BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Cap Petschulat

Thesis Title: Transparency in Formal Proof

Date of Final Oral Examination: 07 May 2009

The following individuals read and discussed the thesis submitted by student Cap Petschulat, and they also evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination, and that the thesis was satisfactory for a master's degree and ready for any final modifications that they explicitly required.

M. Randall Holmes, Ph.D.	Chair, Supervisory Committee
Marion Scheepers, Ph.D.	Member, Supervisory Committee
Andres Eduardo Caicedo, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by M. Randall Holmes, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

TRANSPARENCY IN FORMAL PROOF

by

Cap Petschulat

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Mathematics

Boise State University

August 2009

The thesis presented by *Cap Petschulat* entitled *Transparency in Formal Proof* is hereby approved.

M. Randall Holmes, Advisor	Date
Marion Scheepers, Committee Member	Date
Andrés Eduardo Caicedo, Committee Member	Date
John R. Pelton, Graduate Dean	Date

ABSTRACT

The oft-emphasized virtue of formal proof is correctness; a machine-checked proof adds greatly to our confidence in a result. But the rigors of formalization give rise to another possible virtue, namely clarity. Given the state of the art, clarity and formality are at odds: complexity of formalization obscures the content of proof. To address this, we develop a notion of proof strategies which extend the well-known notion of proof tactics. Beginning with the foundations of logic, we describe the methods and structures necessary to implement proof strategies, concluding with a proof-of-concept implementation in CheQED, a web-based proof assistant.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Formal proof is a formal endeavor. If we encounter difficulties working through a proof, we do not turn to formal proof assistants, for they would multiply our burdens. Contrast working through a difficult calculation, in which case computational tools are boons.

We believe these inequalities between proof and calculation are surmountable, and that we can develop proof tools for casual use. In pursuit of this goal, we develop a notion of *proof strategies* which extend the well-known notion of *proof tactics*. Strategies expose the operations of proof assistants and offer explanations where tactics offer only formal verification.

To support our development of proof strategies, we present CheQED, a web-based proof assistant for first-order logic. In this paper, we discuss CheQEDs logical foundations, sketch proofs of its correctness, and present proofs in propositional logic both with and without proof strategies.

While strategies are but a modest contribution to the goal of making proof assistants more friendly, we believe they are nevertheless representative of the sort of improvements that must be made to facilitate wider use.

1.1 Motivation

Mathematical logic follows a set of rules similar to mathematical computation in that both compound a small number of rules to achieve significant results. Yet computer algebra systems are more accessible and more generally useful than computer logic systems.

Ideally, a proof assistant would be useful to an undergraduate attempting to write homework proofs. Past calculus, most math books admit no method of checking one's work. While verifying proof correctness is, for the mature mathematician, a somewhat mechanical process, this should be all the more encouragement to automate.

Past undergraduate education, many cutting-edge results are of such complexity that their announcement is followed by a lengthy process of community verification. If computer assisted proof development was expected, the length of this process and its attendant confusion would be greatly reduced.

1.2 Focus

Much work in the field of formal proof is devoted to verifying particular results or exploring the ramifications of interesting abstractions and logics. If we survey the body of formalized mathematics across extant proof tools, we may well conclude that the only value of formalization is that it adds to our confidence that a proof or system is correct, albeit at great cost.

For financial reasons, hardware companies routinely formalize the correctness of circuits which enter mass production [1]. For greater certainty, Thomas Hales has begun The Flyspeck Project [2] to formalize his proof of the Kepler Conjecture. This proof, though accepted as correct by the handful of individuals who understand it, will gain wider acceptance once formalized.

Increased confidence is important, but it need not be the only beneficial end of formalization [3]. Mathematics requires communication. Every working mathematician learns to communicate in a peculiar, language which, because humans possess a limited tolerance for detail and tedium, omits large sections of argument. This is both a blessing and a curse. It allows communication more efficient than would otherwise be possible, but at the cost of potential confusion. Modern mathematicians function with only their own mental checks on steps they choose to omit. The absence of automated checks is completely normal and, worse, appears practically inevitable.

If proof assistants are to gain general acceptance, their authors must focus on making formal proofs comprehensible, or even, in some distant limit, illuminating and enjoyable to work with.

1.3 Transparency

With communication and clarity as goals, we observe that proof assistants seldom let us work at the optimal level of detail. At one extreme, simple arguments of the sort taught at a high school level often require great effort and care to formalize. At the other, automated provers dispatch complex arguments with a single command, obscuring points of rhetorical importance. Thus formalization involves both tedium and mystification.

Human authors excel at presenting proofs at an appropriate level of detail. Presentation involves complex decisions about the audience, subject matter, and purpose. It would be unreasonable to expect a computer program to make these decisions appropriately. The best we can hope for is a system that allows authors to write formalizations at whatever level of detail they choose, preserving the feel of casual proofs.

For the reader, proof assistants have the potential to do more than preserve the experience of casual proof. Every argument, down to the finest detail, is captured by formalization. Guided by the reader, an assistant can present proofs at any level of detail, even where the author chose to employ automated procedures for brevity. When reading a formalized proof, the phrase "it trivially follows" should lose its power to instill fear.

1.4 Outline

Chapter 2 establishes the foundations of logic and proof necessary to discuss proof assistants in detail. Chapter 3 describes some guarantees provided by proof assistants and outlines how they might be implemented. Chapter 4 introduces the notion of proof strategies and shows how they make formalized proof more accessible. Chapter

5 presents CheQED, a proof assistant and testing ground for many of the ideas in this thesis.

Chapter 2

FOUNDATIONS

Our goal is to explore how computers can aid proof development. It is important, then, to be very clear about the contents of a proof and the tools for constructing them. The development in this chapter is an amalgam of developments from standard logic texts, cf. [4], [5], and [6].

2.1 Formulas

A **formula** is a finite sequence of symbols. A **syntax** defines a collection of symbols as well as rules for their combination. A formula which conforms to the rules of a given syntax is **well-formed**. We are generally concerned with well-formed formulas; unless stated otherwise, assume formulas are well-formed.

The following syntax, given in Backus-Naur Form [7], allows simple formulas for a fragment of propositional logic:

```
<atom> ::= "a" | "b" | ..."z"
<formula> ::= <atom>
            | "not" <formula>
            | <formula> "or" <formula>
```

According to this syntax, the following are well-formed formulas:

```
a
a or not a
not not a or not b
```

The following are not well-formed:

```
aa
not or b
```

A **semantics** gives meaning to the symbols in a formula. **Variables** may be assigned different meanings under different **interpretations**, while **constants**, have the same meaning under all interpretations. Given a semantics and an interpretation, that is, an assignment of values to variables, a formula may be judged true or false. A formula which is true under all interpretations is **valid**.

In our example fragment of propositional logic, we define a standard semantics by assigning atoms to be variable and other symbols to be constant. The usual logical interpretation of `not`, `or`, and parentheses give meaning to those constants. Given a formula with these semantics and an interpretation, we may judge a formula true or false exactly as intuition guides us: a formula is true if it consists solely of a true variable, or the negation of a false variable, and so on.

2.2 Sequents

We reason about sequents, rather than directly about formulas, because sequents more naturally represent the type of mathematical assertions we would like to make.

A **sequent** consists of two sequences of formulas, the **antecedent** and the **succedent**. If Γ and Δ are sequences of formulas, then $\Gamma \vdash \Delta$ denotes a sequent with antecedent Γ and succedent Δ. Under a particular interpretation, a sequent represents the following assertion: if every formula in the antecedent is true, then at least one formula in the succedent is true.

If a sequent holds for every interpretation, it is **valid**. A valid sequent is also called a **theorem**. We occasionally refer to formulas as theorems; this is justified by the observation that $\vdash A$ is a theorem if and only if A is valid.

In describing rules, we refer to specific formulas within sequents by ordinals. The first formula in either part of a sequent is written nearest the turnstile. In $A, B \vdash C, D$, B is the first formula of the antecedent and C is the first formula of the succedent.

