A Proof-Theoretic Approach to Hierarchical Math Library Organization

Kamal Aboul-Hosn and Terese Damhøj Andersen

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

Abstract. The relationship between theorems and lemmas in mathematical reasoning is often vague. No system exists that formalizes the structure of theorems in a mathematical library. Nevertheless, the decisions we make in creating lemmas provide an inherent hierarchical structure to the statements we prove. In this paper, we develop a formal system that organizes theorems based on *scope*. Lemmas are simply theorems with a local scope. We develop a representation of proofs that captures scope and present a set of proof rules to create and reorganize the scopes of theorems and lemmas. The representation and rules allow systems for formalized mathematics to more accurately reflect the natural structure of mathematical knowledge.

1 Introduction

The relationship between theorems and lemmas in mathematical reasoning is often vague. What makes a statement a lemma, but not a theorem? One might say that a theorem is "more important," but what does it mean for one statement to be "more important" than another? When writing a proof for a theorem, we often create lemmas as a way to break down the complex proof, so perhaps we expect the proofs of lemmas to be shorter than the proofs of theorems. We also create lemmas when we have a statement that we do not expect to last in readers' minds, i.e., it is not the primary result of our work. The way we make these decisions while reasoning provides an inherent hierarchical structure to the set of statements we prove. However, no formal system exists that explicitly organizes proofs into this hierarchy.

Theorem provers such as NuPRL, Coq, and Isabelle provide the ability to create lemmas. But their library structures are flat, and no formal distinction exists between lemmas and theorems [1–3]. The reasons to distinguish lemmas from theorems in these systems is the same as the reasons in papers: to ascribe various levels of importance and to introduce dependency or scoping relationships.

We seek to formalize these notions and provide a proof-theoretic means by which to organize a set of proofs in a hierarchical fashion that reflects this natural structure. Our thesis is that the qualitative difference between theorems and lemmas is in their *scope*. Scope already applies to mathematical notation. Never

in a paper would one need to define the representation of a set ($\{\ldots\}$) nor operators such as union and intersection. Set notation is standard, thus has a global scope that applies to any proof. However, one often defines operators that are only used for a single paper; the author does not intend for the notation to exist in other papers with the same meaning without being defined again. Similarly, a *theorem* is a statement that can be used in any other proof. Its scope is global, just as set notation. A *lemma* is a statement with a local scope limited to a particular set of proofs. We want a system that represents and manipulates scope formally through the structure of the library of proofs.

In this paper, we propose such a system. First, we propose a formal definition of scoping for proof libraries. Next, we describe a representation of proofs that is able to capture this definition of scope based on work by Kozen and Ramanarayanan [4]. We provide a set of formal rules to create and reorganize the scopes of theorems and lemmas.

We believe that the ability to create and manage complex scoping and dependency relationships among proofs will allow systems for formalized mathematics to more accurately reflect the natural structure of mathematical knowledge.

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:

$$\mathsf{ref} : x = x \tag{1}$$
$$\mathsf{sym} : x = y \to y = x \tag{2}$$
$$\mathsf{trans} : x = y \to y = z \to x = z \tag{3}$$
$$\mathsf{cong}_\wedge : x = y \to (z \wedge x) = (z \wedge y) \tag{4}$$
$$\mathsf{cong}_\vee : x = y \to (z \vee x) = (z \vee y) \tag{5}$$
$$\mathsf{cong}_\neg : x = y \to \neg x = \neg y \tag{6}$$

All variables are implicitly universally quantified in these axioms. Suppose we wanted to prove the following elementary fact:

Theorem 1.

$$\forall a, b, c, z. a = b \to a = c \to z \vee (a \wedge b) = z \vee (a \wedge c) \tag{7}$$

Here is how a proof might go. First, we could prove a lemma

Lemma 1.

$$\forall x, y, z. x = y \to z \vee (x \wedge x) = z \vee (x \wedge y) \tag{8}$$

Using $a = b$ and $a = c$ from the statement of our theorem, we could apply the lemma under the substitutions $[x/a, y/b, z/z]$ and $[x/a, y/c, z/z]$ to deduce

$$z \vee (a \wedge a) = z \vee (a \wedge b) \tag{9}$$
$$z \vee (a \wedge a) = z \vee (a \wedge c) \tag{10}$$

Next, we know from applying symmetry to (9) that

$$z \vee (a \wedge b) = z \vee (a \wedge a) \tag{11}$$

Finally we conclude from transitivity, (9), and (11) that

$$z \vee (a \wedge b) = z \vee (a \wedge c)$$

which is what our theorem states.

We may decide that (8) does not apply to theorems other than (7), and consequently, should only have a scope limited to the proof of (7). Our representation of proofs makes explicit the limited scope of (8).

Another important observation is that in all places we use (8), the variable z from (7) is always used for the variable z in the lemma. We may wish not to universally quantify z for both (7) and (8) individually, but instead universally quantify z once and for all so that it can be used by both proofs:

$$\begin{aligned} &\forall z,\ \forall a, b, c, a = b \rightarrow a = c \rightarrow z \vee (a \wedge b) = z \vee (a \wedge c) \\ &\quad \text{and } \forall x, y, x = y \rightarrow z \vee (x \wedge x) = z \vee (x \wedge y) \end{aligned} \tag{12}$$

Moving the quantifier for z looks like a simple task, applying the first order logic rule

$$(\forall z.\varphi) \wedge (\forall z.\psi) \equiv \forall z.(\varphi \wedge \psi)$$

However, the proof of the lemma itself must also change, as must any proof that is dependent on this lemma.

