A bi-directional extensible ad hoc interface between Lean and Mathematica*

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

Minchao Wu
Carnegie Mellon University
Pittsburgh, PA, USA
minchaowu@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. We also establish a connection in the opposite direction, using Mathematica to import and process proof terms from Lean.

Keywords ITP, CAS, proof translation

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 often make sense in the context of computer algebra; the capability and flexibility 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 familiar algorithms discourage many mathematicians from using proof assistants. Conversely, the unreliability of many computer algebra systems, and their lack of proof languages and proof libraries, often makes them unsuitable for mathematical justification.

Integrating computer algebra and proof assistants is one way to reduce this barrier to entry to ITP and to strengthen the justificatory power of computer algebra. 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 [14] and the computer algebra system Mathematica [29]. 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 connection is inspired by the architecture described by Harrison and Théry [19]. A number of features of our bridge distinguish it from earlier work. CAS results imported into the proof assistant can be trusted, verified, or used ephemerally; the translation can be extended in-line with library development, without modifying auxiliary dictionaries or source code; the link works bi-directionally using the same translation procedure, allowing Mathematica to access Lean's library and automation.

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. One can then 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. Alternatively, we can trust the CAS as an oracle, or use it in a purely informative role, where its output does not appear in the final proof term. We provide comprehensive tactics to perform and verify certain classes of computations, such as factoring polynomials and matrices. But all the components of our procedure are implemented transparently in Lean's metaprogramming framework, and they can easily be extended or used for one-off computations from within the Lean environment.

This range of possibilities is intended to make our link attractive to multiple audiences. The working mathematician or mathematics student, 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

*A previous version of this paper was presented at the PxTP 2017 workshop in Brasília, Brazil [22]. The earlier version described only one direction of the link, and was written only by the first author.

Conference'17, July 2017, Washington, DC, USA
2018.

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 ease of metaprogramming in Lean is another draw: users do not need to learn a new programming or tactic language to write complicated translation rules or verification procedures.

The translation procedure used is symmetric and can be used for communication in the reverse direction as well. Mathematica has no built-in notion of proof, although it does have head symbols that express propositions. Rather than establishing an entire proof calculus for these symbols within Mathematica, we export theorem statements to Lean, where they can be verified in an environment designed for this purpose. The resulting proof terms are interpreted in the CAS and can be displayed or processed as needed. Alternatively, we can skip the verification step and display lemmas that are likely to be relevant to Mathematica's goal. In some sense, the link is allowing Mathematica to "borrow" Lean's semantics, proof language, and proof library.

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 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 the interface for running Mathematica from Lean, and shows many examples of the link in action. Section 5 explains the reverse direction. We conclude with a discussion of related and future work.

2 System descriptions

2.1 Lean

Lean is a proof assistant being developed at Microsoft Research [14]. 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 (CIC) [11, 12], an extension of the lambda-calculus with dependent types and inductive definitions. There is a non-cumulative hierarchy of type universes `Sort u`, `u ≥ 0`, with the abbreviations `Prop = Sort 0` and `Type u = Sort (u+1)`. The bottom level `Prop` is impredicative and proof-irrelevant.

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}` signals 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]` signals that a term of that type is to be inferred using type class resolution. The universe argument `u` indicates that `add` is parametric over one universe level.

The dependently typed language implemented in Lean is flexible enough to serve as its own *metaprogramming language* [17]. Data types and procedures implemented in Lean's 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 evaluated 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 `pexpr` and `expr`, which expose the syntax of Lean (pre-)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 solved. 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.

2.2 Mathematica

Mathematica is a popular symbolic computation system developed at Wolfram Research, implementing the Wolfram Language [29]. Along with support for 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 the syntax of the Wolfram Language, 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]`. 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*. In general, non-numeric atoms like `x` and `y` are called *symbols*.
- There is no strong distinction between defined and undefined symbols. The user is free to introduce a new symbol and use it at will. The computational behavior of a head symbol can be fully or partially defined via pattern matching rules, such as `F[x_, y_] := x+y`; the underscores indicate that `x_` and `y_` are patterns.
- 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.

3 The translation procedure

Our bridge can be 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 intended 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.

