A Proof-Theoretic Approach to Tactics

Kamal Aboul-Hosn

Department of Computer Science
Cornell University
Ithaca, New York, USA
kamal@cs.cornell.edu

Abstract. *Tactics* and *tacticals*, programs that represent and execute several steps of deduction, are fundamental to theorem provers providing automated tools for creating proofs quickly and easily. The language used for tactics is usually a full-scale programming language, separate from the language used to represent proofs. Consequently, there is also a separation between the use of theorems in proofs and the use of tactics. Our goal is to represent tactics in a way that allows them to be treated at the same level as proofs and theorems. We also want a representation that allows us to formally translate tactics into the proof steps they represent. We extend a system presented in [1, 2] to represent tactics at the same level as theorems and move freely from tactics to proof steps and provide an example of its usefulness.

1 Introduction

Theorem provers such as Coq, NuPRL, and Isabelle provide extensive tools for users to create proofs quickly with automated methods. Fundamental to these systems is the use of *tactics* and *tacticals*, programs that represent and execute several steps of deduction. The language used for tactics is typically a full-scale programming language, separate from the language used to represent proofs. Consequently, there is also a separation between the use of theorems in proofs and the use of tactics.

Giunchiglia and Traverso succinctly state the properties desired for a tactic language: the tactics should be expressions of a logical language in order to facilitate reasoning about them; and, there should be a correspondence between the tactics as represented in this logical language and the programs that implement the tactics [3]. Syme calls the correspondence *justifications*, hints about the proof manually specified for the automated tactics [4].

The ML-like languages for tactics in Coq, NuPRL, and Isabelle are generally well suited to this task [5–7]. The tactics found in these systems are built from basic inference rules into complex programs that can apply rules, choose between tactics to apply, and analyze the current structure of a proof. These powerful constructs can automate a great deal of the theorem proving process.

Appel and Felty have looked at implementing tactics in higher-order logic programming languages such as Lambda Prolog and Twelf [8, 9]. They use the

power of backtracking in these systems to facilitate the creation and execution of tactics.

All of these approaches share one thing in common: they implement tactics at a system level that is separate from the level of proofs and theorems. The system-level implementation allows the tactics to be more powerful than the underlying logic. The power is particularly important in the automated search for proofs, where the user creates tactics and tacticals so that a theorem prover can complete a proof with little or no interaction. In fact, it is this desire for automation that drives the decision to separate tactics from proofs.

Despite being implemented at different levels, theorems and tactics have much in common. Both store and repeat proof steps. They represent generalized proving techniques used often within the theory in which they exist. Moreover, both provide guidance and hints to a user regarding the completion of a proof; proofs that share a few tactics or theorems are likely to share more. Nevertheless, work in the area of tactics and tacticals focuses on developing automated proof steps at the system level, separate from the underlying logic in which they work.

The power of the separate tactics language comes at a price, as explained by Delahaye [10]. Separating the two languages requires a user to learn two languages when creating proofs and the developer to create a separate infrastructure for debugging and validating tactics.

The separation between tactics and theorems also inhibits our flexibility in proof representation. While tactics may be used to automate proof steps, they are not represented in the completed proof; the tactics merely apply a sequence of elementary inference rules that a user would perform manually without the tactics. There are times when formally representing the tactics at the same level as proofs can be useful, particularly when transferring a proof to a paper. If a step in the proof is repeated several times by a tactic, we may want to perform the step explicitly the first time and then say that the step is "repeated several times in the same way." We want to be able to represent such a statement formally in the proof itself.

Our goal is to represent tactics in a way that allows them to be treated at the same formal level as proofs and theorems, independent of their system-level implementation. Many very useful tactics on commonly used algebras only require simple constructs that can be represented easily in the same way as theorems, not needing Turing-complete languages used in theorem provers. For example, a tactic for substitution of equals for equals require congruence rules for each operation in the algebra, the ability to iterate through several steps of using different congruence rules, and the ability choose the appropriate congruence rule at each step.

We also want a representation that allows us to easily translate tactics into the proof steps they represent using proof-theoretic, formal rules. Such a representation gives us the flexibility to make proofs more general by using the tactics in the representation or more specific by using some or all of the individual proofs steps.

Finally, the representation should be independent of search techniques and algorithms used to implement automated proof search. While these issues are important for a theorem prover, they are system-level decisions orthogonal to the choices made in representing tactics.

In this paper, we propose such a representation. We extend a system presented in [1, 2] to represent tactics at the same level as theorems and move freely from tactics to proof steps. We formalize several common tactics and propose a way to represent them in our proof system. We then provide formal rules for creating and manipulating tactics and their use in proofs. Finally, we provide an extended example for creating a simple tactic and using it.

