Simplifying Casts and Coercions*

Robert Y. Lewis[1] and Paul-Nicolas Madelaine[2]

[1] Vrije Universiteit Amsterdam, The Netherlands
r.y.lewis@vu.nl
[2] École Normale Supérieure, Paris, France
paul-nicolas.madelaine@ens.fr

Abstract

This paper introduces `norm_cast`, a toolbox of tactics for the Lean proof assistant designed to manipulate expressions containing coercions and casts. These expressions can be frustrating for beginning and expert users alike; the presence of coercions can cause seemingly identical expressions to fail to unify and rewrites to fail. The `norm_cast` tactics aim to make reasoning with such expressions as transparent as possible. They are used extensively to eliminate boilerplate arguments in the Lean mathematical library and in external developments.

1 Introduction

Many popular type-theoretic foundations for proof assistants, including the Calculus of Inductive Constructions, do not have native subtypes. Even for numeric types like $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ with a natural chain of inclusions, terms must be cast from one to another with an explicit function application. The numeral `5 : ℕ` is syntactically different from `5 : ℤ` and `5 : ℝ`. To construct the sum of variables `n : ℕ` and `z : ℤ`, one needs either an unwieldy sum operator with type $\mathbb{N} \to \mathbb{Z} \to \mathbb{Z}$ or a way to lift `n` to the larger type $\mathbb{Z}$.

Inserting coercions is a common programming language feature, and proof assistants are no exception: many modern systems will interpret `n + z` in a reasonable way. Combined with type-polymorphic operations and relations like $+$ and $<$ and generic numeral expressions, subtyping concerns can largely be ignored at the input level. However, the ease of input often belies the complexity of the underlying term. Using such terms in practice can go against the grain of intuition, especially for users coming from mathematics, where one almost never makes such distinctions. It is frustrating to realize that work must be done to unify `n < (5 : ℕ)` with `↑n < (5 : ℤ)`, where `↑n` denotes the cast of `n` into $\mathbb{Z}$.

A more intricate example of this frustration appears in the Lean development of the p-adic numbers [8] while proving

```
theorem of_int {p : ℕ} (hp : prime p) (z : ℤ) : padic_norm p ↑z ≤ 1
```

where `padic_norm : ℕ → ℚ → ℚ`. Straightforward manipulation reduces the proof to three goals:

- `prime p ⊢ (1 : ℤ) ≤ ↑p`
- `z ≠ (0 : ℤ) ⊢ -padic_val_rat p ↑z ≤ (0 : ℤ)`
- `z ≠ (0 : ℤ) ⊢ ↑z ≠ (0 : ℚ)`

*We acknowledge support from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka) and from the Dutch Research Council (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

To solve these goals by hand, the user must combine knowledge of library lemmas with lemmas that manipulate casts. The latter obscure the main ideas of the proof:

```
{ rw [←nat.cast_one, nat.cast_le], exact le_of_lt hp.one_lt },
{ rw [padic_val_rat_of_int _ hp.ne_one hz, neg_nonpos],
  exact int.coe_nat_nonneg _ },
{ exact int.cast_ne_zero.2 hz }
```

We introduce a family of tactics implemented in the Lean proof assistant [5] that aim to remove these frustrations. The core tool, `norm_cast`, tries to rewrite an expression containing casts to a normal form determined by a configurable collection of rewrite rules. Variants allow the user to apply lemmas and hypotheses and rewrite the goal "modulo" the presence of casts. The tool was developed to address usability issues raised while formalizing mathematical results in Lean[1] [4, 8]. It is incorporated into Lean's mathematical library mathlib [9], where it is already invoked 221 times, and is also used heavily in external libraries [3].

Using our tool, the script above focuses on the relevant library lemmas:

```
{ exact_mod_cast le_of_lt hp.one_lt },
{ rw [padic_val_rat_of_int _ hp.ne_one hz, neg_nonpos],
  norm_cast; simp },
{ exact_mod_cast hz }
```

Our tool is extensible: adapting it to new theories with new coercions simply requires tagging certain library lemmas. It is largely independent of the underlying logic: for example, there are no roadblocks to implementing it in systems without convertibility. It is not restricted to concrete types like $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$: it supports casts into abstract algebraic structures, casts out of arbitrary subtypes, and in general any cast for which the user can prove at least one lemma of one of the forms described in Section 4.

