Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

A bi-directional extensible interface between Lean and Mathematica *

Robert Y. Lewis · Minchao Wu

Received: date / Accepted: date

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 use Mathematica as an untrusted oracle to guide proof search in the proof assistant and interact with a Mathematica notebook from within a Lean session. In the other direction, we import and process Lean declarations from within Mathematica. The proof assistant library serves as a database of mathematical knowledge that the CAS can display and explore.

Keywords Proof assistant · Formalization · Computer algebra

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

* This paper expands on a workshop paper by the first author [26], which describes an early version of one direction of the interface.

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

Robert Y. Lewis
Vrije Universiteit Amsterdam
E-mail: r.y.lewis@vu.nl

Minchao Wu
Australian National University
E-mail: logic.mcwu@gmail.com

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 [30] and the computer algebra system Mathematica [38]. 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 [23].

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 to guide proof development. The translation can be extended in-line with library development without modifying auxiliary dictionaries or source code. We are able to simulate a Mathematica read–eval–print loop (REPL) in a Lean editor session, which takes as input Mathematica expressions with arbitrary Lean expressions inserted, evaluates these in Mathematica, and displays the result. The link works bi-directionally using the same translation procedure, allowing Mathematica to access Lean's library and automation. The link requires no plugins or modified source code: it is available as a standard Lean library.

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 leading 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 computer algebra systems, and the rigor of Lean with Mathematica as an oracle falls somewhere in between. 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 allows Mathematica to "borrow" Lean's semantics, proof language, and proof library.

The source for this project, and supplementary documentation, is available online.[1] In this paper, we use `Computer Modern` for Lean code and `TeX Gyre Cursor` for Mathematica code.

2 System descriptions

2.1 Lean

Lean is a proof assistant developed at Microsoft Research [30]. It is based on the Calculus of Inductive Constructions (CIC) [13,14], 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 abbreviations `Prop = Sort 0` and `Type u = Sort (u+1)`. The bottom level `Prop` is impredicative and proof-irrelevant. The Lean community develops mathlib, a rapidly growing library of verified mathematics, programming, and tactics; more details on Lean and its library can be found in the mathlib system description [28].

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, consider the signature of the addition operator:

```
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* [19]. Data types and procedures implemented in Lean's C++ code base are exposed as constants, using the keyword

[1] https://robertylewis.com/leanmm/

`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 implementations. Meta-definitions permit unbounded recursion but are otherwise quite similar to standard definitions.

Combined with the declaration of the types `expr` and `pexpr`, which expose the syntax of elaborated and unelaborated 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 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 [38]. 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. The Wolfram language is a symbolic functional language with a simple grammar, making it a good candidate for intertranslation with Lean without having to represent low-level data structures.

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, one can also write `Plus[x, y, z]`. Common notation like `x + y + z` is also supported.
- Alternatively, one 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*. Non-numeric atoms like `Plus`, `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. For example, `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 exchange information between Mathematica and Lean. 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:

```
x : real ⊢ x^2 - 2x + 1 ≥ 0
```

Factoring the left-hand side of the inequality makes this a one-step proof (assuming we've proven 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 as 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 Section 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.

```
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
```

Fig. 1: The Lean expression grammar is captured by the type `expr`. Every Lean expression is uniquely expressed using one of these constructors.

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 below, we focus on the parts of interest for our link. In particular we will not discuss the `macro` constructor.

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 [29]. 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.

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" []),
```

we can write the full form of $x + x$:

```
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 type-class instance in the Lean term. These arguments are irrelevant to Mathematica and can be inferred

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

Fig. 2: The Mathematica expression grammar is captured by the type `mmexpr`. Every Mathematica expression is uniquely expressed using one of these constructors.

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
```

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:

```
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 be interpreted 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 our attention on the latter.

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, since 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. For instance, 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 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`. The `rb_map` type implements such a map as a red–black tree. 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

The translation described in Section 3 is bidirectional. Syntax from either system can be embedded and interpreted in the other. In this section, we describe the interface used for querying Mathematica from a Lean session, along with a number of examples of how this interface is used.

4.1 Connection interface

Because of the cost of launching a new Mathematica kernel, it is undesirable to do so every time Lean makes a query. 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 via its command line IO interface. 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

