École normale supérieure
Département d'informatique

Master 1 internship report

Year 2018–2019

Arithmetic and Casting in Lean

Paul-Nicolas Madelaine

Supervised by Jasmin C. Blanchette and Robert Y. Lewis

At Vrije Universiteit, Amsterdam, Netherlands

Abstract

This is a report of work conducted under the supervision of Jasmin C. Blanchette and Robert Y. Lewis, in the Matryoshka team [1] at Vrije Universiteit. It presents our contribution to two projects of proof automation using the Lean theorem prover. The first one is `norm_cast`, a tool for normalizing casts inside expressions in Lean. The second one is Polya, a heuristic algorithm to prove non-linear inequalities.

Acknowledgments

I would like to thank Jasmin C. Blanchette for giving me the opportunity of doing this internship. I am very grateful to Robert Y. Lewis for his supervision and for helping me settle in Amsterdam. I would also like to thank everyone in the Matryoshka team and everyone I met in the VU during our lunch breaks. Finally I'm thankful to Timothy Bourke for the advice he gave me.

Contents

[1] http://matryoshka.gforge.inria.fr/

1 The Lean Theorem Prover

Lean is a theorem prover and programming language that implements the theory of dependent types [5]. It's similar to Coq, although it is more recent and still under heavy development by Leonardo de Moura and Sebastian Ullrich [2]. Lean is mainly written in C++, but the next version should move a lot of the code base to Lean itself.

Lean's math library, Mathlib [3], is a community driven project aimed at formalizing modern mathematics and offering interactive tools for mathematicians to use.

In this section we try to introduce both the type theory and the syntax of Lean. Lean's online tutorial is a great resource for beginners [1]. Lean offers a framework for meta-programming. It enables to run code in a virtual machine, this enable for mone efficient code. Lean's forthcoming revision, Lean 4, should even allow for compiling to LLVM. One can learn more about meta-Lean in [6].

1.1 Dependent types

Lean implements depent types theory. It has a built-in hierarchy of types. Levels in the hiearchy are called sorts and noted `Sort n` where `n` is a natural number. `Type n` is another notation for `Sort (n+1)`, and `Type 0` can be shortened to `Type`.

Lean also implements the Curry-Howard paradigm. This means propositions are expressed as types. `Sort 0` is the universe for propositions and is also named `Prop`. Building a proof of proposition `p : Prop` is the same as building a term of type `p`.

Dependent types are expressed with the Π operator: `Π (a : α), f a` is the type of functions which take an argument `a` of type α and return a value of type `f a`, where `f` is a family of types : `f : Type → Type`.

The arrow can be seen as a particular case of a Π type, where the body of the type doesn't depend on the argument: `Π (a : α), β` can also be written as $\alpha \to \beta$.

Arguments of a Π type can be implicit, and used for instance to define polymorphism. In that case they are written inside curly brackets. Take for instance the definition of the identity function:

```
def id : Π {α : Type}, α → α := λ {α : Type} (x : α), x
```

1.2 The equation compiler, the elaborator and type classes

The equation compiler is the peace of code responsible for compiling inductive declarations. On the other hand, the elaborator is responsible for filling holes inside Lean's expressions. These holes are called meta-variables, and are typically introduced by implicit arguments. We can give hints to the elaborator: by writing `(e : α)` we tell the elaborator that the expression to the left of the semi-colon should have the type α.

A type class is a particular case of implicit argument. It is noted with square brackets instead of curly brackets, and doesn't require a name:

```
def plus_one {α : Type} [has_add α] [has_one α] (x : α) : α := x + 1
```

[2] https://github.com/leanprover/lean4

[3] https://github.com/leanprover-community/mathlib

Type classes are also replaced by meta-variables, but are treated differently by the elaborator. For instance, in the above example, the elaborator will derive α from the type of `x`. But to derive values of types `has_add` α and `has_one` α the elaborator will trigger a search called type class inference.

Type classes are defined with the `class` operator, and instantiated with the `instance` operator. For instance the `has_add` type class, which stipulates that a particular type implements addition, is defined as follows:

```
class has_add (α : Type) : Type :=
(add : α → α → α)
```

Similarly, the syntax for instantiation is the following:

```
instance nat.has_add : has_add ℕ :=
{ add := λ x y, sorry }
```

`sorry` is a keyword that tells Lean to introduce a meta-variable ignore it. This is a way to write incomplete proofs and declarations.

