Mtac: A Monad for Typed Tactic Programming in Coq

Beta Ziliani
MPI-SWS
beta@mpi-sws.org

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Neelakantan R. Krishnaswami
MPI-SWS
neelk@mpi-sws.org

Aleksandar Nanevski
IMDEA Software Institute
aleks.nanevski@imdea.org

Viktor Vafeiadis
MPI-SWS
viktor@mpi-sws.org

Abstract

Effective support for custom proof automation is essential for large-scale interactive proof development. However, existing languages for automation via *tactics* either (a) provide no way to specify the behavior of tactics within the base logic of the accompanying theorem prover, or (b) rely on advanced type-theoretic machinery that is not easily integrated into established theorem provers.

We present Mtac, a lightweight but powerful extension to Coq that supports dependently-typed tactic programming. Mtac tactics have access to all the features of ordinary Coq programming, as well as a new set of typed tactical primitives. We avoid the need to touch the trusted kernel typechecker of Coq by encapsulating uses of these new tactical primitives in a *monad*, and instrumenting Coq so that it executes monadic tactics during type inference.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

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

1. Introduction

The past decade has seen a dramatic rise in both the popularity and sophistication of interactive theorem proving technology. Proof assistants like Coq and Isabelle are now eminently effective for formalizing "research-grade" mathematics [8, 9], verifying serious software systems [15, 16, 32, 6], and, more broadly, enabling researchers to mechanize and breed confidence in their results. Nevertheless, due to the challenging nature of the verification problems to which these tools are applied, as well as the rich higher-order logics they employ, the mechanization of substantial proofs typically requires a significant amount of manual effort.

To alleviate this burden, theorem provers provide facilities for *custom proof automation*, enabling users to instruct the system how to carry out "obvious" or oft-repeated proof steps automatically. In some systems like Coq, the base logic of the theorem prover is powerful enough that one can use "proof by reflection" to implement automation routines within the base logic itself (*e.g.*, [3]). However, this approach is applicable only to pure decision procedures, and requires them to be programmed in a restricted style (so that their totality is self-evident). In general, one may wish to write automation routines that engage in activities that even a rich type system like Coq's does not sanction—routines, for instance, that do not (provably) terminate on all inputs, that inspect the intensional syntactic structure of terms, or that employ computational effects.

Toward this end, theorem provers typically provide an additional language for *tactic* programming. Tactics support general-purpose scripting of automation routines, as well as fine control over the state of an interactive proof. However, for most existing tactic languages (*e.g.*, ML, Ltac), the price to pay for this freedom is that the behavior of a tactic lacks any static specification within the base logic of the theorem prover (such as, in Coq, a type). As a result of being untyped, tactics are known to be difficult to compose, debug, and maintain.

A number of researchers have therefore explored ways of supporting *typed* tactic programming. One approach, exemplified by Delphin [23], Beluga [21], and most recently VeriML [29], is to keep a strict separation between the "computational" tactic language and the base logic of the theorem prover, thus maintaining flexibility in what tactics can do, but in addition employing rich type systems to encode strong static guarantees about tactic behavior. The main downside of these systems is a pragmatic one: they are not programmable or usable interactively, and due to the advanced type-theoretic machinery they rely on—*e.g.*, for Beluga and VeriML, *contextual modal type theory* [20]—it is not clear how to incorporate them into established interactive theorem provers.

A rather different approach, proposed by Gonthier *et al.* [11] specifically in the context of Coq, is to encapsulate automation routines as *overloaded lemmas*. Like an ordinary lemma, an overloaded lemma has a precise formal specification in the form of a (dependent) Coq type. The key difference is that an overloaded lemma—much like an overloaded function in Haskell—is not proven (*i.e.*, implemented) once and for all up front; instead, every time the lemma is applied to a particular goal, the system will run a user-specified automation routine in order to construct a proof on the fly for that particular instance of the lemma. To program the automation routine, one uses Coq's "canonical structure" mechanism to declare a set of proof-building rules—implemented as Coq terms—that will be fired in a predictable order by the Coq unification algorithm (but may or may not succeed). In effect, one encodes one's automation routine as a *dependently-typed logic program* to be executed by Coq's type inference engine.

The major benefit of this approach is its integration into Coq: it enables users to program tactics in Coq directly, rather than in a

ICFP '13, September 25–27, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2326-0/13/09...$15.00.
http://dx.doi.org/10.1145/2500365.2500579

separate language, while at the same time offering significant additional expressive power beyond what is available in the base logic of Coq. The downside, however, is that the logic-programming style of canonical structures is in most cases not as natural a fit for tactics as a *functional-programming* style would be.[1] Moreover, canonical structures provide a relatively low-level language for writing tactics. The control flow of sophisticated canonical structure programs depends closely on how Coq type inference is implemented, and thus writing even simple tactics requires one to think at the level of the Coq unification algorithm, sometimes embracing its limitations and sometimes working around them. To make up for this, Gonthier *et al.* [11] describe a series of "design patterns" for programming canonical structures effectively. While these design patterns are clearly useful, the desire for them nonetheless suggests that there is a *high-level language* waiting to be born.

1.1 Mtac: A Monad for Typed Tactic Programming in Coq

In this paper, we present a new language—**Mtac**—for typed tactic programming in Coq. Like Beluga and VeriML, Mtac supports general-purpose tactic programming in a direct functional style. Unlike those languages, however, Mtac is not a separate language, but rather a simple extension to Coq. As a result, Mtac tactics (or as we call them, **Mtactics**) have access to all the features of ordinary Coq programming *in addition to* a new set of tactical primitives. Furthermore, like overloaded lemmas, their (partial) correctness is specified statically within the Coq type system itself, and they are fully integrated into Coq, so they can be programmed and used interactively. Mtac is thus, to our knowledge, the first language to support interactive, dependently-typed tactic programming.

The key idea behind Mtac is dead simple. We encapsulate tactics in a *monad*, thus avoiding the need to change the base logic and trusted kernel typechecker of Coq *at all*. Then, we modify the Coq infrastructure so that it executes these monadic tactics, when requested to do so, *during* type inference (*i.e.*, during interactive proof development or when executing a proof script).

More concretely, Mtac extends Coq with:

1. An inductive type family $\bigcirc A$ (read as "maybe A") classifying Mtactics that—if they terminate successfully—will produce Coq terms of type A. The constructors of this type family essentially give the syntax for a monadically-typed tactic language: they include the usual monadic return and bind, as well as a suite of combinators for tactic programming with fixed points, exception handling, pattern matching, and more. (Note: the definition of the type family $\bigcirc A$ does not *per se* require any extension to Coq—it is just an ordinary inductive type family.)
2. A primitive *tactic execution* construct, **run** t, which has type A assuming its argument t is a tactic of type $\bigcirc A$. When (our instrumentation of) the Coq type inference engine encounters **run** t, it executes the tactic t. If that execution terminates, it will either produce a term u of type A (in which case Coq will rewrite **run** t to u) or else an uncaught exception (which Coq will report to the user). If a proof passes entirely through type inference without incurring any uncaught exceptions, that means that all instances of **run** in the proof must have been replaced with standard Coq terms. Hence, there is no need to extend the trusted kernel typechecker of Coq to handle **run**.

Example: Searching in a List. To get a quick sense of what Mtac programming is like, consider the example in Figure 1. Here, search is a tactical term of type $\forall x : A.\ \forall s :$ list $A.\ \bigcirc(x \in s)$. When executed, search x s will search for an element x (of type A) in a list s (of type list A), and if it finds x in s, it will return a proof that $x \in s$. Note, however, that search x s itself is just a Coq term of monadic type $\bigcirc(x \in s)$, and that the execution of the tactic will only occur when this term is **run**.

```
01 Definition search (x : A) :=
02   mfix f [s : list A] :=
03     mmatch s as s' return ○(x ∈ s') with
04     | [l r] l ++ r ⇒
05       mtry
06         il ← f l;
07         ret (in_or_app l r x (or_introl il))
08       with _ ⇒
09         ir ← f r;
10         ret (in_or_app l r x (or_intror ir))
11       end
12     | [s'] (x :: s') ⇒ ret (in_eq _ _)
13     | [y s'] (y :: s') ⇒
14       r ← f s';
15       ret (in_cons y r)
16     | _ ⇒ raise NotFound
17     end.
```

Figure 1. Mtactic for searching in a list.

The implementation of search relies on four new features of Mtac that go beyond what is possible in ordinary Coq programming: it iterates using a potentially unbounded fixed point **mfix** (line 2), it case-analyzes the input list s using a new **mmatch** constructor (line 3), it **raise**-s an exception NotFound if the element x was not found (line 16), and this exception is caught and handled (for backtracking purposes) using **mtry** (line 5). These new features, which we will present in detail in §2, are all constructors of the inductive type family $\bigcirc A$. Regarding **mmatch**, the reason it is different from ordinary Coq **match** is that it supports pattern-matching not only against primitive datatype constructors (*e.g.*, nil and ::) but also against arbitrary terms (*e.g.*, applications of the ++ function for concatenating two lists). For example, search starts out (line 4) by checking whether s is an application of ++ to two subterms l and r. If so, it searches for x first in l and then in r. In this way, **mmatch** supports case analysis of the intensional syntactic structure of open terms, in the manner of VeriML's **holcase** [29] and Beluga's **case** [21].

By **run**-ning search, we can now, for example, very easily prove the following lemma establishing that z is in the list $[x; y; z]$:

Lemma z_in_xyz $(x\ y\ z : A) : z \in [x; y; z] :=$ **run** (search _ _)

Note here that we did not even need to supply the inputs to search explicitly: they were picked up from context, namely the goal of the lemma ($z \in [x; y; z]$), which Coq type inference proceeds to unify with the output type of the Mtactic search.

1.2 Contributions and Overview

In the remainder of this paper, we will:

- Describe the design of Mtac in detail (§2).
- Give a number of examples to concretely illustrate the benefits of Mtac programming (§3).
- Present the formalization of Mtac, along with meta-theoretic results such as type safety (§4).
- Explore some technical issues regarding the integration of Mtac into Coq (§5).
- Compare with related work and discuss future work (§6).

The Coq patch and the examples can be downloaded from:

`http://plv.mpi-sws.org/mtac`

[1] In terms of expressivity, there are tradeoffs between the two styles—for further discussion, see §6.

