Under consideration for publication in J. Functional Programming

Mtac: A Monad for Typed Tactic Programming in Coq

BETA ZILIANI
Max Planck Institute for Software Systems (MPI-SWS)
beta@mpi-sws.org
and
DEREK DREYER
Max Planck Institute for Software Systems (MPI-SWS)
dreyer@mpi-sws.org
and
NEELAKANTAN R. KRISHNASWAMI
University of Birmingham
n.krishnaswami@cs.bham.ac.uk
and
ALEKSANDAR NANEVSKI
IMDEA Software Institute
aleks.nanevski@imdea.org
and
VIKTOR VAFEIADIS
Max Planck Institute for Software Systems (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 type-checker 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.

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 (Gonthier, 2008; Gonthier *et al.*, 2013b), verifying serious software systems (Klein *et al.*, 2010; Leroy, 2009; Ševčík

et al., 2013; Chlipala, 2011b), 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.*, Boutin (1997)). 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 (Poswolsky & Schürmann, 2009), Beluga (Pientka, 2008), and most recently VeriML (Stampoulis & Shao, 2010), 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* (Nanevski *et al.*, 2008a)—it is not clear how to incorporate them into established interactive theorem provers.

A rather different approach, proposed by Gonthier *et al.* (2013a) 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" (Saïbi, 1997) 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 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.* (2013a) 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 $\bigcirc\tau$ (read as "maybe τ") classifying Mtactics that—if they terminate successfully—will produce Coq terms of type τ. 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 $\bigcirc\tau$ 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 τ assuming its argument t is a tactic of type $\bigcirc\tau$. When (our instrumentation of) the Coq type inference engine encounters **run** t, it executes the tactic t. If that execution terminates, it will either

[1] In terms of expressivity, there are tradeoffs between the two styles—for further discussion, see Section 7.

```
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.
```

Fig. 1. Mtactic for searching in a list.

produce a term e of type τ (in which case Coq will rewrite **run** t to e) 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**.

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 $\bigcirc\tau$. 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.*, [] 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** (Stampoulis & Shao, 2010) and Beluga's **case** (Pientka, 2008).

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

$$\textbf{Lemma}\ \mathsf{z\_in\_xyz}\ (x\ y\ z : A) : z \in [x;y;z] := \textbf{run}\ (\mathsf{search}\ \_\ \_)$$

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 article, 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 soundness (§4).
- Explore some technical issues regarding the integration of Mtac into Coq (§5).
- Extend the language to support *stateful* Mtactics (§6).
- Compare with related work and discuss future work (§7).

The source code of the Mtac implementation and the examples can be downloaded from:

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

This article is an extended version of our ICFP 2013 paper (Ziliani *et al.*, 2013). The main differences from the conference version are the inclusion of: completely new material about stateful Mtactics (§6); a revised account of Mtac's operational semantics, in which (following Coq) unification variables are assigned contextual types (§4.1–4.2); details of the type soundness proof for Mtac (§4.3); extended comparison with related work (§7); and a number of mostly minor corrections. One more significant correction is to the operational semantics of the abs construct (the EABS rule in Figure 13, discussed at the end of §4.2), which was missing a key side condition in the original conference paper.

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 Coq with a monadic type constructor $\bigcirc\tau$, representing tactic computations returning results of type τ, along with suitable introduction and elimination forms for such computations. We define $\bigcirc : \mathsf{Type} \to \mathsf{Prop}$ as a normal inductive predicate in CIC—the Calculus of (co-)Inductive Constructions, which is the base logic of Coq (Bertot & Castéran, 2004)—with constructors that reflect our syntax for tactic programming, as 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

$$
\begin{array}{lcl}
\bigcirc & : & \mathsf{Type} \to \mathsf{Prop} \\
\textbf{ret} & : & \forall A.\ A \to \bigcirc A \\
\textbf{bind} & : & \forall A\ B.\ \bigcirc A \to (A \to \bigcirc B) \to \bigcirc B \\
\textbf{raise} & : & \forall A.\ \mathsf{Exception} \to \bigcirc A \\
\textbf{mtry} & : & \forall A.\ \bigcirc A \to (\mathsf{Exception} \to \bigcirc A) \to \bigcirc A \\
\textbf{mfix} & : & \forall A\ P.\ ((\forall x : A.\ \bigcirc (P\ x)) \to (\forall x : A.\ \bigcirc (P\ x))) \\
 & & \quad \to \forall x : A.\ \bigcirc (P\ x) \\
\textbf{mmatch} & : & \forall A\ P\ (t : A).\ \mathsf{list}\ (\mathsf{Patt}\ A\ P) \to \bigcirc (P\ t) \\
\textbf{print} & : & \forall s : \mathsf{string}.\ \bigcirc \mathsf{unit} \\
\textbf{nu} & : & \forall A\ B.\ (A \to \bigcirc B) \to \bigcirc B \\
\textbf{abs} & : & \forall A\ P\ x.\ P\ x \to \bigcirc (\forall y : A.\ P\ y) \\
\textbf{is\_var} & : & \forall A.\ A \to \bigcirc \mathsf{bool} \\
\textbf{evar} & : & \forall A.\ \bigcirc A \\
\textbf{is\_evar} & : & \forall A.\ A \to \bigcirc \mathsf{bool} \\
\mathsf{Patt} & : & \forall A\ (P : A \to \mathsf{Type}).\ \mathsf{Type} \\
\mathsf{Pbase} & : & \forall A\ P\ (p : A)\ (b : \bigcirc (P\ p)).\ \mathsf{Patt}\ A\ P \\
\mathsf{Ptele} & : & \forall A\ P\ C.\ (\forall x : C.\ \mathsf{Patt}\ A\ P) \to \mathsf{Patt}\ A\ P
\end{array}
$$

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

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 $\bigcirc\tau$, we replace the **mfix** constructor with a variant, **mfix**$'$, in "Mendler style" (Mendler, 1991; Hur *et al.*, 2013), *i.e.*, in which references to $\bigcirc$ are substituted with references to a *parameter* $\odot$:

$$
\begin{array}{l}
\textbf{mfix}' : \forall A\ P\ \odot.\ (\forall x : A.\ \odot (P\ x) \to \bigcirc (P\ x)) \to \\
\qquad ((\forall x{:}A.\ \odot (P\ x)) \to (\forall x{:}A.\ \odot (P\ x))) \to \forall x{:}A.\ \bigcirc (P\ x)
\end{array}
$$

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 $\bigcirc\tau$ belongs to an impredicative sort like Prop. (In particular, if $\bigcirc\tau$ 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 $\bigcirc\tau$ 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 $\bigcirc(\bigcirc\tau)$.

Now, on to 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 $\mathsf{Patt}\ A\ P$, which match objects x of type A and return results of type $P\ x$. Binding in the pattern matching clauses is represented as a *telescope*: $\mathsf{Pbase}\ \mathsf{p}\ \mathsf{b}$ describes a ground clause that matches the constant p and has body b, and $\mathsf{Ptele}(\lambda x.\ pc)$ adds the binder x to the pattern matching clause pc. So, for example, $\mathsf{Ptele}(\lambda x.\ \mathsf{Ptele}(\lambda y.\ \mathsf{Pbase}\ (x+y)\ b)$ 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, $\mathsf{Ptele}(\lambda x.\ \mathsf{Pbase}\ x\ b)$ 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 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:

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

where $\mathsf{Ptele}\ x_1 \cdots x_n\ p$ means $\mathsf{Ptele}(\lambda x_1.\ \cdots.\ \mathsf{Ptele}(\lambda x_n.\ p)\cdots)$. Both type annotations in the **mfix** and in the **mmatch** notation (in the latter, denoted $\mathbf{as}\ x\ \mathbf{return}\ \bigcirc\tau$) are optional and can be omitted, in which case the returning type is left to the type inference algorithm to infer. The **mfix** construct accepts up to 5 arguments.

Running Mtactics. Defining $\bigcirc$ as an inductive predicate means that terms of type $\bigcirc\tau$ 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, $\mathbf{run}\ t$, that takes an Mtactic t of type $\bigcirc\tau$ (for some τ), and runs it *at type-inference time* to return a term t' of type τ.

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

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 τ, and thus τ 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.* (2013a). 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 Gonthier *et al.* (2013a), 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:

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

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):

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

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

```
01 Record form h := Form {
02         seq_of :> list ptr;
03         axiom_of : def h → uniq seq_of
04                         ∧ ∀ x. x ∈ seq_of → x ∈ dom h }.
05
06 Definition scan :=
07   mfix f (h : heap) : ○(form h) :=
08     mmatch h with
09     | [x A (v:A)] x ↦ v ⇒ ret (Form [x] ...)
10     | [l r] l • r ⇒
11       rl ← f l;
12       rr ← f r;
13       ret (Form (seq_of rl ++ seq_of rr) ...)
14     | [h'] h' ⇒ ret (Form [] ...)
15     end.
```

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

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 directly automate the abovementioned algorithm: perform all the rearrangements necessary to bring the two pointers in the inequality to the front of the heap, and then apply noalias_manual. However, since such a tactic would end up rearranging the heap twice, it would be computationally expensive and generate big proof terms.

Instead, we pursue a solution analogous to the one given by Gonthier *et al.* (2013a), breaking the problem into two smaller Mtactics scan and search2, which are then 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. (In all cases, a corresponding proof component is returned as well. We omit the details of these proofs here, as they are entirely straightforward.)

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

```
01 Definition search2 x y :=
02   mfix f (s : list ptr) : ○(uniq s → x != y) :=
03     mmatch s with
04     | [s'] x :: s' ⇒ r ← search y s'; ret (foundx_pf x r)
05     | [s'] y :: s' ⇒ r ← search x s'; ret (foundy_pf y r)
06     | [z s'] z :: s' ⇒ r ← f s'; ret (foundz_pf z r)
07     | _ ⇒ raise NotFound
08     end.
```

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

heaps, which are really finite maps from pointers to values, $x \mapsto v$ and $l \bullet r$ are applications not of primitive datatype 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.

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:

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

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

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).

```
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)).
```

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

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:

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

As the two occurrences of $\bigcirc$ 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)
split; refine 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

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

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 != x_2 and x_2 != 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

```
01 Program Definition interactive_search2 x y :=
02   mfix f (s : list ptr) : ○(uniq s → x != y) :=
03     mmatch s with
04     | [s'] x :: s' ⇒ r ← search y s'; ret _
05     | [s'] y :: s' ⇒ r ← search x s'; ret _
06     | [z s'] z :: s' ⇒ r ← f s'; ret _
07     | _ ⇒ raise NotFound
08     end.
09 Next Obligation. ... Qed.
10 Next Obligation. ... Qed.
11 Next Obligation. ... Qed.
```

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

solved. In order to make this inference happen, we use the refine tactic instead of apply (see §5 for further discussion of this point).

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** (Sozeau, 2007), 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 (Stampoulis & Shao, 2010) and Chlipala's CPDT textbook (Chlipala, 2011a). 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.

```
01 Definition prop-tauto :=
02   mfix f (p : Prop) : ○p :=
03     mmatch p as p' return ○p' with
04     | True ⇒ ret I
05     | [p1 p2] p1 ∧ p2 ⇒
06         r1 ← f p1;
07         r2 ← f p2;
08         ret (conj r1 r2)
09     | [p1 p2] p1 ∨ p2 ⇒
10         mtry
11           r1 ← f p1; ret (or_introl r1)
12         with _ ⇒
13           r2 ← f p2; ret (or_intror r2)
14         end
15     | _ ⇒ raise NotFound
16     end.
```

Fig. 7. Mtactic for a simple propositional tautology prover.

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 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 $\mathsf{dyn} = \Sigma p : \mathsf{Prop}.\ p$. (In Coq, this is encoded as an inductive type with constructor Dyn p x, for any $x : p$.)

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. It 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

```
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.
```

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

```
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.
```

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

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 $p = p_1 \to p_2$. Intuitively, in order to prove $p_1 \to p_2$, 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

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

(where A and B are left implicit). It is similar to the operator with the same name in Nanevski (2002) and Schürmann *et al.* (2005). Operationally, $\nu x : \tau.\ f$ (which is notation for **nu** $(\lambda x : \tau.\ f)$) creates a parameter y with type τ, 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 $p_1 \to p_2$, we abstract y from r, using the constructor

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

(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 (Miller, 1991) to instantiate q with a faithful representation of the body. The following lines mirror 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. In case the reader wonders how X could have been instantiated during the recursive call, the answer

is that it could happen during the lookup of a hypothesis to solve the goal. For instance, consider the goal $\exists x.\ P\ x$, for some proposition P, under hypothesis $P\ 42$. After creating a meta-variable X as witness for the existential, the prover will try to solve the goal $P\ X$. At this point, the lookup function will try to unify $P\ X$ with any of the available hypotheses, and will find that $P\ 42$ is unifiable with $P\ X$ by instantiating X with 42.

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

$$\textbf{Definition}\ \mathsf{exmpl} : \forall P\ x.\ P\ x \to \exists y.\ P\ y := \textbf{run}\,(\mathsf{tauto}\ []\ \_).$$

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 related systems like VeriML (Stampoulis & Shao, 2010) and Beluga (Pientka, 2008), in which (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. (For further discussion of those systems, see Section 7.)

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.

