Mtac2: Typed Tactics for Backward Reasoning in Coq

JAN-OLIVER KAISER, MPI-SWS, Germany
BETA ZILIANI, CONICET and FAMAF, UNC, Argentina
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
YANN RÉGIS-GIANAS, IRIF, CNRS, Paris Diderot, and INRIA PI.R2, France
DEREK DREYER, MPI-SWS, Germany

Coq supports a range of built-in tactics, which are engineered primarily to support *backward reasoning*. Starting from a desired goal, the Coq programmer can use these tactics to manipulate the proof state interactively, applying axioms or lemmas to break the goal into subgoals until all subgoals have been solved. Additionally, it provides support for *tactic programming* via OCaml and Ltac, so that users can roll their own custom proof automation routines.

Unfortunately, though, these tactic languages share a significant weakness. They do not offer the tactic programmer any static guarantees about the soundness of their custom tactics, making large tactic developments difficult to maintain. To address this limitation, Ziliani et al. previously proposed **Mtac**, a new typed approach to custom proof automation in Coq which provides the static guarantees that OCaml and Ltac are missing. However, despite its name, Mtac is really more of a metaprogramming language than it is a full-blown tactic language: it misses an essential feature of tactic programming, namely the ability to directly manipulate Coq's proof state and perform backward reasoning on it.

In this paper, we present **Mtac2**, a next-generation version of Mtac that combines its support for typed metaprogramming with additional support for the programming of backward-reasoning tactics in the style of Ltac. In so doing, Mtac2 introduces a novel feature in tactic programming languages—what we call *typed backward reasoning*. With this feature, Mtac2 is capable of statically ruling out several classes of errors that would otherwise remain undetected at tactic definition time. We demonstrate the utility of Mtac2's typed tactics by porting several tactics from a large Coq development, the Iris Proof Mode, from Ltac to Mtac2.

CCS Concepts: • **Theory of computation → Type theory**; **Proof theory**;

Additional Key Words and Phrases: Theorem Proving, Tactic Languages, Metaprogramming, Dependent Types, Coq

ACM Reference Format:
Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: Typed Tactics for Backward Reasoning in Coq. *Proc. ACM Program. Lang.* 2, ICFP, Article 78 (September 2018), 31 pages. https://doi.org/10.1145/3236773

78

1 INTRODUCTION

The Coq proof assistant provides a rich dependently-typed framework in which to formalize mathematics and programming language metatheory. Although Coq proofs ultimately compile down to proof terms in the language of Type Theory, it is not practical for Coq programmers to

Authors' addresses: Jan-Oliver Kaiser, MPI-SWS*, janno@mpi-sws.org; Beta Ziliani, CONICET and FAMAF, UNC, beta@mpi-sws.org; Robbert Krebbers, Delft University of Technology, mail@robbertkrebbers.nl; Yann Régis-Gianas, IRIF, CNRS, Paris Diderot, and INRIA PI.R2, yrg@pps.univ-paris-diderot.fr; Derek Dreyer, MPI-SWS*, dreyer@mpi-sws.org.
* Saarland Informatics Campus.

2475-1421/2018/9-ART78
https://doi.org/10.1145/3236773

write such proof terms manually. Instead, to make proofs feasible to construct, Coq supports a range of built-in *tactics*, which are engineered primarily to support *backward reasoning*. Starting from a desired goal, the Coq programmer can use these tactics to manipulate the proof state interactively, applying axioms or lemmas to break the goal into subgoals until all subgoals have been solved. This style of backward reasoning is convenient because Coq can often infer many details about how to instantiate lemmas by inspection of the current proof state, and as a result the Coq user can omit these tedious details from their interactive proof scripts.

In addition, Coq supports two *tactic languages*—OCaml and Ltac—that enable users to roll their own tactics. With these languages, Coq users can build custom proof automation routines, which simplify proof scripts further by exploiting domain-specific knowledge about the structure of proofs. OCaml is the language used to implement Coq itself, and the main benefit of using it is performance: tactics written in OCaml are compiled rather than interpreted, and can make use of imperative data structures for efficiency. That said, it is not ideal for general-purpose tactic programming because it exposes developers to low-level and potentially unsafe details of Coq's internals, *e.g.,* forcing them to work directly with de Bruijn representations of terms. In contrast, Ltac is a higher-level dynamic language that allows for rapid prototyping of tactics directly within the Coq environment. Ltac is an interpreted language, and thus less efficient than OCaml, but it provides a more convenient representation of proof terms using the concrete syntax of Coq.

Unfortunately, though, both OCaml and Ltac share a common, and rather significant, weakness. Neither offers the tactic programmer any static guarantees about the soundness of their custom tactics: it is easy to define tactics that are accepted by the OCaml compiler (or Ltac interpreter) but that construct ill-typed terms or generate inscrutable errors when they are applied.

To address this limitation, Ziliani et al. [2013, 2015] recently proposed **Mtac**, a new interpreted language for custom proof automation in Coq which provides the static guarantees that OCaml and Ltac are missing. Mtac is based on the key realization that, at its core, tactic programming is essentially functional programming—of the sort already supported by Coq's functional language Gallina—extended with certain *effects*, such as general recursion, syntax inspection, and exception handling. Although these effectful operations are not directly available in Gallina, their static semantics can be described using the *type structure* of Gallina, and they can thus be supported by means of a monadic extension to Gallina—not unlike the way Haskell extends its pure functional core with computational effects via the `IO` monad. Specifically, Mtac extends Coq with a monadic type `M A`, representing the type of Mtac tactics that, when applied, may diverge or fail, but if they terminate successfully, will produce a Coq term of type `A`.

However, despite its name, Mtac is really more of a metaprogramming language than it is a full-blown tactic language. In particular, compared to OCaml and Ltac, it is missing an essential feature of tactic programming, namely the ability to directly manipulate Coq's proof state and perform backward reasoning on it.

In this paper, we present **Mtac2**, a next-generation version of Mtac that combines its support for typed metaprogramming with additional support for the programming of backward-reasoning tactics in the style of Ltac. In so doing, Mtac2 introduces a novel feature in tactic programming languages—what we call *typed backward reasoning*.

1.1 An Example Motivating Typed Backward Reasoning

To motivate the desire for Mtac2's typed backward reasoning more concretely, let us now turn to a real example of tactic programming taken from recent work on the Iris Proof Mode (IPM) [Krebbers et al. 2017]—a tactic library for supporting interactive development of separation-logic proofs in Coq. It has been used for a variety of applications, including program refinements [Krebbers et al. 2017; Timany et al. 2018], program logics for relaxed memory models and object capability

patterns [Kaiser et al. 2017; Swasey et al. 2017], and a safety proof for a realistic subset of the Rust programming language [Jung et al. 2018].

To give a brief impression of IPM, we provide an example proof state of a program verification performed in Iris. The program in question adds the number `4` to itself, then adds the result to the contents of location `x`, which we know to contain `8`. The program's postcondition guarantees that the resulting value is `16` and that ownership of `x` (still with value `8`) is returned.

```
1  "H" : x ↦ 8
2  ----------------------------------------*
3  WP let: "t" := 4 + 4 in
4     !x + "t" {{ v, v = 16 ∗ x ↦ 8 }}
```

To talk about separation logic hypotheses (as opposed to Coq hypotheses), the proof mode introduces new contexts.[1] The context shown in the example above (delimited by `---∗`) is the *spatial* context; it contains assertions describing ownership of resources, such as the *points-to* assertion `x ↦ 8`, which asserts ownership of the location `x` with value `8`.

The remaining part of the proof state is the IPM goal. In our case, we are proving an Iris weakest precondition, *i.e.,* a proof of partial program correctness. The syntax used here is `WP e {{ v, Φ v }}`, which denotes the weakest precondition implying that expression `e` is safe to execute and that its resulting value `v` (should it have one) satisfies the postcondition `Φ`.

IPM offers a variety of tactics to manage separation logic contexts, to handle the various separation logic constructs (separating conjunction, magic wand, higher-order quantification, etc.), and to apply existing separation logic lemmas and theorems. Moreover, it also offers a set of tactics to perform symbolic execution. It is one of these symbolic execution tactics that we focus on in our demonstration: `wp_pure`, a tactic to symbolically execute pure program steps such as beta reductions and arithmetic operations.

In our program above, the next step of computation is an addition of two integers. To instruct IPM to symbolically execute the operation, we invoke `wp_pure (_ + _)`.[2] The proof mode uses type classes to find a sufficient condition for executing the operation, as well as to find the result of the reduction. A sufficient condition for division, for example, could be that the denominator is not `0`. For the addition operation in our example, this condition is trivial, *i.e.,* `True`, and hence automatically solved. The remaining program, now containing the result of the addition, is left as an open goal for the user to prove:

```
1  "H" : x ↦ 8
2  ----------------------------------------*
3  WP let: "t" := 8 in
4     !x + "t" {{ v, v = 16 ∗ x ↦ 8 }}
```

At the core of the symbolic execution infrastructure lies a set of lemmas that justify single steps of symbolic execution. As an example, consider the lemma `tac_wp_pure` in Figure 1, which serves as an umbrella lemma for symbolically executing pure execution steps. To understand the type of `tac_wp_pure`, we need to take a peek behind the sugary syntax curtains. First of all, the proof states shown above are merely syntactic sugar for the underlying entailment relation `envs_entails` Δ P. Here, Δ is the separation logic context, and P the IPM conclusion (or IPM goal). Secondly, the weakest precondition assertions shown in the example above (written `WP e @ s ; E {{ Φ }}` in

[1] Apart from the spatial context shown in the example, IPM also features a *persistent* context. We ignore the persistent context for the purpose of this paper.

[2] Note that it is possible to give a wildcard expression, `_`, to `wp_pure` in which case the tactic will symbolically execute the first pure expression it finds. It is, however, sometimes useful for proof maintainability to explicitly specify the shape of the expression to be executed.

```
1  Lemma tac_wp_pure `{heapG Σ} Δ s E e1 e2 φ Φ :
2    PureExec φ e1 e2 ->
3    φ ->
4    envs_entails Δ (wp s E e2 Φ) ->
5    envs_entails Δ (wp s E e1 Φ).
6
7  Tactic Notation "wp_pure" open_constr(epat) :=
8    lazymatch goal with
9    | |- envs_entails _ (wp ?s ?E ?e ?Φ) =>
10     let e := eval simpl in e in
11     reshape_expr e ltac:(fun K e' =>
12       unify e' epat;
13       eapply (tac_wp_pure _ _ _ (fill K e'));
14       [apply _                          (* PureExec *)
15       |try fast_done                    (* The pure condition for PureExec *)
16       |wp_expr_simpl_subst; try wp_value_head (* new goal *)
17       ])
18    || fail "wp_pure: cannot find" epat "in" e "or" epat "is not a reduct"
19   | _ => fail "wp_pure: not a 'wp'"
20   end.
```

Fig. 1. An IPM lemma, tac_wp_pure, describing the effect of symbolically executing a pure expression; and the Ltac code for wp_pure, a tactic to symbolically execute pure reduction steps.

its full form) is syntactic sugar for `wp s E e Φ`. The arguments `s` and `E` allow Iris to express more general notions of weakest precondition and can be ignored for the purpose of this paper.

The first hypothesis of `tac_wp_pure` is a type class, `PureExec`, which proves that the proposition φ is a sufficient condition for `e1` to reduce to `e2`. Note that in this type class search problem, `e1` is taken as input—it is derived from the current goal—and both φ *and* `e2` are considered outputs. For example, `PureExec (y≠0) (#x div #y) (#(x div y))` is a proof that the integer division of `x` by `y` reduces if the denominator is not `0`, *and* that the result of the reduction is `#(x div y)`, where `div` is now the meta-level integer division function. (The `#` notation denotes integer literals in the expression language.) Thus, once we have established `PureExec` φ `e1 e2`, to prove `env_entails Δ (WP e1 {{ Φ }})`, it suffices to show φ and `env_entails Δ (WP e2 {{ Φ }})`.

IPM provides wrapper tactics for lemmas like `tac_wp_pure`. These tactics perform the necessary pre- and post-processing and error reporting in case of failure. The Ltac tactic wrapping `tac_wp_pure`, called `wp_pure`, is presented in Figure 1 (from Line 7 onward).

The tactic first checks that the IPM goal matches the weakest precondition assertion. If that is the case, it attempts to find an occurrence of the given argument (`epat`) in evaluation position and apply `tac_wp_pure`.

The search for expressions in evaluation position is taken care of by the helper tactic `reshape_expr`. To understand this tactic, note that the language used in the example is based on *evaluation contexts*. To find expressions in evaluation position, it thus suffices to decompose the expression `e` into evaluation contexts `K` and the corresponding redexes `e'`, such that `K` and `e'` can be "plugged" together to yield `e`. This plugging operation is called `fill` in Iris. We give more details on evaluation contexts in Section 5.