$\bigcirc$	:	Type $\to$ Prop
ret	:	$\forall A.\ A \to \bigcirc A$
bind	:	$\forall A\ B.\ \bigcirc A \to (A \to \bigcirc B) \to \bigcirc B$
raise	:	$\forall A.$ Exception $\to \bigcirc A$
mtry	:	$\forall A.\ \bigcirc A \to$ (Exception $\to \bigcirc A) \to \bigcirc A$
mfix	:	$\forall A\ B.\ ((\forall x : A.\ \bigcirc(B\ x)) \to (\forall x : A.\ \bigcirc(B\ x)))$ $\to \forall x : A.\ \bigcirc(B\ x)$
mmatch	:	$\forall A\ B\ (t : A).$ list (Patt $A\ B) \to \bigcirc(B\ t)$
print	:	$\forall s :$ string. $\bigcirc$unit
nu	:	$\forall A\ B.\ (A \to \bigcirc B) \to \bigcirc B$
abs	:	$\forall A\ P\ x.\ P\ x \to \bigcirc(\forall y : A.\ P\ y)$
is_var	:	$\forall A.\ A \to \bigcirc$bool
evar	:	$\forall A.\ \bigcirc A$
is_evar	:	$\forall A.\ A \to \bigcirc$bool
Patt	:	$\forall A\ (B : A \to$ Type). Type
Pbase	:	$\forall A\ B\ (p : A)\ (b : \bigcirc(B\ p)).$ Patt $A\ B$
Ptele	:	$\forall A\ B\ C.\ (\forall x : C.$ Patt $A\ B) \to$ Patt $A\ B$

Figure 2. The $\bigcirc$ and Patt inductive types.

2. Mtac: A Language for Proof Automation

In this section, we describe the syntax and typing of Mtac, our language for typed proof automation.

Syntax of Mtac. Mtac extends CIC, the Calculus of (co-)Inductive Constructions (see *e.g.*, [2]), with a monadic type constructor $\bigcirc A$, representing tactic computations returning results of type A, along with suitable introduction and elimination forms for such computations. We define $\bigcirc$: Type $\to$ Prop as a normal CIC inductive predicate with constructors reflecting our syntax for tactic programming, which are shown in Fig. 2. (We prefer to define $\bigcirc$ inductively instead of axiomatizing it in order to cheaply ensure that we do not affect the logical consistency of CIC.) The $\bigcirc$ constructors include standard monadic return and bind (**ret**, **bind**), primitives for throwing and handling exceptions (**raise**, **mtry**), a fixed point combinator (**mfix**), a pattern matching construct (**mmatch**), and a printing primitive useful for debugging Mtactics (**print**). Mtac also provides more specialized operations for handling parameters and unification variables (**nu**, **abs**, **is_var**, **evar**, **is_evar**), but we defer explanation of those features until §3.2.

First, let us clear up a somewhat technical point. The reason we define $\bigcirc$ as an inductive *predicate* (*i.e.*, whose return sort is Prop rather than Type) has to do with the handling of **mfix**. Specifically, in order to satisfy Coq's syntactic positivity condition on inductive definitions, we cannot declare **mfix** directly with the type given in Figure 2, since that type mentions the monadic type constructor $\bigcirc$ in a negative position. To work around this, in the inductive definition of $\bigcirc A$, we replace the **mfix** constructor with a variant, **mfix**$'$, in "Mendler style" [17, 14], *i.e.*, in which references to $\bigcirc$ are substituted with references to a *parameter* $\odot$:

$$\textbf{mfix}' : \forall A\ B\ \odot.\ (\forall x : A.\ \odot(B\ x) \to \bigcirc(B\ x)) \to \\ ((\forall x{:}A.\ \odot(B\ x)) \to (\forall x{:}A.\ \odot(B\ x))) \to \forall x{:}A.\ \bigcirc(B\ x)$$

The **mfix** from Figure 2 is then recovered simply by instantiating the $\odot$ parameter of **mfix**$'$ with $\bigcirc$, and instantiating its first value parameter with the identity function. However, due to the inherent "circularity" of this trick, it only works if the type $\bigcirc A$ belongs to an impredicative sort like Prop. (In particular, if $\bigcirc A$ were defined in Type, then while **mfix**$'$ would be well-formed, applying **mfix**$'$ to the identity function in order to get an **mfix** would cause a universe inconsistency.) Fortunately, defining $\bigcirc A$ in Prop has no practical impact on Mtac programming. Note that, in CIC, Prop : Type; so it is possible to construct nested types such as $\bigcirc(\bigcirc A)$.

Now, onto the features of Mtac. The typing of monadic **ret** and **bind** is self-explanatory. The exception constructs **raise** and **mtry** are also straightforward: their types assume the existence of an exception type Exception. It is easy to define such a type, as well as a way of declaring new exceptions of that type, in existing Coq (see §5 for details). The print statement **print** takes the string to print onto the standard output and returns the trivial element.

Pattern matching, **mmatch**, expects a term of type A and a sequence of pattern matching clauses of type Patt $A\ B$, which match objects x of type A and return results of type $B\ x$. Binding in the pattern matching clauses is represented as a *telescope*: Pbase p b describes a ground clause that matches the constant p and has body b, and Ptele($\lambda x.\ pc$) adds the binder x to the pattern matching clause pc. So, for example, Ptele($\lambda x.$ Ptele($\lambda y.$ Pbase $(x + y)\ b$) represents the clause that matches an addition expression, binds the left subexpression to x and the right one to y, and then returns some expression b which can mention both x and y. Another example is the clause, Ptele($\lambda x.$ Pbase $x\ b$) which matches any term and returns b.

Note that it is also fine for a pattern to mention free variables bound in the ambient environment (*i.e.*, not bound by the telescope pattern). Such patterns enable one to check that (some component of) the term being pattern-matched is syntactically unifiable with a specific term of interest. We will see examples of this in the search2 and lookup Mtactics in §3.

In our examples and in our Coq development, we often omit inferrable type annotations and use the following notation to improve readability of Mtactics:

$$\begin{array}{lll}
x \leftarrow t; t' & \text{denotes} & \textbf{bind}\ t\ (\lambda x.\ t') \\
\textbf{mfix}\ f\ [x : A] := t & \text{denotes} & \textbf{mfix}\ (\lambda f.\ \lambda x.\ t) \\
\nu x : A.\ t & \text{denotes} & \textbf{nu}\ (\lambda x : A.\ t) \\
\textbf{mmatch}\ t & & \textbf{mmatch}\ (\lambda x.\ T)\ t \\
\quad \textbf{as}\ x\ \textbf{return}\ \bigcirc T\ \textbf{with} & & [\\
\mid [\overline{x_1}]\ p_1 \Rightarrow b_1 & & \quad \text{Ptele}\ \overline{x_1}\ (\text{Pbase}\ p_1\ b_1), \\
\quad \ldots & \text{denotes} & \quad \ldots \\
\mid [\overline{x_m}]\ p_m \Rightarrow b_m & & \quad \text{Ptele}\ \overline{x_m}\ (\text{Pbase}\ p_m\ b_m) \\
\textbf{end} & &] \\
\textbf{mtry}\ t & \text{denotes} & \textbf{mtry}\ t\ (\lambda e. \\
\textbf{with}\ ps\ \textbf{end} & & \quad \textbf{mmatch}\ e\ \textbf{with}\ ps\ \textbf{end})
\end{array}$$

where Ptele $x_1 \cdots x_n\ p$ means Ptele($\lambda x_1 \ldots$ Ptele($\lambda x_n.\ p) \cdots$). The type annotation **as** x **return** $\bigcirc T$ in the **mmatch** notation is optional and can be omitted, in which case the returning type is left to the type inference algorithm to infer.

Running Mtactics. Defining $\bigcirc$ as an inductive predicate means that terms of type $\bigcirc A$ can be destructed by case analysis and induction. Unlike other inductive types, $\bigcirc$ supports an additional destructor: *tactic execution*. Formally, we extend Coq with a new construct, **run** t, that takes an Mtactic t of type $\bigcirc A$ (for some A), and runs it *at type-inference time* to return a term t' of type A.

$$\frac{\Gamma \vdash t : \bigcirc A \qquad \Gamma \vdash t \rightsquigarrow^* \textbf{ret}\ t'}{\Gamma \vdash \textbf{run}\ t : A}$$

We postpone the definition of the tactic evaluation relation, $\rightsquigarrow$, as well as a precise formulation of the rule, to §4, but note that since tactic evaluation is type-preserving, t' has type A, and thus A is inhabited.

3. Mtac by Example

In this section, we offer a gentle introduction to the various features of Mtac by working through a sequence of proof automation examples.

3.1 noalias: Non-Aliasing of Disjointly Allocated Pointers

Our first example, noalias, is taken from Gonthier *et al.* [11]. The goal is to prove that two pointers are distinct, given the assumption that they appear in the domains of disjoint subheaps of a well-defined memory.

In [11], the noalias example was used to illustrate a rather subtle and sophisticated design pattern for composition of overloaded lemmas. Here, it will help illustrate the main characteristics of Mtac, while at the same time emphasizing the relative simplicity and readability of Mtactics compared to previous approaches.

Preliminaries. We will work here with heaps (of type heap), which are finite maps from pointers (of type ptr) to values. We write $h_1 \bullet h_2$ for the disjoint union of h_1 and h_2, and $x \mapsto v$ for the singleton heap containing only the pointer x, storing the value v. The disjoint union may be undefined if h_1 and h_2 overlap, so we employ a predicate def h, which declares that h is in fact defined.

Motivating Example. With these definitions in hand, let us state a goal we would like to solve automatically:

$$\frac{D\ :\ \mathsf{def}\ (h_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (h_2 \bullet x_3 \mapsto v_3))}{x_1\ \texttt{!=}\ x_2 \wedge x_2\ \texttt{!=}\ x_3}$$

Above the line is a hypothesis concerning the well-definedness of a heap mentioning x_1, x_2, and x_3, and below the line is the goal, which is to show that x_1 is distinct from x_2, and x_2 from x_3.