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,

```
Program Definition compare (n1 n2 : nat) : ○(n1 ≤ n2) :=
  r1 ← to_ast [] 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.
```

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

which invokes compare to automatically construct a proof that $i+1 \leq \mathsf{vsize}\ v$ (equivalent to $i < \mathsf{vsize}\ v$).

```
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 $[\mathsf{ith}\ v_1\ 2] \times [\mathsf{ith}\ v_2\ 2]$, 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*[2] 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; it has to

[2] In the literature, this step is also known as *reification*.

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, as well as match against 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 $([\mathsf{Var}\ 0, \mathsf{Var}\ 1], 4)$. 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 the following* term *constructors. We use* _.1 *and* _.2 *to denote projections out of a pair.*

```
Definition syn_zero : term := ([], 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 c and a term t, and *interprets t* into a nat, as follows.

```
interp_vars (c : ctx) (t : list var) :=
  if t is (Var j) :: t' then
    if (vlook c j, interp c t') is (Some v, Some e)
```

```
      then Some (v + e) else None
  else Some 0.

interp (c : ctx) (t : term) :=
  if interp_vars c 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 in the context c) and summing the results. The lookup function vlook c j is omitted here, but it either returns Some j-th element of the context c, or None if c has fewer 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, to_ast is passed as input a variable context, and it extends this context with new variables encountered during reflection. To reflect n_1 in compare, to_ast starts with the empty context [], 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 c 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 c. The third field of ast is a proof formalizing the described properties of term_of and ctx_of.

As shown in Figure 11, the Mtactic to_ast takes the input variable context c and the number n to be reflected, and it 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 using the variable context that is returned from reflection of n_1 (similarly to how the top level of compare is implemented). 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 c (omitted here) searches for n in c and returns a ctx $\times$ nat pair. If n is found, the pair consists of the old context c and the position of n in c. If n is not found, the pair consists of a new context, in which n is cons-ed to c, 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 terms t_1 and t_2 and tries to determine if t_1 and t_2 syntactically represent two $\leq$-related 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.

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

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

Fig. 11. Mtactic for reflecting nat expressions.

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

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

Fig. 12. Algorithm for canceling common variables from terms.

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 n1 n2 (a1 : ast [] n1) (a2 : ast (ctx_of a1) n2) :
      cancel (term_of a1) (term_of a2) →
      n1 ≤ n2.
```

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 and Type Soundness

In this section we present the operational semantics and type soundness of Mtac. The presentation here differs from the one in the original conference version of this article (Ziliani *et al.*, 2013) in that here we give a more accurate treatment of Coq's unification variables and the "contextual types" assigned to them. After presenting the rules of the semantics (§4.1), we provide more motivation for the use of contextual types (§4.2), before giving details of the type soundness proof (§4.3). Along the way, we motivate by example a number of our design decisions.

4.1 Rules of the Semantics

First, some preliminaries. We write A, B, C for CIC type variables, τ for CIC types, ρ for CIC type *predicates* (functions returning types), e for CIC terms, f for CIC functions, and t for Mtactics, *i.e.*, CIC terms of type $\bigcirc\tau$ for some type τ. The operational semantics of Mtac defines the judgment form

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

where Γ is the typing context containing parameters and (let-bound) local definitions, and 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). The syntax for contexts is

$$\Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, x : \tau := e$$
$$\Sigma ::= \cdot \mid \Sigma, ?x : \tau[\Gamma] \mid \Sigma, ?x : \tau[\Gamma] := e$$

These contexts are needed for weak head reduction of CIC terms ($\Sigma;\Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow} e'$), but also for some of Mtac's constructs. The types of unification variables are annotated with a "local context", as in $\tau[\Gamma]$, in which Γ should bind a superset of the free variables of τ. The reason for having such local contexts will be discussed below (§4.2).

As is usual in Coq, we omit function arguments that can be inferred by the type inference engine (typically, the type arguments). In some cases, however, it is required, or clearer, to explicitly spell out all of the arguments, in which case we adopt another convention from Coq: prepending the @ symbol to the function being applied.

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)**.** $v \in \mathit{Values} ::= \mathbf{ret}\ e \mid \mathbf{raise}\ e$.

Figure 13 shows our operational semantics. The first rule (EREDUC) performs a CIC weak head reduction step. As mentioned, 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 (The Coq Development Team, 2012).

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

$$\frac{\Sigma;\Gamma \vdash t \rightsquigarrow (\Sigma';t')}{\Sigma;\Gamma \vdash \mathbf{bind}\ t\ f \rightsquigarrow (\Sigma';\mathbf{bind}\ t'\ f)}\ \textsc{EBindS} \qquad \frac{\Sigma;\Gamma \vdash t \rightsquigarrow (\Sigma';t')}{\Sigma;\Gamma \vdash \mathbf{mtry}\ t\ f \rightsquigarrow (\Sigma';\mathbf{mtry}\ t'\ f)}\ \textsc{ETryS}$$

$$\frac{}{\Sigma;\Gamma \vdash \mathbf{bind}\ (\mathbf{ret}\ e)\ f \rightsquigarrow (\Sigma; f\ e)}\ \textsc{EBindR} \qquad \frac{}{\Sigma;\Gamma \vdash @\mathbf{bind}\ \tau\ \tau'\ (@\mathbf{raise}_\tau\ e)\ \ f \rightsquigarrow (\Sigma; @\mathbf{raise}_{\tau'}\ e)}\ \textsc{EBindE}$$

$$\frac{}{\Sigma;\Gamma \vdash \mathbf{mtry}\ (\mathbf{ret}\ e)\ f \rightsquigarrow (\Sigma;\mathbf{ret}\ e)}\ \textsc{ETryR} \qquad \frac{}{\Sigma;\Gamma \vdash \mathbf{mtry}\ (\mathbf{raise}\ e)\ f \rightsquigarrow (\Sigma; f\ e)}\ \textsc{ETryE}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash ps_i \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ \mathsf{Ptele}\ (\overline{x:\tau})\ (\mathsf{Pbase}\ p\ b) \\ \forall k.\ \Gamma'_k = \Gamma, x_1:\tau_1,\ldots,x_{k-1}:\tau_{k-1} \\ \Sigma, \overline{?y:\tau[\Gamma']};\Gamma \vdash p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} \approx e \triangleright \Sigma' \\ \forall j < i.\ ps_j \text{ does not unify with } e\end{array}}{\Sigma;\Gamma \vdash \mathbf{mmatch}\ e\ ps \rightsquigarrow (\Sigma';\Sigma'(b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\}))}\ \textsc{EMmatch}$$

$$\frac{?x \notin \mathsf{dom}(\Sigma)}{\Sigma;\Gamma \vdash \mathbf{evar}_\tau \rightsquigarrow (\Sigma, ?x:\tau[\Gamma];\mathbf{ret}\ ?x[\mathrm{id}_\Gamma])}\ \textsc{EEvar}$$

$$\frac{\Sigma;\Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ ?x[\sigma] \quad (?x := \_) \notin \Sigma}{\Sigma;\Gamma \vdash \mathbf{is\_evar}\ e \rightsquigarrow (\Sigma;\mathbf{ret}\ \mathsf{true})}\ \textsc{EIsVarT} \qquad \frac{\Sigma;\Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ e' \quad e' \text{ not unif. variable}}{\Sigma;\Gamma \vdash \mathbf{is\_evar}\ e \rightsquigarrow (\Sigma;\mathbf{ret}\ \mathsf{false})}\ \textsc{EIsVarF}$$

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

$$\frac{\Sigma;\Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ x \quad \Gamma = \Gamma_1, x:\tau', \Gamma_2 \quad x \notin \mathsf{FV}(\Gamma_2, \rho)}{\Sigma;\Gamma \vdash @\mathbf{abs}\ \tau\ \rho\ e\ e' \rightsquigarrow (\Sigma;\mathbf{ret}\ (\lambda y.\ e'\{y/x\}))}\ \textsc{EAbs} \qquad \frac{\textit{(* print s to stdout *)}}{\Sigma;\Gamma \vdash \mathbf{print}\ s \rightsquigarrow (\Sigma;\mathbf{ret}\ \langle\rangle)}\ \textsc{EPrint}$$

Fig. 13. Operational small-step semantics.

It is important to note that the decision to evaluate lazily instead of strictly is deliberate. Weak head reduction preserves the structure of terms, which is crucial to avoid unnecessary reductions, and to produce smaller proof terms. Take for instance the search example in §1.1. The proof showing that x is in the list $[z;y;w]{+}{+}[x]$ consists of one application of the lemma in_or_app to the lemma in_eq. In contrast, the proof showing that x is in the list $[z;y;w;x]$ (the result of forcing the execution of $[z;y;w]{+}{+}[x]$) consists of three applications of the lemma in_cons to the lemma in_eq. Were we to use call-by-value reduction, it would force the reduction of the append and thus result in a larger proof term. Moreover, the expressivity of Mtac would be diminished, since reducible patterns would not be matched against in their expanded form.

The next seven rules, concerning the semantics of Mtac fixed points (EFix), **bind** (EBindS, EBindR, and EBindE), and **mtry** (ETryS, ETryR, and ETryE), are all quite standard.

The most complex rule is the subsequent one concerning pattern matching (EMmatch). It matches the term e with some pattern described in the list ps.[3] Each element ps_i of ps is

[3] In some cases, the return type of the pattern matching may depend on the parameters of the type being pattern matched (the **in** keyword in Coq's pattern matching). Mtac, for the moment, does not supply an equivalent keyword for **mmatch**, although it is possible to achieve the same results indirectly by pattern matching the parameters prior to matching the element.

a pair containing a pattern p and a body b, abstracted over a list of (dependent) variables $\overline{x : \tau}$. 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. Then, we replace each variable x in the pattern p with a corresponding unification variable $?y$ and proceed to unify the result with term e. For this, the context Σ is extended with the freshly created unification variables $\overline{?y}$. For each $?y_k$, the type τ_k is created within the context Γ'_k, which is the original context Γ extended with the variables appearing to the left of variable x_k in the list $\overline{x : \tau}$. 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). The unification variables appearing in the body of the pattern are substituted with their definitions, denoted as $\Sigma'(b')$, where b' is the body after substituting the pattern variables with the unification variables, *i.e.*, $b' = b\{?y_1[\mathrm{id}_{\Gamma'_1}]/x_1\}\cdots\{?y_n[\mathrm{id}_{\Gamma'_n}]/x_n\}$. Here, $[\mathrm{id}_{\Gamma'_k}]$ refers to the identity substitution for variables in Γ'_k; its use will be explained in detail in the discussion of Example 4 (§4.2 below). Finally, we require that patterns are tried in sequence, *i.e.*, that the scrutinee, e, 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* (Miller, 1991). Higher-order unification is in general undecidable, but Miller identified a decidable subset of problems, the so-called *pattern fragment*, where unification variables appear only in equations of the form $?f\ x_1\ \ldots\ x_n \approx e$, 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 $c\ e_1\ \ldots\ e_n \approx c\ e'_1\ \ldots\ e'_n$, where c is a constant, the unification algorithm tries to equate each $e_i \approx e'_i$. 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.[4]

Coming back to the rules, next is the rule for $\mathbf{evar}_\tau$ (EEVAR), which simply extends Σ with a fresh uninstantiated unification variable of the appropriate type. Note that it is always possible to solve a goal by returning a fresh unification variable. But a proof is only complete if it is closed and, therefore, before QED time, every unification variable has to be instantiated. The two following rules (EEISVART and EEISVARF) 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 (ENUS and ENUV) 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 rule EABS is for abstracting over parameters. If the first (explicit) argument of **abs** weak-head reduces to a parameter, then we abstract it from the second (explicit) argument of **abs**, thereby returning a function. In order to keep a sound system, we need to check that the return type and the local context do not depend on the variable being abstracted, as discussed below in Example 5.

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 in which 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).

Finally, the last rule (EPRINT) 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.

Example 3. *We show the trace of a simple example, in order to give the reader a feel for the operational semantics. In this example,* $\Gamma = \{h : \mathsf{nat}\}$.

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

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

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

[4] For those familiar with Ltac, there are several differences between our **mmatch** and Ltac's **match**. In particular, the latter (1) does not reduce terms during unification, (2) uses an unsound unification algorithm, unlike Mtac (§4.3), and (3) backtracks upon (any kind of) failure, making it more difficult to understand the flow of tactic execution.

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:

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

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

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

The rule EBINDS 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 $\mathsf{in\_eq}\ h\ []$. 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.

4.2 Unification Variables and Contextual Types

At the beginning of this section we mentioned that unification variables have *contextual types*, but we did not explain why. To motivate them, consider the following scenario. Suppose we define a function f as follows:

$$f := \lambda w : \mathsf{nat}.\ \textbf{run}\ (\textbf{evar}_{\mathsf{nat}})$$

When this function is elaborated, it will become $\lambda w : \mathsf{nat}.\ ?u[w/w]$ for some fresh unification variable $?u : \mathsf{nat}[w : \mathsf{nat}]$. The contextual type of $?u$ specifies that $?u$ may only be instantiated by a term with at most a single free type variable w of type nat, and the *suspended substitution* $[w/w]$ specifies how to transform such a term into one that is well-typed under the current context. (The substitution is the identity at first, because the current context and the context under which $?u$ was created are both $w : \mathsf{nat}$.)