Although either version of the lemma can be used to prove the theorem, note that their meanings are subtly different because of the placement of the quantification. Placing a separate quantification of z as in (8) makes the lemma read: "Lemma 1: For all x, y, and z,..." In this case, z is a variable in the lemma for which we expect there to be a substitution whenever the lemma is used in a proof. Using one quantification for both the theorem and the lemma as in (12) makes the lemma read: "Let z be an arbitrary, but fixed boolean value. Lemma 1: For all x and y..." In this case, z is a fixed constant for the lemma.

In this simple example, using (8) or (12) does not matter. However, in other cases, the choices made for quantification may reflect a general style in one's proofs. One may like lemmas to be as general as possible, universally quantifying any variables that appear in the lemma and relying on no constants. On the other hand, one may want to make lemmas as specific as possible, applying only in a select few proofs in order to minimize the number of quantifications. We want to capture this subtle difference formally in our representation of proofs in order to allow the user to choose the representation that best fits the intended meaning.

3 Proof Representation

For representing theorems and lemmas like those in Section 2, we use proof terms similar to those defined in a paper by Kozen and Ramanarayanan [4]. Their

paper presents a *publish-cite* system, which uses proof rules with an explicit library to formalize the representation and reuse of theorems. The system of [4] 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 use the word "theorem" to mean a theorem, lemma, or axiom. 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

$$\forall x_1, \ldots x_m . \varphi_1 \rightarrow \varphi_2 \rightarrow \cdots \rightarrow \varphi_n \rightarrow \psi \tag{13}$$

where the φ_is 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 (8) with substitutions to get (9) and (10).

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 [5]. For example, a theorem such as (13) 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.

In [4], a library of theorems is represented as a flat list of proof terms. All of the theorems have global scope, i.e., they are able to be cited in any other proof in the library.

The goal of this paper is to provide a scoping discipline so that naming and use of variables can be localized. The proof term itself should tell us in which proofs we can use a lemma. We use a construct similar to the SML `let` expression, which limits the scope of variables in the same way we wish to limit the scope of lemmas.

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

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

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

Table 1. Typing rules for proof terms

In order to represent theorems in a hierarchical fashion, we add two kinds of proof terms:

- a sequence $\tau_1; \ldots; \tau_n$, where $\tau_1, \ldots, \tau_n$ are proof terms. This allows several proofs to use the same lemmas. Sequences cannot occur inside applications.
- an expression $\mathsf{let}\ L_1 = \tau_1 \ldots L_n = \tau_n\ \mathsf{in}\ \tau\ \mathsf{end}$. This term is meant to express the definition of a set of lemmas for use in a proof term τ. The τ_is are proof terms, each bound to an identifier L_i. With the existence of the sequences, each τ_i may define the proof for more than one lemma. The identifiers L_i are arrays, where the j^{th} element, denoted $L_i[j]$, is the name of the lemma corresponding to the j^{th} proof in τ_i not bound to a name in τ_i, denoted $\tau_i[j]$. The let expression binds names to the proofs and limits their scope to proof terms that appear later in the let expression. In other words, a lemma $L_i[j]$ can appear in any proof $\tau_k, k > i$, or in τ. The name of a lemma has the same type as the proof to which it corresponds. This scoping discipline for lemmas corresponds exactly to the variable scoping used in SML let expressions.

These new rules have corresponding typing rules, in Table 2.

$$\frac{\Gamma \vdash \tau_1 : \varphi_1 \quad \ldots \quad \Gamma \vdash \tau_n : \varphi_n}{\Gamma \vdash \tau_1; \ldots; \tau_n : \varphi_1 \wedge \ldots \wedge \varphi_n}$$

$$\frac{\begin{array}{l}\Gamma \vdash \tau_1 : \varphi_1 \\ \Gamma, L_1 : \varphi_1 \vdash \tau_2 : \varphi_2 \\ \ldots \\ \Gamma, L_1 : \varphi_1, \ldots, L_{n-1} : \varphi_{n-1} \vdash \tau_n : \varphi_n \\ \Gamma, L_1 : \varphi_1, \ldots, L_n : \varphi_n \vdash \tau : \varphi\end{array}}{\Gamma \vdash \mathsf{let}\ L_1 = \tau_1 \ldots L_n = \tau_n\ \mathsf{in}\ \tau\ \mathsf{end} : \varphi_1 \to \ldots \to \varphi_n \to \varphi}$$

Table 2. Typing rules for proof terms

The rule for a sequence of proof terms is relatively straightforward; the type of a sequence is the conjunction of the types of the proof terms in the sequence.

The typing rule for the let expression is based on the scoping of the proofs. We must be able to prove that each proof τ_k has type φ_k under the assumption that all variables $L_i, i < k$ have the type φ_i, where τ_i is assigned to L_i. Finally, we must be able to prove that τ has the type φ under the assumption that every L_i has type φ_i.