2 A Motivating Example

Consider reasoning about a Boolean algebra $(B, \vee, \wedge, \neg, 0, 1)$. Boolean algebra is an equational theory, thus contains the axioms of equality:

$$\mathsf{ref} : \forall x. \quad x = x \tag{1}$$

$$\mathsf{sym} : \forall x, y. \quad x = y \ \rightarrow \ y = x \tag{2}$$

$$\mathsf{trans} : \forall x, y, z. \quad x = y \ \rightarrow \ y = z \ \rightarrow \ x = z \tag{3}$$

$$\mathsf{cong}_{\wedge} : \forall x, y, z. \quad x = y \ \rightarrow \ (z \wedge x) = (z \wedge y) \tag{4}$$

$$\mathsf{cong}_{\vee} : \forall x, y, z. \quad x = y \ \rightarrow \ (z \vee x) = (z \vee y) \tag{5}$$

$$\mathsf{cong}_{\neg} : \forall x, y, z. \quad x = y \ \rightarrow \ \neg x = \neg y \tag{6}$$

$$\mathsf{idemp}_{\wedge} : \forall x. \quad x \wedge x = x \tag{7}$$

Let us look at a particular form of tactic. It is easy to see that the axiom $\mathsf{idemp}_{\wedge}$ allows us to prove

$$\forall a. \quad a \wedge a \wedge a = a \tag{8}$$

Once we have the proof of (8), we can use it to prove

$$\forall a. \quad a \wedge a \wedge a \wedge a = a \tag{9}$$

in the following way. From (8) and $\mathsf{cong}_{\wedge}$ with the substitution $[x/a \wedge a \wedge a, y/a, z/a]$, we can deduce

$$a \wedge a \wedge a \wedge a = a \wedge a \tag{10}$$

We then use $\mathsf{idemp}_{\wedge}$ to get

$$a \wedge a = a \tag{11}$$

Finally, we apply trans to (10) and (11) with the substitution $[x/a \wedge a \wedge a \wedge a, y/a \wedge a, z/a]$ to conclude

$$a \wedge a \wedge a \wedge a = a$$

which is true for arbitrary a, yielding our desired conclusion (9). We can continue to prove a theorem like this for $n + 1$ occurrences of a using the proof for n occurrences of a.

The form of this proof is typical: an inductive argument where we use the result from one proof to prove a step in the next proof. We wish to generalize this kind of proof as a tactic that allows one to represent the execution of several steps of the proof either with the tactic itself or with the individual proof steps.

The need to recover the steps is important, particularly for presentation. Imagine one proves a theorem such as (9). Given that the proof steps are similar and repeated, one may wish to state the proof step explicitly once and then capture the rest of the iterations with one statement.

3 Tactic Representation

For representing tactics, we extend a proof representation system designed to create, remember, and reuse proofs from [1, 2]. The papers present a *publish-cite* system, which uses proof rules with an explicit library to formalize the representation and reuse of theorems and lemmas. The system uses universal Horn equational logic, and we do as well, since it is a good vehicle for illustrating the organization and reuse of theorems. There is no inherent limitation in the system that requires the use of this logic; it could be extended to work with more complex deductive systems.

We build theorems from terms and equations. Consider a set of *individual variables* $X = \{x, y, \ldots\}$ and a first-order signature $\Sigma = \{f, g, \ldots\}$. An *individual term* $s, t, \ldots$ is either a variable $x \in X$ or an expression $ft_1 \ldots t_n$, where f is an n-ary function symbol in Σ and $t_1 \ldots t_n$ are individual terms. An equation $d, e, \ldots$ is between two individual terms, such as $s = t$.

A *theorem* is a universally quantified Horn formula of the form

$$\forall x_1, \ldots x_m. \quad \varphi_1 \ \rightarrow \ \varphi_2 \ \rightarrow \ \cdots \ \rightarrow \ \varphi_n \ \rightarrow \ \psi \tag{12}$$

where the φ_i are equations representing *premises*, ψ is an equation representing the conclusion, and $x_1 \ldots x_m$ are the variables that occur in the equations $\varphi_1, \ldots, \varphi_n, \psi$. A formula may have zero or more premises. These universally quantified formulas allow arbitrary specialization through term substitution. An example of this is the use of $\mathsf{cong}_\wedge$ with substitutions to get (10).