More details about the elaborator can be found in [4].

1.3 Inductive types and structures

New types can be declared in Lean with the `inductive` constructor. For instance let's declare a family of types `Vector : Type → ℕ → Type` such that `Vector α n` is a type for lists of length `n` and whose elements are of type α. This is done by defining new constructors, namely `nil` and `cons`:

```
inductive Vector (α : Type) : ℕ → Type
| nil {} : Vector 0
| cons {} {n : ℕ} : α → Vector n → Vector (n+1)
```

The curly brackets just after the names of the constructors are a way of telling Lean that α should be an implicit argument to the constructor, even if it is an explicit argument of the family. Here is the complete type for the `cons` constructor:

```
cons : Π {α : Type} {n : ℕ}, α → Vector α n → Vector α (n+1)
```

For non-inductive types, we can use structures. For instance, let's define a type with two coordinates to represent points in the real plane:

```
structure point : Type :=
(x : ℝ)
(y : ℝ)
```

Lean's come with a syntax to give attributes to declarations. A good example of attributes usage is the type class system. Indeed, a class decration is actually a declaration of a structure that is given the `class` attribute:

```
@[class] structure has_add (α : Type) : Type :=
(add : α → α → α)
```

Similarly, an instance of a type class is a declaration of a value of that type, with the `instance` attribute:

```
@[instance] def nat.has_add : has_add ℕ :=
{ add := λ x y, sorry }
```

1.4 Tactics and the meta-programming framework

Writing proofs as terms becomes quickly unpractical. Like Coq, Lean enables the user to write proofs using tactics. Tactic-style proofs can be introduced by either the `by` keyword or within a `begin` ... `end` block. They allow for interactive and automatic prooving.

Users can write tactics using the meta-programming framework of Lean. The `meta` keyword enables to use datatypes that don't have a representation in the kernel. Thus, code marked as `meta` will always run in the VM, but has access to types that reflect the internals of Lean.

The type `tactic`, for instance, is a state monad encapsulating the state of the proof (i.e. the context and the goals). It allows to write tactics in a monadic do-notation, very similar to Haskell.

Another type usable in meta-Lean is the type `expr`, which reflects a Lean expression. It can be used to build proof-terms, solve goals, etc.

Finally, meta-Lean also allows for more convenient programming, by enabling infinite recursions, and use of more efficient datatypes.

2 Normalizing casts inside Lean expressions

In this section, we present a tool to handle casts inside Lean expressions.

2.1 The problem with casts

Technically, casts are just functions between two types. They are interesting because they are used by the compiler to coerce the type of an expression when it doesn't match the expected type. Take for example the following declaration:

```
def foo (n : ℕ) (m : ℤ) : ℤ := n + m
```

As we've seen in the declaration of the `has_add` type class, addition expects both arguments to be of the same type. Thus, at first glance, this declaration shouldn't type since `n` and `m` do not have the same type.

Still, Lean accepts this declaration. If we ask Lean to print the definition, it returns the following:

```
def foo : ℕ → ℤ → ℤ := λ (n : ℕ) (m : ℤ), ↑n + m
```

The up arrow symbol on the left side of the addition is a notation for the `coe` (as in coercion) function. It is defined in the `has_coe` type class:

```
class has_coe (a : Sort u) (b : Sort v) :=
(coe : a → b)
```

In the above example, an instance of `has_coe ℕ ℤ` needed to be define and was used by the elaborator to coere the left expression.

This can be a very useful feature, but it comes with some drawbacks. Because the elaborator is always filling an expression from left to right, it will often put coercions deeper in the expression than one would expect. Take for instance the following declaration:

```
def f (n m : ℕ) : ℤ := n + m
```

After elaboration, this leads to the following definition:

```
def f : ℕ → ℕ → ℤ := λ (n m : ℕ), ↑n + ↑m
```

But the user may have expected the coercion to happen "earlier" in the expression:

```
def f : ℕ → ℕ → ℤ := λ (n m : ℕ), ↑(n + m)
```

This can lead to some frustrating situations, where "obvious" results need a lot of boilerplate code to be proven:

```
example (n m : ℕ) : (m : ℤ) = n + 1 → m = n + 1 :=
begin
  intro h,
  rw ← int.coe_nat_one at h,
  rw ← int.coe_nat_add at h,
  apply int.coe_nat_inj,
  exact h
end
```

Several people in the Lean community have been asking for a tactic automating this kind of processes for quite some time. Some documentation was written to guide people when facing these kinds of problems, but no work had been put into writing a general algorithm.