```
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, e.g.:

```
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 Verified interaction

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 input. More complex return types can lead to more complex relations. For example, an algorithm using Mathematica's linear programming 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 in Section 4.3. 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 the certificates 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 that attempts 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 [23] 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: each is fairly simple to implement.

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 ring normalization tactic:

```
meta def eq_by_ring (l r : expr) : tactic expr :=
do gl ← mk_app `eq [l, r],
   mk_inhabitant_using gl ring
     <|> 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 `pow_two_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 pow_two_nonneg
```

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

```
(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 generated and checked [32].

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 normalization, 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 such theorems:

```
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 approaches to this, including building proof terms incrementally using Fourier–Motzkin elimination [37]. Alternatively, linear programming techniques can be used to generate certificates of unsatisfiability. A certificate for the unsatisfiability of $\{p_i(\bar{x}) \leq 0 : 0 \leq i \leq n\}$ is a solution to the dual program, that 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 [34].

The tactic `linarith` implemented in Lean's mathlib is in effect a generic certificate checker for linear arithmetic. Given a function that implements a simple certificate-finding interface, `linarith` will preprocess the context, retrieve a certificate, and convert the certificate into a proof of the goal. Unlike Coq's similar `lra` [6], which reflexively calls a proven-correct certificate checker, `linarith` constructs a proof term from the certificate using ring normalization.

By default, `linarith` uses an unverified Fourier–Motzkin solver implemented in Lean to produce certificates. But this solver is a black box to the rest of the tactic. It can be desirable on large problems to use a more efficient algorithm, for instance one based on the simplex method, and `linarith` provides a configuration option to change the search function.

We provide an alternate certificate oracle that uses Mathematica's linear programming functionality. It takes only 20 lines of Lean code to implement a full drop-in replacement for the Fourier–Motzkin module. Most of this is string processing to create the appropriate call to Mathematica; the certificate search is summarized by

```
L = Part[#, 2]& /@
  FindInstance[{cstrs}&&{nngs}&&{pos}, {vars}, Rationals][[1]];
(LCM@@Denominator@L)*L
```

where `{cstrs}&&{nngs}&&{pos}` is a system of linear constraints in variables `{vars}` describing an appropriate solution to the dual problem. The final line guarantees a solution in the nonnegative integers. This function produces a (Mathematica) list of integers, which our link reinterprets in Lean as a list of natural numbers as required by the `linarith` oracle specification.

While communication overhead makes this approach slightly slower than the native Fourier–Motzkin solver on small problems, the Mathematica oracle succeeds on every `linarith` test case, e.g.:

```
example (x y z : ℚ) (h1 : 2*x  < 3*y) (h2 : -4*x + 2*z < 0)
  (h3 : 12*y - 4* z < 0) : false :=
by linarith {oracle := mm_oracle}
```

4.3 Unverified interaction

The applications of CAS to interactive proving go beyond verified computations. We emphasize the word "interactive" in the name of the field: proof assistant users may want to query the CAS for guidance as they work on a proof on their own. This requires a reasonable expectation of correctness from the CAS but no formal verification. Other more credulous users may be willing to take the CAS at its word and trust its output.

In this section we explain a number of ways that the results of Mathematica computations can be used from within Lean without verification.

Error checking. Mathematica's `FindInstance` function is a frontend for a collection of different solvers that try to find variable instantiations that satisfy a given predicate. This function can be used from Lean to check whether a proof goal is in fact provable. We define a tactic `plausibility_check` which fails if Mathematica

```
begin_mm_block

"Solve[Sin[x] == 0 && 2 < x < 4, x, Reals][[1]][[1]][[2]]";

