RESEARCH INTERNSHIP REPORT

Meta-programming with the Lean proof assistant

Supervision by Jasmin C. Blanchette, Johannes Hölzl et Robert Y. Lewis

Vrije Universiteit Amsterdam

April - August 2018

Pablo Le Hénaff

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

VU VRIJE UNIVERSITEIT AMSTERDAM

Abstract

This report expounds work conducted on the Lean proof assistant at the Vrije Universiteit Amsterdam. We first present a short introduction to Lean, containing some of the necessary background. It is followed by the presentation of two original examples of how Lean's meta-programming framework can be used to define proof automation procedures. The first example assists user who work with algebraic structures such as commutative groups by simplifying equalities. The second sample uses recursion and user-defined lemmas to facilitate some proofs using monotonicity of a variety of operators. Finally, we explain our integration of an external tool into Lean, the Nunchaku model finder. Nunchaku can find witnesses for an existential quantification or counter-examples to a wrong assertion. It is sometimes useful as an heuristic to check that a goal can actually be proved, thus saving time not trying to prove a false assertion.

Acknowledgements. I am grateful to Jasmin C. Blanchette, for giving me the opportunity to do this internship; to Johannes Hölzl and Robert Y. Lewis, for the precious answers they gave to my numerous questions; to Alex and Petar, to whom I wish a lot of success for the rest of their PhD project; to Wan and Femke for sharing lunches. This internship was funded by the Matryoshka grant [1], a project led by Jasmin C. Blanchette and mainly based at the Vrije Universiteit Amsterdam. Matryoshka pursues the goal of delivering very high levels of automation to users of proof assistants.

Contents

Figure 1: The Matryoshka project - Fast Interactive Verification through Strong Higher-Order Automation

Introduction

Lean is a disruptive, open-source proof assistant and programming language currently developed at Microsoft Research and Carnegie Mellon University [2]. It implements the theory of dependent types, like the Coq proof assistant [3]. The syntax and spirit of Lean are close to Haskell.

Lean makes it possible to develop algorithms, meta-programs and computer-checked proofs in a same, unified programming language. It is a relatively new research project with few users as compared to other similar software: its development can hence easily include experimental features and modify the language.

The mathematical library of Lean, called Mathlib [4], contains basic formalization of many of the main theories of modern mathematics.

A live online Javascript version of Lean is available at `leanprover.github.io/live/latest/` for quick testing.

1 Lean in a nutshell

This section provides a brief overview of Lean's features. We assume that the reader has some familiarity with functional programming, proof assistants and mathematical logic. For further reference, Lean comes with a detailed tutorial [5] as well as a reference manual [6]. The online dedicated chat room[1] proves useful when looking for information about undocumented features.

1.1 Type system

Lean features like Coq an infinite hierarchy of type universes `Sort`:

```
constants (a : Sort 2) (b : Sort 3)
#check a → b -- Sort 3
```

The expression `Type n`, where `n` is a natural number, is syntactic sugar for `Sort (n+1)`. `Type` and `Prop` are other names for `Type 0` and `Sort 0`, respectively. Lean implements the *propositions-as-types* paradigm, also called the *Curry–Howard correspondence*: proving a proposition `p : Prop` boils down to providing an element of "type" `p`.

Lean has dependent types built-in. Dependent types allow for very expressive types and thus all the sophisticated propositions of higher-order logic. The Π operator is used to express the type of a functions *whose return type depends on the input value*, that is, the expression `Π (a : α), f a` is the type of a function which takes an argument `a` of type α and outputs an element of type `f a`.

Π types generalize polymorphism, a feature available in many programming languages, when α is `Type n`. For instance, the type of the `list.nil` (empty list) constructor of the polymorphic `list` inductive datatype is `Π {T : Type u}, list T`.

When a variable that is Π-bound is not used in the body of the Π type (*id est* the type function is constant), then Lean's pretty-printer shortens the Π type into an arrow type, which are more common in other functional programming languages: `F : Π (a : α), β` will rather be formatted as `F : α → β`.

1.2 Inductive datatypes, structures and type classes

Defining a datatype in Lean amounts to giving a type to each of its constructors:

```
-- Natural numbers
inductive nat
| zero : nat
| succ : nat → nat -- or: '| succ (n : nat) : nat'
```

Structures, or records, provide a useful way to store ordered and named data. They are implemented as inductive types with a single constructor (usually named `mk`) comprising an argument for each of the structure's fields. A structure comes with associated projections, i.e. getter functions for each field.

Lean provides syntactic convenience to define structures. Here is an example, taken from Lean's mathematical library, of a structure representing a mathematical filter, that is, a nonempty set of sets which is *upward closed* and *downward directed*[2]:

```
structure filter (α : Type u) :=
(sets            : set (set α))
(exists_mem_sets : ∃x, x ∈ sets)
(upwards_sets    : ∀{x y}, x ∈ sets → x ⊆ y → y ∈ sets)
(directed_sets   : directed_on (⊆) sets)
```

[1] Available at `leanprover.zulipchat.com`.

