An extensible ad hoc interface between Lean and Mathematica

Robert Y. Lewis

Carnegie Mellon University

September 24, 2017

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

Robert Y. Lewis   Minchao Wu

Carnegie Mellon University

September 24, 2017

Computers in mathematics

Computers are becoming more and more common as tools for mathematicians.

- Typesetting, ArXiV, wikis, search tools
- Graphing, plotting, visualization tools for "experimental mathematics"
- SAT/SMT solvers solving combinatorial problems
- Domain-specific computations: homotopy groups, geometric hulls, (non)linear systems
- Computer algebra systems for symbolic and numeric computation
- Interactive theorem provers for formal verification

Computer algebra systems

Strengths:

- They are easy and useful
- They provide instant gratification
- They support interactive use, exploration
- They are programmable and extensible

Weaknesses:

- The focus is on symbolic computation, rather than abstract definitions and assertions
- They are not designed for reasoning or search
- The semantics is murky
- They are sometimes inconsistent

Interactive theorem provers

Strengths:

- Their languages are expressive and well-specified
- They come with a precise semantics
- Results are fully verified

Weaknesses:

- Formalization is slow and tedious
- It requires a high degree of commitment and experise
- It doesn't promote exploration and discovery

ITP + CAS

By linking the two, we can

- Allow exploration and computation in the proof assistant, without reimplementing algorithms
- Lower the barrier for newcomers to ITP
- Loan a semantics/proof language to CAS

Many projects have attempted to connect the two: verified CAS algorithms, trusting links, verified links, ephemeral links, CAS proof languages.

Contributions

- An extensible procedure to interpret Lean expressions in Mathematica
- An extensible procedure to interpret Mathematica expressions in Lean
- A link allowing Lean to evaluate arbitrary Mathematica commands, and recieve the results
- Tactics for certifying results of particular Mathematica computations
- A link allowing Mathematica to execute particular Lean tactics and recieve the results

Broader picture

Proof assistants, with clear semantics, can serve as glue between many different mathematical tools.

Outline

Introduction

Background: Lean and Mathematica

Linking Lean and Mathematica
- Translating Lean to Mathematica
- Translating Mathematica to Lean

Verifying results

Calling Lean from Mathematica (a preview)

Background: Lean

Lean is a new interactive theorem prover, developed principally by Leonardo de Moura at Microsoft Research, Redmond.

Lean is open source, released under a permissive license, Apache 2.0.

Calculus of constructions with:

- Non-cumulative hierarchy of universes
- Impredicative `Prop`
- Quotient types and propositional extensionality
- Axiom of choice available

See `http://leanprover.github.io`, or Leonardo de Moura's talk at ITP.

Lean as a Programming Language

Lean implements a fast bytecode evaluator:

- It uses a stack-based virtual machine.
- It erases type information and propositional information.
- It uses eager evaluation (and supports delayed evaluation with thunks).
- You can use anything in the Lean library, as long as it is not `noncomputable`.
- The machine substitutes native nats and ints (and uses GMP for large ones).
- It substitutes a native representation of arrays.
- It has a profiler and a debugger.
- It is really fast.

Lean as a Programming Language

Definitions tagged with `meta` are "VM only," and allow unchecked recursive calls.

```
meta def f : ℕ → ℕ
| n := if n=1 then 1
       else if n%2=0 then f (n/2)
       else f (3*n + 1)

#eval (list.iota 1000).map f
```

Lean as a Programming Language

There are algebraic structures that provides an interface to terminal and file I/O.

Lean's built-in package manager is implemented entirely in Lean.

Metaprogramming in Lean

Question: How can one go about writing tactics and automation?

Various answers:

- Use the underlying implementation language (ML, OCaml, C++, . . .).
- Use a domain-specific tactic language (LTac, MTac, Eisbach, . . .).
- Use reflection (RTac).

Metaprogramming in Lean

Lean's answer: go meta, and use Lean itself.

(MTac, Idris, and now Agda do the same, with variations.)

Advantages:

- Users don't have to learn a new programming language.
- The entire library is available.
- Users can use the same infrastructure (debugger, profiler, etc.).
- Users develop metaprograms in the same interactive environment.
- Theories and supporting automation can be developed side-by-side.