As an example, we represent the proofs of (7) and (8) as

$$
\begin{array}{l}
\mathsf{thm} = \\
\quad \mathsf{let\ lem} = \lambda x \lambda y \lambda z \lambda P.(\text{Proof of lemma}) \\
\quad \mathsf{in} \\
\quad\quad \lambda a \lambda b \lambda c \lambda z \lambda Q \lambda R.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ Q))\ \ (\mathsf{lem}\ R) \\
\quad \mathsf{end}
\end{array}
$$

where thm is the name assigned to (7) and lem is the name assigned to (8). For ease of reading, we have omitted the applications of proof terms to individual terms, which represent the substitution for individual variables. P, Q, and R are proofs of type $x = y$, $a = b$, and $a = c$, respectively.

If we choose to universally quantify z only once as in (12), we represent the proof as

$$
\begin{array}{l}
\mathsf{thm} = \\
\quad \lambda z.\mathsf{let\ lem} = \lambda x \lambda y \lambda P.(\text{Proof of lemma}) \\
\quad\quad \mathsf{in} \\
\quad\quad\quad \lambda a \lambda b \lambda c \lambda Q \lambda R.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ Q))\ \ (\mathsf{lem}\ R) \\
\quad\quad \mathsf{end}
\end{array}
$$

As we can see, there is a one-to-one correspondence between the positions of λ-abstractions and where individual variables are universally quantified. We formally develop the proof terms for thm and lem in Section 5.

4 Proof Rules

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

- $\mathcal{L}$ is the library of theorems, $T_1 = \pi_1, \ldots, T_n = \pi_n$, where T_i is an array of identifiers with the j^{th} element denoted $T_i[j]$, naming the j^{th} proof in π_i, denoted $\pi_i[j]$,
- $\mathcal{C}$ is the list of lemmas currently in scope, $L_1 = \tau_1, \ldots, L_m = \tau_m$, with components defined as they are for $\mathcal{L}$, and
- $\mathcal{T}$ 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 an unquantified Horn formula.

In these rules, we use the following notational conventions:

- α and β are proof variables or individual variables.

- $\overline{X}$ is a set of elements $\{X_1, \ldots, X_n\}$, where X_i can be an individual variable or a proof variable.
- $T = \pi$ binds a proof term π to an identifier T. The term π may define the proof for more than one theorem. Therefore, the identifier T is an array, where the j^{th} element, denoted $T[j]$, is the name of the theorem corresponding to the j^{th} proof in π not bound to a name in π, denoted $\pi[j]$.
- $\overline{T} = \overline{\pi}$ is a sequence of bindings $T_1 = \pi_1, \ldots, T_n = \pi_n$.
- $\overline{T} : \overline{\varphi}$ is a sequence of type bindings $T_1 : \varphi_1, \ldots, T_n : \varphi_n$, where $\varphi = \varphi_1 \rightarrow \ldots \rightarrow \varphi_n$.
- $\pi[\overline{x}/\overline{t}]$ means for all i, replace element $x_i \in \overline{x}$ in π with $t_i \in \overline{t}$.
- Given a binding $T = \pi$, $X[T/\pi]$ means for all i, replace $T[i]$ with $\pi[i]$ in X, where X is a proof term, a list of theorems, or a list of proof tasks.
- For a proof term π, a sequence of identifiers $\overline{T} = T_1 \ldots T_n$, and a variable α, $\pi[\overline{T}/\overline{T}\ \alpha]$ means for all i and j, replace $T_i[j]$ with $T_i[j]\ \alpha$, where juxtaposition represents functional application.
- Given a binding $T = \ldots \lambda\alpha_i \lambda\alpha_j \ldots \pi$, $\mathcal{C}[T(i,j)/T(j,i)]$ means for all k, swap the i^{th} and j^{th} term or proof to which $T[k]$ is applied in $\mathcal{C}$.
- $FV(\varphi)$ is the set of free individual variables in the Horn formula φ.

The structure $\mathcal{L}; \mathcal{C}; \mathcal{T}$ must also be well typed, according to the rules in Table 3. The typing rules enforce an order on the list of theorems and lemmas. The rules look very similar to the rules for the let expression.

$$\frac{\begin{array}{l}\Gamma \vdash \pi_1 : \varphi_1 \\ \Gamma, T_1 : \varphi_1 \vdash \pi_2 : \varphi_2 \\ \ldots \\ \Gamma, T_1 : \varphi_1, \ldots, T_{n-1} : \varphi_{n-1} \vdash \pi_n : \varphi_n\end{array}}{\Gamma \vdash \overline{T} = \overline{\pi} : \varphi_1 \rightarrow \ldots \rightarrow \varphi_n}$$

$$\frac{\begin{array}{l}\Gamma \vdash \overline{T} = \overline{\pi} : \varphi_{T1} \rightarrow \cdots \rightarrow \varphi_{Tn} \\ \Gamma, \overline{T} : \overline{\varphi_T} \vdash \overline{L} = \overline{\tau} : \varphi_{L1} \rightarrow \cdots \rightarrow \varphi_{Lm} \\ \Gamma, \overline{T} : \overline{\varphi_T}, \overline{L} : \overline{\varphi_L} \vdash \mathcal{T} : \psi\end{array}}{\Gamma \vdash \overline{T} = \overline{\pi}; \overline{L} = \overline{\tau}; \mathcal{T} : \varphi_{T1} \rightarrow \cdots \rightarrow \varphi_{Tn} \rightarrow \varphi_{L1} \rightarrow \cdots \rightarrow \varphi_{Lm} \rightarrow \psi}$$

Table 3. Typing rules for proof library

We must also have a typing rule for the proof tasks $\mathcal{T}$. The rule is a meta-typing rule on deductions of the form $A \vdash \pi : \varphi$, which we omit for brevity.

