An extensible ad hoc interface between Lean and Mathematica

Robert Y. Lewis

Carnegie Mellon University, Pittsburgh, PA. USA
rlewis1@andrew.cmu.edu

Abstract. We implement a user-extensible ad hoc connection between the Lean proof assistant and the computer algebra system Mathematica. By reflecting the syntax of each system in the other and providing a flexible interface for extending translation, our connection allows for the exchange of arbitrary information between the two systems. We show how to make use of the Lean metaprogramming framework to verify certain Mathematica computations, so that the rigor of the proof assistant is not compromised.

1 Introduction

Many researchers have noted the disconnect between computer algebra and interactive theorem proving. In the former, one typically values speed and flexibility over absolute correctness. To be more efficient or user-friendly, a computer algebra system (CAS) may blur the distinction between polynomial objects and polynomial functions, assume that sufficiently small terms at the end of a series are zero, or resort to numerical approximations without warning. Such simplifying assumptions can make sense in the context of computer algebra; the capabilities of these systems make them indispensable tools to many working mathematicians. These assumptions, though, are antithetical to the goals of interactive theorem proving (ITP), where every inference must be justified by appeal to some logical principle. The strict logical requirements and lack of many familiar algorithms discourage many mathematicians from working with proof assistants.

Integrating computer algebra into proof assistants is one way to reduce this barrier to entry, and bridges between the two types of systems have been built in a variety of ways. We contribute another such bridge, between the proof assistant Lean [11] and the computer algebra system Mathematica [20]. Our connection is inspired by the architecture described by Harrison and Théry [13]. Since Mathematica is one of the most commonly used computer algebra systems, and a user with knowledge of the CAS can extend the capabilities of our link, we hope that the familiarity will lead to wider use.

Our link separates the steps of communication, semantic interpretation, and verification: there is no a priori restriction on the type of information that can be shared between the systems. With the proof assistant in the "master" role, Lean expressions are exported to Mathematica, where they can be interpreted and

manipulated. The results are then imported back into Lean and reinterpreted. Using Lean's metaprogramming framework, one can write scripts that verify properties of the translated results. This style of interaction, where verification happens on a per-case basis after the computation has ended, is called *ad hoc*.

By performing calculations in Mathematica and verifying the results in Lean, we relax neither the rigor of the proof assistant nor the efficiency of the CAS. The CAS can alternatively be used as an untrusted oracle, or can play a purely informative role, where its output does not appear in the final proof term. This range of possibilities is intended to make our link attractive to multiple audiences. The working mathematician, who balks at the restrictions imposed by a proof assistant, may find that full access to a familiar CAS is worth the tradeoff in trust. Industrial users are often happy to trust both large-kernel proof assistants and computer algebra systems; the rigor of Lean with Mathematica as an oracle falls somewhere in between. And certifiable algorithms are still available to users who demand complete trust.

The source for this project, and supplementary documentation, is available at http://www.andrew.cmu.edu/user/rlewis1/leanmm/. In this paper, we use `Computer Modern` for Lean code and `TeX Gyre Cursor` for Mathematica code. We begin by describing some of the salient features of the two systems. Section 3 discusses the translation of Lean expressions into semantically similar Mathematica expressions, and vice versa; Section 4 describes further details of how the link is implemented. In Section 5 we give examples of the link in action. We conclude with a discussion of related and future work.

2 System descriptions: Lean and Mathematica

2.1 Lean

Lean is a proof assistant being developed at Microsoft Research [11]. Written in C++, the system is highly performant. Lean has been designed from the beginning to support strong automation; it aims to eventually straddle the line between an interactive theorem prover with powerful automation, and an automated theorem prover with a verified code base and interactive mode.

Lean is based on the Calculus of Inductive Constructions, an extension of the lambda-calculus with dependent types and inductive definitions. There is a non-cumulative hierarchy of type universes `Sort u`, $u \geq 0$, with the abbreviations `Prop = Sort 0` and `Type u = Sort (u+1)`. The bottom level `Prop` is impredicative and proof-irrelevant. We refer readers to [7], [8], and [11] for more details about the CIC and Lean's implementation. The system is foundationally similar to Coq, but has a smaller kernel.

Lean's standard library uses type classes to implement an abstract algebraic hierarchy. Arithmetic operations, such as $+$ and $*$, and numerals are generic over types that instantiate the appropriate classes. As an example, the addition operator has the signature

```
add {u} : Π {A : Type u} [has_add A], A → A → A.
```

The notation `{A : Type u}` denotes that the argument `A` is an implicit variable, meant to be inferred from further arguments; `has_add : Type u → Type u` is a type class, and the notation `[has_add A]` denotes that a term of that type is to be inferred using type class resolution. The universe argument `u` indicates that `add` is parametric over universe levels.

The dependently typed language implemented in Lean is flexible enough to serve as its own *metaprogramming language* [10]. Data types and procedures implemented in Lean's underlying C++ code base are exposed as constants, using the keyword `meta` to mark a distinction between the object language and this extension. Expressions can be executed in the Lean virtual machine, which replaces these constants with their underlying implementation. Meta-definitions permit unbounded recursion but are otherwise quite similar to standard definitions.