We provide a website[2] which points to our code in the mathlib repository, along with examples of `norm_cast` in use. Our aim in this report is *not* to give a theoretical account of type inclusions, but rather to present a powerful and lightweight procedure that is effective at dealing with these situations in practice.

"Coercion" and "cast" are sometimes used interchangeably in the literature, and "cast" can also refer to the transport of a term `t : A` to the type `B` along a type equality `A = B`. In this description, we take a *cast* `↑ : A → B` to be simply a function; it typically preserves structure, and is often injective, but neither is required. *Casting* a term `t : A` to `B` refers to applying the (often canonical) cast. A *coercion* is a cast that is automatically inserted by the elaborator. We do not consider casts along type equalities.

2 Lean Specifics

While the approach we describe can be adapted to other proof assistants, some details of this report are specific to Lean. Here we describe some of the relevant features of Lean.

Lean's elaborator inserts coercions using type classes [11, 13]. Its generic coercion function has signature

```
coe : Π {a : Sort u} {b : Sort v} [has_lift_t a b], a → b
```

where the arguments `a` and `b` are implicit and the anonymous `has_lift_t` argument is inferred by type class resolution. An instance of the type class `has_lift_t a b` witnesses a transitive

[1] Lean users previously wrote a guide to managing casts by hand. This guide is archived with our supplementary materials.

[2] https://lean-forward.github.io/norm_cast

chain of coercions from `a` to `b`, avoiding loops caused by reflexive instances. When a function application fails to typecheck, the elaborator will insert applications of `coe` and try to resolve the resulting `has_lift_t` goal. Coercions are typically inserted at the leaf nodes of an expression. Users can also manually insert casts by using `coe` directly, with prefix notation ↑.

Type-polymorphic operators and relations like `+` and `<` are also implemented with type classes. Numerals build on top of these. A numeral is represented in binary by nested applications of the following terms:

```
zero : Π (α : Type u) [has_zero α], α
one  : Π (α : Type u) [has_one α], α
bit0 : Π {α : Type u} [has_add α], α → α
bit1 : Π {α : Type u} [has_one α] [has_add α], α → α
```

Any type `α` that instantiates the classes `has_zero α`, `has_one α`, and `has_add α` supports numeral notation, e.g. `(5 : α)`. While in this description we explicitly write the types of all numerals, in practice they are typically inferred.

Lean's powerful metaprogramming framework [6] allows us to implement our tool in the language of Lean itself and include it in mathlib. The framework provides an interface to access the system's routines for unification, type class resolution, simplification, and more. Metaprograms can query and add to a Lean environment. Declarations in an environment can be tagged with parametrized *attributes*, and metaprograms are able to define new attributes, use them to tag declarations, and retrieve lists of tagged declarations.

3 Outline of the Simplification Procedure

The core routine in our procedure takes as input an expression and transforms the expression to one in which applications of the cast function ↑ are simplified. It returns a proof that the resulting expression is equal to the input. In the most common case, where the expression is a proposition, the proof of equality serves as a proof of logical equivalence.

As an example of the expected behavior, we simplify the expression `↑m + ↑n < (10 : ℤ)`, where `m, n : ℕ` are cast to ℤ. The expected form of the output is `m + n < (10 : ℕ)`, in which no casts appear; recall that `+`, `<`, and `10` are polymorphic. Our tool should proceed as follows:

1. Replace the numeral on the right with the cast of a `nat`: `↑m + ↑n < ↑(10 : ℕ)`
2. Factor ↑ to the outside of the left: `↑(m + n) < ↑(10 : ℕ)`
3. Eliminate both casts to get an inequality over ℕ: `m + n < (10 : ℕ)`

Steps 2 and 3 are effectively just applications of Lean's simplification API with certain rewrite lemmas. Step 1 has a slightly different flavor, but we will still be able to use the simplification API to implement this. Since the simplifier will handle cases of these patterns nested inside larger expressions, we can focus on the atomic situation.

