Simplifying Casts and Coercions

Robert Y. Lewis
Paul-Nicolas Madelaine

PAAR
June 30, 2020

Motivation

maths | cast woes — Mar 13, 2019

Johan Commelin 12:52 PM

It's quite humiliating, but how do I kill:

```
example (p : ℕ) [p.prime] : (p : ℝ) > 1 := sorry
```

Kenny Lau 12:54 PM

```
import data.nat.prime data.real.basic

universes u

example (p : ℕ) [p.prime] : (p : ℝ) > 1 :=
show _ < _, by rw [← nat.cast_one, nat.cast_lt]; apply nat.prime.gt_one; assumption
```

Motivation

kbuzzard feat(docs/extras) add doc about coercions between number types (#443) ✓ Latest commit aed8194 on Nov 5, 2018 History

1 contributor

283 lines (199 sloc) | 13.6 KB

Copy Raw Blame

Coercions from numbers

This document is not about coercions in general -- see section 10.6 of TPIL for a general overview. This is an overview of how to work with the coercions `ℕ → ℤ → ℚ → ℝ → ℂ` (maps which mathematicians fondly call "the identity function", and which computer scientists call `↑`) and also the natural coercions from `ℤ` to a general ring and so on.

In brief: this document might help if you have three integers `x y z`, a proof that `x * y = z`, and your goal is `↑x * ↑y = ↑z`, something which you suspect is a statement about real numbers.

The problems people have with coercions.

Here are two types of problems that people run into with coercions.

1. They are faced with a goal which is "obvious in maths", e.g.

```
a b : ℤ
⊢ ↑(a + b) = ↑a + ↑b
```

with the goal being an equality of real numbers.

2. They are faced with a goal which is "the same as a hypothesis", or "the same as something they know how to prove", e.g.

```
a b c : ℤ,
H : a + b * c = 12
⊢ ↑a + ↑b * ↑c = 12
```

These two problems are of a slightly different nature, and require two different solutions. In the next few sections I explain what I hope are enough tricks to make solving questions like this easy. But first here's a warning.

A subtlety with automatic coercions

Motivation

```
import data.complex.basic -- ℕ, ℤ, ℚ, ℝ, ℂ

variables (an bn cn dn : ℕ) (az bz cz dz : ℤ) (aq bq cq
    dq : ℚ)
variables (ar br cr dr : ℝ) (ac bc cc dc : ℂ)

example : (an : ℤ) = bn → an = bn := sorry
example : an = bn → (an : ℤ) = bn := sorry
example : az = bz ↔ (az : ℚ) = bz := sorry
example : (aq : ℝ) = br ↔ (aq : ℂ) = br := sorry
example : (an : ℚ) = bz ↔ (an : ℂ) = bz := sorry
example : (((an : ℤ) : ℚ) : ℝ) = bq ↔ ((an : ℚ) : ℂ) = (bq :
    ℝ) := sorry

example : (an : ℤ) < bn ↔ an < bn := sorry
example : (an : ℚ) < bz ↔ (an : ℝ) < bz := sorry
example : ((an : ℤ) : ℝ) < bq ↔ (an : ℚ) < bq := sorry
```

```
-- zero and one cause special problems
example : 0 < (bq : ℝ) ↔ 0 < bq := sorry
example : aq < (1 : ℕ) ↔ (aq : ℝ) < (1 : ℤ) := sorry

example : (an : ℤ) + bn = (an + bn : ℕ) := sorry
example : (an : ℂ) + bq = ((an + bq) : ℚ) := sorry
example : (((an : ℤ) : ℚ) : ℝ) + bn = (an + (bn : ℤ)) :=
    sorry

example : (((((an : ℚ) : ℝ) * bq) + (cq : ℝ) ^ dn) : ℂ)
    = (an : ℂ) * (bq : ℝ) + cq ^ dn := sorry
example : ((an : ℤ) : ℝ) < bq ∧ (cr : ℂ) ^ 2 = dz ↔
    (an : ℚ) < bq ∧ ((cr ^ 2) : ℂ) = dz := sorry
```

Goal: transparent reasoning about cast expressions

Goal

We want to:

- use the familiar Lean tactic language to reason "modulo casts."
- extend to new casts once relevant properties are proved.
- support abstract types with algebraic structure as well as $\mathbb{N}$, $\mathbb{Z}$, etc.
- support conditional simplification, e.g. $-$ on $\mathbb{N}$ if result isn't cut off.
- do all of this as transparently to the user as possible.

We do *not* try to introduce any deep theory about casts!