2.3 Rules

Rules indicate conclusions we may draw from particular premises. The premises and conclusions of rules are written as patterns. When a collection of sequents matches the patterns of the premises, we may draw the conclusion given by appropriate substitutions into the conclusion pattern.

The standard notation writes the premises above the conclusion:

$$\frac{\text{Premise 1} \qquad \ldots \qquad \text{Premise n}}{\text{Conclusion}}\ \text{(Rule Name)}$$

A sequent is **derivable** from some sequents and a rule when it is the result of applying that rule to those sequents. More generally, a sequent is derivable if it is the result of repeatedly applying rules to some collection of sequents.

Rules with null premises are **axioms**. From axioms, we may derive sequents in the form of the conclusion apropos of nothing.

Rules which preserve validity are particularly important; these are deemed **sound**. If the premises of a sound rule are fulfilled by valid sequents, then the conclusion is a valid sequent as well. Soundness is a guarantee that the rule does something sensible; our goal is to produce valid sequents, and rules which are not sound do not aid us.

We are often interested in collections of rules, not just single rules. The criteria for sensibility in a collection are soundness, consistency, and completeness. A collection is sound exactly when each contained rule is sound. A collection is **consistent** if application of its rules cannot lead to a contradiction, and **complete** if every valid sequent is derivable from its rules.

2.4 Proofs

A **proof** is a tree with a sequent and a rule associated with each node [5]. The sequent at each node is the conclusion of an instance of the associated rule with the sequents at its child nodes as premises. The root of the tree is associated with a valid sequent if all rules are sound and all leaves of the tree are associated with axioms.

An **assumption** is a pseudo-rule with null premise and any sequent as its conclusion. If any leaf of a proof tree is an assumption, then the proof is **partial**. Assumptions are convenient notationally and in practice, but they are not proper rules. Similarly, partial proofs are not true proofs, though we must define them to allow the development of proofs over time.

As we develop proofs, we reason in two directions. **Forward reasoning** begins with sequents or axioms and applies rules to form new sequents. **Backward reasoning** begins with a sequent we believe to be valid, called the **goal**. We attempt to decompose the goal into **subgoals** along with a partial proof, the **justification.** If we verify that all subgoals are valid, the justification proves that the goal itself is valid.

2.5 Tactics

When reasoning backwards, it is useful to write procedures which discover subgoals and justifications. These procedures are **tactics** [8], and may be composed and extended variously. We write the names of tactics in fixed-width font: `prove_proposition`, for instance, is a tactic which proves any propositional tautology.

Since the entire logical basis of a system is contained in its primitive rules, tactics cannot introduce logical errors. Erroneous tactics may fail to discover proofs, but they never find erroneous proofs.

2.6 Classical First-Order Logic

This section presents a collection of rules for classical first-order logic [9]. The collection omits explicit rules for several common propositional operators, though it remains complete as these operators may be formulated in terms of the given operators, and their related rules may be derived from the given rules.

Rules are either logical or structural. Logical rules address the use of logical operators. Structural rules address sequents as pairs of sequences of formulas, allowing manipulation of the sequences without changing the content of any individual formula. For example, order within the sequences does not matter logically, but does matter when applying rules literally, and so structural permutation rules allow us to reorder formulas within a sequent. When casually reasoning about the logical rules, we often apply structural rules unconsciously, but we present them here explicitly for completeness.

In the following, upper-case Roman letters denote formulas while upper-case Greek letters denote sequences of formulas. $A[x]$ denotes that x is some variable of interest in formula A, while $A[z/x]$ denotes the formula A with all free instances of x replaced by z. x denotes any variable, while y denotes a variable which does not appear elsewhere in the sequent. t denotes an arbitrary formula.

$$\frac{}{A \vdash A}\ \text{(Axiom)}$$

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\ \text{(Antecedent Negation)}$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta} \text{ (Succedent Negation)}$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \text{ (Antecedent Disjunction)}$$

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta} \text{ (Succedent Disjunction)}$$

$$\frac{\Gamma, \forall x A[x], A[t/x] \vdash \Delta}{\Gamma, \forall x A[x] \vdash \Delta} \text{ (Antecedent Universal)}$$

$$\frac{\Gamma \vdash A[y], \Delta}{\Gamma \vdash \forall x A[x/y], \Delta} \text{ (Succedent Universal)}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \text{ (Antecedent Weakening)}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta} \text{ (Succedent Weakening)}$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} \text{ (Antecedent Contraction)}$$

$$\frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta} \text{ (Succedent Contraction)}$$

$$\frac{\Gamma, A, \Sigma \vdash \Delta}{\Gamma, \Sigma, A \vdash \Delta} \text{ (Antecedent Permutation)}$$

$$\frac{\Gamma \vdash \Delta, A, \Pi}{\Gamma \vdash A, \Delta, \Pi} \text{ (Succedent Permutation)}$$

2.7 Soundness

We show that several of our rules are sound. We present only a selection as proofs of soundness tend to be similar; in fact, all our rules are sound. For propositional rules, we also present proofs of reversibility. Sequent rules are not generally reversible, but we require reversibility for our later proof of completeness. Though not relevant to the proofs in propositional logic, our quantifier rules are reversible as well.

2.7.1 Antecedent Negation

We show that the Antecedent Negation rule is sound.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta} \text{ (Antecedent Negation)}$$

We will show that if the premise $\Gamma \vdash A, \Delta$ is valid, then the conclusion $\Gamma, \neg A \vdash \Delta$ is also valid.

Assume that the premise is valid and fix an interpretation. If the premise holds vacuously, that is, some formula in Γ is false, then the conclusion also holds vacuously.

In the more interesting case, all formulas in Γ are true. Since the premise is valid, then either A is true or some formula in Δ is true. First assume A is true. Then $\neg A$ is false, so the conclusion $\Gamma, \neg A \vdash \Delta$ holds vacuously. Next assume that A is false so that some formula in Δ must be true. $\neg A$ is then true, as is, by assumption, every formula in Γ, so the conclusion, $\Gamma, \neg A \vdash \Delta$, holds.

Since the conclusion holds for our arbitrarily fixed interpretation, it is valid.

The Antecedent Negation rule is also reversible, that is, if the conclusion is valid, then the premise is also valid.

Assume the conclusion is valid and fix an interpretation. If the conclusion holds vacuously, then either $\neg A$ is false or some formula in Γ is false. If some formula in Γ is false, then the premise also holds vacuously. If every formula in Γ is true and $\neg A$ is false, then A is true. Since A appears in the succedent of the premise, the premise holds.

If the conclusion holds but not vacuously, then every formula in Γ and some formula in Δ is true. This remains the case in the premise, and so the premise holds.

2.7.2 Antecedent Disjunction

We show that the Antecedent Disjunction rule is sound.

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \lor B \vdash \Delta} \text{(Antecedent Disjunction)}$$

We will show that if the premises $\Gamma, A \vdash \Delta$ and $\Gamma, B \vdash \Delta$ are valid, then the conclusion $\Gamma, A \lor B \vdash \Delta$ is also valid.

Assume that the premises are valid and fix an interpretation.

If some formula in Γ is false, then some formula in the conclusion's antecedent is false, and so the conclusion holds vacuously.

Now assume all formulas in Γ are true. If A and B are both false, then $A \lor B$ is also false, so the conclusion holds vacuously. If either of A or B is true, then $A \lor B$ is true, and so the conclusion cannot hold vacuously; we must show that something in its succedent is true. Without loss of generality, assume A is true. Then the validity of the premises gives us that some formula in Δ is true, so some formula in Δ must be true. Thus the conclusion holds and the rule is sound.

The Antecedent Disjunction rule is also reversible, that is, if the conclusion is valid, then both premises are also valid.

Assume the conclusion is valid and fix an interpretation. If the conclusion holds vacuously, then some formula in Γ is false or $A \lor B$ is false. If some formula in Γ is false, then both premises hold vacuously. If $A \vee B$ is false, then both A and B must be false, so again both premises hold vacuously.