The job of `reshape_expr` is to enumerate all possible decompositions of `e` into `K` and `e'` and call its continuation with each decomposition until the continuation succeeds. In our example, the continuation—the remainder of `wp_pure`—makes sure to filter out any redexes `e'` which do not match

the given pattern `epat`. Once a suitable candidate has been found, `wp_pure` applies `tac_wp_pure`. The various subgoals are either dispatched by type class search (`apply _`), treated in a best effort way (`try fast_done` for the proof of φ), or—in the case of the last goal—carefully massaged by simplifying the remaining expression and by checking if we have reached the end of our program, in which case only the postcondition remains to be proven.

The tactic as written above is probably correct. However, Ltac is not helping the developer to enforce that. To see that, let us look at various ways that writing the tactic could have gone wrong:

(1) With Ltac's `lazymatch` we ensure that the shape of the goal matches the conclusion of `tac_wp_pure`. However, Ltac does not guarantee that the lemma we give to `eapply` matches the goal specified in the `lazymatch` branch and we could have tried to apply a completely different lemma. We should be able to enforce that the conclusion of the lemma given to `eapply` indeed matches the shape of the goal.
(2) Similarly, Ltac does not even check if our invocation of `tac_wp_pure` is well-formed, *i.e.,* that its arguments are of the correct type.
(3) Additionally, Ltac guarantees neither that follow-up tactics can be applied to their respective subgoals nor that we supplied a reasonable number of follow-up tactics (*i.e.,* not too many).
(4) Lastly, Ltac offers very limited error raising and handling facilities. The mechanism is not based on ML-like exceptions but instead on the concept of *failures at level n*. Handlers such as the `try` combinator catch errors at level 0 and otherwise decrease the level and propagate the error. This makes it difficult to write modular error handling code that distinguishes intended failure—*e.g.,* for driving the backtracking in `reshape_expr`—from other, unintended failure.

In short: Any change to the various pieces involved in `wp_pure`—IPM definitions, Iris assertions such as `WP`, follow-up tactics, `tac_wp_pure`, and other supporting lemmas—may lead to runtime failures that could have easily been prevented if tactics supported *static typing*.

Using Mtac2 we can write such statically-typed tactics, and in Section 5, we present an Mtac2 version of `wp_pure` which addresses all the shortcomings of the Ltac version:

(1) It statically ensures that the conclusion of `tac_wp_pure` applied to the given arguments matches the current goal.
(2) It statically ensures that each of the arguments given to `tac_wp_pure` has the right type.
(3) It statically ensures that all remaining arguments (*i.e.,* hypotheses) to `tac_wp_pure` are provided for by the follow-up tactics, *and* that each of these follow-up tactics can solve its respective subgoal.
(4) It uses exceptions to drive the backtracking in `reshape_expr` and thus avoids any confusion between backtracking and actual failure.

1.2 Contributions

This paper makes the following contributions:

- We develop Mtac2, an extension of Mtac to support backward reasoning. In particular, we introduce dynamically-typed backward reasoning in the style of Ltac (Section 3). Unlike in Ltac, *all* Mtac2 tactics are defined in Coq itself.
- We further introduce the concept of *typed tactics for backward reasoning*, which follows naturally from the combination of Coq's rich type system and Mtac2's tactic infrastructure (Section 4). With typed tactics, Mtac2 is capable of statically ruling out several classes of errors that would otherwise remain undetected at tactic definition time.
- We demonstrate the utility of Mtac2's typed tactics by porting several tactics from the Coq development of Iris Proof Mode from Ltac to Mtac2 (Section 5).

- We substantially modify the Mtac language to support several missing features and to overcome difficulties that arose during the construction of realistic tactics (Section 6).

Mtac2, along with all the examples in this paper, is available online [Kaiser et al. 2018].

2 THE CORE OF MTAC BY EXAMPLE

In this section, we provide an overview of the basic concepts of the Mtac framework that are shared between the original Mtac1 (*i.e.,* the Mtac implementation as described by Ziliani et al. [2015]) and our new implementation of Mtac2 as described in this paper. This is largely review, since most of these concepts are inherited from the original Mtac1. However, they form the essential building blocks of Mtac2's tactic infrastructure, which we will introduce in the sections that follow. As we proceed, we will note (with footnotes) any points where Mtac2 diverges from Mtac1.

Mtac is a framework for typed metaprogramming in Coq. Mtac takes an unusual approach to metaprogramming in that it represents metaprograms using the language of Coq itself, enlisting Coq's strong type system to ensure the framework's primary guarantee: If a well-typed metaprogram produces a value, that value will be well-typed.[3]

Mtac's primitives are members of a type family, denoted `M`, that forms a monad. The types and syntax of **`bind`** and **`ret`** are given below. The second syntax for bind, **`('(x) <- ta; tb)`**, combines bind and irrefutable pattern matching on the result into a single syntactic construct. It uses Coq's built-in syntax for irrefutable pattern matching on function arguments written (**`fun '(x) => tb`**). As an example, we write **`('(a,b) <- ta; tb)`** to deconstruct a pair.

```
bind: forall {X Y : Type},
        M X -> (X -> M Y) -> M Y
ret: forall {X : Type}, X -> M X
```

```
      x <- ta; tb ≜ bind ta (fun x => tb)
'(x) <- ta; tb ≜ bind ta (fun '(x) => tb)
   ta >>= fb ≜ bind ta fb
```

Mtac uses this monad to encapsulate non-termination and the possibility of failure during the execution of metaprograms. To deal with failure, Mtac offers an exception mechanism akin to that of many languages in the ML family. We can thus express Mtac's guarantee more concretely: Executing a metaprogram of type `M T` will either diverge, raise an (uncaught) exception, or produce a value of type `T`.[4]

Apart from the monadic operations, `M` provides a broad variety of useful primitives, allowing users to call various unification algorithms; inspect, destruct, and construct terms; etc. We illustrate the various primitives of Mtac by developing a simple, incomplete, first-order tautology solver. To keep this tutorial short, we focus on the introduction of connectives and do not deal with the elimination of connectives in hypotheses. The code of this metaprogram is given in Figure 2.

At high-level, the metaprogram `solve_tauto` finds the proof of proposition `P` in two steps: first, by looking for a hypothesis of type `P` in the list `l`, in which case that hypothesis constitutes the proof; and second, if it cannot find such an hypothesis, by decomposing the proposition `P` according to its shape, and recursively calling the solver on the resulting subcomponents of `P`.

We defer for the moment the explanation of the `lookup` metaprogram used to search the list of hypothesis (Line 3), and only point out that it happens before we inspect `P` to avoid unnecessarily taking apart propositions for which we already have a proof.

Mtac offers the **`mmatch`** construct to analyze the shape of arbitrary Coq terms. It takes a scrutinee and a list of branches, each containing a unification candidate and a metaprogram. **`mmatch`** attempts to

[3]Modulo the guard condition on fixed-points appearing in the produced value, which is not statically guaranteed to hold by Mtac metaprograms.

[4]In Mtac2 we modified the original semantics of Mtac1 to never get *stuck*. Indeed, in [Ziliani et al. 2013] certain dynamic checks will block the execution. We modified this behavior to always raise an exception. For instance, if the interpreter encounters an *irreducible* term that is not a primitive of the language, it raises the `StuckTerm` exception.

```
1  Definition solve_tauto : forall (l : list dyn) {P : Prop}, M P :=
2    mfix2 f (l : list dyn) (P : Prop) : M P :=
3      mtry lookup P l
4      with NotFound =>
5        mmatch P as P' return M P' with
6        | True => ret I
7        | [? Q1 Q2] Q1 ∧ Q2 =>
8          q1 <- f l Q1; q2 <- f l Q2; ret (conj q1 q2)
9        | [? Q1 Q2] Q1 ∨ Q2 =>
10         mtry q1 <- f l Q1; ret (or_introl q1)
11         with TautoFail => q2 <- f l Q2; ret (or_intror q2) end
12       | [? (Q1 Q2 : Prop)] Q1 -> Q2 =>
13         ν q1, q2 <- f (Dyn q1 :: l) Q2; abs_fun q1 q2
14       | [? X (Q : X -> Prop)] (exists x : X, Q x) =>
15         x <- evar X; q <- f l (Q x); b <- is_evar x;
16         if b then raise TautoFail else ret (ex_intro Q x q)
17       | _ => raise TautoFail
18       end
19     end.
```

Fig. 2. A very simple first-order tautology solver in Mtac1.

unify the scrutinee with each of these candidates. When a match is found, the code in the respective branch is executed. Note that subsequent failures inside the branch will *not* automatically cause Mtac to backtrack and try the remaining branches. Mtac's `mmatch` is thus closer to Ltac's `lazymatch` and to OCaml's pattern-matching than to Ltac's `match`.

In Line 5, we `mmatch` the proposition `P` against 5 possible shapes, which represent the connectives our solver can handle: `True`, conjunction, disjunction, implication, existential quantification. We also add a catch-all branch (Line 17) to explicitly raise a meaningful exception if the solver is given a connective which it does not support.

In each branch of `mmatch`, the `[? x1 .. xn]` syntax introduces meta-variables `x1` ... `xn` (also called unification variables or existential variables) which the unification algorithm can instantiate. These variables are available in the code of their respective branch.

The most interesting aspect of Mtac's `mmatch` is that—unlike its Ltac counterpart `lazymatch`—it is *statically typed*. Moreover, like Coq's own `match`, Mtac's `mmatch` is an instance of dependent pattern matching. First of all, the type of `mmatch` is expressed as a function of the scrutinee. Mtac imitates Coq's `match` syntax, allowing users to specify the type of `mmatch` with the familiar `mmatch x as x' return P x' with ..` syntax.[5] Second, the type of each branch is specialized by substituting the scrutinee with the branch's candidate.

As an illustration of the dependently-typed nature of Mtac's `mmatch`, consider the case for `True` on Line 6: Instead of having to provide a proof of the proposition `P`—a rather impossible task—we are allowed to construct a proof of `True` instead, since the typechecker is aware of the equivalence between `P` and `True`. Indeed, the `mmatch`'s return type, which we can think of as `fun P' => M P'`, has been instantiated with `True` to yield `M True`. This is exactly the type of `ret I`, where `I` is the constructor of `True`.

Not all cases are as simple as the one for `True`. We turn our attention towards the cases for conjunction and disjunction. To handle these, we want to recursively traverse `Q1` and, if necessary,

[5]The `mmatch` syntax imposes a syntactic restriction on the return type: it has to be an application of `M`. This is not a restriction for any of the examples presented in the paper.

`Q2`. To this end, Mtac offers a general fixed-point combinator, called **`mfix`**,[6] which we can use to acquire proofs of `Q1` and `Q2`. In the case of a conjunction, combining these proofs into a proof of `P` amounts to only two bind operations and an application of the `conj` constructor. If any of the recursive calls raise an exception, it is automatically propagated outwards by the bind operator.

A different strategy is needed for disjunction: We do not know *a priori* which side of the disjunction we will be able to prove. This brings us to the first application of Mtac's backtracking mechanism. To enable backtracking of the program state, Mtac offers the **`mtry`** construct, which executes a given program and handles any exceptions it raises according to a given list of handlers. These handlers behave similarly to **`mmatch`** branches.

The solver uses **`mtry`** in Line 10 to first attempt a proof of the left-hand side of the disjunction. If the recursive call is unable to prove that side of the disjunction, we simply try the other side.

The next case handled by our solver is implication. When proving an implication, we would like to add the left-hand side to a set of facts that can be used to solve the right-hand side. To this end, we use the list of assumptions `l`. This list contains assumptions of heterogeneous types. We encompass this heterogeneity using a universal type called `dyn` which is defined by a single constructor `Dyn: forall {T : Type}, T -> dyn`.[7]

It is worth pointing out that, in the case of implication, we will have to provide a proof term in the form of a function, *i.e.,* `(fun q1 => ...)`. However, we need access to argument `q1` before we return such a function—otherwise, how could we possibly add `q1` to the list of assumptions for the recursive call? This seeming contradiction is resolved by Mtac's **`nu`** construct:

```
nu: forall {X T : Type} (var : name) (x : option X), (X -> M T) -> M T
```

`nu` introduces a new variable in the execution context of the program, having name `var` and type `X`. Then, the continuation is executed providing it with the newly-introduced variable as argument. The result of calling **`nu`** is whatever the continuation returns. Optionally, the variable can be given a body, turning it into a *local definition*. The `name` type enables different ways of indicating the name of the variable. Its most basic constructor, `TheName`, simply contains a string. Mtac defines the notation `(ν x, t)` for `nu (TheName "x") (fun x => t)`.