As a running 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.

In Subsection 3.1 we describe steps 1, 2, and 4. Once we have a valid Mathematica expression, step 3 is trivial. We discuss steps 5 and 6 in Section 4; since checking that a polynomial has been factored correctly is much easier than factoring it in the first place, these are handled easily by simplification and rewriting.

It is worth emphasizing the modularity and extensibility of this approach. Both directions of translation are handled independently, and the translation rules can be extended or changed at will. Translation rules may be arbitrarily complex. Users may choose to use alternate verification procedures, or to forgo the verification step entirely.

3.1 Translating Lean to Mathematica

The Lean expression grammar is presented (in Lean syntax) in Figure 1. The type `expr` is marked with the keyword `meta` because, during evaluation, the Lean virtual machine replaces terms of this type with the kernel's expression datatype. In the explanation, we focus on the parts of interest for our link.

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. (Nondependent 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 [23]. Application of one expression to another is represented by the `app` kind.

```
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 → list expr → expr
```

Figure 1. Lean expression kinds

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.

To represent this syntax in Mathematica, we define

```
mathematica_form_of_expr : expr → 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 that an expression reflected into and immediately back from Mathematica should translate to the original expression without having to inject 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. Instead, 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 "x" "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["x", "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 typeclass 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, allowing 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.
```

```
inductive mmexpr
| sym   : string → mmexpr
| mstr  : string → mmexpr
| mint  : int → mmexpr
| app   : mmexpr → list mmexpr → mmexpr
| mreal : float → mmexpr
```

Figure 2. Mathematica expression kinds

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]`.

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 levels and implicit arguments are omitted. It is not expected to type-check, but one can try to convert it 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.

Mathematica expressions are analogous 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 integers may be used to represent numerals in many different Lean types, expressions built with the `mint` constructor 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. Another removes instances of the symbol `Hold`, which blocks evaluation of sequences of expressions. The Lean translation of `Plus[Hold[x, y, z]]` should reduce to the translation of `Plus[x, y, z]`, but since `Hold[x, y, z]` translates to a sequence of expressions, this does not match either of the previous rule types.

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. The mechanism for extending the translation procedure is thus integrated into theory development; translation rules are first-class members of mathematical libraries, and any project importing a library will automatically have access to its translation rules.

Returning to our example, we have translated the expression `x^2 - 2x + 1` and factored the result, 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 again

```
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 *i*th 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 Querying Mathematica from Lean

4.1 Connection interface

Because of the cost of launching a new Mathematica kernel, it is undesirable to do so every time Mathematica is queried from Lean. Instead, we implement a simple server in Mathematica, which receives requests containing expressions and returns the results of evaluating these expressions. Lean communicates with this server by calling a Python client script. This short script is the only part of the link that is implemented neither in Lean nor in Mathematica.

This architecture ensures that a single Mathematica kernel will be used for as long as possible, across multiple tactic executions and possibly even multiple Lean projects. To preserve an illusion of "statelessness," each Mathematica evaluation occurs in a new context which is immediately cleared. While this avoids accidental leaks of information, it is not a watertight seal, and users who consciously wish to preserve information between sessions can do so.

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

```
meta def mathematica.execute : string → tactic
    mmexpr.
```

This tactic evaluates the input string in Mathematica, and returns a term 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 apply the back-translation `pexpr_of_mmexpr` to produce a `pexpr`.

Another variant, `execute_global : string → tactic mmexpr`, evaluates its input in Mathematica's global context.

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`.

4.2 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.

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 [19] 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 of these constants is not relevant to the behavior of our link.

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 (l r : expr) : tactic expr :=
do gl ← mk_app `eq [l, r],
   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. (The theorem `sq_nonneg` proves that the square of a real number is nonnegative.)

```
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 [24].