Now suppose that we define

$$g := \lambda x\ y : \mathsf{nat}.\ f\ x \qquad h := \lambda z : \mathsf{nat}.\ f\ z$$

and then at some point we attempt to solve the following unification problem:

$$g \approx \lambda x\ y : \mathsf{nat}.\ x \tag{3}$$

Should this unification succeed, and if so, what should it instantiate $?u$ with? First, to solve the unification goal, Coq will attempt to unify $f\ x \approx x$, and then, after β-reducing $f\ x$,

to unify $?u[x/w] \approx x$. This is where the contextual type of $?u$ comes into play. If we did not have the contextual type (and suspended substitution) for $?u$, it would seem that the only solution for $?u$ is x, but that solution would not make any sense at the point where $?u$ appears in h, since x is not in scope there. Given the contextual information, however, Coq will correctly realize that $?u$ should be instantiated with w, not x. Under that instantiation, g will normalize to $\lambda x\ y : \mathsf{nat}.\ x$, and h will normalize to $\lambda z : \mathsf{nat}.\ z$.

In the rules in Figure 13, there are two places where contextual types and suspended substitutions are relevant: the rule for creation of unification variables (EEVAR) and the rule for pattern matching (EMMATCH). In EEVAR, the new unification variable $?x$ is created with contextual type $\tau[\Gamma]$, where Γ is a copy of the local context coming from the evaluation judgment. The returned value is $?x$ applied to the identity suspended substitution id_Γ, as seen in the above example. In EMMATCH, every pattern variable $?y_k$ is created with the contextual type $\tau_k[\Gamma'_k]$. Γ'_k is the context Γ extended with the prior variables in the telescope $x_1,\ldots,x_{k-1}$. In order to marry the context in the contextual type with the local context Γ, each unification variable $?y_k$ is applied, as before, with the identity suspended substitution.

There is another, more subtle use of contextual types in EMMATCH: the unification condition. In Mtac, the unification algorithm is in charge of instantiating unification variables. As we saw in the previous example, it cannot instantiate unification variables arbitrarily, as it may end up with an ill-typed meta-substitution. For this reason, it uses the suspended substitution to know how to instantiate the unification variable. Explaining the complex process of unification in a language like Coq is beyond the scope of this paper, but we can give an intuitive hint: when Coq faces a problem of the form

$$?u[y_1/x_1,\ldots,y_n/x_n] \approx e$$

where the $y_1,\ldots,y_n$ are all distinct, then this equation falls into the pattern fragment discussed above (Miller, 1991). In this case, the solution to the problem is to *invert* the substitution and apply it on the right hand side of the equation, in other words instantiating $?u$ with $e\{x_1/y_1,\ldots,x_n/y_n\}$ (assuming the free variables of e are in $\{y_1,\ldots,y_n\}$). In the example (3) above, for instance, at the point where Coq tries to unify $?u[x/w] \approx x$, the unification problem falls into the pattern fragment and (through inversion) can be solved by instantiating $?u$ with $x\{w/x\}$, that is, w.

Contextual types are useful for other reasons as well. In particular, the following examples show two different potential sources of unsoundness related to the **abs** constructor. Fortunately, thanks to contextual types, plus the restrictions enforced by the rule EABS, neither example presents any risk to the soundness of Mtac.

Example 4. *Example showing how contextual types preclude the assignment of different types to the same term.*

```
01 Definition abs_evar (A : Type) :=
02   X ← evar A;
03   f ← @abs Type (λ B. B) A X : ○(∀ B : Type. B);
04   ret (f nat, f bool).
```

In short, this example takes a type A and creates a unification variable X of type A. Then, it abstracts its type, creating the function f, which is then applied to the types nat and bool.

For readability, we annotated the type of f: $\forall B : \mathsf{Type}.\ B$, and therefore the returned terms $f\ \mathsf{nat}$ and $f\ \mathsf{bool}$ have type nat and bool, respectively. However, they both compute to X, the unification variable, so at first sight it looks like the same expression can be typed with different and incompatible types!

No need to panic here: contextual types solve the conundrum! Let's see what really happens when we execute the Mtactic. First, note that, in order for the **abs**traction in line 3 to succeed, the Mtactic should instantiate the argument A with a type variable. Otherwise, the computation will get *blocked*, as we are only allowed to abstract variables. So let's assume abs_evar is executed in a local context with only variable $A' : \mathsf{Type}$, which is then passed in as the argument. When executing line 2, the unification variable $?u$ is created with contextual type $A'[A' : \mathsf{Type}]$. Then, the variable X is bound to $?u[A'/A']$.

Next, abstracting the type A' from the unification variable results in

$$\lambda B : \mathsf{Type}.\ ?u[A'/A']\{B/A'\},$$

which after applying the substitution is equal to

$$\lambda B : \mathsf{Type}.\ ?u[B/A'].$$

Variable f is bound to this value, and therefore the value resulting from applying f to the different types is

$$(?u[\mathsf{nat}/A'], ?u[\mathsf{bool}/A']).$$

Since the type of the unification variable *depends on* A', the return type is $\mathsf{nat} \times \mathsf{bool}$. But the substitution distinguishes both terms in the pair, so even when they refer to the same unification variable, they're actually referring to different *interpretations* of the value carried by $?u$.

Now, observe that in order to instantiate the unification variable we require a term e such that, no matter what type τ is substituted for A', e must have exactly that type (τ). Of course, such a value does not exist, so $?u$ cannot be instantiated! More precisely, when faced with the unification problem

$$?u[\tau/A'] \approx e$$

for some term e, Coq's unification will not have a solution for $?u$, as it does not fall into the higher-order pattern fragment.

Example 5. *In this example we show why it is necessary to severely restrict occurrences of the variable being abstracted.*

In the previous example the variable being abstracted occurred only in the term (more precisely, in the substitution). If it instead occurred in the return type or in the context, then this could lead to another potential source of unsoundness. To see what can go wrong, let us consider the following example:

```
Definition abs_dep (A : Type) (x : A) :=
  X ← evar nat;
  @abs Type (λ _. nat) A X : ○(Type → nat).
```

After running the first line of the function, X is bound to the term $?u[A/A, x/x]$, where $?u$ is a fresh unification variable with (contextual) type $\mathtt{nat}[A : \mathsf{Type}, x : A]$. If we allow the abstraction of A in X, then we arrive at the ill-typed term

$$\lambda\ B.\ ?u[B/A, x/x]$$

Note that x should have as type the first component of the substitution, which is B, but it has type A instead. For this reason we impose the restriction that the abstracted variable (A in this case) must not occur anywhere in the context aside from the point where it is bound. It is not hard to see what goes wrong when the abstracted variable appears in the return type, so we prohibit this case as well.

It is worth noting that the original implementation of Mtac, described in the conference version of this article (Ziliani *et al.*, 2013), neglected to check these side conditions and was thus unsound.

4.3 Proof of Soundness

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 $\forall x.\ v$ and $x \in \mathsf{FV}(v)$.*
- *It is* **abs** *$e\ e'$ and $e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ e''$ and $(e'' : \_) \notin \Gamma$.*
- *It is* **@abs** *$\tau\ \rho\ e\ e'$ and $e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ x$ and $\Gamma = \Gamma_1, x : \tau, \Gamma_2$ and $x \in \mathsf{FV}(\Gamma_2, \rho)$.*
- *It is* **mmatch** *e ps and no pattern in ps unifies with e.*

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

Theorem 1 (Type soundness)**.** *If $\Sigma; \Gamma \vdash t : \bigcirc\tau$, then either t is a value, or t is blocked, or there exist t' and Σ' such that $\Sigma; \Gamma \vdash t \rightsquigarrow (\Sigma'; t')$ and $\Sigma'; \Gamma \vdash t' : \bigcirc\tau$.*

In order to prove it, we assume the following typing judgment for Coq terms:

$$\Sigma; \Gamma \vdash e : \tau$$

and also assume the following standard properties:

Postulate 1 (Convertibility)**.** *If $\Sigma; \Gamma \vdash e : \tau$ and $\Sigma; \Gamma \vdash \tau \equiv \tau'$ (τ and τ' are convertible up to CIC reduction) then*

$$\Sigma; \Gamma \vdash e : \tau'$$

Postulate 2 (Type preservation of reduction)**.** *If $\Sigma; \Gamma \vdash e : \tau$ and $\Sigma; \Gamma \vdash e \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ e'$ then*

$$\Sigma; \Gamma \vdash e' : \tau$$

Postulate 3 (Meta-substitution)**.** *If $\Sigma; \Gamma \vdash e : \tau$ then $\Sigma; \Gamma \vdash \Sigma(e) : \tau$.*

Postulate 4 (Substitution)**.** *If* $\Sigma;\Gamma_1,x:\tau',\Gamma_2 \vdash e:\tau$ *and* $\Sigma;\Gamma_1 \vdash e':\tau'$ *then*

$$\Sigma;\Gamma_1,\Gamma_2\{e'/x\} \vdash e\{e'/x\} : \tau\{e'/x\}$$

Postulate 5 (Strengthening)**.** *If* $\Sigma;\Gamma_1,x:\tau,\Gamma_2 \vdash e:\tau'$ *and* $x \notin \mathsf{FV}(\Gamma_2,e,\tau')$ *then*

$$\Sigma;\Gamma_1,\Gamma_2 \vdash e:\tau'$$

Postulate 6 (Weakening)**.** *If* $\Sigma;\Gamma_1,\Gamma_2 \vdash e:\tau'$ *and* $\Sigma;\Gamma_1 \vdash \tau : \mathsf{Type}$ *and* $x \notin \mathsf{dom}(\Gamma_1,\Gamma_2)$ *then*

$$\Sigma;\Gamma_1,x:\tau,\Gamma_2 \vdash e:\tau'$$

We also need the following relation between meta-contexts:

Definition 3 (Meta-context extension)**.** *We say that* Σ' *extends* Σ*, written* $\Sigma' \geq \Sigma$*, if the following conditions holds: (1)* Σ' *is well-defined, and (2) every meta-variable in* Σ *occurs also in* Σ' *with the same type and, in case it is an instantiated variable, with the same term. Formally:*

$$\begin{aligned}\Sigma' \geq \Sigma \triangleq\ & \vdash \Sigma' \\ & \wedge \forall ?x:\tau[\Gamma] \in \Sigma.\ ?x:\tau[\Gamma] \in \Sigma' \vee \exists e.\ ?x:\tau[\Gamma] := e \in \Sigma' \\ & \wedge \forall ?x:\tau[\Gamma] := e \in \Sigma.\ ?x:\tau[\Gamma] := e \in \Sigma'\end{aligned}$$

where $\vdash \Sigma'$ *means* Σ' *is well-formed under the implicit global environment. For details of this definition, we refer the reader to the appendix of Asperti* et al. *(2009).*

We postulate that meta-context extension does not alter typechecking:

Postulate 7 (Meta-weakening)**.** *If* $\Sigma;\Gamma \vdash e:\tau$ *and* $\Sigma' \geq \Sigma$*, then*

$$\Sigma';\Gamma \vdash e:\tau$$

From this postulate it follows in particular that if $\Sigma;\Gamma_1,\Gamma_2 \vdash e:\tau$ and $\Sigma;\Gamma_1 \vdash \tau' : \mathsf{Type}$ and $?x \notin \mathsf{dom}(\Sigma)$ then

$$\Sigma, ?x:\tau'[\Gamma_1];\Gamma_1,\Gamma_2 \vdash e:\tau$$

Finally, we require from unification the following:

Postulate 8 (Soundness of unification)**.** *If* $\Sigma;\Gamma \vdash e:\tau$ *and* $\Sigma;\Gamma \vdash e':\tau'$ *and* $\Sigma;\Gamma \vdash e \approx e' \triangleright \Sigma'$ *then*

$$\Sigma';\Gamma \vdash e \equiv e' \quad \textit{and} \quad \Sigma';\Gamma \vdash \tau \equiv \tau' \quad \textit{and} \quad \Sigma' \geq \Sigma$$

We start by proving type preservation:

Theorem 2 (Type preservation)**.** *If* $\Sigma;\Gamma \vdash t \rightsquigarrow (\Sigma';t')$ *and* $\Sigma;\Gamma \vdash t : \bigcirc\tau$*, then* $\Sigma' \geq \Sigma$ *and* $\Sigma';\Gamma \vdash t' : \bigcirc\tau$*.*

Proof. By induction on the reduction relation. We will expand (some of) the implicit parameters for clarity.

Case EREDUC: It follows by preservation of reduction (Postulate 2).

Case EFIX: We have $t = @\mathbf{mfix}\ \tau'\ \rho\ f\ t''$. Recall the type of the fixpoint:

$$\mathbf{mfix} : \forall A\ P.\ ((\forall x : A.\ \bigcirc(P\ x)) \to (\forall x : A.\ \bigcirc(P\ x))) \to \forall x : A.\ \bigcirc(P\ x)$$

By hypothesis we know t is well-typed, and as a consequence, t'' has type τ'. We have to show that

$$\Sigma;\Gamma \vdash f\ (@\mathbf{mfix}\ \tau'\ \rho\ f)\ t'' : \bigcirc(\rho\ t'')$$

It follows immediately from the types of the subterms:

$$f : (\forall x : \tau'.\ \bigcirc(\rho\ x)) \to (\forall x : \tau'.\ \bigcirc(\rho\ x))$$

and

$$\mathbf{mfix}\ f : \forall x : \tau'.\ \bigcirc(\rho\ x)$$

and $t'' : \tau'$.