Intuitively, the truth of the goal follows obviously from the fact that x_1, x_2, and x_3 appear in disjoint subheaps of a well-defined heap. This intuition is made formal with the following lemma (in plain Coq):

$$\begin{aligned}&\mathsf{noalias\_manual} : \forall (h{:}\mathsf{heap})\ (y_1\ y_2{:}\mathsf{ptr})\ (w_1{:}A_1)\ (w_2{:}A_2).\\ &\qquad \mathsf{def}\ (y_1 \mapsto w_1 \bullet y_2 \mapsto w_2 \bullet h) \to y_1\ \texttt{!=}\ y_2\end{aligned}$$

Unfortunately, we cannot apply this lemma using hypothesis D as it stands, since the heap that D proves to be well-defined is not of the form required by the premise of the lemma—that is, with the pointers in question (x_1 and x_2, or x_2 and x_3) at the very *front* of the heap expression. It is of course possible to solve the goal by: (a) repeatedly applying rules of associativity and commutativity for heap expressions in order to rearrange the heap in the type of D so that the relevant pointers are at the front of the heap expression; (b) applying the noalias_manual lemma to solve the first inequality; and then repeating (a) and (b) to solve the second inequality.

But we would like to do better. What we really want is an Mtactic that will solve these kinds of goals automatically, no matter where the pointers we care about are located inside the heap. One option is to write an Mtactic to perform all the rearrangements necessary to put the two pointers at the front, and then apply the lemma above. The main inconvenience with this approach is that each inequality in the goal requires a new rearrangement of the heap, where each pointer has to be found and then "bubbled up" to the front of the heap. Computationally, this takes twice the size of the heap for each pointer.

Instead, we pursue a solution analogous to the one in [11], breaking the problem into two smaller Mtactics scan and search2, combined in a third Mtactic, noalias.

The Mtactic **scan.** Figure 3 presents the Mtactic scan. It scans its input heap h to produce a list of the pointers x appearing in singleton heaps $x \mapsto v$ in h. More specifically, it returns a dependent record containing a list of pointers (seq_of, of type list ptr), together with a proof that, if h is well-defined, then (1) the list seq_of is "unique" (denoted uniq seq_of), meaning that all elements in it are distinct from one another, and (2) its elements all belong to the domain of the heap.

To do this, scan inspects the heap and considers three different cases. If the heap is a singleton heap $x \mapsto v$, then it returns a singleton list containing x. If the heap is the disjoint union of heaps l and r, it proceeds recursively on each subheap and returns the concatenation of the lists obtained in the recursive calls. Finally, if the heap doesn't match any of the previous cases, then it returns an empty list. Note that this case analysis is not possible using Coq's standard **match** mechanism, because **match** only pattern-matches against primitive datatype constructors. In the case of heaps, which are really finite maps from pointers to values, $x \mapsto v$ and $l \bullet r$ are applications not of primitive dataype constructors but of defined functions ($\mapsto$ and $\bullet$). Thus, in order to perform our desired case analysis, we require the ability of Mtac's **mmatch** mechanism to pattern-match against the *syntax* of heap expressions.

```
Record form h := Form {
        seq_of :> list ptr;
        axiom_of : def h → uniq seq_of
                        ∧ ∀ x. x ∈ seq_of → x ∈ dom h }.

Definition scan :=
  mfix f [h : heap] : ○(form h) :=
    mmatch h with
    | [x A (v:A)] x ↦ v ⇒ ret (Form [x] ...)
    | [l r] l • r ⇒
      rl ← f l;
      rr ← f r;
      ret (Form (seq_of rl ++ seq_of rr) ...)
    | [h'] h' ⇒ ret (scan_h [] ...)
    end.
```

Figure 3. Mtactic for scanning a heap to obtain a list of pointers.

```
Definition search2 x y :=
  mfix f [s] : ○(uniq s → x != y) :=
    mmatch s with
    | [s'] x :: s' ⇒ r ← search y s'; ret (foundx_pf x r)
    | [s'] y :: s' ⇒ r ← search x s'; ret (foundy_pf y r)
    | [z s'] z :: s' ⇒ r ← f s'; ret (foundz_pf z r)
    | _ ⇒ raise NotFound
    end.
```

Figure 4. Mtactic for searching for two pointers in a list.

In each case, scan also returns a proof that the output list obeys the aforementioned properties (1) and (2). For presentation purposes, we omit these proofs (denoted with ... in the figures), but they are proven as standard Coq lemmas. (We will continue to omit proofs in this way throughout the paper when they present no interesting challenges. The reader can find them in the source files.)

The Mtactic **search2.** Figure 4 presents the Mtactic search2. It takes two elements x and y and a list s as input, and searches for x and y in s. If successful, search2 returns a proof that, if s is unique, then x is distinct from y. Similarly to scan, this involves a syntactic inspection and case analysis of the input list s.

When s contains x at the head (*i.e.*, s is of the form $x :: s'$), search2 searches for y in the tail s', using the Mtactic search from §1.1. If this search is successful, producing a proof $r : y \in s'$, then search2 concludes by composing this proof together with the assumption that s is unique, using the easy lemma foundx_pf:

$$\begin{aligned}&\mathsf{foundx\_pf} : \forall x\ y : \mathsf{ptr}.\ \forall s : \mathsf{list\ ptr}.\\ &\qquad y \in s \to \mathsf{uniq}\ (x :: s)\ \to x\ \texttt{!=}\ y\end{aligned}$$

(In the code, the reader will notice that foundx_pf is not passed the arguments y and s explicitly. That is because they are inferrable from the type of r, and thus are treated as implicit arguments.)

```
Definition noalias h (D : def h) : ○(∀ x y. ○(x != y)) :=
  sc ← scan h;
  ret (λ x y ⇒
    s₂ ← search2 x y (seq_of sc);
    ret (combine s₂ D)).
```

Figure 5. Mtactic for proving that two pointers do not alias.

If s contains y at the head, search2 proceeds analogously. If the head element is different from both x and y, then it calls itself recursively with the tail. In any other case, it throws an exception.

Note that, in order to test whether the head of s is x or y, we rely crucially on the ability of patterns to mention free variables from the context. In particular, the difference between the first two cases of search2's **mmatch** and the last one is that the first two do *not* bind x and y in their telescope patterns (thus requiring the head of the list in those cases to be syntactically unifiable with x or y, respectively), while the third *does* bind z in its telescope pattern (thus enabling z to match anything).

***The Mtactic* noalias.** Figure 5 shows the very short code for the Mtactic noalias, which stitches scan and search2 together. The type of noalias is as follows:

$$\forall h : \mathsf{heap}.\ \mathsf{def}\ h \to \bigcirc(\forall x\, y.\ \bigcirc(x\ \texttt{!=}\ y))$$

As the two occurrences of ○ indicate, this Mtactic is *staged*: it takes as input a proof that h is defined and first runs the scan Mtactic on h, producing a list of pointers sc, but then it immediately returns *another* Mtactic. This latter Mtactic in turn takes as input x and y and searches for them in sc. The reason for this staging is that we may wish to prove non-aliasing facts about different pairs of pointers in the same heap. Thanks to staging, we can apply noalias to some D just once and then reuse the Mtactic it returns on many different pairs of pointers, thus avoiding the need to rescan h redundantly.

At the end, the proofs returned by the calls to scan and search2 are composed using a combine lemma with the following type:

```
Lemma combine h x y (sc : form h) :
  (uniq (seq_of sc) → x != y) → def h → x != y.
```

This lemma is trivial to prove by an application of the cut rule.

***Applying the Mtactic* noalias.** The following script shows how noalias can be invoked in order to solve the motivating example from the beginning of this section:

```
pose F := run (noalias D)
by split; apply: run (F _ _)
```

When Coq performs type inference on the **run** in the first line, that forces the execution of (the first scan-ning phase of) the Mtactic noalias on the input hypothesis D, and the standard pose mechanism then binds the result to F. This F has the type

$$\forall x\, y : \mathsf{ptr}.\ \bigcirc(x\ \texttt{!=}\ y)$$

In the case of our motivating example, F will be an Mtactic that, when passed inputs x and y, will search for those pointers in the list $[x_1; x_2; x_3]$ output by the scan phase.

The script continues with Coq's standard split tactic, which generates two subgoals, one for each proposition in the conjunction. For our motivating example, it generates subgoals $x_1\ \texttt{!=}\ x_2$ and $x_2\ \texttt{!=}\ x_3$. We then solve both goals by executing the Mtactic F. When F is **run** to solve the first subgoal, it will search for x_1 and x_2 in $[x_1; x_2; x_3]$ and succeed; when F is **run** to solve the second subgoal, it will search for x_2 and x_3 in $[x_1; x_2; x_3]$ and succeed. QED. Note that we provide the arguments to F *implicitly* (as _). As in the proof of the z_in_xyz lemma from §1.1, these arguments are inferred from the respective goals being solved. (We will explain how this inference works in more detail in §5.)

```
Program Definition interactive_search2 x y :=
  mfix f [s] : ○(uniq s → x != y) :=
    mmatch s with
    | [s'] x :: s' ⇒ r ← search y s'; ret _
    | [s'] y :: s' ⇒ r ← search x s'; ret _
    | [z s'] z :: s' ⇒ r ← f s'; ret _
    | _ ⇒ raise NotFound
    end.
Next Obligation. ... Qed.
Next Obligation. ... Qed.
Next Obligation. ... Qed.
```

Figure 6. Interactive construction of search2 using **Program**.

Developing Mtactics Interactively. One key advantage of Mtac is that it works very well with the rest of Coq, allowing us among other things to develop Mtactics interactively.

For instance, consider the code shown in Figure 6. This is an interactive development of the search2 Mtactic, where the developer knows the overall search structure in advance, but not the exact proof terms to be returned, as this can be difficult in general. Here, we have prefixed the definition with the keyword **Program** [27], which allows us to omit certain parts of the definition by writing underscores. **Program** instructs the type inference mechanism to treat these underscores as unification variables, which—unless instantiated during type inference—are exposed as proof obligations. In our case, none of these underscores is resolved, and so we are left with three proof obligations. Each of these obligations can then be solved interactively within a **Next Obligation** . . . **Qed** block.

Finally, it is worth pointing out that within such blocks, as well as within the actual definitions of Mtactics, we could be running other more primitive Mtactics.

3.2 tauto: A Simple First-Order Tautology Prover

With this next example, we show how Mtac provides a simple but useful way to write tactics that manipulate contexts and binders. Specifically, we will write an Mtactic implementing a rudimentary tautology prover, modeled after those found in the work on VeriML [29] and Chlipala's CPDT textbook [5]. Compared to VeriML, our approach has the benefit that it does not require any special type-theoretic treatment of contexts: for us, a context is nothing more than a Coq list. Compared to Chlipala's Ltac version, our version is typed, offering a clear static specification of what the tautology prover produces, if it succeeds.

To ease the presentation, we break the problem in two. First, we show a simple propositional prover that uses the language constructs we have presented so far. Second, we extend this prover to handle first-order logic, and we use this extension to motivate some additional features of Mtac.

