The Next 700 Safe Tactic Languages

Beta Ziliani
FAMAF, Universidad Nacional de Córdoba and CONICET
bziliani@famaf.unc.edu.ar

Yann Régis-Gianas
IRIF, PPS, Université Paris Diderot and $\pi.r^2$, Inria
yrg@irif.fr

Jan-Oliver Kaiser
MPI-SWS and Saarland University
janno@mpi-sws.org

This is an old draft. The project has been renamed to Mtac2.

Abstract

The Coq proof assistant ships with Ltac, a powerful language for custom proof automation. Unfortunately, large verification projects developed in Coq frequently have to employ plugins and libraries to work around Ltac's widely-known limitations. Indeed, Ltac is incomplete, has awkward and obscure semantics, and provides almost no typing information to prevent errors before they occur at runtime.

We present MetaCoq, a new tactic language for Coq. At its core lies Mtac2, an extended and significantly improved version of Mtac. Mtac2 provides the primitives required for tactic development. These primitives have formal semantics and are given a *type* in Coq, allowing for the construction of *safe* tactics: A tactic that passes Coq's typechecker either succeeds or raises a meaningful exception.

At its surface, MetaCoq provides a set of tactics to perform basic proof steps, and a collection of tactic *combinators* to effectively write proofs. MetaCoq's tactics are concise and arguably easy to read and write. With examples we show that MetaCoq is an effective tool for proof developers to create tactics and combinators fitting their needs.

Keywords Interactive theorem proving; custom proof automation; Coq; monads; typed meta-programming; tactics.

1. Introduction

Interactive proof assistants like Coq are now common tools employed by researchers worldwide to inspire confidence in their results. Noteworthy examples include major milestones in the verification of large algebraic proofs (Gonthier et al. 2013a; Hales et al. 2015), and in the verification of large software systems (Klein et al. 2010; Leroy 2009).

These assistants owe their success partially to the rich higher-order logics they encode, which allow for the specification and verification of sophisticated theorems such as the ones listed above. However, what is their gain is also their curse: such highly undecidable logics do not allow for the same level of automation SMT solvers provide for fragments of first-order logic. As a consequence, the proof developer must often write several lines of proofs to solve goals, even trivial ones that do not appear in their traditional pen-and-paper counterparts.

To accommodate for this, proof assistants are equipped with languages to build *tactics*: programs that decompose a given goal into smaller *subgoals* until their truth is self-evident. Coq, in particular, includes two languages: OCaml and Ltac. The former is the language used to implement Coq itself. Tactics written in OCaml are compiled and can make use of imperative data structures—making them very efficient. At the same time, developers are exposed the low level details of Coq's internals, including potentially unsafe operations. Moreover, the tactic development process is slow. For one, the proof developer must reason at the level of *de Bruijn* indices, carefully making sure that such indices refers to the right binders. In addition, OCaml tactics must be compiled and linked to Coq every time the tactic is modified.

Ltac, on the other hand, is a dynamic language that allows for a rapid high-level development of tactics, directly within the Coq environment and without requiring compilation and linking. Moreover, it provides a convenient representation of proof terms using the concrete syntax of Coq: it frees the developer to think to the level of de Bruijn indices. However, this language grew "one hack at a time", and despite its many years in the wild, it still lacks many basic language constructs required for proper tactic development—a fact reflected by a growing number of domain-specific tactic languages that are written either as plugins in OCaml (*e.g.*, Gonthier and Mahboubi 2010), or in sophisticated patterns using Coq's overloading mechanisms (*e.g.*, Krebbers et al. 2017).

Another issue with languages like OCaml and Ltac is that they offer almost no static guarantees: a tactic that is successfully defined—that is, accepted by the OCaml compiler or Ltac's interpreter—may construct an ill-typed term that is only rejected when it is too late to understand where the problem originated from.

Recently, Ziliani et al. (2015) devised Mtac, a new language providing the static guarantees tactic languages are currently missing. Mtac is based on the key realization that a tactic language is just a functional language with certain *effects*, like non-termination and syntax manipulation. These effects can be typed in Gallina, the language in which definitions are written in Coq, by means of a monad—not unlike Haskell's `IOMonad`. As a result, a program of type M τ, where M is the monad and τ is any type in Gallina, has the guarantee that, if it terminates, the resulting term will have type τ.

While Mtac was shown to be superior to Ltac in some aspects, it still suffers from a problem shared with Ltac: it does not include enough language constructs to enable the construction of realistic tactics.

Contributions In this work we present MetaCoq, a new framework based on Mtac for writing *typed* tactics and tactic combinators. Tactics in MetaCoq are easy to write, to combine, and to modularize. MetaCoq consists of:

1. Mtac2, a new version of Mtac with a richer set of language constructs and revised semantics.
2. A novel extendable interface for manipulating goals.
3. Several basic tactics (proving MetaCoq's versatility).
4. A new proof mode to write scripts directly in MetaCoq.

MetaCoq is a plugin for Coq downloadable from:

http://github.com/Mtac2/Mtac2

Roadmap In Section 2 we show two examples to quickly highlight some of MetaCoq's key aspects. In Section 3 we provide the basics from the type of tactics to building and executing a few simple tactics. In Section 4 we present several tactics that allows basic *bookkeeping*, *i.e.*, manipulation of hypotheses. In Section 5 we show how to sequence the execution of MetaCoq tactics, and several useful goal combinators. In Section 6 we present some tactics manipulating inductive types. In Section 7 we show new reduction primitives. In Section 8 we discuss some key changes in Mtac2 semantics with respect to Mtac. Finally, in Section 9 we consider the corpus of related work.

2. MetaCoq by example

The* cut *tactic Our first example is one of the simplest tactics one can think of: the cut tactic. It implements plain *modus ponens*: given a proposition U, it solves goal T by creating subgoals $U \to T$ and U. We choose this particular tactic because—despite the triviality of the task at hand—we were not able to write it as an Ltac tactic. The OCaml version of this tactic shipped with Coq is several times longer than ours: of the total 25 *long* lines of its implementation, 4 are dedicated to setting up the tactic, 10 to check that the type of U is a proposition, and 6 to actually create the proof, including adjusting several de Bruijn indices. In MetaCoq, cut is as simple as can be:

```
Definition cut U : tactic := λ g ⇒
  T ← goal_type g;
  ut ← evar (U → T);
  u ← evar U;
  exact (ut u) g;;
  ret [Goal ut; Goal u].
```

This tactic works as follows: it takes a type U and a *goal* g. A goal has a *type* representing what has to be proven. After extracting the type of the goal, T, it creates two meta-variables (**evar**s, short for **e**xistential **var**iables). Meta-variables represent missing parts of a proof. The first one, ut, has type $U \to T$, while the second one, u, has type U. The notation $\cdot \leftarrow \cdot; \cdot$ is customary monadic binding.

The goal g is solved by applying u to ut using another tactic, exact, and the tactic returns the meta-variables as new subgoals. The double semicolon is notation for binding when the variable is not used.

There are several things to note in this short code. Firstly, it is a standard Coq **Definition**. Indeed, the constructors of the Mtac2 abstract syntax trees are part of an inductive type defined within Gallina. Secondly, cut has type tactic—a new type defined entirely in Gallina. Thirdly, since it is typed in Gallina, U *must* be a Type, since it is part of the non-dependent product $U \to T$. In the same vein, we can rest assured that we did not introduce any bug when building the proof $(ut\ u)$: the typechecker of Coq guarantees that this term has *weak* type[1] T.

The* select *tactical The following example shows how MetaCoq's programming model, extended from Gallina, allows to easily compose tactics in interesting ways, leading to short, readable, and maintainable code.

The scenario is as follows: we want to prove a simple tautology manually, *i.e.*, without calling the tactic tauto. The tautology is $(P \to Q) \to P \to Q$, for propositions P and Q. Since we care about maintainability of the proof, the idea is to provide a robust proof script—one that will not break if names or a positions of hypotheses are modified in the future.

We first present the proof in Ltac, and then show how we can improve it using MetaCoq. The Ltac proof, shown in Figure 1, starts by introducing all the variables, purposely without providing names for them. Naming hypotheses leads to fragile proof scripts, since any change in the order of the hypotheses changes the meaning of those names. Next, the tactic selects the hypothesis with product type ($P \to Q$ in this case) by using Ltac's *goal pattern matching* facility. The hypothesis is then provided to the apply tactic. We are left with goal P—and P in our assumptions. The goal is solved by calling the assumption tactic. □

[1] In this context *weak* means that the term is incomplete (has unresolved meta-variables).

```
Goal ∀ P Q, (P → Q) → P → Q.
Proof.
  intros.
  match goal with
  | [ H : _ → _ ⊢ _ ] ⇒ apply H
  end.
  assumption.
Qed.
```

Figure 1. Proving a tautology using vanilla Coq.

```
Goal ∀ P Q, (P → Q) → P → Q.
MProof.
  intros.
  select (_ → _) apply.
  assumption.
Qed.
```

Figure 2. Proving a tautology using MetaCoq.