[2] A filter $\mathcal{F} \subseteq \mathcal{P}(E)$ on a set E satisfies

$$\mathcal{F} \neq \emptyset\ , \quad \forall x \in \mathcal{F}, \forall y \in \mathcal{P}(E), x \subseteq y \Longrightarrow y \in \mathcal{F} \quad \text{and} \quad \forall (x, y) \in \mathcal{F}^2, \exists z \in \mathcal{F}, z \subseteq x \wedge z \subseteq y\ .$$

Any datatype with a single constructor can be both constructed and destructed *via* pattern matching using the handy *anonymous constructor* notation:

```
example {p q : Prop} (hp : p) (hq : q) : p ∧ q := ⟨ hp, hq ⟩
```

In addition, Lean features type classes, like Haskell. Type class inference allow for automatic retrieval of data, *exempli gratia* type properties and proofs, without having to explicitly specify the fetched data. Type classes are used in the Lean library to define a whole hierarchy of algebraic properties on types, which can be specified as arguments to a proof using square brackets:

```
/- the type α is equipped with an associative multiplication operation
   and any element at the left of both sides of an equality can be canceled -/
example {α} [has_mul α] [is_associative α (*)] [is_left_cancel α (*)] {a b c : α} :
    a * b = a * c → b = c :=
is_left_cancel.left_cancel _ _ _
```

1.3 Logic and formalization

Mixed with dependent types, datatypes are used to implement inside Lean itself almost any element of logic. These definitions are very similar to that of Coq's core [7]. Here are a few examples:

- the `false` proposition is defined as an inductive type with no constructor:

  ```
  inductive false : Prop
  ```

  This implies that the system is *sound*, i.e. no proof of `false` can be produced, except in case of a bug.

- the existential quantifier is implemented as a dependent pair:

  ```
  inductive Exists {α : Sort u} (p : α → Prop) : Prop
  | intro (w : α) (h : p w) : Exists
  ```

  It is then bound to the more convenient ∃ unicode notation.

- disjonction (i.e. the "or" connective) of two propositions `A` and `B` is implemented as a datatype with two constructors - a proof of `A ∨ B` can be constructed using a proof of `A` and the `or.inl` ("left intro") constructor, or with a proof of `B` with the `or.inr` ("right intro") constructor:

  ```
  inductive or (a b : Prop) : Prop
  | inl {} (h : a) : or
  | inr {} (h : b) : or
  ```

- equality is defined as an inductive family:

  ```
  inductive eq {α : Sort u} (a : α) : α → Prop
  | refl : eq a
  ```

  Only `eq a a` is thus inhabited, or provable. Lean will type-check `eq.refl c` as a proof for `a = b` only when `a`, `b` and `c` are *definitionally equal*, that is, when they reduce to a common term:

  ```
  example {α} {a : α} {f : α → α} [inhabited α] :
    (λx, f x) (list.repeat a 3).head = (prod.fst ∘ id ∘ (λx, prod.mk (f x) x)) a :=
  eq.refl (f a)
  ```

Lean's logic is intuitionistic, e.g. the excluded-middle property `∀ (p : Prop), p ∨ ¬p` cannot be derived from its internal definitions. The `classical` namespace/library provides such a proof, assuming the axiom of choice:

```
class inductive nonempty (α : Sort u) : Prop
| intro (val : α) : nonempty

axiom choice {α : Sort u} : nonempty α → α
```

1.4 Automation and programming in Lean

Formalizing proofs using a proof assistant is often long and difficult, as compared to the mathematicians' pen-and-paper proofs. Proof automation is a very active research field and good progress has been achieved in recent years. It can refer to fully-automated reasoning systems like Vampire[3], but also to integrated proof assistants like Lean. One goal of Lean is to make easy the use and development of proof automation procedures in an interactive context, called *tactics*, taking the pain out of interactive theorem proving.

The *interactive tactic mode* of Lean is entered within the `begin` and `end` keywords and enables users to use one of the many built-in interactive tactics. The `by` keyword is mostly used when a single tactic solves the goal. We show here, as an example, a proof of the so-called *drinker's theorem* written in two different ways. With no automation, the proof in pure term style looks like this:

```
import logic.basic -- Mathlib contains some extra logic lemmas.
open classical -- We need classical logic for the case distinction and decidability.
local attribute [instance] prop_decidable -- All propositions are decidable.

variables {α : Type} {p : α → Prop} [inhabited α]

lemma drinker : ∃y, (p y → ∀x, p x) :=
  have h : ∀y, (p y → ∀ x, p x) ↔ (¬p y ∨ ∀ x, p x) := λy, imp_iff_not_or,
  (exists_congr h).mpr $
    exists_or_distrib.mpr $
      by_cases
        (λg : ∀x, p x, or.inr $ (exists_const _).mpr g)
        (λng, or.inl $ not_forall.mp ng)
```

However, this proof can be reduced to a single line when using tactics:

```
lemma drinker : ∃y, (p y → ∀x, p x) := by simp [not_forall_not.mp, forall_and_distrib]
```

As mentioned above, Lean already contains a good automation ecosystem. Two of the most fundamental tactics are the simplification tactic `simp` and the rewriting tactic `rw`. Both are implemented in C++ for the sake of efficiency, but can be heavily tweaked from inside Lean, and used as tools to build other tactics. `simp` belongs to the kind of tactics which make use of an extendable set of lemmas to operate on a goal or hypothesis (using the `at` keyword in interactive mode). Users and library developers can add their own simplification lemmas by tagging them with the `[simp]` attribute.