The proof rules fit into two categories: rules that manipulate the proof tasks and rules that manipulate the structure of proof terms that appear in $\mathcal{C}$.

4.1 Rules for Manipulating Proof Tasks

The first set of rules is in Table 4. Note that the **(reorder)** rule has a side condition $(*)$ explained below. The first four rules are the same as the rules in

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

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

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

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

$$\textbf{(collect)}\quad \frac{\mathcal{L}\ ;\ \overline{M} = \overline{\pi}\ ;\ \vdash \tau_1 : \varphi_1 \ldots \vdash \tau_n : \varphi_n}{\mathcal{L}\ ;\ \begin{array}{l} L = \mathsf{let}\ \overline{M} = \overline{\pi} \\ \quad \mathsf{in}\ \lambda\overline{x}_1.\tau_1; \ldots; \lambda\overline{x}_n.\tau_n\ \mathsf{end} \end{array}\ ;}\ \overline{x}_i = FV(\varphi_i)$$

$$\textbf{(publish)}\quad \frac{\mathcal{L}\qquad\quad ;\ \overline{L} = \overline{\tau}\ ;}{\mathcal{L}, \overline{L} = \overline{\tau}\ ;\qquad\quad ;}$$

$$\textbf{(tcite)}\quad \frac{\mathcal{L}_1, T = \pi, \mathcal{L}_2\ ;\ \mathcal{C}\ ;\ \mathcal{T}}{\mathcal{L}_1, T = \pi, \mathcal{L}_2\ ;\ \mathcal{C}\ ;\ \mathcal{T},\quad \vdash T[j]\ \overline{t} : \varphi[\overline{x}/\overline{t}]}\ T[j] : \forall\overline{x}.\varphi$$

$$\textbf{(lcite)}\quad \frac{\mathcal{L}\ ;\ \mathcal{C}_1, L = \pi, \mathcal{C}_2\ ;\ \mathcal{T}}{\mathcal{L}\ ;\ \mathcal{C}_1, L = \pi, \mathcal{C}_2\ ;\ \mathcal{T},\quad \vdash L[j]\ \overline{t} : \varphi[\overline{x}/\overline{t}]}\ L[j] : \forall\overline{x}.\varphi$$

$$\textbf{(tforget)}\quad \frac{\mathcal{L}_1, T = \pi, \mathcal{L}_2\ ;\ \mathcal{C}\qquad\quad ;\ \mathcal{T}}{\mathcal{L}_1, \mathcal{L}_2[T/\pi]\quad ;\ \mathcal{C}[T/\pi]\ ;\ \mathcal{T}[T/\pi]}$$

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

$$\textbf{(promote)}\quad \frac{\mathcal{L}\ ;\ \mathcal{L}_1, L = \mathsf{let}\ \overline{M} = \overline{\tau}\ \mathsf{in}\ \pi\ \mathsf{end}, \mathcal{L}_2\ ;}{\mathcal{L}\ ;\ \mathcal{L}_1, \overline{M} = \overline{\tau}, L = \pi, \mathcal{L}_2\qquad\quad ;}$$

$$\textbf{(reorder)}\quad \frac{\mathcal{L}\ ;\ \mathcal{C}_1, L = \lambda\alpha_1 \ldots \lambda\alpha_i\lambda\alpha_j \ldots \lambda_n.\pi, \mathcal{C}_2\qquad\quad ;}{\mathcal{L}\ ;\ \mathcal{C}_1, L = \lambda\alpha_1 \ldots \lambda\alpha_j\lambda\alpha_i \ldots \lambda_n.\pi, \mathcal{C}_2[L(i,j)/L(j,i)]\ ;}\ (*)$$

Table 4. Rules for manipulating proof tasks

[4].

The **(collect)** rule works on a set of tasks with no further assumptions, i.e., tasks with completed proofs. The rule

1. gives the collection of the tasks a new name L that does not appear in the library or the current list of lemmas,
2. forms the universal closures of the φ_is and the corresponding λ-closures of the τ_is, and
3. moves the proofs to the list of lemmas currently in scope.

Any lemmas that were in scope for the proof tasks are explicitly made lemmas with the let statement. These lemmas are no longer immediately available to proof tasks. However, one can access a lemma moved into a let by using the **(promote)** rule. If no lemmas currently exist, a let expression is not created and instead the name L is bound to the λ-closures of the τ_is.

The **(publish)** rule moves the current lemmas to the library, at which point they become theorems.

The **(tcite)** rule is the elimination rule for the universal quantifier for theorems in the library. This rule specializes the theorem with a given substitution $[\overline{x}/\overline{t}]$. It is important to note that the proof $\pi_i[j]$ of $T_i[j]$ is not copied into the proof tasks. If this were the case, then β-reduction on the proof could make it impossible to distinguish between a proof that cited $T_i[j]$ and a proof that developed $\pi_i[j][\overline{x}/\overline{t}]$ explicitly. Instead, the name of the theorem serves as a citation token, with the same type as the proof itself. The **(lcite)** rule does the same for lemmas from $\mathcal{C}$.