This proof has two issues: it is rather long for such a trivial task, and it is not modular. Indeed, if we want to generalize the concept of selecting a hypothesis and pass it on to a tactic, as far as we know, it is not doable in Coq today.

Figure 2 shows the proof in *MetaCoq's way*. In essence it is the same as with the previous proof, but *modular*. **MProof** introduces MetaCoq's proof environment. It is followed by a call to the intros tactic. Then, in order to select the hypothesis with product type, it calls the select *tactical*. A tactical is just a tactic that takes another tactic as parameter, and we provide apply as such tactic. As with the previous proof, it also concludes by calling assumption. □

Similarly to the code in Figure 1, the select tactical selects the hypothesis using goal pattern matching:[2]

```
Definition select T (f: T → tactic) : tactic := λ g ⇒
  G ← goal_type g;
  match_goal ([[ (x : T) ⊢ G ]] ⇒ f x) g.
```

In our example, we look for the hypothesis of type $P \to Q$ (without mentioning P or Q directly), and then pass it on to the apply tactic, which in MetaCoq has the following type:

$$\mathsf{apply} : \forall \{T : \mathsf{Type}\}, T \to \mathsf{tactic}.$$

The curly braces around T signify that this argument is *implicit*, *i.e.*, it will be guessed from the following argument. It is interesting to note that this implicit type is equated by the type inference mechanism to the type used by select to perform the search.

[2] For Ltac's *connaisseurs*, MetaCoq's match_goal differs from Ltac's mainly in that it does not perform backtracking, and in the case of several hypotheses matching it takes the first one. But it is encodable in MetaCoq, so if a different behavior is intended, the proof developer can always code a new version!

3. MetaCoq's basics

In this section, we present some of the basic building blocks of MetaCoq. We start with the type for tactics. In essence, we use the conventional definition from LCF-like systems (Harper et al. 1993): a *tactic* is a *program* that takes a *goal* and generates a list of *subgoals*. This definition gives rise to two questions: *(i)* What is a program? *(ii)* What is a goal?

As will become clear below, we need our programs to have access to certain operations that are not encodable in CIC, the pure calculus of Coq. In Krebbers et al. (2017) and Gonthier et al. (2013b), it was shown how to write tactics using Gallina alone by coercing the overloading mechanism of Coq. However, this style of developing tactics is not as direct and simple as one would like. Therefore, in MetaCoq we decided in favor of Mtac (Ziliani et al. 2015). This language provides several operations for meta-programming inside a monad with a type in CIC. For MetaCoq we had to introduce several modifications and new features to it, effectively building a new language: Mtac2. We will inform the reader of the differences between Mtac2 and Mtac when suitable. Appendix A shows the Mtac2 language and its semantics.

Considering the definition of tactics given above, our tactics are Mtac2 programs with the following type:

```
Definition tactic := goal → M (list goal)
```

The type predicate M in the co-domain indicates that a tactic is an Mtac2 program that potentially generates a list of goals. We say "potentially" because it might also fail or loop forever. But if it terminates, then it will return what it promises.

With programs defined, we turn our attention towards goals. A goal consists of a type, the proposition we need to prove, and a *meta-variable* of that type. A meta-variable of type T is a placeholder for a proof term of type T which may or may not be filled depending on the success of the proof search.

On the OCaml side, a specific data constructor classifies terms which are meta-variables. On the MetaCoq side, there is no such data constructor: hence, a meta-variable of type T cannot be distinguished from another term t of the same type by a syntactic check. Actually, such an equality check would *unify* the meta-variable and the term, *i.e.*, it would assign the term t to the placeholder of the meta-variable. To remedy this lack of expressiveness, MetaCoq offers (like Ltac) a special primitive **is_evar** to determine if a given term is a meta-variable or not.

There is another significant difference between the representation of meta-variables in OCaml and their representation in MetaCoq: meta-variables are typed in MetaCoq while they are untyped in OCaml. Therefore, in MetaCoq, it is impossible to ignore that the type of a meta-variable can contain free variables, referring to objects introduced earlier in the proof. To stay meaningful under every typing context, the type of a meta-variable must come with its own typing con-

```
Definition idtac : tactic := λ g ⇒ ret [g].

Definition fail (e : Exception) : tactic := λ g ⇒ raise e.

Definition try (t : tactic) : tactic := λ g ⇒
  mtry t g with _ ⇒ ret [g] end.

Definition or (t u : tactic) : tactic := λ g ⇒
  mtry t g with _ ⇒ u g end.

Definition exact {A} (x : A) : tactic := λ g ⇒
  match g with
  | Goal e ⇒
    b ← munify_cumul x e UniCoq;
    if b then ret nil
    else raise (NotCumul x e)
  | _ ⇒ raise NotAGoal
  end.
```

Figure 3. Basic MetaCoq tactics.

text, represented using a *telescope*. A telescope introduces variables and their types one-by-one in the order of their relative dependencies so that the typing context at the end of the telescope is sufficiently rich to type the goal, i.e.

```
Inductive goal : Type :=
| Goal : ∀ {A: Type}, A → goal
| AHyp : ∀ {A: Type}, option A → (A → goal) → goal.
```

Goal holds the proposition and its proof while AHyp adds a hypothesis to a goal, optionally giving it a definition.

3.1 Basic tactics

We have enough prerequisites now to understand the tactics of Figure 3. The no-op tactic idtac takes a goal g and **ret**urns it in the singleton list $[g]$; fail receives an exception and a goal and **raise**s the exception; try silently ignores any exception from tactic t by catching it with the **mtry-with-end** construct; or first tries t, then u if t fails.

More interestingly, exact solves a goal by instantiating its meta-variable with a given term. First, it takes the meta-variable e from Goal, and then it *unifies* it with the argument x by calling **munify_cumul**. This Mtac2 primitive, discussed in Section 8, unifies two terms using *cumulativity* of universes: an element of certain type is allowed to be coerced into one of a higher type (*e.g.*, Ziliani and Sozeau 2015). If the unification succeeds, the meta-variable is *instantiated* with x and the goal is solved. Therefore, exact returns the empty list. Otherwise, if the goal has introduced hypotheses or unification failed, it raises meaningful exceptions. Here, we assume that e is a meta-variable, although that invariant must be manually maintained. If the invariant does not hold and e is not a meta-variable (*i.e.*, is a proof), then the tactic will only succeed in case e is unifiable with the new proof x.

```
01 Definition apply {T} (c : T) : tactic := λ g ⇒
02   (mfix1 app (d : dyn) : M (list goal) :=
03     mtry exact (elem d) g
04     with _ ⇒
05       mmatch d return M (list goal) with
06       | [? T1 T2 f] @Dyn (∀ x:T1, T2 x) f ⇒
07           e ← evar T1;
08           r ← app (Dyn (f e));
09           ret (Goal e :: r)
10       | _ ⇒ gT ← goal_type g;
11             raise (CantApply c gT)
12       end
13     end) (Dyn c)
```

Figure 4. The apply tactic.

3.2 The apply tactic

A bit more sophisticated than exact is the tactic apply listed in Figure 4. This tactic solves the current goal with the provided proof c (usually a lemma or a hypothesis), as long as the conclusion of c matches the goal. Taking the example from Figure 2, the hypothesis H, with type $P \to Q$, can be applied to the goal Q since the conclusion of H is exactly Q. To do so, apply must provide a proof p for H's antecedent (P), and instantiate the goal with $H\ p$. Instead of doing a proof search for P, apply delays the problem by inserting fresh meta-variables to every antecedent of c. These meta-variables are the subgoals of the tactic.

apply works by iterating over the constructed proof until no more meta-variables can be inserted. At each iteration it tries to solve the goal with the proof constructed so far, and if it fails it tries inserting a new meta-variable. For instance, in the running example it will first consider H as the proof and fail, since the type $P \to Q$ is not unifiable with the goal Q, then insert a new meta-variable $?p$ for P, obtaining proof $H\ ?p$, finally succeeding to solve the goal.