Metaprogramming in Lean

```
meta def find : expr → list expr → tactic expr
| e []          := failed
| e (h :: hs) :=
  do t ← infer_type h,
     (unify e t >> return h) <|> find e hs

meta def assumption : tactic unit :=
do { ctx ← local_context,
     t   ← target,
     h   ← find t ctx,
     exact h }
<|> fail "assumption tactic failed"

lemma simple (p q : Prop) (h₁ : p) (h₂ : q) : q :=
by assumption
```

Background: Mathematica

Mathematica is a powerful and popular computer algebra system developed at Wolfram Research, implementing the Wolfram Language.

It provides a vast variety of functions for manipulating mathematical expressions, as well as tools for manipulating and displaying data.

Background: Mathematica

Some basic Mathematica syntax and terminology:

- Function application
  - `Plus[x, y]`
  - `Plus[x, y, z]`
  - `x + y + z`
  - `Factor[x^2 - 2x + 1]`
  - `x^2 - 2x + 1 // Factor`
- In `Plus[x, y]`, we refer to `Plus` as the *head symbol* and `x`, `y` as the *arguments*
- Head symbols can be given computational behavior via pattern matching rules: `MyFunc[s_String] := Reverse[s]`

Outline

Introduction

Background: Lean and Mathematica

Linking Lean and Mathematica
- Translating Lean to Mathematica
- Translating Mathematica to Lean

Verifying results

Calling Lean from Mathematica (a preview)

Link architecture

We'll focus on using Mathematica from within Lean.
We'll need:

- A Lean-to-Mathematica reflection (in Lean)
- A Lean-to-Mathematica interpretation (in Mathematica)
- Computation functions (in Mathematica)
- A Mathematica-to-Lean reflection (in Lean)
- A Mathematica-to-Lean interpretation (in Lean)
- Verification functions (in Lean)

Lean expression grammar

```
meta inductive expr (elaborated : bool := tt)
| 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

meta def mathematica_form_of_level : level → string := ...
meta def mathematica_form_of_name : name → string := ...
meta def mathematica_form_of_expr : expr → string := ...
```

Lean expression grammar

```
x : real ⊢ sin x : real

app (const ‘sin [])
   (local_const ‘x ‘x binder_info.default (const ‘real []))
```

Applying `mathematica_form_of_expr` produces:

```
App[Const[``sin'', LListNil],
    LocalConst[``x'', ``x'', BID,
                 Const[``real'', LListNil]]]
```

Mathematica translation rules

The head symbols `App`, `Const`, etc. are uninterpreted in Mathematica.

We want to exploit the facts that:

- certain Lean constants correspond to certain Mathematica constants
  - **`sin`** "means" the same as `Sin`
- certain expression patterns in Lean correspond to certain expression patterns in Mathematica
  - λ**`x, t`** "means" the same as `Function[x, t]`

Mathematica translation rules

We define a Mathematica function `LeanForm` using pattern matching rules:

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

LeanForm[App[Const["list.nil", _], _]] :=
  {}

LeanForm[App[App[App[
    Const["list.cons", _], _], h_], t_]] :=
  Join[{LeanForm[h]}, LeanForm[t]]
```

Mathematica translation rules

With the right set of `LeanForm` rules, we can reduce a Lean expression to something semantically meaningful in Mathematica.

`⟦2+3⟧ // LeanForm` evaluates to `Inactive[Plus][2, 3]`

In a local context with `x : real`,
`⟦x^2 - 2*x + 1⟧ // LeanForm // Activate // Factor`
evaluates to `Power[Plus[-1,⟦x⟧], 2]`

Mathematica expression grammar

```
inductive mmexpr
| sym  : string → mmexpr
| str  : string → mmexpr
| int  : int → mmexpr
| real : float → mmexpr
| app  : mmexpr → list mmexpr → mmexpr
```

Mathematica expressions are built out of atoms and applications of expressions to lists of expressions.

Analogous to the representation of Lean expressions in Mathematica, we can represent any Mathematica expression in Lean with a term of type `mmexpr`.

It is easy to implement
`parse_to_mmexpr : string → tactic mmexpr`.

Lean interpretation rules

Mathematica expressions usually correspond to Lean pre-expressions (unelaborated / input-level).
`Plus[x, 1]` is analogous to `add x 1` which elaborates to something longer.

(One exception: the Lean representation of the Mathematica representation of a Lean expression corresponds to the original Lean expression.)

The Lean types `expr` and `pexpr` share the same implementation.
`to_expr : pexpr → tactic expr` attempts to elaborate a `pexpr`.

Lean interpretation rules

We define `pexpr_of_mmexpr : mmexpr → tactic pexpr` which tries to interpret a Mathematica expression into a Lean pre-expression.

New interpretation rules can be declared in Lean:

```
@[app_to_pexpr_keyed]
meta def list_to_pexpr : app_to_pexpr_keyed_rule :=
⟨"List",
 λ ctx args,
  do args' ← args.mfor (pexpr_of_mmexpr ctx),
     return $ args'.foldr (λ h t, ``(%%h :: %%t)) ``([])⟩
```

Lean interpretation rules

Lean and Mathematica handle binders differently.

- Lean: λ and Π bind anonymous variables
- Mathematica: `Function`, `Integral`, many other head symbols bind a specified symbol

The translation procedure must be aware of the local context.