Warming up the Engine: A Simple Propositional Prover. Figure 7 displays the Mtactic for a simple propositional prover, taking as input a proposition p and, if successful, returning a proof of p:

$$\mathsf{prop\text{-}tauto} : \forall p : \mathsf{Prop}.\ \bigcirc p$$

The Mtactic only considers three cases:

- p is True. In this case, it returns the trivial proof I.
- p is a conjunction of p_1 and p_2. In this case, it proves both propositions and returns the introduction form of the conjunction (conj r_1 r_2).
- p is a disjunction of p_1 and p_2. In this case, it tries to prove the proposition p_1, and if that fails, it tries instead to prove

```
Definition prop-tauto :=
  mfix f [p : Prop] : ○p :=
    mmatch p as p' return ○p' with
    | True ⇒ ret I
    | [p₁ p₂] p₁ ∧ p₂ ⇒
          r₁ ← f p₁;
          r₂ ← f p₂;
          ret (conj r₁ r₂)
    | [p₁ p₂] p₁ ∨ p₂ ⇒
          mtry
            r₁ ← f p₁; ret (or_introl r₁)
          with _ ⇒
            r₂ ← f p₂; ret (or_intror r₂)
          end
    | _ ⇒ raise NotFound
    end.
```

Figure 7. Mtactic for a simple propositional tautology prover.

```
01    Definition tauto' :=
02      mfix f [c : list dyn; p : Prop] : ○p :=
03        mmatch p as p' return ○p' with
04        | True ⇒ ret I
05        | [p₁ p₂] p₁ ∧ p₂ ⇒
06              r₁ ← f c p₁ ;
07              r₂ ← f c p₂ ;
08              ret (conj r₁ r₂)
09        | [p₁ p₂] p₁ ∨ p₂ ⇒
10              mtry
11                r₁ ← f c p₁ ; ret (or_introl r₁)
12              with _ ⇒
13                r₂ ← f c p₂ ; ret (or_intror r₂)
14              end
15        | [p₁ p₂ : Prop] p₁ → p₂ ⇒
16              ν (y:p₁).
17                r ← f (Dyn p₁ y :: c) p₂;
18                abs y r
19        | [A (q:A → Prop)] (∀ x:A. q x) ⇒
20              ν (y:A).
21                r ← f c (q y);
22                abs y r
23        | [A (q:A → Prop)] (∃ x:A. q x) ⇒
24              X ← evar A;
25              r ← f c (q X);
26              b ← is_evar X;
27              if b then raise ProofNotFound
28              else ret (ex_intro q X r)
29        | [p':Prop] p' ⇒ lookup p' c
30        end.
```

Figure 8. Mtactic for a simple first-order tautology prover.

the proposition p_2. The corresponding introduction form of the disjunction is returned (or_introl r_1 or or_intror r_2).

- Otherwise, it raises an exception, since no proof could be found.

Extending to First-Order Logic. We now extend the previous prover to support first-order logic. This extension requires the tactic to keep track of a context for hypotheses, which we model as a list of (dependent) pairs pairing hypotheses with their proofs. More concretely, each element in the hypothesis context has the type dyn $= \Sigma p : \mathsf{Prop}.\ p$. (In Coq, this is encoded as an inductive type with constructor Dyn $p\ x$, for any $x : p$.)

```
Definition lookup (p : Prop) :=
  mfix f [s : list dyn] : ○p :=
    mmatch s return ○p with
    | [x s'] (Dyn p x) :: s' ⇒ ret x
    | [d s'] d :: s' ⇒ f s'
    | _ ⇒ raise ProofNotFound
    end.
```

Figure 9. Mtactic to look up a proof of a proposition in a context.

Figure 8 shows the first-order logic tautology prover tauto. The fixed point takes the proposition p and is additionally parameterized over a context (c : list dyn). The first three cases of the **mmatch** are similar to the ones in Figure 7, with the addition that the context is passed around in recursive calls.

Before explaining the cases for $\rightarrow$, $\forall$ and $\exists$, let us start with the last one (line 29), since it is the easiest. In this last case, we attempt to prove the proposition in question by simply searching for it in the hypothesis context. The search for the hypothesis p' in the context c is achieved using the Mtactic lookup shown in Figure 9. lookup takes a proposition p and a context, and traverses the context linearly in the hope of finding a dependent pair with p as the first component. If it finds such a pair, it returns the second component. Like the Mtactic search2 from §3.1, this simple lookup routine depends crucially on the ability to match the propositions in the context *syntactically* against the p for which we are searching.

Returning to the tautology prover, lines 15–18 concern the case where $p = p_1 \rightarrow p_2$. Intuitively, in order to prove $p_1 \rightarrow p_2$, one would (1) introduce a *parameter* y witnessing the proof of p_1 into the context, (2) proceed to prove p_2 having $y : p_1$ as an assumption, and (3) abstract any usage of y in the resulting proof. The rationale behind this last step is that if we succeed proving p_2, then the result is *parametric* over the proof of p_1, in the sense that any proof of p_1 will suffice to prove p_2. Steps (1) and (3) are performed by two of the operators we haven't yet described: **nu** and **abs** (the former is denoted by the νx binder). In more detail, the three steps are:

Line 16: It creates a parameter $y : p_1$ using the constructor **nu**. This constructor has type

$$\mathbf{nu} : \forall (A\ B : \mathsf{Type}).\ (A \rightarrow \bigcirc B) \rightarrow \bigcirc B$$

(where A and B are left implicit). It is similar to the operator with the same name in [19] and [26]. Operationally, $\nu x : A.\ f$ (which is notation for **nu** $(\lambda x : A.\ f)$), creates a parameter y with type A, pushes it into the local context, and executes $f\{y/x\}$ (where $\cdot\{\cdot/\cdot\}$ is the standard substitution) in the hope of getting a value of type B. If the value returned by f refers to y, then it causes the tactic execution to fail: such a result would lead to an ill-formed term because y is not bound in the ambient context. This line constitutes the first step of our intuitive reasoning: we introduce the parameter y witnessing the proof of p_1 into the context.

Line 17: It calls tauto$'$ recursively, with context c extended with the parameter y, and with the goal of proving p_2. The result is bound to r. This line constitutes the second step.

Line 18: The result r created in the previous step has type p_2. In order to return an element of the type $p_1 \rightarrow p_2$, we abstract y from r, using the constructor

$$\begin{aligned}\mathbf{abs} : \forall (A : \mathsf{Type})\ (P : A \rightarrow \mathsf{Type})\ (y : A).\\ P\ y \rightarrow \bigcirc(\forall x : A.\ P\ x)\end{aligned}$$

(with A, P implicit). Operationally, **abs** $y\ r$ checks that the first parameter y is indeed a variable, and returns the function

$$\lambda x : A.\ r\{x/y\}$$

In this case, the resulting element has type $\forall x : p_1.\, p_2$, which, since p_2 does not refer to x, is equivalent to $p_1 \to p_2$. This constitutes the last step: by abstracting over y in the result, we ensure that the resulting proof term no longer mentions the ν-bound variable (as required by the use of **nu** in line 16).

Lines 19–22 consider the case that the proposition is an abstraction $\forall x : A.\ q\ x$. Here, q is the body of the abstraction, represented as a function from A to Prop. We rely on Coq's use of higher-order pattern unification [18] to instantiate q with a faithful representation of the body. The following lines mirrors the body of the previous case, except for the recursive call. In this case we don't extend the context with the parameter y, since it is not a proposition. Instead, we try to recursively prove the body q replacing x with y (that is, applying q to y).

If the proposition is an existential $\exists x : A.\ q\ x$ (line 23), then the prover performs the following steps:

Line 24: It uses Mtac's **evar** constructor to create a fresh unification variable called X.

Line 25: It calls tauto′ recursively, replacing x for X in the body of the existential.

Lines 26–28: It uses Mtac's **is_evar** mechanism to check whether X is still an uninstantiated unification variable. If it is, then it raises an exception, since no proof could be found. If it is not—that is, if X was successfully instantiated in the recursive call—then it returns the introduction form of the existential, with X as its witness.

Now we are ready to prove an example, where P : nat $\to$ Prop:

Definition exmpl : $\forall P\ x.\ P\ x \to \exists y.\ P\ y$:= **run** (tauto [] _).

The proof term generated by **run** is

$$\mathsf{exmpl} = \lambda P\ x\ (H : P\ x).\ \mathsf{ex\_intro}\ P\ x\ H$$

3.3 Inlined Proof Automation

Due to the tight integration between Mtac and Coq, Mtactics can be usefully employed in definitions, notations and other Coq terms, in addition to interactive proving. In this respect, Mtac differs from the related systems such as VeriML [29] and Beluga [21], where, to the best of our knowledge, such expressiveness is not currently available due to the strict separation between the object logic and the automation language.

In this section, we illustrate how Mtactics can be invoked from Coq proper. To set the stage, consider the scenario of developing a library for n-dimensional integer vector spaces, with the main type vector n defined as a record containing a list of nats and a proof that the list has size n:

```
Record vector (n : nat) := Vector {
  seq_of : list nat;
  _ : size seq_of = n}.
```

One of the important methods of the library is the accessor function ith, which returns the i-th element of the vector, for $i < n$. One implementation possibility is for ith to check at run time if $i < n$, and return an option value to signal when i is out of bounds. The downside of this approach is that the clients of ith have to explicitly discriminate against the option value. An alternative is for ith to explicitly request a proof that $i < n$ as one of its arguments, as in the following type ascription:

$$\mathsf{ith} : \forall n{:}\mathsf{nat}.\mathsf{vector}\ n \to \forall i{:}\mathsf{nat}.i < n \to \mathsf{nat}$$

Then the clients have to construct a proof of $i < n$ before invoking ith, but we show that in some common situations, the proof can be constructed automatically by Mtac, and then passed to ith.

```
Program Definition compare (n1 n2 : nat) : ○(n1 ≤ n2) :=
  r1 ← to_ast nil n1;
  r2 ← to_ast (ctx_of r1) n2;
  match cancel (ctx_of r2) (term_of r1) (term_of r2)
       return ○(n1 ≤ n2) with
  | true ⇒ ret (@sound n1 n2 r1 r2 _)
  | _ ⇒ raise NotLeqException
  end.
Next Obligation. ... Qed.
```

Figure 10. Mtactic for proving inequalities between nat's.