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
```

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 a variable to a value. Each list of rules represents

one solution. A `Rule` has no general 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.

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 [28]. 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 [26]. A slight generalization of Farkas' lemma that allows equalities and strict inequalities is known as the Motzkin transposition theorem.

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 the Motzkin transposition theorem.

While passing a list of inequalities to Mathematica may seem different from 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 <, ≥, >, 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
```

Sanity checking. Even non-certifiable computations can sometimes be useful for proof assistant users. Mathematica's `FindInstance` function, for example, 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. This tactic is similar to a very lightweight version of Isabelle's Nitpick [6]. 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
```

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 `prove_by_full_simp` tactic uses Mathematica's `FullSimplify` function to reduce the Bessel function expression on the left, and after checking that it is equal to the one on the right, adds this equality as an axiom in Lean:

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

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)
  ∧ BesselJ 2 (13 / 25) < 76023 / 23000.
```

5 Querying Lean from Mathematica

The scope of use of computer algebra in mathematics is largely limited to exploration and discovery. Finished proofs often avoid using these tools to justify claims, or even fail to mention them entirely. Outside of a few very specific domains, systems like Mathematica have no internal notion of proof or correctness. There are many documented instances of bugs and unexpected behavior in computer algebra systems [2], making concerns about this black-box nature more than just theoretical. Even the semantics for certain computations can be vague; reducing `(x^2 - 1)/(x - 1)` to `x + 1` is correct when considered as polynomial division, but computer algebra systems use this same notation to refer to a function of `x`.

Integrating a proof system into a mature CAS such as Mathematica is an enormous engineering task. A more realistic approach is to use a translation procedure to "borrow" a proof language and semantics from a proof assistant, on translatable domains. A proposition relating the input and output of a CAS evaluation can be exported to and proved in the proof assistant, which can return a proof term. This is morally similar to the ad hoc verification described in the previous section; while no general guarantee is claimed, individual computations can be checked.

5.1 Connection interface

We establish a connection to Lean from Mathematica using Mathematica's external command interface `RunProcess`. It is not necessary to replicate the server-client architecture of Section 4.1, since the cost of launching Lean is low. Lean already implements a server for communication with its editors, and using this interface would be particularly useful for applications requiring a persistent, changing environment. However, this does not apply to the applications described here, so we have opted for the simpler approach.

We implement a function

```
ProveUsingLeanTactic[e_, tac_String]
```

which takes an arbitrary Mathematica expression `e`. It exports this expression to Lean and interprets it as a proposition. If successful, it attempts to prove the proposition using the tactic script `tac`, and returns the resulting proof term to Mathematica. While the tactic script may be arbitrarily complex, it will often be just a proof by `simp`, by `eblast`, or by other general-purpose automation.

A similar function, `ProveInteractively`, opens an interactive Lean session with the translation of `e` as the goal. When the session is terminated, the proof is checked and returned to Mathematica.

More generally, the function

```
RunLeanCommand[e_, cmd_String]
```

will evaluate a given Lean command on an expression `e` and return the result if successful. This is useful when the desired output is not a proof term.

5.2 Applications

Interpreting propositional proofs. Mathematica's built-in `TautologyQ` and `FullSimplify` functions serve as complete SAT solvers. However, both are black boxes: they produce no certificate or justification. Indeed, the system has no established proof language for propositional logic. On the other hand, Lean comes equipped with a number of proof-producing decision procedures for this domain. (For this example, we use `intuit`, as it produces proofs containing few extra constants.)

We define a minimal propositional proof calculus in Mathematica that mirrors the calculus in Lean. That is, we introduce head symbols `AndIntro`, `OrIntroLeft`, `FalseElim`, etc., and add `LeanForm` translation rules that map Lean's `and.intro`, `or.inl`, `false.elim`, etc. to their corresponding symbols. We can then state a propositional theorem in Mathematica, prove it in Lean, and interpret the resulting proof term in our calculus. While it would certainly be possible to implement the Lean proof search procedure in Mathematica directly, this approach ensures that the proof is correct, as it has been checked by Lean.

The resulting Mathematica proof object can be computed with in any number of ways. We implement a function which displays the proof as a natural deduction diagram, as in Figure 3. There is no fundamental reason why this approach cannot be extended to richer logics, such as first-order logic; the difficulties lie in representing a calculus for these logics in Mathematica, and generating proofs in Lean that can be translated to such a calculus. (Many proof tools in Lean use higher-order constructs that may be difficult to directly translate.)