The **(tforget)** rule removes all citations of the forgotten theorems and replaces them with the proofs of the theorems. With the proof instead of the citation token, β-reduction on citations of a theorem can take place during proof normalization, creating the specialized version of the proof we did not create in the **tcite** rule. All citations of the theorems T are replaced with a specialized version of the proof π. The **(lforget)** rule does the same for lemmas in $\mathcal{C}$.

The **(promote)** rule moves a set of lemmas from inside a let expression to the list of lemmas currently in scope. This makes these lemmas again available to be cited.

The **(reorder)** rule changes the order of abstractions in a proof term. Correspondingly, citations of any lemmas defined by that proof term must be changed to have the order of their applications changed. The condition $(*)$ is that if α_i is an individual variable and α_j is a proof variable with type φ, then α_i does not occur anywhere in φ. If α_i did occur in φ and we performed **(reorder)**, φ would contain an unbound variable.

4.2 Rules for Manipulating Proof Terms

The set of rules for manipulating proof terms that appear in $\mathcal{C}$ is in Table 5. These rules do not change any proofs of theorems currently in scope for the proof tasks, so we know that any changes in proofs do not have to be reflected in the current tasks. Some of these rules have side conditions, which are marked with a symbol in $(\cdot)$ and explained below.

The **(push)** rule moves an abstraction from the front of a sequence to each proof in the sequence. This rule does not change the types of the proofs; it only duplicates $\lambda\alpha$. One would anticipate using this rule after performing a **(generalize)**.

The **(pull)** rule is the inverse of the **(push)** rule. It moves an abstraction from the front of every proof in a sequence to the front of the entire sequence. This rule would most likely be used before a **(specialize)**.

$$\textbf{(push)} \quad \frac{\lambda\alpha.(\pi_1; \ldots; \pi_n)}{\lambda\alpha.\pi_1; \ldots; \lambda\alpha.\pi_n}$$

$$\textbf{(pull)} \quad \frac{\lambda\alpha.\pi_1; \ldots; \lambda\alpha.\pi_n}{\lambda\alpha.(\pi_1; \ldots; \pi_n)}$$

$$\textbf{(generalize)} \quad \frac{\lambda\alpha.\mathsf{let}\ \overline{L} = \overline{\pi}\ \mathsf{in}\ \tau\ \mathsf{end}}{\mathsf{let}\ \overline{L} = \overline{\lambda\alpha.\pi[\overline{L}/\overline{L}\ \alpha]}\ \mathsf{in}\ \lambda\alpha.\tau[\overline{L}/\overline{L}\ \alpha]\ \mathsf{end}}$$

$$\textbf{(specialize)} \quad \frac{\mathsf{let}\ \overline{L} = \overline{\lambda\alpha.\pi}\ \mathsf{in}\ \lambda\alpha.\tau\ \mathsf{end}}{\lambda\alpha.\mathsf{let}\ \overline{L} = \overline{\pi[\overline{L}\ \alpha/\overline{L}]}\ \mathsf{in}\ \tau[\overline{L}\ \alpha/\overline{L}]\ \mathsf{end}}\ (**)$$

$$\textbf{(split)} \quad \frac{\mathsf{let}\ \overline{L} = \overline{\pi_L}, \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}}{\mathsf{let}\ \overline{L} = \overline{\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end}}$$

$$\textbf{(merge)} \quad \frac{\mathsf{let}\ \overline{L} = \overline{\pi_L}\ \mathsf{in}\ \mathsf{let}\ \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}\ \mathsf{end}}{\mathsf{let}\ \overline{L} = \overline{\pi_L}, \overline{M} = \overline{\pi_M}\ \mathsf{in}\ \tau\ \mathsf{end}}$$

$$\textbf{(rename)} \quad \frac{\lambda\alpha.\pi}{\lambda\beta.\pi[\alpha/\beta]}\ (\#)$$

Table 5. Rules for manipulating proof terms in $\mathcal{C}$

The **(generalize)** rule moves an abstraction from the outside of a let statement to each proof term in the list of defined lemmas and to the proof term τ. This does not change any theorem whose proof is in τ. The proofs and types of the lemmas $\overline{L}$ do change, because they are now abstracted over another variable.

Correspondingly, we have to change any citations of the lemmas. From the scoping discipline, we know exactly where these citations can be: in the proofs of the lemmas, $\overline{\pi}$, or in the proof τ. Before performing **(generalize)**, all the lemmas and τ referred to the same α. Now, the first abstraction for any of the lemmas is over α. Consequently, any citation of the lemmas must be changed to have the first application be to a term that matches α explicitly. Since all of the proofs referred to the same α before the operation, we can simply use the α in the applications and replace all occurrences of $L_i[j]$ with $L_i[j]\ \alpha$.

The types of the L_is and π_is also change. If α is an individual variable, we add another universal quantification to the front of the type. If α is a proof variable, we add another implication, corresponding to a premise.