We asked the community for examples of statements they would like to see solved by a tool like this, and we went from there to define a tactic that could solve these problems.

This can be done by defining a normal form of an expression in regard to casts. The tactic we want to define, called `norm_cast`, would change the current goal of the proof to its normalized form. In the above example, we could use it to close the goal more efficiently:

```
example {n m : ℕ} : (m : ℤ) = n + 1 → m = n + 1 :=
begin
  norm_cast,
  exact id
end
```

Here, `norm_cast` changes the goal from `↑m = ↑n + 1 → m = n + 1` to `m = n + 1 → m = n + 1`.

Often, casts functions are injective and compatible with many operators. Therefore we can move casts as high (or early) in the expression as possible using compatibility with the operators. We can then eliminate them by injectivity when they meet an equality or an equivalence in the expression. This is the general idea behind the `norm_cast` tactic.

From the beginning, the aim of `norm_cast` was to be a "convenience" tactic, therefore we sat the following guidelines:

- it should be fast and easy to use
- it should be easily extensible by the users as they define new casts
- it should be safe to use in the middle of a proof, i.e. extending the tactic should not break proofs that have already used it

Numerals appeared quite a lot in the examples we were given, since they require even more lemmas to handle, they were quite tricky to implement. Indeed, numerals are not values of a given types, but are implemented for any types that implements addition and contains one. The way we decided to deal with them is to "force" them into the type $\mathbb{N}$.

A lot of examples we were given involved numeric types, like $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, ..., but the same problems can be formulated for more abstract numbers, like cardinals. The tactic should even be able to apply on non-numeric expressions.

2.2 The solution: norm_cast

It is the need for extensivity that may have guided the design of `norm_cast` the most. Indeed, a very extensive and largely used tactic present in Lean is the simplifier, or `simp`. It goes inductively through an expression and, at every step, it searches in a big pool of lemmas the first one that fits to rewrite the current sub-expression. It continues until no lemma can be used to rewrite any sub-expression. This pool of lemmas is maintained by the users with the `simp` attribute.

The simplifier is one of Lean's core tactics. Although the interactive parts are written in meta-Lean, the core functions are links to C++ code. The primary entry point to the C++ simplifier is the following function:

```
meta constant ext_simplify_core
/- The user state type. -/
{α : Type}
/- Initial user data -/
(a : α)
(c : simp_config)
/- Congruence and simplification lemmas.
   Remark: the simplification lemmas at not applied automatically like in the
  simplify tactic.
   the caller must use them at pre/post. -/
(s : simp_lemmas)
/- Tactic for dischaging hypothesis in conditional rewriting rules.
   The argument 'α' is the current user state. -/
(discharger : α → tactic α)
/- (pre a s r p e) is invoked before visiting the children of subterm 'e',
   'r' is the simplification relation being used, 's' is the updated set of
  lemmas if 'contextual' is tt,
   'p' is the "parent" expression (if there is one).
   if it succeeds the result is (new_a, new_e, new_pr, flag) where
     - 'new_a' is the new value for the user data
     - 'new_e' is a new expression s.t. 'e r new_e'
     - 'new_pr' is a proof for 'e r new_e', If it is none, the proof is assumed
  to be by reflexivity
     - 'flag'  if tt 'new_e' children should be visited, and 'post' invoked. -/
```

```
(pre : α → simp_lemmas → name → option expr → expr → tactic (α × expr ×
  option expr × bool))
/- (post a s r p e) is invoked after visiting the children of subterm 'e',
   The output is similar to (pre a r s p e), but the 'flag' indicates whether
   the new expression should be revisited or not. -/
(post : α → simp_lemmas  → name → option expr → expr → tactic (α × expr ×
  option expr × bool))
/- simplification relation -/
(r : name) :
expr → tactic (α × expr × expr)
```

Here are the main arguments we are interested in:

- `s`, the set of lemmas to use
- `pre`, a function called to rewrite a sub-expression before visiting it
- `post`, a function called to rewrite a sub-expression after visiting it

This gives use the base structure for the `norm_cast` tactic: it uses the simplifier core functions with a set of appropriate lemmas to rewrite the expression into its normal form. These lemmas are defined using new attributes specific to `norm_cast`. That way, users can add support for new lemmas by giving them an attribute, just like for the simp tactic.