Next, we must define a proof term. For simplicity, we use the model presented in [1]. Let $\mathcal{P}$ be a set of *proof variables* $p, q, \ldots$. A proof of a theorem is a λ-term abstracted over both the proof variables for each premise of a theorem proven by the proof and the individual terms that appear in the proof. A *proof term* is:

- a variable $p \in \mathcal{P}$
- a constant, referring to the name of a theorem
- an application $\pi\tau$, where π and τ are proof terms
- an application πt, where π is a proof term and t is an individual term
- an abstraction $\lambda p.\tau$, where p is proof variable and τ is a proof term
- an abstraction $\lambda x.\tau$, where x is an individual variable and τ is a proof term

When creating proof terms, we have the typing rules seen in Table 1. These typing rules are what one would expect for a simply-typed λ-calculus. The typing

environment Γ maps variables and constants to types. According to the Curry-Howard Isomorphism, the type of a well-typed λ-term corresponds to a theorem in constructive logic and the λ-term itself is the proof of that theorem [11]. For example, a theorem such as (12) viewed as a type would be realized by a proof term representing a function that takes an arbitrary substitution for the variables x_i and proofs of the premises φ_i and returns a proof of the conclusion ψ.

$$\frac{}{\Gamma, p : e \vdash p : e} \qquad \frac{}{\Gamma, c : \varphi \vdash c : \varphi}$$

$$\frac{\Gamma \vdash \pi : e \ \rightarrow \ \varphi \quad \Gamma \vdash \tau : e}{\Gamma \vdash \pi \ \tau : \varphi} \qquad \frac{\Gamma \vdash \pi : \forall x.\varphi}{\Gamma \vdash \pi \ t : \varphi[x/t]}$$

$$\frac{\Gamma, p : e \vdash \tau : \varphi}{\Gamma \vdash \lambda p.\tau : e \ \rightarrow \ \varphi} \qquad \frac{\Gamma \vdash \tau : \varphi}{\Gamma \vdash \lambda x.\tau : \forall x.\varphi}$$

Table 1. Typing rules for proof terms

We use the following notation throughout the rest of the paper:

- $\overline{x}$ is a set of elements $\{x_1, \ldots, x_n\}$.
- $\varphi[\overline{x}/\overline{t}]$ means for all i, replace $x_i \in \overline{x}$ in φ with $t_i \in \overline{t}$.
- $\overline{\pi : \varphi}$ is the list of proof term typing statements $\pi_1 : \varphi_1, \ldots, \pi_n : \varphi_n$.

In order to use tactics, we introduce a few new proof terms:

- A *case statement*,

$$\begin{aligned} \mathsf{case}\ \delta\ \mathsf{of}\ & {=}\varphi_1 \Rightarrow \pi_1 \\ & \ldots \\ & {=}\varphi_n \Rightarrow \pi_n \\ & \psi_1 \Rightarrow \tau_1 \\ & \ldots \\ & \psi_m \Rightarrow \tau_m \end{aligned}$$

  where $\delta, \varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m$ are formulas and $\pi_1, \ldots, \pi_n, \tau_1, \ldots, \tau_m$ are proof terms. The case statement is very similar to the one in Standard ML. We look at the structure of δ and match it against the types in the body of the statement. There are two kinds of matches that can occur. We can exactly match the type δ with a type φ_i, signified by the $=$, or we match a type δ against a possible unification, ψ_j. The difference is that a type δ matches a case $=\varphi_i$ if $\delta = \varphi_i$, whereas it matches a case ψ_j if there exists a substitution such that $\delta = \psi_j[\overline{x}/\overline{t}]$. The proof to the right of the $\Rightarrow$ of the matched case is a proof of the type δ, as enforced by the type system.
  We use the notation $\overline{=\varphi \Rightarrow \pi}$ to represent $=\varphi_1 \Rightarrow \pi_1 \ldots =\varphi_n \Rightarrow \pi_n$ and $\overline{\psi \Rightarrow \tau}$ to represent $\psi_1 \Rightarrow \tau_1 \ldots \psi_n \Rightarrow \tau_m$.

– A *formula variable* X, representing a quantified or unquantified formula
– A *formula abstraction* $\lambda X.\pi$, where X is a formula variable and π is a proof term. We need this proof term in order to abstract over the δ found in the case statement.

To support tactics, we extend formulas with *recursive types* [12],[13, Ch. 20]. We require the addition of three types:

– A *formula variable* X.
– A *recursive formula* $\mu X.\varphi$, where X is a formula variable and φ is a formula.
– A *sum formula* $\{\delta : \ =\varphi_1 + \ldots + =\varphi_n + \psi_1 + \ldots + \psi_m\}$ where δ,$\varphi_1, \ldots, \varphi_n$,$\psi_1, \ldots, \psi_m$ are formulas. We use the notation $\{\delta : \overline{=\varphi} + \overline{\psi}\}$ to represent $\{\delta : \ =\varphi_1 + \ldots + =\varphi_n + \psi_1 + \ldots + \psi_m\}$. The sum formula is closely related to the case statement, as will be apparent when examining the typing rules. In fact, we refer to an individual $=\varphi_i$ or ψ_j in a sum formula as a case.