Case EBINDS: We have $t = @\mathbf{bind}\ \tau'\ \tau''\ t''\ f$. By the premise of the rule, there exists t''' such that

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

Recall the type of **bind**:

$$\mathbf{bind} : \forall A\ B.\ \bigcirc A \to (A \to \bigcirc B) \to \bigcirc B$$

We have that t has type $\bigcirc\tau''$. We have to show that

$$\Sigma';\Gamma \vdash @\mathbf{bind}\ \tau'\ \tau''\ t'''\ f : \bigcirc\tau''$$

By the inductive hypothesis, we know that $\Sigma' \geq \Sigma$ and

$$\Sigma';\Gamma \vdash t''' : \bigcirc\tau'$$

We conclude by noting that, by Postulate 7 (Meta-weakening), the type of f in the new meta-context Σ' is unchanged.

Case EBINDR: We have $t = @\mathbf{bind}\ \tau'\ \tau''\ (\mathbf{ret}\ e)\ f$. We have to show that

$$\Sigma;\Gamma \vdash f\ e : \bigcirc\tau''$$

Immediate by type of f and e.

Case EBINDE: We have $t = @\mathbf{bind}\ \tau'\ \tau''\ (@\mathbf{raise}_{\tau'}\ e)\ f$. We have to show that

$$\Sigma;\Gamma \vdash @\mathbf{raise}_{\tau''}\ e : \bigcirc\tau''$$

Immediate by type of $@\mathbf{raise}_{\tau''}\ e$.

Cases ETRYS, ETRYR, ETRYE: Analogous to the previous cases.

Case EMMATCH: We have $t = @\mathbf{mmatch}\ \tau'\ \rho\ e\ ps$. The type of **mmatch** is

$$\mathbf{mmatch} : \forall A\ P\ (t : A).\ \mathsf{list}\ (\mathsf{Patt}\ A\ P) \to \bigcirc(P\ t)$$

By the premises of the rule:

$$\Sigma;\Gamma \vdash ps_i \overset{\mathsf{whd}}{\rightsquigarrow}{}^{*}\ \mathsf{Ptele}\ (\overline{x : \tau''})\ (\mathsf{Pbase}\ p\ b) \tag{4}$$

$$\Sigma, \overline{?y : \tau''[\Gamma']};\Gamma \vdash p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} \approx e \triangleright \Sigma' \tag{5}$$

$$\forall j < i.\ ps_j \text{ does not unify with } e \tag{6}$$

That is, reducing *ps* leads to a list whose *i*-th element is a *telescope* (see §2) abstracting variables $\overline{x : \tau''}$ from a pattern p and body b. This pattern, after replacing its abstraction with unification variables, is unifiable with term e, producing a new meta-context Σ'. Every unification variable $?y_k$ is created with type τ_k in a context resulting form extending the context Γ with the variables to the left of the telescope, that is

$$\Gamma'_k = \Gamma, x_1 : \tau_1, \ldots, x_{k-1} : \tau_{k-1}$$

We have to show that

$$\Sigma'; \Gamma \vdash \Sigma'(b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\}) : \bigcirc(\rho\ e)$$

Given the hypothesis that t is well-typed, we know that

$$\Sigma; \Gamma \vdash e : \tau' \tag{7}$$

$$\Sigma; \Gamma, \overline{x : \tau''} \vdash p : \tau' \tag{8}$$

$$\Sigma; \Gamma, \overline{x : \tau''} \vdash b : \bigcirc(\rho\ p) \tag{9}$$

Taking Equation 8 and by weakening of the meta-context we obtain

$$\Sigma, \overline{?y : \tau''[\Gamma']}; \Gamma, \overline{x : \tau''} \vdash p : \tau' \tag{10}$$

By Postulate 4 (Substitution), noting that the variables $\overline{x}$ cannot appear free in τ' or Γ,

$$\Sigma, \overline{?y : \tau''[\Gamma']}; \Gamma \vdash p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} : \tau' \tag{11}$$

Therefore, by Postulate 8 (soundness of unification) and equations 5, 7, and 11, we obtain that both sides of the unification are convertible under the new context Σ'.
Similarly as before, by Equation 9, weakening of meta-context, and substitution, we obtain

$$\Sigma'; \Gamma \vdash b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} : \bigcirc(\rho\ p\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\})$$

(noting that ρ actually does not have any x_k in its set of free variables).
By convertibility, this is equal to

$$\Sigma'; \Gamma \vdash b\{\overline{?y[\mathrm{id}_{\Gamma'}]/x}\} : \bigcirc(\rho\ e)$$

We conclude by Postulate 3 (Meta-substitution).

Case EEVAR: We have $t = \mathbf{evar}_\tau$. It is trivial:

$$\Sigma, ?x : \tau[\Gamma]; \Gamma \vdash \mathbf{ret}\ ?x[\mathrm{id}_\Gamma] : \bigcirc\tau$$

and the new meta-context is an extension of Σ.

Cases EISEVART and EISEVARF: Trivial, both return a boolean value.

Case ENUS: Desugaring and making parameters explicit, we have $t = @\mathbf{nu}\ \tau'\ \tau\ (\lambda x : \tau'.\ t'')$.
The type of **nu** is

$$\mathbf{nu} : \forall A\ B.\ (A \to \bigcirc B) \to \bigcirc B$$

Therefore,

$$\Sigma; \Gamma, x : \tau' \vdash t'' : \bigcirc\tau \tag{12}$$

By the premise of the rule, $\Sigma; \Gamma, x : \tau' \vdash t'' \rightsquigarrow (\Sigma'; t''')$. By the inductive hypothesis with Equation 12,

$$\Sigma'; \Gamma, x : \tau' \vdash t''' : \bigcirc\tau$$

and $\Sigma' \geq \Sigma$. Therefore

$$\Sigma'; \Gamma \vdash \lambda x : \tau'.\ t''' : \tau' \to \bigcirc\tau$$

which allows us to conclude that

$$\Sigma'; \Gamma \vdash \nu x : \tau'.\ t''' : \bigcirc\tau$$

Case ENUV: As in the previous case, we have $t = @\mathbf{nu}\ \tau'\ \tau\ (\lambda x : \tau'.\ v)$. We have to show that

$$\Sigma; \Gamma \vdash v : \bigcirc\tau$$

Since t is well-typed, we know that

$$\Sigma; \Gamma, x : \tau' \vdash v : \bigcirc\tau$$

By the premise of the rule, $x \notin \mathsf{FV}(v)$, and it cannot appear free in τ, so by strengthening (Postulate 5),

$$\Sigma; \Gamma \vdash v : \bigcirc\tau$$

Case EABS: We have $t = @\mathbf{abs}\ \tau'\ \rho\ e\ e'$. Recall the type of **abs**:

$$\mathbf{abs} : \forall A\ P\ x.\ P\ x \to \bigcirc(\forall y : A.\ P\ y)$$

We need to show that

$$\Sigma; \Gamma \vdash \mathbf{ret}\ (\lambda y : \tau'.\ e'\{y/x\}) : \bigcirc(\forall y : \tau'.\ \rho\ y)$$

where, by the premises of the rule, e reduces to x. By preservation of reduction (Postulate 2), we know that x has type τ', and therefore Γ is equivalent (*i.e.*, convertible) to $\Gamma_1, x : \tau', \Gamma_2$, for some context Γ_1 and Γ_2. Also, by the premises of the rule, $x \notin \mathsf{FV}(\Gamma_2, \rho)$.
By t being well-typed, we know

$$\Sigma; \Gamma \vdash e' : \rho\ e$$

which by convertibility is equivalent to (expanding Γ)

$$\Sigma; \Gamma_1, x : \tau', \Gamma_2 \vdash e' : \rho\ x$$

By weakening (Postulate 6),

$$\Sigma; \Gamma_1, x : \tau', \Gamma_2, y : \tau' \vdash e' : \rho\ x$$

and by Postulate 4 (Substitution), noting that by the premise of the rule x does not appear free in ρ nor in Γ_2,

$$\Sigma; \Gamma_1, \Gamma_2, y : \tau' \vdash e'\{y/x\} : \rho\ y$$

By weakening,

$$\Sigma; \Gamma, y : \tau' \vdash e'\{y/x\} : \rho\ y$$

We can conclude that

$$\Sigma; \Gamma \vdash \lambda y : \tau'.\ e'\{y/x\} : (\forall y : \tau'.\ \rho\ y)$$

which is precisely what we have to show.

Case EPRINT: Immediate.

□

As a corollary, we have the main theorem of this section:

Theorem 1 (Type soundness)**.** *If* $\Sigma;\Gamma \vdash t : \bigcirc\tau$*, then either* t *is a value, or* t *is blocked, or there exist* t' *and* Σ' *such that* $\Sigma;\Gamma \vdash t \rightsquigarrow (\Sigma';t')$ *and* $\Sigma';\Gamma \vdash t' : \bigcirc\tau$*.*

Proof. Since t is well-typed with type $\bigcirc\tau$, then the following two cases have to occur: either the head constructor of t is one of the constructors of $\bigcirc$—and it is fully applied—or it is another CIC construct (another constant, a let-binding, etc.).

In the first case we have further three sub-cases:

1. It is a value (of the form **ret** e or **raise** e).
2. It is *blocked*, that is, is of any of the following forms:
   - It is $\nu x.\ v$ and $x \in \mathsf{FV}(v)$.
   - It is **abs** $e\ e'$ and $\Sigma;\Gamma \vdash e \overset{\mathsf{whd}_*}{\rightsquigarrow}{}^* e''$ and $(e'' : \_) \notin \Gamma$.
   - It is **@abs** $\tau'\ \rho\ e\ e'$ and $e \overset{\mathsf{whd}_*}{\rightsquigarrow}{}^* x$ and $\Gamma = \Gamma_1, x : \tau', \Gamma_2$ and $x \in \mathsf{FV}(\Gamma_2, \rho)$.
   - It is **mmatch** $e\ ps$ and no pattern in ps unifies with e.
3. There exist a t' and Σ' such that $\Sigma;\Gamma \vdash t \rightsquigarrow (\Sigma';t')$.

In the first two cases there is nothing left to prove. In the last case we conclude by applying Theorem 2.

If t's head constant is another CIC construct, then it either reduces using the standard CIC reduction rules, in which case the proof follows by preservation of CIC reduction rules, or it does not reduce and hence it is blocked. □

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 (Sacerdoti Coen, 2004; Saïbi, 1997) takes a partially type-annotated term e, a local context Γ, a unification variable context Σ, and an optional expected type τ', and returns its type τ, and produces a fully annotated term e', and updated unification variable context Σ'.

$$\Sigma;\Gamma \vdash_{\tau'} e \hookrightarrow e' : \tau \triangleright \Sigma'$$

If an expected type τ', is provided, then the returned type τ will be convertible to it, possibly instantiating any unification variables appearing in both τ and τ'. 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 (Gonthier *et al.*, 2008) 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 is_true b having type Prop.
3. To perform canonical structure resolution. Ample examples of canonical structures can be found in Gonthier *et al.* (2013a).

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:

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

This rule first recursively elaborates the tactic body, while also unifying the return type τ of the tactic with the expected goal τ' (if present). This results in the refinement of t to a new term t', which is then executed. If execution terminates successfully returning a term e (which from Theorem 1 will have type τ), 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 τ of an Mtactic to be equivalent to the goal τ', but we did not stipulate in what situations the knowledge of τ' is available. Our examples so far assumed τ' 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 tactic refine. 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 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.) In contrast, the apply: tactic (note the colon!) from Ssreflect (Gonthier *et al.*, 2008) *does* support the behavior we desire—*i.e.*, it ensures that information about the goal is available when running Mtactics. Consequently, apply: is what we use in many of the accompanying source files.

One last point about tactics: Mtac is intended as a typed alternative to Ltac for developing custom automation routines, but 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

```
01 Class runner A (t : ○A) := { eval : A }.
02
03 Hint Extern 20 (runner ?t) ⇒
04       (exact (Build_runner t (run t)))
05       : typeclass_instances.
```

Fig. 14. Type class for delayed execution of Mtactics.

vm_compute enables dramatic efficiency gains for reflection-based proofs (Grégoire & Leroy, 2002), 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.

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:

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

Previously we solved this goal by applying the Mtactic F twice to the two subgoals x_1 != x_2 and x_2 != 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 (Sozeau & Oury, 2008). We exploit this functionality in Figure 14, by defining the class runner, which is parameterized over an Mtactic t with return type τ 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 t.

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:

$$\mathsf{rewrite}\ !(\mathsf{eval}\ (F\ \_\ \_))\,.$$

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

```
01 Definition MyException (s : string) : Exception.
02   exact exception.
03 Qed.
04
05 Definition AnotherException : Exception.
06   exact exception.
07 Qed.
08
09 Definition test_ex e :=
10   mtry (raise e) with
11     | AnotherException ⇒ ret ""
12     | MyException "hello" ⇒ ret "world"
13     | [s] MyException s ⇒ ret s
14   end.
```

Fig. 15. Exceptions in Mtac.