The counterpart of **`nu`** is Mtac's **`abs_fun`**[8] construct:

```
abs_fun: forall {X T}, X -> T -> M (X -> T).
```

The **`abs_fun`** construct attempts to convert a term `t`, which may mention variable `x`, into a function `fun x′ => t[x′/x]`, where `t[x′/x]` represents the meta-theoretic substitution of `x` by `x′` in `t`.

A note on soundness. The **`nu`** operator by itself appears to be unsound. It is easy to see that `ν (x : False), ret x` claims to produce a proof of `False` out of thin air. Fortunately, Mtac ensures soundness implementing a dynamic check that prevents ν variables from escaping their scope [Ziliani et al. 2015].

To implement the implication case in our solver, we use **`nu`** to introduce a variable `q1 : Q1`, recursively call the solver with the list of assumptions extended by `q1`, and finally abstract away `q1` with **`abs_fun`**.

The one remaining connective is existential quantification. Our approach here is to introduce a meta-variable, recursively call the solver, and continue only if the meta-variable has been instantiated by the recursive call. In this way, we make sure to not leave any dangling meta-variables in our proof of `P`.

[6] For technical reasons, Mtac cannot provide just one fixed-point combinator for all arities and instead provides **`mfix1`** through **`mfix5`**.
[7] In Coq, an argument surrounded by curly braces is an *implicit argument* and, as such, should not be provided explicitly.
[8] This primitive was called `abs` in Mtac1 but has since been renamed to distinguish it from other abstraction primitives.

Mtac offers two primitives related to meta-variables. The first primitive is `evar`, which is used to introduce meta-variables. Its type is:

```
evar: forall {X : Type}, M X.
```

The second primitive is `is_evar`, which checks if the given term is an *uninstantiated* meta-variable:

```
is_evar: forall {X : Type}, X -> M bool.
```

These two primitives suffice to implement the case for existential quantification. However, we have not yet explained how the meta-variables we introduce are actually instantiated. Fortunately, we do not need to introduce additional code. The `lookup` function will instantiate these variables for us. To see why, consider the proposition given to the recursive call in the case of an existential quantification, `Q x`, where `Q` represents the body of the existential, as a function binding the witness, and `x` is the freshly introduced meta-variable. This recursive call, and any further recursive calls on sub-propositions of `Q x`, will call `lookup`, which in turn, attempts to unify the proposition with each assumption. Crucially, this unification attempt—if successful—will instantiate the meta-variables in both sides of the unification task. Thus, by successfully matching up assumptions with the current proposition, `lookup` also takes care of instantiating meta-variables.

The `lookup` metaprogram is given below.

```
1  Definition NotFound : Exception. constructor. Qed.
2  Fixpoint lookup (P : Prop) (l : list dyn) : M P :=
3    match l with
4    | D :: l => mmatch D with [? (p : P)] Dyn p => ret p | _ => lookup P l end
5    |      [] => raise NotFound
6    end.
```

We first define a new exception called `NotFound`. In Mtac, new exceptions are created with opaque definitions of the type `Exception`, whose definition is given below.

```
Inductive Exception := exception.
```

The `Exception` type is a proposition equivalent to `True`, *i.e.,* an inductive type with a single constructor that takes no arguments. To avoid relying on the name of that constructor, the `constructor` tactic is used, which takes care of selecting the constructor automatically.

To make each inhabitant of `Exception` distinguishable from the other inhabitants of this type, we conclude the definition of new exceptions with `Qed`. In Coq, every definition concluded with `Qed` is opaque, *i.e.,* it cannot be unfolded by reduction. Hence, two distinct opaque identifiers of type `Exception` cannot be equated by the typechecker, even though their definitions are the same.

After creating the exception, we proceed to define the `lookup` function as a regular Coq `fixpoint`, which iterates over the list. For each element of the list, this tactic first tries to match the hypothesis with an element `p` of type `P`, the proposition we are looking for. If they match, `lookup` returns that element. If not, it recurses on the tail of the list.

Execution of `solve_tauto`*:* Mtac includes a tactic `mrun` which executes the metaprogram on the current goal. Note the `_` to avoid writing the proposition: it is deduced from the goal.

```
Example test_tauto: 5 = 7 -> exists x, x = 7.
Proof. mrun (solve_tauto [] _). Qed.
```

This ends our brief tour of Mtac. We now focus on adding support for tactic development in Mtac2.

3 MTAC2: ADDING BACKWARD REASONING TO MTAC1

The tautology solver in Section 2 served the purpose of presenting the core of Mtac, but it did not fully demonstrate the expressive power of Mtac2. In this section, we introduce Mtac2's support

for Ltac-style dynamically-typed backward reasoning. In Section 4 we show how we can usefully combine the statically-typed approach of Mtac1 with the convenience of backward reasoning.

In order to see how we support backward reasoning in Mtac2, let us start with a simple example—in Ltac—which is sufficient to highlight some of the important issues we had to resolve.

Example 3.1 (Tactics in Ltac).

```
Lemma sub_0_r : forall n, n - 0 = n.
Proof. intro n. case n; [ | intro n']; reflexivity. Qed.
```

The code above proves that subtracting `0` from any natural number `n` is equal to `n`. Since subtraction is defined by performing pattern matching on its first arguments, this identity does not hold directly by computation. We must perform case analysis.

In this section we focus on how to build a similar proof script in Mtac2. For that, we need to take a sneak peek at how Coq works internally. The first thing to note about this code is that we have five *statements* (code ended with a period). The first one is stating the lemma, the second is the `Proof` command, which tells Coq that we want to write a proof in Ltac, and the last one is `Qed`, which tells Coq that we are done with the proof. Our interest is in what happens in between, in what we call the *tactic statements*. These statements transform the proof state to incrementally build a proof-term.

After issuing the `Proof` command, Coq produces a *goal*. Internally, a goal in Coq is represented by a *meta-variable*, a hole in the proof-term under construction. This meta-variable has a type, namely the lemma we have to prove. In this case, the meta-variable created for the goal, let us call it $?g$, has type `forall n, n - 0 = n`.

For introducing the variable `n`, according to Coq's logic, `intro` solves (instantiates) $?g$ with an abstraction `(fun n:nat => ?`g_1`)`, where $?g_1$ is a new meta-variable of type `n - 0 = n`. But this type only makes sense in a local context where `n` is introduced. Following the tradition of Contextual Modal Type Theory [Nanevski et al. 2008], we write such contexts as $?g_1$: (`n:nat` $\vdash$ `n - 0 = n`), meaning that $?g_1$ is a meta-variable with type `n - 0 = n` in the local context `n:nat`.

Coq executes each tactic statement under the local context of the current goal. For the first tactic statement, that was the empty context. For the second one, it is `n:nat`, the context of $?g_1$.

Continuing, the `case` tactic solves the goal ($?g_1$) with pattern matching (a `match`). The generated `match` contains two subgoals, one for each of its branches: $?g_{11}$: (`n:nat` $\vdash$ `0 - 0 = 0`) and $?g_{12}$: (`n:nat` $\vdash$ `forall n', (S n') - 0 = S n'`). The first one corresponds to the base case, in which `n` is `0`, and the second one corresponds to the inductive case, in which `n` is the successor `S` of some number `n'`.[9] The `case` tactic returns these two subgoals, which the composition operator (the semi-colon) composes with the tactics listed in `[ | intro n']`. The first subgoal is dealt with by the tactic to the left of the `|`, which is implicitly the identity tactic (`idtac`), while the second subgoal is dealt with by the tactic to the right of the `|`, `intro n'`. The output of this composition is, again, two subgoals: $?g_{11}$, unchanged, and the new subgoal $?g_{121}$: (`n:nat, n':nat` $\vdash$ `(S n') - 0 = S n'`). Finally, the second semi-colon calls `reflexivity` on each of these subgoals, trivially solving them by computation.

In Mtac2 we can write the same proof as follows:

Example 3.2 (Proof of Example 3.1 written in Mtac2).

```
MProof. intro n. case n &> [m: idtac | intro n'] &> reflexivity. Qed.
```

At a high level, the two proof scripts look similar. However, in contrast to the Ltac version, all the tactics used in the Mtac2 proof script are defined in Coq. To understand how Mtac2 makes this possible, we answer the following questions:

[9] The numbering of subgoals is meant to reflect the derivation *tree* of the proof. Internally, Coq manages lists of goals.

```
Inductive goal :=
  | Metavar : forall {A}, A -> goal
  | AHyp : forall {A}, (A -> goal) -> goal
  | HypRem : forall {A}, A -> goal -> goal.
```

Fig. 3. The goal type.

(1) What is a tactic (Section 3.1)?
(2) What is a goal (Section 3.2)?
(3) What is tactic composition (Section 3.3)?

To conclude this section, we will present an excerpt of the **cintro** tactic, which is the more general tactic from which Mtac2's **intro** is derived (Section 3.4).

3.1 What Is a Tactic?

Think of **case** as a procedure that takes the element to do case analysis on, and solves the current goal by creating subgoals for each constructor of the inductive type. If we want a functional interface, in which the current goal is explicitly given, the signature can be:

```
Definition case : forall {A}, A -> goal -> M (list goal) := ...
```

The first two arguments describe the (implicit) type `A` and the element of type `A` to do case analysis on. The third argument is the current goal, and the result is an Mtac2 program returning a list of goals. Instead of explicitly mentioning `goals`, we can first define a type alias `tactic` and use that to give **case** a more intuitive type:

```
Definition tactic := goal -> M (list goal).
Definition case : forall {A}, A -> tactic := ...
```

Readers familiar with LCF-style provers will find this representation familiar. We discuss the differences between LCF tactics and Mtac2 tactics in Section 7. A different, but related notion of tactics—providing additional static guarantees—is presented in Section 4.

The execution of tactics statements is performed by the `mrun` command, which is the entry point to the Mtac2 interpreter. In Mtac2's **MProof** environment, every tactic statement `tac` is implicitly converted to (roughly) `mrun (tac ?g)`, where `?g` is the current Coq goal (meta-variable) at that point in the proof. If and when the tactic (applied to the current goal) successfully finishes executing, it produces a list of goals (meta-variables), which the interpreter registers to Coq as the new goals to solve next.

3.2 What Is a Goal?

In Example 3.1, we saw an example goal and its evolution during the execution of an Ltac proof script. In this section, we describe how we represent goals and their evolution in Mtac2.

The main difficulty to overcome is that, as seen in the second tactic statement of Example 3.1, the goals (meta-variables) returned by a tactic generally live in different contexts, each of which contains its own changes—newly-introduced or deleted hypotheses—with respect to the initial context. To be able to return these goals as a list, we require a representation of their meta-variables that is valid in the initial context. The `goal` type presented in Figure 3 achieves exactly that. It does so by wrapping a goal's meta-variable in constructors which represent the changes relative to the initial context, thus making the representation self-contained with respect to that context. In the remainder of this section, we explain each constructor of the `goal` type in detail.

The first constructor, `Metavar ?g`, is the base case of the inductive type, representing a goal that is well-formed in the *current* context.[10] For example, in Example 3.2, the initial `goal` after `MProof` is `@Metavar (forall n, n - 0 = n) ?g`. (The `@` syntax allows us to specify the type of the meta-variable `?g` explicitly, even though it is marked as an implicit argument.)

The second constructor, `@AHyp P (fun H : P => fg)`, represents a single hypothesis `H` of type `P` introduced into the context of an inner goal `fg`, whose type may depend on `H`.[11] For example, after executing `intro n` in Example 3.2, the goal will be:

```
@AHyp nat (fun n : nat => @Metavar (n - 0 = n) ?g1)
```

Finally, the `HypRem` constructor indicates that a hypothesis which has been introduced previously is marked as removed. This mark makes it possible to implement tactics such as `clear`, which perform destructive updates on the proof context. Note that `HypRem` does not distinguish between hypotheses in the initial context and new hypotheses represented by `AHyp`. Consider, for the sake of illustration, that our initial goal is `@Metavar (True -> n = n) ?g`$_r$, and that we introduce and immediately remove the hypothesis `H` with the proof script `intro H &> select True clear`. Here, for illustration purposes and for reasons that will become clear in the coming sections, we use the `select` tactic instead of referring to `H` directly. This tactic, when given a type (`True` in this case), selects an element of that type from the hypotheses (`H`) and feeds it to the given tactic (`clear`). Using `HypRem`, we represent the resulting goal with:

```
@AHyp True (fun H : True => @HypRem True H (@Metavar (n = n) ?g'r))
```

Spawning new goals. We have already seen that Mtac2 tactics transform a given input goal into (potentially) multiple subgoals. To create a new subgoal, tactic developers use the `evar` primitive to generate a fresh meta-variable and wrap this meta-variable in the `Metavar` constructor. The resulting goal is then returned as part of the list of goals produced by the tactic. We will come back to this when we see an example tactic in Section 3.4.