"Factor["(x^2-2*x+1)"]";

end_mm_block
```

Fig. 3: This code block can be inserted at any top-level position in a Lean source file. It will evaluate the two Mathematica commands successively, and display the output of each in the Lean editor. The input to `Factor` is an antiquoted Lean expression, and the displayed output is also a Lean expression.

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 [7]. 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 plausibility_check; admit

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

Notebook-style interaction. The standard mode of interaction with Mathematica and other CAS tools is through a notebook interface, similar to a read–eval–print loop (REPL) with extra functionality for back references and displaying output. This style encourages using the CAS as a tool for exploration, since the user can interactively change and add to the system state and visualize the output of entries. While some proof assistants do implement REPLs for constructing proofs via tactic application or inspecting declarations in an environment, the notion of evaluation is much more limited in a proof assistant than in a CAS, limiting the ways in which one can use such an interface. Notebooks that support proof assistant languages, such as Observable[2], tend to be used only for small demonstrations.

The appeal of notebook-style interaction with a proof assistant grows with access to the evaluation and visualization tools of a CAS. We have implemented a top-level Mathematica code block command in Lean that approximates this kind of interface (Figure 3). Inside a block of Mathematica code embedded in a Lean file, the user can write arbitrary Mathematica commands, inserting antiquoted Lean expressions at any point. Upon evaluation, these antiquoted expressions are translated to Mathematica expressions. The result of the evaluation is translated back to a Lean expression and displayed in the editor infoview.

One of Mathematica's greatest strengths is its toolbox for generating plots, graphs, and images. Our code embedding makes these available from within Lean (Figure 4). Prefixing the Mathematica command with an annotation `as image` requests that the output be displayed graphically instead of textually. Users can access the full range of Mathematica's visualization tools to plot and inspect Lean terms.

[2] https://observablehq.com/@bryangingechen/hello-lean-prover

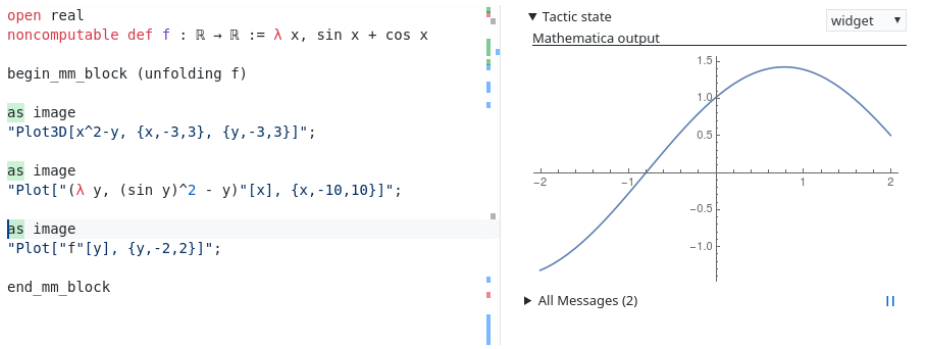

Fig. 4: This embedded Mathematica code block generates three images. The first is given by a pure Mathematica command. The second and third plot antiquoted Lean functions. The parameter `(unfolding f)` at the beginning of the block allows Mathematica to "see through" the definition of `f`. The right hand pane of the editor displays the third plot when the cursor hovers over that line.

The image display makes use of Ayers' *widget* feature for Lean[3] which allows metaprograms to display arbitrary HTML in the editor. The syntactical oddities of embedding Mathematica code reflect limitations of the Lean 3 parser. Lean 4, under development at the time of submission of this paper, features a highly customizable parser with precisely this kind of domain specific language embedding in mind [36].

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

[3] https://github.com/leanprover-community/lean/blob/master/library/init/meta/widget/basic.lean

5 Querying Lean from Mathematica

The 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.

More generally, the exploratory uses of CAS tools rely on databases of definitions and examples. Mathematica features enormous data sets, mathematical and otherwise, but these mainly describe computational objects, with only a few examples from pure mathematics [20]. Connecting Lean to Mathematica allows the Lean library mathlib to serve as a collection of definitions, theorems, and proofs that the CAS can inspect and process.

5.1 Connection interface

We use Mathematica's external command interface `StartProcess` to establish a connection to Lean from a Mathematica notebook. The notebook communicates with a Lean server session via the same interface used by Lean editors. All calls to Lean are directed to a particular server process, which allows for the option to run and query multiple server processes at once. The standard mode of interaction, though, is to begin a session by defining a `ProcessObject` expression