Lean is, in addition to being a proof assistant, a full-fledged programming language, as one can define all kinds of algorithms and data structures and efficiently execute those inside Lean's virtual machine using the `#eval` command:

```
-- Sort a list using Mathlib's proved sorting procedures.
import data.list.sort
#eval [5, 2, 6, 4, 1, 9, 7, 3, 8, 0, 7].merge_sort (≤)
-- Outputs [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9].
```

[3] Vampire (vprover.org) won 28 titles in the CASC automated theorem proving competition since 1999.

Lean's versatility as both a proof and programming language is a key feature when it comes to programming tactics, or interfacing Lean with external tools. The act of programming one's own Lean tactics to handle specific proof situations is referred to as *meta-programming*, because tactics belong to the `meta` fragment of Lean. That means that one can define non-terminating procedures and make reference to some untrusted, fast and native definitions from the C++ code, e.g. `name`, which is Lean identifiers' type, or `expr`, an inductive type representing Lean's expressions abstract syntax tree. This meta-programming framework is thoroughly described by Ebner et al. [8].

Proofs and definitions which are not tagged with `meta` cannot make use of any definition from the `meta` fragment. However, when inside a `meta` declaration (e.g. when writing a tactic), developers can use the whole non-`meta` Lean formalization library, which means that natural numbers or lists are the same as the ones used in proofs. In turn, some of those regular datatypes which are used inside a `meta` declaration can be *reflected* into an `expr`. A use case is shown below.

Lean tactics are implemented as state and alternative *monads*, where the type of the state is the `tactic_state`, which comprises the goal and current local hypotheses. Systematic manipulations of the proof state are hence encapsulated inside a practical monadic box allowing for failure. The `<|>` operator implements backtracking, and the monad framework also brings the handy `do` notation to represent a series of operations. We heavily used the combinators `<$>`, the mapping operation of the `functor` class, and `<*>`, the sequence operation of the `applicative` class. Many canonical other monad instances are already available in the standard library, like the `option` and `list` monads, as well as the monadic `parser` that we used for our Nunchaku integration.

The `io` monad is used for any action involving interaction with *the outside world* and having side effects outside of the tactic state. In this example, we create a tactic that synthesizes a list of random natural numbers using `unsafe_run_io : Π {α : Type}, io α → tactic α`, then pass it to the previous sorting function:

```
import data.list.sort system.io
open tactic
meta def get_random_nat_list (min max length : ℕ) : tactic unit := do
  let l' := list.repeat 0 length,
  l ← l'.mmap $ λ_, unsafe_run_io $ io.rand min max,
  exact `(l) -- `(l) is the reflected version of l.

#eval let l := by get_random_nat_list 0 10000 1000 in l.merge_sort (≤)
-- [4, 7, 13, 36, 37, 39, 42, 45, 46, 70, 71, 74, 81, 90, 93, 94, 102, 113, 136, ...
```

1.5 Manipulating expressions

Lean expressions (typically : a proof goal or a hypothesis type) are represented inside the meta-programming framework with the `expr` datatype (reproduced in figure 2), which reflects the internal representation of the C++ implementation.

Understanding how Lean interprets those different constructors is key to programming tactics. In particular, the `var` constructor correspond to a De Bruijn index, which means that it should only be used after a binding constructor among `lam`, `pi` and `elet`. When traversing an expression and to keep hold of the binding information, one needs to *instantiate* every new binded `var` into a `local_const`. These store a unique name (usually obtained with `tactic.mk_fresh_name`), a pretty-printing name which isn't necessarily unique, as well as the binding type (e.g. if the parameter is implicit or retrieved through type class inference) and the type of the local constant.

```
meta inductive expr (elaborated : bool := tt)
| var      {} : nat → expr
| sort     {} : level → expr
| const    {} : name → list level → expr
| mvar        : name → name → expr → expr
| local_const : name → name → binder_info → expr → expr
| app         : expr → expr → expr
| lam         : name → binder_info → expr → expr → expr
| pi          : name → binder_info → expr → expr → expr
| elet        : name → expr → expr → expr → expr
| macro       : macro_def → list expr → expr
```

Figure 2: Lean's expressions constructors

If `lc` is a fresh local constant and `e` an expression, then `expr.instantiate_var e lc` will traverse `e` and perform changes on every `var n`:

- if `n` corresponds to a legal De Bruijn index in `e`, then the `var` is left unchanged,
- if `n` equates to the number of binders previously seen during the traversal[4], then `var n` is changed to `lc`,
- otherwise, `var n` is decremented and becomes `var (n-1)`.

The `const` constructor refers to a name from the current global environment. The environment is accessed in tactic mode with `get_env` and declarations can then be retreived with `environment.get`. Lean provides a handful of tools to work with such declarations, for instance to know whether a given declaration refers to a function or datatype, and in the latter case to obtain the name of its constructors.

Many utility functions operating on expressions are available in the default Lean library, both inside and outside the `tactic` monad. We used them extensively; reading Lean's core library code is the best way to know their use cases.

An original example of another handy use of monads for working with expressions in Lean is the converter `conv`, also available as a monad transformer. A converter can be fed a transitive binary relation `R : α → α → Prop` on a type α and an element `a : α`. It then attempts at producing an element `b : α` and a proof of `R a b`. The `bind`, `map` and `pure` operations are defined so that one can combine converters together. For instance, if c_1 and c_2 are converters, c_1 `>>` c_2 will conjugate the effects of c_1 and c_2 so that an element is converted with both sequentially, the resulting proof being obtained by transitivity of the binary relation.