Solving goals. Tactics assume they are given as input an *unsolved goal*, *i.e.*, a goal whose metavariable is as yet uninstantiated. To *solve* this goal, tactics instantiate the meta-variable of the goal by unifying it with a term of the appropriate type. (One direct way to do this is using the Mtac2 tactic `exact E`, which instantiates the given goal with the term `E`. See, for example, Line 8 in Figure 5.)

However, tactics cannot generally keep track of which goals they solve, since the instantiation of goal meta-variables can occur as a side effect of another unification operation (*e.g.*, solving a goal `?g` with a meta-variable `?T` as its type will have the effect of instantiating `?T` as well). Thus, we do not assume that the list of goals *produced* by a tactic is free of solved goals and instead place the burden of filtering out solved goals on (i) tactic composition (Section 3.3), and (ii) the Mtac2 interpreter, which receives the list of goals returned by a tactic statement.

To conclude this section on goals, let us briefly turn towards what happens with the goals produced by a tactic statement. The execution of a tactic statement will return a (posibly empty) list of subgoals. The Mtac2 interpreter takes each of the goals (meta-variables) wrapped in the constructors of `goal` and registers all of the unsolved ones as the next goals to be solved.

[10] Due to the lack of sort polymorphism in Coq, the actual implementation of `Metavar` needs to distinguish between `Prop` and `Type` goals.

[11] An additional constructor—omitted for brevity—handles let-bindings in the goal, as well as local definitions such as those introduced by the `pose` tactic.

```
1  Definition open_and_apply (t : tactic) : tactic :=
2    fix open g :=
3      match g return M _ with
4      | Metavar _ _ => t g
5      | @AHyp C f =>
6        nu (FreshFrom f) None (fun x : C => open (f x) >>= close_goals x)
7      | HypRem x f => remove x (open f) >>= rem_hyp x
8      end.
```

Fig. 4. Implementation of open_and_apply, a function used in tactic composition.

3.3 What Is Tactic Composition?

We now turn our attention towards tactic composition. In Mtac2, we compose tactics in two ways:

(1) We can compose a tactic with another tactic, and obtain another tactic. For instance, consider the composition `case n &> reflexivity`. Here, `reflexivity` is applied to every subgoal introduced by the case analysis.
(2) We can compose a tactic with a list of tactics, and obtain another tactic. For instance, in Example 3.2, we compose `case n` with the list[12] of tactics `[m: idtac | intro n']`. Here, each of the tactics in the list is applied to the corresponding subgoal introduced by the case analysis.

To allow the same syntactic operator to be used for both cases of composition, Mtac2 uses type classes [Sozeau and Oury 2008] to overload the operator. The extensibility of type classes allows for other notions of composition involving tactics. For instance, tactics can also be composed with *goal selectors*: tactic combinators that select a—potentially reordered—subset of open goals to be given to the following tactic.

Tactic composition in Mtac2 plays an important role in enforcing a crucial invariant: Tactics will only ever be called with a `Metavar`, not with any other constructor of `goal`. This enables tactic developers to write their code under the assumption that the implicit proof context matches the context of the current goal. Only tactics that manipulate the context, by introducing or clearing hypotheses, will need to introduce the other constructors of `goal`.

Composing two tactics (the first case above) is performed by the `bind` operator for tactics, whose code is given below. Running `bind t u` will first execute the tactic `t`. The resulting goals are filtered by `filter_goals`, removing solved goals. Then, to uphold the invariant mentioned above, `bind` calls `open_and_apply u g'` on any goal `g'` returned by `t`, which in turn executes `u` on an *opened* version of `g'` (see next paragraph). Finally, we concatenate the resulting lists of goals and return them.

```
Definition bind (t u: tactic) : tactic := fun g =>
  gs <- t g >>= filter_goals;
  r <- M.map (fun g' => open_and_apply u g') gs;
  ret (concat r).
```

The code for `open_and_apply` is given in Figure 4. `open_and_apply` ensures that the tactic `t` given to it is called with the `Metavar` constructor. We call this *opening* the goal. `open_and_apply` opens the goal `g` by recursively traversing `g`, and, at every step, updating the proof context in accordance with the outermost constructor of `g`. For `AHyp`, Mtac's `nu` primitive is used to extend the context with an additional hypothesis (Line 6), whose name is chosen to match the binder of the inner goal `f` using the `FreshFrom` constructor of the `name` type. In the case of `HypRem`, we use `remove` to eradicate

[12]The `[m: ..]` syntax for lists stands for Mtac2's universe-polymorphic list type. Section 6.3 contains details on why we could not use Coq's own list type.

```
1  Definition cintro {A} (var : name) (t : A -> tactic) : tactic := fun g =>
2    mmatch g with
3    | [? (P: A->Type) e] @Metavar (forall x:A, P x) e =u>
4      nu var None (fun x =>
5        let Px := reduce (RedWhd [rl:RedBeta]) (P x) in
6        e' <- evar Px;
7        nG <- abs_fun x e';
8        _ <- exact nG g;
9        t x (Metavar e') >>= close_goals x)
10   | Metavar _ => raise NotAProduct
11   | _ => failwith "Not a [Metavar]"
12   end.
```

Fig. 5. The `cintro` tactic.

the hypothesis from the context (Line 7). Here, `remove` implements weakening: the continuation is executed in a context without the variable `x`. Both of these primitives perform well-bracketed changes to the proof state: They are given a function to be executed in the modified context, and the modification of the proof context is undone after the function is finished executing.

When `open_and_apply` reaches the base case, `Metavar`, it executes the tactic `t`. That execution happens in the modified proof context that matches that of the given goal `g`.

After `t` is done executing, we *close* all the goals resulting from the execution of `t` with respect to the initial goal `g`. Closing a goal `g'` with respect to `g` means replicating every occurrence of `AHyp` and `HypRem` from the initial `g` in `g'`. This is done by calls to `close_goals` (Line 6) and `rem_hyp` (Line 7), respectively. The code for these functions is given below. Both of them simply wrap the respective constructor around each goal in the list of goals they are given.

```
1  Definition close_goals {B} (y : B) : list goal -> M (list goal) :=
2    M.map (fun g' => r <- abs_fun y g'; ret (@AHyp B r)).
3  Definition rem_hyp {B} (x : B) : list goal -> M (list goal) :=
4    M.map (fun g' => ret (HypRem x g')).
```

3.4 Example Tactic: `cintro`

To illustrate how tactics make use of the `goal` type, and how the invariant enforced by tactic composition is relied upon, we now take a detailed look at the implementation of an example tactic: Mtac2's *scoped* introduction tactic `cintro`.

The `cintro` tactic shown in Figure 5 is an excerpt of the basic tactic for introduction of a hypothesis in Mtac2. We used it already in Example 3.2, although in its sugared form: `intro n` is notation for `cintro (TheName "n") (fun x => idtac)`.

This tactic takes the name `var` of the variable being introduced, and a continuation `t`. It proceeds by matching the goal against a `Metavar` constructor whose type is a dependent product indexed by type `A`. The body of the product is captured with the type predicate `P`. If the unification succeeds, we introduce the variable into the context with the `nu` operator. With the new variable `x` in context, we create a new meta-variable `e'` with type `P x` after β-reducing it—we do not want our goals β-expanded. (The reduction mechanism of Mtac2 is explained in Section 6.2.) Then, variable `x` is abstracted from `e'`, effectively creating a function whose body is `e'`. We solve the current goal `g` with this term using the `exact` tactic. Having solved the current goal, we proceed to call the tactic `t` with `x` and the new goal `Metavar e'`. Before returning, we close the goals generated by `t` w.r.t. `x`, using the `close_goals` function presented above.

```
1  Definition solve_tauto_mtac2 : tactic :=
2    mfix0 solve_tauto :=
3      assumption || exact I || (split &> solve_tauto)
4      || (left &> solve_tauto) || (right &> solve_tauto)
5      || (introsn_cont solve_tauto 1)
6      || (eexists |1> solve_tauto) || raise TautoFail.
```

Fig. 6. A simple tautology solver in Mtac2 using backward reasoning.

Note how in `cintro` we rely on the invariant upheld by tactic composition and assume that the goal `g` is a `Metavar`. In any other case, the execution fails (Line 11). If, instead, we are given a `Metavar` but that goal is not a product, we raise the exception `NotAProduct` (Line 10).

Concluding remarks. With the type for tactics presented in this section we are able to build the tautology solver now using backward-reasoning tactics à la Ltac (Figure 6). The code is written in Mtac2 (and it cannot be written in Mtac1), but it follows closely what an Ltac user is familiar with. Without going deeply into the details, let us compare the case for conjunction. In the original Mtac1 code from Figure 2 we obtain a proof for each proposition in the conjunction, and return the `conj` constructor applied to those proofs. Instead, in Figure 6 we call the tactic `split`, which in essence applies the `conj` constructor, generating two subgoals, one for each of the propositions. Then, thanks to the composition operator `&>`, we solve each subgoal by recursively applying the tactic.

At this point, the reader may rightfully wonder what has been gained over just writing this tautology solver in Ltac. Indeed, there is not much difference between the Mtac2 code shown in Figure 6 and what one would naturally code up in Ltac, and consequently writing Mtac2 code in this style suffers similar drawbacks. In the next section, however, we will see how we can use the infrastructure built up thus far in order to develop a type of tactics—and another version of the tautology solver—that fruitfully synthesize the backward-reasoning style of Ltac with the much stronger static guarantees about tactic correctness afforded by Mtac1.

4 TYPED BACKWARD REASONING

The type for tactics provided in the previous section is not sufficiently expressive: a `tactic` does convey no information in its type, either about the goals it solves or the subgoals it creates. And some tactics may indeed not have anything to say about that. But there is no need to throw out the baby with the bath water: in this section, we show how Mtac2 also supports the construction of strongly *typed tactics*.

Typed tactics carry *static* information about the goals they solve and the subgoals they create in their type. Additionally, they are allowed to create *dynamic* subgoals whose type is only known at execution time. Moreover, with the infrastructure we have introduced so far, the implementation of typed tactics is surprisingly straightforward. We focus in this section on the key building blocks of typed tactics, and how to use them to build a strongly-typed version of the backward-reasoning tautology solver. We then present a more "real-world" example of typed tactics in Section 5.

Let us look at the main features of typed tactics in action. Figure 7 contains an implementation of the tautology solver using typed tactics. While the implementation may appear a bit verbose compared to its dynamically-typed counterpart in Figure 6, we will soon see how its strong typing makes it significantly more robust.

```
Definition solve_tauto : forall (P:Prop), ttac P :=
  mfix1 solve_tauto (P : Prop) : M _ :=
  mmatch P in Prop as P' return M (P' * _) with
  | True => apply I
  | [? Q1 Q2] Q1 ∧ Q2 =>
    apply (@conj _ _) <*> solve_tauto Q1 <*> solve_tauto Q2
  | [? Q1 Q2] Q1 ∨ Q2 =>
    mtry
      q1 <- apply (@or_introl _ _) <*> solve_tauto Q1;
      mif has_open_subgoals q1 then raise TautoFail else ret q1
    with TautoFail =>
      mtry
        q2 <- apply (@or_intror _ _) <*> solve_tauto Q2;
        mif has_open_subgoals q2 then raise OpenGoals else ret q2
      with OpenGoals => TT.demote end
    end
  | [? (Q1 Q2 : Prop)] Q1 -> Q2 =>
    tintro (fun x:Q1 => solve_tauto Q2)
  | [? X (Q : X -> Prop)] (exists x : X, Q x) =>
    x <- evar X;
    apply (@ex_intro _ _ _) <*> solve_tauto (Q x)
  | _ => TT.try (TT.use T.assumption)
  end.
```

Fig. 7. A simple tautology solver in Mtac2 using typed tactics.

The type of typed tactics. The most important feature of typed tactics is that their type mentions the goal they are applicable to. This is achieved by parameterizing the type of typed tactics, called `ttac`, by the corresponding type of goal:

```
Definition ttac A := M (A * list goal).
```

We call the parameter `A` the *static goal*, to distinguish it from the `goal` type introduced in the previous section. Regular `goals` can be thought of as *dynamic goals*, as they need to be dynamically inspected to find the type of the contained goal. In Figure 7, `solve_tauto`'s static goal is the universally quantified proposition `P`.

A typed tactic produces two components: (i) a proof of `A`, and (ii) a list of dynamic goals. The first component mirrors the way that Mtac1 meta-programs generate proofs, whereas the second component bridges the gap between these meta-programs and Mtac2 tactics. We explain how dynamic goals are useful in typed tactics further below.