Combined with the declaration of the types `expr`, which exposes the syntax of Lean expressions in Lean itself, and `tactic_state`, which exposes the environment and goals of a tactic proof, this metaprogramming framework allows users to write complex procedures for constructing proofs. A term of type `tactic A` is a function `tactic_state → tactic_result A`, where a result is either a success (pairing a new `tactic_state` with a term of type `A`) or a failure. Proof obligations can be discharged by terms of type `tactic unit`; such a term is executed in the Lean virtual machine to transform the original `tactic_state` into one in which all goals have been instantiated. More generally, we can think of a term of type `tactic A` as a program that attempts to construct a term of type `A`, while optionally changing the tactic state.

When writing tactics, the command `do` enables Haskell-like monadic syntax. For example, the following tactic returns the number of goals in the current tactic state. The type of `get_goals` is `tactic (list expr)`, where `list` is the standard (object-level) type defined in the Lean library.

```
meta def num_goals : tactic nat :=
do gs ← get_goals,
   return (length gs)
```

Lean allows the user to tag declarations with *attributes*, and provides an interface `name → tactic (list name)` to retrieve a list of declarations tagged with a certain attribute.

Many features and subtleties of the metaprogramming framework are discussed in [10]. In closing, we note how the framework is used for this project.

We define the function `mm_form_of_expr : expr → string` recursively on the type `expr` to represent Lean syntax in Mathematica. We also declare a meta-constant `mathematica.execute : string → tactic expr`. This program, which is implemented in C++, passes the input string to Mathematica and returns a Lean expression encoding Mathematica's output. (The type `mmexpr` of the output expression, which represents Mathematica term structure, is described in Section 3.2 below.) The program `expr_of_mmexpr : mmexpr → tactic expr` and variants search the context for attributed translation rules, and try to apply these rules to convert the Mathematica expression into a meaningful Lean expression. Finally, various tactics are defined to make use of these results.

2.2 Mathematica

Mathematica is a popular symbolic computation system developed at Wolfram Research, implementing the Wolfram Language [20]. Along with supporting a vast range of mathematical computations, Mathematica includes collections of data of various types and tools for manipulating this data.

Mathematica provides comprehensive tools for rewriting and solving polynomial, trigonometric, and other classes of equations and inequalities; solving differential equations, both symbolically and numerically; computing derivatives and integrals of various types; manipulating matrices; performing statistical calculations, including fitting and hypothesis testing; and reasoning with classes of special functions.

This large library of functions is one reason to choose Mathematica for our linked CAS. Another reason is its ubiquity: Mathematica is frequently used in undergraduate mathematics and engineering curricula. Lean beginners who are accustomed to Mathematica do not need to learn a new CAS language for the advanced features of this link.

For those unfamiliar with Mathematica syntax, we note some features and terminology that will help to understand the code fragments in this paper.

- Function application is written using square brackets, e.g. `Plus[x, y]`. Many functions are variadic: that is, we can also write `Plus[x, y, z]`. Of course, we can use common notation like `x + y + z` instead.
- Alternatively, we can write unary function application in postfix form: `x^2 - 2x + 1 // Factor` is equivalent to `Factor[x^2 - 2x + 1]`.
- In the expression `Plus[x, y]`, we refer to `Plus` as the *head symbol* and `x` and `y` as the *arguments*.
- The Wolfram Language is untyped, so head symbols such as `Plus` and `Factor` can be applied to any argument or sequence of arguments. Evaluation is often restricted to certain patterns: `Plus[2, 3]` will evaluate to `5`, but `Plus[Factor, Plus]` will not reduce. Nevertheless, both are well-formed Mathematica expressions.
- There is no strong distinction between defined and undefined head symbols. The user is free to introduce a new head symbol and use it at will. The computational behavior of this head symbol can be fully or partially defined via pattern matching rules.

3 The translation procedure

Our bridge is used to import information from Mathematica into Lean, usually about some particular Lean expression. The logical foundations and semantics of the two systems are quite different, and we should not expect a perfect correspondence between the two. However, in many situations, an expression in Lean has a counterpart in Mathematica with a very similar "meaning." We can exploit these similarities by ignoring the unsoundness of the translations in both directions and attempting to verify, post hoc, that the resulting expression has the intended properties.

```
meta inductive expr
| var          : nat → expr
| sort         : level → expr
| const        : name → list level → expr
| mvar         : name → expr → expr
| local_const : name → name → binder_info → expr → expr
| app          : expr → expr → expr
| lam          : name → binder_info → expr → expr → expr
| pi           : name → binder_info → expr → expr → expr
| elet         : name → expr → expr → expr → expr
| macro        : macro_def → ∀ n, (fin n → expr) → expr
```

Fig. 1. Lean expression kinds

As a running toy example, suppose we want to show in Lean that

`x : real ⊢ x^2 - 2x + 1 ≥ 0.`

Factoring the left-hand side of the inequality makes this a one-step proof (assuming we've proved that squares are nonnegative). It is nontrivial to write a reliable and efficient polynomial factoring algorithm, but luckily, one is implemented in Mathematica. So we would like to do the following:

1. Transform the Lean representation of $x^2 - 2x + 1$ into Mathematica syntax.
2. Interpret this into the Mathematica representation of the same polynomial.
3. Use Mathematica's `Factor` function to factor the polynomial.
4. Transform this back into Lean syntax, and interpret it as a Lean polynomial.
5. Verify that the new expression is equal to the old.
6. Substitute this equality into the goal.

