A Metaprogramming Framework for Formal Verification

GABRIEL EBNER, Vienna University of Technology, Austria
SEBASTIAN ULLRICH, Karlsruhe Institute of Technology, Germany
JARED ROESCH, University of Washington, USA
JEREMY AVIGAD, Carnegie Mellon University, USA
LEONARDO DE MOURA, Microsoft Research, USA

We describe the metaprogramming framework currently used in Lean, an interactive theorem prover based on dependent type theory. This framework extends Lean's object language with an API to some of Lean's internal structures and procedures, and provides ways of reflecting object-level expressions into the metalanguage. We provide evidence to show that our implementation is performant, and that it provides a convenient and flexible way of writing not only small-scale interactive tactics, but also more substantial kinds of automation.

CCS Concepts: • **Theory of computation → Interactive proof systems**; **Type theory**; • **Software and its engineering → Functional languages**;

Additional Key Words and Phrases: theorem proving, dependent type theory, tactic language, metaprogramming

ACM Reference Format:
Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP, Article 34 (September 2017), 29 pages.
https://doi.org/10.1145/3110278

1 INTRODUCTION

Variants of dependent type theory have been implemented in interactive theorem provers such as Coq [Bertot and Castéran 2004], Agda [Bove et al. 2009; Norell 2008], Matita [Asperti et al. 2011], and Lean [de Moura et al. 2015], and experience has shown that it provides a powerful and expressive framework for the use of formal methods. Dependent type theory makes it possible to define common mathematical data types, constructions, and structures in natural ways, and to define mathematical objects, functions, and instances of those structures. We can also make mathematical assertions and construct proofs of those assertions, all in the very same language. Because dependent type theory has a computational interpretation, we can compute with such definitions and reason about the results of these computations.

Dependent type theory is also flexible enough to serve as its own *metaprogramming language*, that is, a language in which one can write programs that assist in the construction and manipulation of terms in dependent type theory itself [Brady 2013; Christiansen and Brady 2016; van der Walt and Swierstra 2012; Ziliani et al. 2015]. Here we describe the metaprogramming framework currently in use in Lean [de Moura et al. 2015], a new open source theorem prover that is designed to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Association for Computing Machinery.
2475-1421/2017/9-ART34
https://doi.org/10.1145/3110278

RIGHTSLINK

bridge the gap between interactive use and automation. Lean implements a version of the Calculus of Inductive Constructions [Coquand and Huet 1988; Coquand and Paulin 1990], the details of which are described in Section 3. Its elaborator and unification algorithms are designed around the use of type classes, which support algebraic reasoning, programming abstractions, and other generally useful means of expression. Lean also has parallel compilation and checking of proofs, and provides a server mode that supports a continuous compilation and rich user interaction in editing environments such as Emacs and Visual Studio Code. It currently has a conditional term rewriter and several components commonly found in state-of-the-art Satisfiability Modulo Theories (SMT) solvers, such as forward chaining, congruence closure [Nelson and Oppen 1980], handling of associative and commutative operators, and E-matching [Detlefs et al. 2005].

In Lean, definitions are compiled to bytecode that can then be evaluated in an efficient virtual machine. This is one sense in which Lean can be viewed as a programming language. (Native compilation is under development.) We obtain a metaprogramming language by exposing an API for procedures implemented natively in Lean's underlying C++ code base, thus taking us outside the axiomatic framework. Within a Lean source file, the keyword `meta` marks a clear distinction between definitions that make use of such extensions and those that are in the pure object language. Metadefinitions can also call themselves recursively, relaxing the termination restriction imposed by ordinary type theory. Otherwise, definitions and metadefinitions look very much the same, and can be written side-by-side in a Lean source file.

There are a number of advantages to this approach. One is that users do not have to learn a new programming language to write metaprograms; they can work with the same constructs and notation used to define ordinary objects in the theorem prover's library. Another advantage is that everything in that library is available for metaprogramming purposes, including integers, lists, datatype constructors, records, and algebraic structures. Finally, metaprograms can be written and debugged in the same interactive environment, making it possible to develop the object library and supporting automation at the same time.

A key application for metaprogramming is to implement *tactics*, which is to say, procedures which facilitate interactive theorem proving by carrying out straightforward reasoning steps and boilerplate constructions. There are a number of tactic languages currently on offer, including Ltac [Delahaye 2000, 2002], Mtac [Ziliani et al. 2015], and MetaCoq [Ziliani et al. 2017] for Coq and Eisbach [Matichuk et al. 2016] for Isabelle. In the next section, we present examples of ways in which our language can be used to implement such tasks. But a key advantage of our approach is that dependent type theory is a full-scale programming language, allowing for the development of more substantial forms of automation. For example, we show how to implement a robust simplification procedure, which works well even in the absence of directed, confluent rewrite rules. In Section 5, we show that it is straightforward to implement a more powerful version of the `crush` tactic that is a mainstay of *Certified Programming with Dependent Types* [Chlipala 2011]. As an extreme example, we also describe the implementation of a full-blown superposition theorem prover in Section 5.

Contributions. We describe the implementation of a metaprogramming language in the Lean theorem prover.

- We describe Lean's metaprogramming API, which extends the object language with a type that reflects an internal tactic state, and exposes operations that act on the tactic state.
- We explain the mechanisms we use to reflect the syntax of dependent type theory efficiently, providing recursion and pattern matching over expressions. We also describe the quotation mechanism which mediates between the metalanguage and object language.
- We show how to use general support for monads and monadic notation in dependent type theory to define a tactic monad, which supports an imperative style of metaprogramming.

RIGHTSLINK

We explain how to extend the tactic monad with additional state, for example, to develop an interactive tactic language with SMT functionality, or to write automation that makes use of such extra state.
- We describe an extensible way of declaring attributes and assigning them to objects in the environment, with hooks for tactics that can be used to cache results.
- We explain how to install tactics written in our language for use in Lean's interactive tactic environment, allowing users to write tactic proofs in convenient and familiar ways.
- We describe a profiler and a debugging API. The latter can be used to implement a GDB-like debugger, with custom actions that are defined in the same (meta)programming language.

To make the case that our language is efficient, we consider tactics written in other tactic languages, and compare their performance to the performance of Lean tactics that offer similar functionality.

For effective metaprogramming, it is crucial to have efficient means of constructing object-language expressions in the metaprogramming language. Our language incorporates mechanisms for *direct reflection* [Barzilay 2006], in the sense that the same syntax and elaboration mechanisms used for the object language are available in the metaprogramming language. In this sense, it is an advance over the reflection mechanism adopted in Agda [van der Walt and Swierstra 2012] for similar purposes. (The current version of Agda, 2.5.2, has reflection mechanisms more like ours,[1] but the details have not yet been fully documented.)

Our approach to developing a metaprogramming language for interactive theorem proving is most similar to that adopted by Idris [Brady 2013; Christiansen 2014; Christiansen and Brady 2016]. In particular, our quotation mechanism is similar to that of Idris, except that our quotations can be elaborated at tactic execution time, which allows them to work in arbitrary metaprogramming contexts. Idris also exposes a monad that encapsulates a tactic state, and tactics can be called upon to synthesize an expression much like ours can. But, in contrast, our API provides direct access to the tactic state. This, for example, enables us to implement a lazy variant of the tactic monad, as described in Section 4.

An important part of our contribution lies in the methods we have developed to make such an approach performant, including efficient representations of quoted terms. We have shown that the approach scales to the needs of a general interactive theorem proving system with substantial automation. Finally, many of the other contributions listed above, including our attribute manager, our debugging API, and our means of extending the interactive tactic environment, are new.

The code examples presented here can be found in the supplemental files, and have been tested in Lean 3.2.0, which is available at leanprover.github.io. All benchmarks were also conducted with this release.

2 EXAMPLES

To convey a sense of our language, we consider Lean's `assumption` tactic, which attempts to solve the current goal by unifying the conclusion with an assumption in the current context. The implementation is presented in Fig. 1. The `find` tactic takes an expression `e` and a list of expressions `hs`. If there is an element `h` of `hs` whose type unifies with `t`, it performs the unification and returns `h`, and otherwise fails. The `assumption` tactic calls `find` with the target type of the current goal and the local context. If it finds a hypothesis in the context that unifies with the target, it uses that hypothesis to solve the current goal. We use the usual notation >> and >>= for sequencing operations in a monad, and <|> for the "or else" combinator in the `alternative` type class. Below we will also make use of the tactic combinator $\texttt{tac}_1$; $\texttt{tac}_2$, which runs $\texttt{tac}_2$ on every subgoal produced by $\texttt{tac}_1$.

[1] https://agda.readthedocs.io/en/v2.5.2/language/reflection.html

RIGHTSLINK

```
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"
```

Fig. 1. The `assumption` tactic.

In a Lean source file, whenever a term is expected, we can invoke a single tactic to construct that term using the keyword `by`. We can therefore use the `assumption` tactic as follows:

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

When processing this theorem, Lean uses the bytecode evaluator to execute the tactic with the current environment and goal, and constructs the relevant proof.

Notice that our programming language is just an ordinary functional programming language, with monadic notation, recursion over inductively defined data types, and so on. What makes it a *metaprogramming* language is that it includes an API for doing things like reading the current context and goal, unifing expressions, and constructing and manipulating the goal stack. Metaprograms in Lean can also query and modify the environment, and make use of the *attribute manager*, which keeps track of declarations in the environment that have been tagged with various attributes.

Indeed, most tactics intended for interactive use in Lean are currently implemented this way. The metaprogramming language also serves as a gateway to natively implemented automation. For example, user-facing variants of the `simp` tactic, which is used to simplify expressions in the current goal or local context, are defined using a natively implemented `simplify` procedure, which can be called by other metaprograms as well. In Section 8, we evaluate the performance of a `mk_dec_eq_instance` tactic that automatically synthesizes decision procedures for propositional equality for a large class of inductively defined data types.

For another example, we will describe the implementation of a more interesting piece of automation, namely, a robust simplification procedure that is useful in situations where straightforward term rewriting fails. In Sections 4 and 5, we will see that our metaprogramming language implements an SMT state that extends the tactic state with data structures that support E-matching and congruence closure. For a given goal, our tactic, `rsimp`, uses this infrastructure to build equivalence classes of provably equal terms, and then simplifies the goal by rewriting each term to a smallest one in its class.

The `collect_implied_eqs` function uses forward chaining, congruence closure and E-matching to infer equalities, and returns the congruence closure state as a result:

```
meta def collect_implied_eqs : tactic cc_state := ...
```

We will provide the definition later, in Section 4.