Specifically, we describe an Mtactic compare, which automatically searches for a proof that two natural numbers n_1 and n_2 satisfy $n_1 \leq n_2$. compare is incomplete, and if it fails to find a proof, because the inequality doesn't hold, or because the proof is too complex, it raises an exception.

Once compare is implemented, it can be composed with ith as follows. Given a vector v whose size we denote as vsize v, and an integer i, we introduce the following notation, which invokes compare to automatically construct a proof that $i + 1 \leq$ vsize v (equivalent to $i <$ vsize v).

```
Notation "[ 'ith' v i ]" :=
  (@ith _ v i (run (compare (i+1) (vsize v))))
```

The notation can be used in definitions. For example, given vectors v_1, v_2 of fixed size 2, we could define the inner product of v_1 and v_2 as follows, letting Coq figure out automatically that the indices $0, 1$ are within bounds.

```
Definition inner_prod (v1 v2 : vector 2) :=
        [ith v1 0] × [ith v2 0] + [ith v1 1] × [ith v2 1].
```

If we tried to add the summand [ith v_1 2] $\times$ [ith v_2 2], where the index 2 is out of bounds, then compare raises an exception, making the whole definition ill-typed. Similarly, if instead of vector 2, we used the type vector n, where n is a variable, the definition will be ill-typed, because there is no guarantee that n is larger than 1. On the other hand, the following is a well-typed definition, as the indices k and n are clearly within the bound $n + k + 1$.

```
Definition indexing n k (v : vector (n + k + 1)) :=
  [ith v k] + [ith v n].
```

We proceed to describe the implementation of compare, presented in Figure 10. compare is implemented using two main helper functions. The first is the Mtactic to_ast which *reflects* the numbers n_1 and n_2. More concretely, to_ast takes an integer expression, and considers it as a syntactic summation of a number of components. It *parses* this syntactic summation into an explicit list of summands, each of which can be either a constant or a free variable (subexpressions containing operations other than + are treated as free variables).

The second helper is a CIC function cancel which cancels the common terms from the syntax lists obtained by reflecting n_1 and n_2. If all the summands in the syntax list of n_1 are found in the syntax list of n_2, then it must be that $n_1 \leq n_2$ and cancel returns the boolean true. Otherwise, cancel doesn't search for other ways of proving $n_1 \leq n_2$ and simply returns false to signal the failure to find a proof. This failure ultimately results in compare raising an exception. Notice that cancel can't directly work on n_1 and n_2, but has to receive their syntactic representation from to_ast (in the code of compare these are named term_of r_1 and term_of r_2, respectively). The reason is that cancel has to compare names of variables appearing in n_1 and n_2, and has to match against the occurrences of the (non-constructor) function +, and such comparisons and matchings are not possible in CIC.

Alternatively, we could use **mmatch** to implement cancel in Mtac, but there are good reasons to prefer a purely functional Coq implementation when one is possible, as is the case here. With a pure cancel, compare can return a very short proof term as a result (*e.g.*, (sound n_1 n_2 r_1 r_2 _) in the code of compare). An Mtac implementation would have to expose the reasoning behind the soundness of the Mtactic at a much finer granularity, resulting in a larger proof.

We next describe the implementations of the two helpers.

Data Structures for Reflection. There are two main data structures used for reflecting integer expressions. As each expression is built out of variables, constants and $+$, we syntactically represent the sum as term containing a list of syntactic representations of variables appearing in the expression, followed by a nat constant that sums up all the constants from the expression. We also need a type of variable contexts ctx, in order to determine the syntactic representation of variables. In our case, a variable context is simply a list of nat expression, each element standing for a different variable, and the position of the variable in the context serves as the variable's syntactic representative.

```
Definition ctx := list nat
Record var := Var of nat
Definition term := (list var) × nat
```

Example 1. The expression $n = (1+x)+(y+3)$ may be reflected using a variable context $c = [x, y]$, and a term ([Var 0, Var 1], 4). Var 0 and Var 1 correspond to the two variables in c (x and y, respectively). 4 is the sum of the constants appearing in n.

Example 2. The syntactic representations of 0, successor constructor S, addition and an individual variable, may be given as following term constructors. We use _.1 and _.2 to denote projections out of a pair.

```
Definition syn_zero : term := (nil, 0).
Definition syn_succ (t : term) := (t.1, t.2 + 1).
Definition syn_add (t1 t2 : term) :=
    (t1.1 ++ t2.1, t1.2 + t2.2).
Definition syn_var (i : nat) := ([Var i], 0).
```

In prose, 0 is reflected by an empty list of variable indexes, and 0 as a constant term; if t is a term reflecting n, then the successor S n is reflected by incrementing the constant component of t, etc.

We further need a function interp that takes a variable context G and a term t, and *interprets* t into a nat, as follows.

```
interp_vars (G : ctx) (t : list var) :=
  if t is j :: t' then
    if (vlook G j, interp G t') is (Some v, Some e)
      then Some (v + e) else None
  else Some 0.

interp (G : ctx) (t : term) :=
  if interp_vars G t.1 is Some e
  then Some (e + t.2) else None.
```

First, interp_vars traverses the list of variable indices of t, turning each index into a natural number by looking it up into the context G, and summing the results. The lookup function vlook G j is omitted here, but it either returns Some j-th element of the context G, or None if G has less than j elements. Then, interp simply adds the result of interp_vars to the constant part of the term. For example, if the context $c = [x, y]$ and term $t = ([\mathsf{Var}\ 0, \mathsf{Var}\ 1], 4)$, then interp c t equals Some $(x + y + 4)$.

***Reflection by* to_ast.** The to_ast Mtactic is applied twice in compare: once to reflect n_1, and again to reflect n_2. Each time,

```
Record ast (G : ctx) (n : nat) :=
  Ast {term_of : term;
       ctx_of : ctx;
       _ : interp ctx_of term_of = Some n ∧ prefix G ctx_of}

Definition to_ast : ∀ G n. ○(ast G n) :=
  mfix f [G ; n] :=
    mmatch n with
    | 0 ⇒ ret (Ast G 0 syn_zero G ...)
    | [n'] S n' ⇒
      r ← f G n';
      ret (Ast G (S n') (syn_succ (term_of r))
             (ctx_of r) ...)
    | [n1 n2] n1 + n2 ⇒
      r1 ← f G n1; r2 ← f (ctx_of r1) n2;
      ret (Ast G (n1 + n2)
        (syn_add (term_of r1) (term_of r2)) (ctx_of r2) ...)
    | _ ⇒
      ctx_index ← find n G;
      ret (Ast G n (syn_var ctx_index.2) ctx_index.1 ...)
    end.
```

Figure 11. Mtactic for reflecting nat expressions.

```
Fixpoint cancel_vars (G : ctx) (s1 s2 : list var) : bool :=
  if s1 is v :: s1' then v ∈ s2 &&
    cancel_vars G s1' (remove_var v s2)
  else true.

Definition cancel (G : ctx) (t1 t2 : term) : bool :=
  cancel_vars G t1.1 t2.1 && t1.2 ≤ t2.2.
```

Figure 12. Algorithm for canceling common variables from terms.

to_ast is passed as input a variable context, and extends this context with new variables encountered during reflection. To reflect n_1 in compare, to_ast starts with the empty context nil, and to reflect n_2, it starts with the context obtained after the reflection of n_1. This ensures that if the reflections of n_1 and n_2 encounter the same variables, they will use the same syntactic representations for them.

The invariants associated with to_ast are encoded in the data structure ast (Figure 11). ast is indexed by the input context G and the number n to be reflected. Upon successful termination of to_ast, the term_of field contains the term reflecting n, and the ctx_of field contains the new variable context, potentially extending G. The third field of ast is a proof formalizing the described properties of term_of and ctx_of.

Referring to Figure 11, the Mtactic to_ast takes the input variable context G and the number n to be reflected, and traverses n trying to syntactically match the head construct of n with 0, S or $+$, respectively. In each case it returns an ast structure containing the syntactic representation of n, e.g.: syn_zero, syn_succ or syn_add, respectively. In the $n_1 + n_2$ case, to_ast recurses into n_2 by using the variable context returned from reflection of n_1 as an input, similar as in compare. In each case, the Ast constructor is supplied a proof that we omit but can be found in the sources. In the default case, when no constructor matches, n is treated as a variable. The Mtactic find n G (omitted here), searches for n in G, and returns a ctx × nat pair. If n is found, the pair consists of the old context G, and the position of n in G. If n is not found, the pair consists of a new context in which n is cons-ed to G, and the index k, where k is the index of n in the new context. to_ast then repackages the context and the index into an ast structure.

Canceling Common Variables. The cancel function is presented in Figure 12. It takes a variable context G, and terms t_1 and t_2 and tries to determine if t_1 and t_2 syntactically represent two $\leq$-related

$$\frac{\Gamma; \Sigma \vdash t \overset{\mathsf{whd}}{\rightsquigarrow} t'}{\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma; t')} \qquad \frac{}{\mathbf{mfix}\ f\ t \rightsquigarrow f\ (\mathbf{mfix}\ f)\ t}$$

$$\frac{t \rightsquigarrow t'}{\mathbf{bind}\ t\ f \rightsquigarrow \mathbf{bind}\ t'\ f} \qquad \frac{t \rightsquigarrow t'}{\mathbf{mtry}\ t\ f \rightsquigarrow \mathbf{mtry}\ t'\ f}$$

$$\frac{}{\mathbf{bind}\ (\mathbf{ret}\ t)\ f \rightsquigarrow f\ t} \qquad \frac{}{\mathbf{bind}\ (\mathbf{raise}\ t)\ f \rightsquigarrow \mathbf{raise}\ t}$$

$$\frac{}{\mathbf{mtry}\ (\mathbf{ret}\ t)\ f \rightsquigarrow \mathbf{ret}\ t} \qquad \frac{}{\mathbf{mtry}\ (\mathbf{raise}\ t)\ f \rightsquigarrow f\ t}$$

$$\frac{\begin{array}{c} ps_i \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ \mathsf{Ptele}\ \overline{x}\ (\mathsf{Pbase}\ p\ b) \\ \Sigma, \overline{?y} \vdash p\{\overline{?y/x}\} \approx t \triangleright \Sigma', \overline{?y := t'} \\ \forall j < i.\ ps_j \text{ does not unify with } t \end{array}}{\Gamma \vdash (\Sigma; \mathbf{mmatch}\ t\ ps) \rightsquigarrow (\Sigma'; b\{\overline{t'/x}\})}$$

$$\frac{?x \notin \mathsf{dom}(\Sigma)}{(\Sigma; \mathbf{evar}_A) \rightsquigarrow (\Sigma, ?x : A; \mathbf{ret}\ ?x)}$$

$$\frac{e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ ?x \quad (?x := \_) \notin \Sigma}{(\Sigma; \mathbf{is\_evar}\ e) \rightsquigarrow (\Sigma; \mathbf{ret}\ \mathsf{true})} \qquad \frac{e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ t \quad t \text{ not unif. variable}}{(\Sigma; \mathbf{is\_evar}\ e) \rightsquigarrow (\Sigma; \mathbf{ret}\ \mathsf{false})}$$

$$\frac{\Gamma, x : A \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')}{\Gamma \vdash (\Sigma; \nu x : A.\ t) \rightsquigarrow (\Sigma'; \nu x : A.\ t')} \qquad \frac{x \notin \mathsf{FV}(v)}{(\nu x.\ v) \rightsquigarrow v}$$

$$\frac{e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ x \quad (x : A) \in \Gamma}{\Gamma \vdash \mathbf{abs}\ e\ t \rightsquigarrow \mathbf{ret}\ (\lambda y.\ t\{y/x\})} \qquad \frac{\textit{(* print s to stdout *)}}{\mathbf{print}\ s \rightsquigarrow \mathbf{ret}\ \langle\rangle}$$