Step 6 is easily achieved with Lean's `rewrite` tactic. Lean's simplifier handles step 5 – checking that a polynomial has been factored correctly is much easier than factoring it in the first place. And, once we have a valid Mathematica expression, step 3 is trivial. We describe steps 1, 2, and 4 here.

3.1 Translating Lean to Mathematica

The Lean expression grammar is presented (in Lean syntax) in Figure 1; we elaborate below. For the sake of brevity, we will not discuss the implementations of `name`, `level`, or `binder_info`.

Each Lean expression exists in an environment, which contains the names, types, and definitions of previous declarations. The `const` kind accesses a previous declaration, instantiated to particular universe levels if the declaration is parametric. In addition to declarations in its environment, an expression may refer to its local context, which contains variables and hypotheses of kind `local_const`. In the toy example introduced above, `x` is a local constant. A local constant has a unique name, a formatting name, and a type.

The expression kinds `lam` and `pi` respectively represent lambda-abstraction and the dependent function type. (Non-dependent function types are degenerate

cases of pi types.) Each contains a name for the bound variable, the type of the variable, and the expression body. Bound variables of kind `var` are anonymous within the body, being represented by De Bruijn indices [16]. Application of one expression to another is represented by the `app` kind.

Type universes are implemented by the expression kind `sort`. Metavariables represent placeholders in partially constructed expressions; the `mvar` kind holds the name and type of the placeholder. Let expressions (`elet`) bind a named variable with a type and value within a body. We do not describe macro expressions, as they are not supported by our link.

To represent this syntax in Mathematica, we define `mathematica_form_of_expr : expr` $\to$ `string` by recursion over the `expr` datatype. We associate a Mathematica head symbol `LeanVar`, `LeanSort`, `LeanConst`, etc. to each constructor of `expr`. Names, levels, lists of levels, and binder information are also represented.

Some of the information contained in a Lean expression has little plausible use in Mathematica, or is needlessly verbose: for example, it is hard to contrive a scenario in which the full structure of a Lean `name` is used in the CAS. Nonetheless, we do not strip any information at this stage, to preserve the property that an expression reflected into and immediately back from Mathematica should translate to the original expression without any additional information.

In our running example, we work on the expression $x^2 - 2x + 1$. The fully-elaborated Lean expression and its Mathematica representation are too long to print here, but they can be viewed in the supplementary documentation; we consider the more concise example of $x + x$. If we use strings to stand in for terms of type `name`, natural numbers in place of universe levels, and the string `"bi"` in place of the default `binder_info` argument, and we abbreviate

```
𝒳 := local_const "17.27" "x" "bi" (const "real" []),
```

the full form of $x + x$ is

```
app (app (app (app (const "add" [0]) (const "real" []))
              (const "real.has_add" [])) 𝒳 𝒳.
```

The corresponding Mathematica expressions are

```
X := LeanLocal["17.27", "x", "bi", LeanConst["real", {}]]

LeanApp[LeanApp[LeanApp[LeanApp[LeanConst["add", {0}],
  LeanConst["real", {}]], LeanConst["real.has_add", {}]],
  X], X].
```

Since the head symbols `LeanApp`, `LeanConst`, etc. are uninterpreted in Mathematica, this representation is not yet useful. We wish to exploit the fact that many Lean terms have semantically similar counterparts in Mathematica. For instance, the Lean constants `add` and `mul` behave similarly to the Mathematica head symbols `Plus` and `Times`; both systems have notions of application, although they handle the arity of applications differently; and Mathematica's concept of a "pure function" is analogous to lambda-abstraction in Lean.

We thus define a translation function `LeanForm` in Mathematica that attempts to interpret the syntactic representation. Mathematica functions are typically defined using pattern matching. The `LeanForm` function, then, will look

for familiar patterns (e.g. `add A h x y`, in Mathematica syntax) and rewrite them in translated form (e.g. `Plus[LeanForm[x], LeanForm[y]]`). Users can easily extend this translation function by asserting additional equations; a default collection of equations is loaded automatically.

For our factorization example, we want to convert Lean arithmetic to Mathematica arithmetic. Among other similar rules, we will need the following:

```
LeanForm[LeanApp[LeanApp[LeanApp[LeanApp[LeanConst["add", _],
 _], _], x_], y_]] := Inactive[Plus][LeanForm[x],LeanForm[y]]
```

Note that this pattern ignores the type argument and type-class instance in the Lean term. These arguments are irrelevant to Mathematica and can be inferred again by Lean in the back-translation. We block Mathematica's computation with the `Inactive` head symbol; otherwise, Mathematica would eagerly simplify the translated expression, which can be undesirable. The function `Activate` strips these annotations and allows reduction.