If the conclusion holds but not vacuously, then $A \vee B$ is true and some formula in Δ is true. Since the succedent of both premises is the same, we need only consider the antecedents. If A is true, then the premise involving A holds. If A is false, then the premise involving A holds vacuously. The case for B being symmetric, we have shown that the rule is reversible.

2.7.3 Antecedent Universal

We show that the Antecedent Universal rule is sound.

$$\frac{\Gamma, \forall x A[x], A[t/x] \vdash \Delta}{\Gamma, \forall x A[x] \vdash \Delta} \text{(Antecedent Universal)}$$

Recall that x denotes any variable, while t denotes any term. Assume that the premise is valid and fix an interpretation.

If some formula in Γ is false, then both the premise and the conclusion hold vacuously. Similarly, if the formula $\forall x A[x]$ is false, then both the premise and conclusion hold vacuously. Assume that all formulas in Γ are true and that $\forall x A[x]$ is true.

If $A[t/x]$ is false, then certainly $\forall x A[x]$ is false, for t itself witnesses a value of x for which A does not hold. This contradicts our assumption and thus cannot occur.

If $A[t/x]$ is true, then some formula in Δ is true. Since the succedent is the same in the premises and the conclusion, the conclusion holds.

2.7.4 Succedent Universal

We show that the Succedent Universal rule is sound.

$$\frac{\Gamma \vdash A[y], \Delta}{\Gamma \vdash \forall x A[x/y], \Delta} \text{(Succedent Universal)}$$

Recall that x denotes any variable, while y denotes any variable which does not appear elsewhere in the sequent. Assume that the premise is valid and fix an interpretation.

If some formula in Γ is false, then both the premise and the conclusion hold vacuously. Assume that all formulas in Γ are true.

If some formula in Δ is true, then both the premise and conclusion hold. Otherwise, $A[y]$ must be true.

y is a variable which appears in A but nowhere else in the sequent. The assumption that the premise is valid tells us that the premise holds under any interpretation, that is, under the assignment of any value to y. Since y does not appear elsewhere, it does not appear in Γ or Δ and thus the value of y does not affect the truth of formulas therein. If we consider interpretations which hold all other assignments constant while varying only in the value of y, then the validity of the premises tells us exactly that $A[y]$ holds for any value of y, so $\forall x A[x/y]$ is true.

2.8 Practicalities

Primitive rules capture the full logical core of the system. **Derived rules** are convenient compositions of primitive rules that do not affect the logical power of the system. Primitive rules are often unpleasant to use for any significant proof task, and so derived rules allow proofs to proceed through fewer steps, and allow steps to embody more recognizable, significant operations.

Many tactics find subgoals for goals matching the conclusions of rules. We denote such tactics by naming them after the corresponding rule. Thus `antecedent_negation` is a tactic which finds subgoals for statements with negation in the antecedent, corresponding to the Antecedent Negation rule.

`antecedent_negation` applies only to goals in which the first formula of the antecedent is the negation of some other formula (recall that the first sequent is nearest the turnstile); if this condition is not met, the tactic fails and does not advance the goal. If the goal is of the form $\Gamma, \neg A \vdash \Delta$, application of `antecedent_negation` produces a subgoal of $\Gamma \vdash A, \Delta$ with the Antecedent Negation rule as its justification.

Chapter 3

PROOF ASSISTANCE

A **proof assistant** is a computer program which aids in the construction of proofs. While the degree and kind of aid available through assistants varies wildly from assistant to assistant, most provide the following guarantees and services:

1. The assistant guarantees that formulas and statements entered by the user are well-formed.

2. The assistant guarantees that theorems arise only from the application of primitive rules.

3. The assistant provides a means for the user to write derived rules and tactics.

4. When reasoning backwards, the assistant tracks the subgoals and justifications necessary to validate the goal.

5. The assistant provides a means to explore formalized theorems and proofs.

When describing interactions with an assistant, it is useful to provide examples. We write these examples in an ad-hoc notation similar to that implemented by console-based assistants.

Interactions with the assistant appear in fixed-width font. Lines beginning with `>` represent user input, while the rest represent the assistant's output. Many sessions begin with the `goal` command, which sets the goal to which the assistant applies subsequent tactics:

```
> goal "|- not A"
|- not A
> succedent_negation
A |-
...
```

3.1 Well-Formedness

In casual mathematics, we adhere to loose rules of syntax. So long as our formulas convey meaning intuitively, they are admissible to human readers. We have less flexibility when formalizing; while we may introduce abbreviations and alternate notations, we must describe, in terms the assistant understands, exactly what our symbols mean. If the assistant encounters a symbol or formula it cannot decipher, then it cannot provide any assistance; any attempt to interpret an unfamiliar formula introduces uncertainty and, likely, incorrectness.

To guarantee that formulas are well-formed, the assistant translates raw user input, stored as strings of characters, into data structures which indicate the role of each character or group of characters in a mathematical formula.

3.1.1 Representation

In choosing data structures to represent mathematical formulas, we might start from our theoretical development of logic and write structures which mimic mathematical objects directly. Constants, variables, logical operators, predicates, quantifiers, and any other relevant objects would each be represented by their own data structure.

Along with the data structures themselves, we must also define how the structures interact. Logical operators operate on logical formulas, not integers or sets. Quantifiers quantify over variables, not predicates. With each new structure comes a full set of rules to define how it interacts with all other structures.

Introducing a new structure for each bit of notation, we quickly overwhelm ourselves with a tangled web of structures and interaction rules. To manage this complexity, we might seek a unifying abstraction, a system with only a few structures and rules which can, nevertheless, represent all the objects we wish to represent.

3.1.2 Lambda Calculus

The lambda calculus provides a particularly straightforward and compact representation for formulas. Based on the notion of function application, the lambda calculus defines only four basic structures [10]:

1. Constants: 1, π
2. Variables: x, $area$

3. Combinations: (fx), $(f(gx)y)$

4. Abstractions: $\lambda x(fxx)$

Variables and constants represent values which can and cannot change according to their interpretation, respectively. Combinations represent function application: (fx) is the application of the function f to the variable or constant x. Abstractions represent function definition: $\lambda x(fxx)$ defines a function of one variable, x, which passes x twice to f, a function of two variables.

Currying

Combinations represent function application for functions of one variable. To represent functions of higher arity, we employ a **curried** form. In standard mathematical notation, $f(x, y)$ is the application of the binary function f to the variables x and y; f operates on the ordered pair (x, y). In the lambda calculus, we encode f as a function of one variable, returning another function of one variable, which is then applied to the second variable: $((fx)y)$. With this understanding, we write (fxy) as shorthand for the curried version. When we discuss types in Section 3.1.3, our function types reflect this; f has type $\alpha \to \beta \to \gamma$ instead of $(\alpha \times \beta) \to \gamma$, eliminating the need for ordered pairs or cartesian product types.

Examples

Constants and variables in the lambda calculus map directly to constants and variables in formulas, so we will not belabor this explanation with specific examples.

Logical operators are encoded as constants: thus $\vee$ is a constant. Formulas involving logical operators are encoded as function applications, where the operator itself is regarded as a function: $A \vee B$ is thus $(\vee AB)$.

Quantifiers require a slightly more complex encoding. Quantifiers, like logical operators, are themselves constants: thus $\forall$ is a constant. To capture the notion of a bound variable, formulas involving quantifiers are encoded as function applications taking an abstraction as an argument: $\forall x \phi(x)$ is thus $(\forall(\lambda x \phi(x)))$.

3.1.3 Types

The lambda calculus provides data structures for formulas. An extension to the lambda calculus, the simply-typed lambda calculus, allows us to express the rules for the interactions of these structures.

The simply-typed lambda calculus extends the basic lambda calculus by attaching a type to each term.

The syntax of types is defined recursively:

$$\tau ::= \alpha \tag{3.1}$$

$$\mid \tau \rightarrow \tau \tag{3.2}$$

α represents an atomic type, while τ represents any type; note that the first and second instances of τ in $\tau \rightarrow \tau$ need not represent the same type. Types of the form $\tau \rightarrow \sigma$ represent functions taking an argument of type τ and returning a value of type σ.