```
@[app_to_pexpr_keyed]
meta def function_to_pexpr : app_to_pexpr_keyed_rule :=
⟨"Function", λ ctx args, match args with
| [sym x, bd] :=
  let v := mk_local_const x,
  do  bd' ← pexpr_of_mmexpr (ctx.insert x v) bd,
      return $ mk_lambda' v bd'
| _ := failed
end⟩
```

Lean interpretation rules

Three types of translation rules: symbol, keyed app, unkeyed app

```
@[sym_to_expr]
meta def true_to_pexpr : sym_to_expr_rule :=
⟨"True", `(true)⟩

@[app_to_pexpr_unkeyed]
meta def app_inactive_to_pexpr : app_to_pexpr_unkeyed_rule
| env (app (sym "Inactive") [t]) l :=
    pexpr_of_mmexpr env (app t l)
| _ _ _ := failed
```

Communication protocol

So far, we have

- `mathematica_form_of_expr : expr → string` to represent a Lean expression in Mathematica
- `LeanForm` to interpret a Lean expression in Mathematica
- The entire Mathematica library to compute with interpreted expressions
- `lean_form_of_mm_expr : string → tactic mmexpr` to represent a Mathematica expression in Lean
- `pexpr_of_mmexpr : mmexpr → tactic pexpr` to interpret a Mathematica expression in Lean

To glue them together, we need
`evaluate_in_mathematica : string → tactic string`

Communication protocol

We implement a simple server in Mathematica. The server listens for requests, evaluates the requests, and returns the result as a string.

A corresponding Python client sends its argument to the server for evaluation.

`evaluate_in_mathematica : string → tactic string` calls the client using Lean's IO monad, which provides a command-line interface to external programs.

Communication protocol

So we can combine these functions into

```
meta def evaluate_command_on_expr
     (cmd : string → string) (e : expr) : tactic pexpr :=
let e' := mathematica_form_of_expr e in
do  mm ← lean_form_of_mm_expr (cmd e'),
    pexpr_of_mmexpr mm
```

More concretely, we could write

```
meta def factor (e : expr) : tactic expr :=
do tp ← infer_type e,
   pe ← evaluate_command_on_expr
          (λ s, s ++ "// LeanForm // Activate // Factor")
          e,
   to_expr ``(%%pe : %%tp)
```

Outline

Introduction

Background: Lean and Mathematica

Linking Lean and Mathematica
- Translating Lean to Mathematica
- Translating Mathematica to Lean

Verifying results

Calling Lean from Mathematica (a preview)

Certification

There's little reason to trust the output of Mathematica, and less reason to trust this translation process.

For many computations, verifying that a result has some property is easier than computing the result itself.
E.g. if `p` is a polynomial, it is easier to verify that `Factor[p] = p` than to compute `Factor[p]`.

We pair this translation procedure with a set of task-specific verification procedures.

Example: factoring polynomials

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

meta def assert_factor (e : expr) (nm : name) : tactic unit :=
do fe ← factor e,
   pf ← eq_by_simp e fe,
   note nm pf

example (x : ℝ) : 1 - 2*x + 3*x^2 - 2*x^3 + x^4 ≥ 0 :=
begin
 assert_factor 1 - 2*x + 3*x^2 - 2*x^3 + x^4 using h,
 rewrite h,
 apply sq_nonneg
end
```

Example: linear arithmetic

Motzkin transposition theorem

Let P, Q, R be matrices and $\mathbf{p}, \mathbf{q}, \mathbf{r}$ vectors.