```
Lean = LeanMode[]
```

and to send all requests to this single `ProcessObject`. As in Section 4.1, this server–client interface allows us to preserve Lean environment state as needed and to avoid the cost of launching a new process for each query.

A low level communication function

```
SendToLeanServer[p_ProcessObject, content_String]
```

evaluates an arbitrary server request `content` in Lean process `p`. We build higher level tools on top of this. For instance, a function

```
RunLeanTactic[p_ProcessObject, x_, t_String]
```

takes an arbitrary Mathematica expression `x` and a string `t` naming a Lean tactic, and returns the result of calling `t` on the Lean interpretation of `x`. It does so by constructing the top-level syntax needed to perform this operation and passing this syntax to `RunLeanTactic`.

In[6]:= `GetLeanInfo[Lean, "nat.exists_infinite_primes"]`

Out[6]=

Category	Theorem
Type	∀ (n : ℕ), ∃ (p : ℕ), n ≤ p ∧ nat.prime p
Description	▼Documentation Euclid's theorem. There exist infinitely many prime numbers. Here given in the form: for every `n`, there exists a prime number `p ≥ n`.

Fig. 5: Retrieving the declaration information from `nat.exists_infinite_primes`.

5.2 Applications

Querying about certain declarations. Mathematica contains many databases ranging over a huge variety of topics. A major motivation to connect Lean to Mathematica is to treat the proof assistant library as another such database. As a simple example of this kind of use, we implement a function