If τ is a type and x is a term in the lambda calculus, we write $x : \tau$ to assert that x has type τ.

Examples

Constants and variables representing constants and variables in formulas have atomic types. If A is a variable representing a proposition, it has type *proposition*.

Logical operators have function types indicating they operate on propositional formulas and have themselves a propositional value. Thus $\vee$ has type *proposition* $\rightarrow$ *proposition* $\rightarrow$ *proposition*.

Quantifiers have function types that indicate they operate on abstractions and have a propositional value. When working with set theory, we might define $\forall$ to have type (*set* $\rightarrow$ *proposition*) $\rightarrow$ *proposition*.

3.1.4 Type Checking

The rules for typing superficially resemble our rules for first order logic. Indeed, they are given in quite the same spirit, and we could implement them as rules in our system. At present, however, we discuss these rules as they are implemented in the proof assistant software itself for the purpose of ensuring that formulas are well-formed, not for the purpose of developing proofs directly.

Upper-case Greek letters represent collections of type associations. The sequent $\Gamma \vdash x : \tau$ represents the assertion that if all the type associations in Γ hold, then the type association $x : \tau$ also holds.

The rules for typing are as follows [11]:

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \text{(Axiom)}$$

$$\frac{\Gamma, x : \sigma \vdash y : \tau}{\Gamma \vdash (\lambda x.y) : \sigma \rightarrow \tau} \text{(Abstraction)}$$

$$\frac{\Gamma \vdash x : \sigma \rightarrow \tau \qquad \Gamma \vdash y : \sigma}{\Gamma \vdash (xy) : \tau} \text{(Combination)}$$

A formula is properly typed if its type may be derived from application of these typing rules, otherwise it is improperly typed. With a complete collection of proper type associations similar to those given in 3.1.3, properly typed formulas correspond precisely to well-formed formulas.

3.1.5 Type Inference

Given a typed term and a collection of type associations, we may use the rules of the previous section to determine whether the term is properly typed. In checking that a formula presented to the assistant is well-formed, however, we do not generally begin with a typed term. Instead we begin with only a term, absent type information.

Naively, we could require the user to provide type associations for all terms, but this is a significant labor unto itself. Our goal is to shift the burden of recognizing well-formed formulas to the proof assistant, not to further burden the user. And, in fact, we can do much better; given an untyped term, there is a procedure we can follow to infer types for the term and all its components. If the procedure succeeds, then the term is properly typed.

Note that type inference is, by our earlier definition, backwards reasoning. The procedure which infers types for terms is akin to a tactic, automating the steps to prove that a particular term has a particular type.

Hindley-Milner type inference involves two steps [12]. First, it generates a collection of type equations from our given term. These type equations arise from reading the type rules from bottom to top. If it encounters a combination (xy) for instance, it generates type equations which assert that the type of x is $\sigma \to \tau$ and the type of y is σ.

Once it has generated a collection of type equations, the inference procedure attempts to show that all type equations may be satisfied simultaneously. This process, called unification, is simple and quite fast. If unification succeeds, the term is properly typed. If it fails, it is not, and the inference procedure can provide information about the particular type equations which prevented unification.

Example

When performing type inference, we usually begin with known types for constant symbols. For symbols of unknown type, inference assigns type variables, denoted $\sigma_0, \ldots, \sigma_n$. Thus when inferring types for the formula `or a (not a)`, we begin with the following type assignments:

$$or : proposition \rightarrow proposition \rightarrow proposition \tag{3.3}$$

$$not : proposition \rightarrow proposition \tag{3.4}$$

$$a : \sigma_0 \tag{3.5}$$

From these and inspection of the formula itself, we generate the following type equations:

$$\sigma_0 = proposition \tag{3.6}$$

$$proposition = proposition \tag{3.7}$$

The first equation arises from the application of a as the first argument to the function *or*. a has type σ_0 and the argument to *or* has type *proposition*, so the types must be equivalent. In the second equation, we first note that the function *not* is applied to a variable, so the resulting type is the return type of *not*, that is, *proposition*. Since this result is applied as the second argument to *or*, which has type *proposition*, the two types must be equal.

These equations clearly unify; assigning $\sigma_0 = \mathit{proposition}$ leads to no contradiction. a is thus a proposition and the formula is well-typed.

In a malformed formula like `or a not`, we would instead generate the following contradictory type equations:

$$\sigma_0 = \mathit{proposition} \tag{3.8}$$

$$\mathit{proposition} = \mathit{proposition} \rightarrow \mathit{proposition} \tag{3.9}$$

In this case, the *proposition* cannot be unified with *proposition* $\rightarrow$ *proposition*, confirming that the formula is malformed.

3.1.6 Substitution

Once we've constructed well-formed formulas, we often need to manipulate them. Rules for propositional logic only require us to extract formulas linked by logical operators. In terms of the lambda calculus representation, this corresponds to inspecting the operator and operand of a combination and creating a new formula from the operand alone. Rules for quantifiers, however, require that we substitute arbitrary formulas for variables and vice versa.

Proper substitution requires some care. Consider the formula $\forall y(y = x)$. Assuming a reasonable definition of equality and a model which contains more than one object, the original formula does not hold for any particular x. Substituting z for x does not change this, nor should any semantically correct substitution. But if we blindly substitute y for x, the formula becomes tautologous: $\forall y(y = y)$.

Bound variables are semantically different from free variables in that their precise names are immaterial. $\forall y(y = x)$ is semantically equivalent to $\forall z(z = x)$. When substituting into a formula involving bound variables, then, we take care to rename the bound variable when it occurs within the formula being substituted in. In our example, a semantically sensible renaming yields $\forall y'(y' = y)$. For formulas involving several bound variables, this renaming must be performed recursively to ensure that renaming does not, itself, capture other bound variables.

3.2 The Primacy of Primitives

As we formalize proofs in a proof assistant, we inevitably wish to prove successively more complex results. Where initially we prove results in logic, later we wish to prove results in set theory, arithmetic, calculus, or any other field. If we are forced to write proofs about calculus using the primitive rules of logic, we will quickly become frustrated.

It is tempting to allow the user to write new rules appropriate to each domain of mathematics and use them as primitive rules. This is undesirable, however, if only because it is difficult to write primitive rules that are truly correct. Furthermore, if proofs in different domains relied on different primitive rules, it would be very difficult to ensure the correctness of proofs based on two separate collections of primitive rules.

Rather than allow the introduction of arbitrary primitives, we prefer to introduce a minimal collection of primitive rules and a mechanism for deriving compound rules.

Our primitive rules, as presented in 2.6, allow the construction of simple formulas involving negation and disjunction. We would like to introduce rules for other standard logical operators, like conjunction.

A conjunction rule for the antecedent appears as follows:

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, \neg(\neg A \vee \neg B) \vdash \Delta}\text{(Antecedent Conjunction)}$$

This conjunction rule is permissible because we implement it purely in terms of primitive rules. Antecedent Conjunction is equivalent to the following:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma, A, B \vdash \Delta}{\Gamma, A \vdash \neg B, \Delta}\text{(Succedent Negation)}}{\Gamma \vdash \neg A, \neg B, \Delta}\text{(Succedent Negation)}}{\Gamma \vdash \neg A \vee \neg B, \Delta}\text{(Succedent Disjunction)}}{\Gamma, \neg(\neg A \vee \neg B) \vdash \Delta}\text{(Antecedent Negation)}$$

3.3 Proof State

Proofs are naturally expressed as trees of rules and sequents, but it is difficult to interact with trees when developing proofs. Most proof assistants present a textual interface, requiring proofs to be written in a manner similar to computer programs. Some provide an interactive proof environment, similar to an interpreter for a programming language.

When we develop proofs by reasoning backwards, we begin with a single formula, the goal. By applying tactics, we decompose the goal into subgoals. In turn we decompose those subgoals again, until finally the proof is complete.