Figure 13. Operational small-step semantics.

expressions by cancelling common terms, as we described previously. First, the helper cancel_vars iterates over the list of variable representations of t_1, trying to match each one with a variable representation in t_2 (in the process, removing the matched variables by using yet another helper function remove_vars, omitted here). If the matching is successful and all variables of t_1 are included in t_2, then cancel merely needs to check if the constant of t_1 is smaller than the constant of t_2.

We conclude the example with the statement of the correctness lemma of cancel, which is the key component of the soundness proof for compare. We omit the proof here, but it can be found in our Coq files.

Lemma sound n_1 n_2 (a_1 : ast [] n_1) (a_2 : ast (ctx_of a_1) n_2) :
  cancel (ctx_of a_2) (term_of a_1) (term_of a_2) →
  $n_1 \leq n_2$.

In prose, let a_1 and a_2 be reflections of n_1 and n_2 respectively, where the reflection of a_1 starts in the empty context, and the reflection of a_2 starts in the variable context returned by a_1. Then running cancel in the final context of a_2 over the reflected terms of a_1 and a_2 returns true only when it is correct to do so; that is, only when $n_1 \leq n_2$.

4. Operational Semantics

In this section, we let e, e' range over CIC terms and t, t' over Mtactics, *i.e.*, CIC terms of type $\bigcirc A$ for some type A. The operational semantics of Mtac defines the judgment form

$$\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$$

where Γ is the typing context containing parameters and (let-bound) local definitions, while Σ and Σ' are contexts for unification variables $?x$. Both kinds of contexts contain both variable declarations (standing for parameters and uninstantiated unification variables, respectively) and definitions (let-bound variables and instantiated unification variables, respectively).

$$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, x : A := e$$
$$\Sigma ::= \cdot \mid \Sigma, ?x : A \mid \Sigma, ?x : A := e$$

These contexts are needed for weak head reduction of CIC terms ($\Gamma; \Sigma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow} e'$), but also for some of Mtac constructs. Except where noted, every rule just passes around the contexts, so to avoid clutter we omit them. We also omit the types, although we assume that the terms are well-typed in their given contexts and we ensure that this invariant is maintained throughout execution.

Tactic computation may either (a) terminate successfully returning a term, **ret** e, (b) terminate by throwing an exception, **raise** e, (c) diverge, or (d) get blocked. (We explain the possible reasons for getting blocked below.) Hence we have the following tactic values:

Definition 1 (Values). $v \in \text{Values} ::= \mathbf{ret}\ e \mid \mathbf{raise}\ e\,.$

Figure 13 shows our operational semantics. The first rule performs a CIC weak head reduction step. Weak head reduction requires both contexts because, among other things, it will unfold definitions of variables and unification variables in head position. For a precise description of Coq's standard reduction rules, see §4.3 of Coq's Reference Manual [31].

Next, we have the standard unfolding of Mtac fixpoints. The next six rules are quite standard and concern the semantics of **bind** and **mtry**: note the symmetry between **bind**/**ret** and **mtry**/**raise**.

The most complex rule is the next one concerning pattern matching. It matches the term t with some pattern described in the list ps. Each element ps_i of ps is a pair containing a pattern p and a body b, abstracted over a list of (dependent) variables $\overline{x : A}$. Since patterns are first class citizens in CIC, ps_i is first reduced to weak head normal form in order to expose the pattern and the body. The normalization relation is written $\overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}$ and, as with the weak head reduction relation, it requires the two contexts that we omit for clarity. Then, we replace each variable x with a corresponding unification variable $?y$ in p, and proceed to unify the result with term t. For this, the context Σ is extended with the freshly created unification variables $\overline{?y}$. After unification is performed, a new unification variable context is returned that might not only instantiate the freshly generated unification variables $\overline{?y}$, but may also instantiate previously defined unification variables. (Instantiating such unification variables is important, for instance, to instantiate the existentials in the tautology prover example of §3.2). We actually require that unification unifies all the freshly generated variables, so that we can safely remove them after substituting them in the body, thereby avoiding context pollution. Finally, we require that patterns are tried in sequence, *i.e.*, that the scrutinee, t, should not be unifiable with any previous pattern ps_j. In case no patterns match the scrutinee, the **mmatch** is blocked.

The semantics for pattern matching is parametric with respect to the unification judgment and thus does not rely on any particular unification algorithm. (Our implementation uses Coq's standard unification algorithm.) We observe that our examples, however, implicitly depend on *higher-order pattern unification* [18]. Higher-order unification is in general undecidable, but Miller identified a decidable subset of problems, the *pattern* fragment, where unification variables appear only in equations of the form $?f\ x_1\ \ldots\ x_n \approx t$, with $x_1, \ldots, x_n$ distinct variables. The $\forall$ and $\exists$ cases of the tautology prover (§3.2) fall into this pattern fragment, and their proper handling depends on higher-order pattern unification.

Another notable aspect of Coq's unification algorithm is that it equates terms up to definitional equality. In particular, if a pattern match at first does not succeed, Coq will take a step of reduction on the scrutinee, try again, and repeat. Thus, the ordering of two patterns in a **mmatch** matters, even if it seems the patterns are syntactically non-overlapping. Take for instance the search example in §1.1. If the pattern for concatenation of lists were moved *after* the patterns for consing, then the consing patterns would actually match against (many) concatenations as well, since the concatenation of two lists is often reducible to a term of the form h :: t.

Related to this, the last aspect of Coq's unification algorithm that we depend on is its *first-order approximation*. That is, in the presence of an equation of the form $c\ t_1\ \ldots\ t_n \approx c\ t'_1\ \ldots\ t'_n$, where c is a constant, the unification algorithm tries to equate each $t_i \approx t'_i$. While this may cause Coq to miss out on some solutions, it has the benefit of being simple and predictable. For instance, consider the equation

$$?l{+}{+}?r \approx [\,]{+}{+}(h :: t)$$

that might result from matching the list $[\,]{+}{+}(h :: t)$ with the pattern for concatenation of lists in the search example from §1.1, with $?l$ and $?r$ fresh unification variables. Here, although there exist many solutions, the algorithm assigns $?l := [\,]$ and $?r := (h :: t)$, an assignment that is intuitively easy to explain.

Coming back to the rules, next is the rule for $\mathbf{evar}_A$, which simply extends Σ with a fresh uninstantiated unification variable of the appropriate type. The two following rules govern **is_evar** e and check whether an expression (after reduction to weak head normal form) is an uninstantiated unification variable.

The next two rules define the semantics of the νx binder: the parameter x is pushed into the context and the execution proceeds until a value is reached. The computed value is simply returned if it does not contain the parameter, x; otherwise, $\nu x.\ v$ is blocked. The latter rule is for abstracting over parameters. If the first argument of **abs** weak-head reduces to a parameter, then we abstract it from the second argument of **abs**, thereby returning a function.

The astute reader may wonder why we decided to have νx and **abs** instead of one single constructor combining the semantics of both. Such a combined constructor would always abstract the parameter x from the result, therefore avoiding the final check that the parameter is not free in the result. The reason we decided to keep **nu** and **abs** separate is simple: it is not always desirable to abstract the parameters in the same order as they were introduced. This is the case, for instance, in the Mtactic skolemize for skolemizing a formula (provided in the Mtac distribution). Moreover, sometimes the parameter is not abstracted at all, for instance in the Mtactic fv for computing the list of free variables of a term (also provided in the Mtac distribution).

Coming back to the rules, finally, the last rule replaces a printing command with the trivial value $\langle\rangle$. Informally, we also print out the string s to the standard output, although standard I/O is not formally modeled here.

Assuming that unification is sound, we can show that Mtac reduction is type-preserving.

Theorem 1 (Type preservation)**.** If $\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$ and $\Gamma; \Sigma \vdash t : \bigcirc A$, then $\Gamma; \Sigma' \vdash t' : \bigcirc A$.

As mentioned earlier, Mtactic execution can block. Here, we define exactly the cases when execution of a term is blocked.

Definition 2 (Blocked terms)**.** A term t is *blocked* if and only if the subterm in reduction position satisfies one of the following cases:

- It is not an application of one of the $\bigcirc$ constructors and it is not reducible using the standard CIC reduction rules ($\overset{\mathsf{whd}}{\rightsquigarrow}$).
- It is $\nu x.\ v$ and $x \in \mathsf{FV}(v)$.
- It is **abs** $e\ t$ and $e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ e'$ and $(e' : \_) \notin \Gamma$.
- It is **mmatch** $t\ ps$ and no pattern in ps unifies with t.

With this definition, we can then also establish a standard type safety theorem for Mtac.

Theorem 2 (Type safety)**.** Whenever $\Gamma; \Sigma \vdash t : \bigcirc A$, then either t is a value, or t is blocked, or there exist t' and Σ' such that $\Gamma \vdash (\Sigma; t) \rightsquigarrow (\Sigma'; t')$ and $\Gamma; \Sigma' \vdash t' : \bigcirc A$.