The parameter `A` serves two purposes: (i) documenting what the typed tactic does, and (ii) guiding the implementation of the typed tactic itself. It is the second point that brings us to the next, and arguably most important, feature of typed tactics.

Statically-checked application of lemmas. Even basic tactics such as **`split`**, **`left`**, **`right`**, and `exist` are essentially wrappers around constructors of inductive types or—put differently—lemmas that express the introduction rules for these types. These lemmas have informative types, but wrapping them in dynamically-typed tactics hides these types. This issue becomes even more apparent when we think about the **`exact`** and **`apply`** tactics, whose invocation documents very explicitly the lemma used and, therefore, the type of goals they can be applied to. Typed tactics aim to exploit this statically-available type information to provide a very strong guarantee: If Coq's typechecker

accepts the typed tactic, every application of a lemma contained within matches the type of the respective static goal.

We will now explain how Mtac2 achieves these guarantees for typed tactics using the tautology solver of Figure 7. Note that, in the following, we define several typed tactics that have the same name as existing dynamically-typed tactics. To distinguish them, we prefix dynamically-typed tactics with the `T` namespace and statically-typed tactics with the `TT` namespace.

Let us first consider the case in which a lemma exactly solves the static goal at hand. For this case, Mtac2 offers the `apply` function whose code is given below. `apply` returns the given lemma (as the proof of the static goal) and no additional dynamic goals, as indicated by the empty list `[m:]`.[13]

```
Definition apply {A} (a : A) : ttac A := ret (a, [m:]).
```

An example of `TT.apply` can be seen in Line 4, where the constructor of `True`, called `I`, solves the static goal immediately. It bears repeating that attempting to `TT.apply` a lemma whose type is not `True` would result in a static typechecking error at definition time of the tactic.

We consider the case presented above the base case of applying lemmas and build on that to handle the more general case of a `ttac A` applying a lemma of type `X -> .. -> Y -> A`. To this end, we introduce the *typed specialization operator* `tspec`. The operator is left-associative and is written `<*>`.

Consider the conjunction case of our typed-tactic version of `solve_tauto`. The constructor `conj` expects proofs of `Q1` and `Q2`. In Line 6, we use `tspec` to delegate these two hypotheses to recursive calls to `solve_tauto`, much in the same way that, in a hypothetical Ltac implementation of the solver, one could write `apply (@conj _ _); [solve_tauto|solve_tauto]`.

However, typed tactics give us static guarantees that dynamically-typed Ltac-style backward reasoning cannot provide. The type of the static goal in this branch of the solver is `Q1 ∧ Q2`. This matches the conclusion of `@conj _ _` (where Coq infers the underscores to be `Q1` and `Q2`), whose type is `Q1 -> Q2 -> Q1 ∧ Q2`. The first application of `tspec` with `solve_tauto Q1` refines this to `Q2 -> Q1 ∧ Q2`. The second application of `tspec` provides the one remaining argument of type `Q2`. The remaining type, `Q1 ∧ Q2`, matches the static goal expected in this branch. (Note that we can elide the arguments to recursive calls: they can be inferred.)

If we were to pass a proof of `Q2` instead of `Q1` to the `TT.apply` invocation, the typechecker would reject our definition. If we were to omit either argument, the typechecker would reject our definition. If we were to apply `tspec` a third time, the typechecker would reject our definition. In short, the typed tactic combinators `apply` and `tspec` afford us the same kinds of static guarantees that we expect from function application in strongly-typed functional languages.[14]

The implementation of `tspec` is given below. `tspec` executes its arguments, typed tactics `f` and `x`, from left to right and collects their dynamic subgoals. As the proof term generated by the typed tactic `f` is of type `A -> B`, we can instantiate it with the proof delivered by `x`, yielding a proof of `B`.

```
Definition tspec {A B} (f: ttac (A -> B)) (x: ttac A) : ttac B :=
  ''(b, gb) <- f; ''(a, ga) <- x; ret (b a, gb ++ ga).
```

Dynamic goals in typed tactics. Let us now turn towards the dynamic goals returned by typed tactics. To illustrate the usefulness of this component, we have changed the way `solve_tauto` works. In contrast to both previous versions, the typed-tactic version does not fail when the proposition cannot be (fully) proven. Instead, it operates in a best-effort way, solving as many subgoals that come up during the traversal of the proposition as possible, but potentially leaving some open.

[13] Recall that the `[m: ..]` syntax refers to Mtac2's universe-polymorphic list type (see Section 6.3).
[14] It is worth pointing out that typed tactics form an applicative functor [McBride and Paterson 2008].

Best-effort strategies present a challenge to the typed tactics infrastructure: Whether a subgoal is solved or not depends on dynamic factors such as hypotheses introduced during the execution of the tactic, type class search, and other best-effort tactics being used in the implementation. The second component of `ttac` reconciles the static nature of typed tactics with such uncertainties by allowing typed tactics to optionally spawn dynamic goals.

The simplest such typed tactic is `demote`, which demotes a static goal to a dynamic one. The `demote` tactic is implemented in terms of `to_goal`, a helper function which takes care of creating one new dynamic goal from the current static goal. The code for both functions is given below.

```
Definition to_goal (A : Type) : M (A * goal) := a <- evar A; ret (a, Metavar a).
Definition demote {A : Type} : ttac A := ''(a, g) <- to_goal A; ret (a, [m: g]).
```

To see how the best-effort strategy is reflected in the new implementation of `solve_tauto`, consider the disjunction case. When the tactic encounters a disjunction, it now only commits to either side if that side can be solved without any remaining subgoals. This is necessary because we no longer have a clear indicator of which side to choose due to potential remaining subgoals even in a successful execution of the solver. If neither branch can be solved this way, we demote the current static goal (Line 15) to a dynamic one.

Using an approach similar to that of `demote`, we can implement a typed tactic called **use** which applies a regular, dynamically-typed `tactic` to a static goal:

```
1  Definition use {A} (t : tactic) : ttac A :=
2      ''(a, g) <- to_goal A;
3      gs <- t g;
4      ret (a, gs).
```

An example of **use** can be found in Line 22, where we delegate the static goal of type `P` to the dynamically-typed **assumption** tactic. Note that **use** does not assert that the tactic `t` solves the static goal, *i.e.,* that it solves the static goal without spawning new dynamic goals. A variant of **use**, called **by′**, enforces that assertion.[15] It makes use of `filter_goals`, mentioned already in Section 3.3, for filtering out solved goals before asserting that the list of goals is empty. We present an example of **by′** in Section 5. Its implementation is given below.

```
1  Definition by' {A} (t : tactic) :=
2    ''(m: a, g) <- to_goal A;
3    gs <- (t g >>= T.filter_goals);
4    match gs with
5    | [m:] => ret (m: a, [m:])
6    | _ => failwith "Goal not solved"
7    end.
```

Another very useful combinator for typed tactics is `TT.try`, which demotes the static goal in case its argument, another typed tactic, fails. The code[16] of `TT.try` is given below. If demoting the goal is not desired, tactic developers can fall back to using **mtry** directly. Line 22 contains an example application of `TT.try`, which allows the goal `P` to remain unsolved when no matching assumption can be found.

```
Definition try {A} (t : ttac A) : ttac A :=
  mtry t with _ => demote : M _ end.
```

From dynamic to static goals. Tactics such as **use** and **by′** form one side of static-dynamic interoperability and *demoting* goals is the key to achieving that. The other side, unsurprisingly, involves

[15] It is called **by′** because **by** is a keyword in the Coq grammar and cannot be used as the name of a definition.
[16] The type annotation (`.. : M _`) is only included to help Coq's typechecker and can be safely ignored by the reader.

promoting dynamic goals to static ones—an operation that is fundamentally about reflecting dynamically acquired knowledge (*e.g.*, by matching on the goal) into static knowledge. Accordingly, the key to promoting dynamic goals to static goals is the `match_goal` construct—similar to Ltac's `lazymatch goal` but statically typechecked—that reflects the knowledge about the goal type into the type of every branch.

The code of `solve_tauto` above does not make use of the new `match_goal` combinator, as it does not live at the boundary between typed and regular tactics. If, however, we wanted to lift `solve_tauto` into a regular tactic, we could easily do so by using `match_goal`. The code for this is given below. It matches the goal against any proposition `P` and then calls `solve_tauto P`.

```
Definition solve_tauto_regular : tactic :=
  match_goal with
  | [[?(P : Prop) |- P] ] => solve_tauto P
  end.
```

Note that the branch of `match_goal` is statically typed: its type is `ttac P`. Calling a typed tactic that does not match that type will result in a typechecking error at definition time of the wrapper.

Type class search in typed tactics. In addition to all the typed tactics shown above, our case study makes use of another typed tactic, `TT.apply_`, which triggers type class search on the current goal.[17] We saw an example of a type class, `PureExec`, in Section 1. A type class instance of `PureExec` φ `e1 e2` signifies that the expression `e1` reduces to `e2` given that sufficient precondition φ holds. In Section 5.3, we will use `TT.apply_` to search for instances of `PureExec` for a given candidate redex `e1`.

5 CASE STUDY: IRIS PROOF MODE

In this section we return to the Iris Proof Mode (IPM) case study that we introduced in Section 1.1. Before showing how we can use Mtac2 to build more robust tactics that address the challenges we posed, we give a brief introduction to IPM.

The core feature of IPM is that it enables one to carry out separation logic proofs with the look and feel of proofs in Coq. Let us take a look at a basic proof of a simple lemma in separation logic:

```
Lemma sep_exist A (P R : uPred M) (Ψ : A → uPred M) :
  P ∗ (∃ a, Ψ a) ∗ R -∗ ∃ a, Ψ a ∗ P.
Proof.
  iIntros "[HP [HPsi HR]]". iDestruct "HPsi" as (x) "HPsi".
  iExists x. iSplitL "HPsi"; iAssumption.
Qed.
```

This lemma looks like an ordinary lemma in Coq; the crucial difference is that we are using the connectives of the separation logic (separating conjunction ∗ and magic wand -∗). After calling the first tactic, we end up with the following goal:

```
A : Type
P, R : uPred M
Ψ : A → uPred M
_____________________________________(1/1)
"HP" : P
"HPsi" : ∃ a, Ψ a
"HR" : R
-------------------------------------∗
Ψ x ∗ P
```

[17] The name `apply_` is a reference to Ltac's way of triggering type class search: `apply _`.

As we see here, an additional context appeared that contains the hypotheses in separation logic. The tactics of IPM behave much like the ordinary Coq tactics, but operate on the separation logic context. For example, after calling `iDestruct "HPsi" as (x) "HPsi"`, the hypothesis `"HPsi"` will be turned into `Ψ x` and `x` will be added to the Coq context.

Now, let us take a look at what is going on. IPM uses a deeply embedded representation of the contexts of separation logic `envs`, and the above goal is just a pretty printed version of the entailment relation `envs_entails`:

```
1  Record envs (M : ucmraT) :=
2    Envs { env_persistent : env (uPred M); env_spatial : env (uPred M) }.
3  Definition envs_entails : forall {M}, envs M -> uPred M -> Prop := ...
```

The type `env A` represents association lists with values of type `A` and strings as keys. The type `ucmraT` represents user-chosen ghost state—a parameter necessary to instantiate Iris—and the type `uPred M` is the type of Iris propositions, which depends on that ghost state [Jung et al. 2017].

Although the example above displayed just the *spatial context* `env_spatial`, there is in fact another context `env_persistent` for the so called *persistent* hypotheses. For those readers not familiar with Iris, this context can be safely ignored for the purpose of this paper.

5.1 Symbolic Execution in Ltac

Apart from providing tactics for manipulating the logical connectives of separation logic, IPM offers a set of tactics for symbolically executing program steps. These tactics provide a great deal of convenience to IPM users by taking care of applying the appropriate proof rule, automatically selecting required hypotheses, and introducing new ones, as well as simplifying and reshaping the remaining goals. In the introduction we already saw `wp_pure`—a symbolic execution tactic for pure reduction steps. IPM also supplies tactics for heap operations: `wp_load`, `wp_store`, `wp_cas`, etc.

IPM tactics generally rely on a set of helper tactics and type classes and instances thereof. In the following, we first give a detailed account of the various bits and pieces that go into the `wp_pure` tactic for symbolic execution and then explain how to port it to Mtac2.