Note that the constructed proof has different types at each cycle (*e.g.*, $P \to Q$ and Q). Therefore, we pack the proof in a *weak* σ-type representing an *element* (the constructed proof) along with its *type*:

```
Record dyn := Dyn { type : Type; elem : type }
```

In Coq's vernacular, Dyn is the constructor of the record dyn, and it takes two arguments, one for each field (type and elem). However, we make the type implicit and simply write Dyn e for packing element e of some type, as we did with Goal. type and elem are functions that, when given a dyn, returns the first and the second value of the record, respectively.

We can now have a detailed look into the code of Figure 4. In line 2, we construct a fixpoint app using Mtac2's fixpoint combinator, **mfix1**, allowing potentially diverging computations. In contrast, Coq's fixpoint combinator only allows terminating recursion on structurally decreasing arguments.

The argument of the fixpoint is the proof d constructed at each iteration, wrapped in a dyn. In line 3, we try to solve goal g with the proof in d using the tactic exact.

If the goal is not solved, an exception is raised and *caught* in line 4, as part of the standard exception handler "**mtry** ... **with**". In lines 5 and 6, we inspect d to find out if it has product type. The keyword **mmatch** introduces a higher-order pattern matching to introspect Coq terms using higher-order unification. As with Coq dependent pattern matching, a **return** clause helps the typechecker to refine the type of the scrutinee by introducing type equalities in the context. The patterns used in the branch must introduce unification variables explicitly using the syntax $[?\ X\ Y\ \ldots]$.

As customary in Coq, we use the @ symbol to provide all the arguments of a constructor, even the implicit ones like the type argument of Dyn. If d indeed has function type $\forall x : T_1, T_2$, we create a meta-variable e with type T_1 (line 7), and we recursively call app with f, the proof, applied to e (line 8). At line 9, we return the list of goals r with the new goal e *pushed* in front of it. If d does not contain an element of product type, it must be because it is fully applied, but it does not have the right type to solve the goal, so we raise a meaningful exception (line 11). The tactic packs the term c in a dyn and feeds it to the fixpoint (line 13).

3.3 Executing tactics

MetaCoq provides several ways for executing tactics, according to the user's need.

The* MProof *environment. The new **MProof** proof environment, already mentioned in the introduction, is the main option to execute tactics. It also supports the execution of any Mtac2 term producing an element of the expected type. For instance, the following are two (rather trivial) proofs of $n + 0 = n$ for some n in the context. The first one uses the exact tactic with the existing lemma add_0_r, and the second one directly returns the lemma.

```
Goal n + 0 = n.
MProof. exact (add_0_r _). Qed.

Goal n + 0 = n.
MProof. ret (add_0_r _). Qed.
```

Note that in both cases the argument of the lemma is obtained from the type of the goal.

The* mrun *tactic. Alternatively, we provide a new OCaml tactic called mrun. This tactic enables the execution of MetaCoq code (a tactic or an Mtac2 program) in existing proofs.

```
Goal n + 0 = n.
Proof. mrun (exact (add_0_r _)). Qed.
```

4. Tactics manipulating variables and binders

In this section, we target one of the most important processes in proof writing: *bookkeeping*. Indeed, according to Ssreflect's creators (Gonthier and Mahboubi 2010), a substantial

```
Definition intro0 (var : string) : tactic := λ g ⇒
  mmatch g return M list goal with
  | [? A P e] @Goal (∀ x : A, P x) e ⇒
    nu var None (λ x : A ⇒
      e' ← evar (P x);
      fe' ← abs_fun (P := P) x e';
      exact fe' g)
  | _ ⇒ raise NotAProduct
  end.
```

Figure 5. The intro0 tactic.

amount of lines of code involve moving hypotheses from the goal to the context and back. Bookkeeping requires careful manipulation of variables and binders, and we show with examples an original and unified set of operators devised to that end.

More concretely, this section presents: a simple tactic to introduce a variable in the context (§4.1); a tactic to introduce a definition in the context (§4.2); an improvement to these tactics so they can be *composed* with other tactics (§4.3) (the composition operator will be presented in Section 5); a tactic to generalize a variable from the goal (§4.4); a tactic to create a fixpoint (§4.5); and a tactic to clear a variable from the context (§4.6).

4.1 Introducing a variable in the context

Let us assume we want to solve a very simple goal:

$$\forall P : \mathsf{Prop}, P \to P$$

The first tactic we can employ to make progress in a proof is the intros tactic.

```
intros P p
```

This tactic introduces a variable P of type Prop and a variable p of type P in our context. Mathematically, this step of the proofs corresponds to a "let P be a proposition and let p be a proof of that proposition. . . ". Operationally, this tactic must create a function abstraction for each product, with the new goal having a context extended with the freshly introduced variable. Following the running example, the original goal is partially solved by the term

$$\lambda P \Rightarrow \lambda p \Rightarrow ?e$$

where $?e$ is the new goal; a fresh meta-variable having access to P and p. The interested reader is invited to read (Ziliani et al. 2015, Section 4.2) for details on how meta-variables track their context.

The intros tactic in its full form allows for the introduction of several variables and definitions (let-bindings). In this section, we consider only one variable, and in the coming section we consider only one let-binding. For the actual tactic shipped with MetaCoq the reader is invited to read the source files.

The code listed in Figure 5 shows tactic intro0. This tactic *mostly* works, but has an issue that we will solve in Section 4.3. intro0 works as follows: It pattern matches the goal to see if it has a product type ($\forall x : A, P\ x$), for some A and P. As we did with the apply tactic before, we use the @ symbol to provide all the arguments of Goal. If the goal is not a product, or it is not a goal, it raises an exception (line 8).

If it is a product, it introduces a variable having the *name* provided (*var*, a string), and executes the code inside the closure. This is performed *via* the **nu** operator. This operator has the following type:

$$\textbf{nu} : \forall \{A\ B\}, \mathsf{string} \to \mathsf{option}\ A \to (A \to \mathsf{M}\ B) \to \mathsf{M}\ B$$

It takes the name n of the variable, a definition d (optionally), and a closure f, and it executes the code in the closure under the local context extended with a variable named after n. It has a necessary restriction: the variable n must not appear free in the return value of f, otherwise the result would become ill-typed outside of the scope of n.

The closure creates a meta-variable (line 5) having type $P\ x$, where x will be replaced by the variable introduced, and P represents the co-domain of the product. Then, in line 6, x is abstracted from the just created meta-variable. The operator **abs_fun**, created to that effect, has the following type:

$$\textbf{abs\_fun} : \forall \{A\} \{P{:}A \to \mathsf{Type}\} (x : A), P\, x \to \mathsf{M}\ (\forall x, P\, x)$$

It takes a variable x and an expression e of type $P\ x$, and creates a λ-abstraction $\lambda x \Rightarrow e$ "closing" e w.r.t. x. In this case, the expression e is the meta-variable.

Finally, in line 7, the goal g is solved using the result fe' of the previous line.

For readers already familiar with Mtac, we have implemented two changes in these operators:

1. The **nu** operator takes a name, which must be unique, and an optional definition. Previously, the name was given by the name of the variable in the closure, and it was not restricted to be unique. But this created several naming issues. It was also not possible to introduce a definition, a crucial requirement for the tactic in the coming section.
2. The **abs_fun** operator is less restrictive than the former **abs** operator. In order to maintain soundness, this operator must check that the free variables of the term e does not themselves depend on the variable x being abstracted. In the former version, this check required that all variables in the context, and not only those occurring in e, do not depend on x.

4.2 Introducing a let-binding

Consider the following goal stating that computing twice the Fibonacci number of x is equal to multiplying it by 2:

$$\forall x, \textbf{let}\, fx := \mathsf{fib}\ x\ \textbf{in}\, fx + fx = 2 * fx$$

```
01 Definition intro_let0 (var : string) : tactic := λ g ⇒
02   mmatch g return M (list goal) with
03   | [? A (d:A) P e] @Goal (let x := d in P x) e ⇒
04     nu var (Some d) (λ x : A ⇒
05       e' ← evar (P x);
06       fe' ← abs_let (P := P) x d e';
07       exact fe' g)
08   | _ ⇒ raise NotAProduct
09   end.
```

Figure 6. The intro_let0 tactic

If we have in our library a theorem stating $\forall n, n + n = 2 * n$, it is not optimal to expand the definition of the let-binding: we should instead push the definition in our list of hypotheses to apply the general theorem to the goal. We can do this again using the **nu** operator, and a new operator: **abs_let**.