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.

Controlling the Size of Terms. Some tactics tend to generate unnecessarily big terms. Take for instance the to_ast Mtactic from Figure 11. In the case of an addition, this Mtactic constructs an instance of the ast structure using the values projected out from calling the Mtactic recursively. That means that the final proof term will contain several copies of a structure, one for each projector called on that structure. As a matter of fact, for a type that admits a term of size n we can end up constructing a term with size as big as n^2! For reducing the size of such proof terms, it is therefore critical to give the Mtac programmer the ability to force reduction steps to be taken in certain places.

Mtac is thus equipped with three different unit operators: **ret**, **retS**, **retW**. The first one we have already seen in the rest of the work; the other two are new and are used extensively in the source files. They both reduce the term prior to returning it: **retS** using the simplification strategy from the simpl tactic, and **retW** using the weak head reduction

strategy. As a result, **run**-ning the following terms produces different outputs:

$$\textbf{ret}\ (1+1) \rightsquigarrow 1+1$$
$$\textbf{retS}\ (1+1) \rightsquigarrow 2$$
$$\textbf{retW}\ (1+1) \rightsquigarrow \mathsf{S}\ (0+1)$$

To observe the utility of such reduction operators, suppose G is a context (in the sense of Section 3.3, that is, an element of type ctx). An intermediate step of the to_ast tactic, invoked with argument G, may result in a term such as

$$@\mathsf{term\_of}\ G\ 0\ (@\mathsf{Ast}\ G\ 0\ \mathsf{syn\_zero}\ G\ (\mathsf{zero\_pf}\ G)),$$

which has four copies of G in it. However, if this term is returned via **retS**, it will be reduced via simplification to

$$\mathsf{syn\_zero}.$$

For details on the reduction strategies see The Coq Development Team (2012).

6 Stateful Mtactics

So far, we have seen how Mtactics support programming with a variety of effects, including general recursion, syntactic pattern matching, exceptions, and unification, that are not available in the base logic of Coq. If, however, we compare these to the kinds of effects available in mainstream programming languages, we will immediately spot a huge omission: mutable state! This omission is costly: the lack of imperative data structures can in some cases preclude us from writing efficient Mtactics.

One example is the tautology prover from Section 3.2, whose running time can be asymptotically improved by using a hashtable. To see this, consider the complexity of proving the following class of tautologies:

$$p_1 \to \ldots \to p_n \to p_1 \wedge \ldots \wedge p_n$$

in a context where $p_1\ \ldots\ p_n : \mathsf{Prop}$. First, the tautology prover will perform n steps to add hypotheses $h_1 : p_1, \ldots, h_n : p_n$ to the context. Then, it will proceed to decompose the conjunctions and prove each p_i by searching for it in the context. Since the lookup Mtactic performs a linear search on the context, the overall execution will take $O(n^2)$ steps in total. Had we used a more clever functional data structure for implementing the context, such as a balanced tree, we could reduce the lookup time to logarithmic and the total running time to $O(n \log n)$. By using a hashtable to represent the context, however, we can do much better: we can achieve a constant-time lookup, thereby reducing the total proof search time to $O(n)$.

Rather than directly adding hashtables to Mtac, in this section, we show how to extend Mtac with two more primitive constructs: arrays and a hashing operator (§6.1). With these primitives, we implement a hashtable in Mtac (§6.2) and use it to implement a more efficient version of the tautology prover (§6.3), obtaining the expected significant speedup (§6.6).

The model of mutable state that we decided to incorporate to Mtac is very flexible and allows us to write complex imperative structures, but that comes at a price. First of

$\bigcirc$	:	$\mathsf{Type} \to \mathsf{Prop}$
...		
hash	:	$\forall A.\, A \to \mathsf{N} \to \bigcirc\mathsf{N}$
array_make	:	$\forall A.\, \mathsf{N} \to A \to \bigcirc(\mathsf{array}\ A)$
array_get	:	$\forall A.\, \mathsf{array}\ A \to \mathsf{N} \to \bigcirc A$
array_set	:	$\forall A.\, \mathsf{array}\ A \to \mathsf{N} \to A \to \bigcirc\mathsf{unit}$
array_length	:	$\forall A.\, \mathsf{array}\ A \to \mathsf{N}$

Fig. 16. The new array primitives of the $\bigcirc$ inductive type.

all, combining mutable state with parameters (*i.e.*, the **nu** and **abs** operators) and syntax inspection (*i.e.*, the **mmatch** operator) is tricky and requires special care in the design of the operational semantics (§6.4). Second, the interactive nature of a proof assistant like Coq enforces certain constraints (§6.5). Despite these difficulties, the language enhanced with mutable state remains sound (§6.7).

6.1 New Language Constructs

Figure 16 shows the new language constructs, where N stands for binary encoded natural numbers and array τ is an abstract type representing arrays of type τ (this type will be discussed later). The new constructs are explained next:

- **hash** *e n* takes term *e* and natural number *n* and returns the *hash* of *e* as a natural number between 0 and $n-1$, ensuring a fair distribution.
- **array_make** *n e* creates an array with *n* elements initialized to *e*.
- **array_get** *a i* returns the element stored in position *i* in the array *a*. If the position is out of bounds, it raises an exception.
- **array_set** *a i e* stores element *e* in position *i* in the array *a*. If the position is out of bounds, it raises an exception.
- **array_length** *a* returns the size of array *a*.[5]

For each of these constructs, the type parameter *A* is treated as an implicit argument.

In the rest of the section we will use *references*, which are arrays of length 1. We will use a notation resembling that of ML: ref *e* creates a reference of type Ref τ, where $e : \tau$, $!r$ returns the current contents of reference *r*, and $r ::= e$ updates *r* to *e*.

6.2 A Dependently Typed Hashtable in Mtac

The aforementioned primitives are enough to build stateful algorithms and data structures, in particular a dependently typed hashtable. We present the hashtable below, introducing first a required module:

[5] Note that this operation returns a pure natural number rather than a member of type $\bigcirc$N. This design choice is debatable, since it means that one *can* produce confusing results if one sets out to use arrays unhygienically. But ultimately it is not relevant for soundness because the length component in our representation of arrays (described below) plays no role in the their Mtac reduction rules (Figure 20).

```
Definition t A := array A.

Definition make {A} n (c : A) := array_make n c.
Definition length {A} (a : t A) := array_length a.
Definition get {A} (a : t A) i := array_get a i.
Definition set {A} (a : t A) i (c : A) := array_set a i c.

Definition iter {A} (a : t A) (f : N → A → ○unit) : ○unit := ...
Definition init {A} n (f : N → ○A) : ○(t A) := ...
Definition to_list {A} (a : t A) : ○(list A) := ...
Definition copy {A} (a b : t A) : ○unit := ...
```

Fig. 17. The Array module.

The Array **module.** Figure 17 presents the wrapper module Array, emulating the one in OCaml. Besides the four primitive operations on arrays (make, length, get, set), the module also provides an iterator iter, an array constructor with an initialization function init (which throws an exception if asked to create a 0-length array), a conversion function from array to list, and a copy function that copies the elements from the first array into the second one, if there is enough space, and fails otherwise. We omit the code for these, but they are standard and can be found in the source files.

The HashTbl **module.** Figure 18 presents the module HashTbl, which implements a (rudimentary) dependently typed hashtable. At a high level, given types τ and $\rho : \tau \to \mathsf{Type}$, a hashtable maps each key x of type τ to a term of type $\rho\ x$. More concretely, it consists of a pair of references, one containing the load of the hash (how many elements were added), and the other containing an array of "buckets". Each bucket, following the standard *open hashing* strategy for conflict resolution, is a list of Σ-types packing a key with its element. In open hashing, each bucket holds all the values in the table whose keys have the same hash, which is the index of that bucket in the array.

The hashtable is created with an initial size (initial_size) of 16 buckets, and every time it gets expanded it increases by a factor (inc_factor) of 2. The threshold to expand the hashtable is when its load reaches 70% of the buckets.

The function quick_add adds key x, mapped to element y, to the table. It does so by hashing x, obtaining a position i for a bucket l, and adding the existential package containing both x and y to the head of the list. Note that this function does not check whether the threshold has been reached and is thus not intended to be called from the outside—it is just used within the HashTbl module as an auxiliary function.

The function expand first creates a new array of buckets doubling the size of the original one, and then it iterates through the elements of the original array, adding them to the new array. We omit the code of the iterator function iter for brevity.

The function add first checks if the load exceeds the threshold and, if this is the case, proceeds to expand the table. It then adds the given key-element pair to the table. Finally, the count of the load is increased by one. Since the load is a binary natural, and for binary

```
Definition t A (P : A → Type) :=
  (Ref N × Ref (Array.t (list {x : A & P x}))).

Definition initial_size := 16.
Definition inc_factor := 2.
Definition threshold := 7.

Definition create A P : ○(t A P) :=
  n ← ref 0; a ← Array.make initial_size []; ra ← ref a;
  ret (n, ra).

Definition quick_add {A P} a (x : A) (y : P x) : ○unit :=
  let n := Array.length a in i ← hash x n; l ← Array.get a i;
  Array.set a i (existT _ x y :: l).

Definition iter {A P} (h : t A P) (f : ∀ x : A. P x → ○unit) : ○unit := ...

Definition expand {A P} (h : t A P) : ○unit :=
  let (_, ra) := h in a ← !ra; let size := Array.length a in
  let new_size := size × inc_factor in new_a ← Array.make new_size [];
  iter h (λ x y. quick_add new_a x y);; ra ::= new_a.

Definition add {A P} (h : t A P) (x : A) (y : P x) :=
  let (rl, ra) := h in load ← !rl; a ← !ra; let size := Array.length a in
  (if threshold × size ≤ 10 × load then expand h else ret tt);;
  a ← !ra; quick_add a x y;;
  new_load ← retS (N.succ load); rl ::= new_load.

Definition find {A P} (h : t A P) (x : A) : ○(P x) :=
  let (_, ra) := h in a ← !ra;
  let size := Array.length a in
  i ← hash x size; l ← Array.get a i;
  List.find x l.

Definition remove {A P} (h : t A P) (x : A) : ○unit :=
  let (rl, ra) := h in a ← !ra;
  let size := Array.length a in
  i ← hash x size; l ← Array.get a i;
  l' ← List.remove x l;
  Array.set a i l';;
  load ← !rl; new_load ← retS (N.pred load);
  rl ::= new_load.
```

Fig. 18. The HashTbl module.

naturals the successor (succ) operation is a function, we force the evaluation of succ *load* by simplifying it with **retS**.

The function find first obtains the bucket l corresponding to the hashed index i of the key x, and then performs a linear search using the function

$$\mathsf{List.find}\ (A : \mathsf{Type})\ (P : A \to \mathsf{Type})\ (x : A)\ (l : \mathsf{list}\ \{z : A\ \&\ P\ z\}) : \bigcirc(P\ x)$$

We omit the code of this function since it is similar to the lookup function from §3.2.

Similarly, the function remove removes an element from the hashtable by obtaining the bucket l of the key x and removing the element using the auxiliary function

$$\begin{aligned}&\mathsf{List.remove}\ (A : \mathsf{Type})\ (P : A \to \mathsf{Type})\ (x : A)\ (l : \mathsf{list}\ \{z : A\ \&\ P\ z\})\\ &\quad : \bigcirc(\mathsf{list}\ \{z : A\ \&\ P\ z\})\end{aligned}$$

This function throws the exception NotFound if the element is not in the list, and only removes one copy if the key x appears more than once.

6.3 The Tautology Prover Revisited

The code for the new tautology prover with hashing of hypotheses is listed in Figure 19. The type used to represent contexts is defined in the first line: it is a dependently typed hashtable whose keys are propositions and whose elements are proofs of those propositions. The prover itself begins on line 3. The cases for the trivial proposition, conjunction and disjunction (lines 5–17), as well as for $\forall$ (lines 23–26), are similar to the original ones, except that the context is not threaded through recursive calls but rather updated imperatively.

The implication, existential, and base cases require more drastic changes. The implication case is modified to extend the hashtable (line 20) with the mapping of parameter x with hypothesis p_1. In order to avoid leaving garbage in the hashtable, the added element is removed after the recursive call. Since the recursive call may raise an exception, the removal is performed before re-throwing the exception.

The base case (line 36) is modified to perform the lookup in the hashtable.

Finally, the existential case (line 27), requires a bit of explanation. It starts, as in the previous prover, by creating a unification variable for the witness X (line 28). Then it differs substantially. The reason why we need to change its behavior is simple: in the previous prover we were expecting it to find the solution for the witness by unification during the lookup in the base case. Since now we have a hash table for the lookup, there is no unification process going on, and therefore no instantiation for the witness can occur.