The typing rules for the new proof terms are in Table 2. With the presence of abstraction over type variables, we need to type the formulas with kinds [13, Ch. 29]. The kinds primarily provide information for matching a formula with a case in a sum formula. Kinds are built from a base kind $*$ and the first-order signature $\Sigma = \{f, g, \ldots\}$. A *kind term* s_*, t_* is a *base kind* $*$ or an expression $f\ t_{1*} \ldots t_{n*}$ where f is an n-ary function symbol in Σ and $t_{1*}, \ldots, t_{n*}$ are kind terms. A kind equation d_*, e_* is between two kind terms, such as $s_* = t_*$.

For the most part, kind information is implicit; the kind $s_* = t_*$ of an equation $s = t$ is formed by replacing all variables in s and t with $*$. However, we may want to be explicit about kind information when the kind is more specific than the type. For example, the type $x = y$ implicitly has the kind $* = *$. If we mean for it to represent a more specific kind, say, $*\vee* = *\wedge*$ in our Boolean algebra example, we would have to specify the kind explicitly with the notation $(x = y : *\vee* = *\wedge*)$. A type's explicit kind can never be less specific than its implicit kind, i.e., $x \wedge y = y$ cannot have the kind $* = *$. We use the explicit kinds to match formulas with cases in the sum formula.

The type of a case statement with a formula variable X is the sum formula formed from the types of the proofs in the body of the statement. The second and third typing rules allow us to be more specific about a proof with a sum formula type. The type of a proof with a formula δ is δ if either δ is equal to one of the φ_i or δ unifies with or has the same kind as one of the ψ_i.

The type of the formula abstraction is the universal quantification over that formula. It is important to note that this is not the same as an abstraction over a proof variable p with the type φ. A term $\lambda p : \varphi.\pi$ would have the type $\varphi \ \rightarrow \ \psi$, where ψ is the type of π. When typing the application of a formula abstraction, the replacement of X with φ requires us to use the kind information. The only place such type variables appear is in case statements.

Finally, we have typing rules for proof terms with recursive types. The two typing rules correspond to unfolding and folding the proof term. We take an equi-recursive approach to the recursive types. In other words, $\mu X.\varphi$ is equivalent to $\varphi[X/\mu X.\varphi]$.

$$\frac{\Gamma \vdash \pi_1 : \varphi_1 \quad \ldots \quad \Gamma \vdash \pi_n : \varphi_n \quad \Gamma \vdash \tau_1 : \psi_1 \quad \ldots \quad \Gamma \vdash \tau_m : \psi_m}{\Gamma \vdash \mathsf{case}\ X\ \mathsf{of}\ \overline{=\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \{X : \overline{\varphi} + \overline{\psi}\}}$$

$$\frac{\Gamma \vdash \pi : \{\delta : \overline{=\varphi} + \overline{\psi}\}}{\Gamma \vdash \pi : \delta}\ \varphi_i = \delta$$

$$\frac{\Gamma \vdash \pi : \{\delta : \overline{=\varphi} + \overline{\psi}\}}{\Gamma \vdash \pi : \delta}\ \begin{array}{l}\psi_i[\overline{x}/\overline{t}] = \delta \text{ or} \\ \delta : e_*, \psi_i : e_*\end{array}$$

$$\frac{\Gamma \vdash \pi : \psi}{\Gamma \vdash \lambda X.\pi : \forall X.\psi} \qquad \frac{\Gamma \vdash \pi : \forall X.\psi}{\Gamma \vdash \pi\ \varphi : \psi[X/\varphi]}$$

$$\frac{\Gamma \vdash \lambda p.\pi : \mu X.\varphi}{\Gamma \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi} \qquad \frac{\Gamma \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi}{\Gamma \vdash \lambda p.\pi : \mu X.\varphi}$$

Table 2. Typing rules for new proof terms

From the standpoint of an automated theorem prover, it is our type system that does most of the work of finding the correct steps to apply from a tactic. Most of this work is in choosing the correct case when applying a case statement to a type δ. Without any restrictions, δ may match several cases, requiring the type system to search though an exponential number of possible proofs. It is this search problem that makes implementing theorem prover tactics difficult. We regard the search problem as an implementation issue separate from the issue of formally representing tactics that we deal with in this paper. For the sake of this paper, we assume that when matching a type against possible cases in a case statement, we only explore the first match found, which removes the need for search at all.