Figure 6 shows an analogous tactic to intro0, but for introducing let-bindings. It differs in three aspects: it pattern matches a let (line 3), it introduces the definition with the **nu** operator (with the argument (Some d) in line 4), and it abstracts the variable using **abs_let** (line 6). This new operator has the following type:

$$\textbf{abs\_let} : \forall \{A\ P\} (x{:}A) (t{:}A), P\ x \to \mathsf{M}\ (\textbf{let}\ x := t\ \textbf{in}\ P\ x)$$

In the running example, assuming variable x was introduced previously, the partial term fe' used by intro_let0 to solve the goal is

$$\textbf{let}\, fx := \mathsf{fib}\ x\ \textbf{in}\ ?e'$$

with $?e'$ having type $fx + fx = 2 * fx$.

4.3 Returning a new goal

The tactics from sections 4.1 and 4.2 have a problem: they return an empty list of goals. (Remember from Figure 3 that exact, the tactic called at the end of intro0 and intro_let0, returns no subgoal.) Hence, the freshly created metavariable $?e'$ has not been registered as a goal by the proof engine. If we want to compose these tactics with any other tactic, as custom in Coq scripts, and as we will see in Section 5, it just would not work.

This is the motivation behind the AHyp constructor in the goal type. The general idea will be to create a Goal containing the freshly generated meta-variable, and *seal* it in a similar fashion as we did before using the **abs_fun** operator.

We explain the additional lines required in intro0, and let the reader figure out the necessary changes to intro_let0. The call to the exact tactic in line 7 of Figure 5 must be replaced with:

```
exact fe' g;;
nG ← abs_fun x (Goal e');
ret [AHyp None nG]
```

```
01 Definition generalize {A} (x:A) : tactic := λ g ⇒
02   gT ← goal_type g;
03   aT ← abs_prod x gT;
04   e ← evar aT;
05   mmatch aT with
06   | [? Q : A → Type] (∀ z:A, Q z) ⇒ [H]
07     let e' := match H in _ = T return T with
08       | eq_refl _ ⇒ e
09       end in
10     exact (e' x) g;;
11     ret [Goal e]
12   | _ ⇒ failwith "generalize: should never happen"
13   end.
```

Figure 7. The generalize tactic

First, we have the same call to the exact tactic. But now it is composed with the double-semicolon. The code continues by packing e', the new meta-variable, with Goal, and sealed it inside a λ-abstraction using **abs_fun**. Finally, we return the singleton list containing a goal created with AHyp and the new goal nG.

4.4 Generalizing a variable

The intros tactic *moves* a variable from the goal to the list of hypothesis. When we want to do the opposite, we are *generalizing* a variable from the goal, a situation that occurs, for instance, when we need to weaken the induction predicate.

Figure 7 shows the tactic generalize created to that end. It receives a variable x and a goal g, and abstracts x from the goal's type gT. **abs_prod** is used to that effect. This new operator has the following type:

$$\textbf{abs\_prod} : \forall\{A{:}\mathsf{Type}\}\ (x{:}A), \mathsf{Type} \rightarrow \mathsf{M\ Type}$$

Note a difference w.r.t. **abs_fun** and **abs_let**: instead of taking a type predicate P and an element of $P\ x$, it only takes a type. And the returning type is Type. This is because the returning type cannot say anything interesting about the returned value. If, for instance, the goal is $x > 0$ and we abstract x from it, we obtain the type $\forall x, x > 0$ which has type Type.

We use the resulting type aT to create a new meta-variable e (line 4), which will become the new goal. Ideally, we would like to solve the current goal g with e applied to x. However, e's type is aT: there is no information that it is actually a product taking an element of the type of x.

Usually types help us construct concise and sound tactics, like in the cut tactic from the introduction.There are cases, luckily not too often, where they actually get on our way. We hit one such unfortunate situation in our generalize tactic. We have to cast e to have product type.

In lines 5 to 9 we pattern match aT to ensure it has type $\forall z{:}A, Q\ z$, for some Q. The details are unimportant, but basically in lines 7 to 9 we are casting e, of type aT, to

```
01 Theorem plus_n_O : ∀ n:nat, n = n + 0.
02 MProof.
03   fix_tac "IH" 1.
04   destructn 0.
05   - reflexivity.
06   - intro n'. simpl. rewrite ← IH. reflexivity.
07 Qed.
```

Figure 8. Using fix_tac.

```
01 Definition fix_tac f i : tactic := λ g ⇒
02   gT ← goal_type g;
03   r ← nu f None (λ f:gT⇒
04     new_goal ← evar gT;
05     fixp ← n_etas i new_goal;
06     fixp ← abs_fix f fixp i;
07     new_goal ← abs_fun f (Goal new_goal);
08     ret (fixp, AHyp None new_goal) );
09   let (f, new_goal) := r in
10   exact f g;;
11   ret [new_goal].
```

Figure 9. The fix_tac tactic to create a fixpoint.

have type $\forall z{:}A, Q\ z$. Then we can successfully apply x to the resulting term e' and solve the goal in line 10.

4.5 Creating a fixpoint

Figure 9 shows the fix_tac tactic. It takes a name f and a number i and, similar to Coq's fix, it allows for the construction of proofs involving recursion on the i-th argument of the goal.

As an example, Figure 8 uses fix_tac to prove by induction that 0 is an identity on the right. The proof starts by calling fix_tac with arguments IH and 1. IH will be the inductive hypothesis, having exactly the same type as the goal. The number 1 indicates that we are doing recursion on the first product, that is, n. The proof is mostly uninteresting, except for line 6 where the inductive hypothesis is used.

Coming back to Figure 9, the tactic works as follow, line by line. On line 3, using **nu** we introduce a variable with name f having the same type as the goal, gT. On line 4, a new goal (meta-variable) is created, also having type gT. It will have f in its context. On line 5, the goal is η-expanded i-th times. For instance, for the running example *fixp* will be $(\lambda n : \mathsf{nat} \Rightarrow ?new\_goal\ n)$ (the function n_etas can be found in the accompanying code). Note that it will have the same type as the goal, gT. On line 6, f is abstracted from *fixp* using another new operator:

$$\textbf{abs\_fix} : \forall\{A{:}\mathsf{Type}\}, A \rightarrow A \rightarrow \mathsf{nat} \rightarrow \mathsf{M}\ A$$

When calling **abs_fix** with arguments f, *fixp* and i it creates a fixpoint f decreasing on the i-th argument of *fixp* (which has to be a function with at least i binders). For the details on

```
01 Definition clear {A} (x:A) : tactic := λ g ⇒
02   gT ← goal_type g;
03   e ← remove x (evar gT);
04   exact e g;; ret [Goal e].
```

Figure 10. The clear tactic.

Coq's fixpoints we refer the reader to the manual (The Coq Development Team 2016).

The following lines follow a similar pattern to previous tactics: the new goal is sealed with the hypothesis, and the current goal is solved using the new goal.

4.6 Clearing a variable from the context

In order to remove clutter when working with several hypotheses, proof developers may "throw away" unnecessary hypotheses by calling the clear tactic.

Figure 10 presents clear. In essence, clearing a variable x from a goal essentially encompasses creating a new goal, having access to the same list of hypotheses that the current goal except x, and solving the current goal with it. Line 3 creates the new goal using **evar**, but inside another new operator:

$$\textbf{remove} : \forall\{A{:}\mathsf{Type}\}\{B{:}\mathsf{Type}\}, A \to \mathsf{M}\ B \to \mathsf{M}\ B$$

This operator takes a variable z and a piece of code t, and it executes t in the same context as the code executing **remove**, but without x. To avoid unsound results, **remove** ensures that t does not have z—or any other variable depending on z—free, raising an exception if this condition is not satisfied.

In clear, calling **evar** inside **remove** will create a meta-variable without x in its context, as required. The clear tactic then instantiates the current goal with the created meta-variable and returns the new goal.

5. Composing tactics in MetaCoq

In Ltac it is custom to compose tactics with the semicolon operator. This operator acts on different structures, having different semantics depending on what is being composed. In MetaCoq we emulate the same behavior, and we extended it to handle also *goal selectors*.

MetaCoq's composition operator takes the notation from the pipe redirection operator in terminals: &>. On the left-hand side of a &> there is always a tactic. On the right it accepts the following types of operands (and it can be extended to handle others, as will be noted soon):

A single tactic. When composing two tactics, the second tactic is applied to all the subgoals generated by the first tactic. For instance, we can "redirect" all the goals generated by the destruct tactic to the assumption tactic by writing:

```
destruct n &> assumption
```

destruct performs a case analysis on the argument, in this case n. If, for instance, n is a natural number, it generates two subgoals, one for zero and one for successor. In the script above all of them are solved using the tactic assumption.