[4] Lean's De Bruijn indices are zero-based.

2 An algebraic equation simplifier

2.1 Motivation and goal

Despite being a powerful tool, Lean's simplifier does not systematically perform cancellations in algebraic structures that allow it, such as groups or rings, as illustrated by the following piece of code:

```
example {α} [group α] (a b c : α) (h : a * b * c = a * c) : b = 1 :=
begin
  -- simp at h, exact h -- doesn't work
  -- Instead, we have to use the rewriter with special settings:
  rw [ ←mul_one a ] at h {occs := occurrences.pos [2]},
  exact (mul_left_cancel (mul_right_cancel h))
end
```

Here, we would like `h` to be simplified into `b = 1` so that we can use it to prove the goal. Here is the same proof without tactics:

```
example {α} [group α] (a b c : α) (h : a * b * c = a * c) : b = 1 :=
mul_left_cancel $ mul_right_cancel $ eq.trans h $ congr_arg (*c) $ eq.symm $ mul_one a
```

One can notice how cumbersome such a simplification proof can be, especially when facing equations with many variables. Our goal is to define a procedure which can perform better than the integrated simplifier or rewriter in those cases.

Here is how such a process could be systematized:

1. detect an equality hypothesis `h : lhs = rhs` where the two expressions `lhs` and `rhs` are built upon a common associative commutative[5] binary operation,
2. possibly retrieve a common element `a` in the tree structures describing the operation's applications of `lhs` and `rhs`, failing if there is none,
3. provide an auxiliary `lhs'` with a proof that `a + lhs' = lhs` and an auxiliary `rhs'` such that `a + rhs' = rhs`, `a` being "pulled left" of the tree structure using associativity and commutativity (or "pulled right" depending on the available cancelation property),
4. associate the proofs using `eq.trans` (transitivity of equality) to obtain a proof of the assertion: `lhs = rhs → a + lhs' = a + rhs'`,
5. use `is_left_cancel.left_cancel` with adequate parameters to obtain an expression with type: `a + lhs' = a + rhs' → lhs' = rhs'`,
6. combine, and replace the hypothesis `h : lhs = rhs` with `h' : lhs' = rhs'`,
7. repeat the procedure until it fails, so that it is guaranteed that no term common to both sides is left.

2.2 AC term rewriting in Lean

Our implementation could rely on already existing Lean tools. Lean indeed features a powerful native congruence closure algorithm [9] that can handle AC (associativity-commutativity) theories. It also features a pair of useful AC *metaconstant* tactics implemented in C++:

```
meta constant flat_assoc : expr → expr → expr → tactic (expr × expr)
meta constant perm_ac : expr → expr → expr → expr → expr → tactic expr
```

[5] Such a procedure could also be extended to a non-commutative context, but the description is of little interest, with many corner cases and boilerplate manipulations.

Given the following expressions[6]:

```
α : Type
op : α → α → α
assoc : ∀ (a b c : α), a * b * c = a * (b * c)
comm : ∀ (a b : α), a * b = b * a
e₁ : α
e₂ : α
```

`flat_assoc op assoc e₁` produces a pair of expressions ⟨`e₁'`, `pr`⟩ where `e₁'` is `e₁` rewritten with a flattened parenthesis structure[7] and `pr` is a proof that `e₁ = e₁'`. On the other hand, `perm_ac op assoc comm e₁ e₂` produces a proof of `e₁ = e₂` if the two expressions are equal modulo AC.

Here is a simple example where `perm_ac` is used to generate an equality proof, in a very specific setting:

```
example {α : Type} [comm_group α] {a b c : α} :
  a * b * c = c * b * a := by do
α ← get_local `α,
mul ← mk_mapp `has_mul.mul [α, none],
assoc ← mk_mapp `is_associative.assoc [α, mul, none],
comm ← mk_mapp `is_commutative.comm [α, mul, none],
(lhs, rhs) ← target >>= match_eq,
perm_ac mul assoc comm lhs rhs >>= exact
```

The `mk_mapp` tactic handles function application with implicit parameters. We could use this `perm_ac` tactic at step n°3 of our procedure scheme.

2.3 Implementation

In our didactic implementation however, we use a custom auxiliary recursive function producing individual proofs for the "pull left" conversion. Both sides of the equality hypothesis are cast into a binary operation structure:

```
meta inductive binop_tree
| node (left_child : binop_tree) (right_child : binop_tree) : binop_tree
| leaf (value : expr) : binop_tree

/- Parses only the given binary operation `op`.
   This is in case multiple operations are mixed (+, *, ...). -/
meta def get_tree_op (op : expr) : expr → tactic binop_tree
| e@(app (app op_ a) b) := -- Pattern match against the `expr.app` constructor.
  (unify op_ op >> binop_tree.node <$> get_tree_op a <*> get_tree_op b)
  <|> (pure $ binop_tree.leaf e) -- If unification fails, we just create a leaf.
| e := pure $ binop_tree.leaf e
```

Once we have a binary tree representing both the left and right-hand side of our equality hypothesis, we can easily deduce a term `elim_term` to be canceled from both sides. The following function provides a list of booleans for easy and efficient recursion:

```
/- Produces a list of booleans representing the path from the root of the tree
   to the first occurence (from the left) of e among the leaves. -/
meta def get_position (e : expr) : binop_tree → tactic (list bool)
| (leaf v) := unify v e >> pure []
| (node l r) := (list.cons ff <$> get_position l) <|> (list.cons tt <$> get_position r)
```

[6] The types correspond to the output of the `infer_type` tactic. These variables have type `expr`.
[7] e.g. $a * (b * (c * d) * e)$ becomes $a * b * c * d * e$, i.e. $(((a * b) * c) * d) * e$.

