Dependently typed superposition in Lean

Gabriel Ebner
Matryoshka 2018
2018-06-27

TU Wien
Introduction

Calculus

Implementation

Conclusion
Lean

- Proof assistant based on dependent type theory
  - terms, types, formulas, proofs are all expressions
  - small kernel (unlike Coq)

- uses axiom of choice and classical logic
  - but avoided outside of proofs

- Tactics/metaprograms are defined in the object language
Syntax

c -- constants
x -- variables
t s
λ x : t, s
Π x : t, s -- often written ∀ or →
Sort u -- Prop, Type

• e.g.

∀ α : Type, ∀ β : Type, ∀ r : ring α,
∀ m : module α β r, ∀ x : β,
1 · x = x
Syntax

c  -- constants
x  -- variables
t s
\lambda x : t, s
\Pi x : t, s  -- often written \forall or \rightarrow
Sort u  -- Prop, Type

• e.g.

  \forall \alpha : Type, \forall \beta : Type, \forall r : \text{ring} \alpha,
  \forall m : \text{module} \alpha \beta r, \forall x : \beta,
  \text{smul} \alpha \beta r m (\text{one} \alpha r) x = x
• Superposition prover

• Implemented 100% in Lean
  • First “large” metaprogram

• Uses Lean expressions, unification, proofs

• 2200 lines of code
  • (including toy SAT solver)

• Think of metis in Isabelle
Intended logic

• Complete for first-order logic with equality

• Higher-order not a focus
  • but don't fail on lambdas
  • no encoding for applicative FOL

• Some inferences for inductive data types
Introduction

Calculus

Implementation

Conclusion
• How to represent a clause $a, b \vdash c, d$?

1. $\neg a \lor \neg b \lor c \lor d$

2. $a \rightarrow b \rightarrow c \lor d$

3. $a \rightarrow b \rightarrow \neg c \rightarrow \neg d \rightarrow \text{false}$
   • also used in Coq by Bezem, Hendriks, de Nivelle 2002
Intuitionistic reasoning

• Actually: $a \rightarrow b \rightarrow (c \rightarrow F) \rightarrow (d \rightarrow F) \rightarrow F$
  
  • ($F$ is a definition for the original goal)

→ often intuitionistic proofs
  
  • e.g. for assumptions like $\forall x, \land A \rightarrow \lor B$
  • want to avoid classical reasoning on types

• also makes use of decidable instances
Quantifiers

\[ \forall \alpha, \forall \beta, \forall r : \text{ring } \alpha, \]
\[ \forall m : \text{module } \alpha \beta r, \forall x : \beta, \]
\[ (\text{smul } \alpha \beta r m (\text{one } \alpha r) x = x \implies F) \implies F \]

- only perform inferences on non-dependent literals
- when literals are resolved away, we get new non-dependent literals
∀ α, ∀ β, ∀ r : ring α, module α β r → β → F

• only perform inferences on non-dependent literals

• when literals are resolved away, we get new non-dependent literals
∀ α, ∀ β, 
ring α → β → F

• only perform inferences on non-dependent literals

• when literals are resolved away, we get new non-dependent literals
Skolemization

• Skolemization not sound in general
  • requires non-empty domain

→ add extra (implicit) argument to Skolem function
  • $\forall x, P \rightarrow \exists y : \alpha, Q x y$ becomes
    $\forall x, \forall z : \alpha, P \rightarrow \forall x, Q x (fzx)$

• automatically discharged for nonempty instances
Refinements

- Subsumption
- Term ordering
- Literal selection
- Demodulation
- Splitting (Avatar)
Term ordering

- Standard lexicographic path order

- Curried applications $fabc$ are treated as $f(a, b, c)$

- Type parameters, type-class instances are also arguments!
  - does not seem to be a performance problem

- $\lambda$ and $\Pi$ expressions are treated as (unknown) variables
Avatar-style splitting

- Splits clause into variable-disjoint components

\[
\begin{align*}
\vdash Px, Qy & \quad \vdash Px, Qy \\
\vdash Px & \quad \vdash Qy
\end{align*}
\]
Avatar-style splitting