Numerals in Lean are type-parametric and are represented using the constants `zero`, `one`, `bit0`, and `bit1`. To illustrate, the type signature of the latter is

```
bit1 {u} : Π {A : Type u}, [has_add A] → [has_one A] → A → A
```

and the numeral 6 is represented as `bit0 (bit1 one)`; the type of this numeral is expected to be inferable from context. We can use rules similar to the above to transform Lean numerals into Mathematica integers:

```
LeanForm[LeanApp[LeanApp[LeanApp[LeanApp[
  LeanConst["bit1", _], _], _], _], t_]] := 2*LeanForm[t]+1.
```

Applying `LeanForm` will not necessarily remove all occurrences of the head symbols `LeanApp`, `LeanConst`, etc. This is not a problem: we only need to translate the "concepts" with equivalents in Mathematica. Unconverted subterms – for instance `X`, which contains applications of `LeanLocal` and `LeanConst` – will be treated as uninterpreted constants by Mathematica, and the back-translation described below will return them to their original Lean form.

In our running example (keeping the abbreviation `X`), applying the `LeanForm` and `Activate` functions produces the expression

```
Plus[1,Times[-2, X], Power[X, 2]].
```

Applying `Factor` produces `Power[Plus[-1, X], 2]`.

The expression

```
X := LeanLocal["17.27", "x", "bi", LeanConst["real",{}]]
```

has been treated as a constant throughout the process, and contains information that will rarely if ever be of use in Mathematica. The excess information does little harm here, but in more complex situations, carrying around this excess information can be unwieldy. We provide Mathematica functions `LeanCollapse` and `LeanInflate` to reduce this excess baggage during computation.

```
inductive mmexpr
| sym   : string → mmexpr
| str   : string → mmexpr
| mint  : signed_num → mmexpr
| mreal : float → mmexpr
| app   : mmexpr → list mmexpr → mmexpr
```

Fig. 2. Mathematica expression kinds

3.2 Translating Mathematica to Lean

Mathematica expressions are composed of various atomic number types, strings, symbols, and applications, where one expression is applied to a list of expressions. We represent this structure in Lean with the data type `mmexpr` (Figure 2).

The result of a Mathematica computation is reflected into Lean as a term of type `mmexpr`. This is analogous to the original export of our Lean expression into Mathematica; it remains to interpret it as something meaningful.

A *pre-expression* in Lean is a term where universe, implicit, and inferable arguments are omitted. It is not expected to type check, but can be turned into a type-correct term via elaboration. For instance, the pre-expression `` `(add nat.one nat.one) `` elaborates to `add.{0} nat nat.has_add nat.one nat.one`. The notation `` `(...) `` instructs Lean's parser to interpret the quoted text as a term of type `pexpr`. Pre-expressions share the same structure as expressions.

Most Mathematica expressions correspond to pre-expressions: they may be type-ambiguous, and contain less information than their Lean counterparts. Thus we normally expect to interpret terms of type `mmexpr` as pre-expressions, and to use the Lean elaborator to turn them into full expressions. However, in rare cases an `mmexpr` may already correspond to a full expression: the unmodified representation of a Lean expression, sent back into Lean, should interpret as the original expression. We provide two extensible translation functions, `expr_of_mmexpr` and `pexpr_of_mmexpr`, to handle both of these cases. Since the implementations are similar, we focus on the latter here.

The function `pexpr_of_mmexpr : trans_env → mmexpr → tactic pexpr` takes a translation environment and an `mmexpr`, and, using the attribute manager, attempts to return a pre-expression. (Since the tactic monad includes failure, the process may also fail if no interpretation is found.) Interpreting strings as pre-expressions, or, indeed, as expressions, is straightforward. Since Mathematica ints may be used to represent numerals in many different Lean types, expressions built with `mint` are interpreted as untyped numeral pre-expressions.

The `sym` and `app` cases are more complex: this part of the translation procedure is extensible by the user. We define three classes of translation rules:

- A sym-to-pexpr rule, of type `string × pexpr`, identifies a particular Mathematica symbol with a particular pre-expression. For example, the rule `("Real", `(real))` instructs the translation to replace the Mathematica symbol `Real` with the Lean pre-expression `const "real"`.

– A keyed app-to-pexpr rule is of type `string × (trans_env → list mmexpr → tactic pexpr)`. When the procedure encounters an `mmexpr` of the form `app (sym head) args` – that is, the Mathematica head symbol `head` applied to a list of arguments `args` – it will try to apply all rules that are keyed to the string `head`. The rules for interpreting arithmetic expressions follow this pattern: a rule keyed to the string `"Plus"` will interpret `Plus[`t_1`, ..., `t_n`]` by folding applications of `add` over the translations of t_1 through t_n.
– An unkeyed app-to-pexpr rule is of type `trans_env → mmexpr → list mmexpr → tactic pexpr`. If the head of the application is a compound expression, or if no keyed rules execute successfully, the translation procedure will try unkeyed rules. One such rule attempts to translate the head symbol and arguments independently, and fold application over these translations.