A list of tactics. When composing a tactic with a list of tactics, the first tactic in the list is applied to the first subgoal generated by the tactic on the left, the second tactic in the list to the second subgoal, and so on. For instance, consider the following example where n is again a natural number:

```
destruct n &> [assumption; exact X]
```

(Note the standard notation for Coq lists $[t_1; \ldots; t_n]$ to represent a list with elements t_1 to t_n.) In this case, the first subgoal is solved by the assumption tactic, and the second one by some hypothesis named X.

A goal selector. Goal selectors are just functions from a list of goals to a list of goals, and they enable the filtering or reordering of goals.

```
Definition selector := list goal → M (list goal).
```

For instance, the following goal selector takes the n-th goal of the list (starting from 0) using the standard list operation nth_error:

```
Definition snth n : selector := λ l ⇒
  match nth_error l n with
  | None ⇒ raise NoGoalsLeft
  | Some g ⇒ ret [g]
  end.
```

We can then solve the first subgoal composing snth:

```
destruct n &> snth 0 &> assumption
```

In fact, since it is so common to use such selector, we provide a suitable notation for it:

```
destruct n |1> assumption
```

In order to make &> work with these variations we take advantage of the overloading mechanisms readily available in Coq. In fact, a MetaCoq user can overload the operator with another type of operand if suitable. Since it is not the purpose of this work to explain overloading, and there is nothing special about it in our use in MetaCoq, we defer the reader to the source code for details.

This is yet another advantage of MetaCoq over traditional approaches: we have at our disposal the full Gallina language and its tools to build our tactics. We already mentioned Coq's typechecker and overloading mechanisms, but we can also take advantage of the **Program** command to build tactics interactively, the module system (not only for *namespacing*!), etc.

A little detour. As a matter of fact, Coq is equipped with two different mechanisms: Type Classes (Sozeau and Oury 2008) and Canonical Structures (Mahboubi and Tassi 2013). In MetaCoq we use the latter for two practical reasons: 1) The resolution mechanism of Type Classes is less predictable, and sometimes reduces terms unnecessarily, and 2) the lead author has more experience with Canonical Structures.

In the following section we will see how the composition operator deals with variables introduced by previous tactics.

5.1 Composing intros

Someone used to Ltac will naturally write the following code to introduce a variable in the hypotheses and use it afterwards in a tactic, here destruct:

$$\mathsf{intros}\ n\ \&\!>\ \mathsf{destruct}\ n$$

However, this does not work in MetaCoq. The reason is simple: we decided to make the &> operator agnostic about the names introduced by intros. That is, intros does not have a special status like in Ltac. Indeed, in Ltac other tactics introducing names, like rename or Ssreflect's move, do not enjoy such special treatment and cannot be used effectively in Ltac procedures. In contrast, for us intros is just a tactic like any other.

Our alternative to the broken script above, which we strongly argue is better than Ltac's semicolon, is to use *scoped intros*, as we will see in Section 5.2. This said, we do not prevent the use of intros followed by a &>, as long as the name is not used. For instance, the following is a perfectly valid MetaCoq code:

$$\mathsf{intros}\ a\ b\ c\ \&\!>\ \mathsf{assumption}$$

The tactic assumption will see in the list of hypotheses that there are three, named a, b, and c.

Is for this kind of legal uses of intros that we created the goal type as a telescope, extending the telescope in Section 4.3 with the introduced variable.

If we recall the tactics we have seen so far, every time a tactic in MetaCoq receives a goal g, it is always assumed the telescope is *open*: g is just Goal and not AHyp. In order to keep this invariant, the composition operator must open goals prior to pass them on. It does so by calling the tactical open_and_apply from Figure 11.

open_and_apply takes a tactic t and a goal g, and for each AHyp in g it introduces the variable (or definition) in the context, and recurses. When it reaches the Goal, it executes t with the opened goal. After recursing, it seals back the list of goals generated by t by calling auxiliary functions (let_)close_goals.

In order to ensure the new variables introduced respect the names introduced before, *e.g.*, by the intros tactic, we obtain the names from the binders of the telescope calling the following operator:

$$\textbf{get\_binder\_name} : \forall\{A : \mathsf{Type}\}, A \to \mathsf{M\ string}$$

This operator returns the name of the binder in a function, a let-binding, or a product. Otherwise, it raises an exception.

```
Definition open_and_apply (t : tactic) : tactic :=
  fix open g :=
    match g return M _ with
    | Goal _ ⇒ t g
    | @AHyp C None f ⇒
      x ← get_binder_name f;
      nu x None (λ x : C⇒
        open (f x) ≫ close_goals x)
    | @AHyp C (Some t) f ⇒
      x ← get_binder_name f;
      nu x (Some t) (λ x : C⇒
        open (f x) ≫ let_close_goals x)
    end.
```

Figure 11. The tactical open_and_apply.

5.2 Scoped introduction of variables

Let us ponder on the information provided in sections 4.1, 4.3, and 5.1, and consider the following short code:

$$\mathsf{intros}\ H\ \&\!>\ \mathsf{assumption}$$

The following actions regarding variable H are taking place:

1. H is added to the context (intros).
2. A new goal is created (intros).
3. H is abstracted from the goal created in (2) (intros).
4. H is re-introduced (&>).
5. H is abstracted from the result of assumption (&>).

It seems a bit too bureaucratic for such a simple task. Luckily, there is a simple solution: instead of using &> to compose intros, use *scoped intros*:

$$\mathsf{cintros}\ H\ \{\text{- assumption -}\}$$

This tactic only performs steps 1, 2, and 5. The only interesting bit in the implementation of cintros is that it is a Coq *recursive* notation, allowing for an arbitrary number of variables without involving explicit recursion:

```
Notation "'cintros' x .. y '{-' t '-}'" :=
  (intro_cont (λ x ⇒ .. (intro_cont (λ y ⇒ t)) ..))
  (at level 0, x binder, y binder, right associativity).
```

6. Working with inductive types

Inductive types are an integral part of Coq, and in Mtac2 we provided several primitives to deal with them: **constrs**, **destcase**, and **makecase**.

```
Inductive Vector (T : Type) : (∀ n : nat), Type :=
| vnil : Vector T 0
| vcons (n : nat) (t : T) : Vector T n → Vector T (S n).
```

Figure 12. An inductive type of length-indexed lists.

```
Definition constructor (n : nat) : tactic := λ g ⇒
  A ← goal_type g;
  match n with
  | 0 ⇒ raise ConstructorsStartsFrom1
  | S n ⇒
      l ← constrs A;
      match nth_error (snd l) n with
        | Some x ⇒ apply (elem x) g
        | None ⇒ fail CantFindConstructor g
      end
  end.
```

Figure 13. The constructor tactic.

In order to understand these primitives we need to understand some of inductive types's main components: *parameters* and *indices*: the former are arguments to both the inductive type itself and all constructors. They are implicitly quantified over in every constructor and must not be changed in any occurrence of the inductive type in the constructor's signature. Parameters must be the first arguments of an inductive type, a restriction which is enforced syntactically.

Indices, on the contrary, are arguments only to the inductive type, and they are allowed to change in every constructor's signature.

Figure 12 contains an example of an inductive type that describes length-indexed lists (often called vectors). The argument T is a parameter, n is an index, and the constructors are vnil and vcons. Note, particularly, that the constructors's signatures mention parameter T without quantifying over it explicitly, and that they instantiate the occurrences of Vector T with different indices.

The Mtac2 primitive **constrs** takes an inductive type and returns a pair containing a dyn and a list of dyns. The first element contains the same inductive type, applied to all of its parameters but without the indices. For instance, if we call **constrs** on Vector nat 1, it will return Vector nat. The second element contains the constructors, also applied to the parameters (*e.g.*, vnil nat and vcons nat). Note that we need to wrap these elements in dyns since their types are not easily expressed in terms of the inductive type provided.

Using **constrs**, we can easily implement a constructor tactic (Figure 13), which solves a goal of type T by applying its n-th constructor.

With **constrs** taking care of the inspection of inductive definitions, we are still missing primitives to handle their *destruction*: dependent pattern matching.

In Coq, pattern matching is done with syntactic match constructs that are generated for every inductive definition. For instance, the following is a template example of a dependent match on the Vector type:

```
match v in Vector _ k return P k with
| vnil ⇒ (* goal: P 0 *)
| vcons n' a v' ⇒ (* goal: P (S n') *)
end
```

Note that the return type may depend on the specific indices of the given inductive value.

As pattern matching terms only exist for specific inductive types, it is impossible to construct a pattern matching term on a statically unknown inductive type. This represents an obstacle when considering **mmatch** to identify pattern matching terms unless the inductive type is known when the tactic is typechecked.