- $P\mathbf{x} > \mathbf{p}, Q\mathbf{x} \geq \mathbf{q}, R\mathbf{x} = \mathbf{r}$ has no solution $\mathbf{x}$

if and only if

- $P^T\mathbf{y}_1 + Q^T\mathbf{y}_2 + R^t\mathbf{y}_3 = 0$ has a solution $\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3$ with $\mathbf{y}_1, \mathbf{y}_2 \geq 0$ and either
  - $\mathbf{y}_1 \cdot \mathbf{p} + \mathbf{y}_2 \cdot \mathbf{q} + \mathbf{y}_3 \cdot \mathbf{r} < 0$ or
  - $\mathbf{y}_1 \cdot \mathbf{p} + \mathbf{y}_2 \cdot \mathbf{q} + \mathbf{y}_3 \cdot \mathbf{r} \leq 0$ and $\mathbf{y}_1 > 0$.

This theorem gives a notion of certificate for linear arithmetic.

Example: linear arithmetic

We can use Mathematica to generate witnesses to the MTT and apply them to produce fully checkable proofs.

```
example (a b c d e f g : ℤ)
 (h1 : 1*a + 2*b + 3*c + 4*d + 5*e + 6*f + 7*g ≤ 30)
 (h2 : (-1)*a ≤ 4)
 (h3 : (-1)*b + (-2)*d ≤ -4)
 (h4 : (-1)*c + (-2)*f ≤ -5)
 (h5 : (-1)*e ≤ -3)
 (h6 : (-1)*g ≤ -2) : false :=
by not_exists_of_linear_hyps h1 h2 h3 h4 h5 h6
```

Example: sanity checking

Many computations in Mathematica are not easily certifiable, but can still be useful in interactive proofs.

`sanity_check` runs the Mathematica command `FindInstance` to search for an assignment satisfying the hypotheses and the negation of the goal. The tactic fails if an assignment is found.

```
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
```

Example: Mathematica as an oracle

There is a spectrum of trust levels. Some users may be comfortable using Mathematica as an oracle.

```
meta def full_simp (e : expr) : tactic (expr × expr) :=
do pe ← evaluate_command_on_expr
        (λ t, t ++ "//LeanForm//Activate//FullSimplify")
        e,
   eqtp ← to_expr ``(%%e = %%pe),
   ax_name ← add_axiom eqtp,
   proof ← mk_const ax_name,
   return (val, proof)

example (x : ℝ) :
       x*BesselJ 2 x + x*BesselJ 0 x = 2*BesselJ 1 x :=
by prove_by_full_simp
```

Example: Mathematica as an oracle

We can also use Mathematica to obtain approximations of constants and axiomatize these bounds:

```
example :=
begin
  approx (100*BesselJ 2 0.52) (0.00001 : ℝ),
  trace_state
end

/-
approx : 12887461 / 3900000 < 100 * BesselJ 2 (13 / 25)
        ∧ 100 * BesselJ 2 (13 / 25) < 12887539 / 3900000
⊢ true
-/
```

Outline

Introduction

Background: Lean and Mathematica

Linking Lean and Mathematica
- Translating Lean to Mathematica
- Translating Mathematica to Lean

Verifying results

Calling Lean from Mathematica (a preview)

Calling Lean from Mathematica

(A preview)

Mathematica has no built in notion of "proof" or "correctness." For some functions, it's not even clear what the intended semantics are.

We can consider a "proposition" to be true if applying `FullSimplify` evaluates to `True`, but this is of limited scope, and `FullSimplify` is a black box.

Idea: translate Mathematica "propositions" to Lean, where they have semantics and a proof language.

Calling Lean from Mathematica

(A preview)

`ProveUsingLeanTactic[p_, tac_]` takes

- an expression `p`
- a Lean tactic string `tac`

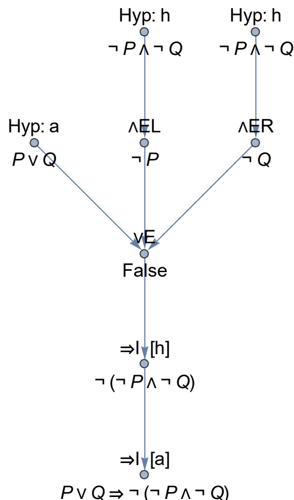
It translates `p` into a Lean expression `p'`, attempts to prove `p'` using `tac`, and returns the resulting proof term.

Calling Lean from Mathematica

(A preview)

We can add rules to (try to) translate the resulting proof.

```
DiagramOfFormula[
 ForAll[{P, Q},
  Implies[
   Or[P, Q],
   Not[And[Not[P], Not[Q]]]
  ]
 ]
]
```



Calling Lean from Mathematica

(A preview)

We can try to:

- Verify the output of `FullSimplify` or other computations
- Discover missing side conditions
- Extract “interesting” parts of proofs
- “Type-check” or “elaborate” certain Mathematica expressions
- Explore the Lean library