The obtained boolean list is then passed to a function producing a converter that "pulls left" the given `elim_term`:

```
/- Converts e.g. (a + (e + b)) to (e + (a + b)).
   Only works with the equality relation. -/
meta def pull_core (elim_term : expr) : list bool → conv unit
| [] := skip
| [ff] := skip
| [tt] := apply_const ``is_commutative.comm
| (ff::l) :=
/- Case where the expression has the shape A+B
   with elim_term present in A.
   1st step: convert to (elim_term + A') + B
   2nd step: convert to elim_term + (A' + B) -/
  congr_core (congr_core skip (pull_core l)) skip
  >> apply_const ``is_associative.assoc
| (tt::l) :=
/- Here, elim_term is in B.
   1st step: convert to A + (elim_term + B')
   2nd step: apply custom lemma to get elim_term + (A + B') -/
  congr_core skip (pull_core l)
  >> apply_const ``op_left_comm
```

In this code:

- `skip : conv unit` is the identity monadic converter,
- `congr_core : Π {α β : Type}, conv_t m α → conv_t m β → conv_t m (α × β)`[8] is a converter combinator which converts both a function and its argument with the given pair of converters,
- `apply_const : name → conv_t m unit` converts using the given lemma or theorem name.

We also developped special modules to handle units (e.g. 1 and 0) when necessary. Here are a few examples of the final converter `cancel_conv` in action:

```
example {a b c : ℤ} : b + (a + (c + a)) = c + a + b ↔ a = 0 :=
by conversion cancel_conv

example {α} {a b c d : α} [add_semigroup α] [has_mul α]
[is_right_cancel α (+)] [is_left_cancel α (+)] :
  a + b*c + c = a + d + c ↔ b*c = d :=
by conversion cancel_conv
```

The `conversion` tactic matches a goal `lhs R rhs` where R is a binary relation and runs the given converter on `R` and `lhs` to try to exact a corresponding proof. We tried several approaches and the `conv` framework appears very convenient and powerful. We can also easily derive an interactive tactic simplifying and replacing an hypothesis in the current proof state, using our cancelation converter as basis.

Lean 4 will probably feature an improved simplification tactic with the ability to better handle such cases.

[8] The `m` argument is the base monad for the `conv_t` transformer, usually `tactic`.

3 An extendable monotonicity prover

3.1 Idea

Proving mathematics with Lean sometimes involves solving goals such as this very simple example:

```
p q r : Prop,
h : p → q,
hp : p,
hr : r
⊢ q ∧ r
```

This proof is straighforward: the terms `and.intro (h hp) hr` or `⟨ h hp, hr ⟩` satisfy the goal. However, we can notice that the conjonction operator $\wedge$ is monotonous (in both arguments) with respect to the partial order relation $\to$ on propositions:

$$(p \to q) \to p \wedge r \to q \wedge r\,.$$

Recall that the $\to$ operator associates to the right. Calling `g` a proof of the above implication, we have `(g h) : p ∧ r → q ∧ r`. We can thus change the goal to `p ∧ r` and then `exact ⟨hp, hr⟩`.

We can find the same pattern in a variety of common cases, some of which are described below:

- The same applies to the or ($\vee$) operator: `(p → q) → p ∨ r → q ∨ r`. But we want to use full generality and state that for any predicates `p, q : α → Prop`, we have

  $$\left.\begin{array}{l}\forall\ (a\ b : \alpha),\ r\ a\ b \to p\ a \to p\ b\\ \forall\ (a\ b : \alpha),\ r\ a\ b \to q\ a \to q\ b\end{array}\right\} \to \forall\ (a\ b : \alpha),\ r\ a\ b \to p\ a \vee q\ a \to p\ b \vee q\ b\,.$$

  Note that the premises are also monotonicity properties.

- Let $\preccurlyeq : \alpha \to \alpha \to \mathtt{Prop}$ be a transitive binary relation on a type α. Then for every `c : α` we have:

  $$a \preccurlyeq b \to c \preccurlyeq a \to c \preccurlyeq b$$

- Let $\preccurlyeq : \alpha \to \alpha \to \mathtt{Prop}$ be any binary relation on a type α. Then for every `p : Prop` we have the trivial assertion:

  $$a \preccurlyeq b \to p \to p$$

- When `A, B : set α` and `p : α → Prop`,

  $$A \supseteq B \to \forall\ x \in A,\ p\ x \to \forall\ x \in B,\ p\ x\,.$$

Based on these assertions, plus some others, our aim is to develop an automation procedure that, given `a b : α`, an assumption `h : a ≼ b` and a goal `f a` where `f` is monotonous with respect to $\preccurlyeq$ and $\to$, provides a proof to the assertion

$$a \preccurlyeq b \to f\ b \to f\ a$$

thus allowing to change the current goal to `f b`, as explained above. This is of course not possible for all goals.