Figure 8 (from Line 7 onward) contains the Ltac code of the `wp_pure` tactic. `wp_pure` starts by examining the goal: Symbolic execution only applies to weakest precondition IPM goals. If the tactic finds such a goal, it computes a simplified version of the expression therein and hands that to another helper tactic called `reshape_expr`. The abridged code of `reshape_expr` is given in Figure 9. Its job is to 1) iteratively compute all possible decompositions of the given expression into evaluation contexts and redex, and 2) call the continuation given to it with each decomposition until the continuation succeeds. We will now explain how this decomposition into evaluation contexts works.

Evaluation contexts. The language used in the default instantiation of Iris, `heap_lang`, is a lambda calculus with sums, products, and references. Its operational semantics is formalized using evaluation contexts [Felleisen and Hieb 1992]—expressions with a hole for the expression in evaluation position. The mechanization of `heap_lang` deviates from the standard presentation of evaluation contexts by representing them as lists of evaluation context *items*. The list structure reflects the recursive nature of evaluation contexts: They are linear trees. The development includes a plugging function, called `fill`, which inserts an expression into the hole of an evaluation context to form a closed expression. Due to the use of evaluation context *items*, `fill` is defined as a fold over a plugging function on those items. That function is called `fill_item`. The (abridged) definition of evaluation contexts and the code of `fill_item` are given in Figure 10.

The purpose of using `reshape_expr` becomes clear once we turn our attention to the continuation provided by `wp_pure`. The call to `unify` in Line 12 accepts only those redexes `e'` that match the pattern

```
Lemma tac_wp_pure `{heapG Σ} Δ s E e1 e2 φ Φ :
  PureExec φ e1 e2 ->
  φ ->
  envs_entails Δ (wp s E e2 Φ) ->
  envs_entails Δ (wp s E e1 Φ).

Tactic Notation "wp_pure" open_constr(epat) :=
  lazymatch goal with
  | |- envs_entails _ (wp ?s ?E ?e ?Φ) =>
    let e := eval simpl in e in
    reshape_expr e ltac:(fun K e' =>
      unify e' epat;
      eapply (tac_wp_pure _ _ _ (fill K e'));
      [apply _                          (* PureExec *)
      |try fast_done                    (* The pure condition for PureExec *)
      |wp_expr_simpl_subst; try wp_value_head (* new goal *)
      ])
   || fail "wp_pure: cannot find" epat "in" e "or" epat "is not a reduct"
  | _ => fail "wp_pure: not a 'wp'"
  end.
```

Fig. 8. The type of Lemma tac_wp_pure; Ltac code for wp_pure. (Unchanged from Figure 1.)

```
Ltac reshape_expr e tac := let rec go K e :=
  match e with
  | _ => tac K e
  | App ?e1 ?e2 => reshape_val e1 ltac:(fun v1 => go (AppRCtx v1 :: K) e2)
  | App ?e1 ?e2 => go (AppLCtx e2 :: K) e1
  | ...
  end in go (@nil ectx_item) e.
```

Fig. 9. The reshape_expr Ltac tactic, showing only the representative application case.

```
Inductive ectx_item :=
  | AppLCtx (e2 : expr)
  | AppRCtx (v1 : val)
  | ...
```

```
Definition fill_item (Ki : ectx_item) (e : expr) :=
  match Ki with
  | AppLCtx e2 => App e e2
  | AppRCtx v1 => App (of_val v1) e
  | ...
  end.
```

Fig. 10. Evaluation contexts and the fill_item function (reduced to the representative application case).

epat given to wp_pure by the user. If the match does not succeed, Ltac's backtracking mechanism ensures that reshape_expr calls the continuation with the next decomposition. If the redex e' indeed matches epat, wp_pure applies tac_wp_pure, whose type is shown in Figure 8. The lemma serves as glue between wp_pure and a type class—PureExec—used to express sufficient conditions for pure reduction steps. It witnesses the fact that, given such a sufficient condition φ to reduce e1 to e2, it suffices to prove φ and continue the proof with e2.

The remaining code of wp_pure deals with the subgoals spawned by applying tac_wp_pure. A call to type class search (apply _) solves the first subgoal: an instance of PureExec, establishing the sufficient condition φ for the pure reduction step. That condition, in turn, is taken care of on

a best-effort basis by an IPM variant of Coq's `done` tactic, `fast_done`. Then, `wp_pure` attempts to call its last helper tactic, `wp_value_head`, whose job it is to check if the current expression is fully reduced to a value. If that is the case, it suffices to show the postcondition Φ for that value and the goal is transformed accordingly using beta reduction—`lazy beta`—to substitute the value in Φ. The code of both helper tactics and the lemmas supporting them is given below.

```
Lemma tac_wp_value `{heapG Σ} Δ s E Φ e v :
  IntoVal e v ->
  envs_entails Δ (Φ v) -> envs_entails Δ (WP e @ s; E {{ Φ }}).
Ltac wp_value_head := eapply tac_wp_value; [apply _|lazy beta].
```

```
Lemma tac_wp_expr_eval `{heapG Σ} Δ s E Φ e e' :
  e = e' ->
  envs_entails Σ (WP e' @ s; E {{ Φ }}) -> envs_entails Δ (WP e @ s; E {{ Φ }}).
Ltac wp_expr_simpl_subst t :=
  try (eapply tac_wp_expr_eval; [simpl_subst; reflexivity|]).
```

5.2 Problems with the Ltac Implementation

The Ltac implementation of `wp_pure` suffers from several problems:

Goal-Lemma Mismatch. While `wp_pure` and its helper tactics do make use of Ltac's `lazymatch` to inspect the shape of the goal, Ltac offers no way of statically exploiting this knowledge to ensure that the conclusion of lemmas applied with `eapply` indeed match the shape of the goal. A mistake here will lead to a failure at execution time of the tactic.

Lemma-Arguments Mismatch. Ltac offers no static typechecking. Thus, even if we use the right lemmas, we cannot statically ensure that we invoke them with arguments of the right type, or even with the right number of arguments. A mistake here will manifest itself only during the execution of the tactic.

Subgoal-Tactic Mismatch. Similarly, there is no guarantee that follow-up tactics—even highly specific ones such as `wp_value_head`—can be applied to the subgoals they were assigned to. Again, developers (and their users) will only find out about such mistakes when executing the tactic.

Non-Modular Error Handling. Due to the lack of modular error handling in Ltac, any such runtime failure will cause `reshape_expr` to backtrack and try different decompositions of the expression—a behavior meant to be invoked only when the decomposition does not yield an expression in evaluation position for which a `PureExec` instance exists.

We address all of these issues in the Mtac2 port of `wp_pure` described below.

5.3 Symbolic Execution in Mtac2

The code of the Mtac2 version of `wp_pure`—now a typed tactic—is given in Figure 11. The code follows the structure of the Ltac version. The remainder of this section addresses the differences between both versions and explains how the Mtac2 version resolves the issues we identified with the Ltac version. In particular, in Section 5.3.1, we show `reshape_expr` in Mtac2 (Line 4), and explain the new backtracking protocol driven by the `TryNextDecomposition` exception (Lines 6, 9). In Section 5.3.2, we show the Mtac2 implementation of `wp_expr_simpl_subst` and `wp_value_head`. Finally, in Section 5.3.3, we briefly describe how `wp_pure` can be converted into a regular, dynamically-typed `tactic`.

5.3.1 reshape_expr. The `reshape_expr` tactic is the backbone of `wp_pure`, performing the heavy lifting of decomposing expressions and also backtracking if the current decomposition is not suitable. In Mtac2, `reshape_expr` takes on an additional role on the level of types: refining the static goal of its continuation to depend not on the original expression `e` but instead on the decomposition `fill K e'`. This is the key ingredient to `wp_pure`'s well-typedness, which crucially relies on the

```
1  Definition wp_pure `{heapG Σ} {Δ s e E Φ} (epat : expr):
2    ttac (envs_entails Δ (WP e @ s ; E {{ Φ }})) :=
3    let e := rsimpl e in
4    mtry reshape_expr_wp e (fun K e' =>
5      mtry unify_or_fail UniEvarconv e' epat
6      with NotUnifiable e' epat => raise TryNextDecomposition end;;
7      `e2 φ <- evar _;
8      TT.apply (tac_wp_pure _ _ _ (fill K e') e2 φ _)
9        <**> (mtry TT.apply_ with _ => raise TryNextDecomposition end)
10                                        (* PureExec *)
11       <**> TT.try (T.try fast_done)     (* The pure condition for PureExec *)
12       <**> `e' <- evar _;             (* new goal *)
13             wp_expr_simpl_subst e'
14             <**> TT.try wp_value_head)
15   with ReshapeExprExc =>
16     mfail "wp_pure: cannot find " epat " in " e " or " epat " is not a reduct"
17   end.
```

Fig. 11. The Mtac2 version of wp_pure.

```
1  Definition ReshapeExprExc : Exception. constructor. Qed.
2  Definition TryNextDecomposition : Exception. constructor. Qed.
3  Definition reshape_expr (e : expr)
4    {B : expr -> Type} (tac : forall K e', M (B (fill K e'))) : M (B e) :=
5    (mfix2 go (K : list ectx_item) (e' : expr) : M (B (fill K e')) :=
6      mtry tac K e' with TryNextDecomposition =>
7        match e' as e' return  M (B (fill K e')) with
8        | App e1 e2 =>
9          mtry reshape_val e1 (fun v1 => go (AppRCtx v1 :: K) e2)
10         with ReshapeExprExc => go (AppLCtx e2 :: K) e1 end
11       | ...
12       | Var _ | Lit _ | Fork _ | Rec _ _ _ => raise ReshapeExprExc
13       end
14     end) [] e.
15 Definition reshape_expr_wp `{irisG heap_lang Σ} {s E Φ} (e : expr)
16     (tac : forall K e', ttac (envs_entails Δ (WP fill K e' @ s ; E {{ Φ }}))) :
17   ttac (envs_entails Δ (WP e @ s ; E {{ Φ }})) :=
18   reshape_expr e (B:=fun e => envs_entails _ (WP e @ _ ; _ {{ _ }}) *m _) tac.
```

Fig. 12. The Mtac2 version of reshape_expr, showing the same cases displayed in the Ltac version in Figure 9 and, additionally, an explicit branch for all cases *intended* to fail.

application of tac_wp_pure (Line 8 of Figure 11) to `fill K e'`. Thus, reshape_expr plays an important role in solving the Goal-Lemma Mismatch issue.

The Mtac2 code of reshape_expr is shown in Figure 12. The entire tactic is statically typechecked. Note, though, that we do not restrict the tactic to work on IPM goals (weakest preconditions), which reduces visual clutter quite significantly. In Line 15, we provide an instantiation of reshape_expr, called reshape_expr_wp, which has the more specific type relevant to the application of this tactic in wp_pure. In addition to its well-typedness, reshape_expr is also guaranteed to exhaustively cover

all constructors of the `expr` type. This is enforced by our use of Coq's **`match`** construct. Indeed, Coq forces us to add a branch for all expressions we do not intend to decompose (Line 12).

The type of `reshape_expr` witnesses the tactic's correctness. Its type promises to prove any property `B` of expression `e` while the continuation `tac` only provides proofs of `B` for expressions of the form `fill K e'`. Thus, to call `tac` in Line 6 with a decomposition `K` and `e'`, we need to ensure that `fill K e'` is equal to `e`. This condition is initially true because `fill [] e = e`. At every recursive call, we have to pass a decomposition of `fill K e'` into a new evaluation context `K'` and expression `e''` such that `fill K' e''` is still equal to the previous decomposition `fill K e'`. If we fail to do so, the typechecker will not accept our definition. Thus, we ensure that the continuation `tac` is always called with a valid decomposition of `e`.

Let us take a closer look at the case for application of `e1` to `e2` in Line 8. First, in Line 9, we attempt to reshape `e1` into a value. This is done by calling another helper tactic, `reshape_val`. `reshape_val`, similarly to `reshape_expr`, certifies in its type that the value `v1` given to its continuation is equal to `e1` modulo conversion to an expression. If `reshape_val` succeeds, we push the evaluation context `(e1 ○)`—represented by `AppRCtx v1`—onto the list of evaluation contexts and recurse with `e2` as the decomposition candidate. If `reshape_val` fails to find a value equivalent to `e1` (which it signals by raising `ReshapeExprExc`), we instead proceed by extending the list of evaluation contexts by `(○ e2)`, *i.e.,* `AppLCtx e2`, and recurse with `e1` as the decomposition candidate.

In addition to the benefits mentioned above, the Mtac2 version of `reshape_expr` puts its clients in full control of backtracking. To this end, we introduce a new exception, `TryNextDecomposition`. Only if the client raises this exception does `reshape_expr` backtrack. The other exception, `ReshapeExprExc`, is used solely to signal actual failure. This strict separation solves the issue of Non-Modular Error Handling from the Ltac version.

