is an alternative to (SP) that performs better in practice.

- *Equality factoring*:

$$\text{(EF)} \quad \frac{s \simeq t \vee u \simeq v \vee R}{\sigma(t \not\simeq v \vee u \simeq v \vee R)}$$

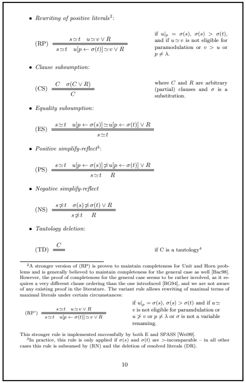
if $\sigma = mgu(s, u)$, $\sigma(t) \not> \sigma(s)$ and $\sigma(s \simeq t)$ eligible for paramodulation.

- *Rewriting of negative literals*:

$$\text{(RN)} \quad \frac{s \simeq t \quad u \not\simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \not\simeq v \vee R}$$

if $u|_p = \sigma(s)$ and $\sigma(s) > \sigma(t)$.

9



- *Rewriting of positive literals*[2]:

$$\text{(RP)} \quad \frac{s \simeq t \quad u \simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \simeq v \vee R}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$, and if $u \simeq v$ is not eligible for paramodulation or $v > u$ or $p \neq \lambda$.

- *Clause subsumption*:

$$\text{(CS)} \quad \frac{C \quad \sigma(C \vee R)}{C}$$

where C and R are arbitrary (partial) clauses and σ is a substitution.

- *Equality subsumption*:

$$\text{(ES)} \quad \frac{s \simeq t \quad u[p \leftarrow \sigma(s)] \simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t}$$

- *Positive simplify-reflect*[3]:

$$\text{(PS)} \quad \frac{s \simeq t \quad u[p \leftarrow \sigma(s)] \not\simeq u[p \leftarrow \sigma(t)] \vee R}{s \simeq t \quad R}$$

- *Negative simplify-reflect*

$$\text{(NS)} \quad \frac{s \not\simeq t \quad \sigma(s) \not\simeq \sigma(t) \vee R}{s \not\simeq t \quad R}$$

- *Tautology deletion*:

$$\text{(TD)} \quad \frac{C}{}$$

if C is a tautology[4]

[2] A stronger version of (RP) is proven to maintain completeness for Unit and Horn problems and is generally believed to maintain completeness for the general case as well [Bac98]. However, the proof of completeness for the general case seems to be rather involved, as it requires a very different clause ordering than the one introduced [BG94], and we are not aware of any existing proof in the literature. The variant rule allows rewriting of maximal terms of maximal literals under certain circumstances:

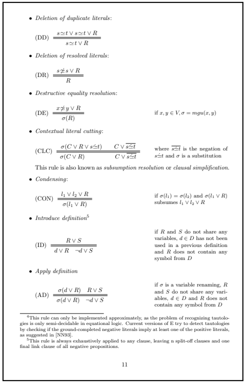
$$\text{(RP')} \quad \frac{s \simeq t \quad u \simeq v \vee R}{s \simeq t \quad u[p \leftarrow \sigma(t)] \simeq v \vee R}$$

if $u|_p = \sigma(s)$, $\sigma(s) > \sigma(t)$ and if $u \simeq v$ is not eligible for paramdulation or $u \not> v$ or $p \neq \lambda$ or σ is not a variable renaming.

This stronger rule is implemented successfully by both E and SPASS [Wei99].

[3] In practice, this rule is only applied if $\sigma(s)$ and $\sigma(t)$ are $>$-incomparable – in all other cases this rule is subsumed by (RN) and the deletion of resolved literals (DR).

10



- *Deletion of duplicate literals*:

$$\text{(DD)} \quad \frac{s \simeq t \vee s \simeq t \vee R}{s \simeq t \vee R}$$

- *Deletion of resolved literals*:

$$\text{(DR)} \quad \frac{s \not\simeq s \vee R}{R}$$

- *Destructive equality resolution*:

$$\text{(DE)} \quad \frac{x \not\simeq y \vee R}{\sigma(R)}$$

if $x, y \in V, \sigma = mgu(x, y)$

- *Contextual literal cutting*:

$$\text{(CLC)} \quad \frac{\sigma(C \vee R \vee s \dot{\simeq} t) \qquad C \vee \overline{s \dot{\simeq} t}}{\sigma(C \vee R) \qquad C \vee \overline{s \dot{\simeq} t}}$$

where $\overline{s \dot{\simeq} t}$ is the negation of $s \dot{\simeq} t$ and σ is a substitution

This rule is also known as *subsumption resolution* or *clausal simplification*.

- *Condensing*:

$$\text{(CON)} \quad \frac{l_1 \vee l_2 \vee R}{\sigma(l_1 \vee R)}$$

if $\sigma(l_1) = \sigma(l_2)$ and $\sigma(l_1 \vee R)$ subsumes $l_1 \vee l_2 \vee R$

- *Introduce definition*[5]

$$\text{(ID)} \quad \frac{R \vee S}{d \vee R \quad \neg d \vee S}$$

if R and S do not share any variables, $d \in D$ has not been used in a previous definition and R does not contain any symbol from D

- *Apply definition*

$$\text{(AD)} \quad \frac{\sigma(d \vee R) \quad R \vee S}{\sigma(d \vee R) \quad \neg d \vee S}$$

if σ is a variable renaming, R and S do not share any variables, $d \in D$ and R does not contain any symbol from D

[4] This rule can only be implemented approximately, as the problem of recognizing tautologies is only semi-decidable in equational logic. Current versions of E try to detect tautologies by checking if the ground-completed negative literals imply at least one of the positive literals, as suggested in [NN93].