In a textual interface, active goals are usually represented in an array. Of these goals, one is marked active; tactics are applied to the active goal. A proof session begins with one element in the goal array. As tactics break goals into subgoals, they replace the goals in the goal array with subgoals. If a tactic completely proves a particular goal, it is eliminated from the goal array. The user may change the active goal at will, and some assistants allow single commands to apply tactics across multiple goals.

In a graphical interface, it is easier to present the entire proof tree while highlighting active goals. In CheQED, for instance, the user applies tactics to a particular goal simply by clicking on buttons displayed near the subgoals.

3.4 Exploration

At minimum, a proof assistant must track theorems which it has previously verified. Proofs of significant theorems build upon lemmas and theorems previously proved. In order to prove interesting results, an assistant must be able to reference and combine previous results.

To provide a better presentation of previously verified results, the assistant can also record the tactics and rules invoked in the construction of each proof. This usually takes the form of a record of steps the user took in creating the formalization, sometimes sanitized to omit missteps or redundancies.

At best, the assistant can present a full, structured record of its operations. This includes the tactics and rules invoked in the construction of a proof, but extends the record of user steps with a record of steps the assistant took, as well. The possibilities afforded by this full record are discussed in the next chapter.

Chapter 4

TRANSPARENCY

Mathematical development, as all development, is characterized by phases of learning and creating. When we learn of a new result, we study it and seek to understand its proof. When we create a new result, we use known results as foundations, confident that they are correct and unburdened by concern for their proof.

Proof assistants are often employed in the creation phase, as their guarantees of correctness let us advance new results with confidence. But less thought is given to the role of assistants in the learning phase. The internal operations of assistants are often opaque, and the explanatory power of formal proof is often weak. Crudely, the goal of most formalization is to say, "I have formally verified this result," not "I have explained this result clearly."

There are exceptions: the Marcel proof assistant is designed as a teaching tool. To reduce the potential for confusion, Marcel eschews tactics entirely, leaving the user to guide the assistant at every step. This approach is effective in many cases but untenable for proofs of sufficient size. In what follows, we present a means by which a proof assistant may exploit the efficiency of tactics while retaining the transparency of assistants like Marcel.

4.1 Motivation

Most assistants include a tactic, which we call `prove_proposition`, to prove arbitrarily complex propositional goals. `prove_proposition` is possible for two reasons. First, the rules for propositional logic are complete, and so any tautology is guaranteed to admit a proof. Second, the tactic is guaranteed to halt both when then goal is valid, more importantly, when the goal is not valid. This last point is essential, as a similar tactic written to prove goals in first-order logic can not be guaranteed to halt for invalid goals.

The appeal of `prove_proposition` is obvious: letting the assistant automatically prove a goal is much less work than proving it ourselves. Consider the following session which verifies the law of the excluded middle in the absence of automation:

```
> goal "|- A or not A"
|- A or not A
> succedent_disjunction
|- A, not A
> succedent_permutation 2
|- not A, A
> succedent_negation
A |- A
> qed
done
```

The validity of the goal is obvious at first glance, and yet the assistant requires several tactic applications to produce a proof. One necessary tactic is even structural, further differentiating the formal proof from the casual proof.

If, instead, we use `prove_proposition`, our interactions with the tool better resemble our actual thought process:

```
> goal "|- A or not A"
|- A or not A
> prove_proposition
done
```

The `prove_proposition` tactic eliminates obvious steps, but it proves non-obvious goals just as well:

```
> goal "|- ((a and b) implies c) iff (a implies (b implies c))"
|- ((a and b) implies c) iff (a implies (b implies c))
> prove_proposition
done
```

The goal is indeed a theorem, which is good to know. In some situations, this may be all we want to know. But what if we want to see the details, to know how the tactic arrived at its proof? We would like to write the following and expose the automated steps:

```
> goal "|- A or not A"
|- A or not A
> prove_proposition
done
> expand
|- A or not A
succedent_disjunction
|- A, not A
prove_proposition
```

Unlike `prove_proposition`, most proof assistants do not implement the analog of `expand`. The remainder of this chapter explores why this is the case and presents an implementation of such a command.

4.2 Completeness

Our `prove_proposition` tactic halts when applied to any goal. If the goal is a tautology, the tactic will find a proof. The efficacy of the tactic establishes the completeness of our rules for propositional logic.

We define `prove_proposition` in pseudo-code as follows:

```
def prove_proposition:
  if axiom applies:
    axiom
  else if succedent_negation applies:
    succedent_negation
    prove_proposition
  else if succedent_disjunction applies:
    succedent_disjunction
    prove_proposition
  else if antecedent_negation applies:
    antecedent_negation
    prove_proposition
  else if antecedent_disjunction applies:
    antecedent_disjunction
    prove_proposition on first subgoal
    prove_proposition on second subgoal
  else if negation or disjunction is the nth formula in succedent:
    succedent_permutation n
    prove_proposition
  else if negation or disjunction is the nth formula in antecedent:
    antecedent_permutation n
    prove_proposition
```

```
else
  do nothing
```

If A is a valid propositional formula, then `prove_proposition` will find a proof of $\vdash A$ in finite time. Further, if A is not valid, the tactic will nevertheless halt in a finite number of steps.

Our proof proceeds by induction on the number of logical operators in the goal. If a goal contains no logical operators, then either the axiom rule applies or the goal is not a tautology. If the axiom rule applies, then the goal is proved and the tactic halts. If the axiom rule does not apply, then the goal is invalid: both sides of the goal sequent will be sets of propositional letters, and the two sets will be disjoint, so the interpretation assigning "true" to each propositional letter on the left and "false" to each propositional letter on the right witnesses the claimed invalidity.

Assume the result holds for all goals containing n or fewer operators. Let $\Gamma \vdash \Delta$ be a sequent containing $n+1$ operators. Either Γ or Δ contains at least one formula involving negation or disjunction. Using a single permutation tactic, we can ensure that such a formula appears in the first position in the sequent so that an appropriate operator tactic applies. After permutation, our sequent must match one of four cases:

$\Gamma, \neg A \vdash \Delta$

$\Gamma \vdash \neg A, \Delta$

$\Gamma, A \vee B \vdash \Delta$

$\Gamma \vdash A \vee B, \Delta$

In each case, application of the appropriate primitive tactic produces one or two subgoals each containing n operators. Since the hypothesis holds for each of these subgoals, it also holds for our present goal. Thus the tactic halts in all cases.

When `prove_proposition` operates on a goal which does not contain a tautology, it nevertheless produces subgoals which indicate assumptions which would make the goal valid.

While we have shown that `prove_proposition` halts, we have not yet shown that it will find a valid proof for any tautology. To show this we observe that the primitive rules for propositional logic are reversible; that is, as defined, a rule's conclusion holds if and only if its premises hold. When the tactic decomposes the goal into subgoals, the goal is valid if and only if the subgoals are valid. If the original goal is valid, all the subgoals must be valid, that is, solved by axioms, and so the partial proof produced by the tactic is in fact a complete proof. If the original goal is not valid, then some subgoal is not valid.

The sequent in an invalid subgoal with no remaining logical operators suggests an interpretation under which the original goal does not hold: if we assign all variables in the antecedent to be true, and all variables in the succedent to be false, then the original goal does not hold.

4.3 Tactical Myopia

Tactics are programs. Their inputs are goals, their outputs partial proofs. This is a wonderful abstraction when our aim is to create sophisticated tactics; the proof assistant places no restrictions on our implementation. We may choose to write the tactic in the same implementation language as the assistant, or in a different language entirely. We may write the tactic cleanly, or sloppily; so long as it finds a relevant partial proof, it is successful.

When our aim is to read through a proof, the abstract nature of tactics is a curse. If a tactic advances a proof through steps we don't immediately understand, we are left in the dark. At best, we can reconstruct the tactic's steps manually, or by reading its source code. More typically, we are just as lost as we would be without a proof assistant.