Each of these steps must be justified by lemmas in the library, of course. They would not be sound for arbitrary types, operations, and relations. Users of our tool tag certain declarations with the attribute `norm_cast`, for example:

```
@[norm_cast]
theorem nat.cast_add {α : Type} [add_monoid α] [has_one α] (m n : ℕ) :
  (↑(m + n) : α) = ↑m + ↑n := ...
```

Our tool sorts these tagged declarations into three categories.

- move lemmas equate expressions with casts at the root to expressions with casts further toward the leaves, e.g. `↑(m + n) = ↑m + ↑n`. By mathlib convention, such lemmas are stated with the root cast on the left of the equation; Step 2 uses them as right-to-left rewrite rules.
- elim lemmas relate expressions with casts to expressions without casts, e.g. `↑a < ↑b ↔ a < b`. Such lemmas are stated with the expression containing casts on the left of the relation; Step 3 uses them as left-to-right rewrite rules. These lemmas are not restricted to propositional equivalences: they can also be used to modify polymorphic operations, e.g. `‖↑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, e.g. `↑(1 : ℕ) = (1 : ℤ)` and `↑↑n = ↑n`. Such lemmas are stated with the expression containing the larger number of casts on the left; Step 1 uses them alongside move lemmas to justify that `(10 : ℤ) = ↑(10 : ℕ)`, and they are used in the heuristic splitting step described below.

To simplify expressions where casts come from a variety of sources, we must sometimes split casts into pieces. Suppose `n : ℕ` and `z : ℤ`, and consider the goal `↑n + ↑z = (2 : ℚ)`. (We call the pattern `P (↑x) (↑y)`, where `P` is a binary function or relation taking two arguments of the same type and `x` and `y` are of different types, the *heuristic splitting pattern*.) We cannot rewrite the left hand side to `↑(n + z)`, since the addition would not be well typed. However, move and squash lemmas justify a transformation to `↑↑n + ↑z = ↑↑(2 : ℕ)`, where the inner casts go $\mathbb{N} \to \mathbb{Z}$ and the outer $\mathbb{Z} \to \mathbb{Q}$. We transform this to `↑(↑n + z) = ↑↑(2 : ℕ)` and then `↑n + z = ↑(2 : ℕ)`. Finally, squash lemmas reduce the right hand side to the native numeral `(2 : ℤ)`.

4 Implementation

The core simplification routine has type `expr → tactic (expr × expr)`, taking in an expression and returning it in simplified form with a proof that the output is equal to the input. Lean's simplifier API provides methods for traversing and rewriting an expression from the leaf nodes outward ("bottom up") and in reverse ("top down"). Our routine consists of four successive simplifier passes.

1. Working top down, replace each numeral `(num : α)` with `↑(num : ℕ)`. Justify these replacements with move lemmas to move casts down through applications of the constants `bit0` and `bit1`, and squash lemmas to change `↑(0 : ℕ)` and `↑(1 : ℕ)` to `(0 : α)` and `(1 : α)`.
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* described below.
3. Working top down, clean up any unused repeated casts that were inserted by the heuristic by rewriting with squash lemmas.
4. Working top down, restore numerals to their natively typed form as in Step 1. This is again justified by move and squash lemmas.

The 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`, and `X` and `Y` are not equal. The procedure tries to find a coercion from `X` to `Y` or vice versa. The existence of a coercion is expressed as a type class instance, so this can be tested by trying to resolve a type class goal `has_lift_t X Y`. Supposing the former coercion is found, the procedure tries to replace `↑x` with `↑↑x`, where the nested coercions go from `X` to `Y` to `Z`. This is justified using squash lemmas.

