Integration of general-purpose automated theorem provers in Lean

Gabriel Ebner

Formal Methods in Mathematics
2020-01-08

Vrije Universiteit Amsterdam

Hammers

- “Magic button that proves all your theorems”
- e.g. Sledgehammer for Isabelle/HOL
  - popular, also: HOLyHammer, CoqHammer, etc.
- User-friendly integration of automated reasoning tools in proof assistants

General idea

```
example (x y z : nat) : x.gcd y | (x*z).gcd y :=
by hammer
```

General purpose: should work for anything, no setup

Typical setup

1. Find already proven lemmas that look "useful"
   ("premise selection", "relevance filter")

2. Pass lemmas and goal to efficient external prover (e.g. Vampire, E, etc.)
   - Requires encoding into logic of prover

3. Import generated proof
   - Popular strategy: mine names of used lemmas,
     and reconstruct using slow prover

Features

(Based on approach in CoqHammer (Czaja, Kaliszyk 2018))

Assign to every lemma a set of features based on its type:

- Every constants c that occurs in the type
- The pair (f, g) for every subterm $f a_1 \ldots (g \ldots) \ldots a_n$

Ignore:

- eq, and, ...
- Type classes, and type class instance arguments.

Implementation

- Cosine similarity with TF-IDF (term frequency-inverse document frequency)
  - Common way to calculate similarity between documents (= sequence/set of words) with lots of variations.
  - Here: document = lemma, word = feature.

1. Assign to every lemma the characteristic function of its feature set $\in \mathbb{R}^{|F|}$
2. Scale each coordinate by how rarely it occurs globally
3. Compute similarity of a and b as $\frac{a \cdot b}{\|a\| \|b\|}$

- Implemented in C++ (for performance reasons)

Issue: type classes

```
theorem le_of_lt { α} [preorder α] {a b : α} :
  a < b → a ≤ b :=
sorry
```

```
example (a b : nat) : ¬ a < b ∨ a ≤ b :=
by hammer
```

- Should find `le_of_lt` because it talks about the preorder `nat`, even though the name `preorder` does not occur in the goal.

```
theorem le_of_lt' {a b : nat} : a < b → a ≤ b :=
sorry
```

- Should not prefer `le_of_lt'` either.

Applicative translation

Translation to single-sorted first-order logic, like CoqHammer:

- Binary function $a(x, y)$ for application xy
- Predicate $p(x)$: (proposition) x is inhabited
- Relation $t(x, y)$: x has type y
- Equality is translated as equality.
- Constant s means `Type u`.

For each constant to be exported, we write one formula expressing its type.

Example translation

```
theorem nat.le_succ : ∀ (n : nat),
  @has_le.le.{0} nat nat.has_le n (nat.succ n)
```

$$\forall n, t(n, \mathtt{nat}) \to p(a(a(a(a(\mathtt{has\_le.le}, \mathtt{nat}), \mathtt{nat.has\_le}), n), a(\mathtt{nat.succ}, n)))$$

Example translation

```
theorem nat.le_succ : ∀ (n : nat),
  @has_le.le.{0} nat nat.has_le n (nat.succ n)
```

$\forall n, t(n, \texttt{nat}) \to p(a(a(a(a(\texttt{has\_le.le}, \texttt{nat}), \texttt{nat.has\_le}), n), a(\texttt{nat.succ}, n)))$

```
fof(cnat_o_le__succ, axiom,
 (![Xn_n3]: (t(Xn_n3, cnat) => p(a(a(a(a(chas__le_o_le, cnat
   ), cnat_o_has__le), Xn_n3), a(cnat_o_succ, Xn_n3)))))).
```

Unsoundness

Translation is unsound (= does not preserve unprovability).

$\rightarrow$ "spurious" proofs

Two main reasons:

1. Definitional equality and propositional equality are identified.
2. `Type u` and `Type (u+1)` are identified.

Type class coherence

We often need to show that two type class instances are equal.

E.g. if you want to apply `le_refl` to natural numbers:

```
p(a(a(a(a(chas__le_o_le, X_ga_n2),
  a(a(cpreorder_o_to__has__le, X_ga_n2), X__inst__1_n3)),
  Xa_n4),
  Xa_n4))
```

vs.

```
p(a(a(a(a(chas__le_o_le, cnat),
  cnat_o_has__le),
  Xx_n18),
  Xx_n18))
```

$\rightarrow$ Heuristically add extra equations relating type class instances.

