

A Lean tactic for normalising ring expressions with exponents (short paper)

Anne Baanen

Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
`t.baanen@vu.nl`

Abstract. This paper describes the design of the normalising tactic `ring_exp` for the Lean prover. This tactic improves on existing tactics by adding a binary exponent operator to the language of rings. A normal form is represented with an inductive family of types, enforcing various invariants. The design can also be extended with more operators.

1 Introduction

In interactive theorem proving, normalising tactics are powerful tools to prove equalities. Given an expression a , these tactics return an expression a' in normal form together with a proof that $a = a'$. In the mathematical library for the Lean theorem prover, `mathlib`, [4], the `ring` tactic normalises polynomials, i.e. expressions in a commutative (semi)ring [5]. The `ring` tactic can be directly invoked by the user and is called by the decision procedure `linarith`. The utility of the `ring` tactic is evident from the fact that it is invoked over 300 times in `mathlib`.

The `ring` tactic in Lean, and the tactic in Coq it is based on, use a Horner normal form representation of polynomials [2]. The Horner form represents a polynomial f with either of two cases: either it is constant ($f(x) = c$), or it is of the form $f(x) = g(x) * x + c$. This representation allows `ring` to uniquely and efficiently represent any expression consisting of the operators $+$ and $*$, rational numerals and variables. Problems arise for expressions containing other operators than $+$ and $*$, such as the Mersenne numbers of the form $2^n - 1$. A fundamental assumption is that the degree of each term is a constant number, meaning the Horner form cannot be simply modified to represent 2^n where x is a variable, or more generally to represent the exponentiation operator $^$ applied to compound expressions. The result is that `ring` cannot prove that $2^{n+1} - 1 = 2 * 2^n - 1$.

The `ring_exp` tactic is intended to be a normalisation tactic whose domain is a strict superset of `ring`'s, making it a drop-in replacement that can also solve goals such as $2^{n+1} - 1 = 2 * 2^n - 1$. In particular, `ring_exp` should support the operators $+$, $*$ and $^$, rational numerals and variables, and do so without sacrificing too much of the efficiency of `ring`. This paper describes the design and engineering challenges encountered in implementing `ring_exp`.

The paper discusses the version of `ring_exp` merged into `mathlib` as of commit 5c09372658.¹ Additional code and setup instructions are available online.²

2 Design

The `ring_exp` tactic uses a similar normalisation scheme to the original `ring` tactic. The input from the tactic system is an abstract syntax tree representing the expression to normalise. An `eval` function maps inputs to a type `ex` of normalised expressions. From the `ex` representation, the normalised output expression is constructed by the `simple` function, which also returns a proof that the input and output expressions are equal. The normal form should be designed in such a way that values of type `ex` are equal if and only if the input expressions are equal.

The language of (semi)rings implemented by `ring`, with binary operators $+$, $*$ and optionally $-$ and $/$, is extended in `ring_exp` with a binary exponentiation operator $^$. The input expression can consist of these operators applied to other expressions, with two base cases: rational numerals such as 0, 37 and $\frac{2}{3}$ and *atoms*. An atom is any expression which is not of the above form, e.g. a variable name x or a function application $\sin(x - 2)$, and is treated as an opaque variable in the expression. Two such expressions are considered equal if all variable assignments in any commutative ring give the same outcome.

Using a suitable representation of the normal form is crucial to easily guarantee correctness of the normaliser. The `ex` normal form used by `ring_exp` is a tree with operators at the nodes and atoms at the leaves, with certain restrictions on which subnodes may occur for each node. Compared with the abstract syntax tree, this will prohibit certain non-normalised subexpressions. The `ex` type captures these restrictions through its parameter in the enum `ex_type`, creating an inductive family of types. Each constructor allows only certain members of the `ex` family in its arguments and returns a specific type of `ex`:

```
inductive ex : ex_type → Type
| zero   : ex sum      -- 0
| sum    : ex prod → ex sum → ex sum  -- +
| coeff  : coeff → ex prod
| prod   : ex exp  → ex prod → ex prod  -- *
| exp    : ex base → ex prod → ex exp   -- ^
| var    : atom → ex base
| sum_b  : ex sum → ex base
```

The `sum_b` constructor allows sums as the base of a power, analogously to the brackets in $(a + b)^c$. For readability, we will write the representation in symbols instead of the constructors of `ex`. Thus, $n^1 * 1$ stands for `prod (exp (var n) (coeff 1)) (coeff 1)`, in the normalised form $(2 + 0)^{n^1 * 1} * 1 + (-1) + 0$ of the Mersenne number $2^n - 1$.