• Splits clause into variable-disjoint components

\[ \frac{\vdash Px, Qy}{s_1 \vdash Px} \quad \frac{\vdash Px, Qy}{s_2 \vdash Qy} \quad \vdash s_1, s_2 \]

• \( s_1 := \forall x \ Px \)
• \( s_2 := \forall y \ Qy \)

• No inferences are performed on these splitting atoms
→ sent to SAT solver instead
Lean-specific rules

\[
\frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta}
\]
if \(A\) has a type-class instance

• such as inhabited, is_associative, ...

\[
\frac{\Gamma \vdash \Delta, \text{cons } a \text{ } b = \text{cons } c \text{ } d}{\Gamma \vdash \Delta, \text{cons } a \text{ } b = \text{nil}, \Gamma \vdash \Delta}
\]
\[
\frac{\Gamma \vdash \Delta, a = c \land b = d}{\Gamma \vdash \Delta, a = c \land b = d}
\]
General approach

• reuse built-in data structures
  • expressions
  • proofs
  • unifier

• Lean’s unifier essentially does:
  • pattern unification
  • definitional reduction
  • some heuristics

• rewriting only at first-order argument positions
Proof construction

• unification, type inference only work with local context

→ clauses are stored in the local context:
  
  cls_0: A → (B → F) → F
  cls_1: (A → F) → F
  cls_2: B → F

\[ \vdash F \]

⚠️ does not work with universe polymorphism
Proof construction

- unification, type inference only work with local context

→ clauses are stored in the local context:

cls_0: A → (B → F) → F
cls_1: (A → F) → F
cls_2: B → F
cls_3: (B → F) → F

⊢ F

⚠ does not work with universe polymorphism
Proof construction

- unification, type inference only work with local context

→ clauses are stored in the local context:

- \( \text{cls}_0 : A \rightarrow (B \rightarrow F) \rightarrow F \)
- \( \text{cls}_1 : (A \rightarrow F) \rightarrow F \)
- \( \text{cls}_2 : B \rightarrow F \)
- \( \text{cls}_3 : (B \rightarrow F) \rightarrow F \)
- \( \text{cls}_4 : F \)

\[ \vdash F \]

⚠ does not work with universe polymorphism
Proof construction

• unification, type inference only work with local context

→ clauses are stored in the local context:
  cls_0: A → (B → F) → F
  cls_1: (A → F) → F
  cls_2: B → F
  cls_3: (B → F) → F
  cls_4: F
  ⊢ F

• avoids exponential blowup

• post-processing step to remove unused subproofs

⚠ does not work with universe polymorphism
SAT proof construction

- For each literal on the trail, store proof of:
  \[ A \quad A \rightarrow F \quad (A \rightarrow F) \rightarrow F \]

  - decision literals have fresh local constants
  - propagated literals have actual proofs

- on conflict, add lambdas for the decision literals

- produces intuitionistic proofs
meta structure prover_state :=
(active : rb_map clause_id derived_clause)
(passive : rb_map clause_id derived_clause)
(newly_derived : list derived_clause)
(prec : list expr)
-- ...

meta def prover := state_t prover_state tactic
Introduction

Calculus

Implementation

Conclusion
Universe polymorphism

• Hypothesis in the local context cannot be universe polymorphic
  • You can’t even have $\forall x, ~ x + [ ] = x$ for all types

• Possible workaround: create a new environment
  • extra type-checking
  • not intended use (errors and warnings are printed directly)

→ manually implement proof handling
Performance: metavariables

• For unification, we instantiate \( \forall x \, Px \rightarrow F \) as \( P\_m\_1 \rightarrow F \)
  • built-in unification uses metavariables

• Afterwards, we quantify over the free metavariables

• Pretty slow

→ Lean 4 will expose temporary metavariables
Performance: unifier

• unpredictable performance

• performance problem even with few clauses

→ do some prefiltering

→ implement term indexing
  • non-trivial, idiomatic code relies on definitional equality
Other future work

• Simplifier integration
  • different term order for $x \cdot y = y \cdot x$

• AC redundancy checks

• Heterogeneous equality, congruence lemmas
Conclusion

- Not yet production-ready
  - performance subpar
  - missing support for universe polymorphism

- Lean 4 should bring useful APIs
  - temporary metavariables

- long term: proof reconstruction for “leanhammer”