The **(specialize)** rule does the opposite of **(generalize)**. A variable that was universally quantified for the lemmas L now becomes a constant for them when we move α to the outside of the let. As stated, the rule requires $\lambda\alpha$ to precede every proof π. This is not actually a requirement for correctness, but it makes stating the side condition easier. The side condition $(**)$ is that any citation of a lemma $L_i[j]$ is of the form $L_i[j]\ \alpha$. In other words, the same variable used in the λ-abstraction for the lemma must be the first variable to which the

lemma is applied. Otherwise, the proof may no longer be correct, since another term used in the place of α may have different assumptions than those of α. Given this condition and the scoping discipline, we know exactly which citations need to change: those of the form $L_i[j]\ \alpha$ that appear in the π_is or in τ.

The **(split)** rule takes a list of lemma definitions and separates them into two sets of definitions, one in the same place and one nested in a new let expression within the in part of the original let. The proofs of the lemmas do not change at all, so no citations need to change. The **(merge)** rule is the inverse of the **(split)** rule.

The **(rename)** rule changes the name of a single variable. The side condition (#) is that the new name β must not occur anywhere in π. This corresponds to α-conversion.

Soundness for the proof system requires that a sequence of applications of the rules transforms a proof term of a type φ into a new proof term of a type ψ that is equivalent modulo first-order equivalence. Let $\pi \Rightarrow \tau$ mean that the proof term τ is derivable from π using our proof rules in one step.

Theorem 2. *If $\pi \Rightarrow \tau$ and $\Gamma \vdash \pi : \varphi$, then $\Gamma \vdash \tau : \psi$, where φ and ψ are equivalent modulo first-order equivalence.*

Proof. The proof is by induction on the proof terms.

In order to prove the cases for **(generalize)**, we need a couple lemmas about substitution. We state the lemmas as meta-typing rules.

Lemma 2.

$$\frac{\Gamma, p : \varphi_p, L : \varphi \vdash \tau : \psi}{\Gamma, p : \varphi_p, L : \varphi_p \to \varphi \vdash \tau[L/L\ p] : \psi}$$

where $L = \pi$ does not appear in τ.

Lemma 3.

$$\frac{\Gamma, L : \varphi \vdash \tau : \psi}{\Gamma, L : \forall x.\varphi \vdash \tau[L/L\ x] : \psi}$$

where $L = \pi$ does not appear in τ.

Proof. The proof for both lemmas is by induction on proof terms.

We need similar lemmas for the **(specialize)** rule as well. The details of the proof are omitted due to space constraints. It is interesting to note, however, that the proof of soundness demonstrates that the types for let expressions and our environment $\mathcal{L}, \mathcal{C}, \mathcal{T}$ are correct.

5 A Constructive Example

To demonstrate the use of the proof rules, we develop the proofs of (8) and (7). Recall, we wish to prove

$$\forall a, b, c, z.a = b \to a = c \to z \vee (a \wedge b) = z \vee (a \wedge c) \tag{14}$$

using the lemma

$$\forall x, y, z. x = y \to z \vee (x \wedge x) = z \vee (x \wedge y) \tag{15}$$

We use the following axioms

$$\mathsf{sym} : \forall x, y. x = y \to y = x \tag{16}$$
$$\mathsf{trans} : \forall x, y, z. x = y \to y = z \to x = z \tag{17}$$
$$\mathsf{cong}_\wedge : \forall x, y, z. x = y \to (z \wedge x) = (z \wedge y) \tag{18}$$
$$\mathsf{cong}_\vee : \forall x, y, z. x = y \to (z \vee x) = (z \vee y) \tag{19}$$

The library $\mathcal{L}$ initially contains all of our axioms. Until we need them, we omit both $\mathcal{L}$ and $\mathcal{C}$ for readability. We also omit term substitutions when performing cites.

First, we prove the lemma. By **(ident)**, we have

$$P : x = y \vdash P : x = y \tag{20}$$

We use **(tcite)** with the substitutions $[x/x, y/y, z/x]$ and **(assume)** to add

$$P : x = y \vdash \mathsf{cong}_\wedge : x = y \to (x \wedge x) = (x \wedge y) \tag{21}$$

Applying **(mp)** to (20) and (21) gives

$$P : x = y \vdash \mathsf{cong}_\wedge P : (x \wedge x) = (x \wedge y) \tag{22}$$

We use **(tcite)** with the substitutions $[x/x \wedge x, y/x \wedge y, z/z]$ and **(assume)** to add

$$P : x = y \vdash \mathsf{cong}_\vee : (x \wedge x) = (x \wedge y) \to z \vee (x \wedge x) = z \vee (x \wedge y) \tag{23}$$

Applying **(mp)** to (22) and (23) gives

$$P : x = y \vdash \mathsf{cong}_\vee \mathsf{cong}_\wedge P : z \vee (x \wedge x) = z \vee (x \wedge y) \tag{24}$$

Now we apply **(discharge)** to (24) to get

$$\vdash \lambda P. \mathsf{cong}_\vee \mathsf{cong}_\wedge P : x = y \to z \vee (x \wedge x) = z \vee (x \wedge y) \tag{25}$$

We can use the **(collect)** rule to add (25) to our current term, given it the name lem. Our entire state is

$$\mathcal{L}; \mathsf{lem} = \lambda x \lambda y \lambda z \lambda P. \mathsf{cong}_\vee \mathsf{cong}_\wedge P : \forall x, y, z. x = y \to z \vee (x \wedge x) = z \vee (x \wedge y);$$