MetaCoq solves this problem by providing the **destcase** and **makecase** primitives—responsible for inspecting and, respectively, creating pattern matching terms.

When **destcase** is given a pattern matching term, it will return a record specifying the inductive type, the inductive value, the return type, and the branch associated with every single constructor. The branches are functions of the respective constructor's arguments except for parameters of the inductive type. Both the return type and the branches are wrapped in dyn since their type cannot easily be expressed in terms of the input type or the inductive type.

destcase's counterpart **makecase** takes such a record and constructs the corresponding pattern matching term. It assumes that the list of branch functions is in the order in which the constructors were defined and that parameters are not abstracted over—an invariant maintained by both **constrs** and **destcase**.

With **constrs** and **makecase** we were able to build a simple destruct tactic. Unfortunately, space forbids us to show it here, and we are currently working on a significant improvement of this tactic.

7. Reducing terms

Tactics often need to reduce terms in order to build more compact proofs, or simply to avoid goals to be cluttered with unnecessary information. In Mtac2 we drastically improved Mtac's reduction primitives in two aspects:

1. Reduced terms remain equal to the original term for the typechecker. Indeed, in the former version the reduction primitives were tied to the monadic unit (**ret**), making it impossible to inform the typechecker, without a coercion, that the returned term was convertible to the original one.
2. Several reduction functions were added.

In order to achieve (1), we instruct the interpreter of Mtac2 to not reduce let-bindings. Then, we inspect the head of the term in the definition of the let-binding: if it is a special constant, then we call the reduction engine. If it is not, then the let-binding is reduced as custom. For instance, the following code performs β-weak-head reduction on a term t before returning it:

let x := reduce (RedWhd [RedBeta]) t **in ret** x

We define the constant reduce to be just the identity function. Therefore, for Coq's typechecker x is just a definition for t: whenever t is expected, x is also accepted.

As for (2), the following is the list of reduction functions included in Mtac2: RedWhd *flags* for (weak) head reduction, RedStrong *flags* for strong reduction, RedSimpl for the heuristic used by Coq's simpl tactic, and RedOneStep for one step head reduction. As for the flags, currently it accepts a list with the following: RedBeta, RedDelta RedIota, RedZeta, for $\beta, \delta, \iota, \zeta$ reduction, respectively (see The Coq Development Team (2016, ch. 4.3) for details). Additionally, it includes flags RedDeltaX and RedDeltaC for unfolding of just variables or constants, respectively (RedDelta performs both).

8. Unification primitives + backtracking semantics: enabling pattern matching

In Mtac the evaluation of **mmatch** was mainly performed on the OCaml interpreter side. Thanks to new unification primitives and backtracking semantics we managed to write **mmatch** directly in Mtac2. Similarly, we developed a tactic to perform *goal pattern matching* directly in MetaCoq.

8.1 Unification primitives

In Mtac2 we have two new primitives for unification:

munify : $\forall\{A\}(x\ y : A)$, Unif $\rightarrow$ M (option $(x = y)$)
munify_cumul : $\forall\{A\ B\}(x : A)(y : B)$, Unif $\rightarrow$ M bool

In the first one, x and y must have the same type. As a result, it produces a proof of $x = y$ (or returns None if they are not unifiable). In the second one, x and y may have different types A and B, respectively, and it will succeed if x and y are unifiable and A is a Type *lower or equal* in the universe hierarchy. For instance, if x is True, having Prop type, and y is a meta-variable with type Type, then it will succeed. Note that it cannot return a proof of x being equal to y because their types are not strictly equal.

The argument with Unif type switches among different algorithms: UniCoq uses the algorithm described in Ziliani and Sozeau (2015), inspired in MetaCoq's needs. UniMatch uses the same algorithm, but it prevents the instantiation of meta-variables in the term on the right (y above) and reduction of the term on the left (x above). UniMatchNoRed is like UniMatch but it also prevents reduction on y. Finally, UniEvarconv uses Coq's own unification algorithm.

8.2 Backtracking semantics

In Mtac, meta-variables were conceived as one-time-write pointers. Therefore, the creation and instantiation of meta-variables was not rolled-back when an exception was raised. In the new semantics of Mtac2, meta-variables created—or instantiated—inside a **mtry** block are discarded—or *un-instantiated*—before executing the **with** block.

To give an example, in the following code two meta-variables are created, e_1 outside the **mtry** block, and and e_2 inside. e_1 is instantiated with number 1 inside the **mtry** block. After the exception is raised, e_2 is erased and e_1 remains un-instantiated.

```
e1 ← evar nat;
mtry
  e2 ← evar nat;
  munify e1 1 UniCoq;;
  raise exception
with _ ⇒ ret 0 end
```

For space reasons we are not able to show how we implement pattern matching, and goal pattern matching, using these two concepts, but the interested reader is invited to find the implementation in the accompanying code.

9. Related work

Type for tactics LCF (Gordon et al. 1979) introduced *backward reasoning* based on tactics. A tactic in LCF enjoys the type (Gordon 2015) goal $\rightarrow$ list goal $\times$ procedure where *procedure* validates the process of reducing the input goals to a list of subgoals. The resemblance with our type for tactics is striking. (Remember, it is goal $\rightarrow$ M (list goal).) The differences between the traditional LCF type for tactics and ours are nonetheless essential.

First, we do not need a validation procedure to be safe. Indeed, many tactics are already sufficiently typed to be automatically justified by Coq's typechecker. Other tactics may be weakly typed: their types do not say what they prove. By contrast, in a strongly typed metalanguage like VeriML (Stampoulis 2012), every tactic is specified and justified by a very precise type. As this very strong typing can make some tactics hard to write, we decide to allow some form of dynamic typing. Note that the overall process will never be unsafe since Coq follows the *de Bruijn principle*: its kernel will eventually recheck the final proof entirely.

Second, as noted by Matita's designers (Sacerdoti and Tassi 2009), the LCF type for tactics has many deficiencies: it ignores meta-variables; it forces a form of locality during goal solving; it forbids the rendering of partial proofs; it complicates the implementation of declarative languages and mechanisms to structure scripts. While Matita's type for tactics exposes a global environment and the entire context in a low-level OCaml type, we encapsulate the available effectful operations inside a monad, which simplifies tactic composition. In the same vein, Spiwack (2010) proposes a backtracking monad for tactics, which has been integrated in the proof engine of Coq. Such a backtracking monad can be implemented in MetaCoq but this is not the default computational model, which is deterministic, hence more predictable.

Types of tactics SSReflect (Gonthier and Mahboubi 2010) provides an alternative to Ltac's tactics. It is designed to incorporate small-scale reflection steps in proofs, and more generally to improve the robustness of proof script with respect to changes. Unfortunately, due to its poor interaction with Ltac, SSReflect's scripts are hard—if not impossible—to modularize. We believe that MetaCoq can be used to implement tactic languages similar to SSReflect directly in Coq.

In contrast to SSReflect, Rtac (Malecha and Bengtson 2016) provides reflection-based *tactics* in pure Gallina. As in MetaCoq, an Rtac tactic is simply a Coq term. Yet, contrary to MetaCoq, the semantics of Rtac stays inside Coq. (There is no need for an external interpreter.) Therefore, while MetaCoq's interpreter must produce a concrete effect-free proof-term to serve as a trusted proof, a tactic application in Rtac is directly a (more compact) proof. As the Cybele experiment shows (Claret et al. 2013), certain effects—like non-termination—can be executed outside Coq, returning a *witness*, and replayed inside Coq.

In declarative proof languages like Isar (Wenzel 1999) or Corbineau's declarative mode (Corbineau 2007), a proof is written following forward reasoning instead of backward reasoning. Such declarative proofs mimick pencil and paper proofs and they are usually easier to read than traditional LCF proofs. We believe that MetaCoq is expressive enough to allow such declarative languages to be simply implemented as *embedded domain specific languages*, i.e. as libraries of combinators. Similarly, we expect that the implementation of the IPM (Interactive Proof Mode for embedded logics) (Krebbers et al. 2017) can be significantly simplified and improved by using MetaCoq instead of plain Ltac and overloading.

Acknowledgments We are very thankful to: Béatrice Carré, for contributing to the creation of the **MProof** environment; Jacques-Pascal Deplaix, for constructing the former primitive for executing Ltac tactics within Mtac2 and for writing several MetaCoq's tactics; Thomas Refis for his contribution in the discussion about `match_goal`; and Matthieu Sozeau, for his invaluable help with the Coq API.

References

G. Claret, L. del Carmen González Huesca, Y. Régis-Gianas, and B. Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *ITP*. Springer, 2013.