Displaying significant proof steps. Interpreting arbitrary proofs in Mathematica may be too much to ask for, as the target language and translation rules may become arbitrarily complicated. An easier task is to interpret lemmas or relevant steps used to produce a proof. Lean's simplifier, heuristic quantifier instantiation procedure, and other general-purpose proof tactics search for lemmas in the Lean library to solve a goal. It is possible to inspect the proof terms generated by these tactics and extract theory lemmas, or in some cases, to implement versions of these tactics that produce a list of lemmas used. The types of the instances of these lemmas appearing in a proof term can be interpreted in Mathematica and displayed. Finding all and only the "interesting" lemmas is a difficult and poorly specified problem, but it is reasonable to implement a first-pass heuristic.

As an example, we do so in the context of set normalization. Mathematica has no built-in handling for arbitrary sets, but proofs of propositions such as $A \cap (B \cup \bar{A}) = A \cap B$ are easily found with Lean's simplifier. Noting that the relevant lemmas used by `simp` state that `A ∩ (B ∪ -A) = (A ∩ B) ∪ (A ∩ -A)`, `A ∩ -A = ∅`, and `(A ∩ B) ∪ ∅ = A ∩ B`, we can return these lemmas to Mathematica and display them as a "proof

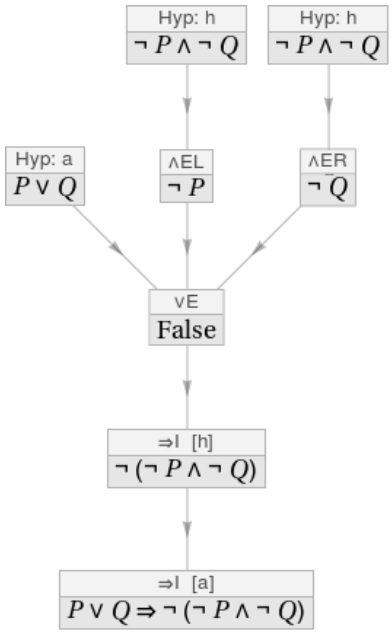

Figure 3. A natural deduction diagram generated from a Lean proof term

sketch." Note that there is no need to add translation rules for these lemmas themselves; alignments between the constants for union, intersection, complement, and equality are enough. This limits the need for a long list of translations and makes the procedure relatively robust to the introduction of new simplifier rules.

A similar application involves the use of a relevance filtering algorithm. Given a target expression, such an algorithm will return a list of declarations in the environment that, heuristically, may be useful to prove the target. Both symbolic and probabilistic relevance filters have been implemented in other systems, and are used for lemma selection for tools such as Isabelle's Sledgehammer [7]. We have implemented a rudimentary relevance filter in Lean. Using this tool, one can state a conjecture in Mathematica and receive a list of facts that may be of use to prove it, without depending on automation in Lean to actually find a proof.

A soft typing system for Mathematica. The Wolfram Language is untyped, and its notion of a well-formed expression is very permissive. Unexpected evaluation, and the lack of expected evaluation, due to "ill-typed" expressions are common issues for Mathematica users, and can be difficult to diagnose. There is interest and preliminary work in establishing a soft typing system on some subset of Mathematica functions, particularly the mathematically oriented functions. Such a system would give a better notion of meaningful Mathematica expressions; by exploiting the Curry–Howard correspondence, a sufficiently strong type system could also provide an internal proof language.

In order to be usable in practice, such a type system requires algorithms for type checking and elaboration. If we construct a type system in Mathematica that is compatible with Lean's, we can avoid reimplementing these components by using those in the ITP. Indeed, a prototype presented at the 2016 Wolfram Technology Conference [10] implements a version of the calculus of constructions in Mathematica, along with a rudimentary type checker. We establish translation rules between this implementation and Lean, which allow us to elaborate Mathematica "pre-expressions" in the proof assistant and type-check the resulting terms in either system. The soft typing system is still at an early stage, and will certainly change in the future, but by maintaining this connection, we will reduce the number of auxiliary tools needed to make the type system useful.