Example: We show the trace of a simple example to get a grasp of the operational semantics. In this example, $\Gamma = \{h : \mathsf{nat}\}$.

$$\textbf{let}\ s := (h :: [\,]){+}{+}[\,]\ \textbf{in}\ \mathsf{search}\ h\ s$$

We want to show that the final term produced by running this Mtactic expresses the fact that h was found at the head of the list on the left of the concatenation, that is,

$$\mathsf{in\_or\_app}\ (h :: [\,])\ [\,]\ (\mathsf{or\_introl}\ (\mathsf{in\_eq}\ h\ [\,]))$$

First, the **let** is expanded, obtaining

$$\mathsf{search}\ h\ ((h :: [\,]){+}{+}[\,])$$

Then, after expanding the definition of search and β-reducing the term, we are left with the fixpoint being applied to the list:

$$(\textbf{mfix}\ f\ [s : \mathsf{list}\ A] := \ldots)\ ((h :: [\,]){+}{+}[\,])$$

At this point the rule for **mfix** triggers, exposing the **mmatch**:

$$\textbf{mmatch}\ ((h :: [\,]){+}{+}[\,])\ \textbf{with} \ldots \textbf{end}$$

Thanks to first-order approximation, the case for append is unified, and its body is executed:

$$\textbf{mtry}\ il \leftarrow f\ (h :: [\,]);\ \textbf{ret}\ \ldots\ \textbf{with}\ \_ \Rightarrow \ldots\ \textbf{end} \qquad (1)$$

where f stands for the fixpoint. The rule for **mtry** executes the code for searching for the element in the sublist $(h :: [\,])$:

$$il \leftarrow f\ (h :: [\,]);\ \textbf{ret}\ (\mathsf{in\_or\_app}\ (h :: [\,])\ [\,]\ h\ (\mathsf{or\_introl}\ il)) \qquad (2)$$

The bind rule triggers, after which the fixpoint is expanded and a new **mmatch** exposed:

$$\textbf{mmatch}\ (h :: [\,])\ \textbf{with}\ \ldots\ \textbf{end}$$

This time, the rule for append fails to unify, but the second case succeeds, returning the result in_eq h $[\,]$. Coming back to (2), il is replaced with this result, getting the expected final result that is in turn returned by the **mtry** of (1).

As a last remark, notice how at each step the selected rule is the only applicable one: the semantics of Mtac is deterministic.

5. Implementation

This section presents a high-level overview of the architecture of our Mtac extension to Coq, explaining our approach for guaranteeing soundness even in the possible presence of bugs in our Mtac implementation.

The main idea we leverage in integrating Mtac into Coq is that Coq distinguishes between fully and partially type-annotated proof terms: Coq's type inference (or *elaboration*) algorithm transforms partially annotated terms into fully annotated ones, which are then fed to Coq's kernel type checker. In this respect Coq follows the typical architecture of interactive theorem provers, ensuring that all proofs are ultimately certified by a small trusted kernel. Assuming that the kernel is correct, no code outside this kernel may generate incorrect proofs. Thus, our Mtac implementation modifies only the elaborator lying outside of Coq's kernel, and leaves the kernel type checker untouched.

Extending Elaboration. The typing judgment used by Coq's elaboration algorithm [24, 25] takes a partially type-annotated term e, a local context Γ, a unification variable context Σ, and an optional expected type B, and returns its type A, and produces a fully annotated term e', and updated unification variable context Σ'.

$$\Gamma; \Sigma \vdash_B e \hookrightarrow e' : A \ \triangleright\ \Sigma'$$

If an expected type B, is provided, then the returned type A will be convertible to it, possibly instantiating any unification variables appearing in both A and B. The elaboration judgment serves three main purposes that the kernel typing judgment does not support:

1. To resolve implicit arguments. We have already seen several cases where this is useful (*e.g.*, in §1.1), allowing us to write underscores and let Coq's unification mechanism replace them with the appropriate terms.
2. To insert appropriate coercions. For example, Ssreflect [10] defines the coercion $\mathsf{is\_true} : \mathsf{bool} \to \mathsf{Prop} := (\lambda b.\ b = \mathsf{true})$. So whenever a term of type Prop is expected and a term b of type bool is encountered, elaboration will insert the coercion, thereby returning the term $\mathsf{is\_true}\ b$ having type Prop.
3. To perform canonical structure and type class resolution. Ample examples of canonical structures can be found in Gonthier *et al.* [11]. A type class example will be shown towards the end of this section.

We simply extend the elaboration mechanism to perform a fourth task, namely to run Mtactics. We achieve this by adding the following rule for handling **run** t terms:

$$\frac{\Gamma; \Sigma \vdash_{\bigcirc B} t \hookrightarrow t' : \bigcirc A \triangleright \Sigma' \qquad \Gamma \vdash (\Sigma'; t') \rightsquigarrow^* (\Sigma''; \mathbf{ret}\ e)}{\Gamma; \Sigma \vdash_B \mathbf{run}\ t \hookrightarrow e : A \triangleright \Sigma''}$$

This rule first recursively type-checks the tactic body, while also unifying the return type A of the tactic with the expected goal B (if present). This results in the refinement of t to a new term t', which is then executed. If execution terminates successfully returning a value e (which from Theorem 2 will have type A), then that value is returned. Therefore, as a result of elaboration, all **run** t terms are replaced by the terms produced when running them, and thus the kernel type checker does not need to be modified in any way.

Elaboration and the* apply *Tactic. We have just seen *how* the elaborator coerces the return type A of an Mtactic to be equivalent to the goal B, but we did not stipulate in what situations the knowledge of B is available. Our examples so far assumed B was given, and this was indeed the case thanks to the specific ways we invoked Mtac. For instance, at the end of §1.1 we proved a lemma by *direct definition—i.e.*, providing the proof term directly—and in §3.1 we proved the goal by calling the Ssreflect tactic apply: (note the colon!). In both these situations, we were conveniently relying on the fact that Coq passed the knowledge of the goal being proven into the elaboration of **run**.

Unfortunately, not every tactic does this. In particular, the standard Coq tactic apply (without colon) does not provide the elaborator with the goal as expected type, so if we had written apply (**run** $(F\ \_\ \_)$), the Mtactic F would have been executed on unknown parameters, resulting in a different behavior from what we expect. (Specifically, it would have unified the implicits with the first two pointers appearing in the heap, succeeding only if, luckily, these are the pointers in the goal.)

To ensure that information about the goal is available when running Mtactics, we recommend installing Ssreflect [10]. However, we note that using the standard Coq tactic refine instead of apply also works.

One last point about tactics: Mtac is intended as a typed alternative to Ltac for developing custom automation routines, and it is neither intended to replace the built-in tactics (like apply) nor to subsume all uses of existing Coq tactics. For example, the OCaml tactic vm_compute enables dramatic efficiency gains for reflection-based proofs [12], but its performance depends critically on being *compiled*. Mtac is interpreted, and it is not clear how it could be compiled, given the interaction between Mtac and Coq unification.

```
Class runner A (f : ○A) := { eval : A }.

Hint Extern 20 (runner ?f) ⇒
      (exact (Build_runner f (run f)))
      : typeclass_instances.
```

Figure 14. Type class for delayed execution of Mtactics.

Delaying Execution of Mtactics for Rewriting. Consider the goal from §3.1, after doing pose $F :=$ **run** (noalias D), unfolding the implicit is_true coercions for clarity:

$$\frac{\begin{array}{l} D\ :\ \mathsf{def}\ (h_1 \bullet (x_1 \mapsto v_1 \bullet x_2 \mapsto v_2) \bullet (h_2 \bullet x_3 \mapsto v_3)) \\ F\ :\ \forall x\ y.\ \bigcirc((x \mathrel{!=} y) = \mathsf{true}) \end{array}}{(x_1 \mathrel{!=} x_2) = \mathsf{true} \land (x_2 \mathrel{!=} x_3) = \mathsf{true}}$$

Previously we solved this goal by applying the Mtactic F twice to the two subgoals $x_1 \mathrel{!=} x_2$ and $x_2 \mathrel{!=} x_3$. An alternative way in which a Coq programmer would hope to solve this goal is by using Coq's built-in rewrite tactic. rewrite enables one to apply a lemma one or more times to reduce various *subterms* of the current goal. In particular, we intuitively ought to be able to solve the goal in this case by invoking rewrite !(**run** $(F\ \_\ \_)$), where the ! means that the Mtactic F should be applied repeatedly to solve any and all pointer inequalities in the goal. Unfortunately, however, this does not work, because—like Coq's apply tactic—rewrite typechecks its argument without knowledge of the expected type from the goal, and only later unifies the result with the subterms in the goal. Consequently, just as with apply, F gets run prematurely.

Fortunately, we can circumvent this problem, using a cute trick based on Coq's type class resolution mechanism.

Type classes are an advanced Coq feature similar to canonical structures, with the crucial difference that their resolution is triggered by proof search *after* elaboration [28]. We exploit this functionality in Figure 14, by defining the class runner, which is parameterized over an Mtactic f with return type A and provides a value, eval, of the same type. We then declare a **Hint** instructing the type class resolution mechanism how to build an instance of the runner class, which is precisely by running f.

The details of this implementation are a bit of black magic, and beyond the scope of this paper to explain fully. But intuitively, all that is going on is that eval is *delaying* the execution of its Mtactic argument until type class resolution time, at which point information about the goal to be proven is available.

Returning to our example, we can now use the following script:

rewrite !(eval $(F\ \_\ \_)$) .

This will convert the goal to is_true true $\land$ is_true true, which is trivially solvable.

In fact, with eval we can even employ the standard apply tactic, with the caveat that eval creates slightly bigger proof terms, as the final proof term will also contain the unevaluated Mtactic inside it.

A Word about Exceptions In ML, exceptions have type exn and their constructors are created via the keyword **exception**, as in

exception MyException **of** string

Porting this model into Coq is difficult as it is not possible to define a type without simultaneously defining its constructors. Instead, we opted for a simple yet flexible approach. We define the type Exception as isomorphic to the unit type, and to distinguish each exception we create them as *opaque*, that is, irreducible. Figure 15 shows how to create two exceptions, the first one parameterized over a string. What is crucial is the sealing of the definition with **Qed**, signaling to Coq that this definition is opaque. The example test_ex illustrates the catching of different exceptions.

```
Definition MyException (s : string) : Exception.
  exact exception.
Qed.

Definition AnotherException : Exception.
  exact exception.
Qed.

Definition test_ex e :=
  mtry (raise e) with
    | AnotherException ⇒ ret ""
    | MyException "hello" ⇒ ret "world"
    | [s] MyException s ⇒ ret s
  end.
```

Figure 15. Exceptions in Mtac.