P. Corbineau. A declarative language for the Coq proof assistant. In *TYPES*, pages 69–84. Springer, 2007.

G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *JFR*, 3(2):95–152, 2010.

G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP*. Springer, 2013a.

G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *JFP*, 23(4):357–401, July 2013b.

M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF: a mechanized logic of computation. *LNCS*, 78:161, 1979.

M. J. C. Gordon. Tactics for mechanized reasoning: a commentary on milner (1984) 'the use of machines to assist in rigorous proof'. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 373 (2039), 2015. ISSN 1364-503X.

T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu, and R. Zumkeller. A formal proof of the kepler conjecture. *arXiv*, 1501.02155, 2015.

R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, Jan. 1993. ISSN 0004-5411.

G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *CACM*, 53(6):107–115, 2010.

R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, 2017.

X. Leroy. Formal verification of a realistic compiler. *CACM*, 52 (7):107–115, 2009.

A. Mahboubi and E. Tassi. Canonical structures for the working Coq user. In *ITP*. Springer, 2013.

G. Malecha and J. Bengtson. Extensible and efficient automation through reflective tactics. In *ESOP*. Springer, 2016.

A. A. W. R. C. Sacerdoti and C. E. Tassi. A new type for tactics. *PLMMS09*, page 22, 2009.

M. Sozeau and N. Oury. First-class type classes. In *TPHOLs*, Berlin, 2008. Springer.

A. Spiwack. An abstract type for constructing tactics in Coq. In *Proof Search in Type Theory*, 2010.

A. Stampoulis. *VeriML: A dependently-typed, user-extensible and language-centric approach to proof assistants*. PhD thesis, Yale University, 2012.

The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.5*, 2016. URL `http://coq.inria.fr`.

M. Wenzel. Isar—a generic interpretative approach to readable formal proof documents. In *TPHOLs*. Springer, 1999.

B. Ziliani and M. Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *ICFP*, New York, 2015. ACM.

B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *JFP*, 25, 2015.

A. Mtac2 Language and Its Semantics

In this appendix we show the Mtac2 language to its full extent, together with its big-step operational semantics.

The type of the operations are given in the following inductive type:

$\textsf{\textbf{Inductive}}\ \mathsf{M} : \mathsf{Type} \to \mathsf{Prop} :=$
$|\ \textsf{\textbf{ret}} : \forall \{A\},\ A \to \mathsf{M}\ A$
$|\ \textsf{\textbf{bind}} : \forall \{A\}\ \{B\},$
$\quad \mathsf{M}\ A \to (A \to \mathsf{M}\ B) \to \mathsf{M}\ B$
$|\ \textsf{\textbf{mtry}} : \forall \{A\},\ \mathsf{M}\ A \to (\mathsf{Exception} \to \mathsf{M}\ A) \to \mathsf{M}\ A$
$|\ \textsf{\textbf{raise}} : \forall \{A\},\ \mathsf{Exception} \to \mathsf{M}\ A$
$|\ \mathsf{tfix1} : \forall \{A\}\ \{B\!: A \to \mathsf{Type}\},$
$\quad ((\forall\ x : A,\ \mathsf{M}\ (B\ x)) \to (\forall\ x : A,\ \mathsf{M}\ (B\ x))) \to$
$\quad \forall\ x : A,\ \mathsf{M}\ (B\ x)$

$|\ \textsf{\textbf{is\_var}} : \forall \{A\},\ A \to \mathsf{M}\ \mathsf{bool}$
$|\ \textsf{\textbf{nu}} : \forall \{A\ B\},\ \mathsf{string} \to \mathsf{option}\ A \to (A \to \mathsf{M}\ B) \to$
$\quad \mathsf{M}\ B$
$|\ \textsf{\textbf{abs\_fun}} : \forall \{A\ P\}\ (x : A),\ P\ x \to \mathsf{M}\ (\forall\ x,\ P\ x)$
$|\ \textsf{\textbf{abs\_let}} : \forall \{A\ P\}\ (x : A)\ (t : A),\ P\ x \to$
$\quad \mathsf{M}\ (\textsf{\textbf{let}}\ x := t\ \textsf{\textbf{in}}\ P\ x)$
$|\ \textsf{\textbf{abs\_prod}} : \forall \{A\}\ (x : A),\ \mathsf{Type} \to \mathsf{M}\ \mathsf{Type}$
$|\ \textsf{\textbf{abs\_fix}} : \forall \{A\},\ A \to A \to \mathsf{N} \to \mathsf{M}\ A$
$|\ \textsf{\textbf{get\_binder\_name}} : \forall \{A\},\ A \to \mathsf{M}\ \mathsf{string}$
$|\ \textsf{\textbf{remove}} : \forall \{A\ B\},\ A \to \mathsf{M}\ B \to \mathsf{M}\ B$

$|\ \textsf{\textbf{evar}} : \forall\ (A : \mathsf{Type}),\ \mathsf{option}\ (\mathsf{list}\ \mathsf{Hyp}) \to \mathsf{M}\ A$
$|\ \textsf{\textbf{is\_evar}} : \forall \{A\},\ A \to \mathsf{M}\ \mathsf{bool}$

$|\ \textsf{\textbf{print}} : \mathsf{string} \to \mathsf{M}\ \mathsf{unit}$
$|\ \textsf{\textbf{pretty\_print}} : \forall \{A\},\ A \to \mathsf{M}\ \mathsf{string}$

$|\ \textsf{\textbf{hypotheses}} : \mathsf{M}\ (\mathsf{list}\ \mathsf{Hyp})$

$|\ \textsf{\textbf{destcase}} : \forall \{A\}\ (a : A),\ \mathsf{M}\ (\mathsf{Case})$
$|\ \textsf{\textbf{constrs}} : \forall \{A\}\ (a : A),\ \mathsf{M}\ (\mathsf{dyn} \times (\mathsf{list}\ \mathsf{dyn}))$
$|\ \textsf{\textbf{makecase}} : \forall\ (C : \mathsf{Case}),\ \mathsf{M}\ \mathsf{dyn}$

$|\ \textsf{\textbf{munify}}\ \{A\}\ (x\ y : A) : \mathsf{Unif} \to \mathsf{M}\ (\mathsf{option}\ (x = y))$
$|\ \textsf{\textbf{munify\_cumul}}\ \{A\ B\} : A \to B \to \mathsf{Unif} \to \mathsf{M}\ \mathsf{bool}$

In the following figures we show the big-step semantics of the language. The judgment

$$\Sigma; \Gamma \vdash t \Downarrow (\Sigma'; v)$$

specifies: under local context Γ and meta-context Σ, evaluating code t produces a new meta-context Σ' and a value v. Values are: **ret** e or **raise** e, for some Coq term e.

CONJECTURE 1 (Type soundness). *If* $\Sigma; \Gamma \vdash t : \mathsf{M}\ \tau$ *and there exist* t' *and* Σ' *such that* $\Sigma; \Gamma \vdash t \Downarrow (\Sigma'; v)$ *then* Σ' *is an extension of* Σ *and* $\Sigma'; \Gamma \vdash v : \mathsf{M}\ \tau$ *.*

$\boxed{\text{Values}}$

ERET
$$\frac{}{\Sigma; \Gamma \vdash \textsf{\textbf{ret}}\ e \Downarrow (\Sigma; \textsf{\textbf{ret}}\ e)}$$

ERAISE
$$\frac{\mathsf{FV}(e) \notin \Gamma \qquad \mathsf{FMV}(e) = \emptyset}{\Sigma; \Gamma \vdash \textsf{\textbf{raise}}\ e \Downarrow (\Sigma; \textsf{\textbf{raise}}\ e)}$$

EFAILURE
$$\frac{\Sigma; \Gamma \vdash t \not\Downarrow}{\Sigma; \Gamma \vdash t \Downarrow (\Sigma; \textsf{\textbf{raise}}\ \mathsf{Failure})}$$

$\boxed{\text{Reduction}}$

EWHD
$$\frac{\Sigma; \Theta, \Gamma \vdash t \overset{\mathsf{whd}}{\rightsquigarrow}_{\beta\delta\iota} t' \qquad \Sigma; \Gamma \vdash t' \Downarrow (\Sigma; v)}{\Sigma; \Gamma \vdash t \Downarrow (\Sigma; v)}$$

EREDUCE
$$\frac{\Sigma; \Theta, \Gamma \vdash t \overset{\hat{r}}{\rightsquigarrow} t' \qquad \Sigma; \Gamma \vdash u\{t'/x\} \Downarrow (\Sigma; v)}{\Sigma; \Gamma \vdash \textsf{\textbf{let}}\ x := \mathsf{reduce}\ r\ t\ \textsf{\textbf{in}}\ u \Downarrow (\Sigma; v)}$$