```
GetLeanInfo[p_ProcessObject, decl_String]
```

that displays information about the declaration named `decl` found in the Lean process `p` (Figure 5). This information includes the declaration's category (is it an axiom, a definition, a theorem?), its type, and any documentation associated with it. The natural language description in this output is taken from the same source as the mathlib API documentation [18].

By default, `GetLeanInfo` returns a structure whose fields are strings, as this is most convenient to print and display. But it is very easy to instead retrieve the type as an expression. While an arbitrary mathlib declaration is unlikely to fully translate to a Mathematica counterpart, one may wish to perform further processing on the syntax tree of the type. It is also a simple matter to retrieve the body of the declaration, as we do in some examples below.

Displaying propositional proofs. Mathematica's `TautologyQ` and `FullSimplify` functions serve as complete SAT solvers. However, both are black boxes, in that 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 uses a small grammar of proof rules.

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.

We emphasize here that the input formula is a pure Mathematica object, for example `Implies[Or[P, Q], Not[And[Not[P], Not[Q]]]]`. Generating these proofs does not require any knowledge of Lean syntax or the encoding of Mathematica syntax in Lean.

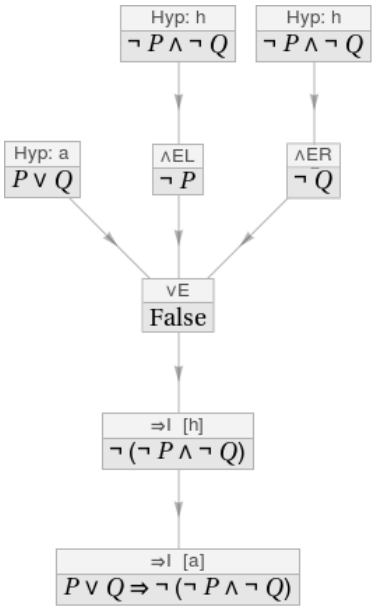

Fig. 6: A natural deduction diagram generated from a Lean proof term.

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 (Figure 6). 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 arbitrary proofs. Natural deduction-style proofs are most commonly seen in pure propositional and first-order logic, making the diagrams of the previous section particularly familiar. However, the presentation style can be adapted to richer logics. Lean's `#explode` command, implemented in mathlib, formats a Lean proof term in a way resembling a Fitch natural deduction diagram.

Natural deduction proofs are not known for their brevity. The textual output of `#explode`, viewed in a Lean editor session, can be overwhelmingly long. It is much more enlightening to be able to expand and fold the output. We implement a function `GetLeanProof[p_ProcessObject, decl_String]` in Mathematica that retrieves the exploded output of the Lean declaration named `decl` from the Lean process `p` and formats it as an interactive object in the Mathematica notebook. Users can drill down into the details of the proof as deep as needed to understand its structure.

Figure 7 displays the beginning of an exploded proof of the fact that there are infinitely many primes. At each node of the diagram there appears a goal, labeled with a unique index. The goal may be referenced by this index in subsequent nodes. Each node is justified by the application of a particular rule: the original goal is proved by an application of universal introduction, after which we introduce a fresh variable `n` by the assumption rule. Some nodes are justified by the application of library lemmas, for instance node 29, in which the goal `n! + 1 ≠ 1` is justified by applying the lemma `ne_of_gt`. Depend on the rule applied, a node may have arguments, represented as nodes themselves. An application of universal introduction takes two arguments, the newly introduced variable and a proof of the remaining goal; introducing this variable is an atomic step with no arguments.

Due to the Curry–Howard correspondence, the same technique can be used to inspect data-valued expressions, e.g. to unfold parts of a declaration defining a natural number.

This application demonstrates how mathematicians can use our link to explore the proof assistant library without leaving the CAS. It takes advantage of Math-

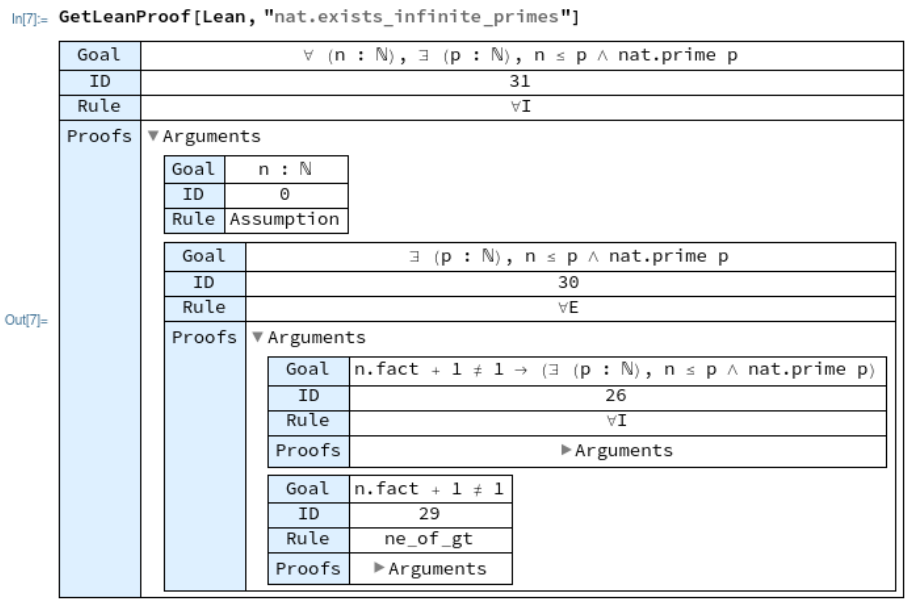

Fig. 7: An expandable Fitch-style diagram generated from a Lean proof term.

ematica's highly developed user interface to visualize the details of arguments in a way that is nearly impossible to do from within Lean, especially for proofs that use substantial automation or that do not have easily legible proof scripts.

Proving Mathematica goals. As with the examples from propositional logic, the Wolfram Language is rich enough to express some propositions that can be proved in Lean. We implement a function