Rules of these three types can be declared by the user and tagged with the corresponding attribute. The translation procedure uses Lean's caching attribute manager to collect relevant rules at runtime.

Returning to our example, we have translated `x^2 - 2x + 1` and factored it to produce `Power[Plus[-1, X], 2]`. This is reflected as the Lean `mmexpr`

```
app (sym "Power") [app (sym "Plus") [mint -1, X], mint 2],
```

where

```
X := app (sym "LeanLocal") [str "17.27", str "x", str "bi",
                            app (sym "LeanConst") [str "real", []]].
```

Applying `pexpr_of_mmexpr` produces the pre-expression `pow_nat (add (neg one) x) (bit0 one)`, which elaborates to the expression

```
pow_nat real real_has_pow_nat (add real real_has_add (neg real
    real_has_neg (one real real_has_one) x) (bit0 nat nat_has_add one
    nat nat_has_one) : real.
```

Formatted with standard notation and implicit arguments hidden, we have constructed the term `x : real ⊢ (x + -1)^2 : real` as desired.

3.3 Translating binding expressions

Lean's expression structure uses anonymous bound variables to implement its `pi`, `lam`, and `elet` binder constructs. Mathematica, in contrast, has no privileged notion of a binder. The Lean pre-expression `λ x, x + x` is analogous to the Mathematica expression `Function[x, x+x]`, but the underlying representation of the latter is an application of the `Function` head symbol to two arguments, the symbol `x` and the application expression `Plus[x, x]`. Structurally it is no different from `List[x, x+x]`.

To properly interpret binder expressions, both translation routines need a notion of an environment. We extend the Mathematica function `LeanForm` with another argument, a list of symbols `env` tracking binder depth. When the translation routine encounters a binding expression, it creates a new symbol, prepends it to the `env`, and translates the binder body under this extended environment; a bound variable `LeanVar[i]` is interpreted as the ith entry in `env`.

In the opposite translation direction, a translation environment is a map from strings (names of symbols) to expressions, that is, `trans_env := rb_map string expr`. When translating a Mathematica expression such as `Function[x, x+x]`, the procedure extends the environment by mapping `x` to a placeholder variable, translates the body under this extended environment, and then abstracts over the placeholder. Unlike in Lean, where `pi`, `lam`, and `elet` expressions are the only expressions that encode binders, there are many Mathematica head symbols (e.g. `Function`, `Integrate`, `Sum`) that must be translated this way.

4 Connection Interface

Mathematica includes a library called MathLink for interfacing with external C and C++ programs. The library provides tools for bidirectional communication, allowing C programmers to access Mathematica functions and vice versa, although we currently make use of only one direction. The method that uses these bindings is the only part of our link implemented in C++. This method receives a string from the Lean tactic framework, executes it in Mathematica, and returns an `mmexpr` constructed from the resulting output.

Because of the cost of launching a new Mathematica kernel, it is desirable to keep a single link open as long as possible. One "Lean server" process is used for the duration of each interactive editor session; a single Mathematica kernel persists as long as this process is active. When using Lean in "batch mode" to process files or directories, we again only launch one Mathematica kernel. To minimize side effects caused by maintaining a single kernel, each command is executed in a new context.

The translation procedure is exposed in Lean using the tactic framework via the declaration

```
meta constant mathematica.execute : string → tactic expr.
```

This tactic executes the input string in Mathematica, and returns an `expr` with type `mmexpr` representing the result of the computation. From this basic tactic, it is easy to define variants such as

```
run_command_using : (string→string) → expr → string → tactic pexpr.
```

The first argument is a Mathematica command, including a placeholder bound variable, which is replaced by the Mathematica representation of the `expr` argument. The `string` argument is the path to a file which contains auxiliary definitions, usable in the command. This variant will evaluate the resulting `expr` as an `mmexpr` and apply the back-translation `pexpr_of_mmexpr` to produce a `pexpr`.

Going back to our running example from Section 3, assuming `e` is the unfactored expression, we would call

```
run_command_on (λ s, s ++ " // LeanConvert // Activate // Factor") e
```

to produce a pre-expression representing the factored form of `e`. (Recall that the Mathematica syntax `x // f` reduces to `f[x]`.) In fact, we can define

```
meta def factor (e : expr) : tactic pexpr :=
run_command_on (λ s, s ++ " // LeanConvert // Activate // Factor") e,
```

or a variant that elaborates the result into an `expr` with the same type as `e`.

5 Verification of results

So far we have described how to embed a Lean expression in Mathematica, manipulate it, and import the result back into Lean. At this point, the imported result is simply a new expression: no connection has been established between the original and the result. In our factoring example, we expect the two expressions to be equal; if we were computing an antiderivative, we would expect the derivative of the result to be equal to the original. More complex return types can lead to more complex relations. For example, an algorithm using Mathematica's linear arithmetic tools to verify the unsatisfiability of a system of equations may return a certificate that must be converted into a proof of falsity.