We use Lean's user attribute API to define an attribute `norm_cast`. This attribute is applied by the user to a lemma at or after the time of declaration. It tags the lemma for use in the procedure. A `norm_cast` lemma has the form `lhs = rhs` or `lhs ↔ rhs`, typically preceded by a sequence of quantifiers. In nearly all cases, the attribute handler can automatically classify a lemma as elim, move, or squash.

Head casts are applications of casts that appear at the root of the expression tree, as in `↑↑(x+y)`, and *internal casts* appear elsewhere. Let $\mathcal{H}(\texttt{e})$ and $\mathcal{I}(\texttt{e})$ denote the number of head casts and internal casts in `e`. Based on the number and positions of applications of casts, we classify a lemma as

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 $\mathcal{H}(\texttt{lhs}) > \mathcal{H}(\texttt{rhs})$.

When a lemma does not fit in any of these categories is tagged with the `norm_cast` attribute, an error is produced.

This classification applies to both `=` and ↔ lemmas. While users can override the classification by providing a parameter to the attribute, this is done for only 4 out of 424 attributions in mathlib. Lean's user attribute API allows us to cache the set of classified lemmas in a format convenient for the simplifier, so the classifier creates very little overhead.

A previous version of `norm_cast` relied on users classifying their lemmas manually. However, misclassified lemmas can lead to errant behavior that is hard to diagnose. From the perspective of library maintenance [12], it is much cleaner to automatically classify the rewrite rules, and we slightly redesigned the procedure to make this possible.

5 Interface

We provide a suite of tactics built around the core `norm_cast` functionality. These try to replicate the behavior of other Lean tactics "modulo casts," so that users can use familiar idioms while ignoring the presence of casts.

The core tactic `norm_cast` simplifies the current goal. Alternatively, `norm_cast at h` simplifies the type of a hypothesis `h`. These versions of the tactic are useful for cleaning up the proof state; while they rarely close a proof obligation, they make the goal easier to work on and hypotheses easier to use.

A variant `exact_mod_cast t` simplifies both the goal and the type of the expression `t`, and tries to use `t` to close the goal; `apply_mod_cast t` does similar, but allows arguments to `t` to generate new subgoals. To close the goal with a hypothesis in the local context, `assumption_mod_cast` will try `exact_mod_cast` on all plausible candidates. Finally, `rw_mod_cast [l1, ..., ln]` will use a list of lemmas to sequentially rewrite the goal, calling `norm_cast` in between rewrite steps. This generalizes the behavior of Lean's standard rewrite tactic `rw`. These variants all serve a common purpose, namely, to generalize Lean's core tactic language so that it works as expected

in the presence of casts. The `exact`, `apply`, `assumption`, and `rw` tactics are fundamental, and the `mod_cast` versions serve to extend the situations in which they can be used.

We also add move and squash lemmas into a custom simp lemma collection and define a tactic `push_cast` that simplifies with this collection; note that `push_cast` does not directly use the `norm_cast` method. Calling `push_cast` simplifies an expression in the opposite direction to `norm_cast`, meaning that casts get pushed toward the leaf nodes of expressions. This does not allow casts to be eliminated over relations, but can be useful in its own right.

6 Examples

The `norm_cast` test file[3] in mathlib demonstrates the tool in action. As a first example, we walk through a test where the heuristic splitting procedure is needed:

```
n : ℕ, z : ℤ, h : ↑n - ↑z < (5 : ℚ)  ⊢  ↑n - z < (5 : ℤ)
```

Using `exact_mod_cast h` will simplify `h` to match the goal, which is already in normal form. After changing `(5 : ℚ)` to `↑(5 : ℕ)`, `norm_cast` will fail to fire any move or elim rewrites. It will notice that `↑n - ↑z` matches the heuristic splitting pattern, and rewrite `↑n` to `↑↑n`, where the inner cast goes $\mathbb{N} \to \mathbb{Z}$ and the outer goes $\mathbb{Z} \to \mathbb{Q}$. A move rule will then match, rewriting the expression to `↑(↑n - z) < ↑(5 : ℕ)`. While both sides of the `<` are now casts to $\mathbb{Q}$, the left comes from $\mathbb{Z}$ and the right from $\mathbb{N}$, so no elim rule will fire; instead, `norm_cast` will match the entire expression to the heuristic splitting pattern and rewrite the right side to `↑↑(5 : ℕ)`. It can then rewrite with an elim lemma `↑a < ↑b ↔ a < b` to obtain `↑n - z < ↑(5 : ℕ)`, and finally normalize the numeral on the right to `(5 : ℤ)`.