Simply-typed higher-order logic

Types:

- Booleans
- Base types: `nat`, `list nat`, etc.
- Function types: $\tau_1 \to \tau_2$

Terms (formulas are terms of Boolean type):

- Constants: `nat.add`, etc.
- Application: ts
- Variable: x
- Lambdas: $\lambda x\, t$

(We use closed Lean expressions as names for constants and base types.)

Two phases

$$\text{Lean} \xrightarrow{\text{abstraction}} \text{HOL} \xrightarrow{\text{type instantiation}} \text{HOL}$$

- sound translation
- enables provers to do non-first-order reasoning
  - built-in support for $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \ldots$
  - synthesize lambdas
  - induction
- solves type class coherence issue
- mitigates issue with type classes in relevance filter

Abstraction

Turn `∀ {α : Type u} [preorder α] (a : α), a ≤ a`
into `∀ a :` ‘**`?m_1`**’`,` ‘**`@has_le.le ?m_1 ?m_2.to_has_le`**’ `a a`

- Replace non-HOL subterms by HOL constants.
  - dependent applications
  - pi types
  - types like `list nat`
  - …
- Instance-implicit arguments are also included in the constants.

Type instantiation

Turn `∀ a : ‘`**`?m_1`**`’, ‘`**`@has_le.le ?m_1 ?m_2.to_has_le`**`’ a a`
into `∀ a : ‘`**`nat`**`’, ‘`**`@has_le.le nat nat.preorder.to_has_le`**`’ a a`

- Unify the constants in the HOL terms
  - `@has_le.le ?m_1 ?m_2.to_has_le` occurs in lemma
  - `@has_le.le nat nat.has_le` occurs in goal

  → Instantiate lemma by unifying `?m_1 =?= nat` and `?m_2.to_has_le =?= nat.has_le`.
- Also solves additional type-class constraints. E.g. a lemma about Archimedian fields might have an assumption `archimedian α` which does not occur in any constant.

Limitations

- equality between types: `m = n → zmod m = zmod n`
  - → Bundle the non-type arguments? That is, translate to `Σ n, zmod n`.
- dependent families: `∀ i, fin i` is translated to a base type
- proof arguments:
  `@roption.get : ∀ α, ∀ o : roption α, o.dom → α`
  - Just elide them? (Only affects nonemptiness of α here.)

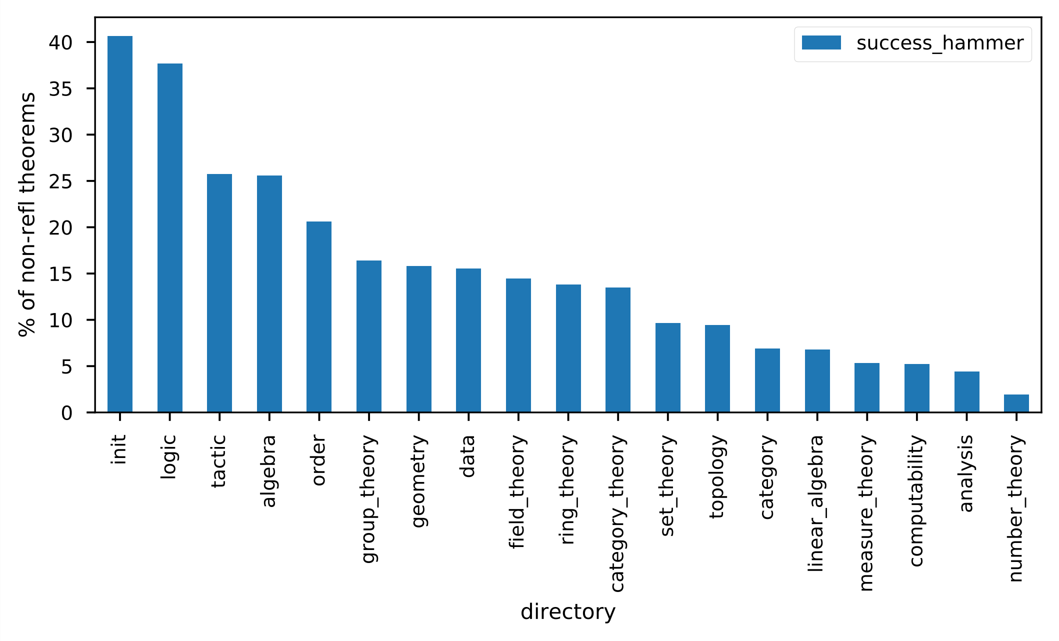
Implementation

- Basic relevance filter (in C++)
- Applicative first-order encoding
    - interfaces with Vampire
- HOL encoding
    - interfaces with E-HO
- Proof reconstruction using `super`
    - Small superposition prover written in (meta-)Lean