It can be noted that, thanks to this approach, `norm_cast` benefits from the fact that the simplifier is used intensively in Lean and lots of work has been put into making it efficient. Relying heavily on the simplifier, although convenient, doesn't automatically solve all our concerns about efficiency, extensibility and safeness. For instance, if we give the `move_cast` attribute to two lemmas that are both directions of the same equality, then the rewrite process will most likely loop forever. Similarly, the normal form is determined by how the users set the attributes.

For these reasons, we tried to define the attributes used by the tactic as clearly as possible.

- `move_cast`: this attribute is for lemmas that describe the behaviour of a cast function in regard to other functions and operators. `move_cast` lemmas are usually of the following shapes:

  ```
  Π ..., ↑(P a1 ... an) = P ↑a1 ... ↑an
  Π ..., ↑(P a1 ... an) ↔ P ↑a1 ... ↑an
  ```

  For example:

  ```
  @[move_cast] theorem coe_nat_add (m n : ℕ) : (↑(m + n) : ℤ) = ↑m + ↑n
  ```

- `elim_cast`: this attribute is for lemmas that state the injectivity of a cast function. `elim_cast` lemmas are usually of the following shapes:

  ```
  Π ..., P ↑a1 ... ↑an = P a1 ... an
  Π ..., P ↑a1 ... ↑an ↔ P a1 ... an
  ```

For example:

```
@[elim_cast] theorem coe_nat_inj' {m n : ℕ} : (↑m : ℤ) = ↑n ↔ m = n
```

- `squash_cast`: this attribute is for lemmas that state the compatibility between two casts. For instance that the composition of two cast functions is a cast function, or that the cast of one is still one. For example:

```
@[squash_cast] theorem cast_coe_nat (n : ℕ) : ((n : ℤ) : α) = n
```

The reason of the distinction between the `move_cast` and `elim_cast` attributes is that, by convention, lemmas that we want to use to move casts are written in Mathilb in the opposite direction that the one we want to use in `norm_cast`: they move casts from the top down. Therefore we define a separate attribute with a specif pre-processing step to revert the lemmas.

The core function of `norm_cast` is `derive`:

```
derive : expr → tactic (expr × expr)
```

It takes one Lean expression and returns two. The first output is the normalized input, and the second is a proof term that it is equal to the input.

The derive function proceeds in 4 steps, every one being a call to the simplifier with specific arguments:

- Step 1 is a pre-processing of numerals: it goes through the expression from the top down and replaces sub-expressions that are numerals. This new expression is the same numeral, but typed as a natural and then casted into the original type of the expression.
- Step 2 is where most of the rewriting happens: it goes through the expression from the bottom up, and uses `move_cast` and `elim_cast` lemmas to rewrite sub-expressions. If a sub-expression cannot be rewritten, it goes through the heuristic function. We will give more details about the heuristic later, but the main idea is to "break" casts into the composition of two.
- Step 3 is a call to the simplifier with the set of `squash_cast` lemmas. This is used as a clean-up after the heuristic functions, which can creates expressions of the form ...
- Step 4 is a post-process of numerals: in the same fashion as step 1, we replace casts of numerals with numerals.

As we mentionned, the algorithm will sometimes call a heuristic function. It is used to split casts functions into the compositions of two, to enable the tactic to move a cast function past an operator. For instance, given two variables `(n : ℕ)` and `(m : ℤ)`, let's normalize `↑n + ↑m = (2 : ℚ)`. We can't rewrite it as `↑(n + m) = (2 : ℚ)`, since the types of `n` and `m` don't match. But we know that `(n : ℚ) = ((n : ℤ) : ℚ)`, so we can rewrite it as `↑↑n + ↑↑m = (2 : ℚ)`, then as `↑(n + ↑m) = (2 : ℚ)` and finally as `n + ↑m = (2 : ℤ)`.

The heuristic will apply to any expression that corresponds to the application of an operator to two arguments. It will check that both sides are coerced from different types. Then,

it will try to proove from the pool of `squash_cast` lemmas that one of the coercion can be expressed as the composition of the other one with a third one. If it succeeds, it will rewrite that side of the expression and roll back to the simplifier.