Consider the situation of a student learning a new proof in the traditional way, with a sketch of the proof and a teacher. When the student reaches an impasse, unable to connect one step to the next, the teacher might suggest an intermediate step or a new way to look at the problem. Were the teacher to respond, instead, with a detailed derivation of the goal from first principles and no further interpretation or assistance, the student would be understandably perplexed.

The teacher's detailed derivation is correct, but it is not terribly useful to the student, for it lacks explanatory power. Yet we find ourselves in a similar situation when we use tactics; their results are indisputable but nevertheless unhelpful.

As all good computer scientists know, all problems in computer science can be solved with another layer of abstraction. And so it is that we introduce the notion of a strategy to bring explanatory power to formal proofs.

4.4 Strategies

Tactics produce partial proof; what's needed for greater transparency is partial explanation. A **strategy** is a procedure which produces a tree of tactic and strategy applications, called a **plan**, appropriate to advancing a particular goal. As interpreted by the proof assistant, plans play the same role as tactics: they decompose a goal into subgoals and a justification. But plans differ from tactics in that they explicitly encode an explanation of how the justification is developed.

For the user reading a proof developed with strategies, the first impression is identical to that of a proof developed with tactics: the proof assistant displays exactly the steps the author used to write the proof. But for each step in the proof, the proof assistant can expand the proof to reveal the strategy's inner workings.

In a proof which invokes `prove_proposition` as a strategy, the user can expand that step to show what `prove_proposition` did as applied to the goal. We saw one example of this in 4.1, where `prove_proposition` applied to a disjunction invoked `succedent_disjunction` before recursively invoking itself. Of course, `prove_proposition` invokes different tactics as it is applied to different goals, and this is the value of `expand`: the user can see precisely how a strategy works as applied to a particular goal.

4.5 Implementation

We define the syntax of plans in terms of a single binary combinator, `branch`. The first argument is a strategy or tactic, and the second is a (possibly empty) list of strategies or tactics.

When the assistant applies a plan to a particular goal, it applies the first argument to the current goal. This yields some list of subgoals; its length must match the length of the second argument. If the list of subgoals is nonempty, the assistant applies the first strategy or tactic in its list to the first subgoal, the second strategy or tactic to the second subgoal, and so on.

In the example applying `prove_proposition` to the law of the excluded middle, expanding `prove_proposition` once produces the following plan:

```
branch(succedent_disjunction, [branch(prove_proposition, [])])
```

Since strategies produce plans and not tactics, in order to check a proof using strategies, the assistant must fully expand all plans so that they use only tactics. The previous example, fully expanded, appears as follows:

```
branch(succedent_disjunction,
 [branch(succedent_permutation,
   [branch(succedent_negation,
     [branch(qed, [])])])])
```

4.6 Discussion

The idea for strategies arose from a parallel with computer programming. When a program acts in a mystifying way, it is useful to run it in a debugger and trace its execution. Debuggers allow programmers to check their assumptions line-by-line through the program's source code. This is especially useful when certain parts of the program are run conditionally; the source code describes a range of possibilities, but the debugger shows exactly which possibility matches reality.

Tactics are programs, and so the idea of debugging naturally applies. Where debuggers provide a powerful and technical interface to a program's inner operations, plans and strategies attempt to expose proof operations more directly. This is possible largely because the operations necessary to construct a proof are simpler than the operations necessary to support general-purpose computation.

4.7 Related Work

The operation of tactics is usually opaque, in that they map goals to subgoals with no record of the tactics from which subgoals emerge. This is the case by default in HOL Light, for instance [13]. Proof recording is available in a rewrite of significant portions of the prover, and then only for a subset of possible HOL Light proofs.

MetaMath's web interface presents proofs with hyperlinks to relevant sub-proofs [14]. The MetaMath system, however, relies entirely on primitive rules, and so there is little control over the level of detail; all presentation occurs at the primitive level. The facility to use previously proved theorems offers a limited form of organization, but nothing like the flexibility afforded by strategies.

Chapter 5

CHEQED

5.1 Introduction

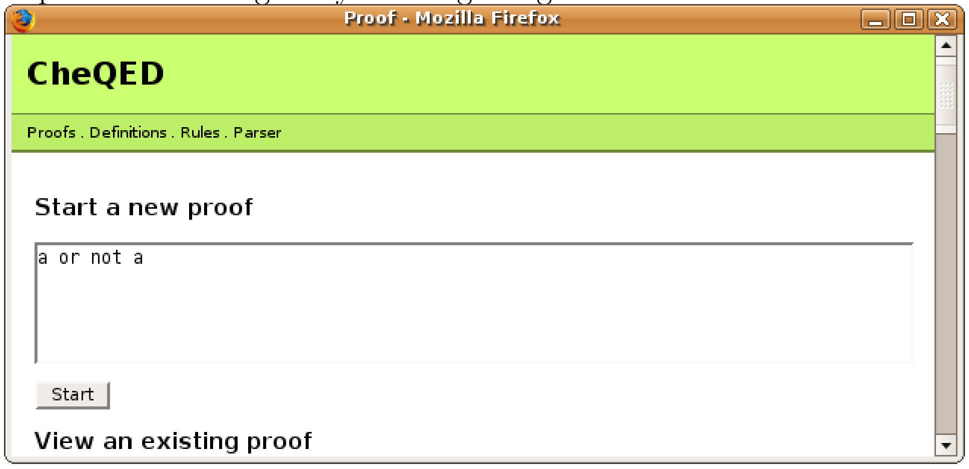
CheQED is the most mature of several proof assistants written in the course of developing this thesis. The first of these assistants began by mimicking the propositional parts of Marcel [15] and adding a tactic to prove tautologies automatically. Later assistants, inspired by HOL Light [13], moved from a naive representation of formulas to a representation based on the lambda calculus. CheQED builds on these ideas, but is intended to explore ideas related more to the user's experience with formal proof than the underlying mathematics.

User interaction with CheQED occurs through an HTML and JavaScript interface, accessible through any modern web browser. Users develop proofs through a mixture of mouse clicks and text entry, interacting directly with nodes of the proof tree.

5.2 Proofs in CheQED

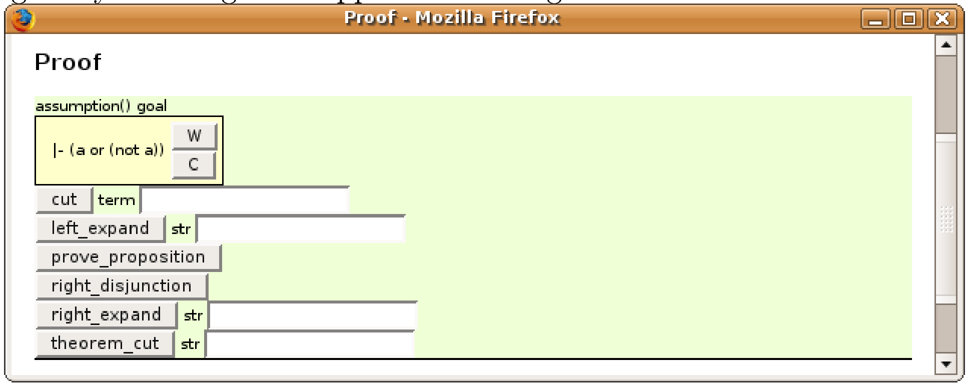
We again work through the example of Section 4.1, this time in CheQED, to illustrate interaction with the system and our implementation of proof strategies.