All inferred equalities and their proofs can be retrieved on demand from the `cc_state` data structure. The function `size` orders the terms in each equivalence class, counting the number of non-applicative subterms:

RIGHTSLINK

```
meta def size : expr → nat
| (app f a) := size f + size a
| _         := 1
```

The built-in function `cc_state.fold_eqc` provides a fold operation on the equivalence classes in the congruence closure state. Instead of writing `cc_state.fold_eqc ccs`, we can use the convenient "projector notation" `ccs.fold_eqc`; because `ccs` is a local declaration, the elaborator will not interpret it as a namespace, but infer the correct namespace `cc_state` from the type of `ccs`. The other two arguments of `fold_eqc` indicate the class and the starting point for the fold. We can now define the `choose` function that traverses the equivalence class of `e` and selects a smallest term in the class:

```
meta def choose (ccs : cc_state) (e : expr) : expr :=
ccs.fold_eqc e e $ λ (best_so_far curr : expr),
  if size curr < size best_so_far then curr else best_so_far
```

As in Haskell, the notation `f $ g t` is an alternative way of writing `f (g t)` that reduces the number of parentheses. With these definitions, `rsimp` can be implemented in just a few lines. It first constructs the congruence closure using `collect_implied_eqs`, and then calls a top-down simplifier to rewrite the target of the current goal, replacing subterms by their minimal representatives.

```
meta def rsimp : tactic unit :=
do ccs ← collect_implied_eqs,
   try $ simp_top_down $ λ t, do
      let root := ccs.root t,
      let t'   := choose ccs root,
      p        ← ccs.eqv_proof t t',
      return (t', p)
```

The `simp_top_down` function is written in the metaprogramming language, and uses the built-in simplifier to rewrite the target in a top-down manner. It expects a single argument: a tactic that takes an expression `t` as input, and returns a pair `(t', p)` consisting of a simplified expression `t'` and an expression `p` representing a proof of `t = t'`.

For an example of how `rsimp` works, suppose we have an environment with the following constants and properties:

```
constants (f : nat → nat → nat) (g : nat → nat) (p : nat → nat → Prop)
axioms (fax : ∀ x, f x x = x) (pax : ∀ x, p x x)
```

Suppose `fax` is marked as a simplification rule. Then the following proof works:

```
example (a b c : nat) (h₁ : a = g b) (h₂ : a = b) : p (f (g a) a) b :=
by rsimp; apply pax
```

Simplifying using the hypotheses h_1 and h_2 and axiom `fax` as rewriting rules would not work, since we would rewrite the goal to `p (f (g b) b) b` or `p (f (g (g b)) (g b)) b`, depending on the order these rewriting rules are applied. In both cases, the tactic `apply pax` would fail since the goal is not an instance of `p ?x ?x`.

In contrast, `rsimp` infers that the following terms are all equal, using `collect_implied_eqs`, and replaces any occurrence of them with `b`, which is a smallest element of this equivalence class.

```
{g b, f (g a) a, f (g a) (g a), g a, b, a}
```

For another example, consider an environment with a constant `p` as above and a `max` operation on the natural numbers which is associative, commutative, idempotent. Suppose further that `max` satisfies `max a (a + b) = a + b` for every `a` and `b`. Using `rsimp` reduces the goal in the next example

RIGHTSLINK

to `p d d`. Once again, the built-in simplifier would get stuck, because it fails to rewrite the term in such a way that the cancellation law applies.

```
example (a b c d : nat) : d = max c (a + b) →
                          p (max a (max d (b + a))) (max d (a + b)) :=
by intros; rsimp; apply pax
```

Many optimizations are possible. For example, the implementation of `rsimp` above recomputes the size of subterms without using a cache. If we were to profile its behavior using the tools described in Section 7, the profiler would enable us to detect this bottleneck. It is entirely straightforward, however, to implement a caching version of the procedure in our language. In fact, Lean's standard library includes a more sophisticated implementation of `rsimp` which works along these lines.

3 THE FRAMEWORK

Lean is based on a version of the Calculus of Inductive Constructions [Coquand and Huet 1988; Coquand and Paulin 1990]. Specifically, the logical framework has dependent function types (Pi types), a hierarchy of non-cumulative universes, inductively defined types, and quotient types [Avigad et al. 2017; Constable 1998]. The bottom universe, denoted `Prop`, is impredicative, and proof irrelevant in the sense that given `p : Prop` and `s t : p`, the kernel treats `s` and `t` as definitionally equal. The kernel implements a minimal form of inductive types [Dybjer 1994] with eliminators that allow only primitive structural recursion. In contrast to systems like Coq [Bertot and Castéran 2004] and Agda [Norell 2008], more elaborate forms of recursion are compiled down to these eliminators [Goguen et al. 2006], making the kernel smaller and more reliable.

Function definitions are immediately compiled to bytecode that can be executed by Lean's virtual machine. The virtual machine implements an eager evaluation strategy, but lazy evaluation can be simulated using thunks. Since it is not possible to define cyclic data structures in Lean, the virtual machine uses reference counting for memory management. Computations with natural numbers and integers are performed natively using machine integers for small numbers, and using big integers from the GNU multiple precision arithmetic library (GMP) for large numbers. The bytecode compiler also replaces the type `array α n` defined in the standard library with a native implementation based on the approach suggested by Baker [1991], with one extra optimization: if the reference counter for the array is 1 at run time, the array is destructively updated without any additional overhead.

Proofs and types are erased during compilation. Given a type `α` and predicate `p : α → Prop`, we can define the subtype `{x // p x}`. An element of this subtype is a pair consisting of an element `x : α` and a proof of `p x`. Since this proof is erased during compilation, the subtype merely represents a statically enforced contract. Similarly, we can define a quotient type from a base type and an equivalence relation. Functions on such a quotient type are compiled to functions on the base type, constrained to respect the equivalence relation.

Recall that our goal is to use dependent type theory as a metaprogramming language to construct expressions of dependent type theory itself. We achieve this by adding extra *metaconstants* to the language that, from the point of view of the axiomatic foundation, are simply opaque constants. When the virtual machine evaluates an expression, it associates these extra constants to data types and procedures that are implemented internally, as part of Lean's underlying C++ code base. For example, one such metaconstant, `tactic_state`, represents the internal elaborator state in the context where the tactic is invoked. Another, `environment`, represents a full Lean environment with declared constants and defined objects.

The metaprogramming API provides access to internal functions that operate on these. The API also includes a type constructor, `rb_map`, which provides an efficient implementation of red-black

RIGHTSLINK

```
inductive level
| zero   : level
| succ   : level → level
| max    : level → level → level
| imax   : level → level → level
| param  : name → level
| mvar   : name → level
```

```
inductive expr
| var    : nat → expr
| lconst : name → name → expr
| mvar   : name → expr → expr
| sort   : level → expr
| const  : name → list level → expr
| app    : expr → expr → expr
| lam    : name → binfo → expr → expr → expr
| pi     : name → binfo → expr → expr → expr
| elet   : name → expr → expr → expr → expr
```

Fig. 2. The type expr for reflecting expressions in dependent type theory.

trees with arbitrary data and keys. We make no attempt to keep the API minimal; our framework makes it easy to expose procedures written in native C++, and we do this freely whenever it will make the language more performant or convenient.

As we have already noted, definitions that make use of such extensions are tagged with the keyword `meta`. Metadefinitions are also allowed to perform arbitrary recursive calls, but otherwise they look much the same as ordinary definitions in the object language.

The higher-order nature of the CIC makes it easy to define a generic monad structure and extensions, and Lean's parser supports notation for monad operations via type class inference. The standard library straightforwardly defines all the usual instances, such as the option (maybe) monad, the list (nondeterministic) monad, the state monad, and the either (exception) monad. The tactic monad is simply another instance, defined in terms of the `tactic_state` primitive. It combines the state and either monads, allowing for the possibility that tactics can fail. The tactic monad is also an instance of the `alternative` type class, which means that it supports the `<|>` operator.

Tactics can be fallible, which is to say, they can construct expressions that are not well typed. Soundness is guaranteed by the fact that every object is checked by a small, trusted kernel before it is added to the environment.

3.1 Working with Expressions

Since metaprogramming involves using expressions in the metaprogramming language to describe and construct expressions in the object language, it is important to have convenient ways of passing between the two.

Names are organized into hierarchical namespaces in Lean; for example, `nat.succ_ne_zero` refers to the theorem `succ_ne_zero` in the `nat` namespace. The standard library defines a data type `name` that reflects these names. Tactics that construct expressions or access information in the environment use this data type. The parser provides special support to construct names: the expression `` `nat.succ_ne_zero `` is parsed to a suitable expression of type `name`. If we use two backticks, as in `` ``succ_ne_zero ``, then Lean resolves the name against the environment at parse time, and returns the full name for the corresponding object. So `` ``succ_ne_zero `` is parsed as the same expression as `` `nat.succ_ne_zero ``, provided that the `nat` namespace is open and `nat.succ_ne_zero` is defined. The parser reports an error if it cannot resolve the name.

More interestingly, Lean defines a type `expr` that reflects the internal representation of expressions in dependent type theory, as depicted in Fig. 2. Lean's type universes, called `Sort 0`, `Sort 1`, `Sort 2`, ... are parameterized by the `level` type, also depicted in Fig. 2. Because definitions can

RIGHTSLINK

be polymorphic over universes, the definition includes parameters that vary over the levels. The operations `max` and `imax` are used to express constraints that arise from the underlying logical rules, and metavariables (`mvars`) are used to solve constraints that arise during the elaboration process. `Prop` and `Type` are just syntactic sugar for `Sort 0` and `Sort 1` respectively, and `Type u` is syntactic sugar for `Sort (u+1)`.

Lean uses a *locally nameless* approach to deal with variables [McBride and McKinna 2004]: bound variables (`vars`) are essentially de Bruijn indices, whereas local constants (`lconsts`, also known as free variables) in a local context have both a user-facing name and unique internal name. As with levels, metavariables (`mvars`) are used to solve implicit variables, or placeholders, that arise in the elaboration process. Metavariables have a unique internal name and a type. Otherwise, the constructors are a straightforward reflection of the core syntax of dependent type theory: an expression is either a basic sort (e.g. `Sort u`), a constant (e.g. `nat`), an application, a lambda expression (e.g. `λ (x : α), s`), a Pi type (e.g. `Π (x : α), β`), or a let expression (e.g. `let x : α := t in s`). In Lean, `Π (x : α), β` can also be written as `∀ (x : α), β`.

When a function takes an argument that can generally be inferred from context, Lean allows us to specify that this argument should, by default, be left implicit. This is done by putting the arguments in curly brackets:

```
cons : ∀ {α : Type u}, α → list α → list α
```

Similarly, we use square brackets instead of curly brackets to inform the system that arguments should be inferred by type class resolution. The type `binfo` stores these binder annotations.

```
add : ∀ {α : Type u} [has_add α], α → α → α
```

Using the same mechanisms that work with other inductive types, users can define functions and tactics using recursion and pattern matching on the type `expr`. For example, we can easily count the number of arguments in a function application as follows:

```
meta def num_args : expr → nat
| (app f a) := num_args f + 1
| e         := 0
```

For efficiency, the virtual machine uses the internal Lean representation for expressions, and the constructors and the recursor for `expr` are interpreted by the virtual machine with internal procedures that carry out the corresponding operations. The `expr` type is a faithful representation of the internal data type; in this sense, we have *reflected* Lean expressions into the object language.