Now we start on the proof of the theorem. First we use **(ident)** to add the task

$$Q : a = b \vdash Q : a = b \tag{26}$$

Next, we use **(lcite)** with the substitutions $[x/a, y/b, z/z]$ and **(assume)** to get our lemma from the current term

$$Q : a = b \vdash \mathsf{lem} : a = b \to z \vee (a \wedge a) = z \vee (a \wedge b) \tag{27}$$

Applying **(mp)** to (26) and (27) gives

$$Q : a = b \vdash \mathsf{lem}\ Q : z \vee (a \wedge a) = z \vee (a \wedge b) \tag{28}$$

We now use **(cite)** with the substitutions $[x/z \vee (a \wedge a), y/z \vee (a \wedge b)]$ and **(assume)** to introduce

$$Q : a = b \vdash \mathsf{sym} : z \vee (a \wedge a) = z \vee (a \wedge b) \to z \vee (a \wedge b) = z \vee (a \wedge a) \tag{29}$$

Applying **(mp)** to (28) and (29) gives

$$Q : a = b \vdash \mathsf{sym}\ (\mathsf{lem}\ Q) : z \vee (a \wedge b) = z \vee (a \wedge a) \tag{30}$$

Next, we use **(ident)** to introduce

$$R : a = c \vdash R : a = c \tag{31}$$

Next, we use **(lcite)** with the substitutions $[x/a, y/c, z/z]$ and **(assume)** to get our lemma from the current term again

$$R : a = c \vdash \mathsf{lem} : a = c \to z \vee (a \wedge a) = z \vee (a \wedge c) \tag{32}$$

Applying **(mp)** to (31) and (32) gives

$$R : a = c \vdash \mathsf{lem}\ R : z \vee (a \wedge a) = z \vee (a \wedge c) \tag{33}$$

Applying **(tcite)** with the substitutions $[x/z \vee (a \wedge b), y/z \vee (a \wedge a), z/z \vee (a \wedge c)]$ allows us to add

$$\vdash \mathsf{trans} : z \vee (a \wedge b) = z \vee (a \wedge a) \to z \vee (a \wedge a) = z \vee (a \wedge c) \to z \vee (a \wedge b) = z \vee (a \wedge c) \tag{34}$$

Applying **(assume)** to (30), (33), and (34) gives

$$Q : a = b, R : a = c \vdash \mathsf{sym}\ (\mathsf{lem}\ Q) : z \vee (a \wedge b) = z \vee (a \wedge a) \tag{35}$$

$$Q : a = b, R : a = c \vdash \mathsf{lem}\ R : z \vee (a \wedge a) = z \vee (a \wedge c) \tag{36}$$

$$Q : a = b, R : a = c \vdash \mathsf{trans} : (a \wedge b) = z \vee (a \wedge a) \to z \vee (a \wedge a) = z \vee (a \wedge c) \to z \vee (a \wedge b) = z \vee (a \wedge c) \tag{37}$$

Two applications of **(mp)** using (35), (36), and (37) gives

$$Q : a = b, R : a = c \vdash \mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ Q))\ (\mathsf{lem}\ R) : z \vee (a \wedge b) = z \vee (a \wedge c) \tag{38}$$

We use **(discharge)** on each assumption in (38) to get

$$\vdash \lambda Q.\lambda R.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ Q))\ (\mathsf{lem}\ R) : a = b \to a = c \to z \vee (a \wedge b) = z \vee (a \wedge c) \tag{39}$$

We can use the **(collect)** rule to add (39) to our current term, give it the name thm, and make lem a lemma by introducing a let expression. Our new $\mathcal{C}$ term is

$$
\begin{array}{l}
\mathsf{thm} = \\
\quad \mathsf{let\ lem} = \lambda x \lambda y \lambda z \lambda P.\mathsf{cong}_{\vee}\mathsf{cong}_{\wedge} P : \forall x, y, z.x = y \rightarrow z \vee (x \wedge x) = z \vee (x \wedge y) \\
\quad \mathsf{in} \\
\qquad \lambda a \lambda b \lambda c \lambda z \lambda Q.\lambda R.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ Q))\ (\mathsf{lem}\ R) : \forall a, b, c, z.a = b \rightarrow a = c \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow z \vee (a \wedge b) = z \vee (a \wedge c) \\
\quad \mathsf{end}
\end{array}
$$

At this point, we could apply **(publish)** to add thm to the library. However, we may first wish to make thm and lem use the same z. To do this, we apply **(reorder)** to the term several times to get

$$
\begin{array}{l}
\mathsf{thm} = \\
\quad \mathsf{let\ lem} = \lambda z \lambda x \lambda y \lambda P.\mathsf{cong}_{\vee}\mathsf{cong}_{\wedge} P : \forall z, x, y.x = y \rightarrow z \vee (x \wedge x) = z \vee (x \wedge y) \\
\quad \mathsf{in} \\
\qquad \lambda z \lambda a \lambda b \lambda c \lambda Q.\lambda R.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ Q))\ (\mathsf{lem}\ R) : \forall z, a, b, c.a = b \rightarrow a = c \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow z \vee (a \wedge b) = z \vee (a \wedge c) \\
\quad \mathsf{end}
\end{array}
$$