Here instead we do the following: we still try to find a solution recursively (line 30) for when the solution is trivial or does not depend on the witness—for instance, consider the case $\exists p : \mathsf{Prop}.\ p$ with trivial solution

$$\mathsf{ex\_intro}\ (\lambda p : \mathsf{Prop}.\ p)\ \mathsf{True}\ \mathsf{I}$$

If no solution is found (*i.e.*, an exception ProofNotFound is raised), then we create another unification variable for the proof r and return the proof term ex_intro $q\ X\ r$ (line 34). That is, we return a proof with a hole, expecting the proof developer to fill the hole later on. For instance, if we run the tautology prover on the example

$$\forall x : \mathsf{nat}.\ \exists y : \mathsf{nat}.\ y \leq x$$

then Coq will ask, as a subgoal, for a proof of $?X \leq x$. The proof developer can proceed to provide the standard proof of $0 \leq x$, thereby instantiating $?X$ with 0.

```
01 Definition ctx := HashTbl.t Prop (λ x. x).
02
03 Definition tautoh' (c : ctx) := mfix f (p : Prop) : ○p :=
04   mmatch p as p' return ○p' with
05   | True ⇒ ret I
06   | [p1 p2] p1 ∧ p2 ⇒
07       r1 ← f p1 ;
08       r2 ← f p2 ;
09       ret (conj r1 r2)
10   | [p1 p2] p1 ∨ p2 ⇒
11       mtry
12         r1 ← f p1 ;
13         ret (or_introl r1)
14       with _ ⇒
15         r2 ← f p2 ;
16         ret (or_intror r2)
17       end
18   | [p1 p2 : Prop] p1 → p2 ⇒
19       ν (x:p1).
20         HashTbl.add c p1 x;;
21         mtry r ← f p2; HashTbl.remove c p1;; abs x r
22         with [e] e ⇒ HashTbl.remove c p1;; raise e end
23   | [A (q:A → Prop)] (∀ x:A. q x) ⇒
24       ν (x:A).
25         r ← f (q x);
26         abs x r
27   | [A (q:A → Prop)] (∃ x : A. q x) ⇒
28       X ← evar A;
29       mtry
30         r ← f (q X);
31         ret (ex_intro q X r)
32       with ProofNotFound ⇒
33         r ← evar (q X);
34         ret (ex_intro q X r)
35       end
36   | [x] x ⇒ HashTbl.find c x
37   end.
38
39 Definition tautoh P :=
40   c ← HashTbl.create Prop (λ x. x);
41   tautoh' c P.
```

Fig. 19. Tautology prover with hashing of hypotheses.

6.4 Operational Semantics

As we mentioned in the introduction to this section, we need to take special care when combining mutable state with parameters and syntax inspection.

Mutable State and Parameters. The combination of these two features requires us to adjust the operational semantics for the **nu** (ν) operator in order to preserve soundness. More precisely, we need to ensure that if we store a parameter x in an array or reference, then we shouldn't be able to read from that location outside the scope of x. Take for instance the following example:

Definition wrong $:= r \leftarrow$ **ref** 0; $(\nu\ x{:}\mathsf{nat}.\ r ::= x)$;; $!r$.

In this code, first a reference r is created, then a new parameter x is created and assigned to r. Later, outside of the scope of x, r is dereferenced and returned. Without checking the context of the element being returned, the result of this computation would be undefined.

With unification variables we also have this problem—we can encode a similar example using **evar** instead of **ref** and **mmatch** instead of the assignment. But with unification variables, the contextual type of the unification variable would prevent us from performing the instantiation, therefore effectively ensuring that "nothing goes wrong". For mutable state, on the other hand, it would be too restrictive to restrict assignments to only include variables coming from the context where the array was created. Take for instance the tautology prover: there, in the implication case, the parameter x is added to a hashtable that was created outside the scope of x.

Thus, for mutable state we take a different, more "dynamic" approach: before returning from a νx binder we *invalidate* all the cells in the state that refer (in the term or in the type) to x. If later on a read is performed on any such cell, the system simply gets blocked.

Mutable State and Syntax Inspection. The problem with combining mutable state and the **mmatch** constructor concerns (lack of) abstraction. In short, we would like to be able to prevent the user from breaking abstraction and injecting a spurious reference (a location that is not in the domain of the store or with a different type from the one in the store) in the execution of a Mtactic, as that would result in a violation of type soundness. However, we have been unable to find any simple and elegant way of preventing the user from doing that, and so instead we choose to incur the cost of dynamic safety checks when performing stateful operations.

In order to understand the problem, let us first explain how it is handled in Haskell, and why we cannot simply port the same approach to Mtac. In Haskell, the **runST** command (Launchbury & Peyton Jones, 1994) enables one to escape the stateful ST monad by executing the stateful computation inside, much as with Mtac's **run** and $\bigcirc$ monad.

Since **runST** uses an effectful computation to produce a term of a pure Haskell type, it is essential to the soundness of **runST** that its computation runs inside a fresh store. Moreover, it is important to prevent the user from attempting to access an old reference within a new call to **runST**, as in the following code (here in Mtac syntax).

let $r :=$ **run** (**ref** 0) **in run** $!r$

After creating the reference r in the first **run**, the reference is read in the second. In Haskell, this situation is prevented by giving **runST** the following more restrictive type:

$$\forall A.\ (\forall S.\ \mathsf{ST}\ S\ A) \to A$$

where S is a phantom type parameter representing the state and A is the return type of the monad, which may not mention S. By making the monad parametric in the state type, the typechecker effectively ensures that the computation being executed can neither leak any references to nor depend on any references from its environment. For example, in the code above, the command **ref** 0 has type $\mathsf{ST}\ S\ (\mathsf{Ref}\ S\ \mathsf{nat})$ for some S, but this cannot be generalized to **runST**'s argument type $(\forall S.\ \mathsf{ST}\ S\ A)$ because $A = \mathsf{Ref}\ S\ \mathsf{nat}$ mentions S.

Crucially, Haskell's built-in abstraction mechanisms also hide the *constructor* for the Ref type. That is, in Haskell the user is not entitled to pattern match an element of the Ref type simply because its constructor is hidden from the user.

Unfortunately, the same techniques will not help us in Mtac. The problem arises from the following observation: for any constant c with type τ, if Mtac can create it, Mtac can inspect it. That is, we can always pattern match a term of type τ and get a hold of its head constant (*e.g.*, c). The same happens with references. Say that we create a new type $\mathsf{Ref}\ A$ with constructor

$$\mathsf{cref} : \forall A.\ \mathsf{Loc} \to \mathsf{Ref}\ A$$

Say further that we hide the constructor from the user (by using the module system of Coq), so that the proof developer is not entitled to write the following code:

$$\textbf{mmatch}\ x\ \textbf{with}\ [l : \mathsf{Loc}]\ \mathsf{cref}\ l\ \Rightarrow \ldots\ \textbf{end}$$

There is still a way of achieving the same behavior with the following code:

$$\textbf{mmatch}\ x\ \textbf{with}\ [(c : \mathsf{Loc} \to \mathsf{Ref}\ A)\ (l : \mathsf{Loc})]\ c\ l\ \Rightarrow \ldots\ \textbf{end}$$

since, if Mtac is (somehow) allowed to construct a cref, then nothing can prevent it from matching a meta-variable with a cref.

And this can be disastrous from the perspective of soundness. Imagine, for instance, that the location of a reference with type A is accessed at type B:

```
mmatch (x, y) with
| [(c1 : Loc → Ref A) (c2 : Loc → Ref B) l1 l2] (c1 l1, c2 l2) ⇒ !(c2 l1)
end
```

One option for solving this problem might be to forbid **mmatch** from pattern matching an element containing a reference. However, this solution is too strict, as it would disallow legitimate matches on terms of data types containing references.

Instead, the solution that we have adopted in Mtac is not to hide cref at all, but rather to bite the bullet and perform the necessary dynamic checks to ensure that the code above gets blocked. More precisely, under Mtac semantics, the read of $c_2\ l_1$ will get blocked as it is reading from the location l_1 at a different type (B) from the expected one (A). Similarly, in the previous example, the attempt to read the reference r in **run** $!r$ will get blocked since **run** $!r$ executes in a fresh store and r is not in the domain of that store.

With these considerations in mind, we can now move on to explain the actual rules. We extend the judgment from Section 4 to include input and output stores. A store σ is a map from locations l to arrays annotated with the type τ of their elements. An array element d

is either null or a Coq term:

$$\sigma ::= \cdot \mid l \mapsto [d;\ldots;d]_\tau, \sigma$$
$$d ::= \mathsf{null} \mid e$$

The new judgment is

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

On the Coq side, we have an inductive type of locations Loc and an inductive array type, where the number in carray represents the length of the array:

$$\begin{array}{lcl} \mathsf{array} & : & \mathsf{Type} \to \mathsf{Type} \\ \mathsf{carray} & : & \forall A.\ \mathsf{Loc} \to \mathsf{N} \to \mathsf{array}\ A \end{array}$$

Since we need to transform numbers and locations from Coq to ML and vice versa, we write $\overline{e}$ for the interpretation of Coq numbers/locations e into ML (implicitly reducing e to normal form) and $\underline{e}$ for the interpretation of ML numbers/locations e into Coq.

Figure 20 shows the new rules concerning stateful computations. The first rule (ENUV) presents the aforementioned modifications to the original rule for the νx binder. The changes (highlighted) show that, upon return of the νx binder, we invalidate all the arrays whose type contains x and all the array positions referring to x. The invalidate judgment is given in Figure 21.[6]

The second rule (EHASH) performs the **hash**-ing of a term e, limited by s. The element is hashed using an ML function hash, which we omit here.

The rule EAMAKE creates an n-element array initialized with element e. A fresh location pointing to the new array is appended to the state, and this location, together with n, are returned as part of the carray constructor.

The next two rules (EAGETR and EAGETE) describe the behavior of the getter. In both rules the array is first weak head-reduced in order to obtain the carray constructor applied to location l. Then the rules differ according to the case: if the index i is within the bounds of the array, and the element at index i is defined (*i.e.*, not null), then the type of the array τ is unified with the type coming from the carray constructor (τ'), potentially instantiating new meta-variables. If the index is outside the bounds of the array, an error OutOfBounds is raised.

The two rules for the setter (EASETR and EASETE) are similar: we first check that the index is within the bounds and that the type of the array unifies with the one from the constructor, and if so, the position i of the array is updated. Otherwise, an exception is raised.

[6] For performance reasons, in the implementation we don't traverse the entire state. Instead, we maintain a map relating each parameter x introduced by a νx binder to a list of array positions. When the parameter x goes out of scope, all the array positions associated with it are invalidated. The map is populated in each operation array_set $a\ i\ e$ in two steps: First, previous occurrences of the position (a,i) are eliminated from the map. Second, for each parameter x occurring free in e, the position (a,i) is added to the list mapped to x. In the current implementation, the removal of previous occurrences of (a,i) from the map is performed by a linear pass over the entire map, thus visiting each array position containing free parameters in its element. We note that (a) the map typically contains fewer elements than the entire state, and (b) this process can be optimized further with a better choice of data structures.

$$\frac{x \notin \mathsf{FV}(v) \quad \mathsf{invalidate}\ \sigma\ x = \sigma'}{\Sigma;\Gamma;\ \sigma\ \vdash (\nu x.\ v) \rightsquigarrow (\Sigma;\ \sigma'\ ; v)}\textsc{ENuV}$$

$$\frac{h = \mathsf{hash}(e) \mod \bar{s}}{\Sigma;\Gamma;\sigma \vdash \mathbf{hash}\ e\ s \rightsquigarrow (\Sigma;\sigma;\mathbf{ret}\ \underline{h})}\textsc{EHash}$$

$$\frac{l \text{ fresh}}{\Sigma;\Gamma;\sigma \vdash \mathbf{array\_make}\ \tau\ n\ e \rightsquigarrow (\Sigma; l \mapsto [e;\overset{n}{\ldots};e]_\tau, \sigma; \mathbf{ret}\ (@\mathsf{carray}\ \tau\ \underline{l}\ n))}\textsc{EAMake}$$

$$\frac{\Sigma;\Gamma \vdash a \overset{\mathsf{whd}}{\rightsquigarrow}{}^* @\mathsf{carray}\ \tau'\ l\ \_ \quad \bar{i} < n \quad e_{\bar{i}+1} \neq \mathsf{null} \quad \Sigma;\Gamma \vdash \tau \approx \tau' \triangleright \Sigma'}{\Sigma;\Gamma;\sigma,\bar{l} \mapsto [e_1,\ldots,e_n]_\tau \vdash \mathbf{array\_get}\ a\ i \rightsquigarrow (\Sigma';\sigma;\mathbf{ret}\ e_{\bar{i}+1})}\textsc{EAGetR}$$

$$\frac{\Sigma;\Gamma \vdash a \overset{\mathsf{whd}}{\rightsquigarrow}{}^* \mathsf{carray}\ l\ \_ \quad \bar{i} \geq n}{\Sigma;\Gamma;\sigma,\bar{l} \mapsto [e_1,\ldots,e_n]_\tau \vdash \mathbf{array\_get}\ a\ i \rightsquigarrow (\Sigma;\sigma;\mathbf{raise}\ \mathsf{OutOfBounds})}\textsc{EAGetE}$$

$$\frac{\Sigma;\Gamma \vdash a \overset{\mathsf{whd}}{\rightsquigarrow}{}^* @\mathsf{carray}\ \tau'\ l\ \_ \quad \bar{i} < n \quad \Sigma;\Gamma \vdash \tau \approx \tau' \triangleright \Sigma'}{\Sigma;\Gamma;\sigma,\bar{l} \mapsto [e_1,\ldots,e_n]_\tau \vdash \mathbf{array\_set}\ a\ i\ e \rightsquigarrow (\Sigma';\sigma,\bar{l} \mapsto [e_1,\overset{\bar{i}-1}{\ldots},e,\ldots,e_n]_\tau;\mathbf{ret}\ \langle\rangle)}\textsc{EASetR}$$

$$\frac{\Sigma;\Gamma \vdash a \overset{\mathsf{whd}}{\rightsquigarrow}{}^* \mathsf{carray}\ l\ \_ \quad \bar{i} \geq n}{\Sigma;\Gamma;\sigma,\bar{l} \mapsto [e_1,\ldots,e_n]_\tau \vdash \mathbf{array\_set}\ a\ i\ e \rightsquigarrow (\Sigma;\sigma;\mathbf{raise}\ \mathsf{OutOfBounds})}\textsc{EASetE}$$

Fig. 20. Operational small-step semantics of references.

$$\mathsf{invalarr}\ [e_1,\ldots,e_n]_\tau\ x = [e'_1,\ldots,e'_n]_\tau \qquad \forall i \in [1,n].\ e'_i = \begin{cases} \mathsf{null} & \text{if } x \in \mathsf{FV}(e) \\ e_i & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathsf{invalidate}\ []\ x &= [] \\
\mathsf{invalidate}\ (l \mapsto a_\tau, \sigma)\ x &= \mathsf{invalidate}\ \sigma\ x && x \in \mathsf{FV}(\tau) \\
\mathsf{invalidate}\ (l \mapsto a_\tau, \sigma)\ x &= (l \mapsto \mathsf{invalarr}\ a_\tau\ x, \mathsf{invalidate}\ \sigma\ x) && x \notin \mathsf{FV}(\tau)
\end{aligned}$$

Fig. 21. Invalidation of array positions whose contents are out of scope.