Lean's metaprogramming language provides a *quotation* mechanism to construct expressions: if `t` is any expression, `` `(t) `` denotes the corresponding object of type `expr`. In the next example, the tactic after `do` creates an expression and applies it to the current goal.

```
example : true ∧ true :=
by do apply `(and.intro trivial trivial)
```

Here the expression `and.intro trivial trivial` is elaborated *when the metaprogram is defined*, and the resulting tactic is then executed on the current goal.

The metaprogramming language also provides us with a convenient way to embed an expression that will be elaborated to an object level expression *when the metaprogram is run*. To that end, the language implements a type `pexpr` that reflects an internal representation of Lean *pre-expressions*. Roughly, these are raw expressions that have not been elaborated yet. Any quotation of the form `` ``(t) `` is intercepted by the parser, which constructs the raw expression corresponding to `t`. What makes pre-expressions especially useful is that they can contain *antiquotations*, namely, values of type *expr* that should be inserted when a metaprogram is executed. These are indicated by a double

RIGHTSLINK

percent sign. For example, when executed, the metaprogram below introduces the hypothesis `p` with name `h`, stores the expression for the local constant named `h` in `e`, substitutes the value of `e` into the pre-expression `or.inl %%e`, elaborates the result to an expression, and applies it.

```
example (p : Prop) : p → p ∨ false :=
by do e ← intro `h, to_expr ``(or.inl %%e) >>= apply
```

Alternatively, we can use the `refine` tactic, which expects a pre-expression as its argument:

```
example (p : Prop) : p → p ∨ false :=
by do e ← intro `h, refine ``(or.inl %%e)
```

The `refine` tactic internally calls the `to_expr` function to elaborate its argument. Note that, in either case, the object named `h` does not come into existence until the tactic is executed. The implicit arguments to `or.inl` (namely, left and right sides of the disjunction), are also synthesized when the tactic is executed, using both `e` and the goal `p ∨ false`, which are available at that time.

To handle antiquotations, the parser desugars an expression like ``` ``(or.inl %%h) ``` to ``` pexpr.subst ``(λ h, or.inl h) h ```, which causes the virtual machine to carry out the desired substitution at runtime. Antiquotations can also be used in single-backticks, though that often requires providing additional information that cannot be inferred at parse time. For example, below we use the `@` annotation to make the arguments to the or-introduction rule explicit:

```
example (p : Prop) : p → p ∨ false :=
by do e ← intro `h, t ← infer_type e, apply `(@or.inl %%t false %%e)
```

We explicitly infer the left disjunct, `p`, when the tactic is executed, and insert it into the quoted expression then. The expression after the `do` is elaborated in an empty context; the local constant `p` only comes into existence when the tactic is executed.

When using the double-backtick notation ``` ``(t) ```, names occurring in `t` are resolved when the tactic is parsed, as with the notation ``` ``n ``` for names. To allow names to be resolved when a tactic is executed, Lean provides yet a third option, with the use of triple backticks. In the following example, the name `h` in `or.inl h` is correctly assigned to the local constant that is introduced by the `intro` tactic:

```
example (p : Prop) : p → p ∨ false :=
by do intro `h, refine ```(or.inl h)
```

Using ``` ``(or.inl h) ``` instead would produce an error, since `h` does not denote anything when the tactic is defined.

Of the three forms of quotation just described, only the first, most restricted one is available in Idris [Christiansen 2014].[2] Such quoted `expr`s can also be used for pattern matching. For example, the code on the left hand side of Fig 3 is just syntactic sugar for the one on the right hand side. As in Idris, implicit arguments in the quoted term are ignored during matching.

By default, Lean compiles pattern matching to recursors such as

```
nat.cases_on : ∀ {C : nat → Sort u} (n : nat), C zero → (∀ a, C (succ a)) → C n
```

However, this approach is quite inefficient for matching name and string literals since it would produce several nested recursor applications to match even simple name literals. We avoid this inefficiency by compiling them to `if-then-else` expressions instead of recursor applications.

[2] As described in Christiansen and Brady [2016], Idris' quotation notation was later overloaded so that it can also return pre-expression representations, but the quoted term is still elaborated immediately. We cover the same functionality with a coercion from `expr` to `pexpr`.

RIGHTSLINK

```
meta def is_not : expr → option expr
| `(not %%a)      := some a
| `(%%a → false) := some a
| _               := none
```

```
meta def is_not : expr → option expr
| (app (const ``not _) a)      := some a
| (pi _ _ a (const ``false _)) := some a
| _                            := none
```

Fig. 3. Pattern matching with quoted terms.

With the idioms above, `intro `n` denotes a tactic that introduces a variable with the name `n`, and we can write `to_expr ```(e) >>= apply` to denote the tactic that applies expression `e` to the current goal. In an interactive theorem proving environment, it is convenient to blur the distinction between object language and metalanguage and write `intro n` and `apply e` instead. In Section 6, we will show how to embed tactics in a simple front end that supports these more convenient forms of expression, which avoids the use of the cumbersome ```(...) syntax. At the same time, by maintaining a distinct type for object language expressions and giving our metaprogramming language a rich set of primitives for manipulating them, we provide users with the ability to carry out complex operations on syntax in natural and straightforward ways.

3.2 Tactics and the Tactic State

Lean's `tactic` monad is defined in Lean itself, relying only on the primitive `tactic_state` type. Tactics are deterministic: they produce either a single new state, or fail. The `<|>` combinator supports backtracking: if `t1` and `t2` are tactics, `t1 <|> t2` is the tactic which executes `t1` and then backtracks and tries `t2` if `t1` fails. The `tactic` monad is an instance of the generic `interaction_monad` construction that combines the state and either monads and represents potentially unsuccessful, stateful, interactive commands. Its `exception` constructor stores an (optional) error message, (optional) position information, and the state at the time of failure. We store the error message in a thunk for performance, since the error message should be evaluated lazily when using the `<|>` combinator.

```
meta inductive result (state : Type) (α : Type)
| success   : α → state → result
| exception : option (unit → format) → option pos → state → result

meta def interaction_monad (state : Type) (α : Type) :=
state → result state α

meta def tactic := interaction_monad tactic_state
```

The primitive (meta) type `tactic_state` represents the internal elaborator state. This is a purely functional data structure, so we can easily store snapshots. (The use of such snapshots is implicit, for example, in the implementation of `<|>`.) The tactic state contains the `environment`, the goal stack, and the metavariable context. The goal stack is essentially a "to do" list of goals, and each goal is represented by a metavariable. For example, the metavariable `?m : q → p ∧ q` can represent the following goal:

```
(p q : Prop) (h : p) ⊢ q → p ∧ q
```

The local context `(p q : Prop) (h : p)` for `?m`, also known as the set of hypotheses, is stored in the metavariable context in a mapping from metavariables to local contexts. We use the notation `ctx ⊢ ?m : q → p ∧ q` to denote the local context for `?m`. The metavariable context also stores the

RIGHTSLINK

partial assignment of terms to metavariables. The local context is a collection of local declarations of the form `h : t` or `h : t := v`. Each local declaration has an internal unique internal name (uid), a user-facing name `h`, its de Bruijn level, a type `t`, and an optional value `v` for declarations that correspond to let-bindings. We store in each local constant the uid of its local declaration.

The local context is implemented with three mappings, named `uid_to_ldecl`, `dbl_to_ldecl`, and `name_to_ldecls`. The mapping `uid_to_ldecl` from uid to local declaration allows us to retrieve the local declaration for a local constant in $O(\log n)$ time, where n is the size of the local context. The mapping `dbl_to_ldecl` from de Bruijn levels to local declarations is used to retrieve the last local declaration and all local declarations that occur after a given declaration in the local context. Finally, the mapping `name_to_ldecls` from names to lists of local declarations is used for performing name resolution, and retrieving local declarations by facing name.

The primitive (meta) type `environment` contains all definitions, lemmas, axioms and inductive datatype declarations. The environment also contains front-end extensions in which additional data is stored, like notation declarations. Lean provides primitives to access and update the environment, such as the following:

```
get              : environment → name → exceptional declaration
add              : environment → declaration → exceptional environment
constructors_of  : environment → name → list name
```

The functions `get` and `add` can be used to retrieve a declaration by name from an environment or to add a new declaration to it, respectively. In case of failure, they return a formatted error message. The function `constructors_of env n` returns all constructors for the inductive datatype named `n` in `env`.

The following primitive tactics retrieve and update the goal stack:

```
get_goals : tactic (list expr)
set_goals : list expr → tactic unit
```

With these two primitives, it is straightforward to implement a tactic, `focus tac`, which "focuses" the attention on the first goal. It replaces the goal stack with a stack containing only the first goal `g`, then executes `tac`, and updates the goal stack with the subgoals `new_gs` produced by `tac` and the other goals `gs`.

```
meta def focus {α} (tac : tactic α) : tactic α :=
do g::gs  ← get_goals, set_goals [g],
   a      ← tac,
   new_gs ← get_goals, set_goals (new_gs ++ gs),
   return a
```

The `done` tactic, which fails if the goal stack is not empty, can be defined as follows:

```
meta def done : tactic unit := do [] ← get_goals, return ()
```

It is also straightforward to define tactic combinators such as $\texttt{tac}_1\texttt{; tac}_2$, as well as `all_goals tac`, which applies `tac` on every goal on the goal stack.

We say the first goal on the goal stack is the *main goal*. Several primitive tactics act with respect to the main goal, such as these:

```
infer_type  : expr → tactic expr
unify       : expr → expr → tactic unit
mk_instance : expr → tactic expr
intro       : name → tactic expr
```


RIGHTSLINK

```
meta def mk_name_set_attr (attr_name : name) : tactic unit :=
do let t := `(caching_user_attribute name_set),
   let v := `({name     := attr_name,
               descr    := "name_set attribute",
               mk_cache := λ ns, return (name_set.of_list ns),
               dependencies := [] } : caching_user_attribute name_set),
   add_meta_definition attr_name [] t v,
   register_attribute attr_name

meta def get_name_set_for_attr (attr_name : name) : tactic name_set :=
do let cnst := expr.const attr_name [],
   attr ← eval_expr (caching_user_attribute name_set) cnst,
   caching_user_attribute.get_cache attr

run_cmd mk_name_set_attr `no_rsimp
attribute [no_rsimp] or.comm or.assoc
run_cmd get_name_set_for_attr `no_rsimp >>= trace -- {or.comm, or.assoc}
```

Fig. 4. Generating new attributes using metaprogramming.