6. Related Work

Languages for Typechecked Tactics. In the last five years there has been increasing interest in languages that support safe tactics to manipulate proof terms of dependently typed logics. Delphin [23], Beluga [21, 22, 4], and VeriML [29, 30] are languages that, like Mtac, fall into this category. By "safe" we mean that, if the execution of a tactic terminates, then the resulting proof term has the type specified by the tactic.

But, unlike Mtac, these prior systems employ a strict separation of languages: the computational language (the language used to write tactics) is completely different from the logical language (the language of proofs), making the meta-theory heavier than in Mtac. Indeed, our proof of type safety is completely straightforward, as it inherits from CIC all the relevant properties such as type preservation under substitution. Having a simple meta-theory is particularly important to avoid precluding future language extensions—indeed, extensions of the previous systems have often required a reworking of their meta-theory [30, 4].

Another difference between these languages and Mtac is the logical language they support. For Delphin and Beluga it is LF [13], for VeriML it is λHOL [1], and for Mtac it is CIC [2]. CIC is the only one among these that provides support for computation at the term and type level, thereby enabling proofs by reflection (*e.g.*, see §3.3). Instead, in previous systems term reduction must be witnessed explicitly in proofs. To work around this, VeriML's computational language includes a construct letstatic that allows one to stage the execution of tactics, so as to enable equational reasoning at typechecking time. Then, proofs of (in-)equalities obtained from tactics can be directly injected in proof terms generated by tactics. This is similar to our use of **run** in the example from §3.3, with the caveat that letstatic cannot be used within definitions, as we did in the inner_prod example, but rather only inside tactics.

In Beluga and VeriML the representation of objects of the logic in the computational language is based on Contextual Modal Type Theory [20]. Therefore, every object is annotated with the context in which it is immersed. For instance, a term t depending only in the variable x is written in Beluga as $[x.\ t]$, and the typechecker enforces that t has only x free. In Mtac, it is only possible to perform this check dynamically, writing an Mtactic to inspect a term and rule out free variables not appearing in the set of allowed variables (the interested reader may find an example of this Mtactic in the Mtac distribution). On the other hand, the syntax of the language and the meta-theory required to account for contextual objects are significantly heavier than those of Mtac.

Delphin shares with Mtac the $\nu x : A$ binder from [26, 19]. In Delphin, the variable x introduced by this binder is distinguished with the type $A^{\#}$, in order to statically rule out offending terms like $\nu x : A.$ **ret** x. In Mtac, instead, this check gets performed dynamically. Yet again, we see a tension between the simplicity of the meta-theory and the static guarantees provided by the system. In Mtac we favor the former.

```
01 Structure tagged_heap := Tag {untag :> heap}.
02 Definition default_tag := Tag.
03 Definition ptr_tag := default_tag.
04 Canonical Structure union_tag h := ptr_tag h.
05
06 Structure form (s : list ptr) := Form {
07   heap_of :> tagged_heap;
08   _ : def heap_of → uniq s ∧
09         ∀ x. x ∈ s → x ∈ dom heap_of}.
10
11 Canonical Structure union_form s₁ s₂ h₁ h₂ :=
12   Form (s₁ ++ s₂) (union_tag (h₁ • h₂)) ...
13
14 Canonical Structure ptr_form A x (v : A) :=
15   Form [:: x] (ptr_tag (x ↦ v)) ...
16
17 Canonical Structure default_form h :=
18   Form [::] (default_tag h) ...
```

Figure 16. Scan tactic in lemma overloading style.

From all these systems, VeriML is the only system that provides ML-style references at the computational level. References are useful for writing efficient tactics. For instance, Stampoulis and Shao [29] first present a tautology prover similar to the one in §3.2, with a linear list lookup function. Then, they replace the list of hypotheses with a hash table to efficiently store and lookup hypotheses. In our implementation of Mtac, we have begun to look into this, and the interested reader can find in the Mtac distribution a similar optimized version of the tautology prover, but we have yet to work out its meta-theory.

Finally, a key difference between Mtac and all the aforementioned systems is the ability to program Mtactics interactively, as shown at the end of §3.1. None of the prior systems supports this.

Proof Automation Through Lemma Overloading. At heart, one of the key ideas of Mtac is to get tactic execution to be performed by Coq's type inference engine. In that sense, Mtac is closely related to (and indeed was inspired by) Gonthier *et al.*'s work on lemma overloading using canonical structures [11].

However, as explained in the introduction, whereas Mtac supports a functional style of programming, the style of programming imposed by lemma overloading is that of (dependently-typed) logic programming. For instance, Figure 16 shows the scan algorithm from §3.1 rewritten using canonical structures. Without going into detail, the structure (*a.k.a.* record) form in line 9 is the backbone of the tactic. The (tagged) heap heap_of is the input to the algorithm, and the list of pointers s and the (unnamed) axiom are the output of the algorithm. The "tagging" of the heap (lines 1 to 4) is required in order to specify an order in which the canonical instance declarations in lines 13 to 20 (much like type class instances in Haskell) should be considered during canonical instance resolution.

The reason for using a parameter of the structure (s) to represent one of the outputs of the algorithm is tricky to explain. More generally, knowing where to place the inputs and outputs of overloaded lemmas, and how to compose them together effectively, requires deep knowledge of the unification algorithm of Coq. In fact, the major technical contribution of Gonthier *et al.*'s paper [11] is the development of a set of common "design patterns" to help in dealing with these issues. For instance, in order to encode the noalias tactic as a composition of several overloaded lemmas, Gonthier *et al.* employ a rather sophisticated "parametrized tagging" pattern for reordering of unification subproblems.

In contrast, the Mtac encoding of noalias is entirely straightforward functional programming. Admittedly, the operational semantics of Mtac's **mmatch** construct is also tied to the unification algorithm of Coq, and the lack of a clear specification of this algorithm is an issue we hope to tackle in the near future. But, crucially, the high-level control flow of Mtactics is easy to understand *without* a detailed knowledge of Coq unification.

That said, there are some idioms that canonical structures support but Mtactics do not. In particular, their logic programming style makes them openly extensible (as with Haskell type classes, new instances can be added at any time), whereas Mtactics are closed to extension. It also enables them to be applied in both *backward* and *forward* reasoning, whereas Mtactics are unidirectional.

Simulable Monads. Claret *et al.* [7] present *Cybele*, a framework for building more flexible proofs by reflection in Coq. Like Mtac, it provides a monad to build effectful computations, although these effects are compiled and executed in OCaml. Upon success, the OCaml code creates a *prophecy* that is injected back into Coq to simulate the effects in pure CIC. On the one hand, since the effects Cybele supports must be replayable inside CIC, it does not provide meta-programming features like Mtac's **mmatch**, **nu**, **abs**, and **evar**, which we use heavily in our examples. On the other hand, for the kinds of effectful computations Cybele supports, the proof terms it generates ought to be smaller than those Mtac generates, since Cybele enforces the use of proof by reflection. The two systems thus offer complementary benefits, and can in principle be used in tandem.

Acknowledgments

We are deeply grateful to Chung-Kil Hur for suggesting the Mendler-style encoding of **mfix**, to Arnaud Spiwack for suggesting the use of telescopes for representing patterns, and to Georges Gonthier for the neat type class trick for delaying execution of Mtactics. We would also like to thank Jesper Bengtson and Jonas Jensen, who tested an earlier version of Mtac and gave useful feedback that helped us in polishing the implementation, as well as Nils Anders Danielsson and the anonymous reviewers for their very helpful comments. This research was partially supported by the Spanish MINECO projects TIN2010-20639 Paran10 and TIN2012-39391-C04-01 Strongsoft, AMAROUT grant PCOFUND-GA-2008-229599, and Ramon y Cajal grant RYC-2010-0743.

References

[1] H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of automated reasoning*. Elsevier, 2001.

[2] Y. Bertot, P. Castéran, G. Huet, and C. Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, 2004.

[3] S. Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, volume 1281 of *LNCS*. Springer, 1997.

[4] A. Cave and B. Pientka. Programming with binders and indexed data-types. In *POPL 2012*. ACM, 2012.

[5] A. Chlipala. Certified programming with dependent types. `http://adam.chlipala.net/cpdt`, 2008.

[6] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI 2011*. ACM, 2011.

[7] G. Claret, L. del Carmen González Huesca, Y. Régis-Gianas, and B. Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *ITP 2013*, volume 7998 of *LNCS*. Springer, 2013.

[8] G. Gonthier. Formal proof — the four-color theorem. *Notices of the AMS*, 55(11):1382–93, 2008.

[9] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP 2013*, volume 7998 of *LNCS*. Springer, 2013.

[10] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical report, INRIA, 2008.

[11] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, 2011.

[12] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *ICFP*, 2002.

[13] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, Jan. 1993.

[14] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *POPL 2013*. ACM, 2013.

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

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

[17] N. P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991.

[18] D. Miller. Unification of simply typed lamda-terms as logic programming. In *ICLP 1991*. MIT Press, 1991.

[19] A. Nanevski. Meta-programming with names and necessity. In *ICFP 2002*. ACM, 2002.

[20] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *ACM Trans. Comput. Logic*, 9(3), June 2008.

[21] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL 2008*. ACM, 2008.

[22] B. Pientka and J. Dunfield. Programming with proofs and explicit contexts. In *PPDP 2008*. ACM, 2008.

[23] A. Poswolsky and C. Schürmann. System description: Delphin – a functional programming language for deductive systems. *ENTCS*, 228:113–120, 2009.

[24] C. Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004.

[25] A. Saïbi. Typing algorithm in type theory with inheritance. In *POPL 1997*. ACM, 1997.

[26] C. Schürmann, A. Poswolsky, and J. Sarnat. The ∇-calculus. Functional programming with higher-order encodings. In *TLCA 2005*, volume 3461 of *LNCS*. Springer, 2005.

[27] M. Sozeau. Subset coercions in Coq. In *TYPES 2006*, volume 4502 of *LNCS*. Springer, 2007.

[28] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs 2008*, volume 5170 of *LNCS*. Springer, 2008.

[29] A. Stampoulis and Z. Shao. VeriML: Typed computation of logical terms inside a language with effects. In *ICFP 2010*. ACM, 2010.

[30] A. Stampoulis and Z. Shao. Static and user-extensible proof checking. In *POPL 2012*. ACM, 2012.

[31] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4*, 2012.

[32] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.