¹ Available at <https://github.com/leanprover-community/mathlib/tree/5c09372658>

² https://github.com/lean-forward/ring_exp

Table 1. Associativity and distributivity properties of the $+$, $*$ and $^{\wedge}$ operators.

	$+$	$*$	$^{\wedge}$
$+$	$(a + b) + c = a + (b + c)$	—	—
$*$	$(a + b) * c = a * c + b * c;$ $a * (b + c) = a * b + a * c$	$(a * b) * c = a * (b * c)$	—
$^{\wedge}$	$a^{b+c} = a^b + a^c$	$(a * b)^c = a^c * b^c$	$a^{b^c} = a^{b*c}$

The types of the arguments to each constructor are determined by the associativity and distributivity properties of the operators involved, summarised in Table 1. Since addition does not distribute over either other operator (as represented by the empty entries on the $+$ row), an expression with a sum as outermost operator cannot be rewritten so that another operator is outermost. Thus, the set of all expressions should be represented by **ex sum**. Since products do not distribute over the only other operator $^{\wedge}$, the next outermost operator after $+$ will be $*$. By associativity (the diagonal entries of the table) the left argument to $+$ should have $*$ as outermost operator; otherwise we can apply the rewrite rule $(a + b) + c \mapsto a + (b + c)$. Analogously, the left argument to the **prod** constructor is not an **ex prod** but an **ex exp**, and the right argument to the **exp** constructor is not an **ex exp** but an **ex prod**.

Adding support for a new operator will take relatively little work: extend the table of associativity and distributivity relations, insert the constructor in **ex** using the table to determine the relevant **ex_type**s, and an operation on **ex** that interprets the operator.

To construct the correctness proof showing that the input and output expressions are equal, each value in **ex** also contains a record of type **ex_info**, which holds a correctness proof for the (sub)expression, together with some auxiliary values. The operations on **ex** combine the **ex_info** fields to give the **ex_info** for the resulting normal form, using a correctness lemma: for example, **add_pf_z_sum** : $ps = 0 \rightarrow qs = qs' \rightarrow ps + qs = qs'$ constructs this proof for the input expression $ps + qs$ when ps normalises to 0. The proof corresponding to the input expression is then used to rewrite it into the output expression.

3 Complications

While the **ex** type enforces that some normalisation rules are always applied, others cannot be easily expressed on the type level. For instance, the $+$ and $*$ operators are also commutative: $a + (b + 0)$ and $b + (a + 0)$ represent equal expressions. In normal form, sums and products are in sorted order: if $a < b$, then $a + (b + 0)$ will be valid and $b + (a + 0)$ invalid. Additionally, the recursive nature of the structure of expressions means any expression a can also be represented as $(a)^1 * 1 + 0$; if the left argument to $^{\wedge}$ is 0 or $a * b + 0$, the expression is not in normal form. Not all these invariants can be enforced on the type level, careful

programming must do the rest. A mistake in maintaining these invariants is not fatal: invariants only protect completeness, not soundness, of `ring_exp`.

Careful treatment of numerals in expressions is required for acceptable run-time without sacrificing completeness. The tactic should not unfold expressions like $x * 1000$ as 1000 additions of the variable x . Representing numerals with the `coeff` constructor requires an extra step to implement addition: when terms overlap, differing only in the coefficients as for $a * b^2 * 1 + a * b^2 * 2$, their sum is given by adding their coefficients: $a * b^2 * 3$. Moreover, when the coefficients add up to 0, the correct representation is not $a * b^2 * 0 : \text{ex prod}$ but $0 : \text{ex sum}$. The same extra step occurs for exponents: $x^{a*b^2*1} * x^{a*b^2*2} = x^{a*b^2*3}$. Both cases are handled by a function `add_overlap` which returns the correct sum if there is overlap, or indicates that there is no such overlap. By choosing the order on expressions such that overlapping terms will appear adjacent in a sum, `add_overlap` can be applied in one linear scan.

A subtle complication arises when normalising in the exponent of an expression $a \wedge b$: b is always a natural number but the type of a is any ring. To correctly compute a normalised expression for b , the tactic needs to keep track of the type of b . The calculations of the `eval` function are thus done in an extension of the `tactic monad` [1], called the `ring_exp_m monad`. Using a reader monad transformer [3], `ring_exp_m` stores the type of the current expression as a variable which can be swapped out when calling `eval` on exponents.

Implementing subtraction and division also requires more work, since semirings in general do not have well-defined $-$ or $/$ operators. The tactic uses type-class inference to determine whether the required extra structure exists on the type. When this is the case, the operators can be rewritten: $a - b$ becomes $a + (-1) * b$ in a ring and a/b becomes $a * b^{-1}$ in a field.