```
ProveUsingLeanTactic[p_ProcessObject, x_,t_String]
```

that takes an arbitrary Mathematica expression `x` and a Lean tactic string `t` and tries to translate and prove the Mathematica expression, finally displaying the exploded proof. While in principle this accepts any tactic proof, it is not convenient to interactively write a proof from the Mathematica notebook, and so this is typically used for statements that can be solved with one or two lines.

As mentioned at the beginning of this section, we see this as informally lending the semantics of Lean to Mathematica expressions. The semantic value of the proposition `x` in Mathematica is merely if and whether it simplifies to `True` or `False`. With our translation and link, we provide a true proof language for some subset of these expressions.

Displaying significant proof steps. The full output of `#explode` may sometimes contain too much detail, as the logical steps like universal introduction may be unenlightening. An alternate way to visualize a proof is to identify and display important lemmas or relevant steps used within. 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 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 [8]. 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.

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 [23] 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 [16] 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.

This skeptical approach is also taken in some projects that require one specific type of computer algebra computation instead of a generic link. Harrison [22] computes Wilf–Zeilberger certificates in Maxima and verifies them in HOL Light. While this work does establish an interface between the two systems, Harrison notes the convenience of a "manual" version, where users generate certificates in Maxima and transfer them to HOL Light by hand. Chyzak, Mahboubi, Sibut-Pinote, and Tassi [12,27] use certificates from Maple in the verification of a critical

lemma for proving the irrationality of $\zeta(3)$. Here the CAS results are only transferred manually. These instances of manual translation between a CAS notebook and a proof assistant suggest future work on an integrated user interface, described in the final section of this paper.

Ballarin and Paulson [4] provide a connection between Isabelle and the computer algebra library Σ^{IT} [10] that is more trusting than the previous skeptical 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 [35] 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 [17], whose CoqEAL library implements a number of algorithms in Coq.

Kerber, Kohlhase, and Sorge [25] 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 [24] 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. Adams et al. [1] use PVS to verify side conditions generated in computations in Maple; Gottliebsen, Kelsey, and Martin [21] make use of similar ideas. Systems such as Analytica [5] and Theorema [11] provide ATP- or ITP-style behavior from within Mathematica. Axiom [15] 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 can communicate with Mathematica in true notebook style with access to the environment at a particular point in a Lean file. In the imagined Mathematica notebook, quoted Lean pre-expressions will be elaborated in the environment at the demarcated point in the Lean file, and then reflected and processed in Mathematica. The results can then be easily exported to the Lean file. The notebook is a standard way of interacting with computer algebra systems and contributes to their utility in exploration and discovery. While the embedded Mathematica code blocks described in Section 4.3 are a first-degree approximation to this kind of interaction, a full-fledged notebook interface is more natural to use. Our link

implements all the translation and communication features needed for such an integration but substantial UI engineering is needed to make it a reality.

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. Improvements to the foreign function interface in Lean 4 may allow the physical connection to be made more robust.