5.3.2 wp_simpl_subst and wp_value_head. Since the tactic `wp_expr_simpl_subst` applies the lemma `tac_wp_expr_eval` directly, it lends itself very well to conversion into a typed tactic. It is worth noting that `wp_expr_simpl_subst` represents a *partial application* of `tac_wp_expr_eval`: It only attempts to prove the first hypothesis of the lemma. The remaining hypothesis is a static subgoal which we can easily represent in the type of the tactic.

```
1  Definition wp_expr_eval `{hG : heapG Σ} {Δ s E Φ e} e' :
2    ttac (envs_entails Δ (WP e' @ s ; E {{ Φ }}) ->
3          envs_entails Δ (WP e  @ s ; E {{ Φ }})) :=
4    TT.apply (tac_wp_expr_eval Δ s E Φ e e')
5    <*> (TT.use (simpl_subst;; T.reflexivity)).
```

Similarly, `wp_value_head` is a very thin layer around `tac_wp_value_head` and, thus, very easily converted to a typed tactic. As `wp_value_head` is the very last tactic applied by `wp_pure` (and similar tactics), we use `TT.try` to return any unsolved subgoal as a dynamic goal after simplifying any beta redexes.

```
1  Definition wp_value_head `{heapG Σ} {Δ} {s} {E} {e} {Φ} :
2             ttac (envs_entails Δ (WP e @ s; E {{ v, Φ v }})) :=
3    e' <- evar _;
4    TT.apply (tac_wp_value Δ s E Φ e e')
5      <*> TT.apply_ <*> TT.use (T.treduce (RedStrong [rl: RedBeta])).
```

Both tactics have very informative static types which we can exploit to statically ensure the absence of any Subgoal-Tactic Mismatch.

Note that the `simpl_subst` tactic used by `wp_expr_simpl_subst` has also been ported to Mtac2. We ask the interested reader to peruse the Coq development for further details.

5.3.3 A dynamically-typed `tactic` *for* `wp_pure`. The Mtac2 version of `wp_pure` presented in Figure 11 is a typed tactic whose type documents exactly which goals the tactic can be applied to. Users of IPM, however, may not need the static types of `wp_pure` and may also want to easily call the tactic *without* having to specify all its arguments. To this end, we define a `tactic` wrapper around `wp_pure`, which uses the **`match_goal`** construct to convert the dynamic goal to the static goal required by `wp_pure`. The code for this wrapper is given below:

```
1  Definition wp_pure_wrap (epat : expr) : tactic :=
2    match_goal with
3    | [[? Σ (hG : heapG Σ) Δ s E e Φ |- envs_entails Δ (WP e @ s ; E {{ Φ }})] ] =>
4      wp_pure epat
5    | [[?G |- G ] ] => mfail "Goal does not match an IPM weakest precondition."
6    end.
```

5.4 Benefits of the Mtac2 Implementation

In the previous sections, we identified various components in the Mtac2 version of `wp_pure` that contributed to resolving the issues we identified with the Ltac version in Section 5.2. We summarize these points here:

Goal-Lemma Mismatch The Mtac2 version of `wp_pure` takes full advantage of the strong types given to its helper tactics. The dependently-typed tactic `reshape_expr` refines the static goal of the innermost continuation to

```
ttac (@envs_entails (iResUR Σ) Δ (@wp iG s E (fill K e′) Φ))
```

Coq is then able to statically match the type of the lemma being applied in `wp_pure`, `tac_wp_pure`, to this static goal—ensuring that any mismatch between these two types is reported at *definition* time of the tactic.

Lemma-Arguments Mismatch The Mtac2 version of `wp_pure` makes uses of the statically-typed `TT.apply` tactic (Line 8). To be able to give a static type to the application of `tac_wp_pure`, we first introduce meta-variables to fully instantiate the lemma (Line 7). Once introduced, these variables suffice to statically type the application of the lemma, and, thus, to ensure the absence of any mismatches between the lemma's type and its arguments at *definition* time.

Subgoal-Tactic Mismatch The types given to `wp_expr_simpl_subst` and `wp_value_head` are informative enough to ensure that they can indeed be applied to the subgoal corresponding to `tac_wp_pure`'s last hypothesis. The tactics used for the two preceding subgoals—type class search via `TT.apply_` and the best-effort `fast_done` tactic—do not have interesting types: They claim to prove, or at least attempt to prove, any goal given to them.

Non-Modular Error Handling To make use of the new backtracking mechanism in the Mtac2 version of `reshape_expr`, `wp_pure` throws the designated backtracking exception `TryNextDecomposition` if 1) the redex `e′` does not match the pattern (`epat`) specified by the user, or 2) no sufficient condition was found for the reduction of `e′`. This fixes the erratic backtracking behavior that the original Ltac version may exhibit.

Before concluding this section we would like to point out that the extra work that types impose is quickly amortized, since most of the symbolic execution tactics of Iris follow the same pattern as `wp_pure` and can therefore reuse the abstractions defined here.

6 TECHNICAL DETAILS CONCERNING SOME NEW FEATURES IN MTAC2

This section presents a subset of the most important changes introduced in Mtac2.

```
Inductive RedFlags : Set :=
| RedBeta | RedDelta | RedMatch
| RedFix | RedZeta
| RedDeltaC | RedDeltaX
| RedDeltaOnly: list dyn -> RedFlags
| RedDeltaBut: list dyn -> RedFlags.
```

```
Inductive Reduction : Set :=
| RedNone | RedSimpl
| RedOneStep: list RedFlags -> Reduction
| RedWhd: list RedFlags -> Reduction
| RedStrong: list RedFlags -> Reduction
| RedVmCompute.
```

Fig. 13. Reduction API.

6.1 Backtracking Semantics

When it comes to the semantics of the interpreter, the most notable change is the treatment of the meta-context in the presence of exceptions. The meta-context is the context where the information about meta-variables lies. In Mtac1 this context was monotonically growing, since meta-variables were never disposed of. As a consequence of this design decision, we had to carefully garbage-collect unused meta-variables, since Coq treats each unsolved meta-variable as an unsolved subgoal (called a "shelved goal"), even when it does not occur in the resulting proof term. Therefore, the semantics of the `mmatch` construct had to be implemented in the interpreter, because it had to delete the meta-variables of those patterns that did not match the scrutinee.

In Mtac2, instead, we decided to have the meta-context backtrack at each `mtry` if an exception is `raise`d. In a big-step semantics, this will read as follows (Σ and Γ are the meta- and the local contexts, respectively):

$$\frac{\Sigma, \Gamma \vdash \texttt{t} \Downarrow (\Sigma', \texttt{ret e})}{\Sigma, \Gamma \vdash \texttt{mtry t f} \Downarrow (\Sigma', \texttt{ret e})}\text{ETryV} \qquad \frac{\Sigma, \Gamma \vdash \texttt{t} \Downarrow (\Sigma', \texttt{raise e}) \quad \text{FMV}(\texttt{e}) \cap (\Sigma' \setminus \Sigma) = \emptyset}{\Sigma, \Gamma \vdash \texttt{mtry t f} \Downarrow (\Sigma, \texttt{f e})}\text{ETryE1}$$

$$\frac{\Sigma, \Gamma \vdash \texttt{t} \Downarrow (\Sigma', \texttt{raise e}) \quad \text{FMV}(\texttt{e}) \cap (\Sigma' \setminus \Sigma) \neq \emptyset}{\Sigma, \Gamma \vdash \texttt{mtry t f} \Downarrow (\Sigma, \texttt{f ExceptionNotGround})}\text{ETryE2}$$

Note in ETryE1 the extra condition when `t` evaluates to `raise e`: the free meta-variables in `e` must not occur in the difference of Σ' and Σ. Otherwise, the term `e` would be ill-typed in the returned context (Σ). When this condition is not met, the exception is replaced with an exception `ExceptionNotGround` (rule ETryE2).

Thanks to the new semantics and a primitive for unification, we were able to encode pattern matching (`mmatch` and `match_goal`) directly in Gallina.

6.2 Term Reduction

Coq is equipped with several reduction strategies: call-by-name (`hnf`), call-by-value (`cbv`), simplification (`simpl`, a heuristic to improve readability of terms with reducible patterns), and call-by-value run in a virtual machine (`vm_compute`). The first two, `hnf` and `cbv`, can be parameterized over the kind of reduction: β-reduction, `match`-reduction, `fix`-reduction, ζ-reduction (reduction of `let-in`s), and δ-reduction (unfolding of defined constants or variables). In the last case, it is possible to specify which identifiers to unfold or which *not* to unfold. In Mtac2 we took the original datatypes of Mtac1 for describing reduction, and extended them to include all of these strategies (Figure 13).

Another difference from Mtac1 is that in Mtac1 the reduction was performed as part of the `ret` construct, which accepted an argument of the `Reduction` type additionally to the element to return. A big problem with this approach was that a reduced term was not considered equal to the original term by the typechecker, leading to unnecessary coercions in the code. For example, in `x <- ret e; e'`, the definitional equality of `x` and `e` would not be known to `e'`. In Mtac2, we solve this problem by playing an interesting trick: we introduce an explicit `reduce` function, whose

definition is the identity, but which signifies to the Mtac interpreter that a reduction should be performed. Formally speaking, here is the definition of `reduce`:

```
Definition reduce (r : Reduction) {A:Type} (x : A) := x.
```

And here is how the interpreter deals with `reduce`:

$$\frac{\texttt{e reduces to e}' \text{ with strategy } \texttt{r} \quad \Sigma, \Gamma \vdash \texttt{t}\{\texttt{e}'/\texttt{x}\} \Downarrow (\Sigma', \texttt{v})}{\Sigma, \Gamma \vdash \texttt{let x := reduce r e in t} \Downarrow (\Sigma', \texttt{v})}$$

6.3 Polymorphic Universes

To allow meta-meta-programming, *i.e.,* Mtac2 programs that construct Mtac2 programs, the `M` in Mtac2 was declared *universe polymorphic* [Sozeau and Tabareau 2014].

While Coq supports universe polymorphism, its standard library still uses an older mechanism called template universe polymorphism for all inductive types defined therein. Template universe polymorphism poses a significant challenge because it effectively punishes generic programming: uses of generic datatypes (such as `list`) that do not instantiate all their arguments result in global universe constraints on all further instances of `list`. As a consequence, this mechanism interacts very badly with the kind of generic programming that happens not just in Mtac2 meta-programs but also, for example, in Coq libraries defining Haskell-style type classes such as `FMap`, and instances thereof for standard datatypes.

As a result, Mtac2 could not be implemented on top of Coq's standard library. While the Coq developers are discussing moving the standard library away from template universe polymorphism[18], Mtac2 now ships with a properly universe-polymorphic copy of a very small subset of the standard library used exclusively for implementing tactics and meta-programs. In fact, when we mentioned product types, *e.g.,* `A * list goal` in Section 4, we introduced a white lie: in the code we actually employ a distinct, bespoke notation for Mtac2's product type.

6.4 Opaque Definitions

In Mtac1 the primitives of the language were defined as constructors of the inductive type `M`. However, this design had two issues:

(1) Universes: When `M` was made universe polymorphic (see Section 6.3), it became parameterized over the union of all universes required by its constructors. As a consequence, even simple primitives such as `ret`—which itself only requires at most two universes—were forced to carry a prohibitively large number of universes.
(2) Fixpoints: As described in the original paper [Ziliani et al. 2013], the fixpoint operators of Mtac1 had a dummy subset predicate in order to circumvent the *positivity restriction* of Coq. This predicate would then be instantiated with the identity function. Though not a problem in practice, it added a seemingly unnecessary indirection.

In Mtac2, we solve both issues by taking inspiration from the `Exception` type (see Section 2) and making the `M` type into a trivial inductive type:

```
Inductive M : Type -> Prop := mkt : forall {A}, M A.
```

Primitives can then be defined as opaque wrapper definitions around the constructor `mkt`. As an example, we give below the definition of Mtac2's unary fixpoint combinator. Note the negative occurrence of `M`, which no longer requires any indirection.

```
1 Definition fix1 : forall {A: Type} (B: A->Type),
2   ((forall x: A, M (B x)) -> (forall x: A, M (B x))) -> forall x: A, M (B x).
3   intros; constructor. Qed.
```

[18] https://github.com/coq/coq/issues/6093, Coq Pull Request #6093: Future of template polymorphism.

7 RELATED AND FUTURE WORK

The original Mtac paper [Ziliani et al. 2015] includes a detailed comparison between the core of Mtac and other languages for tactic metaprogramming, such as Beluga [Pientka 2008] and VeriML [Stampoulis and Shao 2010]. We focus here on tactic languages for backward reasoning and on recent metaprogramming techniques related to type theory.