For completeness, atoms should be considered up to definitional equality: $(\lambda x, x) a$ and $(\lambda x y, x) a b$ reduce to the same value a , so they should be treated as the same atom. The `ring_exp_m monad` contains a state monad transformer to keep track of which atoms are definitionally equal. The state consists of a list of all distinct atoms encountered in the whole input expression, and any comparisons between atoms are instead made by comparing their indices in the list. An additional benefit is that the indices fix a consistent ordering on the atoms in an expression, used to keep arguments to commutative operators in sorted order.

Within atoms, there may be subexpressions that can be normalised as well. Instead of running the normaliser directly, `ring_exp` calls the built-in tactic `simp` with the normaliser as an argument. The `simp` tactic calls a given normaliser on each subexpression, rewriting it when the normaliser succeeds.

4 Optimisations

An important practical consideration in implementing `ring_exp` is its efficiency, especially running time. Among the 300 calls to `ring` in `mathlib`, approximately half are invocations on linear expressions by the tactic `linarith`. Since `ring_exp`

is intended to work as a drop-in replacement for `ring`, its performance characteristics, especially for linear expressions, should be comparable.

Optimising the code was a notable part of the implementation of `ring_exp`, guided by Lean’s built-in profiler. Originally, the tactic used Lean’s elaborator to fill in implicit arguments and typeclass instances when constructing terms. Profiling revealed that up to 90% of running time could be spent in elaboration. The solution was to supply all arguments explicitly and maintain a cache of typeclass instances, also caching the expressions for the constants 0 and 1. It was possible to make these changes without large changes to the codebase, because the required extra fields were hidden behind the `ring_exp_m` and `ex_info` types.

Since the tactic works bottom up, constructing normal forms by applying each operator to the normal form of its operands, after each operation the sub-proofs can be discarded. Each `ex` value carries its proof of normalisation, but as soon as an operation, such as adding two expressions a and b , has finished, the normalisation proofs of a and b are no longer needed and are deleted. Similarly, any proof that reduces to reflexivity is deleted. Not only does discarding these terms reduce memory usage, the terms to be typechecked are smaller, directly reducing running time.

The result of these optimisations can be quantified by comparing the running time of the `ring` and `ring_exp` tactics: `ring_exp` should not be noticeably slower than `ring`. As a benchmark, we can compare the performance of `ring` and `ring_exp` on randomly generated expressions.³ The performance measure is the tactic execution time reported by the Lean profiler, running on a 3 GHz Intel® Core™ i5-8500 CPU with 16 GB of RAM. On arbitrary expressions, the benchmark indicates that `ring_exp` is a constant factor of approximately 3.41 times slower than `ring`; on linear expressions such as passed by `linarith`, `ring_exp` is 1.68 times slower than `ring`.

5 Conclusions

Compared with the `ring` tactic, the `ring_exp` tactic can deal with a strict superset of expressions, and can do so without sacrificing too much speed. The extensible nature of the `ex` type should make it simple to add support for more operators to `ring_exp`. Independently, it should be possible to adapt the `ex` type to other algebraic structures such as lattices or vector spaces. Although more optimisations are needed to fully equal `ring` in efficiency, the `ring_exp` tactic already achieves its goal of being a more general normalisation tactic. These results are as much a consequence of engineering effort as of theoretical work.

Acknowledgements Floris van Doorn, Mario Carneiro and Robert Y. Lewis reviewed the code and suggested improvements. Many thanks for the help!

The author has received funding from the NWO under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

³ The benchmark program and an analysis script is available at https://github.com/lean-forward/ring_exp.

References

- [1] Gabriel Ebner et al. “A metaprogramming framework for formal verification”. In: *Proc. ACM Program. Lang.* 1 (Aug. 2017). DOI: [10.1145/3110278](https://doi.org/10.1145/3110278).
- [2] Benjamin Grégoire and Assia Mahboubi. “Proving equalities in a commutative ring done right in Coq”. In: *Theorem Proving in Higher Order Logics*. Ed. by J. Hurd and T. Melham. Springer Berlin Heidelberg, 2005, pp. 98–113.
- [3] Sheng Liang, Paul Hudak, and Mark Jones. “Monad transformers and modular interpreters”. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’95. San Francisco, California, USA: ACM, 1995, pp. 333–343. DOI: [10.1145/199448.199528](https://doi.org/10.1145/199448.199528).
- [4] Leonardo de Moura et al. “The Lean theorem prover (system description)”. In: *Automated Deduction*. Ed. by A. P. Felty and A. Middeldorp. CADE-25. Cham: Springer International Publishing, 2015, pp. 378–388.
- [5] The mathlib Community. “The Lean mathematical library”. In: *9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Ed. by J. Blanchette and C. Hrițcu. CPP 2020. ACM, 2020.