3.2 Implementation

We define in Lean a structure `monot` that encapsulates this kind of monotonicity properties, along with a `[monot]` user attribute that serves to collect user-defined `monot` properties:

```
structure monot {α : Type*} {β : Type*}
  (r : α → α → Prop) (p : β → β → Prop) (f : α → β) : Prop :=
(monot : ∀a b, r a b → p (f a) (f b))

@[user_attribute]
meta def monot_attribute : user_attribute := { name := `monot, descr := "Monot rules" }
```

Users can then tag their own lemmas with `@[monot]`, for instance[9]:

```
@[monot]
lemma monot.finset_card {α} :
  monot ((⊆) : finset α → finset α → Prop) (≤) finset.card :=
⟨ λ_ _, finset.card_le_of_subset ⟩
```

We then define a tactic `monot_subst (hrel : expr) (hyps : list expr) : tactic unit` comprising several simple steps:

1. pattern-match the inferred type of the `hrel` argument to retrieve an expression with shape `r x y` where `r` is a reflexive transitive binary relation on a type α,
2. extract the predicate `p : α → Prop` such that the goal unifies with `p x` by abstracting any occurrence of the sub-expression `x` in the goal by a λ-bound `var`,
3. ask the auxiliary *monot-prover* to find a proof `g` for `monot (flip r) implies p`, possibly making use of the extra proofs provided through `hyps`, then apply the proof `g hrel` to the current goal, which changes the latter to `p y`.

The auxiliary prover is a recursive function partly relying on the list of `@[monot]` lemmas provided by the user, which makes it extendable. It also uses type class inference. It is based on the `apply` tactic, which takes an `e : expr` and unifies the (inferred) type of `e`, or one of its conclusions, with the goal, providing the corresponding proof. The possible subsequent missing premises then create each a new goal. In our prover, recursion is achieved through the `all_goals` tactic which attempts to prove all the remaining goals with a single given tactic. The main structure of the prover thus looks like this:

```
/- tries to solve a goal of the shape ‘monot r₁ r₂ f‘
   using ns (names of lemmas tagged with @[monot])
   and hs (local hypotheses provided in the interactive context) -/
meta def monot_aux (ns : list name) (hs : list expr) : tactic unit := do
  (first (hs.map apply) >> skip) <|> first (ns.map applyc),
  all_goals monot_aux
```

In addition to this, we add a strictly decreasing natural number parameter, to prevent unpredicted infinite looping recursion. Moreover, great care must be taken regarding which lemmas are tagged with `@[monot]`, and in which order. That's why we actually make distinct cases for lemmas regarding e.g. the monotonicity of composition of two functions, which requires special treatment because it can always be instantiated with identity and create loops:

```
lemma monot.comp {α β γ} {f : α → β} {g : β → γ}
  {r : α → α → Prop} {r' : β → β → Prop} {r'' : γ → γ → Prop}
  (hg : monot r' r'' g) (hf : monot r r' f) :
  monot r r'' (g ∘ f) :=
⟨ λ a b hab, hg.monot (f a) (f b) $ hf.monot a b hab ⟩
```

Here are a two cases in which our tactic becomes useful. First, to find proofs of linear inequalities:

```
example {x : ℤ} (hx : x ≤ -2) : 3+x+x≤0 :=
begin
  monot_subst hx,
  -- The new goal is now (3 + -2 + -2 ≤ 0), i.e. (-1 ≤ 0).
  simp -- or trivial
end
```

[9] A `finset` is constructed as a `multiset` with no duplicates. `multiset` α is the quotient of `list` α by the *equality modulo permutation* equivalence relation.

`monot_subst` can also be used with sets, or set-like objects:

```
example {α : Type*} {p : α → Prop} {A B : finset α} {n : ℕ} (h : A ⊆ B) :
  finset.card A ≤ n ∧ ∀x∈A, p x :=
begin
  monot_subst h, -- The goal is now: ‘finset.card B ≤ n ∧ ∀x ∈ B, p x‘.
  admit
end
```

4 Integration of an external counter-example generator

4.1 Introduction and example

Nunchaku is a model finder, or counter-example generator, developed by Simon Cruanes and Jasmin C. Blanchette with the aim of replacing Nitpick, the latter being specific to Isabelle. Conversely, Nunchaku is designed to be integrated in any proof assistant, and corresponding bindings have already been developed for Isabelle and Coq. No Lean integration was developed so far.

Nunchaku works as a frontend to several different solvers. It is currently compatible with CVC4, Paradox, Kodkod or SMBC (see [10]). It understands higher-order logic and polymorphism, but not dependent types. The integration will thus be first limited to a certain set of problems that don't include e.g. dependent records like the `group` type class. It ultimately takes the form of an interactive Lean tactic that translates in several steps the focused goal into a complete Nunchaku problem, runs Nunchaku on this input, parses the output and gives a hopefully meaningful answer to the user.