6.5 Use Once and Destroy

Allowing state to persist across multiple Mtactic **run**s is possible but technically challenging. For the present work, we have favored a simple implementation, and therefore restricted Mtactics to only use mutable state internally, as in Launchbury & Peyton Jones (1994).

If we were to consider generalizing to handle persistent state, we would encounter at least two key challenges. First, since Coq is interactive, the user is entitled to undo and redo operations. Allowing state to persist across multiple **run**s would thus require the ability to rollback the state accordingly. While this is doable using *persistent arrays* (Baker, 1991) with reasonable amortized space and time complexity, there is still a technical challenge: how to know to which state to rollback.

A second challenge arises from the module system of Coq. In particular, a developer may create references and modify them in different modules. Then, importing a module

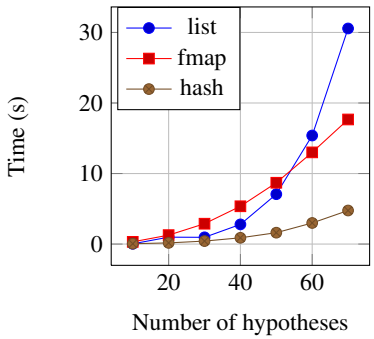

Fig. 22. Performance of three different implementations of tautology provers: using a list, a finite map over a balanced tree, and a hashtable.

from another file should replay the effects in that file in order to ensure a consistent view of the storage among different modules.

As mentioned above, we decided to leave this problem for future work and throw away the state after the execution of a Mtactic. While this decision may sound restrictive, it is flexible enough for us to be able to encode interesting examples, such as the one presented in this section.

The rule for elaborating the **run** operator is modified to start the evaluation of the Mtactic with an empty state:

$$\frac{\Sigma;\Gamma \vdash_{\bigcirc\tau'} t \hookrightarrow t' : \bigcirc\tau \triangleright \Sigma' \qquad \Sigma';\Gamma;\ [] \vdash t' \rightsquigarrow^* (\Sigma'';\sigma;\mathbf{ret}\ e)}{\Sigma;\Gamma \vdash_{\tau'} \mathbf{run}\ t \hookrightarrow e : \tau \triangleright \Sigma''}$$

6.6 A Word on Performance

Despite the various dynamic checks performed by our stateful extensions to Mtactics, we can obtain a significant speedup by using mutable state. To support our claim, we implemented three different versions of the tautology prover, using different data structures (both functional and imperative). Figure 22 shows the time it takes for each prover to solve tautologies on an increasing number of hypotheses. The slowest one is the functional prover from Section 3.2, which takes constant time for extending the context and linear time (in the size of the context) for lookup.

The second prover uses a functional map implemented with a balanced tree. As keys for the map we use the natural numbers coming from hashing the hypotheses, similarly to what the prover from Figure 19 does. Both extending and querying the context take logarithmic time to compute.

Finally, the third prover is the one presented in this section, which performs both extension and lookup of the context in amortized constant time. As expected, it greatly outperforms the other two.

Before moving on to prove soundness of the system, we want to stress that the safety check done in the getter and setter rules (*i.e.*, the unification of the type of the array with the one coming from the carray constructor) does not result in a significant performance cost. In particular, it is possible to cache the results, effectively avoiding the overhead

in the most common cases (which cover almost all of the accesses). Even in our rather naive implementation, our experimental results shows that the invalidation of cells and the unification of types in the getter and setter operations causes only minor slowdowns (less than 5%).

6.7 Soundness in the Presence of State

The soundness theorem must now be extended to consider the store. We start by extending the definition of blocked terms to consider the new cases:

Definition 4 (Blocked terms)**.** *A term t is* blocked *if and only if the subterm in reduction position satisfies one of the cases of Definition 2, or:*

- It is an array operation (get or set), and the array or the index do not normalize to the array constructor or a natural number, respectively.
- It is an array operation (get or set), the array and the index normalize to $@\mathsf{carray}\ \tau\ l\ \_$ and i respectively, for some τ, l and i, but either
  1. l is not in the state σ, or
  2. there exists $l \mapsto a_{\tau'} \in \sigma$, but τ does not unify with τ' or $a_i = \mathsf{null}$.

To state the preservation theorem, we need to say what it means for a store to be *valid*. A store σ is *valid* in contexts Γ and Σ if for every array a with element type τ in σ, every element of a is either null or has type τ. Formally,

$$\frac{}{\Sigma;\Gamma \vdash \cdot\ \mathsf{valid}} \qquad \frac{\forall j \in [1..n].\ e_j = \mathsf{null} \ \vee\ \Sigma;\Gamma \vdash e_j : \tau \qquad \Sigma;\Gamma \vdash \sigma\ \mathsf{valid}}{\Sigma;\Gamma \vdash (l \mapsto [e_1,\ldots,e_n]_\tau, \sigma)\ \mathsf{valid}}$$

We now extend the proof of type preservation, where we need to also ensure preservation of the validity of the store.

Theorem 3 (Type preservation with references)**.** *If* $\Sigma;\Gamma \vdash t : \bigcirc\tau$ *and* $\Sigma;\Gamma \vdash \sigma\ \mathsf{valid}$ *and* $\Sigma;\Gamma;\sigma \vdash t \rightsquigarrow (\Sigma';\sigma';t')$*, then* $\Sigma' \geq \Sigma$ *and* $\Sigma';\Gamma \vdash t' : \bigcirc\tau$ *and* $\Sigma';\Gamma \vdash \sigma'\ \mathsf{valid}$.

Proof. The proof poses little challenge. We only consider the new or modified cases.

Case ENUV: We have $t = \nu x.\ v$. Since x goes out of scope in the returning term, in order to have a valid state we need to remove all the arrays with types referring to x, and invalidate all array positions with contents referring to x. This is precisely what the invalidate judgment does.

Case EHASH: Trivial, as it is returning a number.

Case EAMAKE: We have $t = \mathsf{array\_make}\ \tau'\ n\ e$. Let l be a fresh location. By hypothesis we know that e has type τ', so the new store $\sigma' = l \mapsto [e;\overset{n-2}{\ldots};e]_{\tau'}, \sigma$ contains an array of elements of type τ'. Therefore, the new state is valid, and the returned value $@\mathsf{carray}\ \tau'\ l\ n$ has type $\mathsf{array}\ \tau'$.

Case EAGETR: We have $t = \mathsf{array\_get}\ \tau'\ a\ i$. By the premise of the rule, a normalizes to $@\mathsf{carray}\ \tau'\ l\ \_$ and i to i' such that $l \mapsto [e_1,\ldots,e_n]_\tau \in \sigma$ for some $e_1,\ldots,e_n$. Also by hypothesis, $i' < n$, τ unifies with τ', and $e_{i'+1} \neq \mathsf{null}$. Therefore, $e_{i'+1}$ has type (convertible with) τ'. Note that, by soundness of unification (Postulate 8), $\Sigma' \geq \Sigma$.

Case EAGETE: Trivial.

Case EASETR: We have $t = \textsf{array\_set}\ \tau'\ a\ i\ e$. As in the EAGETR case, we also have that $a \overset{\mathsf{whd}*}{\rightsquigarrow}$ @carray $\tau'\ l\ \_$ and there exist $e_1,\ldots,e_n$ such that $l \mapsto [e_1,\ldots,e_n]_\tau \in \sigma$. We need to show that $\Sigma;\Gamma \vdash \sigma'$ valid, where $\sigma' = l \mapsto [e_1,\overset{i-1}{\ldots},e,\ldots,e_n]_\tau,\sigma$. By hypothesis and the premises of the rule, we know that e has type τ', unifiable with τ. Therefore, updating the i-th position cannot invalidate the store. Note that, as in the previous case, by soundness of unification (Postulate 8), $\Sigma' \geq \Sigma$.

Case EASETE: Trivial.

□

As before, our main theorem follows as an immediate corollary:

Theorem 4 (Type soundness with references)**.** *If* $\Sigma;\Gamma \vdash t : \bigcirc\tau$ *and* $\Sigma;\Gamma \vdash \sigma$ valid*, then either* t *is a value, or* t *is blocked, or there exist* t'*,* Σ' *and* σ' *such that* $\Sigma;\Gamma;\sigma \vdash t \rightsquigarrow (\Sigma';\sigma';t')$ *and* $\Sigma';\Gamma \vdash t' : \bigcirc\tau$ *and* $\Sigma';\Gamma \vdash \sigma'$ valid*.*

As a second corollary, we have that the new rule for **run** t (in §6.5) is also sound: since the initial empty store under which t is run is trivially valid, Theorem 4 tells us that the type τ of t is preserved, and hence that the returned term e has type τ.

7 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 (Poswolsky & Schürmann, 2009), Beluga (Pientka, 2008; Pientka & Dunfield, 2008; Cave & Pientka, 2012), and VeriML (Stampoulis & Shao, 2010, 2012) 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 soundness 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 (Stampoulis & Shao, 2012; Cave & Pientka, 2012).

Another difference between these languages and Mtac is the logical language they support. For Delphin and Beluga it is LF (Harper *et al.*, 1993), for VeriML it is λHOL (Barendregt & Geuvers, 2001), and for Mtac it is CIC (Bertot & Castéran, 2004). 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 (Nanevski *et al.*, 2008a).[7] Therefore, every object is annotated with the context in which it is immersed. For instance, a term t depending only on 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 (Schürmann *et al.*, 2005; Nanevski, 2002). 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.

Of all these systems, VeriML is the only one that provides ML-style references at the computational level. Our addition of mutable state to Mtac is clearly inspired by the work of Stampoulis & Shao (2010), although, as we do not work with Contextual Modal Type Theory, we are able to keep the meta-theory of references quite simple.

Beluga's typechecker is constantly growing in complexity in order to statically verify the completeness and correctness of tactics (through *coverage* and termination checking). If a tactic is proved to cover all possible shapes of the inspected object, and to be terminating, then there is no reason to execute it: it is itself a meta-theorem one can trust. This concept, also discussed below, represents an interesting area of future research for Mtac.

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 support 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 (Gonthier *et al.*, 2013a).

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 23 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

[7] The contextual types of CMTT are not to be confused with the lightweight "contextual types" that Mtac assigns to unification variables (§4.2). In Mtac, we only use contextual types to ensure soundness of unification, inheriting the mechanism from Coq. Mtac's contextual types are essentially hidden from the user, whereas in VeriML and Beluga they are explicit in the computational language.

```
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) ...
```

Fig. 23. Scan tactic in lemma overloading style.

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.* (2013a) 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 "parameterized 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.

Typechecked Tactics Through Reflection. There is a large, mostly theoretical, body of work on using the theory of a proof assistant to reason about tactics written for the same proof assistant. The high-level idea is to *reflect* (a fraction of) the object language of the proof assistant into a Term datatype inside the same proof assistant. Tactics are

then constructed using this datatype, and can be verified just like any other procedure built inside the proof assistant. If the tactic is proven to be correct, then it can be safely used as an axiom, without having to spend time executing it, or checking its result.

While this idea is appealing, the circularity that comes from reasoning about the logic of a proof assistant within itself endangers the soundness of the logic, and therefore special care must be taken. In theory, one can avoid this circularity by restricting the objects of the language that can be reflected, and by establishing a hierarchy of assistants: an assistant at level k can reason about tactics for the assistant at level $k-1$ or below.[8] This concept is discussed in depth theoretically in Allen *et al.* (1990); Howe (1992); Constable (1992); Pollack (1995); Harrison (1995); Artëmov (1999). More recently, Devriese & Piessens (2013) provide, to the best of our knowledge, the first attempt of implementing it, but without arriving yet at a practical and sound implementation.