We now apply **(specialize)** to move λz to the front of the **let** expression

$$
\begin{array}{l}
\mathsf{thm} = \\
\quad \lambda z.\mathsf{let\ lem} = \lambda x \lambda y \lambda P.\mathsf{cong}_{\vee}\mathsf{cong}_{\wedge} P : \forall x, y.x = y \rightarrow z \vee (x \wedge x) = z \vee (x \wedge y) \\
\quad \mathsf{in} \\
\qquad \lambda a \lambda b \lambda c \lambda Q.\lambda R.\mathsf{trans}\ (\mathsf{sym}\ (\mathsf{lem}\ Q))\ (\mathsf{lem}\ R) : \forall z, a, b, c.a = b \rightarrow a = c \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow z \vee (a \wedge b) = z \vee (a \wedge c) \\
\quad \mathsf{end}
\end{array}
$$

6 Related Work

Several people have looked at the problem of proof reuse and library organization. Limiting the scope of variables and assumptions is handled by Isabelle's *locales*, which limit the use of a set of local variables and assumptions to a current theory [6, 7]. In fact, the system allows one to create nested locales and move them outward in the nesting, corresponding to our **(specialize)** rule. However, theorems themselves are not a part of these locales and cannot be moved in the same way; the library of theorems is still a flat structure, without a complete notion of scope for theorems.

Melis and Schairer have looked at proof reuse in formal software verification [8]. In their proofs, subgoals are often very similar, so the reuse of completed proofs is instrumental in reducing the time required to verify programs. They have a notion of a lemma, where a proof used in an earlier subgoal can be reused within later subgoals of the same proof. The system can attempt to detect

these similar proofs automatically or the user can specify them. However, the relation between these subgoals is never stored in the proof, so a later analysis of the proof would not reflect the fact that similar subgoals were found and reused. Moreover, lemmas are not stored or reusable in different theorems. Given the similarities within proofs, one can imagine that there would also be several similarities between proofs for which storage of some of the more fundamental lemmas could be justified.

Lorigo et al. have worked on applying WWW search techniques to obtain information about the structure of libraries of proofs and theorems. In [9], they describe how this can be used to find the structure of mathematical topics and categories of theorems in libraries depending on inter theorem usage. The approach is meant to be used with already existing libraries of formal mathematics, and work one way, in the sense that it gathers information from the library and presents it to the user, but does not re-order the theorems in the library itself into the discovered relationships. In contrast, our approach intrinsically groups related theorems and lemmas already during their proof and keeps them together unless specifically moved by the user.

7 Future Work

We see many benefits to an automated theorem prover using a library with such a formal hierarchical structure. First of all, we would expect the structure of the library to indicate which theorems are more closely related–theorems that use the same variables, assumptions, or lemmas would be grouped together in let expressions and share abstractions. Large mathematical libraries could naturally be broken down into smaller parts based on these groupings.

One can imagine several heuristics that could be improved by the structure of the library. A system could first look at citing lemmas currently in scope before searching the entire library. The number of lemmas in scope is likely to be smaller than the number of theorems. Heuristics that automatically detect similar subproofs and create lemmas from them should also be possible. Given the formal structure of proofs, finding shared lemmas is a form of common subexpression elimination. In discovering these lemmas automatically, the library takes on the structure natural to the theorems proven. It could also provide guidance to a user proving a new theorem, knowing that the current proof being worked on and other theorems already proven share a few lemmas.

Currently, we have a basic implementation of all of the operations in a system that works on Kleene algebra with tests [10]. The system, written in Java, has a command line interface that allows one to create, manipulate, and save proofs in a tree structure, which corresponds naturally to the let expressions and local scoping. We hope to add to the system the ability to view and manipulate the library as a figure, given that the tree structure lends itself well to direct graphical depiction. One would easily be able to see and to alter the relationship between theorems while their manipulations would be guided by a strong underlying formalism.

Acknowledgements

We are indebted to Dexter Kozen, Ganesh Ramanarayanan, and Stephen Chong for valuable ideas and comments. This work was supported in part by NSF Grant CCR-0105586 and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

1. Kreitz, C.: The Nuprl Proof Development System, Version 5: Reference Manual and User's Guide. Department of Computer Science, Cornell University. (2002)
2. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V7.3. (2002) http://coq.inria.fr.
3. Wenzel, M., Berghofer, S.: The Isabelle System Manual. (2003)
4. Kozen, D., Ramanarayanan, G.: A proof-theoretic approach to knowledge acquisition. Technical Report 2005-1985, Computer Science Department, Cornell University (2005)
5. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry–Howard isomorphism. Available as DIKU Rapport 98/14 (1998)
6. Kammüller, F.: Modular reasoning in isabelle. In McAllester, D.A., ed.: CADE. Volume 1831 of Lecture Notes in Computer Science., Springer (2000) 99–114
7. Ballarin, C.: Locales and locale expressions in isabelle/isar. In Berardi, S., Coppo, M., Damiani, F., eds.: TYPES. Volume 3085 of Lecture Notes in Computer Science., Springer (2003) 34–50
8. Melis, E., Schairer, A.: Similarities and reuse of proofs in formal software verification. In: EWCBR. (1998) 76–87
9. Lorigo, L., Kleinberg, J.M., Eaton, R., Constable, R.L.: A graph-based approach towards discerning inherent structures in a digital library of formal mathematics. In: MKM. (2004) 220–235
10. Kozen, D.: Kleene algebra with tests. Transactions on Programming Languages and Systems **19** (1997) 427–443