Credulous users may simply decide to trust the translation and CAS computation, and assert without proof that the result has an expected property. An example using this approach is given at the end of this section. Of course, the level of trust needed to do this is unacceptably high for many situations. We are often interested in performing *certifiable* calculations in Mathematica, and using this certificate to construct proofs in Lean.

It would be hopeless to expect one tool to verify all results. Rather, for each common computation, we will have a tactic script to (attempt to) prove the appropriate relation between input and output. "Uncommon" or one-off computations can be verified in-line by the user. This method of separating search (or computation) and verification is discussed at length by Harrison and Théry [13], and by many others. It turns out that a surprising number of algorithms are able to generate certificates to this end.

The tactics used in this section, along with more examples, are available in the supplementary information to this paper. These examples are not meant to be exhaustive, but rather to illustrate the ease with which Mathematica can be accessed; with the possible exception of the linear arithmetic tactic, each is fairly simple to implement. The Lean library is still under development, and some types and functions used here are in fact axiomatized constants, but the implementation is not relevant to the behavior of our link.

5.1 Factoring

In our running example, we have used Mathematica to construct the Lean expression `(x + -1)^2 : real`. We expect to find a proof that `x^2 - 2*x + 1 = (x + -1)^2`. This type of proof is easy to automate with Lean's simplifier:

```
meta def eq_by_simp (e1 e2 : expr) : tactic expr :=
do gl ← mk_app `eq [e1, e2],
   mk_inhabitant_using gl simp <|> fail "unable to simplify"
```

Using this machinery, we can easily write a tactic `factor` that, given a polynomial expression, factors it and adds a constant to the local context asserting equality.

```
example (x : ℝ) : x^2-2*x+1 ≥ 0 :=
by factor x^2-2*x+1 using q; rewrite q; apply sq_nonneg
```

We provide more examples of this tactic in action in the supplementary material, including one in which `x^10-y^10` factors into

```
(x + -1 * y) * (x + y) * (x^4 + -1 * x^3 * y + x^2 * y^2 + -1 * x *
    y^3 + y^4) * (x^4 + x^3 * y + x^2 * y^2 + x * y^3 + y^4).
```

In general, factoring problems are easily handled by this type of approach, since the results serve as their own certificates. Factoring integers is a simple example of this (to verify, simply multiply out the prime factors); dually, primality certificates can be checked as in Pratt [17].

Factoring matrices is slightly more complex. Mathematica implements a number of common matrix decomposition methods, whose computation can be verified in Lean by re-multiplying the factors. We can use these tools to, e.g., define a tactic `lu_decomp` which computes and verifies the LU decomposition of a matrix.

```
example : ∃ l u, is_lower_triangular l ∧ is_upper_triangular u
        ∧ l ** u = [[1, 2, 3], [1, 4, 9], [1, 8, 27]] := by lu_decomp
```

5.2 Solving polynomials

Mathematica implements numerous decision procedures and heuristics for solving systems of equations. Many of these are bundled into its `Solve` function. Over some domains, it is possible to verify solutions in Lean using the simplifier, arithmetic normalizer, or other automation. Lean's `norm_num` tactic, which reduces arithmetic comparisons between numerals, is well-suited to verifying solutions to systems of polynomial equations. The tactic `solve_polys` uses `Solve` and `norm_num` to prove theorems such as

```
example :  ∃ x y : ℝ, 99/20*y^2 - x^2*y + x*y = 0
  ∧ 2*y^3 - 2*x^2*y^2 - 2*x^3 + 6381/4 = 0 := by solve_polys.
```

Users familiar with Mathematica may recall that `Solve` outputs a list of lists of applications of the `Rule` symbol, each mapping each variable to a value. A `Rule` has no close correspondent in Lean, and it would involve some contortion to translate this output and extract a single solution in the proof assistant. However, it is easy to perform this transformation within Mathematica, and processing the result of `Solve` *before* transporting it back to Lean makes the procedure much simpler to implement. This type of consideration appears often: some transformations are more easily achieved in one system or the other.

5.3 Linear arithmetic

Many proof assistants provide tools for automatically proving linear arithmetic goals, or equivalently for proving the unsatisfiability of a set of linear hypotheses. There are various techniques for doing this, including building proof terms incrementally using Fourier–Motzkin elimination [19]. Alternatively, linear programming can be used to generate certificates of unsatisfiability. In this setting, a certificate for the unsatisfiability of $\{p_i(\bar{x}) \leq 0 : 0 \leq i \leq n\}$ is a list of rational coefficients $\{c_i : 0 \leq i \leq n\}$ such that $\sum_{0 \leq i \leq n} c_i \cdot p_i = q > 0$ for some constant polynomial q; equivalently, this list serves as a witness for Farkas' lemma [18].