Nunchaku accepts several different input formats. We choose the native, best-supported `.nun` format for problems. To give an idea of how Nunchaku is to be used, we provide here a sample problem. Suppose we want to prove in Lean the putative assertion `∀n : ℕ, n + n = 3*n`. Here is what our tactic would ask Nunchaku to solve for the negated goal `¬(∀n : ℕ, n + n = 3*n)`:

```
data nat :=
    | nat__zero
    | nat__succ nat.

rec nat__add : (nat -> (nat -> nat)) :=
    (forall a. (forall b. ((nat__add a (nat__succ b)) = (nat__succ (nat__add a b))))));
    (forall a. ((nat__add a nat__zero) = a)).

rec nat__mul : (nat -> (nat -> nat)) :=
    (forall a. (forall b. ((nat__mul a (nat__succ b)) = (nat__add (nat__mul a b) a))));
    (forall a. ((nat__mul a nat__zero) = nat__zero)).

data has_add alpha :=
    | has_add__mk (alpha -> (alpha -> alpha)).

rec has_add__add : (pi alpha. ((has_add alpha) -> (alpha -> (alpha -> alpha)))) :=
    (forall x_0. ((has_add__add (has_add__mk x_0)) = x_0)).

rec bit0 : (pi alpha. ((has_add alpha) -> (alpha -> alpha))) :=
    (forall s. (forall a. ((bit0 s a) = (has_add__add s a a)))).

data has_one alpha :=
    | has_one__mk alpha.

rec has_one__one : (pi alpha. ((has_one alpha) -> alpha)) :=
    (forall x_0. ((has_one__one (has_one__mk x_0)) = x_0)).

rec bit1 : (pi alpha. ((has_one alpha) -> ((has_add alpha) -> (alpha -> alpha)))) :=
    (forall s_1. (forall s_2. (forall a. ((bit1 s_1 s_2 a) =
        (has_add__add s_2 (bit0 s_2 a) (has_one__one s_1)))))).

goal
    ( ~(forall (n : nat). ((nat__add n n) = (nat__mul (bit1 (has_one__mk (nat__succ nat__zero))
        (has_add__mk nat__add) (nat__succ nat__zero)) n)))).
```

Notice that numerous structures and projections are present. This problem could be made simpler by β-reducing structure projections before translating. However, we would then lose some information that is useful to Lean's pretty-printer, and the formatted output would be less user-friendly.

In this particular case, Nunchaku finds a model, or counter-example to the universal quantification. We ask the output to be formatted using s-expressions:

```
(SAT
 ((val
   (_witness_of
    (forall ((n nat))
     (= (nat__add n n)
      (nat__mul
       (bit1 nat (has_one__mk nat (nat__succ nat__zero))
        (has_add__mk nat nat__add) (nat__succ nat__zero)) n))))
   (nat__succ nat__zero))))
```

Our tactic then parses this output, and pretty-prints the following indication to the user:

```
SAT:
Witness of
  ∀n : ℕ, n + n = 3*n
is
  1
```

An auxiliary tactic can alternatively retrieve the parsed output, and inform the user that the current goal cannot be proved.

4.2 Lean to Nunchaku problem translation

The translation pipeline starts in tactic mode with the goal as an `expr` and the `environment`, which contains thousands of declarations. Only a portion of this data is required to express a minimal Nunchaku problem for the goal at stake.

The first translation step thus takes a Lean expression and outputs an ordered list of classified needed declarations:

```
meta inductive nun_needed_st : Type
| Inductive : name → nun_needed_st
| Structure : name → nun_needed_st
| Definition : name → nun_needed_st

meta def get_sorted_needed_st : tactic (list nun_needed_st) := sorry
```

This tactic takes the goal, and extracts all references to declarations from the environment (the `expr.const` constructor) it contains. Then, it classifies these declarations with `nun_needed_st` and recursively walks through statements, using the lemmas (or types of constructors) defining them in Lean to get new declarations to explore. Lean's equation compiler indeed automatically provides equational lemmas for recursive definitions. Here are the two lemmas generated for the addition of natural numbers:

```
nat.add.equations._eqn_1 : ∀ (a : ℕ), nat.add a 0 = a
nat.add.equations._eqn_2 : ∀ (a b : ℕ), nat.add a (nat.succ b) = nat.succ (nat.add a b)
```

Here is the (ordered) list of needed statements for the goal `∃n : ℕ, n^2 = 16`:

```
[Inductive nat, Definition nat.add, Definition nat.mul, Definition nat.pow,
    Structure has_add, Definition bit0]
```

The statements are retrieved in a correct topological order, that is, their list gives a correct evaluation order with respect to the dependency relation. Names referring to a structure projection are classified under the structure's name, and names referring to a datatype constructor under the name of the datatype. Statements that are part of Nunchaku's builtins, e.g. equality or logical connectives, are removed from the list.

```
namespace nun
meta inductive ty : Type -- Nunchaku types.
| Prop_
| Type_
| Arrow_ : ty → ty → ty
| Pi_ : name → ty → ty -- We use Lean's `name` type.
| Var_ : name → bool → ty -- The bool argument is whether the var is local.
| App_ : name → list ty → ty
```

Figure 3: Nunchaku types AST

We then define datatypes to contain Nunchaku's AST (abstract syntax tree) (partly reproduced in figure 3). We follow the specifications given in the documentation [10]. Contrary to Lean, Nunchaku makes a distinction between terms and types, which is reflected in our AST. A Nunchaku problem is a list of statements; those include datatype definitions, (co)inductive predicates or recursive functions, which correspond respectively to the `data`, `rec` and `pred` Nunchaku keywords.