2.3 Integration into Mathlib

This work resulted in a pull request on the Mathlib repository [4]: alongside the main tactic it added a set of interactive tactics that wrap the `derive` function with usual tactics:

- `norm_cast`, to normalize the goal or an expression of the context.
- `exact_mod_cast`, to normalize the goal and a lemma, then close the goal with the lemma.
- `rw_mod_cast`, the same as the `rw` tactic except it normalizes the goal at every step.
- `assumption_mod_cast`, to try `exact_mod_cast` on every lemma in the context

. The PR also added the `norm_cast` attributes to the corresponding lemmas for the usual numeric types: $\mathbb{N}$ $\mathbb{Z}$ $\mathbb{Q}$ $\mathbb{R}$ $\mathbb{C}$. Finally, some boiler-plate code in proofs about the p-adic numbers was simplified using `norm_cast`, as an example of what could be done with the tactic.

There was a lot of feedback from the users, which resulted in a second pull request [5] that is still pending, and should be merged shortly. Mainly, it will include:

- extended support for numerals (as we described above). The first version was simplier but didn't covered as many cases as the new one.
- extended documentation answering frequently asked questions
- a few bug fixes, of course

Altogether, people seem to be very happy with the tactic, and we are quite happy with how it turned out. In our opinion it is a good example of how attributes can be used to define user-extensible tactics in Lean, and succeeds at reducing the hassle of dealing with casts.

3 Normalizing field expressions

In this section, we present our work on a heuristic tool for solving non-linear inequalities.

3.1 The Polya algorithm

Polya is a proof-producing method to verify inqualities, originally written in Python by Robert Y. Lewis as part of his thesis [8]. It is a clever heuristic that uses different solvers, or modules, sharing a common data structure, the blackboard. The algorithm is targeted at reasonnably sized non-linear inequalities, that would defeat linear solvers while being straightforward to proove informally.

[4] https://github.com/leanprover-community/mathlib/pull/988
[5] https://github.com/leanprover-community/mathlib/pull/1103

There is an ongoing work by Robert Y. Lewis to write Polya as a tactic in Lean [6] , but it has been left on standby for a few years. We tried to revive it, since Mathlib still doesn't offer a tactic to solve similar problems.

Since the last update of the code, a new version of Lean came out and Mathlib evolved too. Consequently, a lot of things were broken and we had to bring the code to date before getting to work. We broke the code in independant blobs to update them one at a time. This made debugging easier and helped us take a fresh look at the project, even if later we rolled back to a less granular structure.

The blackboard is the heart of Polya: it is a pool of proven inequalities. These inequalities are expressed over normalized terms and come with a proof certificate. The blackboard is initialized with a set of hypothesis. Polya then runs a number of modules on the blackboard, which try to derive new inequalities from the existing ones. Functionning by steps, Polya adds the newly proven inequalities to the blackboard at the end of each step. It stops when a couple of contradictory inequalities have been derived, or when a certain number of steps has been exceeded. From the proof certificates of these inequalities, Polya can then build two contradictory proof terms, and thus a proof of false. At the moment, Polya has two modules: a multiplicative one and an additive one. More details about the internals of the algorithm can be found in [2].

After updating the code, these are the main points to work on in Polya:

- Prove that the normalizer is correct.
- Complete the multiplicative module.
- Complete the proof-rebuilding process.

In this section we describe our work on the normalizer. We chose to rewrite it from scratch together with a proof of correctness, rather than working around the old one. We also took the opportunity to write it in a modular way as a normalizer of expressions in any field, rather than confining ourselves to the reals. That way it can easily be used in different set ups, on different types, in different tactics, etc.

We based our work on the ring tactic written for the Coq proof-assistant [7].

3.2 The normal form

Let's begin by presenting the normal form used in Polya. It can be seen as a semi-normal form over field expressions with rational coefficients but with the peculiarity that we ignore the axiom of distribution (except for coefficients). A truely normal form that considers all axioms would be similar to the Horn normal form used in [7]. This would imply unfolding everything to a single sum, but it is key in Polya to keep an alternance between additive forms and multiplicate forms. Unfolding terms completely would defeat the purpose of the multiplicative module.