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 [9], verify its side conditions in Z3 [31], visualize the results in Mathematica, and export relevant formulas to LaTeX. Unified frameworks have been proposed and implemented [33] 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.

Acknowledgements We acknowledge Jeremy Avigad, Jasmin Blanchette, Ian Ford, Johannes Hölzl, José Martín-García, Leonardo de Moura, James Mulnix, Michael Trott, and the Lean community for help, suggestions, and support.

References

1. Adams, A., Dunstan, M., Gottliebsen, H., Kelsey, T., Martin, U., Owre, S.: 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, pp. 27–42. Springer-Verlag, London, UK, UK (2001). URL http://dl.acm.org/citation.cfm?id=646528.695189
2. Bailey, D.H., Borwein, J.M., Kapoor, V., Weisstein, E.W.: Ten problems in experimental mathematics. American Mathematical Monthly **113**(6), 481–509 (2006)
3. Ballarin, C., Homann, K., Calmet, J.: Theorems and algorithms: An interface between Isabelle and Maple. In: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation, ISSAC '95, pp. 150–157. ACM, New York, NY, USA (1995). DOI 10.1145/220346.220366. URL http://doi.acm.org/10.1145/220346.220366
4. Ballarin, C., Paulson, L.C.: 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)
5. Bauer, A., Clarke, E., Zhao, X.: Analytica – an experiment in combining theorem proving and symbolic computation. Journ. Autom. Reas. **21**(3), 295–325 (1998). DOI 10.1023/A:1006079212546. URL http://dx.doi.org/10.1023/A:1006079212546
6. Besson, F.: Fast reflexive arithmetic tactics the linear case and beyond. In: T. Altenkirch, C. McBride (eds.) Types for Proofs and Programs, pp. 48–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
7. Blanchette, J., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. Interactive Theorem Proving pp. 131–146 (2010)
8. Blanchette, J.C., Kaliszyk, C., Paulson, L.C., Urban, J.: Hammering towards QED. Journal of Formalized Reasoning **9**(1), 101–148 (2016)
9. Bosma, W., Cannon, J., Playoust, C.: The magma algebra system i: The user language. J. Symb. Comput. **24**(3–4), 235–265 (1997). DOI 10.1006/jsco.1996.0125. URL https://doi.org/10.1006/jsco.1996.0125
10. Bronstein, M.: σit—a strongly-typed embeddable computer algebra library. In: International Symposium on Design and Implementation of Symbolic Computation Systems, pp. 22–33. Springer (1996)

11. Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., Windsteiger, W.: Theorema 2.0: Computer-assisted natural-style mathematics. Journal of Formalized Reasoning **9**(1), 149–185 (2016). DOI 10.6092/issn.1972-5787/4568. URL https://jfr.unibo.it/article/view/4568
12. Chyzak, F., Mahboubi, A., Sibut-Pinote, T., Tassi, E.: A computer-algebra-based formal proof of the irrationality of $\zeta(3)$. In: G. Klein, R. Gamboa (eds.) Interactive Theorem Proving, pp. 160–176. Springer International Publishing, Cham (2014)
13. Coquand, T., Huet, G.: The Calculus of Constructions. Inform. and Comput. **76**(2-3), 95–120 (1988). DOI 10.1016/0890-5401(88)90005-3. URL http://dx.doi.org/10.1016/0890-5401(88)90005-3
14. Coquand, T., Paulin, C.: Inductively defined types. In: COLOG-88 (Tallinn, 1988), *Lec. Notes in Comp. Sci.*, vol. 417, pp. 50–66. Springer, Berlin (1990). DOI 10.1007/3-540-52335-9_47. URL http://dx.doi.org/10.1007/3-540-52335-9_47
15. Daly, T.: Axiom: The 30 year horizon. Lulu Incorporated (2005)
16. Delahaye, D., Mayero, M.: Dealing with algebraic expressions over a field in Coq using Maple. Journal of Symbolic Computation **39**(5), 569 – 592 (2005). DOI http://dx.doi.org/10.1016/j.jsc.2004.12.004. URL http://www.sciencedirect.com/science/article/pii/S0747717105000283. Automated Reasoning and Computer Algebra Systems (AR-CA)
17. Dénès, M., Mörtberg, A., Siles, V.: A refinement-based approach to computational algebra in Coq. In: Interactive theorem proving, *Lecture Notes in Comput. Sci.*, vol. 7406, pp. 83–98. Springer, Heidelberg (2012). DOI 10.1007/978-3-642-32347-8_7. URL http://dx.doi.org/10.1007/978-3-642-32347-8_7