The `nun_needed_st` list is then passed to a function translating each individual name into a list of Nunchaku statements. For instance, the translation of a structure will produce one statement for the datatype, and some others for the projections. Two separate recursive tactics are dedicated to the translation of a Lean `expr` into a Nunchaku term or type. The `Sort` hierarchy of Lean is flattened. An `expr.pi` is translated into either a Nunchaku `Arrow_` or polymorphic `Pi_` and the procedure fails in case a non-prenex type polymorphism or a proper dependent type is detected.

So far, Lean names are kept intact. However, Nunchaku doesn't allow the same set of characters in identifiers as Lean. That's why another step traverses the AST, turning each Lean name into a unique Nunchaku name. We use for this the `state_t` monad transformer from Lean's core library to maintain the names mapping. The mapping itself uses Lean's red-black map `rb_map` native implementation, and is kept for the inverse translation of names.

Finally, the resulting AST is formatted using Lean's `format` library, with respect to Nunchaku's input syntax. A unique file is created that is filled inside the `io` monad with the formatted input, and Nunchaku is run as an external process using `io.cmd`.

4.3 Inverse translation

We ask Nunchaku to format its output using s-expressions, which makes parsing easier. The parsed data is stored into a tree structure:

```
inductive stree (T : Type)
| node : stree → stree → stree
| leaf : T → stree
| nil {} : stree
```

Using the fixpoint combinator[10] `fix : Π {α : Type}, (parser α → parser α) → parser α` from Lean's `parser` library, we define a `sexpr_parser : Π {T : Type}, parser T → parser (stree T)` such that `sexpr_parser P` parses s-expressions whose base term is parsed by `P`.

The parsing of the output of section 4.1's example gives the following tree. It is formatted using `()` for the `stree.nil` constructor and `[l r]` for `stree.node l r`:

```
[SAT [[[val [[_witness_of [[forall [[[n [nat ()]] ()] [[= [[nat__add [n [n ()]]] [[nat__mul
    [[bit1 [nat [[has_one__mk [nat [[nat__succ [nat__zero ()]] ()]]] [[has_add__mk [nat [
    nat__add ()]]] [[nat__succ [nat__zero ()]] ()]]]]] [n ()]]] ()]]] ()]]] ()]] [[nat__succ
     [nat__zero ()]] ()]]] ()] ()]]
```

[10] The fixpoint recursive parser `fix F` satisfies `fix F = F (fix F)`.

After parsing, the `stree` structure is turned into a custom datatype reflecting Nunchaku's output. For the reconstruction of Lean's expressions, we reuse our unique names mapping.

4.4 Limits to the translation, related and future work

Our translation, taken "as is", is still a prototype. It is not bug-free and many corner cases are still to implement for it to be fully functional. Using our tactic with dependent types that are not propositions will result in a tactic failure. Also, we didn't implement the translation of quotient types, even though they belong to Nunchaku's specifications.

Cruanes and Blanchette [11] describe how any dependent type could be translated to Nunchaku, using the `asserting` Nunchaku command. Different translations are proposed for type classes like Lean's `monoid` or `group` and for regular dependent types like `fin (n : ℕ) : Type`, which is an implementation of $\mathbb{Z}/n\mathbb{Z}$. We would like to eventually implement these ideas; this would make the Lean-Nunchaku bridge more useful.

Nunchaku isn't the first external program that is integrated with Lean. Lewis [12] describes a similar process to create an interface between Lean and Wolfram Mathematica, a popular computer algebra system.

Future work and conclusion

This work can be extended in many ways. The algebraic simplifier could handle more cases including distributivity or factorization, which would require some sophisticated algorithms on tree structures. All these tactics, including the Nunchaku integration, would need to find real-life use cases while formalizing a big theorem.

Lean 4, successor to Lean 3, is being developed and will bring several changes to Lean's API. These changes will make it necessary to refactor the whole mathematical components library, as well as previously developed tactics.

The Lean Forward meeting, to be held in January at the VU Amsterdam, will gather for the first time Lean developers and users in an interactive workshop. It will hopefully give birth to new ideas that would make interactive theorem proving always more accessible.

References

[1] "The matryoshka project: Fast interactive verification through strong higher-order automation." [Online]. Available: matryoshka.gforge.inria.fr

[2] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *CADE*, 2015.

[3] "The coq proof assistant." [Online]. Available: coq.inria.fr

[4] "The lean mathematical components library." [Online]. Available: github.com/leanprover/mathlib

[5] J. Avigad, L. M. de Moura, and S. Kong, "Theorem proving in lean."

[6] J. Avigad, G. Ebner, and S. Ullrich, *The Lean Reference Manual*.

[7] "The coq proof assistant - library coq.init.logic." [Online]. Available: coq.inria.fr/library/Coq.Init.Logic.html

[8] G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura, "A metaprogramming framework for formal verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 34:1–34:29, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3110278

[9] D. Selsam and L. de Moura, "Congruence closure in intensional type theory," *CoRR*, vol. abs/1701.04391, 2017. [Online]. Available: http://arxiv.org/abs/1701.04391

[10] "Nunchaku documentation." [Online]. Available: https://nunchaku-inria.github.io/nunchaku/

[11] S. Cruanes and J. C. Blanchette, "Extending nunchaku to dependent type theory," *arXiv preprint arXiv:1606.05945*, 2016.

[12] R. Y. Lewis and M. Wu, "A bi-directional extensible ad hoc interface between lean and mathematica."