For instance $(x + y) * (x + y)$ is normalized to $(x + y)^2$ rather than $x^2 + 2xy + y^2$. By increasing the exponant we can see how unfolding the expression completely would defeat the goal of Polya. Let have x_1, x_2, y_1, y_2 such that $0 < x_1 < x_2$ and $0 < y_1 < y_2$. Then we have $(x_1 + y_1)^n < (x_2 + y_2)^n$. The proof for this is very simple, no matter how big n actually

[6] https://github.com/robertylewis/lean_polya

is. But if we set n to 100 and unfold everything, the problem becomes extremely hard to solve. Of course the modules are free to unfold some terms and add new inequalities to the board, but since Polya is aimed at such non-linear problems, it shouldn't automatically unfold everything.

3.3 Proof by reflection

We use the same proof architecture as in [7], namely a proof by reflection. Generally, this means that the equality between the normalized and the input expressions holds from a property on the normalizer function. By running the normalization in the kernel, we know that its output is equal to its input. This is to be compared with the way a tactic like the simplifier works: it runs in the VM but it produces a proof term together with the normalized expression. The immediate trade-off between the two methods is that either the normalization happens in the kernel, where it is slow, and the proof term is very short (we just refer the correctness theorem), or the normalization happens in the VM, but has to produce a longer proof term that needs to be checked.

We want to define a normalizing function modulo some of the axioms of fields. We would like to be able to define such a function inductively on the structure of the terms. Lean doesn't allow to do this on arbitary types that verify the field structure: we need to use an intermediate representation. Therefore we introduce the `term` datatype. It represents the structure of a term inside a field. It has constructors for every operation we want to reflect from the Lean expressions. These operators are defined on every type that instantiates the `discrete_field` type class. We add special constructors for variables (or atoms) and numerals.

```
inductive term : Type
  | atom : num → term
  | add  : term → term → term
  | sub  : term → term → term
  | mul  : term → term → term
  | div  : term → term → term
  | neg  : term → term
  | inv  : term → term
  | numeral : ℕ → term
  | pow_nat : term → ℕ → term
  | pow_int : term → ℤ → term
```

We also introduce an evaluation over these terms. Given a type α that verifies the axioms of fields (defined by the `discrete_field` type class), and given a dictionnary to map atoms to values of type α, we can define the following `eval` function:

```
def term.eval (ρ : dict α) : term → α
  | (atom i)  := ρ.val i
  | (add x y) := eval x + eval y
  | (sub x y) := eval x - eval y
  | (mul x y) := eval x * eval y
  | (div x y) := eval x / eval y
  | (neg x)   := - eval x
  | (inv x)   := (eval x)⁻¹
  | (numeral n)   := (n : α)
  | (pow_nat x n) := eval x ^ n
```

```
| (pow_int x n) := eval x ^ n
```

Where `dict α` is a datatype for the dictionnary:

```
structure dict (α : Type) :=
(val : num → α)
```

Then we define some `meta` function that will build a term `t : term` together with a dictionnary `ρ : dict α` from a Lean expression `e` of type `α`. If that function is well defined, we can rely on the reduction in the kernel to prove `e = eval ρ t` by reflexivity.

We could define our normalization function on terms: `norm : term → term`, and proof our main correctness theorem:

```
∀ (ρ : dict α) (t : term), eval ρ (norm t) = eval ρ t
```

But that would not be cenvenient, since a lot of the operators on fields can be defined as combinations of others. Therefore we define a second intermediate representation, that is going to be our main framework:

```
inductive nterm (γ : Type) [const_space γ] : Type
  | atom  : num → nterm
  | const : γ → nterm
  | add   : nterm → nterm → nterm
  | mul   : nterm → nterm → nterm
  | pow   : nterm → znum → nterm

def nterm.eval (ρ : dict α) : nterm γ → α
  | (atom i)  := ρ.val i
  | (const c) := ↑c
  | (add x y) := eval x + eval y
  | (mul x y) := eval x * eval y
  | (pow x n) := eval x ^ (n : ℤ)
```

This is parametrized by a type `γ` that instantiates the `const_space` type class. The `const_space` type class is just a wrapper for a field with a total order that is decidable. `γ` is meant to be a representation of rationals, and thus of the coefficients inside the term. The reason we don't just use the `ℚ` datatype present in Mathlib is that it is very inefficient. Since computation will happen in the kernel, and since we will make a lot of operations over the coefficients, we want them to be represented as efficiently as possible.