A function type for backwards tactics. LCF [Gordon et al. 1979] introduced *backward reasoning* based on tactics. A tactic in LCF enjoys the type `goal -> goal list * procedure`, where `procedure` validates the process of reducing the input goal to a list of subgoals [Gordon 2015]. Although similar to the type presented in Section 6, there are two essential differences.

First, Coq follows the *de Bruijn principle* and will eventually typecheck the proof term produced by the tactic. Therefore Coq does not need to know how the proof was generated.

Second, the Mtac monad implicitly carries within it information about partial proofs (*i.e.,* meta-variables). This information is not present in the LCF tactic type shown above and, as noted by Asperti et al. [2009], this omission limits the expressiveness of tactics.

For this reason, most modern proof assistants enrich the type to include a context for meta-variables, as well as other relevant information about the proof state. In Coq, for instance, it used to be the case that tactics explicitly received and returned the context of meta-variables (`metavars`): `goal * metavars -> goal list * metavars`. However, this type was not rich enough to support tactics operating on multiple goals. Therefore, it was recently replaced with a monadic type that, in addition to the context of meta-variables, embeds information about the *focused* goals (a set of goals currently under consideration). This approach was developed by Spiwack [2010] and is beautifully summarized by Pédrot [2016]. It would be interesting to study if this enriched type could be encoded in Mtac2 as is, or if it will require further support from the interpreter.

Tactic languages with a deep embedding of the logic. In recent years there is a trend among interactive theorem provers: provide a high-level metalanguage containing a deep embedding of the logic, with a mechanism to quote/unquote terms to/from the deeply-embedded representation. This enables programming of tactic primitives without having to resort to the language in which the prover is built (*e.g.,* OCaml in the case of Coq).

A major limitation of deep-embedding approaches is that meta-programs and tactics cannot be given meaningful types to describe their behavior in the way that Mtac2 tactics can (thanks to Mtac's shallow embedding of terms). Also, the deep-embedding approaches must pay the cost of quoting/unquoting of terms. On the other hand, the deep-embedding approaches are more flexible in that they permit the construction of arbitrary terms (even ill-typed ones).

Among deep-embedding approaches there are two broad classes, according to whether the metalanguage for tactic programming is the base logic of the prover or a different language. The former class includes Idris [Christiansen and Brady 2016], Agda [Agda Development Team 2018], and Template Coq [Malecha et al. 2018]. The advantages of reusing the base logic of the prover—advantages that Mtac shares—are that 1) the user is not forced to learn a new language, and 2) one has access to the full power of the base logic when constructing meta-programs/tactics. The latter class includes Ltac2 [Pédrot 2018] and Lean [Ebner et al. 2017], which provide bespoke metalanguages for writing tactics. Ltac2 is a new tactic language intended as a replacement for Ltac. It has ML-style syntax, typing, and semantics, and provides a large (and extensible) API, with access to the internals of Coq. Lean has several interesting features, such as SMT integration and a debugging API, and it performs incredibly well according to the benchmarks in Ebner et al. [2017].

Tactic languages with their semantics coded in the prover's logic: Rtac [Malecha and Bengtson 2016] provides reflection-based tactics in pure Gallina (the base language of Coq). As with Mtac2,

an Rtac tactic is simply a Coq term. Yet, contrary to Mtac2, the semantics of Rtac stays inside Coq. (There is no external interpreter.) Therefore, while Mtac2's interpreter must produce a concrete proof term to serve as a trusted proof, a tactic application in Rtac is in itself a proof, and therefore it must not contain any effectful computation in it. The execution of reflective tactics is usually faster than checking a large proof term, although the proof search process of the tactic has to be executed twice, once for constructing the proof, and once after the proof is complete and checked by Coq's kernel.

Both limitations, the lack of effects and the need for replaying the proof, are not necessarily restrictions, as the Cybele framework shows [Claret et al. 2013]. Like Rtac, the Cybele framework allows the construction of reflective tactics for Coq, but it permits certain effects—like non-termination. Under the hood, a reflective tactic expects information to mimic the effect—like the number of steps. This information, called a *witness*, is obtained from an external oracle, which is nothing more than a variation of the tactic itself, which is extracted to OCaml, compiled, and executed. The witness provides enough information to safely replay the reflective tactic inside Coq. The cost of searching a successful path in the proof search process is paid only once, in the execution of the compiled tactic. When replaying the proof inside Coq, the witness includes enough information to just execute the known successful branches.

That said, a fundamental limitation that Rtac and Cybele share is that their tactics are required to be statically proven sound, meaning that their computation must *always* yield a valid proof. This is in contrast to Mtac2, which allows the use of dynamically-checked meta-programming constructs such as its primitives `nu` and `abs_fun`. As a consequence, the construction of tactics in Rtac and Cybele requires a significant amount of boilerplate ensuring their unconditional correctness. To alleviate this burden, Rtac provides soundness proofs for basic tactics, and an Ltac tactic to automatically solve the user's tactic's soundness proof built on top of them. Another limitation of Rtac is that it lacks support for goals with dependent types. As our case study routinely deals with dependently-typed goals, we believe this to be a major restriction.

Shallowly-embedded tactic languages. Recently, Cauderlier [2018] built a tactic language similar to Mtac for Dedukti [Boespflug et al. 2012]. An interesting aspect of this work lies in its use of Dedukti's rewrite rules to build the interpreter, thereby enabling the construction of language primitives directly within the prover. Then, to execute tactics, it uses Meta Dedukti [Cauderlier and Thiré [n. d.]], a tool for Dedukti that normalizes terms according to a rewrite theory (in this case, the interpreter of tactics). Unlike with Rtac and Cybele, there is no need to justify the soundness of the rules inside the language because, as in Mtac, Dedukti typechecks the output of a tactic after the fact.

Cauderlier [2018] also includes a type for tactics. That type, however, does not return a list of goals but a single goal instead. As a consequence, tactics that operate on multiple subgoals need to take a corresponding number of continuations as arguments, instead of simply composing the tactic with a list of tactics as is done in Mtac2. Consider, for example, a proof script that splits a conjunction `P ∧ Q`, applies two tactics `tacP` and `tacQ` to their respective subgoals, and then applies `tacB` to both subgoals. In Mtac2, we can write this as `split &> [m: tacP | tacQ] &> tacB`. In the system presented by [Cauderlier 2018], however, this is written roughly as `split (tacP; tacB) (tacQ; tacB)` (where we use `;` to signify the `bind` operation). Note that the `tacB` tactic has to be repeated in both continuations since there is no way to `merge` the proof script after splitting.

When compared to Mtac, the expressiveness of Cauderlier [2018] is somewhat limited. For a start, Meta Dedukti has no notion of meta-variables, and they cannot support the kind of dynamic checks we use in Mtac, which limits the kind of primitives that can be coded. For instance, in Mtac we provide primitives for introducing terms in the context (the `nu` primitive) and for creating an

abstraction (`abs_fun`). The former dynamically checks that the variable does not escape the context. This check cannot be coded directly in Meta Dedukti and forces a restricted form of operator which combines the effect of `nu` and `abs_fun`, called `mintro`.

Future work: Performance. The most important direction for future work is improving Mtac2's performance. The tactics we ported for the IPM case study are often slower than their Ltac counterparts—sometimes by a factor of 5. This is not entirely surprising, as Mtac2 has not seen nearly as much performance tuning as Ltac. Nonetheless, the current development version of Mtac2 (as of June 2018) is now able to comfortably outperform Ltac in a more specialized benchmark developed in Gross et al. [2018].[19] Generally, Mtac2's static types allow us to perform far less dynamic typechecking than Ltac and we therefore expect to eventually outperform Ltac in most, if not all, situations.

While there is no shortage of (potential) low-hanging fruit to improve the performance of Mtac2's interpreter, we plan to additionally focus on building profiling tools that empower Mtac2 users to understand the performance of their own tactics and meta-programs.

Additionally, we would like to introduce specialized *term decomposition* patterns, *i.e.,* new kinds of `mmatch` patterns that can be used to, for example, decompose an application into its head symbol and arguments. Such specialized patterns offer a great potential for performance improvements: They avoid costly meta-variables that are otherwise needed to perform these decompositions with the existing `mmatch` patterns. Initial experiments with a decomposition pattern for application have been very promising and were instrumental in leading Mtac2 to outperform Ltac in the benchmark mentioned above.

ACKNOWLEDGMENTS

We would like to thank the many contributors to the Mtac2 project, especially: 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 of Mtac2's tactics; Thomas Refis, for his contribution to the discussion about `match_goal`; Pierre-Marie Pédrot, for greatly improving Mtac2's perfomance with his suggestions; Matthieu Sozeau, for his invaluable help with the Coq API; Emilio Jesús Gallego Arias, for helping port Mtac2 to recent versions of Coq; and Jason Gross, for giving us the opportunity to test and improve Mtac2 using his performance case-study. We are also thankful to the anonymous reviewers for their help improving the presentation of the paper.

This research was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 683289).

REFERENCES

The Agda Development Team. 2018. Reflection. Retrieved March 16, 2018 from https://agda.readthedocs.io/en/v2.5.3/language/reflection.html

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2009. A new type for tactics. *PLMMS* (2009), 22.

Mathieu Boespflug, Quentin Carbonneaux, Olivier Hermant, and Ronan Saillard. 2012. Dedukti: A universal proof checker. In *Journées communes LTP - LAC*. https://hal-mines-paristech.archives-ouvertes.fr/hal-01537578

Raphaël Cauderlier. 2018. Tactics and certificates in Meta Dedukti. In *ITP (ITP '18)*.

Raphaël Cauderlier and François Thiré. [n. d.]. Meta Dedukti. http://deducteam.gforge.inria.fr/metadedukti/

David Christiansen and Edwin Brady. 2016. Elaborator reflection: Extending Idris in Idris. In *ICFP*. 284–297. https://doi.org/10.1145/2951913.2951932

[19] The benchmarks presented in Gross et al. [2018] were done using a now-outdated version of Mtac2 which suffered from a particularly unfortunate performance regression leading to exponential slowdown.

Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. 2013. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *ITP (LNCS)*, Vol. 7998. 67–83. https://doi.org/10.1007/978-3-642-39634-2_8

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. *PACMPL* 1, ICFP (2017), 34:1–34:29. https://doi.org/10.1145/3110278

Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. *TCS* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

M. J. C. Gordon. 2015. 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).

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF*. LNCS, Vol. 78. https://doi.org/10.1007/3-540-09724-4

Jason Gross, Andres Erbsen, and Adam Chlipala. 2018. Reification by Parametricity: Fast Setup for Proof by Reflection, in Two Lines of Ltac. In *ITP (ITP '18)*.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL, 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2017. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Submitted for publication* (2017).

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *ECOOP (LIPIcs)*, Vol. 74. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Coq repository for Mtac2. https://github.com/Mtac2/Mtac2/.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. 205–217. https://doi.org/10.1145/3093333.3009855

Gregory Malecha, Abhishek Anand, Simon Boulier, and Matthieu Sozeau. 2018. Reflection library for Coq. Retrieved March 16, 2018 from https://github.com/Template-Coq/template-coq

Gregory Malecha and Jesper Bengtson. 2016. Extensible and efficient automation through reflective tactics. In *ESOP (LNCS)*, Vol. 9632. 532–559. https://doi.org/10.1007/978-3-662-49498-1_21

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *JFP* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual modal type theory. *TOCL* 9, 3 (2008), 23:1–23:49. https://doi.org/10.1145/1352582.1352591

Pierre-Marie Pédrot. 2016. The Coq tactic engine. Retrieved March 16, 2018 from http://coqhott.gforge.inria.fr/blog/coq-tactic-engine/

Pierre-Marie Pédrot. 2018. A standalone implementation of Ltac2 as a Coq plugin. Retrieved March 16, 2018 from https://github.com/ppedrot/ltac2/

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*. 371–382. https://doi.org/10.1145/1328438.1328483

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs (LNCS)*, Vol. 5170. 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

Matthieu Sozeau and Nicolas Tabareau. 2014. Universe Polymorphism in Coq. In *ITP (LNCS)*, Vol. 8558. 499–514. https://doi.org/10.1007/978-3-319-08970-6_32

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

Antonis Stampoulis and Zhong Shao. 2010. VeriML: Typed computation of logical terms inside a language with effects. In *ICFP*. 333–344. https://doi.org/10.1145/1863543.1863591

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *PACMPL* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *PACMPL* 2, POPL, 64:1–64:28. https://doi.org/10.1145/3158152

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: A monad for typed tactic programming in Coq. In *ICFP*. 87–100. https://doi.org/10.1145/2500365.2500579

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2015. Mtac: A monad for typed tactic programming in Coq. *JFP* 25 (2015). https://doi.org/10.1017/S0956796815000118