ELET
$$\frac{t\text{'s head is not reduce} \qquad \Sigma; \Gamma \vdash u\{t/x\} \Downarrow (\Sigma; v)}{\Sigma; \Gamma \vdash \textsf{\textbf{let}}\ x := t\ \textsf{\textbf{in}}\ u \Downarrow (\Sigma; v)}$$

$\boxed{\text{Standard operations}}$

EFIX
$$\frac{\Sigma; \Gamma \vdash f\ (\textsf{\textbf{mfix}}\ f)\ t \Downarrow (\Sigma; v)}{\Sigma; \Gamma \vdash \textsf{\textbf{mfix}}\ f\ t \Downarrow (\Sigma; v)}$$

EBINDR
$$\frac{\Sigma; \Gamma \vdash t \Downarrow (\Sigma'; \textsf{\textbf{ret}}\ e)}{\Sigma; \Gamma \vdash \textsf{\textbf{bind}}\ t\ f \Downarrow (\Sigma'; f\ e)}$$

EBINDE
$$\frac{\Sigma; \Gamma \vdash t \Downarrow (\Sigma'; \textsf{\textbf{raise}}\ e)}{\Sigma; \Gamma \vdash @\textsf{\textbf{bind}}\ \tau\ \tau'\ t\ f \Downarrow (\Sigma; @\textsf{\textbf{raise}}\ \tau'\ e)}$$

ETRYR
$$\frac{\Sigma; \Gamma \vdash t \Downarrow (\Sigma'; \textsf{\textbf{ret}}\ e)}{\Sigma; \Gamma \vdash \textsf{\textbf{mtry}}\ t\ f \Downarrow (\Sigma'; \textsf{\textbf{ret}}\ e)}$$

ETRYE
$$\frac{\Sigma; \Gamma \vdash t \Downarrow (\Sigma'; \textsf{\textbf{raise}}\ e)}{\Sigma; \Gamma \vdash \textsf{\textbf{mtry}}\ t\ f \Downarrow (\Sigma; f\ e)}$$

Variable handling

EIsVar

$$\frac{b = e \text{ is a variable}}{\Sigma; \Gamma \vdash \textsf{\textbf{is\_var}}\ e \Downarrow (\Sigma; \textsf{\textbf{ret}}\ \check{b})}$$

ENu

$$\frac{\hat{x} \notin \Theta, \Gamma \qquad \Sigma; \Gamma, \hat{x} := u : \tau \vdash t \Downarrow (\Sigma'; v) \qquad \text{if } v = \textsf{\textbf{ret}}\ e \text{ then } \hat{x} \notin \mathsf{FV}(e)}{\Sigma; \Gamma \vdash \nu x := u : \tau, t \Downarrow (\Sigma'; v)}$$

EAbsFun

$$\frac{\mathsf{deps}(\hat{x}, e) \subseteq \{\hat{x}\} \qquad \mathsf{deps}(\hat{x}, \rho) = \emptyset}{\Sigma; \Gamma \vdash @\textsf{\textbf{abs\_fun}}\ \tau\ \rho\ x\ e \Downarrow (\Sigma; \textsf{\textbf{ret}}\ (\lambda y \Rightarrow e\{y/x\}))}$$

EAbsProd

$$\frac{\mathsf{deps}(\hat{x}, \tau') \subseteq \{\hat{x}\}}{\Sigma; \Gamma \vdash @\textsf{\textbf{abs\_prod}}\ \tau\ x\ \tau' \Downarrow (\Sigma; \textsf{\textbf{ret}}\ (\forall y, \tau'\{y/x\}))}$$

EAbsLet

$$\frac{\mathsf{deps}(\hat{x}, e) \subseteq \{\hat{x}\} \qquad \mathsf{deps}(\hat{x}, \rho) = \emptyset}{\Sigma; \Gamma \vdash @\textsf{\textbf{abs\_let}}\ \tau\ \rho\ x\ e\ e' \Downarrow (\Sigma; \textsf{\textbf{ret}}\ (\textsf{\textbf{let}}\ y := e\ \textsf{\textbf{in}}\ e'\{y/x\}))}$$

EAbsFix

$$\frac{\mathsf{deps}(\hat{x}, e) \subseteq \{\hat{x}\} \qquad \tau \text{ has at least } \hat{n} \text{ products}}{\Sigma; \Gamma \vdash @\textsf{\textbf{abs\_fix}}\ \tau\ x\ e\ n \Downarrow (\Sigma; \textsf{\textbf{ret}}\ (\textsf{\textbf{fix}}_n\ f \Rightarrow e\{f/x\}))}$$

EGetBinderName

$$\frac{e = x \in \Gamma \text{ or } e = \lambda x \Rightarrow e' \text{ or } e = \forall x, e' \text{ or } e = \textsf{\textbf{let}}\ x := d\ \textsf{\textbf{in}}\ e'}{\Sigma; \Gamma \vdash \textsf{\textbf{get\_binder\_name}}\ e \Downarrow (\Sigma; \textsf{\textbf{ret}}\ \text{``}\check{x}\text{''})}$$

ERemove

$$\frac{\hat{x} \notin \mathsf{FV}(\Gamma_2) \qquad \hat{x} \notin \mathsf{FV}(e) \qquad \Sigma; \Gamma_1, \Gamma_2 \vdash e \Downarrow (\Sigma'; v)}{\Sigma; \Gamma_1, x := d : \tau, \Gamma_2 \vdash \textsf{\textbf{remove}}\ x\ e \Downarrow (\Sigma'; v)}$$

Meta-Variables

EIsEVar

$$\frac{b = e\text{'s head is an undefined meta-variable}}{\Sigma; \Gamma \vdash \textsf{\textbf{is\_evar}}\ e \Downarrow (\Sigma; \textsf{\textbf{ret}}\ \check{b})}$$

EEvarSome

$$\frac{?x \notin \mathsf{dom}(\Sigma) \qquad e = \mathsf{Some}\ \Gamma' \qquad \Sigma \vdash \hat{\Gamma}'}{\Sigma; \Gamma \vdash \textsf{\textbf{evar}}\ \tau\ e \Downarrow (\Sigma, ?x : \tau[\hat{\Gamma}']; \textsf{\textbf{ret}}\ ?x[\mathrm{id}_{\Gamma'}])}$$

EEvarNone

$$\frac{?x \notin \mathsf{dom}(\Sigma)}{\Sigma; \Gamma \vdash \textsf{\textbf{evar}}\ \tau\ \mathsf{None} \Downarrow (\Sigma, ?x : \tau[\Gamma]; \textsf{\textbf{ret}}\ ?x[\mathrm{id}_{\Gamma}])}$$

Unification

EMunify

$$\frac{\Sigma; \Gamma \vdash e \approx_r e' \triangleright \Sigma'}{\Sigma; \Gamma \vdash \textsf{\textbf{munify}}\ e\ e'\ r \Downarrow (\Sigma'; \textsf{\textbf{ret}}\ (\mathsf{Some}\ (\mathsf{eq\_refl}\ \mathsf{e})))}$$

EMunifyNot

$$\frac{\Sigma; \Gamma \vdash e \not\approx_r e'}{\Sigma; \Gamma \vdash \textsf{\textbf{munify}}\ e\ e'\ r \Downarrow (\Sigma; \textsf{\textbf{ret}}\ \mathsf{None})}$$

EMunifyCumul

$$\frac{\Sigma; \Gamma \vdash e \lessapprox_r e' \triangleright \Sigma'}{\Sigma; \Gamma \vdash \textsf{\textbf{munify\_cumul}}\ e\ e' r \Downarrow (\Sigma'; \textsf{\textbf{ret}}\ \mathsf{true})}$$

EMunifyCumulNot

$$\frac{\Sigma; \Gamma \vdash e \not\lessapprox_r e'}{\Sigma; \Gamma \vdash \textsf{\textbf{munify\_cumul}}\ e\ e' r \Downarrow (\Sigma; \textsf{\textbf{ret}}\ \mathsf{false})}$$

Printing

EPrint

$$\frac{\text{print } s \text{ to stdout}}{\Sigma; \Gamma \vdash \textsf{\textbf{print}}\ s \Downarrow (\Sigma; \textsf{\textbf{ret}}\ \langle\rangle)}$$

EPrettyPrint

$$\frac{\text{converts } e \text{ to string } s}{\Sigma; \Gamma \vdash \textsf{\textbf{pretty\_print}}\ e \Downarrow (\Sigma; \textsf{\textbf{ret}}\ \check{s})}$$