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 [19] 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 of proof search to proof checking, and the relation to the class NP. Delahaye and Mayero [15] provide a similar link between Coq and Maple, specialized to proving field identities. Both projects tackle only the scenario in which the proof assistant drives the CAS.

Ballarin and Paulson [4] provide a connection between Isabelle and the computer algebra library Σ^{IT} [8] 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 [3], allows the Isabelle user to introduce equalities derived in the CAS as rewrite rules. A third example by Seddiki, Dunchev, Khan-Afshar, and Tahar [27] connects HOL Light to Mathematica via OpenMath, introducing results from the CAS as HOL axioms.

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 [16], whose CoqEAL library implements a number of algorithms in Coq.

Kerber, Kohlhase, and Sorge [21] 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 [20] 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; Gottliebsen, Kelsey, and Martin [18] make use of similar ideas. Systems such as Analytica [5] and Theorema [9] provide ATP- or ITP-style behavior from within Mathematica. Axiom [13] 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 correspondence.

6.2 Future work

There is much 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 read-evaluate-print loop augmented by the current Lean environment. The REPL is a standard way of interacting with computer algebra systems, and contributes to their utility in exploration and discovery. Allowing this kind of interaction within Lean would greatly advance the goals of this project.

The server interface described in Section 4.1 only supports sequential evaluation of Mathematica commands. Both systems support parallel computation, and integrating the two could increase the utility of this link for large projects. Similarly, the physical connection between Mathematica and Lean can be strengthened by communicating with a Lean server. This would avoid the (small) cost of launching many instances of Lean, and would allow the possibility of maintaining state.

With the exception of the server running in Mathematica, the components of this link are generally adaptable to other computer algebra systems. More broadly, we see this project as part of a general trend. The various computer-based tools used in mathematical research, by and large, are independent of each other. It requires quite a lot of copying, pasting, and translating to – for example – compute an expression in Magma, verify its side conditions in Z3, visualize results in Mathematica, and export relevant formulas to LaTeX. Unified frameworks have been proposed and implemented [25], but are not widely used. Because they provide a strict logical foundation, precise semantics, and possibility of verification, proof assistants are strong candidates to center translation networks between systems.

Acknowledgments

Many thanks to Jeremy Avigad, Jasmin Blanchette, Leonardo de Moura, Ian Ford, Johannes Hölzl, José Martín-García, James Mulnix, Michael Trott, and the Lean working group at CMU.

References

[1] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. 2001. 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)*. Springer-Verlag, London, UK, UK, 27–42. http://dl.acm.org/citation.cfm?id=646528.695189

[2] David H Bailey, Jonathan M Borwein, Vishaal Kapoor, and Eric W Weisstein. 2006. Ten problems in experimental mathematics. *Amer. Math. Monthly* 113, 6 (2006), 481–509.

[3] Clemens Ballarin, Karsten Homann, and Jacques Calmet. 1995. Theorems and Algorithms: An Interface Between Isabelle and Maple. In *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*. ACM, New York, NY, USA, 150–157. https://doi.org/10.1145/220346.220366

[4] Clemens Ballarin and Lawrence C. Paulson. 1999. A pragmatic approach to extending provers by computer algebra. *Fund. Inform.* 39, 1-2 (1999), 1–20. Symbolic computation and related topics in artificial intelligence (Plattsburg, NY, 1998).

[5] Andrej Bauer, Edmund Clarke, and Xudong Zhao. 1998. Analytica – An Experiment in Combining Theorem Proving and Symbolic Computation. *Journ. Autom. Reas.* 21, 3 (1998), 295–325. https://doi.org/10.1023/A:1006079212546

[6] Jasmin Blanchette and Tobias Nipkow. 2010. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *Interactive Theorem Proving* (2010), 131–146.

[7] Jasmin C Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. 2016. Hammering towards QED. *Journal of Formalized Reasoning* 9, 1 (2016), 101–148.

[8] Manuel Bronstein. 1996. ΣIT—A strongly-typed embeddable computer algebra library. In *International Symposium on Design and Implementation of Symbolic Computation Systems*. Springer, 22–33.