[5] This rule is always exhaustively applied to any clause, leaving n split-off clauses and one final link clause of all negative propositions.

11

Proof trace

But often, provers produce at least a proof trace:

- list of formulas that were derived to obtain the proof
- sometimes with more informations
  - premises
  - name of the inference rules
  - theory
  - ...

Example of trace: TSTP format

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

List of formulas

- each annotated by an inference tree whose leafs are other formulas

```
cnf(c_0_60,plain,
    ( join(X1,join(X2,X3)) = join(X2,join(X1,X3)) ),
    inference(rw,[status(thm)],
      [inference(spm,[status(thm)],[c_0_30,c_0_18]),
       c_0_30])).
```

Example of trace: TSTP format

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

List of formulas

- each annotated by an inference tree whose leafs are other formulas

```
cnf(c_0_60,plain,
    ( join(X1,join(X2,X3)) = join(X2,join(X1,X3)) ),
    inference(rw,[status(thm)],
      [inference(spm,[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:

- Reprove each step using a Dedukti producing tool
- Combine the proofs of the steps 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 reprove the steps

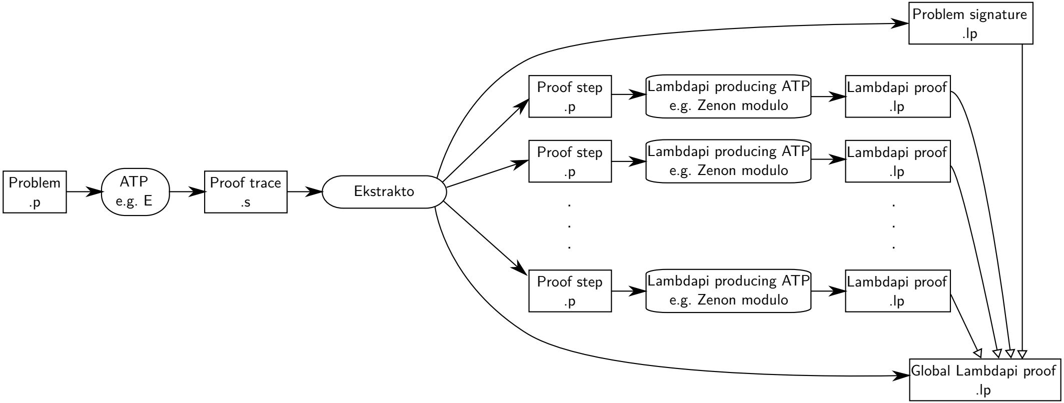
Ekstrakto

[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:

- Zenon modulo and ArchSat

Results

Percentage of Lambdapi proofs on the extracted TPTP files

Prover	% **E**	% VAMPIRE
ZenonModulo	87%	60%
ArchSAT	92%	81%
ZenonModulo ∪ *ArchSAT*	95%	85%

Percentage of complete Lambdapi proofs

Prover	% **E** TSTP	% VAMPIRE TSTP
ZenonModulo	45%	54%
ArchSAT	56%	74%
ZenonModulo ∪ *ArchSAT*	69%	83%

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 Ekstrakto cannot work for them

Main instance: Skolemization

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

Present in the CNF transformation used by almost all ATPs

Skonverto

[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

Content

Implementation of a constructive proof of Skolem theorem by [Dowek and Werner 2005]

- in the context of first-order natural deduction

```
symbol axiom : Prf (∀ (λ X, ∃ (λ Y, (p X (s Y)))));

symbol goal
  (ax_tran : Prf (∀ (λ X1 : El ι, ∀ (λ X2 : El ι, ∀ (λ X3 : El
     (p X1 X2) ⇒ ((p X2 X3) ⇒ (p X1 X3))))))))
  (ax_step : Prf (∀ (λ X1 : El ι, (p X1 (s (f X1))))))
  (ax_congr : Prf (∀ (λ X1 : El ι, ∀ (λ X2 : El ι,
     (p X1 X2) ⇒ (p (s X1) (s X2)))))))
  (ax_goal : Prf (¬ (∃ (λ X4 : El ι, ((p a (s (s X4))))))))
  : Prf ⊥
 ≔  ax_goal  (∃I (λ X4 : El ι, p a (s (s X4))) (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)))));
```

```
symbol goal
  (ax_tran : Prf (∀ (λ X1 : El ι, ∀ (λ X2 : El ι, ∀ (λ X3 : El
     (p X1 X2) ⇒ ((p X2 X3) ⇒ (p X1 X3)))))))
  (ax_step : Prf (∀ (λ X, ∃ (λ Y, (p X (s Y))))))
  (ax_congr : Prf (∀ (λ X1 : El ι, ∀ (λ X2 : El ι,
     (p X1 X2) ⇒ (p (s X1) (s X2)))))))
  (ax_goal : Prf (¬ (∃ (λ X4 : El ι, ((p a (s (s X4))))))))
  : Prf ⊥
≔ ax_goal (λ r h, ∃E (λ z, p a (s z)) (ax_step a) r
        (λ z a1, ∃E (λ z0, p z (s z0)) (ax_step z) r
        (λ z0 a2, h z0 (ax_tran a (s z) (s (s z0)) a1
            (ax_congr z (s z0) a2))))));
```

Outline

- Introduction
- What is Dedukti
- Intrumenting provers for Dedukti proof production
- Reconstructing proofs
- Conclusion

Conclusion

Dedukti as a universal back-end for proof checking and interoperability

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
- also for proof assistants
    - PVS, Atelier B