Given a set of hypotheses in Lean that express linear inequalities, we can prove their unsatisfiability by generating a list of such coefficients (in Mathematica), automatically proving (in Lean) that these coefficients have the necessary properties, and applying a verified proof of Farkas' lemma.

While passing a list of inequalities to Mathematica may seem different than passing an expression such as `x^2 - 2*x + 1`, we are able to use the same translation procedure. The expression `x + 1 ≤ 2*y` has type `Prop`, which is to say it is a type living in the lowest universe level `Sort 0`. A term of this type is a proof of the claim `x + 1 ≤ 2*y`. In our factorization example, we translated a term of type `real`, whereas here we translate the *type* of a hypothesis. But in dependent type theory, types are terms themselves, and we are able to represent any term in Mathematica. In Lean we define

```
le {u} : Π {A : Type u} [has_le A], A → A → Prop.
```

We reduce this in Mathematica using the rule

```
LeanForm[LeanApp[LeanApp[LeanApp[LeanApp[LeanConst["le",
  _], _], _], x_], y_]] = Inactive[LessEqual][x, y]
```

and define similar rules for $<$, $\geq$, $>$, and $=$.

Once the hypotheses have been translated to Mathematica, we must set up and solve the appropriate linear program. (Note that we are not trying to solve the hypotheses as given, but rather to find a certificate of their unsatisfiability.) A program provided in the supplementary materials to this paper shows how to use the Mathematica function `FindInstance` to produce the desired list of rational coefficients. This list is translated back to Lean, where it can be elaborated with type `list rat`. Once this list is confirmed to meet the requirements of Farkas' lemma, the lemma is applied to produce a proof of false.

```
example (x y : ℝ) (h1 : 2*x + 4*y ≤ 4) (h2 : -x ≤ 1)
        (h3 : -y ≤ -5) : false :=
by not_exists_of_linear_hyps h1 h2 h3
```

5.4 Sanity checking

Even non-certifiable computations can sometimes be useful for proof assistant users. The `FindInstance` function, for instance, can be used to check that a goal is in fact provable. We define a tactic `sanity_check`, which fails if Mathematica is able to find a variable assignment that satisfies the local hypotheses and the negation of the current goal. The first example below fails when Mathematica decides that the goal does not follow; the second succeeds.

```
example (x : ℝ) (h1 : sin x = 0) (h2 : cos x > 0) : x = 0 :=
by sanity_check; admit

example (x : ℝ) (h1 : sin x = 0) (h2 : cos x > 0)
             (h3 : -pi < x ∧ x < pi) : x = 0 :=
by sanity_check; admit
```

5.5 Axiomatized computations

Since it is possible to declare axioms from within the Lean tactic framework, we can axiomatize the results of Mathematica computations dynamically. This allows us to access a wealth of information within Mathematica, at least when we are not concerned about complete verification. One interesting application is to query Mathematica for special function identities. While these identities may be difficult to formally prove, trusting Mathematica allows us to find some middle ground. The `mk_bessel_eq` tactic uses Mathematica's `FullSimplify` function to reduce the equation on the left:

```
example : ∀ x, x*BesselJ 2 x + x*BesselJ 0 x = 2*BesselJ 1 x :=
by mk_bessel_eq
```

We can also define a tactic that uses Mathematica to obtain numerical approximations of constants, and axiomatizes bounds on their accuracy:

```
approx (100 * BesselJ 2 (13 / 25)) (0.001 : ℝ)
```

declares an axiom stating that

```
75977 / 23000 < 100 * BesselJ 2 (13 / 25) < 76023 / 23000.
```

6 Concluding thoughts

6.1 Related work

The following discussion is not meant to be comprehensive, but rather to indicate the many ways in which one can approach connecting ITP and computer algebra.

Harrison and Théry [13] describe a "skeptical" link between HOL and Maple that follows a similar approach to our bridge. Computation is done in a standard, standalone version of the CAS, and sent to the proof assistant for certification. The running examples used are factorization of polynomials and antiderivation. The discussion is accompanied by an illuminating comparison between proof search and proof checking, and the relation to the class NP.

Ballarin and Paulson [3] provide a connection between Isabelle and the computer algebra library Σ^{IT} [5] that is more trusting than the previous approach. They distinguish between sound and unsound algorithms in computer algebra: roughly, a sound algorithm is one whose correctness is provable, while an unsound algorithm may make unreasonable assumptions about the input data. Their link accepts sound algorithms in the CAS as oracles. A similarly trustful link between Isabelle and Maple, by Ballarin, Homann, and Calmet [2] allows the Isabelle user to introduce equalities derived in the CAS as rewrite rules.

A related, more skeptical, approach is to formally verify CAS algorithms and incorporate them into a proof assistant via reflection. This approach is taken by Dénès, Mörtberg, and Siles [12], whose CoqEAL library implements a number of algorithms in Coq.

Kerber, Kohlhase, and Sorge [15] describe how computer algebra can be used in proof assistants for the purpose of proof planning. They implement a minimal

CAS, which is able to produce high-level sketch information. This sketch can be processed into a proof plan, which can be further expanded into a detailed proof.