We provide several rules for creating and manipulating proofs. The rules allow one to build proofs constructively. They manipulate a structure of the form $\mathcal{L}; \mathcal{T}$, where

- $\mathcal{L}$ is the library of theorems, $T_1 = \pi_1, \ldots, T_n = \pi_n$
- $\mathcal{T}$ is a list of annotated *proof tasks* of the form $A \vdash \pi : \varphi$, where A is a list of assumptions, π is a proof term, and φ is a formula.

The proof rules can easily be extended to handle theorem scoping as in [2].

In Table 3, we present the rules for basic proof manipulation. The rules are very similar to the ones in [1]. One difference is that the **(ident)** and **(assume)** rules allow one to introduce assumptions with formula types and not just equations. We also add the **(inst)** rule, which allows us to instantiate variables over which a proof term is abstracted. Before, this was handled by the **(cite)** rule, but new rules give us the ability to have term abstractions in proof tasks, so we need to instantiate explicitly.

We also have **(norm$_t$)** and **(norm$_p$)** rules for performing β-reduction on applications of λ-abstractions over terms and proofs, respectively. It is important

$$\textbf{(assume)}\ \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \tau : \psi}{\mathcal{L}\ ;\ \mathcal{T},\quad A, p : \varphi \vdash \tau : \psi}$$

$$\textbf{(ident)}\ \frac{\mathcal{L}\ ;\ \mathcal{T}}{\mathcal{L}\ ;\ \mathcal{T},\quad p : \varphi \vdash p : \varphi}$$

$$\textbf{(mp)}\ \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi : \varphi\ \rightarrow\ \psi\quad A \vdash \tau : \varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi\ \tau : \psi}$$

$$\textbf{(discharge)}\ \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A, p : e \vdash \tau : \psi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \lambda p.\tau : e\ \rightarrow\ \psi}$$

$$\textbf{(publish)}\ \frac{\mathcal{L}\ ;\ \mathcal{T},\quad \vdash \pi : \varphi}{\mathcal{L}, T = \lambda\overline{x}.\pi : \forall\overline{x}.\varphi\ ;\ \mathcal{T}}$$

$$\textbf{(cite)}\ \frac{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T}}{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T},\quad \vdash \pi : \varphi}$$

$$\textbf{(inst)}\ \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \lambda x.\pi : \forall x.\varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi\ t : \varphi[x/t]}$$

$$\textbf{(norm}_t\textbf{)}\ \frac{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash (\lambda x.\pi)\ t}{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash \pi[x/t] : \varphi}$$

$$\textbf{(norm}_p\textbf{)}\ \frac{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash (\lambda p.\pi)\ \tau}{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash \pi[p/\tau] : \varphi}$$

$$\textbf{(forget)}\ \frac{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T}}{\mathcal{L}_1, \mathcal{L}_2[T/\pi]\quad\ ;\ \mathcal{T}[T/\pi]}$$

Table 3. Proof Rules for Basic Theorem Manipulation

to note that the **(norm$_t$)** rule does not replace x in a proof in a case of a case statement where we perform unification if x occurs in the type for that case. In other words, for the proof term

$$\mathsf{case}\ X\ \mathsf{of}\ \overline{=\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \{X :\ \overline{=\varphi}\ +\ \overline{\psi}\}$$

we do not replace x in τ_i if it occurs in ψ_i. We do, however, replace x in any of the π_i and φ_i in which they occur. This behavior is not unlike the case statement in Standard-ML. The **(forget)** rule allows us to remove a theorem from the library. With the possibility of recursive proof terms, the **(forget)** rule must perform its replacement of T with π and normalization repeatedly until T no longer appears.

In Table 4, we introduce the proof rules to create, use, and manipulate theorems and tactics. The **(case)** rule combines existing proof tasks into a case statement. The types variable X can be unified with one of the types $\varphi_1, \ldots, \varphi_n$ or matched exactly with one of types of the assumptions $p_1, \ldots, p_m$. These types must be equations. The **(decase$^=$)** and **(decase)** allow us to determine which

$$\textbf{(case)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A, \overline{p : e} \vdash \pi_1 : \varphi_1 \quad \ldots \quad A, \overline{p : e} \vdash \pi_n : \varphi_n}{\mathcal{L}\ ;\ \mathcal{T},\quad \vdash \mathsf{case}\ X\ \mathsf{of}\ \overline{=e \Rightarrow p}, \overline{\varphi \Rightarrow \pi} : \{X :\ \overline{=e}\ +\ \overline{\varphi}\}}$$

$$\textbf{(decase}^{=}\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \mathsf{case}\ \delta\ \mathsf{of}\ \overline{=\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \delta}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi_i : \delta}\ \varphi_i = \delta$$

$$\textbf{(decase)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \mathsf{case}\ \delta\ \mathsf{of}\ \overline{=\varphi \Rightarrow \pi}, \overline{\psi \Rightarrow \tau} : \delta}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \tau_i[\overline{x}/\overline{t}] : \delta}\ \psi_i[\overline{x}/\overline{t}] = \delta$$