The tactic `infer_type t` returns the type of `t` with respect to the environment and metavariable context in the `tactic_state` and the local context of the main goal. The tactic `unify t s` tries to unify the terms `t` and `s`, which may instantiate metavariables contained in `t` and `s`. Given an expression `t`, such as `comm_ring int`, the tactic `mk_instance t` tries to construct a term of type `t` using the built in type class resolution procedure. The tactic `intro h` replaces the main goal `ctx ⊢ ?m : (∀ a : t, s[a])` with a new goal `ctx (h : t) ⊢ ?m' : s[h]`, and updates the partial assignment of metavariables in the context with `?m := (λ a : t, ?m'[h])`, where `?m'[h]` represents a *delayed abstraction*. The assignment for `?m'` may contain the local constant `h`, and the delayed abstraction indicates that `h` should be replaced with the de Bruijn variable with index 0 (i.e., `expr.var 0`). In Section 1, we used the `target` tactic to retrieve the type of the main goal. It could be defined as follows:

```
meta def target : tactic expr := do g::gs ← get_goals, infer_type g
```

Proofs of lemmas are elaborated asynchronously in Lean, and users can discharge other subgoals in different execution threads using the `prove_goal_async` tactic. This tactic is actually implemented in Lean using the `delay` function. Here, `task α` is the primitive (meta) type of parallel computations that will return a value of type `α`.

```
delay {α : Type} (f : unit → α) : task α
```

Another useful control tactic is `try_for n tac`, which executes tactic `tac` for at most `n` "heartbeats." The number of heartbeats is roughly a limit on the number of memory allocations performed by `tac`. This tactic hence implements a deterministic timeout. The tactic `try_for` is particularly useful when we want to build a tactic that tries many different tactics for short periods of time.

3.3 User-Defined Attributes

As an example of an advanced metaprogramming technique, we discuss the use of *user-defined attributes*. In Fig. 4, `mk_name_set_attr` declares a new attribute, with the given name, that caches a set of declarations that the attribute has been applied to. Metaprograms like `mk_name_set_attr` that

RIGHTSLINK

manipulate the current environment can be run on the top level via `run_cmd`, so such an attribute can be declared and used as shown in Fig. 4. The `trace` tactic, which is used to return output to users, is described in Section 7.

The `[no_rsimp]` attribute is in fact used in the library implementation of `rsimp` for blacklisting simplification lemmas, for which it relies on the fast membership checking of the native `name_set` type. The `name_set` will be cached as long as the set of declarations in the environment with that attribute or any of the `dependencies` attributes remains unchanged. (In our example, the `dependencies` field is left empty.) The implementation of `mk_name_set_attr` works by dynamically constructing a term of type `caching_user_attribute name_set`, adding it to the environment as a new declaration, and registering that declaration with the attribute manager.

When implementing the retrieval function `get_name_set_for_attr`, we encounter a new problem: given a reflected term that describes a value of type `caching_user_attribute name_set`, we would like to evaluate it at run time so that we can pass it to the following function:

```
caching_user_attribute.get_cache {α : Type} : caching_user_attribute α → tactic α
```

As can be seen in Fig. 4, we can do this with the `eval_expr` tactic:

```
eval_expr (α : Type) [reflected α] (e : expr) : tactic α
```

The type class parameter `[reflected α]` is necessary to provide a *safe* implementation: in order to check that `e` does indeed describe a term of type `α`, we also need a reflected description of the latter. (Indeed, a value of type `Type`, like `α`, does not even have a run-time representation.) The `reflected` type class is an opaque container for a term reflecting a known value. It is special in that the elaborator will synthesize the value of a parameter `[reflected α]` from the term passed for `α`, as long as this term is either closed (as in `get_name_set_for_attr`) or its free variables recursively have `reflected` instances. We have actually made implicit use of the latter feature in `mk_name_set_attr`, where we were able to use the local variable `attr_name` in the quotation because the elaborator was able to find an instance of `reflected attr_name`. We did not need to demand such an instance by parameter because for simple types such as `name` we can construct a universal instance via dependent pattern matching.[3]

```
meta def reflect {α : Type} (a : α) [h : reflected a] : reflected a := h
meta def has_reflect (α : Type) := Π (a : α), reflected a

meta instance name.reflect : has_reflect name
| anonymous        := reflect anonymous
| (mk_string  s n) := (reflect (λ n, mk_string  s n)).subst (name.reflect n)
| (mk_numeral i n) := (reflect (λ n, mk_numeral i n)).subst (name.reflect n)
```

Only for reflections of non-computational types such as sorts, propositions, and types containing these do we need to pass along an instance as with `eval_expr`.

3.4 Mechanics

A definition or lemma may contain multiple occurrences of the `by` construct. Moreover, a subterm synthesized by a tactic may occur in the goal of another tactic. For example, in the following simple lemma the proof term for `2*n > 0` produced by the tactic `arith` occurs in the main goal for the tactic `intros; apply pax`.

[3]The `has_reflect` type class is comparable to the `Quotable` interface in Idris, which however returns the equivalent of `expr` instead of our special, dependently-typed `reflected` type.

RIGHTSLINK

```
constant (p : ∀ n, n > 0 → Prop)
axiom (pax : ∀ n h, p (2*n) h)
```

```
lemma ex (n : nat) :
  n > 1 → p (2*n) (by arith) :=
by intros; apply pax
```

When our elaborator encounters a subterm `by tac`, it substitutes the subterm with a fresh metavariable `?m`, elaborates the tactic `tac` and compiles it into bytecode, and then stores the result into a mapping `mtacs` from metavariables to tactics. We remark that the tactic `tac` is *not* executed immediately. Before executing `mtacs[?m]` for some metavariable `?m`, the elaborator synthesizes any metavariable `?m'` in `mtacs` occurring in the type of `?m` or its local context. This mechanism guarantees that the tactic `arith` is executed before `intros; apply pax` in the example above.

Large definitions and lemmas containing many nested `by tac` subterms are frequently used in the standard library, but they seldom depend on each other as in the previous example. However, even when there is no dependency, Lean users found it useful to have *checkpoints* in the elaboration process where pending tactics in `mtacs` are executed. For example, when elaborating `let x := v in s` any tactic contained in `v` is executed before we elaborate `s`. In this way, an unrecoverable elaboration error in `s` will not mask a tactic failure in `v`.

4 EXTENDING THE TACTIC STATE

We have seen that Lean implements a `tactic` monad that is a state monad of the internal data represented by the metaconstant `tactic_state`. When it comes to writing automation, it is often useful to keep track of additional state. A benefit of using a functional programming language for metaprogramming is that we can take advantage of standard methods of extending and transforming monads [Liang et al. 1995]. In this section, we describe some useful applications of such methods.

Our tactic states are deterministic, in that a tactic acts on the current state and produces a new state or fails. In contrast, in Isabelle's tactic framework, tactics produce a lazy *stream* of resulting states. This has the benefit that one can define a composition operation that backtracks and tries alternatives in the case of failure. For example, in Lean's ordinary tactic framework, the following proof fails:

```
example (p q : Prop) : q → p ∨ q :=
by intros; constructor; assumption
```

The `constructor` tactic locally chooses the left introduction rule for the disjunction and succeeds, but then the subsequent `assumption` tactic fails. In Isabelle, the analogous tactic script will succeed because `constructor` returns both the left and right result states, and the composition operation retains only the combinations that ultimately succeed.

We can simulate this behavior in Lean by defining a new tactic monad:

```
meta def lazy_tactic (α : Type) := tactic_state → lazy_list (α × tactic_state)
```

The `lazy_tactic` type constructor becomes a monad with the bind operation that (lazily) applies the second tactic to all possible outcomes of the first, and (lazily) joins the sequence of possible results:

```
meta def bind {α β} : lazy_tactic α → (α → lazy_tactic β) → lazy_tactic β :=
λ t₁ t₂ s, join (for (t₁ s) (λ p, t₂ p.1 p.2))
```

Ordinary tactics are lifted to lazy tactics which return either the empty lazy list if the tactic fails, or a singleton if it succeeds. "Running" a lazy tactic is then just a matter of applying the first element of the lazy list, failing if the list is empty.

Of course, another approach to threading nondeterministic choices is to use a continuation-passing style. For example, even in the ordinary tactic monad we can define a version of the

RIGHTSLINK

constructor tactic that takes a continuation, `cont`, to be applied after the tactic tries each of the possible constructors:

```
meta def constructor' (cont : tactic unit) : tactic unit := ...
```

Then the following would succeed:

```
example (p q : Prop) : q → p ∨ q := by intros; constructor' assumption
```

This approach provides automated procedures with more fine-grained control over a backtracking search.

An example that is more central to the long-term goals of the Lean project is an extension of the tactic monad to support data structures that are commonly used in SMT solvers, which incrementally keep track of facts and equations that are available in a context. In Lean, it is easy to define the `state_t` monad transformer, which extends any monad with additional state:

```
def state_t (σ : Type) (m : Type → Type) [monad m] (α : Type) : Type :=
σ → m (α × σ)
```

Lean's standard library defines an SMT extension of the `tactic` monad:

```
meta constant smt_goal : Type
meta def smt_state  := list smt_goal
meta def smt_tactic := state_t smt_state tactic
```

Here, `smt_goal` is an internally represented structure that extends the usual tactic goal with additional data structures, including configuration information, a database of facts, union-find data structures to represent classes of provably equal terms, and information relevant to other built-in decision procedures, e.g. for the theory of associative and commutative operators. The `smt_state` contains one `smt_goal` for each goal in the goal stack.

The smt tactic `smt_tactic.intro (h : name) : smt_tactic unit` invokes the regular `intro` tactic, and updates the internal data structures stored in the `smt_goal`. For example, if the new hypothesis is of the form `h : a = b`, then the equivalence classes for `a` and `b` are automatically merged.

Any tactic that does not change the set of hypotheses can be automatically lifted to an smt tactic. Tactics that change the set of hypotheses have to be modified to preserve the validity of the additional data types. One possibility, albeit crude and inefficient, is to simply forget the `smt_goal` state and reconstruct it from scratch.

Constructing proofs in the `smt_tactic` monad, whether automatically or interactively, feels very different from constructing proofs in the ordinary `tactic` monad. The approach involves adding facts to the local context until the conclusion of the goal becomes immediate. The Lean `smt_tactic` framework already includes tactics to perform fundamental operations like forward chaining and congruence closure, and additional decision procedures will be added over time.