Alternatively, one can build a CAS inside a proof assistant without reflection, such that proof terms are carried through the computation. Kaliszyk and Wiedijk [14] implement such a system in HOL Light, exhibiting techniques for simplification, numeric approximation, and antiderivation.

Going in the opposite direction, CAS users may want to access ATP or ITP systems. One example of a link in this direction is Adams et al [1], who use PVS to verify side conditions generated in computations in Maple. Systems such as Analytica [4] and Theorema [6] provide ATP- or ITP-style behavior from within Mathematica. Axiom [9] and its related projects provide a type system for computer algebra, which is claimed to be "almost" strong enough to make use of the Curry–Howard isomorphism.

6.2 Future work

We have described a master–slave relationship between Lean and Mathematica respectively. The Wolfram Language is able to express "propositions," and has some capacity for evaluating such propositions, but has no notion of proof. Reversing the relationship, so that a Mathematica user could use Lean to (automatically or interactively) verify propositions, is a promising direction for future work. The tools described in this paper, particularly the Mathematica-to-Lean translation and the physical MathLink connection, are both necessary building blocks for such a project. This approach has some caveats. Since Mathematica is untyped, naive translations may be ambiguous; since there is not a well-specified logic underlying the CAS, one must worry about inconsistencies between its built-in assumptions and the logical rules of the proof assistant. There is no analogue to the ad hoc verification that we use in the original direction. Nonetheless, it is an interesting direction to explore.

There is much room room for an improved interface under the current ITP–CAS relationship. We imagine a link integrated with Lean's supported editors, where the user effectively has access to the Mathematica REPL augmented by the current Lean environment.

Finally, Lean connects to Mathematica using MathLink, which only supports one connection at a time; thus we cannot process calls to Mathematica in parallel. Both systems support parallel computation, but integrating the two is a substantial engineering task.

Acknowledgments. Many thanks to Jeremy Avigad, Jasmin Blanchette, Leonardo de Moura, Ian Ford, Johannes Hölzl, José Martín-García, and Michael Trott.

References

1. A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In

Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, TPHOLs '01, pages 27–42, London, UK, UK, 2001. Springer-Verlag.
2. C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation*, ISSAC '95, pages 150–157, New York, NY, USA, 1995. ACM.
3. C. Ballarin and L. C. Paulson. A pragmatic approach to extending provers by computer algebra. *Fund. Inform.*, 39(1-2):1–20, 1999. Symbolic computation and related topics in artificial intelligence (Plattsburg, NY, 1998).
4. A. Bauer, E. Clarke, and X. Zhao. Analytica – an experiment in combining theorem proving and symbolic computation. *Journ. Autom. Reas.*, 21(3):295–325, 1998.
5. M. Bronstein. σit—a strongly-typed embeddable computer algebra library. In *International Symposium on Design and Implementation of Symbolic Computation Systems*, pages 22–33. Springer, 1996.
6. B. Buchberger, T. Jebelean, T. Kutsia, A. Maletzky, and W. Windsteiger. Theorema 2.0: Computer-assisted natural-style mathematics. *Journal of Formalized Reasoning*, 9(1):149–185, 2016.
7. T. Coquand and G. Huet. The Calculus of Constructions. *Inform. and Comput.*, 76(2-3):95–120, 1988.
8. T. Coquand and C. Paulin. Inductively defined types. In *COLOG-88 (Tallinn, 1988)*, volume 417 of *Lec. Notes in Comp. Sci.*, pages 50–66. Springer, Berlin, 1990.
9. T. Daly. *Axiom: The 30 year horizon.* Lulu Incorporated, 2005.
10. L. de Moura, G. Ebner, J. Roesch, and S. Ullrich. The Lean theorem prover (presentation), January 2017.
11. L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover. *http://leanprover.github.io/files/system.pdf*, 2014.
12. M. Dénès, A. Mörtberg, and V. Siles. A refinement-based approach to computational algebra in Coq. In *Interactive theorem proving*, volume 7406 of *Lecture Notes in Comput. Sci.*, pages 83–98. Springer, Heidelberg, 2012.
13. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *J. Automat. Reason.*, 21(3):279–294, 1998.
14. C. Kaliszyk and F. Wiedijk. *Certified Computer Algebra on Top of an Interactive Theorem Prover*, pages 94–105. Springer, Berlin, Heidelberg, 2007.
15. M. Kerber, M. Kohlhase, and V. Sorge. Integrating computer algebra into proof planning. *J. Automat. Reason.*, 21(3):327–355, 1998.
16. C. McBride and J. McKinna. Functional pearl: I am not a number–I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 1–9, New York, NY, USA, 2004. ACM.
17. V. R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.
18. A. Schrijver. *Theory of linear and integer programming.* Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons Ltd., Chichester, 1986. A Wiley-Interscience Publication.
19. H. Williams. Fourier's method of linear programming and its dual. *The American Mathematical Monthly*, 93(9):681–695, 1986.
20. S. Wolfram. *An Elementary Introduction to the Wolfram Language.* Wolfram Media, Incorporated, 2015.