$$\textbf{(fold)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \lambda p.\pi : \mu X.\varphi}$$

$$\textbf{(unfold)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \lambda p.\pi : \mu X.\varphi}{\mathcal{L}\ ;\ \mathcal{T},\quad A \vdash \pi[p/\lambda p.\pi] : \mu X.\varphi}$$

$$\textbf{(publish}^{r}\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T},\quad p : \mu X.\psi \vdash \pi : \varphi}{\mathcal{L}, p = \lambda \overline{x}.\lambda p.\pi : \forall \overline{x}.\mu X \varphi\ ;\ \mathcal{T}}\ \mu X.\psi = \forall \overline{x}.\mu X \varphi$$

$$\textbf{(forget}_1\textbf{)}\quad \frac{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T},\quad A \vdash T\ \tau : \psi}{\mathcal{L}_1, T = \pi : \varphi, \mathcal{L}_2\ ;\ \mathcal{T},\quad A \vdash \pi\ \tau : \psi}$$

$$\textbf{(norm}_f\textbf{)}\quad \frac{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash (\lambda X.\pi)\ \psi}{\mathcal{L}\ ;\ \mathcal{T}\quad A \vdash \pi[X/\psi] : \varphi}$$

Table 4. Proof Rules for Tactics

case the type δ matches and replace the case statement with the proof term for that specific case.

The rules **(fold)** and **(unfold)** are standard rules one would expect for dealing with recursive types. The **(publish^r)** rule allows us to publish recursive proof terms. In other words, these are tactics that use themselves in the proof. Recursion of this nature is very important for tactics; we want to be able to repeat proof steps several times, such as in our example in Section 2. The rule takes a proof task with a single assumption of a recursive type and moves it to the library. The name assigned to the theorem is the same as the proof variable in the assumption. It is also necessary that the type of the proof variable and the type of the proof term added to the library are equivalent.

We add the **(forget$_1$)** rule, which functions much like **(forget)**, except we replace a theorem name with the proof of that theorem in only a single application in a single proof task and we do not remove the theorem from the library. This rule allows us to make explicit one step in the application of a tactic. Finally, the **(norm$_f$)** rule performs β-reduction on applications of λ-abstractions over formulas.

The steps in creating a tactic with several cases that recursively call the tactic would be as follows:

1. Use the **(assume)** and **(ident)** rules to add a proof variable with the type of the tactic to be created.

2. Create the proof terms for the cases of the tactic, using the assumption added in step 1 for the recursive calls.
3. Use the **(case)** rule to combine the proof terms created in step 2 into a single case statement.
4. Use the **(publish^r)** rule to publish the new tactic.

4 A Constructive Example

We can provide a tactic for our example in Section 2. First, we give a general description of the proof steps in our tactic. For a given x and a, if we want to prove $x \wedge a = a$, we use a recursive tactic that is quantified over an equation Y. If Y is of the form $x = x$, then we use ref to prove the equation true. If Y is of the form $x \wedge a = a$, then it suffices to apply trans to proofs of $x \wedge a = a \wedge a$ and $a \wedge a = a$. The latter follows directly from $\mathsf{idemp}_\wedge$. For the former, we use $\mathsf{cong}_\wedge$ on a proof of $x = a$, which we obtain by recursively calling the tactic.

Let

$$\varphi_R \quad = \quad \mu X.\forall x.\forall a.\forall Y. \quad X \ \rightarrow \ \{Y : x = x \ + \ x \wedge a = a\}$$

First, we use **(ident)** to create a proof task

$$R : \varphi_R \vdash R : \varphi_R \tag{13}$$

Next, let us create the cases of our tactic. We first create what will be the "base case" for our recursion. We use **(cite)**, **(inst)**, and **(assume)** to get the proof task

$$R : \varphi_R \vdash \mathsf{ref}\ x : x = x \tag{14}$$

For the recursive case, we use **(inst)** on (13) and the fact that we use equi-recursive types to get

$$\begin{aligned} R : \varphi_R \vdash R\ x\ a\ (x = a : * \wedge * = *) & \\ : \varphi_R \ \rightarrow \ \{(x = a : * \wedge * = *) : \ x = x \ + \ x \wedge a = a\} & \end{aligned} \tag{15}$$

We have made the kind of $x = a$ explicit in order to make sure it matches the $x \wedge a = a$ case in our sum formula type in φ_R. Next, we use **(mp)** on (15) and (13) to get

$$R : \varphi_R \vdash R\ x\ a\ (x = a : * \wedge * = *)\ R : (x = a : * \wedge * = *) \tag{16}$$