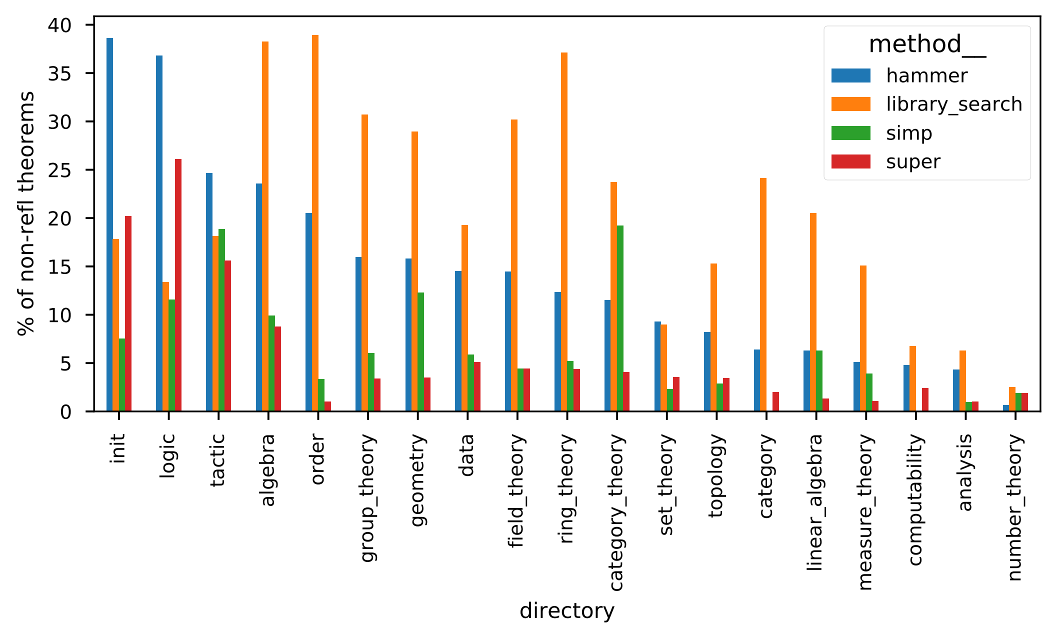
Experiment setup

- 31112 theorems in mathlib + core (everything that's a `declaration.thm`)
- Try tactic at the same point in the file as the theorem.
  - Applicative translation with 10 selected lemmas
  - Monomorphizing translation with 10/100 selected lemmas
  - `super` with 10 selected lemmas
  - `library_search`
  - `simp`
  - `refl`
- Time limit of 30s for external provers + `try_for 100000`
  - longest total runtime is 125s