Since there is no satisfying type for that purpose in Mathlib right now, we tried to be as modular as possible, so as to integrate future work on rationals. At the moment we just use `nterm`$\mathbb{Q}$, since it doesn't block computation completely on simple examples, as long as we avoid numbers with large denominators.

We also defined a type class `morph γ α` that has to be instantiated to prove that `γ` is compatible with `α`. It states that there must be a cast between `γ` and `α`, that is compatible with the axioms of fields:

```
class morph (γ : Type) [discrete_field γ] (α : Type) [discrete_field α] :=
  (cast   : has_coe γ α)
  (morph_zero : ((0 : γ) : α) = 0)
```

```
(morph_one : ((1 : γ) : α) = 1)
(morph_add : ∀ a b : γ, ((a + b : γ) : α) = a + b)
(morph_neg : ∀ a : γ, ((-a : γ) : α) = -a)
(morph_mul : ∀ a b : γ, ((a * b : γ) : α) = a * b)
(morph_inv : ∀ a : γ, ((a⁻¹ : γ) : α) = a⁻¹)
(morph_inj : ∀ a : γ, (a : α) = 0 → a = 0)
```

The final model for terms inside a field used by our normalizer can be written as:

```
variables {α : Type} [discrete_field α] {ρ : dict α} --our field and
  dictionnary
variables {γ : Type} [const_space γ] [morph γ α] --our coefficients
```

We now write a funtion from `term` to `nterm γ` together with its correctness theorem:

```
namespace term

def to_nterm : term → nterm γ | (atom i)  := ↑i
  | (add x y) := nterm.add (to_nterm x) (to_nterm y)
  | (sub x y) := nterm.add (to_nterm x) (nterm.mul (nterm.const (-1)) (to_nterm
  y)
  | (mul x y) := nterm.mul (to_nterm x) (to_nterm y)
  | (div x y) := nterm.mul (to_nterm x) (nterm.pow (to_nterm y) (-1))
  | (neg x)   := nterm.mul (nterm.const (-1) (to_nterm x)
  | (inv x)   := nterm.pow (to_nterm x) (-1)
  | (numeral n)   := nterm.const (n : γ)
  | (pow_nat x n) := nterm.pow (to_nterm x) (n : znum)
  | (pow_int x n) := nterm.pow (to_nterm x) (n : znum)

theorem correctness {x : term} :
  nterm.eval ρ (@to_nterm γ _ x) = eval ρ x := ...

end term
```

The proof for the correctness theorem is done by induction on the input. This can be seen as the first step in our normalizing process.

3.4 The normalizing algorithm

We will now present some of the aspects of the normalization algorithm, or how to complete the following code:

```
def norm : nterm γ → nterm γ := ...
theorem correctness : ∀ (t : nterm γ), eval ρ (norm t) = eval ρ t := ...
```

In fact, here we need to add some complexity to our scheme of proof. Indeed, take a term like `x * x⁻¹` where `x` is an atom. Since zero is not assumed to be invertible, this term can't be normalized to `1` without making the assumption that `eval ρ x ≠ 0`. We want our normalizer to cancel multiplication of inverses, so we have to keep track of the assumptions we make along the way. In practice, we compute a list of `nterm γ` together with the normalized terms, and we prove that the normalization is correct under the assumption that every term in the list evaluates to a nonzero value. This sums up to the following declarations:

```
def norm : nterm γ → nterm γ := ...
def norm_hyps : nterm γ → list (nterm γ) := ...
theorem correctness : ∀ (t : nterm γ), (∀ x ∈ norm_hyps t, eval ρ x ≠ 0) →
  eval ρ (norm t) = eval ρ t := ...
```

Internally, the `norm` function will switch between an additive and a multiplicative form.

```
def sterm (γ : Type) [const_space γ] := list (nterm γ × γ)
def pterm (γ : Type) [const_space γ] := list (nterm γ × ℤ)
```

`sterm` represents a sum as a list of the sum's terms with their coefficients. Similarly, `pterm` represents a product as a list of the product's terms with their exponant. Evaluation are defined over them as for `nterm` and `term`. We also define a function to roll back to `nterm`, it has to be proven correct. The alternation between both representation is captured by the `alt` type:

```
inductive alt (γ) [const_space γ] : bool → Type
| sform : list (nterm γ) → sterm γ → alt tt
| pform : list (nterm γ) → pterm γ → alt ff
```

The first argument of `alt`'s constructors is a list of the assumptions made. From there, the details of the proof are a bit tedious, but the idea is to define operations on `alt` for the different constructors of `nterm` that are proven correct in regards to evaluation. These operations have to produce a normalized term.