We now consider the `noalias` separation logic example from Gonthier et al. [2013], also used in Ziliani et al. [2015]. In this example, we have a type `heap`, which consists of finite maps from pointers (type `ptr`) to values (type `val`). We write $h_1 \bullet h_2$ for the disjoint union of h_1 and h_2, and $x \mapsto v$ for the singleton heap containing only the point `x` storing the value `v`. We use the proposition `is_def h` to denote that `h` is well defined. These operators satisfy the following three properties.

```
hcomm    : ∀ x y, x • y = y • x
hassoc   : ∀ x y z, (x • y) • z = x • (y • z)
hnoalias : ∀ h y₁ y₂ w₁ w₂, is_def (h • y₁ ↦ w₁ • y₂ ↦ w₂) → y₁ ≠ y₂
```

We can use the `smt_tactic` monad to prove the following example, considered by Gonthier et al. [2013] and Ziliani et al. [2015].

RIGHTSLINK

```
attribute [ematch] hcomm hassoc hnoalias
example (h₁ h₂ : heap) (x₁ x₂ x₃ x₄ : ptr) (v₁ v₂ v₃ : val) :
  is_def (h₁ • (x₁ ↦ v₁ • x₂ ↦ v₂) • h₂ • (x₃ ↦ v₃)) → x₁ ≠ x₂ ∧ x₂ ≠ x₃ :=
by using_smt (intros; eblast)
```

The command `attribute` is used to tag `hcomm`, `hassoc` and `hnoalias` with `[ematch]`. By default, the `ematch` tactic instantiates universal lemmas and axioms tagged with this attribute. The `using_smt` tactic has type `smt_tactic unit → tactic unit`, and the `eblast` tactic is defined in the standard library as follows:

```
meta def eblast : smt_tactic unit := repeat (ematch; try close)
```

That is, it keeps applying E-matching until the goal is closed or no new lemma instance is produced. The tactic `congruence` available in Coq also implements a congruence closure algorithm that instantiates hypotheses that assert quantified equalities. However, it is not incremental, and each `congruence` tactic invocation has to reconstruct the state of the congruence closure graph from scratch. Moreover, users have no access to the equivalence classes computed by this tactic. For example, the tactic `collect_implied_eqs` used in Section 2 uses the tactic `to_cc_state` to retrieve the state of the congruence closure procedure:

```
meta def collect_implied_eqs : tactic cc_state :=
focus $ using_smt $ do
  add_lemmas_from_facts, eblast,
  (done; return cc_state.mk) <|> to_cc_state
```

Note that if the goal has been solved (i.e. `done` succeeds), the tactic just returns the empty state.

The `state_t` transformer is equally useful for automation implemented solely in Lean. In the next section, we will discuss the implementation of a superposition theorem prover that uses it to extend the `tactic` monad in a similar way.

5 APPLICATIONS

5.1 A Crush-Like Tactic

We now consider a running example used in the Chapter "Proving in the Large" of *Certified Programming with Dependent Types* [Chlipala 2011]. Fig. 5 defines a basic language of arithmetic expressions (type `exp`), an interpreter `eeval` for it, and two transformations: `times`, which scales up every constant in the expression, and `reassoc`, which rewrites an expression using associativity of addition and multiplication. We can prove basic properties about `times` and `reassoc` as follows:

```
attribute [simp] mul_add times reassoc eeval
lemma eeval_times (k e) : eeval (times k e) = k * eeval e :=
by induction e; rsimp

lemma reassoc_correct (e) : eeval (reassoc e) = eeval e :=
by induction e; rsimp; cases (reassoc e2); rsimp
```

The annotation `[simp]` instructs `rsimp` to use the lemma `mul_add`, that states that multiplication distributes over addition, and the automatically generated lemmas for each equation of the recursive functions `eeval`, `times` and `reassoc`. The tactic `cases (reassoc e2)` performs case analysis on the given term, i.e. it creates a subgoal for each constructor (`Const`, `Plus` and `Mul`). This extra step is necessary because of the nested `match`-expressions used to define `reassoc`. These proofs are already small, but we can make them even smaller by writing automation that searches for the hypotheses we need to induct and perform case analysis on. Fig. 6 contains a very simple tactic that performs

RIGHTSLINK

```
inductive exp : Type
| Const (n : nat) : exp
| Plus (e1 e2 : exp) : exp
| Mult (e1 e2 : exp) : exp

def eeval : exp → nat
| (Const n)    := n
| (Plus e1 e2) := eeval e1 + eeval e2
| (Mult e1 e2) := eeval e1 * eeval e2

def times (k : nat) : exp → exp
| (Const n)    := Const (k * n)
| (Plus e1 e2) := Plus (times e1)
                       (times e2)
| (Mult e1 e2) := Mult (times e1) e2
```

```
def reassoc : exp → exp
| (Const n)    := (Const n)
| (Plus e1 e2) :=
  let e1' := reassoc e1 in
  let e2' := reassoc e2 in
  match e2' with
  | (Plus e21 e22) := Plus (Plus e1' e21) e22
  | _              := Plus e1' e2'
  end
| (Mult e1 e2) :=
  let e1' := reassoc e1 in
  let e2' := reassoc e2 in
  match e2' with
  | (Mult e21 e22) := Mult (Mult e1' e21) e22
  | _              := Mult e1' e2'
  end
```

Fig. 5. Basic language of arithmetic expressions, an interpreter for it, and two transformations `times` and `reassoc`.

```
meta def try_list {α} (tac : α → tactic unit) : list α → tactic unit
| []      := failed
| (e::es) := (tac e >> done) <|> try_list es

meta def induct (tac : tactic unit) : tactic unit :=
collect_inductive_hyps >>= try_list (λ e, induction' e; tac)

meta def split (tac : tactic unit) : tactic unit :=
collect_inductive_from_target >>= try_list (λ e, cases e; tac)

meta def search (tac : tactic unit) : nat → tactic unit
| 0     := try tac >> done
| (d+1) := try tac >> (done <|> all_goals (split (search d)))

meta def nano_crush (depth : nat := 1) :=
do hs ← mk_relevant_lemmas, induct (search (rsimp' hs) depth)
```

Fig. 6. A simple search procedure.

this kind of search. The tactic `try_list` takes a tactic `tac` which is parametrized by a value of type α and a list of candidates. It tries `tac` for each element in the candidate list until one of them closes all subgoals. The tactic `induct` uses `try_list` to guess the hypothesis to induct on, and then applies the given tactic `tac`. The auxiliary tactic `collect_inductive_hyps` collects all hypotheses `h : t`, where `t` is an inductive datatype. We can define a similar tactic for guessing terms. The function

RIGHTSLINK

`collect_inductive_from_target` traverses the main goal searching for subterms we can perform case analysis on, using the `cases` tactic. The tactic `search` implements a very simple bounded search. Finally, our `nano_crush` tactic uses an auxiliary procedure `mk_relevant_lemmas` to collect the defining equations for all relevant functions occurring in the main goal, where a function is deemed "relevant" if it is defined in the current file or in an open namespace. This list of equational lemmas is stored in `hs`. The `nano_crush` procedure then guesses a variable to induct on, and tries to close all goals using `search` and `rsimp' hs`, where `rsimp'` is a variant of the `rsimp` tactic described before which allows us to specify which lemmas should be used. The default search depth for `nano_crush` is 1, and it can easily prove the lemmas `eeval_times` and `reassoc_correct`:

```
attribute [simp] mul_add
lemma eeval_times (k e) : eeval (times k e) = k * eeval e := by nano_crush
lemma reassoc_correct (e) : eeval (reassoc e) = eeval e := by nano_crush
```

The Lean standard library includes a more sophisticated implementation called `mini_crush` which uses a best-first search strategy.

5.2 A Superposition Prover

To show that our metaprogramming language scales to larger applications as well, we implemented a proof-of-concept superposition prover in Lean. It implements a standard superposition calculus [Nieuwenhuis and Rubio 2001] for first-order logic with equality, including term ordering, literal selection, subsumption, and demodulation. Going beyond standard inferences, we also implemented the AVATAR clause splitting scheme as described in Voronkov [2014]. This clause splitting scheme requires tight integration with a SAT solver, which we implemented in the metaprogramming language as well. We did not implement some important performance optimizations, such as indexing and AC redundancy elimination.

The superposition prover provides a tactic called `super`, which takes a list of lemma names as arguments. These lemmas are included in the proof search. For example, we can use `super` to prove that a right inverse in a monoid is also a left inverse:

```
example {α} [monoid α] [has_inv α] : (∀ x : α, x * x⁻¹ = 1) →
                                     ∀ x : α, x⁻¹ * x = 1 :=
by super with mul_assoc mul_one
```

The use of `with` here is an example of the parser extensions we will discuss in Section 6.

The general approach of superposition theorem proving is to first transform the problem into clause normal form, and then refute this (inconsistent) set of clauses by saturation, by inferring consequences and adding them to the set until we derive the empty clause, which represents falsity. Usually, a clause is a universally quantified disjunction of literals $\forall\ x_1\ \ldots\ x_n,\ \neg a_1 \lor \ldots \lor \neg a_i \lor b_1 \lor \ldots \lor b_j$. In `super` we represent clauses using implications instead: $\forall\ x_1\ \ldots\ x_n,\ a_1 \to \ldots \to a_i \to \neg b_1 \to \ldots \to \neg b_j \to$ `false`. This has the advantage that it allows us to avoid classical reasoning and produce constructive proofs for a large class of problems (including the example above), by replacing all occurrences of `false` with the original conclusion.

During saturation, inferences produce new clauses. The `super` tactic generates the proof terms for the new clauses immediately, and the clause data structure keeps track of both the formula and its proof, as well as the number of quantifiers and literals:

```
meta structure clause := (num_quants num_lits : ℕ) (proof type local_false : expr)
```

These newly-derived clauses are stored in the local context. We add an additional hypothesis for each derived clause using the `assertv` tactic. This is important since it keeps the proof terms small;

RIGHTSLINK

otherwise they could grow exponentially. The `intern_clause` function replaces the proof term in a derived clause by the hypothesis in the local context.[4]

```
private meta def intern_clause (c : derived_clause) : prover derived_clause :=
do hyp_name ← mk_clause_name c.id.to_nat,
   assertv hyp_name c.type c.proof,
   c.update_proof <$> get_local hyp_name
```

Most inferences in the superposition calculus require unification. The `super` tactic uses Lean's built-in procedure to compute the most general unifier. Since Lean's unifier operates on metavariables, we first instantiate quantifiers with fresh metavariables, and then re-introduce quantifiers for the unassigned metavariables at the end. The function `try_factor'` computes the clause where the two unifiable literals with indices i and j have been factored (that is, merged because they are equal after unification):

```
private meta def try_factor' (c : clause) (i j : nat) : tactic clause :=
do (qf, mvars) ← c.open_metan c.num_quants,
   unify_lit (qf.get_lit i) (qf.get_lit j),
   qfi         ← qf.inst_mvars, guard $ clause.is_maximal gt qfi i,
   (at_j, cs)  ← qf.open_constn j, hyp_i ← cs.nth i,
   let qf' := (at_j.inst hyp_i).close_constn cs,
   clause.meta_closure mvars qf'
```

The global proof search is organized using the `state_t` transformer around the `tactic` monad as described in the previous section. The state keeps track of which clauses have already been used for inferences (the active set), the clauses that still need to be used (the passive set), and other bookkeeping data.

```
meta structure prover_state :=
(active passive : rb_map clause_id derived_clause)
(newly_derived : list derived_clause) (prec : list expr)
(locked : list locked_clause) (sat_solver : cdcl.state)
...
meta def prover := state_t prover_state tactic
```

6 INTERACTIVE PROVING

As we have seen, we can execute arbitrary terms of the tactic monad using the `by` keyword. To help construct such tactic terms, we provide common syntactic sugar such as the `do` notation and quotation literals. While this *programmatic* view is good for defining new tactics, end users are accustomed to more *declarative* representations, such as a sequence of tactics, each with its own convenient syntax, that can be stepped through and inspected interactively.