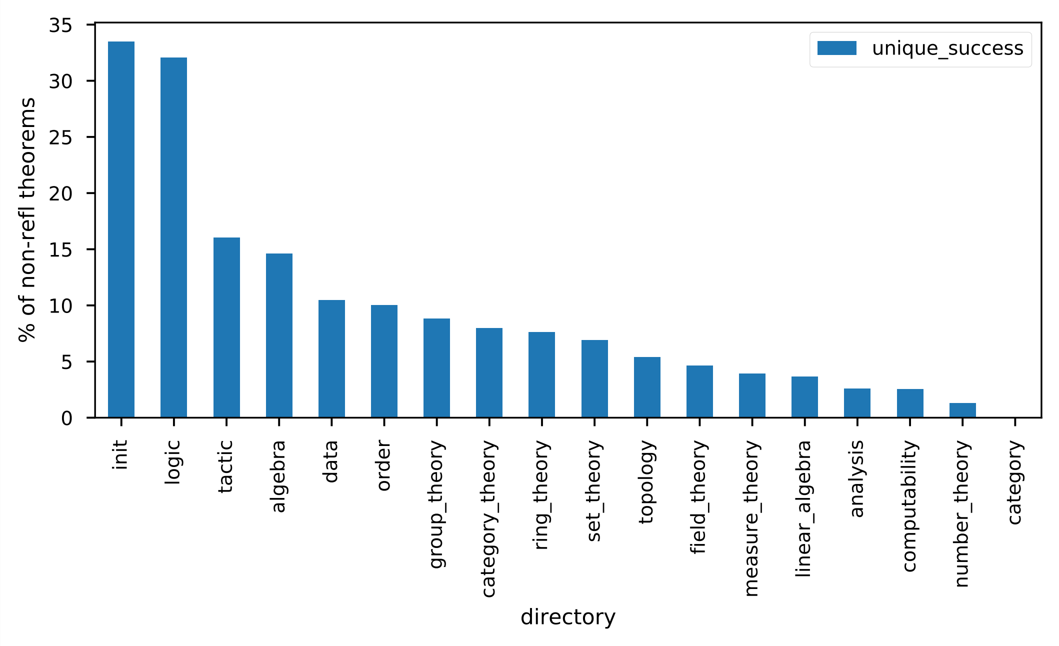
Success rate



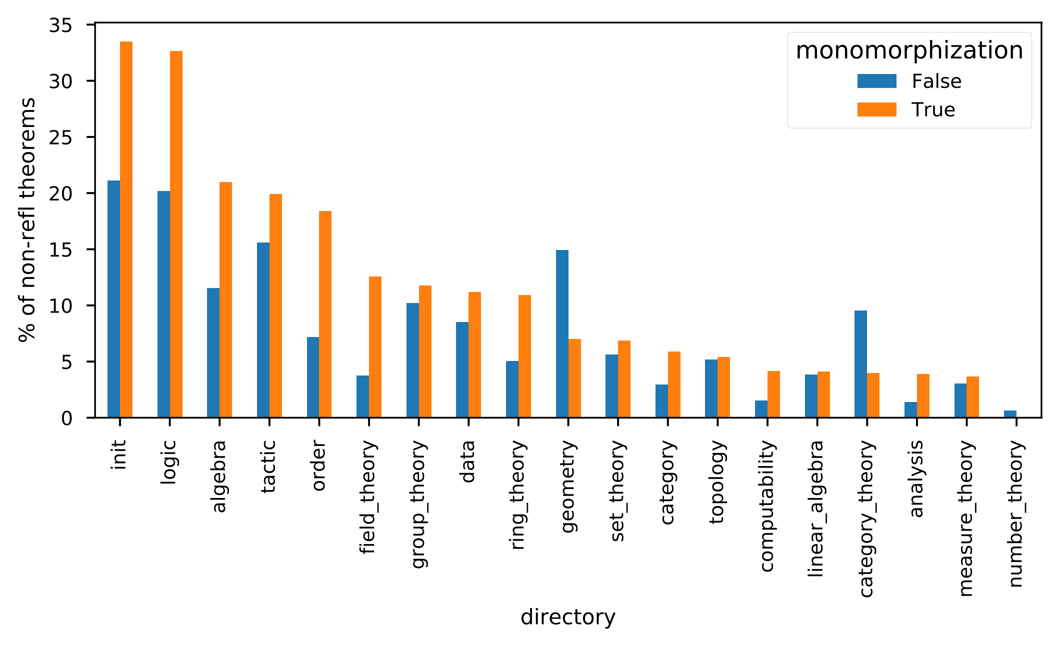
Success rate, compared



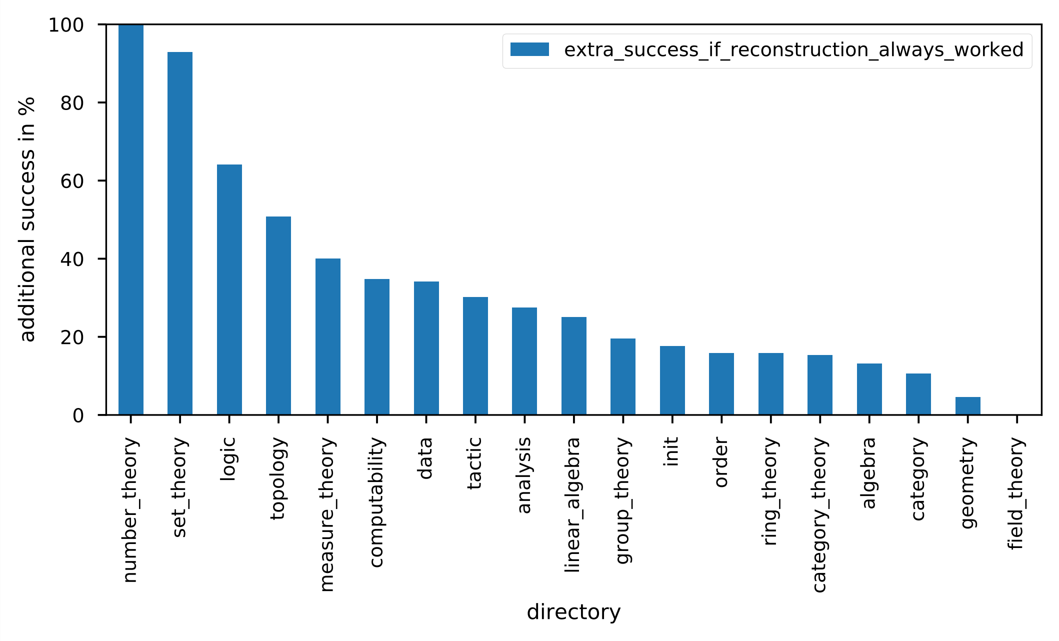
Unique successes (i.e., not by library_search or simp; incl. super)



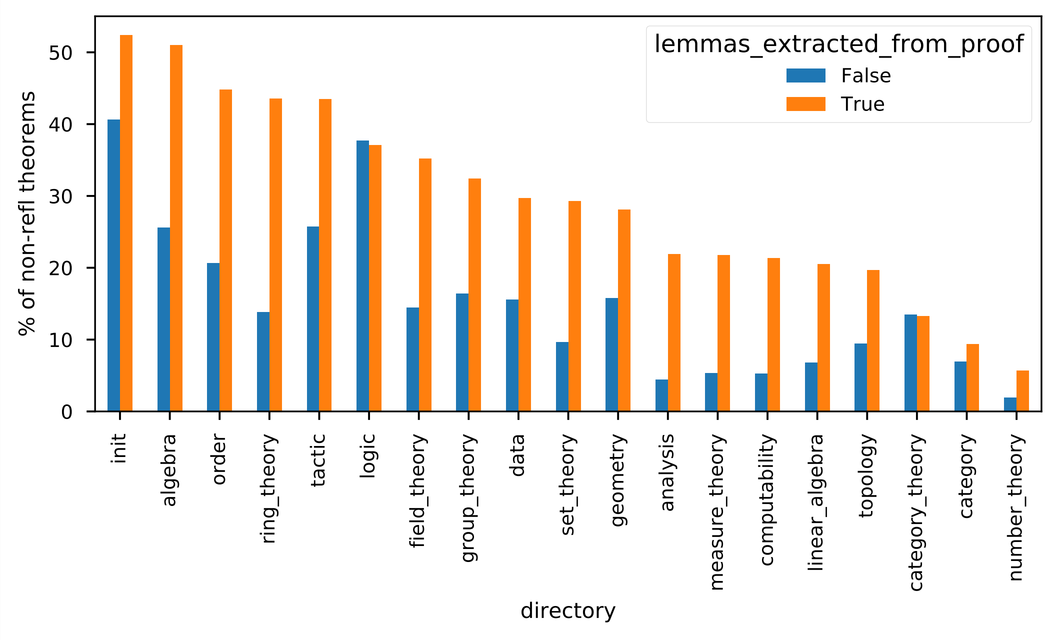
Effect of monomorphization



Robustness of reconstruction



Lots of room for improvements—lemma selection



Introduction

Premise selection

Applicative translation to FOL

Monomorphizing translation to HOL

Empirical results

Demonstration

Conclusion

Conclusion

- Library design (such as type classes) has an effect on hammers
- Promising results, there is lots of room for improvement
- Next steps:
  - Improve premise selection
  - Copy-pastable tactic snippets
  - Increase cleverness of HOL translation

Lots of room for improvements—parsing failures