For instance, let's consider addition. Without giving details about the code, the addition of two `alt` terms will begin by switching both to their `sform` representation, then merging them. This includes merging both lists while keeping a lexicographical order, adding coefficients of terms present in both lists, and finally cancelling terms whose coefficient is zero.

Finally, terms are added to the list of hypothesis in the special case where we cancel a term inside a product form, since this term has to be assumed to evaluate to a nonzero value.

3.5 Interactive code

Let's present a bit of the meta-code that wraps the normalizer into a tactic. As we have seen, meta-code is mainly needed to build a term (and a dictionnary) from a Lean expression. Meta-code is also used to produce meta-variables corresponding to the assumptions made for the normalizer's correctness theorem. These meta-variables are then given as new goals to the user.

Finally, we use a nice trick to temper the lack of performance of running the normalizer in the kernel. Since every datatype we use in the normalizer has a representation in the VM, we can run the whole normalizer in the VM. That way, it can run very efficiently, even though we can't obtain any proof term for the correctness this way. But, since the normalizer will only be used to pre-proccess hypothesis for another tactic, we don't actually need to prove the correctness of the normalizing step if the following tactic fails.

Shematically, if `t` is the term to be normalized, and `nt` the term obtained by normalizing `t` in the VM, we create a meta-variable of type `nt = norm t`. That way we can use the correctness theorem to prove `eval ρ nt = eval ρ t`. Once the tactic succeded, we need to provide a term to replace that meta-variable in the final result. Since the result from the `norm`

function running in the VM should be the same as in the kernel, we can rely on reduction in the kernel to build that term, by reflexivity. This will trigger the normalization in the kernel, but only after everything else succeded, and we know that we actually use our normalized terms inside a proof term.

To give an example of how the normalizer can be used in a larger tactics, we wrote a `field` tactic [7]. It normalizes two expressions, and if the normalized expressions match, it proves the equality between the two inputs. Using the trick mentionned above, it will only compute the normal forms in the kernel if they already matched in the VM.

3.6 Future work

Since the normalizer was designed to run in the kernel, efficiency of the code is a major concern. Here we present a few ideas to future improvements.

The major bottleneck in the normalizer right now is the representation of rationals. Adding a new datatype for rationals in Mathlib directly benefit Polya, since we made it as easy as possible to plug in a new type. We mentionned this to some users in the community, maybe that will result in a PR for Mathlib.

It seems like the second best way to improve the tactic is to reduce the cost of switching between the `sform` and `pform` representations. We tried to write `norm` as a direct induction on the `nterm` datatype, but we couldn't do it easily without using well-foundness arguments. Here we stambled on an issue with Lean's type system: from [3] we learn that definitional equality is included but not equivalent to equality. This is to ensure that definitional equality is computable. In particular, this results in well-foundness "blocking" the reduction of the `norm` function in the kernel, so we couldn't retrieve the actual normalized term.

To solve this we tried to define new intermediate types to replace `nterm` in a way that allows us to write `norm` as a direct induction, without the use of well-foundness. More precisely, we wanted to capture the alternance between the additive and multiplicative forms more finely than with the `alt` type, using mutually inductive types instead. This is more tricky than we thought, because of some limitations in the equation compiler when working with mutually inductive types.

Another improvement that we are considering is to change the normal form altogether. We could define it in a way that helps minimizing the number of operations on coefficients, and that is simplier to express inductively on terms. Of course, we have to make sure that the new normal form is still compatible with Polya's modules.

[7] https://github.com/lean-forward/field

References

[1] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. 2017.

[2] Jeremy Avigad, Robert Y. Lewis, and cody Roux. *A heuristic prover for real inequalities*. 2016.

[3] Mario Carneiro. *The Type Theory of Lean*. 2019.

[4] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. *Elaboration in Dependent Type Theory*. 2015.

[5] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. *The Lean Theorem Prover (system description)*.

[6] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. *A Metaprogramming Framework for Formal Verification*. 2017.

[7] Benjamin Grégoire and Assia Mahboubi. *Proving Equalities in a Commutative Ring Done Right in Coq*. 2005.

[8] Robert Y. Lewis. *Two Tools for Formalizing Mathematical Proofs*. 2018.