In order to gain this level of convenience, we introduce a special variant of `by` that takes a single tactic name and parses its arguments according to syntax rules encoded in the tactic's signature. We have already seen it in action in Section 5:

```
by super with mul_assoc mul_one
```

This is desugared to the following regular term, in which the extra parentheses deactivate the special handling:

```
by (super [] [`mul_assoc, `mul_one])
```

[4]The actual implementation of `intern_clause` is slightly more complicated, since it also keeps track of the splitting assertions.

RIGHTSLINK

```
/-- An opaque type representing Lean's native parser state. -/
meta constant parser_state : Type
meta def parser := interaction_monad parser_state
/-- Parse an identifier and produce it as a quoted name. -/
meta constant ident : parser name
/-- Parse the given token. `tk` must be a registered token. -/
meta constant tk (tk : string) : parser unit
/-- Parse an unelaborated expression using the given right-binding power. -/
meta constant qexpr (rbp := std.prec.max) : parser pexpr
/-- Parse `with` followed by a list of identifiers. -/
meta def with_ident_list := (tk "with" *> ident*) <|> return []
meta def parse {α : Type} [has_reflect α] (p : parser α) : Type := α
```

Fig. 7. Exposing Lean's native parser as a monadic parser combinator. `*>` and `*` are notations for the applicative combinators `seq_right` and `many`, respectively. Note that e.g. `parse with_ident_list` is definitionally equal to `list name`, so that as far as the definition of `super` is concerned, the argument is just a list of names.

For executing a sequence of tactics, we may use `begin ... end` blocks:

```
begin intro h, refine or.inl h end
```

This proof is equivalent to the following proof from Section 3:

```
by do intro `h, refine ```(or.inl h)
```

Using `begin ... end` has the added benefit that Lean will record the proof state at the beginning of each tactic so that it can be inspected by editors.

Let us now take a look at the detailed signature of `super`:

```
meta def tactic.interactive.super (extra_clause_names : parse ident*)
              (extra_lemma_names : parse with_ident_list) : tactic unit
```

By default, when a parsing `by` or a `begin ... end` block the parser will look for interactive tactics in the namespace `tactic.interactive`. Users can switch to a different namespaces by using e.g. `begin[smt] ... end`, which will instead search the namespace `smt_tactic.interactive`.

When parsing the arguments of an interactive tactic, we handle parameters of type `parse p` specially by giving over control to the user-defined parser `p`. Parsers can be built from a few exported primitives, which are described in Figure 7, using the standard applicative and monadic combinators. We are planning to reuse the same parser monad to implement user-defined syntax at the expression level and commands at the top level.

We have used the parser interface to build a variety of tactics, such as a rewriter that accepts a list of rewrite steps:

```
lemma inv_eq_of_mul_eq_one [group α] {a b : α} (h : a * b = 1) : a⁻¹ = b :=
by rw [←mul_one a⁻¹, ←h, ←mul_assoc, mul_left_inv, one_mul]
```

To help users inspect the intermediate proof states of such tactics, we provide a helper tactic `save_info : pos → tactic unit` that stores the current proof state at a given position, so that it can then be queried and displayed by editors. This is the same tactic that is inserted when desugaring `begin ... end` blocks.

RIGHTSLINK

```
meta def trace_step (p : tactic_state → name → bool) (sz : nat) : vm nat :=
do curr_sz ← call_stack_size, guard (sz ≠ curr_sz),
   ts      ← ts_from_current_frame,
   fn      ← curr_fn,
   when (p ts fn) $ do {
       put_str $ "tactic state at " ++ to_string fn ++ "\n",
       put_str $ to_string ts,
   }, return curr_sz

@[vm_monitor] meta def my_vm_monitor : vm_monitor nat :=
{ init := 0, step := trace_step (λ s fn, fn = ``nano_crush.search) }

set_option debugger true
lemma eeval_times (k e) : eeval (times k e) = k * eeval e := by nano_crush
```

Fig. 8. A VM step tracer.

7 DEVELOPMENT TOOLS

Yet another advantage of reusing Lean as a metaprogramming language is the ability to use tools both for programs and tactics. For example, Lean has a profiler that polls the state of the virtual machine at regular intervals whose duration can be set by the user. The output summarizes the time spent in each function call:

```
380ms   100.0%   tactic.interactive.super
370ms    97.4%   super._lambda_5
360ms    94.7%   super.run_prover_loop._lambda_21
360ms    94.7%   super.run_prover_loop._lambda_13
...
```

Users can even access this information in the interactive editor interface, by using `set_option profiler true` and hovering over the relevant theorem, definition, or example command.

Lean also provides an API for debugging programs interactively. Rather than use a fixed debugger, we allow the full use of the programming language to configure the behavior of a virtual machine monitor. The approach is similar to the one used for interacting with the tactic state: we define a new opaque monad `vm` and expose operations in this monad to inspect the virtual machine internals. We first define a virtual machine monitor as a structure which maintains some state of type α, an initial value for that state, and a monadic action which computes the next state:

```
meta structure vm_monitor (α : Type) := (init : α) (step : α → vm α)
```

The implementation of the monitor API knows the structure of the `vm_monitor`. It uses the first field to initialize the internal state, and then invokes the `step` function on the state after each instruction executed by the VM. Fig. 8 shows the implementation of a simple execution tracer, `trace_step`, which is parametrized by a predicate `p`. If the call stack size has changed, it tries to find the first `tactic_state` object in the current stack frame using `ts_from_current_frame`. Then, it retrieves the name of the function being executed using `curr_fn`, and prints the state if `p ts fn` returns true. Then we use `trace_step` to trace the tactic state whenever the function `nano_crush.search`, defined in Fig. 6, is executed while synthesizing the proof for the `eeval_times` lemma. The output trace for this example will contain intermediate states such as this:

```
k n : nat ⊢ eeval (times k (Const n)) = k * eeval (Const n)
```


RIGHTSLINK

The above monitor is just an example of what is possible with the `vm_monitor` API. There are many modifications we could make to this simple version, including caching, trace summaries, and increased interactivity. For users for whom a simple debugger is sufficient, we have used the `vm_monitor` API to implement a standard command line interface debugger in the style of GDB. This debugger is included in the standard library of Lean.

Lean also offers helpful debugging facilities specific to metaprogramming as well. The tactic API includes a polymorphic `trace` function which can be used to output trace information from within the `tactic` monad. When writing a tactic `foo`, for example, metaprogrammers can use the command `declare_trace foo` to declare a new tracing option with that name. They can then design the tactic to output useful trace information conditionally, using `when_tracing `foo` as a guard. Both metaprogrammers and end users can then enable tracing by writing `set_option trace.foo true` in a Lean source file, for example, to gather additional information whenever `foo` does not behave as they expected.

8 EXPERIMENTAL EVALUATION

We have invested a good deal of effort to make our implementation performant, using efficient data-structures and caching, developing efficient quotation and anti-quotation mechanisms, and using aggressive optimizations in the bytecode compiler and evaluator. As noted in Section 3, all terms are checked by Lean's kernel before they are added to the environment, so any bugs in the compiler or evaluator do not compromise the soundness of the system. We also use the locally nameless approach described in Section 3.1 and we found that it greatly simplifies caching data-structures for operations such as inferring types, type class resolution, and so on. The goal of this section is to provide evidence that our implementation is practical and efficient.

A simple backtracking solver for equalities in commutative monoids was used in Malecha and Bengtson [2016] to compare the performance of Ltac and Rtac. For comparison, we implemented this solver in Lean as well. The solver proves an equality such as $(b * a) * c = a * (c * b)$ by iterative cancellation. In each step, we pick an element on the left side and rearrange the term so that the element is the left subterm of the function on the left side. For example, if we pick b, we would get $b * (a * c) = a * (c * b)$. We then do the same on the right side, again for all elements. After these two steps, all possible combinations of elements appear as the left subterms. For each of the combinations, we try to cancel the left subterms if they are equal.

Each of these rearrangement and cancellation operations is represented as an implication that can be used with the `apply` tactic. In Lean, we can prove all of these lemmas with a very short custom decision procedure using the SMT state. (Differing slightly from the Coq version, we used plain equality instead of setoid equality, since it is customary to use quotient types instead of setoids in Lean.)

```
meta def qf_monoid := using_smt (intros; eblast)
lemma plus_unit_p : a = b → o * a = b := by qf_monoid
lemma plus_assoc_p1 : a * (b * c) = d → (a * b) * c = d := by qf_monoid
lemma plus_assoc_p2 : b * (a * c) = d → (a * b) * c = d := by qf_monoid
lemma plus_comm_p : a * b = c → b * a = c := by qf_monoid
-- and similarly for the right side of the equation
lemma plus_cancel : a = c → b = d → a * b = c * d := by qf_monoid
```

The solver itself is a straightforward combination of the `<|>` combinator and the `applyc` tactic (Fig. 9). The `applyc n` tactic is a simple variant of the `apply` tactic, and is defined as `mk_const n >>= apply`. We compared the performance[5] of this tactic with the Ltac and Rtac versions[6] from

RIGHTSLINK

```
meta def iter_right :=
applyc ``plus_unit_c <|>
applyc ``plus_assoc_c1 >> iter_right <|>
applyc ``plus_assoc_c2 >> iter_right <|>
applyc ``plus_cancel >> reflexivity

meta def cancel :=
iter_left <|>
applyc ``plus_comm_p >> iter_left
```

```
meta def iter_left :=
applyc ``plus_unit_p <|>
applyc ``plus_assoc_p1 >> iter_left <|>
applyc ``plus_assoc_p2 >> iter_left <|>
iter_right <|>
applyc ``plus_comm_p >> iter_right

meta def solve : tactic unit :=
repeat $ reflexivity <|> cancel
```

Fig. 9. Backtracking solver for commutative monoids.