A proof session begins by entering the goal:

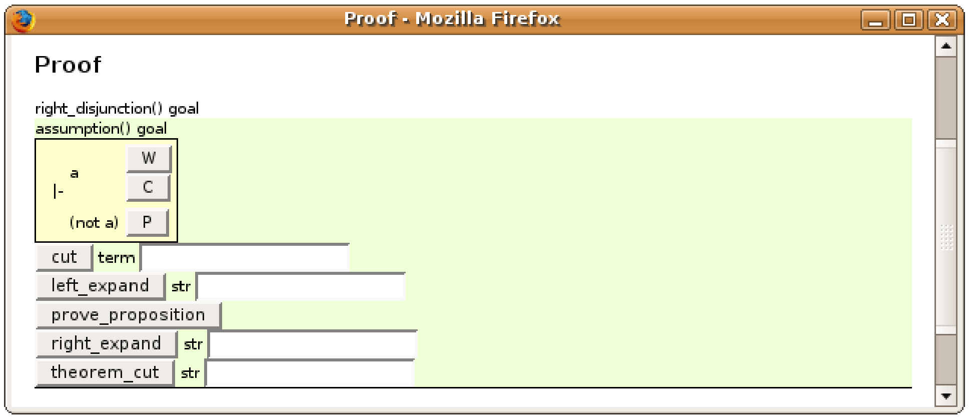


```
a or not a
```

Once the goal is parsed and type-checked, CheQED allows the user to advance the goal by choosing from applicable strategies:



Some strategies, like `cut`, require additional user input and thus provide a text input box. Others, like `right_disjunction`, require no additional input. If we click the button labeled `right_disjunction`, the tactic is applied and we are given a new goal:



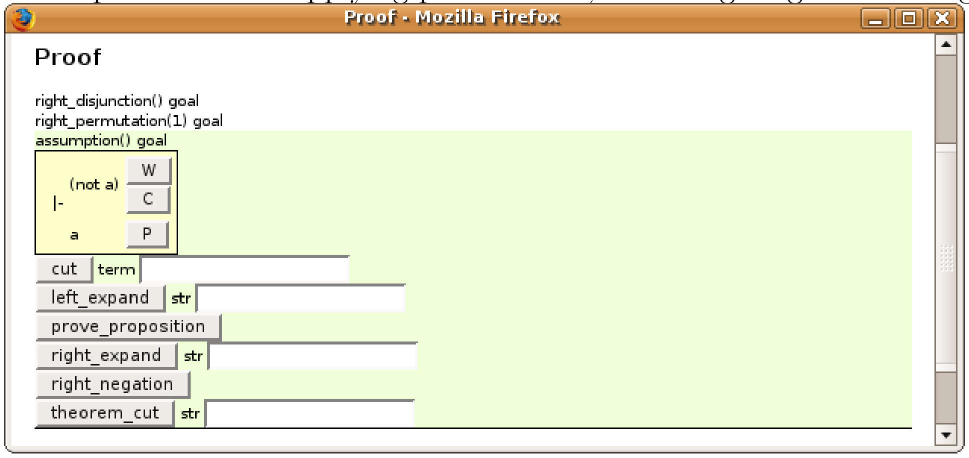
Proof - Mozilla Firefox

Proof

right_disjunction() goal
assumption() goal

a
|-
(not a)

Buttons next to the goal apply structural rules: `W` for weakening, `C` for contraction, and `P` for permutation. Applying permutation, we are again given a new goal:



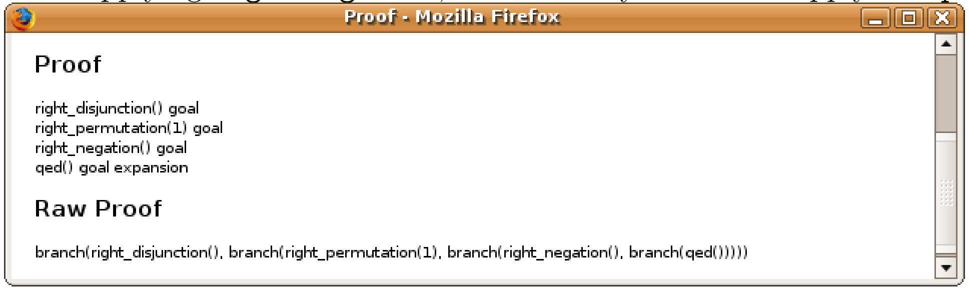
Proof - Mozilla Firefox

Proof

right_disjunction() goal
right_permutation(1) goal
assumption() goal

(not a)
|-
a

After applying `right_negation`, we are finally allowed to apply the `qed` strategy:



Proof - Mozilla Firefox

Proof

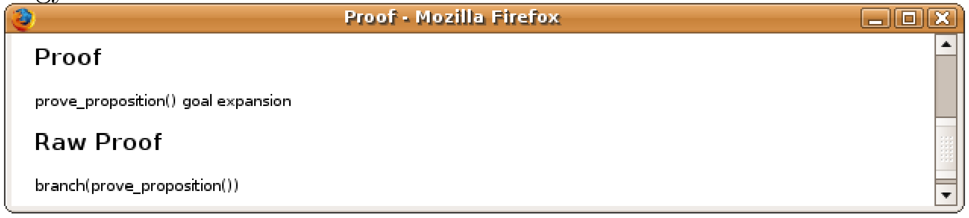
right_disjunction() goal
right_permutation(1) goal
right_negation() goal
qed() goal expansion

Raw Proof

branch(right_disjunction(), branch(right_permutation(1), branch(right_negation(), branch(qed()))))

`qed` finishes any proof in which identical formulas appear on each side of the goal; in more complex proofs, `qed` applies any necessary permutations before applying the primitive `axiom` tactic.

Of course, we could have finished this proof in one step by using the `prove_proposition` strategy:


Proof - Mozilla Firefox

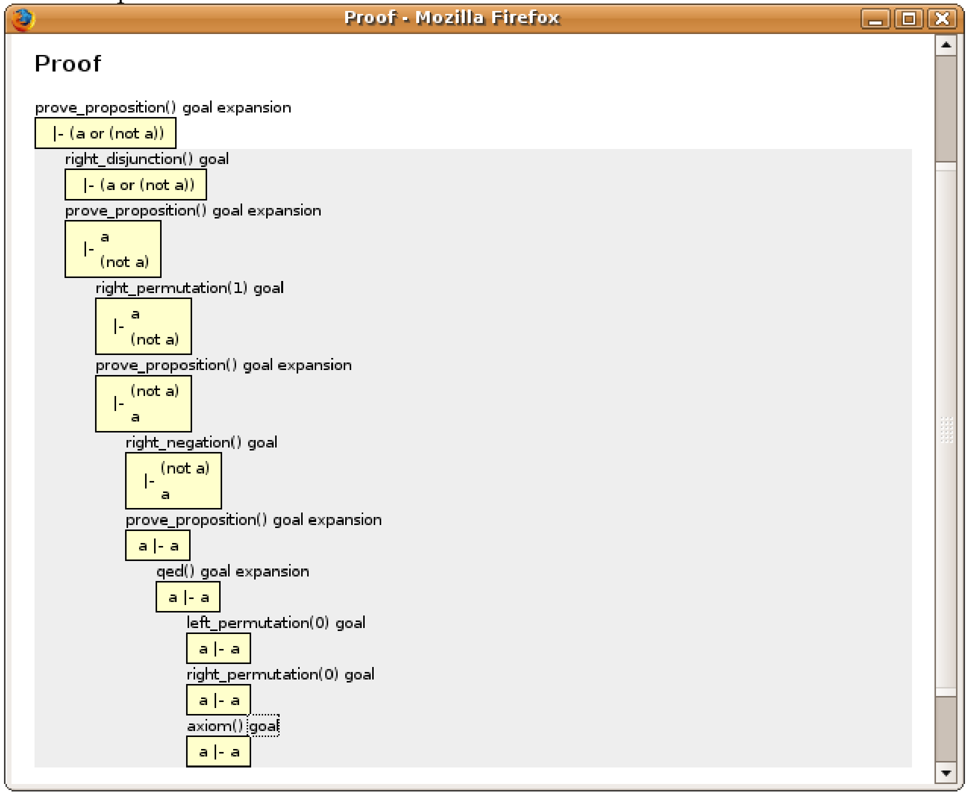
Proof

prove_proposition() goal expansion

Raw Proof

branch(prove_proposition())

Having finished the proof in one step, we may nevertheless unfold the plan and see the full proof:


Proof - Mozilla Firefox

Proof

```
prove_proposition() goal expansion
|- (a or (not a))
    right_disjunction() goal
    |- (a or (not a))
    prove_proposition() goal expansion
       a
    |-
       (not a)
        right_permutation(1) goal
           a
        |-
           (not a)
        prove_proposition() goal expansion
           (not a)
        |-
           a
            right_negation() goal
               (not a)
            |-
               a
            prove_proposition() goal expansion
            a |- a
                qed() goal expansion
                a |- a
                    left_permutation(0) goal
                    a |- a
                    right_permutation(0) goal
                    a |- a
                    axiom() goal
                    a |- a
```

5.3 Features and Shortcomings

CheQED is not a fully-functional proof assistant, in that it is frustrating to use for proofs of any appreciable size. It is, at present, a proof-of-concept for proof strategies.

