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 $a = a'$. For instance, in `mathlib`, the mathematical library for the Lean theorem prover [6], the `ring` tactic normalises expressions in a commutative (semi)ring [8]. The `ring` tactic can be directly invoked by the user and is called by the decision procedure `linarith`. The utility of `ring` 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 [3]. The Horner form represents a polynomial f with one of two cases: either it is constant ($f(x) = c$), or it is of the form $f(x) = c + x * g(x)$. This representation allows `ring` to uniquely and efficiently represent any polynomial, i.e. any expression consisting of the operators $+$ and $*$, rational numerals and variables. Problems arise when expressions include other operators than $+$ and $*$, such as the exponentiation operator $^\wedge$. The Horner form fundamentally assumes the degree of a term is a constant integer, so it cannot be simply modified to represent variable exponents, or more generally to represent $^\wedge$ applied to compound expressions. The result is that `ring` cannot prove that $2^{n+1} - 1 = 2 * 2^n - 1$ for a free variable $n : \mathbb{N}$.

The `ring_exp` tactic is a new normalisation tactic whose domain is a strict superset of `ring`'s, making it a drop-in replacement. In particular, `ring_exp` supports the operators $+$, $*$ and $^\wedge$, rational numerals and variables, and does 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]

[1] Available at https://github.com/leanprover-community/mathlib/tree/5c09372658
[2] https://github.com/lean-forward/ring_exp

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. The normal form should be designed in such a way that values of type `ex` are equal if and only if the input expressions can be proved equal using the axioms of semirings. From the `ex` representation, the normalised output expression is constructed by the `simple` function. Both `eval` and `simple` additionally return a proof showing the input and output expressions are equal. The `ring_exp` tactic does not use reflection but directly constructs proof terms, as is typical for tactics in `mathlib` [8].

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 in every commutative ring they evaluate to equal values, for any assignment to the atoms.

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, but prohibits classes of non-normalised expressions by restricting which subnode can occur for each node. The `ex` type captures these restrictions through a parameter in the enum `ex_type`, creating an inductive familty of types. Each constructor allows specific members of the `ex` family in its arguments and returns a specific type of `ex`:

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

Each constructor additionally takes an `ex_info` record for auxiliary information used to construct correctness proofs. 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, the term `prod (exp (var n) (coeff 1)) (coeff 1)` (with `ex_info` fields omitted) can be written as $n^1 * 1$ in the normalised form $(2+0)^{n^1 * 1} * 1 + (-1) + 0$ of the Mersenne number $2^n - 1$.

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 seen

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

	$+$	$*$	$\hat{}$
$+$	$(a+b)+c = a+(b+c)$	—	—
$*$	$(a+b)*c = a*c+b*c$; $a*(b+c) = a*b+a*c$	$(a*b)*c = a*(b*c)$	—
$\hat{}$	$a^{b+c} = a^b + a^c$	$(a*b)^c = a^c * b^c$	$\left(a^b\right)^c = a^{b*c}$

from 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 $*$ distributes over $+$ but not over $\hat{}$, 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 left argument to `exp` is an `ex base`.

The `eval` function interprets each operator in the input expression as a corresponding operation on `ex`, building a normal form for the whole expression out of normalised subexpressions. The operations on `ex` build the correctness proof of normalisation out of the proofs for subexpressions using a correctness lemma: for example, `add_pf_z_sum : ps = 0 → qs = qs' → ps + qs = qs'` constructs this proof for the input expression `ps + qs` when `ps` normalises to 0. These correctness proofs are stored in the `ex_info` record.

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

3 Complications

The `ex` type enforces that distributivity and associativity rules are always applied, but commutative semirings have more equations. In a normal form, arguments to commutative operators should be sorted according to some linear order $\prec$: if $a \prec b$, then $a+(b+0)$ is normalised and $b+(a+0)$ is not. Defining a linear order on `ex` requires an order on atoms; definitional equality of atoms is tested in the `tactic` monad [2], so a well-defined order on atoms cannot be easily expressed on the type level. Additionally, the recursive structure of expressions means any expression a can also be represented as $(a)^1 * 1 + 0$; if the left argument to $\hat{}$ is 0 or $a*b+0$, the expression is not in normal form. These invariants which are not enforced on the type level, are instead maintained by careful programming. A mistake in maintaining these invariants is not fatal: invariants only protect completeness, not soundness, of `ring_exp`.

Efficient handling of numerals in expressions is required for acceptable runtime 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 $a * b^2 * 0$: `ex prod` but 0 : `ex sum`. Coefficients must also be handled correctly in 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 ^ b`: `b` is always a natural number but the type of `a` is an arbitrary 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, called the `ring_exp_m` monad. Using a reader monad transformer [5], `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 typeclass 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: `(λ x, x) a` and `(λ 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 induce an order on atoms, which is used to sort arguments to commutative operators. 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`. Profiling revealed that up to 90% of running time could be spent on inferring implicit arguments and typeclass instances. The solution was to pass all argu-

ments explicitly and maintain a cache of typeclass instances, also caching the expressions for the constants `0` and `1`. It was possible to apply this solution 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 subproofs 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. This results in smaller proof terms, reducing memory usage and type checking time.

The result of these optimisations can be quantified by comparing the running time 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 3.88 times slower than `ring`; on linear expressions such as passed by `linarith`, `ring_exp` is 1.67 times slower than `ring`.

5 Discussion

The `ring` tactic for Coq and Lean can efficiently convert expressions in the language of semirings to normal form. A normalizing procedure for polynomials is also included with HOL Light [4] and Isabelle/HOL [7], and decision procedures exist that support exponential functions [1]; none of these allows compound expressions in exponents.

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

Floris van Doorn, Mario Carneiro and Robert Y. Lewis reviewed the code and suggested improvements. Jasmin Blanchette and Robert Y. Lewis read this paper and gave useful suggestions. Many thanks for the help!

[3] The benchmark program and analysis scripts are available at https://github.com/lean-forward/ring_exp.

References

1. Akbarpour, B., and Paulson, L.C.: Extending a Resolution Prover for Inequalities on Elementary Functions. In: Dershowitz, N., and Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, pp. 47–61. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
2. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., and Moura, L. de: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. 1 (2017)
3. Grégoire, B., and Mahboubi, A.: Proving equalities in a commutative ring done right in Coq. In: Hurd, J., and Melham, T. (eds.) Theorem Proving in Higher Order Logics, pp. 98–113. Springer Berlin Heidelberg (2005)
4. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., and Camilleri, A. (eds.) Formal Methods in Computer-Aided Design, pp. 265–269. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
5. Liang, S., Hudak, P., and Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '95, pp. 333–343. ACM, San Francisco, California, USA (1995)
6. Moura, L. de, Kong, S., Avigad, J., Doorn, F. van, and Raumer, J. von: The Lean theorem prover (system description). In: Felty, A.P., and Middeldorp, A. (eds.) Automated Deduction. CADE-25, pp. 378–388. Springer International Publishing, Cham (2015)
7. Nipkow, T., Wenzel, M., and Paulson, L.C. (eds.): 1. The Basics. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
8. The mathlib Community: The Lean mathematical library. In: Blanchette, J., and Hriţcu, C. (eds.) 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. CPP 2020. ACM (2020)