Malecha and Bengtson [2016] on the same benchmark set. In practice, we expect users to use the built-in simplifier or the SMT state to solve this kind of problem instead, so we also compared the performance of those two tactics. Figure 10 shows the results of the comparison: the Lean version of the solver is several times faster than the Rtac version, and the performance difference increases with the benchmark size. At size 10, the Lean version is just twice as fast, and at size 100 it is more than six times as fast. The built-in SMT automation is slightly faster than the backtracking cancellation solver; the simplifier is slightly slower, although still faster than the Rtac version.[7]

It is easy to optimize the hot spots in this solver without changing the general structure, or even the function signatures of the metadefinitions. Instead of tentatively calling `apply` and backtracking when unification fails, we can directly pattern match on the target and fill in the implicit arguments to the lemmas manually. This speeds up our implementation of the cancellation solver by a factor of 3.5, and it is then over 20 times faster than the Rtac version (for the benchmark of size 100).

```
meta def iter_left :=   -- and similarly for iter_right
do t ← target, match t with
| `(o * %%a = %%b) := apply `(@plus_unit_p %%a %%b)
| `((%%a * %%b) * %%c = %%d) :=
  (apply `(@plus_assoc_p1 %%a %%b %%c %%d) >> iter_left) <|>
  (apply `(@plus_assoc_p2 %%a %%b %%c %%d) >> iter_left)
| _ := iter_right <|> applyc `plus_comm_p >> iter_right
end
```

As a second benchmark, we compared a tactic that automatically derives type class instances. Lean's standard library contains the `mk_dec_eq_instance` tactic that automatically generates instances of the `decidable_eq` type class, witnessing that the equality relation on a type is decidable. In order to prove that $x = y$ is decidable, we case split on x and y using the `cases` tactic, and then solve each of the resulting subgoals. Idris also has such a tactic in its Pruviloj library [Christiansen and Brady 2016]. We compared the Lean and Idris[8] versions on simple datatypes: booleans, lists, and

[5] All benchmarks were performed on a Linux machine running NixOS, with a 4.2GHz Intel i7-6700K CPU and 16G RAM.
[6] We used Coq 8.5 and version 1.0.2 of `mirror-core` for the comparison.
[7] Malecha has implemented a similar tactic in Mtac, and reports that the performance is significantly slower than that of Rtac; see https://gmalecha.github.io/reflections/2016/experimenting-with-mtac. Unfortunately, we were unable to get the code to run successfully on our examples.
[8] We used Idris 0.99 for the comparison.

RIGHTSLINK

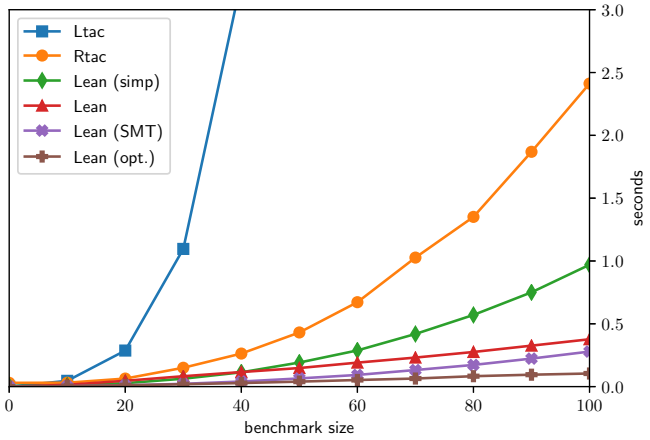

Fig. 10. Runtime on the monoid cancellation benchmark. *Ltac* and *Rtac* are the implementations presented in Malecha and Bengtson [2016]. *Lean* refers to the same solver implemented in Lean as described above. In addition, we also compared other possible implementations of a commutative monoid equation tactic in Lean: *(simp)* refers to Lean's built-in simplifier, *(SMT)* to the integrated SMT state with E-matching and congruence closure, and *(opt.)* to the optimized version of the cancellation solver. The measurements include both proving and type-checking time.

length-indexed vectors. The following table shows the runtime[9] and peak memory consumption on these benchmarks. In all examples, Lean's `mk_dec_eq_instance` tactic is several orders of magnitude faster, and uses less memory.

	Boolean	List	Vector
Idris	0.3s (520M)	10.7s (2516M)	18.4s (4018M)
Lean	0.002s (60M)	0.007s (61M)	0.012s (63M)

An important performance optimization in the Lean VM concerns the built-in `expr` type. The inductive data type for expressions, shown in Fig. 2, is replaced in the VM by a thin wrapper around the C++ type. Passing an expression from the VM to native code and back only incurs the small constant overhead of constructing and destructing the wrapper. By contrast, Idris performs a deep copy to transfer expressions from the metalanguage to the implementation language and vice versa.

To measure the overhead introduced by the deep copy, we benchmarked the weak head normal form operation. We construct a (unary) natural number of a given size and then iterate the WHNF. With Lean's approach the main cost lies in the creation of the numeral—the WHNF is computed in constant time as it only needs to check that the root symbol is a constructor and can then return. Fig. 11 compares the runtime difference between 1 and 10 iterations on several numeral sizes. In Lean, there is little difference. However in Idris, the expression is copied by the WHNF operation in every iteration, so the runtime for 10 iterations is significantly larger than for 1.

The straightforward encoding of interactive tactic scripts as programs in the metalanguage is reasonably efficient as well. To evaluate the performance, we imported automatically generated proof traces as tactic scripts. The GAPT system [Ebner et al. 2016] contains an experimental proof

[9]For Idris, we measured the runtime of `idris --check` and subtracted the runtime for an empty file (2.15s) to eliminate startup overhead. For Lean, we used the `--profile` command-line switch.

RIGHTSLINK

```
meta def make : ℕ → expr
| 0 := `(nat.zero)
| (i+1) := `(nat.succ %%(make i))

meta def bench (iters size : ℕ) :=
whnf_loop iters (make size)
```

```
meta def whnf_loop : ℕ → expr → tactic
    unit
| (i+1) t :=
    do nf ← whnf t,
       whnf_loop i nf
| 0     _ := return ()
```

size x iterations	100x1	100x10	200x1	200x10	1000x1	1000x10	10^6x1	10^6x10
Idris	90ms	590ms	300ms	1400ms	1810ms	6630ms	-	-
Lean	0.1ms	0.1ms	0.2ms	0.2ms	1.2ms	1.2ms	540ms	526ms

Fig. 11. Performance effect of the deep-copying optimization for expressions.

export from its own sequent calculus to Lean tactic scripts. The exported theory is a shallow embedding, and the proof traces consist mainly of `apply` and `intro` tactics, such as the following two lines taken from the middle from an exported proof:

```
apply (gapt.lk.ImpLeftRule hyp.h_269), intros hyp.h_286,
apply (gapt.lk.LogicalAxiom hyp.h_13 hyp.h_286), intros hyp.h_286,
```

We exported a proof with 531 inferences, which resulted in a tactic script with 1062 tactics. Lean takes 3.25s to compile it, whereas a literal translation to a Coq file takes 2.09s to compile in Coq 8.6. As described in Section 6, the interactive tactic framework inserts extra function calls to gather information for display in the editors, and this accounts for a large part of the difference in performance. Directly generating a binary >>-tree of apply and intro tactics improves the runtime to 2.08s. However, elaborating applications of >> requires higher-order unification: we need to infer the monad. With a monomorphized version of >> for tactics, the runtime shrinks to 1.63s. Additionally, Lean's elaborator supports multiple elaboration modes; we can annotate any function with an attribute that tells the elaborator how to elaborate its applications. The `[elab_simple]` attribute selects the fastest elaboration strategy, which does not propagate the expected type of the application to the arguments:

```
@[elab_simple] meta def tactic_seq {α β} (t₁ : tactic α) (t₂ : tactic β) :=
do t₁, t₂, return ()
```

With this last optimization, the runtime is now at 1.55s (a 52% improvement over the initial version). Most of these optimizations could be easily implemented in the elaboration procedure for interactive tactics, should this inefficiency turn out to be a problem. These experiments suggest, however, that the overhead introduced by type classes, monads, and so on is not a significant problem in practice.

9 RELATED WORK

The challenge of developing suitable metalanguages to support interactive theorem proving is as old as the pursuit of interactive theorem proving itself. ML was originally designed as a metalanguage for implementing theorem provers in the LCF framework, with the object logic represented directly in that language and user interaction taking place within an interpretive ML environment [Gordon 2000]. Contemporary theorem provers like HOL4 and HOL Light still operate in that mode.

The desire for better infrastructure for working with the object logic — for example, the desire for parsers and proof languages that are designed specifically for that purpose — pushes for a

RIGHTSLINK

separation of the interaction and implementation languages. Users of contemporary systems like Coq, Isabelle, and Lean need not even be aware of the language that implements the core logic (in these cases, OCaml, Poly/ML, and C++ respectively). But developing effective libraries for theorem proving and verification requires developing not just bodies of definitions and theorems but also procedures that make it possible to use them effectively, and this has encouraged the development of tactic languages that recapture the ability to develop such procedures effectively.

Coq's Ltac [Delahaye 2000, 2002] is perhaps the most widely used tactic language today. It is a domain specific language, with constructs tailored specifically to support common theorem-proving tasks, including nonlinear pattern matching over expressions in the context and goal, and backtracking search. Ltac does not make a sharp distinction between object and metalanguage expressions, as we do; patterns are expressed by writing ordinary Coq expressions, sometimes with holes or extra variables. Although the language has been extended over the years, however, it lacks the flexibility and performance of a full-fledged programming language. Ltac's backtracking combinators are easily defined in our approach, but we have given up some of the convenience of Ltac's pattern-matching primitives in favor of such flexibility and performance. Isabelle's new tactic language, Eisbach [Matichuk et al. 2016], is similar to Ltac in that it is a domain-specific language, with primitives that are well adapted to the Isabelle environment.

Although our language is dependently typed, the type system does not play a strong role in specifying tactic behavior. For example, knowing that a term `t` has type `tactic expr` tells us that we can use it to act on the tactic state and that it returns an expression, but it does not tell us anything about what it does to the tactic state or what kind of expression it returns. In recent years, a number of researchers developed typed tactic languages that specify tactic behavior more precisely [Baelde et al. 2014; Pientka 2010; Stampoulis and Shao 2010]. The Mtac language [Ziliani et al. 2015], and its successor, MetaCoq [Ziliani et al. 2017], extend the object language of Coq with a monad ○A, read "maybe A," for tactics that attempt to construct an element of type A. They extend the base logic with general recursion, syntactic pattern matching, exceptions, and unification, as well as a primitive mutable arrays. At runtime, Coq's virtual machine translates Mtac primitives to the appropriate internal actions on the tactic state, as is the case with our approach.

As with Ltac, there are no reflected types corresponding to object level names and expressions; the construction of syntactic objects is mediated by Mtac's typed primitives. Conflating the meta-language and object language in this way introduces subtleties in the evaluation strategy [Ziliani et al. 2015, Section 6.4]; using Mtac requires keeping in mind that some values in the context are ordinary values used by the tactic, while others are terms (not necessarily closed) that should only be handled by Mtac's special term-matching primitives.