For the rest of the example, we do not show the kind of $x = a$ for readability. To use congruence of $\wedge$, we use **(cite)**, **(inst)**, and **(assume)** to add the task

$$R : \varphi_R \vdash \mathsf{cong}_\wedge\ x\ a\ a : x = a \ \rightarrow \ x \wedge a = a \wedge a \tag{17}$$

We combine (17) and (16) using **(mp)** to get

$$R : \varphi_R \vdash \mathsf{cong}_\wedge\ x\ a\ a\ (R\ x\ a\ (x = a)\ R) : x \wedge a = a \wedge a \tag{18}$$

For the proof of $a \wedge a = a$, we use **(cite)**, **(inst)**, and **(assume)** to add the proof task

$$R : \varphi_R \vdash \mathsf{idemp}_\wedge\ a : a \wedge a = a \tag{19}$$

We introduce transitivity with **(cite)**, **(inst)**, and **(assume)**

$$\begin{aligned} R : \varphi_R \vdash \mathsf{trans}\ (x \wedge a)\ (a \wedge a)\ a :\ & x \wedge a = a \wedge a \\ & \rightarrow\ a \wedge a = a \\ & \rightarrow\ x \wedge a = a \end{aligned} \tag{20}$$

Two applications of **(mp)** with (20),(18), and (19) give the completed recursive case for our tactic:

$$\begin{aligned} R : \varphi_R \vdash \mathsf{trans}\ & (x \wedge a)\ (a \wedge a)\ a && : x \wedge a = a \\ & (\mathsf{cong}_\wedge\ x\ a\ a\ (R\ x\ a\ (x = a)\ R)) \\ & (\mathsf{idemp}_\wedge\ a) \end{aligned} \tag{21}$$

Now we use the **(case)** rule to combine (14) and (21) for our tactic:

$$\begin{aligned} R : \varphi_R \vdash\ & \mathsf{case}\ Y\ \mathsf{of} && : \{Y :\ x = x\ +\ x \wedge a = a\} \\ & (x = x) \Rightarrow \mathsf{ref}\ x \\ & (x \wedge a = a) \Rightarrow \mathsf{trans}\ (x \wedge a)\ (a \wedge a)\ a \\ & \qquad\qquad (\mathsf{cong}_\wedge\ x\ a\ a \\ & \qquad\qquad\quad (R\ x\ a\ (x = a)\ R)) \\ & \qquad\qquad (\mathsf{idemp}_\wedge\ a) \end{aligned}$$

Finally, we use the **(publish^r)** rule to publish the tactic as

$$\begin{aligned} R = \lambda x.\lambda a.\lambda Y.\lambda R.\, & \mathsf{case}\ Y\ \mathsf{of} \\ & (x = x) \Rightarrow \mathsf{ref}\ x \\ & (x \wedge a = a) \Rightarrow \mathsf{trans}\ (x \wedge a)\ (a \wedge a)\ a \\ & \qquad\qquad (\mathsf{cong}_\wedge\ x\ a\ a\ (R\ x\ a\ (x = a)\ R)) \\ & \qquad\qquad (\mathsf{idemp}_\wedge\ a) \end{aligned}$$

The type of this tactic is

$$\forall x.\forall a.\forall Y.\quad \varphi_R\ \rightarrow\ \{Y :\ x = x\ +\ x \wedge a = a\}$$

Notice that $\forall x.\forall a.\forall Y.\quad \varphi_R\ \rightarrow\ \{Y : x = x\ +\ x \wedge a = a\}$ is equal to φ_R, which is necessary for applying the rule.

We now have a tactic that given an x of the form $a \wedge \ldots \wedge a$ will provide a proof of $a \wedge \ldots \wedge a = a$. If applied to a term that is not of this form, the tactic will not have a type.

We can now apply the tactic to create a new proof. We use the **(cite)** and **(inst)** rules just as we do on theorems to create the proof task

$$\vdash R\ (b \wedge b \wedge b)\ b\ (b \wedge b \wedge b \wedge b = b) : \varphi_R\ \rightarrow\ b \wedge b \wedge b \wedge b = b$$

We then use **(cite)** and **(mp)** to get the conclusion we desire.

$$\vdash R\ (b\wedge b\wedge b)\ b\ (b\wedge b\wedge b\wedge b = b)\ R : b\wedge b\wedge b\wedge b = b \tag{22}$$