That said, CheQED is the product of a number of design decisions not directly related to the implementation of proof strategies. We present these as a list of features:

- Formulas entered by the user are represented as terms of the simply-typed lambda calculus and appropriately type-checked.
- Primitive rules and tactics are distinguished from derived rules and strategies so that users cannot build derived rules or strategies which do anything beyond that permitted by primitive rules and tactics.
- Proofs are visually presented in an expandable tree; hidden steps taken by strategies are revealed by clicking on the appropriate part of the proof.
- User-written derived rules and strategies may introduce new notation by defining a translation to existing notation. The parser automatically adapts itself to new notation.
- New theories may be introduced as axiomatic formulas; formulas designated as axioms may be introduced into the antecedent of any sequent. The axioms of Zermelo set theory are included in this way.

For practical use, CheQED has a number of shortcomings. The most significant among these are as follows:

- Proofs are not linked together, so previously verified results cannot be invoked in subsequent proofs.

- There is no facility for including values in formulas, so proofs which would normally involve numerals or set-builder notation are awkward.
- Proof steps cannot be altered, so mistaken steps may require restarting the proof.
- Some lambda notation creeps into definitions, which can be confusing given that the logic exposed is otherwise classical first-order logic.

5.4 Comparison

CheQED was heavily influenced by our experience with other proof assistants. Initial versions closely resembled Marcel, though later versions diverge significantly. CheQED implements strategies and tactics for automated reasoning where Marcel intentionally omits them. CheQED's representation of terms in the lambda calculus was inspired by HOL Light, though the systems differ in that HOL Light formulates primitive inference rules for lambda calculus terms directly, whereas CheQED's rules operate on terms interpreted as formulas of first-order logic.

We chose first-order logic as the basis for CheQED because of its accessibility. From the beginning, our goals included discovering why proof assistants were not more user-friendly. Given that proof assistants implementing vastly different logics were nevertheless equally unfriendly, we judged that the choice of a nonstandard logic was not likely to solve problems of usability.

CheQED's logical foundations are minimal; beyond first-order logic, the basis for any significant mathematical theory must be provided, by the user, as axioms. CheQED supports axiom schemas to represent theories, like ZFC, with infinitely many axioms. Schemas are restricted in such a way that they are truly just schemas and not some form of higher-order logic. This sets CheQED apart from the other assistants described in this section; HOL Light implements a higher-order logic directly in its primitive rules, while Marcel implements the set theory NFU of [16], a variant of New Foundations, with a basis in the sequent calculus given in [17].

5.5 Implementation

CheQED is written in the Python programming language using the Django web framework. CheQED's parser is generated by PLY, a pythonic implementation of lex and yacc. Beyond this, all code, including that for formula representation, type checking, formula construction via primitive rules, proof checking, and strategies were written for CheQED by the author.

CheQED makes extensive use of Python's interpreted features. Derived rules and strategies may be written by the user in Python and loaded by CheQED at runtime. User-written extensions may take advantage of CheQED's parser and pattern-matching features naturally, with patterns in CheQED's formula notation (rather than native Python notation) when calling other procedures.

Chapter 6

CONCLUSION

We began with the assertion that formal proof, and proof assistants in particular, can be improved and made more accessible. As Maple and Mathematica regularly appear in present-day undergraduate curricula, so we believe proof assistants will appear in the future. Given the present state of assistants, the realization of this belief remains far off.

The enormity of the task that is writing a useful proof assistant along with significant proofs is hard to understate. Modern proof assistants, including HOL, Coq, and Isabelle, are the culmination of hundreds of man-years of work. Freek Wiedijk, compiling a list of formalization projects, notes many single proofs that may take years to formalize, assuming the use of an existing proof assistant [18].

Still, improvements to proof assistants themselves may significantly reduce the effort needed to write formalizations. One can easily imagine that an assistant of sufficient friendliness to be of use in general undergraduate education would also make the formalization of foundational texts straightforward, whereas formalizing a full text is presently a monumental undertaking.

In developing the notion of proof strategies, we have demonstrated one small improvement to the usability of proof assistants. Proof strategies were born of the idea that confusion leads to frustration, and specifically to frustration of a sort anathema to the enjoyable use of a proof assistant. While strategies do not make it easier to prove a particular result, they do make proofs easier to understand. From the perspective of a student or beginning user, understanding formal proofs is the first step to writing them. The easier it is to understand formal proofs, the more people, we believe, will want to write them.

REFERENCES

[1] Thomas C. Hales. *Formal Proof.* Notices of The AMS, Volume 55, Number 11, pp. 1370-1380.

[2] Thomas C. Hales. *Flyspeck: A Blueprint for the Formal Proof of the Kepler Conjecture.* `http://www.math.pitt.edu/~thales/papers/flypaper.pdf`, accessed 2009-03-18.

[3] Freek Wiedijk. *The QED Manifesto.* `http://www.cs.ru.nl/~freek/qed/qed.html`, accessed 2009-03-18.

[4] H.-D. Ebbinghaus, J. Flum, W.Thomas. *Mathematical Logic.* Springer, New York, 1994.

[5] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, Inc., Orlando, Florida, 1986.

[6] Randall Holmes. *Proofs, Sets, and Logic.* Manuscript. `http://math.boisestate.edu/~holmes/indstudy/proofsetslogic.pdf`, accessed 2009-03-18.

[7] Wikipedia article. *Backus-Naur Form.* `http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form`, accessed 200903-18.

[8] Lawrence C. Paulson. *Designing a Theorem Prover.* pp. 415-461 in Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures. Edited by S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum. Oxford University Press, New York, 1992.

[9] Wikipedia article. *Sequent calculus.* `http://en.wikipedia.org/wiki/Sequent_calculus`, accessed 2009-03-18.

[10] J. Roger Hindley, Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus.* London Mathematical Society Student Texts 1. Cambridge University Press, Cambridge, UK, 1986.

[11] H.P. Barendregt. *Lambda Calculi with Types.* pp. 117-279 in Handbook of Logic in Computer Science, Volume 2, Background: Computational Structures. Edited by S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum. Oxford University Press, New York, 1992.

[12] Luis Damas, Robin Milner. *Principal type-schemes for functional programs.* POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, pp. 207-212.

[13] John Harrison. *HOL Light Tutorial.* `http://www.cl.cam.ac.uk/~jrh13/hol-light/tutorial_220.pdf`, accessed 2009-03-18.

[14] Norman Megill. *Metamath: A Computer Language for Pure Mathematics.* `http://us.metamath.org/downloads/metamath.pdf`, accessed 2009-03-18.

[15] Randall Holmes. *Documentation for marcel.sml.* `http://math.boisestate.edu/~holmes/marcelstuff/marcelmanual.pdf`, accessed 2009-03-18.

[16] Jensen, R.B. *On the consistency of a slight(?) modification of Quine's NF.* Synthese 19 (1969), pp. 250-263.

[17] Crabbé, M., *The Hauptsatz for stratified comprehension: a semantic proof.* Mathematical Logic Quarterly 40, 1994, pp 481-489.

[18] Freek Wiedijk. *Projects.* `http://www.cs.ru.nl/~freek/projects/index.html`, accessed 2009-03-18.