Although the use of typing mechanisms to describe tactic behavior is attractive and undoubtedly useful for some purposes, it is not clear whether it is well-suited to some of the more elaborate kinds of automation discussed here. For example, a common idiom in automated theorem proving is to begin a procedure by calling the simplifier to normalize or canonize expressions in the goal. Any typing information associated to such a use of the simplifier cannot be very informative, since the effect on the goal cannot be predicted in advance. Finally, as its authors note, "Mtac is interpreted, and it is not clear how it could be compiled, given the interaction between Mtac and Coq unification" [Ziliani et al. 2015, p. 35]. We are hoping to ultimately take advantage of compilation for better performance.

Another approach to developing automation relies on the use of *computational reflection*, that is, showing that a desired theorem is equivalent to an assertion about the result of a computation represented within the language of the theorem prover, and then carrying out the evaluation within the logical foundation. This is the approach taken by Rtac [Malecha and Bengtson 2016; Malecha 2014], which we referred to in Section 8. Reflection seems to be well suited to certain types of

RIGHTSLINK

automation, but it is complementary to our approach. It relies on reflecting the tactic framework within the object language of the theorem prover, and this does not easily scale, whereas a key advantage of our approach is that we can easily expose structures and procedures implemented natively in Lean. Reflection also requires trusting whatever mechanism is used to carry out the computations efficiently, whereas faulty tactics or even a faulty implementation of the virtual machine does not threaten soundness in Lean, since the terms that our tactics construct are always checked by the kernel in the end.

As noted in the introduction, our approach to develop a tactic language is similar to that of Idris [Brady 2013; Christiansen 2014; Christiansen and Brady 2016], but there are differences. Idris' tactic monad is a primitive, whereas ours is defined from `tactic_state`. Also, quotations in Idris are elaborated immediately, whereas ours can be elaborated when the metaprogram is executed. The latter approach allows referencing local assumptions in the current main goal, which is important for writing interactive proofs. Additionally, Idris' elaborator reflection requires the metaprogrammers be aware of the order in which metavariables are solved, while Lean solves metavariables in a dependency-directed manner. As indicated in Section 8, Idris is subject to performance bottlenecks that we manage to avoid.

As far as development tools are concerned, Coq also has both a profiler and debugger for Ltac. In contrast to our approach, these are built in to the theorem prover and can be used only for programs written in the tactic language, not on programs written in the term language. Isabelle provides specialized support for tracing the simplifier [Hupel 2014]. Mtac provides a primitive for tracing in the form of a `print` statement.

10 CONCLUSION

We have described a practical and efficient metaprogramming framework for theorem proving in dependent type theory. Our experiments show that our language and API are performant, and provide flexible means of writing not only small-scale interactive tactics, but also more substantial kinds of automation.

We view this as important progress towards our overarching goal of bridging the gap between interactive and automated reasoning. Users who develop libraries for interactive use can now more easily develop special-purpose automation to go with them, thereby encoding procedural heuristics and expertise alongside factual knowledge. At the same time, users who want to use Lean as a back end to assist in complex verification tasks now have flexible means of adapting Lean's libraries and automation to their specific needs. As a result, our metaprogramming language opens up new opportunities, allowing for more natural and intuitive forms of interactive reasoning, as well as for more flexible and reliable forms of automation.

ACKNOWLEDGMENTS

Avigad's work has been partially supported by Air Force Office of Scientific Research MURI FA9550-15-1-0053 and National Science Foundation DMS-1615444. Ebner's work has been funded by the Vienna Science and Technology Fund (WWTF) project VRG12-004, and co-funded by the Austrian Science Fund (FWF) project W1255-N23.

We are very grateful to David Christiansen, Johannes Hölzl, Greg Malecha, and the anonymous referees for extensive comments, corrections, and advice.

REFERENCES

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2011. The Matita Interactive Theorem Prover. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction. Proceedings (Lecture*

RIGHTSLINK

Notes in Computer Science), Nikolaj Bjørner and Viorica Sofronie-Stokkermans (Eds.), Vol. 6803. Springer, 64–69. DOI: http://dx.doi.org/10.1007/978-3-642-22438-6_7

Jeremy Avigad, Leonardo de Moura, and Soonho Kong. 2017. Theorem Proving in Lean. (2017). https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *J. Formalized Reasoning* 7, 2 (2014), 1–89. DOI: http://dx.doi.org/10.6092/issn.1972-5787/4650

Henry G. Baker. 1991. Shallow Binding Makes Functional Arrays Fast. *SIGPLAN Not.* 26, 8 (Aug. 1991), 145–147. DOI: http://dx.doi.org/10.1145/122598.122614

Eli Barzilay. 2006. *Implementing Direct Reflection in NuPRL*. Ph.D. Dissertation. Cornell University.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, Berlin.

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 73–78. DOI: http://dx.doi.org/10.1007/978-3-642-03359-9_6

Edwin Brady. 2013. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. DOI: http://dx.doi.org/10.1017/S095679681300018X

Adam Chlipala. 2011. *Certified Programming with Dependent Types*. MIT Press. http://adam.chlipala.net/cpdt/.

David Raymond Christiansen. 2014. Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection. In *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014*, Sam Tobin-Hochstadt (Ed.). ACM, 1:1–1:9. DOI: http://dx.doi.org/10.1145/2746325.2746326

David R. Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 284–297. DOI: http://dx.doi.org/10.1145/2951913.2951932

Robert L. Constable. 1998. Types in Logic, Mathematics and Programming. In *Handbook of Proof Theory*. Stud. Logic Found. Math., Vol. 137. North-Holland, Amsterdam, 683–786. DOI: http://dx.doi.org/10.1016/S0049-237X(98)80025-6

Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Inform. and Comput.* 76, 2-3 (1988), 95–120.

Thierry Coquand and Christine Paulin. 1990. Inductively Defined Types. In *COLOG-88 (Tallinn, 1988)*. Lecture Notes in Comput. Sci., Vol. 417. Springer, Berlin, 50–66.

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 378–388. DOI: http://dx.doi.org/10.1007/978-3-319-21401-6_26

David Delahaye. 2000. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Proceedings (Lecture Notes in Computer Science)*, Michel Parigot and Andrei Voronkov (Eds.), Vol. 1955. Springer, 85–95. DOI: http://dx.doi.org/10.1007/3-540-44404-1_7

David Delahaye. 2002. A Proof Dedicated Meta-Language. *Electr. Notes Theor. Comput. Sci.* 70, 2 (2002), 96–109. DOI: http://dx.doi.org/10.1016/S1571-0661(04)80508-5

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 3 (May 2005), 365–473.

Peter Dybjer. 1994. Inductive Families. *Formal Asp. Comput.* 6, 4 (1994), 440–465. DOI: http://dx.doi.org/10.1007/BF01211308

Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. 2016. System Description: GAPT 2.0. In *International Joint Conference on Automated Reasoning, IJCAR (Lecture Notes in Computer Science)*, Nicola Olivetti and Ashish Tiwari (Eds.), Vol. 9706. Springer, 293–301.

Healfdene Goguen, Conor McBride, and James McKinna. 2006. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation*. Springer, 521–540.

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2013. How to Make Ad Hoc Proof Automation Less Ad Hoc. *J. Funct. Program.* 23, 4 (2013), 357–401. DOI: http://dx.doi.org/10.1017/S0956796813000051

Mike Gordon. 2000. From LCF to HOL: a short history. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 169–186.

Lars Hupel. 2014. Interactive Simplifier Tracing and Debugging in Isabelle. In *Intelligent Computer Mathematics - International Conference, CICM 2014. Proceedings (Lecture Notes in Computer Science)*, Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban (Eds.), Vol. 8543. Springer, 328–343. DOI: http://dx.doi.org/10.1007/978-3-319-08434-3_24

RIGHTSLINK

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. DOI: http://dx.doi.org/10.1145/199448.199528

Gregory Malecha and Jesper Bengtson. 2016. Extensible and Efficient Automation Through Reflective Tactics. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Proceedings (Lecture Notes in Computer Science)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 532–559. DOI: http://dx.doi.org/10.1007/978-3-662-49498-1_21

Gregory Michael Malecha. 2014. *Extensible Proof Engineering in Intensional Type Theory*. Ph.D. Dissertation. Harvard University. http://gmalecha.github.io/publication/2015/02/01/extensible-proof-engineering-in-intensional-type-theory.html

Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. 2016. Eisbach: A Proof Method Language for Isabelle. *J. Autom. Reasoning* 56, 3 (2016), 261–282. DOI: http://dx.doi.org/10.1007/s10817-015-9360-2

Conor McBride and James McKinna. 2004. Functional Pearl: I Am Not a Number — I Am a Free Variable. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2004*, Henrik Nilsson (Ed.). ACM, 1–9. DOI: http://dx.doi.org/10.1145/1017472.1017477

Greg Nelson and Derek C Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *Journal of the ACM (JACM)* 27, 2 (1980), 356–364.

Robert Nieuwenhuis and Albert Rubio. 2001. Paramodulation-Based Theorem Proving. In *Handbook of Automated Reasoning*, John Alan Robinson and Andrei Voronkov (Eds.). Vol. 1. Elsevier and MIT Press, 371–443.

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Revised Lectures (Lecture Notes in Computer Science)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.), Vol. 5832. Springer, 230–266. DOI: http://dx.doi.org/10.1007/978-3-642-04652-0_5

Brigitte Pientka. 2010. Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In *Functional and Logic Programming, 10th International Symposium, FLOPS 2010. Proceedings (Lecture Notes in Computer Science)*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.), Vol. 6009. Springer, 1–12. DOI: http://dx.doi.org/10.1007/978-3-642-12251-4_1

Antonis Stampoulis and Zhong Shao. 2010. VeriML: typed computation of logical terms inside a language with effects. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional programming, ICFP 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 333–344. DOI: http://dx.doi.org/10.1145/1863543.1863591

Paul van der Walt and Wouter Swierstra. 2012. Engineering Proof by Reflection in Agda. In *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012 (Lecture Notes in Computer Science)*, Ralf Hinze (Ed.), Vol. 8241. Springer, 157–173. DOI: http://dx.doi.org/10.1007/978-3-642-41582-1_10

Andrei Voronkov. 2014. AVATAR: The Architecture for First-Order Theorem Provers. In *Computer Aided Verification, CAV (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 696–710.

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A Monad for Typed Tactic Programming in Coq. *J. Funct. Program.* 25 (2015). DOI: http://dx.doi.org/10.1017/S0956796815000118

Beta Ziliani, Yann Régis-Gianas, and Jan-Oliver Kaiser. 2017. The Next 700 Safe Tactic Languages. (2017). Preprint.

RIGHTSLINK