The `norm_cast` family

We introduce `norm_cast`, a family of tactics for the Lean proof assistant.

- Implemented in Lean as metaprograms: no changes to source code.
- Meet the desiderata in the previous slide.
- Part of Lean's standard library mathlib.
  - Invoked hundreds of times in mathlib alone.

Algorithm idea

The core component: `norm_cast`, a simplification tactic.

- Variants are assembled around the core routine.

The workflow:

- Users tag library lemmas with the `@[norm_cast]` attribute.
- Users call "mod-cast" tactics when faced with goals containing casts.
- The "mod-cast" tactics call the `norm_cast` simplification routine, which classifies these tagged lemmas and uses them at the appropriate stage of simplification.

Quick demo!

Lemma classification

- move lemmas equate expressions with casts at the root to expressions with casts further toward the leaves
  - $\uparrow$`(m + n) = `$\uparrow$`m + `$\uparrow$`n`
- elim lemmas relate expressions with casts to expressions without casts
  - $\uparrow$`a < `$\uparrow$`b` $\leftrightarrow$ `a < b`
  - $\|\uparrow$`a`$\|$ `=` $\|$`a`$\|$ for a real valued norm function defined on all normed spaces
- squash lemmas equate expressions with one or more casts at the root to expressions with fewer casts at the root
  - $\uparrow$`(1 : ℕ) = (1 : ℤ)`
  - $\uparrow\uparrow$`n = `$\uparrow$`n`

Lemma classification

Define

- $\mathcal{H}(\texttt{e})$:= number of cast applications that appear at the root of *e*
- $\mathcal{I}(\texttt{e})$:= number of non-head casts in *e*

We classify a lemma with type $lhs = rhs$ or $lhs \leftrightarrow rhs$:

- elim if $\mathcal{H}(\texttt{lhs}) = 0$ and $\mathcal{I}(\texttt{lhs}) \geq 1$
- move if $\mathcal{H}(\texttt{lhs}) = 1, \mathcal{I}(\texttt{lhs}) = \mathcal{H}(\texttt{rhs}) = 0$, and $\mathcal{I}(\texttt{rhs}) \geq 1$.
- squash if $\mathcal{H}(\texttt{lhs}) \geq 1, \mathcal{I}(\texttt{lhs}) = \mathcal{I}(\texttt{rhs}) = 0$, and $\mathcal{H}(\texttt{lhs}) > \mathcal{H}(\texttt{rhs})$.

The algorithm

1. Replace each numeral `(num : α)` with $\uparrow$`(num : ℕ)`.
   - move, squash
2. Working bottom up, move casts upward by rewriting with move lemmas and eliminate them when possible by rewriting with elim lemmas. If no rewrite rules apply to a subexpression that matches the heuristic splitting pattern, fire the *splitting procedure*.
3. Clean up any unused repeated casts that were inserted by the heuristic.
   - squash
4. Restore numerals to their natively typed form as in Step 1.
   - move, squash

Key implementation detail: Lean's built in simplifier

Heuristic splitting procedure

Fires on an expression of the form `P (↑x) (↑y)`, where

- `P` is a binary function or relation
- `x : X` and `y : Y` are both cast to type `Z`
- `X` and `Y` are not equal

Example: $((n : \mathbb{N}) : \mathbb{R}) \leq ((z : \mathbb{Z}) : \mathbb{R}) \Rightarrow ((n : \mathbb{N}) : \mathbb{Z}) \leq (z : \mathbb{Z})$
The procedure tries to find a coercion from `X` to `Y` (or vice versa).

Then tries to replace `↑x` with `↑↑x`, where the nested coercions go from `X` to `Y` to `Z`. This is justified using squash lemmas.

Interface

- `norm_cast`: simplify the goal or hypotheses
- `exact_mod_cast h`: simplify the goal and the term `h` and use `h` to close the goal
- `apply_mod_cast h`: similar, don't close the goal
- `assumption_mod_cast`: find a hypothesis that closes the goal
- `rw_mod_cast`: performs a list of rewrites, simplifying in between steps

Quick demo!

Motivation

Library *designers* think about how casts behave, and tell `norm_cast`.

Library *users* get to ignore all the details.

Users should never have to know the names of “contentless” lemmas that only manipulate casts.

Success?

`norm_cast` is used hundreds of times in mathlib and as a component of other tactics.

Part of the “default toolbox” for new users.

Buzzard, Commelin, Massot: `norm_cast` “greatly alleviates . . . pain” in their formalization of perfectoid spaces.