We may want to make one step of the application of the tactic R explicit. First, we use the **(forget$_1$)** rule on (22) to replace the name of the tactic with its body and then use the normalize rules to perform β-reduction to get

$$\begin{array}{l} \vdash \mathsf{case}\ (b\wedge b\wedge b\wedge b = b)\ \mathsf{of} \\ \quad (x = x) \Rightarrow \mathsf{ref}\ x \\ \quad (x\wedge a = a) \Rightarrow \mathsf{trans}\ (x\wedge a)\ (a\wedge a)\ a \\ \qquad\qquad\qquad\qquad (\mathsf{cong}_\wedge\ x\ a\ a\ (R\ x\ a\ (x = a)\ R)) \\ \qquad\qquad\qquad\qquad (\mathsf{idemp}_\wedge\ a) \end{array} : b\wedge b\wedge b\wedge b = b \tag{23}$$

We can then use **(decase)** to replace the case statement with the specific case that is matched, where $b\wedge b\wedge b\wedge b = b$ unifies with $x\wedge a = a$ under the substitution $[x/b\wedge b\wedge b, a/b]$.

$$\begin{array}{l} \vdash \mathsf{trans}\ (b\wedge b\wedge b\wedge b)\ (b\wedge b)\ b \\ \qquad (\mathsf{cong}_\wedge\ (b\wedge b\wedge b)\ b\ b \\ \qquad\ (R\ (b\wedge b\wedge b)\ b\ (b\wedge b\wedge b = b)\ R)) \\ \qquad (\mathsf{idemp}_\wedge\ b) \end{array} : b\wedge b\wedge b\wedge b = b \tag{24}$$

Now one of the steps of the proof is explicit while the others are implicitly captured in the application of the tactic R.

5 Conclusion

We have presented a proof-theoretic approach in which tactics are treated at the same level as theorems and proofs. The proof rules allow us to create, manipulate, and apply tactics in a way that is completely formal and independent of system-level decisions regarding proof search. Many important tactics can be represented in the relatively simple system we have demonstrated, particularly in algebras such as our Boolean example.

Representing tactics at this level has several advantages for automated theorem provers, from both a user perspective and a developer perspective. For users, powerful tactics can be created without needing to learn a separate tactics language. However, the power of the language used to implement the theorem prover can be harnessed to make proof search as complete and efficient as desired. Additionally, when combined with the work in [2], tactics can be put into a local scope and abstractions can be manipulated just as we can with theorems, a powerful ability lacking from current theorem provers.

In the future, we plan to have a full implementation of tactics added to the Java implementation mentioned in [2] for Kleene algebra with tests [14]. We can then investigate the ability of the system to discover repeated citations in proofs that can be abstracted out at lemmas. Given the structure of tactics and theorems, detecting similar subproofs is a form of common subexpression elimination, a process we call *proof refactorization*. With this system, one could easily use tactics with a strong underlying formalism guiding their manipulation.

Acknowledgements

We would like to thank Dexter Kozen and Sigmund Cherem for valuable ideas and comments. This paper is dedicated in loving memory to the author's grandfather, Orlen F. Rice, who passed away during its writing.

References

1. Kozen, D., Ramanarayanan, G.: A proof-theoretic approach to knowledge acquisition. Technical Report 2005-1985, Computer Science Department, Cornell University (2005)
2. Aboul-Hosn, K., Damhøj Andersen, T.: A proof-theoretic approach to hierarchical math library organization. In Kohlhase, M., ed.: MKM. Volume 3863 of Lecture Notes in Computer Science., Springer (2005) 1–16
3. Giunchiglia, F., Traverso, P.: Program tactics and logic tactics. Annals of Mathematics and Artificial Intelligence **17**(3-4) (1996) 235–259
4. Syme, D.: Three tactic theorem proving. In: TPHOLs '99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics, London, UK, Springer-Verlag (1999) 203–220
5. Kreitz, C.: The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide. Department of Computer Science, Cornell University. (2002)
6. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V7.3. (2002) http://coq.inria.fr.
7. Wenzel, M., Berghofer, S.: The Isabelle System Manual. (2003)
8. Felty, A.: Implementing tactics and tacticals in a higher-order logic programming language. Journal of Automated Reasoning **11**(1) (1993) 43–81
9. Appel, A.W., Felty, A.P.: Dependent types ensure partial correctness of theorem provers. J. Funct. Program. **14**(1) (2004) 3–19
10. Delahaye, D.: A tactic language for the system Coq. In Parigot, M., Voronkov, A., eds.: LPAR. Volume 1955 of Lecture Notes in Computer Science., Springer (2000) 85–95
11. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry–Howard isomorphism. Available as DIKU Rapport 98/14 (1998)
12. Morris, J.H.: Lambda calculus models of programming languages. Technical report, Massachuseets Instititue of Technology, Laboratory for Computer Science (1968)
13. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
14. Kozen, D.: Kleene algebra with tests. Transactions on Programming Languages and Systems **19**(3) (1997) 427–443