[9] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, and Wolfgang Windsteiger. 2016. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *Journal of Formalized Reasoning* 9, 1 (2016), 149–185. https://doi.org/10.6092/issn.1972-5787/4568

[10] Lin Cong, Yihe Dong, Ian Ford, Robert Y. Lewis, José Martín-García, and James Mulnix. 2016. Progress in Pure Mathematics. Wolfram Technology Conference.

[11] Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Inform. and Comput.* 76, 2-3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

[12] Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88 (Tallinn, 1988)*. Lec. Notes in Comp. Sci., Vol. 417. Springer, Berlin, 50–66. https://doi.org/10.1007/3-540-52335-9_47

[13] Timothy Daly. 2005. *Axiom: The 30 year horizon*. Lulu Incorporated.

[14] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2014. The Lean Theorem Prover. *http://leanprover.github.io/files/system.pdf* (2014).

[15] David Delahaye and Micaela Mayero. 2005. Dealing with algebraic expressions over a field in Coq using Maple. *Journal of Symbolic Computation* 39, 5 (2005), 569 – 592. https://doi.org/10.1016/j.jsc.2004.12.004 Automated Reasoning and Computer Algebra Systems (AR-CA).

[16] Maxime Dénès, Anders Mörtberg, and Vincent Siles. 2012. A refinement-based approach to computational algebra in Coq. In *Interactive theorem proving*. Lecture Notes in Comput. Sci., Vol. 7406. Springer, Heidelberg, 83–98. https://doi.org/10.1007/978-3-642-32347-8_7

[17] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 34.

[18] Hanne Gottliebsen, Tom Kelsey, and Ursula Martin. 2005. Hidden verification for computational mathematics. *Journal of Symbolic Computation* 39, 5 (2005), 539 – 567. https://doi.org/10.1016/j.jsc.2004.12.005 Automated Reasoning and Computer Algebra Systems (AR-CA).

[19] John Harrison and L. Théry. 1998. A skeptic's approach to combining HOL and Maple. *J. Automat. Reason.* 21, 3 (1998), 279–294. https://doi.org/10.1023/A:1006023127567

[20] Cezary Kaliszyk and Freek Wiedijk. 2007. Certified Computer Algebra on Top of an Interactive Theorem Prover. In *Proceedings of the 14th Symposium on Towards Mechanized Mathematical Assistants: 6th International Conference (Calculemus '07 / MKM '07)*. Springer-Verlag, Berlin, Heidelberg, 94–105. https://doi.org/10.1007/978-3-540-73086-6_8

[21] Manfred Kerber, Michael Kohlhase, and Volker Sorge. 1998. Integrating computer algebra into proof planning. *J. Automat. Reason.* 21, 3 (1998), 327–355. https://doi.org/10.1023/A:1006059810729

[22] Robert Y. Lewis. 2017. An extensible ad hoc interface between Lean and Mathematica.. In *Proof eXchange for Theorem Proving*.

[23] Conor McBride and James McKinna. 2004. Functional Pearl: I Am Not a Number–I Am a Free Variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/1017472.1017477

[24] Vaughan R Pratt. 1975. Every prime has a succinct certificate. *SIAM J. Comput.* 4, 3 (1975), 214–220.

[25] Florian Rabe. 2013. The MMT API: a generic MKM system. In *International Conference on Intelligent Computer Mathematics*. Springer, 339–343.

[26] Alexander Schrijver. 1986. *Theory of linear and integer programming*. John Wiley & Sons Ltd., Chichester. xii+471 pages. A Wiley-Interscience Publication.

[27] Ons Seddiki, Cvetan Dunchev, Sanaz Khan-Afshar, and Sofiène Tahar. 2015. *Enabling Symbolic and Numerical Computations in HOL Light*. Springer International Publishing, Cham, 353–358. https://doi.org/10.1007/978-3-319-20615-8_27

[28] H.P. Williams. 1986. Fourier's Method of Linear Programming and its Dual. *The American Mathematical Monthly* 93, 9 (1986), 681–695.

[29] Stephen Wolfram. 2015. *An Elementary Introduction to the Wolfram Language*. Wolfram Media, Incorporated. https://books.google.com/books?id=eflvjgEACAAJ