There is nothing special about the embedding domain $\mathbb{Q}$ in the above example. The theorem holds when `n` and `z` are embedded into any linear ordered ring.

```
example (α : Type) [linear_ordered_ring α] (n : ℕ) (z : ℤ) (h : ↑n - ↑z < (5 : α)):
  ↑n - z < (5 : ℤ) :=
by exact_mod_cast h
```

Lean's simplifier supports conditional rewriting, and `norm_cast` makes use of this support. Note that the following example does not hold when `n > m`, since `m - n = 0`.

```
example (m n : ℕ) (h : n ≤ m) : ↑(m - n) = (↑m - ↑n : ℤ) :=
by norm_cast
```

The `norm_cast` family of tactics is used throughout mathlib. It is particularly useful in the development of the p-adic numbers $\mathbb{Q}_p$ and integers $\mathbb{Z}_p$ [8]. The rationals $\mathbb{Q}$ are embedded in the p-adics, and the definition of $\mathbb{Q}_p$ requires working with a natural number p embedded in $\mathbb{Z}$ and $\mathbb{Q}$; furthermore, $\mathbb{Z}_p$ is a subtype of $\mathbb{Q}_p$. This development makes 64 calls to tactics in the `norm_cast` family.

A lemma in the development of $\mathbb{Q}_p$ bounds the p-adic norm of an integer:

```
lemma le_of_dvd {n : ℕ} {z : ℤ} (hd : ↑(p^n) ∣ z) :
  padic_norm p ↑z ≤ ↑p ^ -(↑n : ℤ)
```

The mathlib proof of this lemma calls `exact_mod_cast` four times, to close subgoals:

- `0 ≤ p ⊢ 0 ≤ ↑p`
- `1 ≤ p ⊢ 1 ≤ ↑p`

[3] https://github.com/leanprover-community/mathlib/blob/master/test/norm_cast.lean

- `↑(p ^ n) | z ⊢ ↑p ^ n | z`
- `↑z ≠ 0 ⊢ z ≠ 0`

The proof originally written without `norm_cast` contains five explicit references to cast lemmas, and uses an explicit intermediate step that is unnecessary in the mathlib proof:

```
have hp' : (↑p : ℚ) ≥ 1, from
  show ↑p ≥ ↑(1 : ℕ), from cast_le.2 (le_of_lt hp.gt_one)
```

The tool is particularly useful alongside the `lift` tactic, which conditionally embeds terms in other types. In the following library lemma about the extended nonnegative reals `ennreal`, lifting two `ennreal`s to the type of nonnegative reals is justified by hypotheses that they are not infinite. In the resulting goal

```
a b : nnreal ⊢ ennreal.to_real ↑a ≤ ennreal.to_real ↑b ↔ ↑a ≤ ↑b
```

the casts on the left are `nnreal → ennreal`; the goal is discharged immediately by `norm_cast`.

```
lemma to_real_le_to_real {a b : ennreal} (ha : a ≠ ⊤) (hb : b ≠ ⊤) :
  ennreal.to_real a ≤ ennreal.to_real b ↔ a ≤ b :=
by { lift a to nnreal using ha, lift b to nnreal using hb, norm_cast }
```

Buzzard, Commelin, and Massot use `norm_cast` 53 times in their definition of a perfectoid space [3]. A typical use case is to match hypotheses from the `units` subtype of a monoid to goals stated in the monoid itself, e.g.:

```
γ γ₀ : units (Γ₀ R), h : γ₀ * γ₀ ≤ γ  ⊢  ↑γ₀ * ↑γ₀ ≤ ↑γ
```

These goals are rarely provable by conversion because of the complicated algebraic structure on the types involved. While traditional formalizations often make design decisions to limit the presence of coercions, they seem to be unavoidable in deep mathematical formalizations. Buzzard, Commelin, and Massot write that `norm_cast` "greatly alleviates ... pain" in their project.