18. van Doorn, F., Ebner, G., Lewis, R.Y.: Maintaining a library of formal mathematics. In: C. Benzmüller, B. Miller (eds.) Intelligent Computer Mathematics, pp. 251–267. Springer International Publishing, Cham (2020)
19. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proceedings of the ACM on Programming Languages **1**(ICFP), 34 (2017)
20. Ford, I.: Semantic representation of general topology in the wolfram language. In: H. Geuvers, M. England, O. Hasan, F. Rabe, O. Teschke (eds.) Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings, *Lecture Notes in Computer Science*, vol. 10383, pp. 163–177. Springer (2017). DOI 10.1007/978-3-319-62075-6_12. URL https://doi.org/10.1007/978-3-319-62075-6_12
21. Gottliebsen, H., Kelsey, T., Martin, U.: Hidden verification for computational mathematics. Journal of Symbolic Computation **39**(5), 539 – 567 (2005). DOI https://doi.org/10.1016/j.jsc.2004.12.005. URL http://www.sciencedirect.com/science/article/pii/S0747717105000295. Automated Reasoning and Computer Algebra Systems (AR-CA)
22. Harrison, J.: Formal proofs of hypergeometric sums. J. Autom. Reason. **55**(3), 223–243 (2015). DOI 10.1007/s10817-015-9338-0. URL https://doi.org/10.1007/s10817-015-9338-0
23. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and Maple. J. Automat. Reason. **21**(3), 279–294 (1998). DOI 10.1023/A:1006023127567. URL http://dx.doi.org/10.1023/A:1006023127567
24. Kaliszyk, C., Wiedijk, F.: 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, pp. 94–105. Springer-Verlag, Berlin, Heidelberg (2007). DOI 10.1007/978-3-540-73086-6_8. URL http://dx.doi.org/10.1007/978-3-540-73086-6_8
25. Kerber, M., Kohlhase, M., Sorge, V.: Integrating computer algebra into proof planning. J. Automat. Reason. **21**(3), 327–355 (1998). DOI 10.1023/A:1006059810729. URL http://dx.doi.org/10.1023/A:1006059810729
26. Lewis, R.Y.: An extensible ad hoc interface between Lean and Mathematica. In: C. Dubois, B.W. Paleo (eds.) Proceedings of the Fifth Workshop on Proof eXchange for Theorem Proving, PxTP 2017, Brasília, Brazil, 23-24 September 2017, *EPTCS*, vol. 262, pp. 23–37 (2017). DOI 10.4204/EPTCS.262.4. URL https://doi.org/10.4204/EPTCS.262.4
27. Mahboubi, A., Sibut-Pinote, T.: A formal proof of the irrationality of $\zeta(3)$ (2019)
28. The mathlib Community: The Lean mathematical library. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, p. 367–381. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3372885.3373824. URL https://doi.org/10.1145/3372885.3373824

29. McBride, C., McKinna, J.: Functional pearl: I am not a number–I am a free variable. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell, Haskell '04, pp. 1–9. ACM, New York, NY, USA (2004). DOI 10.1145/1017472.1017477. URL http://doi.acm.org/10.1145/1017472.1017477
30. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover. http://leanprover.github.io/files/system.pdf (2014)
31. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS, pp. 337–340 (2008)
32. Pratt, V.R.: Every prime has a succinct certificate. SIAM Journal on Computing **4**(3), 214–220 (1975)
33. Rabe, F.: The MMT API: a generic MKM system. In: International Conference on Intelligent Computer Mathematics, pp. 339–343. Springer (2013)
34. Schrijver, A.: Theory of linear and integer programming. Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons Ltd., Chichester (1986). A Wiley-Interscience Publication
35. Seddiki, O., Dunchev, C., Khan-Afshar, S., Tahar, S.: Enabling Symbolic and Numerical Computations in HOL Light, pp. 353–358. Springer International Publishing, Cham (2015). DOI 10.1007/978-3-319-20615-8_27. URL https://doi.org/10.1007/978-3-319-20615-8_27
36. Ullrich, S., de Moura, L.: Beyond notations: Hygienic macro expansion for theorem proving languages. In: N. Peltier, V. Sofronie-Stokkermans (eds.) Automated Reasoning, pp. 167–182. Springer International Publishing, Cham (2020)
37. Williams, H.: Fourier's method of linear programming and its dual. The American Mathematical Monthly **93**(9), 681–695 (1986)
38. Wolfram, S.: An Elementary Introduction to the Wolfram Language. Wolfram Media, Incorporated (2015). URL https://books.google.com/books?id=efIvjgEACAAJ