More pragmatically, the programming language Agda suports a lightweight implementation of reflection.[9] It does so with two primitives, **quoteGoal** and **unquoteGoal**. The first one reflects the current goal into a Term datatype, and the second one solves the current goal by the proof term that results from interpreting a term of the same datatype. This mechanism avoids the circularity problem mentioned before in two ways: first, as mentioned above, by restricting the object language supported and, second, by dynamically typechecking the proof term before solving a goal with it. That is, like in Mtac, a tactic is never trusted, and therefore cannot be used as an axiom. An example of proof automation using Agda's reflection mechanism, and its limitations, can be found in van der Walt & Swierstra (2013).

All of the aforementioned works require tactics to be written using a de Bruijn encoding of objects, in contrast to the direct style promoted in Mtac. In Agda this can be annoying, but not fatal. However, if tactics have to be proven correct, as in Devriese & Piessens (2013), then the overhead of verifying a tactic quickly negates the benefit of using it. Another disadvantage is that tactics are restricted to use the pure language of the assistant, and therefore cannot use effects like non-termination and mutable state.

Recent work by Malecha *et al.* (2014) restricts the use of reflection to reflective *hints*. Hints are lemmas reflected into an inductive datatype, similar to what the reflection mechanism of Agda does, packed together with a proof of soundness. These hints are then used by a specialized auto tactic that reflects the goal and tries to prove it automatically using the reflective hints. This work shares with the reflection mechanism of Agda the de Bruijn encoding of terms, but, unlike in Devriese & Piessens (2013), the soundness proof is local to the hint: the auto tactic will be in charge of performing the recursion, so the effort required to verify a hint is significantly smaller. Unlike Mtac, this work does not aim at a full language for meta-programming, although there are ongoing efforts to extend it into a general purpose tactic language, Rtac (Malecha & Bengtson, 2015).

Simulable Monads. Claret *et al.* (2013) present *Cybele* , a framework for building more flexible proofs by reflection in Coq. Like Mtac, it provides a monad to build effectful

[8] This is reminiscent of the universe level hierarchy in type theory.

[9] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Reflection`

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.

Effectful Computations in Coq. There is a large body of work incorporating or modeling effectful computations in Coq. Concerning the former, in Armand *et al.* (2010) the kernel of Coq is extended to handle machine integers and *persistent arrays* (Baker, 1991). As discussed in Section 6, in Mtac we favored the more generic reference model à la ML, and we did not need to modify the kernel. That said, if the reduction of terms in Armand *et al.* (2010) were extended to handle terms containing unification variables, the developer could potentially choose between our interpreted implementation of generic arrays and their natively implemented persistent arrays.

Concerning the latter, there are two different approaches when it comes to modeling state in Coq. The more traditional one, exemplified by Nanevski *et al.* (2008c,b, 2010) and Miculan & Paviotti (2012), is to encapsulate the effectful computations in a monad, and provide two different views of it. Inside the prover, this monad performs the computation in an inefficient functional (state-passing) way: easy to verify, but slow to execute. Then, the verified code is extracted into a programming language with imperative features (typically Haskell or OCaml) using the efficient model of imperative computation that these languages provide.

Vafeiadis (2013) argues that the monadic style of programming imposed by the previous approach is inconvenient, as it drags all functions that use stateful operations into the monadic tarpit, from which they cannot escape even if they are observably pure. He proposes an alternative called *adjustable references*, to enable the convenient use of references within code that is not observably stateful. An adjustable reference is like an ML reference cell with the addition of an invariant ensuring that updates to the cell are not observable. As with the previous approach, the code written in the prover is extracted into efficient OCaml code.

In contrast to the above approaches, stateful Mtactics do not offer any formal model for reasoning about code that uses references. State is restricted to the language of Mtactics, whose operational behavior cannot be reasoned about within Coq itself.

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, Scott Kilpatrick, Antonis Stampoulis, and the anonymous

reviewers of this article (and of the original conference version) 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

Allen, Stuart F., Constable, Robert L., Howe, Douglas J., & Aitken, William E. (1990). The semantics of reflected proof. *IEEE Symposium on Logic in Computer Science (LICS)*.

Armand, Michaël, Grégoire, Benjamin, Spiwack, Arnaud, & Théry, Laurent. (2010). Extending Coq with imperative features and its application to SAT verification. *International Conference on Interactive Theorem Proving (ITP)*.

Artëmov, Sergei N. (1999). On explicit reflection in theorem proving and formal verification. *Conference on Automated Deduction (CADE)*.

Asperti, A., Ricciotti, W., Sacerdoti Coen, C., & Tassi, E. (2009). A compact kernel for the calculus of inductive constructions. *Sadhana*, **34**(1), 71–144.

Baker, Henry G. (1991). Shallow binding makes functional arrays fast. *SIGPLAN Notices*, **26**(8), 145–147.

Barendregt, Henk, & Geuvers, Herman. (2001). Proof-assistants using dependent type systems. *Pages 1149–1238 of:* Robinson, Alan, & Voronkov, Andrei (eds), *Handbook of automated reasoning*. Elsevier Science Publishers B. V.

Bertot, Yves, & Castéran, Pierre. (2004). *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Boutin, Samuel. (1997). Using reflection to build efficient and certified decision procedures. *International Symposium on Theoretical Aspects of Computer Software (TACS)*.

Cave, Andrew, & Pientka, Brigitte. (2012). Programming with binders and indexed data-types. *ACM Symposium on Principles of Programming Languages (POPL)*.

Chlipala, Adam. (2011a). *Certified programming with dependent types*. MIT Press. `http://adam.chlipala.net/cpdt/`.

Chlipala, Adam. (2011b). Mostly-automated verification of low-level programs in computational separation logic. *Pages 234–245 of: ACM Conference on Programming Languages Design and Implementation (PLDI)*.

Claret, Guillaume, del Carmen González Huesca, Lourdes, Régis-Gianas, Yann, & Ziliani, Beta. (2013). Lightweight proof by reflection using a posteriori simulation of effectful computation. *International Conference on Interactive Theorem Proving (ITP)*.

Constable, Robert L. (1992). Metalevel programming in constructive type theory. *Pages 45–93 of:* Broy, Manfred (ed), *Programming and mathematical method*. NATO ASI Series, vol. 88. Springer-Verlag New York, Inc.

Devriese, Dominique, & Piessens, Frank. (2013). Typed syntactic meta-programming. *International Conference on Functional Programming (ICFP)*.

Gonthier, Georges. (2008). Formal proof — the four-color theorem. *Notices of the AMS*, **55**(11), 1382–93.

Gonthier, Georges, Mahboubi, Assia, & Tassi, Enrico. (2008). *A small scale reflection extension for the Coq system*. Tech. rept. INRIA.

Gonthier, Georges, Ziliani, Beta, Nanevski, Aleksandar, & Dreyer, Derek. (2013a). How to make ad hoc proof automation less ad hoc. *Journal of Functional Programming (JFP)*, **23**(4), 357–401.

Gonthier, Georges, Asperti, Andrea, Avigad, Jeremy, Bertot, Yves, Cohen, Cyril, Garillot, François, Le Roux, Stéphane, Mahboubi, Assia, O'Connor, Russell, Ould Biha, Sidi, Pasca, Ioana, Rideau, Laurence, Solovyev, Alexey, Tassi, Enrico, & Théry, Laurent. (2013b). A machine-checked proof of the odd order theorem. *International Conference on Interactive Theorem Proving (ITP)*.

Grégoire, Benjamin, & Leroy, Xavier. (2002). A compiled implementation of strong reduction. *International Conference on Functional Programming (ICFP)*.

Harper, Robert, Honsell, Furio, & Plotkin, Gordon. (1993). A framework for defining logics. *Journal of the ACM (JACM)*, **40**(1), 143–184.

Harrison, John. (1995). *Metatheory and reflection in theorem proving: A survey and critique*. Technical Report CRC-053. SRI Cambridge, Millers Yard, Cambridge, UK.

Howe, Douglas J. (1992). Reflecting the semantics of reflected proof. *Pages 227–250 of: Proof theory*. Cambridge University Press.

Hur, Chung-Kil, Neis, Georg, Dreyer, Derek, & Vafeiadis, Viktor. (2013). The power of parameterization in coinductive proof. *ACM Symposium on Principles of Programming Languages (POPL)*.

Klein, Gerwin, Andronick, June, Elphinstone, Kevin, Heiser, Gernot, Cock, David, Derrin, Philip, Elkaduwe, Dhammika, Engelhardt, Kai, Kolanski, Rafal, Norrish, Michael, Sewell, Thomas, Tuch, Harvey, & Winwood, Simon. (2010). seL4: Formal verification of an operating-system kernel. *Communications of the ACM (CACM)*, **53**(6), 107–115.

Launchbury, John, & Peyton Jones, Simon L. (1994). Lazy functional state threads. *ACM Conference on Programming Languages Design and Implementation (PLDI)*.

Leroy, Xavier. (2009). Formal verification of a realistic compiler. *Communications of the ACM (CACM)*, **52**(7), 107–115.

Malecha, Gregory, & Bengtson, Jesper. (2015). Rtac: A fully reflective tactic language. *International Workshop on Coq for PL (CoqPL)*.

Malecha, Gregory, Chlipala, Adam, & Braibant, Thomas. (2014). Compositional computational reflection. *International Conference on Interactive Theorem Proving (ITP)*.

Mendler, Nax Paul. (1991). Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, **51**(1–2), 159–172.

Miculan, Marino, & Paviotti, Marco. (2012). Synthesis of distributed mobile programs using monadic types in coq. *International Conference on Interactive Theorem Proving (ITP)*.

Miller, Dale. (1991). Unification of simply typed lamda-terms as logic programming. *International Conference on Logic Programming (ICLP)*.

Nanevski, Aleksandar. (2002). Meta-programming with names and necessity. *International Conference on Functional Programming (ICFP)*.

Nanevski, Aleksandar, Pfenning, Frank, & Pientka, Brigitte. (2008a). Contextual modal type theory. *ACM Transactions on Computational Logic (TOCL)*, **9**(3), 23:1–23:49.

Nanevski, Aleksandar, Morrisett, Greg, & Birkedal, Lars. (2008b). Hoare type theory, polymorphism and separation. *Journal of Functional Programming (JFP)*, **18**(5-6), 865–911.

Nanevski, Aleksandar, Morrisett, Greg, Shinnar, Avi, Govereau, Paul, & Birkedal, Lars. (2008c). Ynot: Dependent types for imperative programs. *International Conference on Functional Programming (ICFP)*.

Nanevski, Aleksandar, Vafeiadis, Viktor, & Berdine, Josh. (2010). Structuring the verification of heap-manipulating programs. *ACM Symposium on Principles of Programming Languages (POPL)*.

Pientka, Brigitte. (2008). A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. *ACM Symposium on Principles of Programming Languages (POPL)*.

Pientka, Brigitte, & Dunfield, Joshua. (2008). Programming with proofs and explicit contexts. *International Symposium on Principles and Practice of Declarative Programming (PPDP)*.

Pollack, Robert. (1995). On extensibility of proof checkers. *TYPES*.

Poswolsky, Adam, & Schürmann, Carsten. (2009). System description: Delphin – a functional programming language for deductive systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, **228**, 113–120.

Sacerdoti Coen, Claudio. (2004). *Mathematical knowledge management and interactive theorem proving*. Ph.D. thesis, University of Bologna.

Saïbi, Amokrane. (1997). Typing algorithm in type theory with inheritance. *ACM Symposium on Principles of Programming Languages (POPL)*.

Schürmann, Carsten, Poswolsky, Adam, & Sarnat, Jeffrey. (2005). The ∇-calculus. Functional programming with higher-order encodings. *International Conference on Typed Lambda Calculi and Applications (TLCA)*.

Sozeau, Matthieu. (2007). Subset coercions in Coq. *TYPES*.

Sozeau, Matthieu, & Oury, Nicolas. (2008). First-class type classes. *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*.

Stampoulis, Antonis, & Shao, Zhong. (2010). VeriML: Typed computation of logical terms inside a language with effects. *International Conference on Functional Programming (ICFP)*.

Stampoulis, Antonis, & Shao, Zhong. (2012). Static and user-extensible proof checking. *ACM Symposium on Principles of Programming Languages (POPL)*.

The Coq Development Team. (2012). *The Coq Proof Assistant Reference Manual – Version V8.4*.

Vafeiadis, Viktor. (2013). Adjustable references. *International Conference on Interactive Theorem Proving (ITP)*.

van der Walt, Paul, & Swierstra, Wouter. (2013). Engineering proof by reflection in Agda. *Implementation and application of functional languages (IFL)*.

Ševčík, Jaroslav, Vafeiadis, Viktor, Zappa Nardelli, Francesco, Jagannathan, Suresh, & Sewell, Peter. (2013). CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM (JACM)*, **60**(3), 22:1–22:50.

Ziliani, Beta, Dreyer, Derek, Krishnaswami, Neelakantan R., Nanevski, Aleksandar, & Vafeiadis, Viktor. (2013). Mtac: A monad for typed tactic programming in Coq. *International Conference on Functional Programming (ICFP)*.