The performance of `norm_cast` is not a major concern: it depends mainly on the speed of Lean's simplifier, which is heavily optimized and called regularly on complex goals. Since `norm_cast` calls the simplifier with a restricted set of rewrite rules, it normally sees close to best-case performance. It is very uncommon in practice to see `norm_cast` take more than a fraction of a second to simplify an expression. The rare cases in which it does are cases where other tactics, including full and definitional simplification, also tend to struggle. We do not know of any benchmark suites for this type of problem and have not done extensive performance testing.

7 Conclusion

The `norm_cast` family of tactics can be seen as a variant of simplification procedures, which are common tools in proof assistants. Indeed, `push_cast` is a straightforward application of Lean's simplifier, and similar functionality is found in many other systems, often in the default set of simplification lemmas.

Isabelle's standard simplifier [10] is more powerful than Lean's, but to our knowledge, the system has no tool like `norm_cast`. Some theories may set up simp lemmas in a style that approximates our procedure, particularly for use with `transfer` [7]. But it appears that approaches to managing and eliminating casts tend to be ad hoc combinations of simplification and manual work.

In Coq, unification hints [1] can sometimes help to unify terms that differ in the placement of coercions. When the necessary definitional equalities hold, the behavior of `assumption_mod_cast` and `exact_mod_cast` can be replicated by computation. The same is true in Lean, and one benefit of `norm_cast` is that it works even in the absence of definitional equalities, as is often the case with abstract algebraic structures. It is difficult to replicate the behavior of `rw_mod_cast` using computation without providing more explicit information. The `ppsimpl` preprocessing tool [2], which tries to eliminate inconvenient types and constants from a goal, shares some design features with `norm_cast`.

The `norm_cast` family aims to eliminate a source of frustration found when formalizing mathematical topics. The metaprogramming features of Lean allow it to be implemented in a lightweight and extensible way. Its development was inspired by discussion between mathematical formalizers and tactic writers. We hypothesize that there are many other similarly lightweight tools that would help to move proof assistants closer to the mathematical vernacular.

The tool is inherently coupled to its ambient library, meaning that it is only effective when the proper lemmas are tagged for its use. We thus consider it a mistake to consider tactic writing and library development separately. The `norm_cast` tool and its corresponding lemma attributions are part of mathlib, and despite not being themselves definitions or proofs, they constitute a different, procedural, kind of mathematical knowledge.

Acknowledgments. We thank Jasmin Blanchette for helpful comments, Gabriel Ebner for contributions to the code, and the mathlib community for extensive stress-testing. We thank an anonymous reviewer for enlightening examples of how such problems can be dealt with in other systems.

References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 84–98, 2009. doi:10.1007/978-3-642-03359-9_8.

[2] Frédéric Besson. `ppsimpl`: a reflexive Coq tactic for canonising goals. CoqPL, 2017.

[3] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 299–312, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373830.

[4] Sander R. Dahmen, Johannes Hölzl, and Robert Y. Lewis. Formalizing the Solution to the Cap Set Problem. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITP.2019.15.

[5] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[6] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017. doi:10.1145/3110278.

[7] Brian Huffman and Ondřej Kunčar. Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 131–146, Cham, 2013. Springer International Publishing.

[8] Robert Y. Lewis. A formal proof of Hensel's lemma over the p-adic integers. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 15–26, 2019. doi:10.1145/3293880.3294089.

[9] The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

[10] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[11] Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura. Tabled typeclass resolution, 2020. arXiv:2001.04301.

[12] Floris van Doorn, Gabriel Ebner, and Robert Y. Lewis. Maintaining a library of formal mathematics, 2020. arXiv:2004.03673.

[13] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76, 1989. doi:10.1145/75277.75283.