THEOREM PROVING IN ELEMENTARY ANALYSIS

by

Joanna Porter Guild

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Mathematics

Boise State University

June 2007

The thesis presented by *Joanna Porter Guild* entitled *Theorem Proving in Elementary Analysis* is hereby approved.

M. Randall Holmes, Advisor	Date
Alexander Feldman, Committee Member	Date
Justin Moore, Committee Member	Date
John R. Pelton, Graduate Dean	Date

ABSTRACT

We describe how the theorem prover Marcel, a program written by Randall Holmes, can be used to prove a result from introductory analysis. We were given the axioms for an ordered field, and we completed a proof of the theorem that every positive real number has a square root. Marcel implements first-order logic via a Gentzen-style sequent calculus, and we first give an outline of the rules for this sequent calculus. We go on to describe how these rules are implemented in Marcel and give example proofs completed using Marcel. This is followed by a brief overview of the axioms for an ordered field and how these axioms were declared in Marcel. We give an outline of an intuitive pencil and paper-style proof of the fact that every positive number has a square root, and we go on to describe a formal proof, completed using Marcel, of the same fact. Finally, we give a few comments regarding the possibilities of using Marcel as an educational tool.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

The main goal of this paper is to describe how the theorem prover Marcel [4], a program written by Randall Holmes, was used to prove a nontrivial result from introductory analysis. We were given the axioms for an ordered field, and we completed a proof of the theorem that every positive real number has a square root [6]. Marcel is designed as an educational tool, and implements first-order logic via a Gentzen-style sequent calculus.

We first give an outline of the sequent calculus. We give a set of primitive rules and a fuller set of rules used in practice. The additional rules can each be derived from the primitive rules and basic definitions from first-order logic, and we give an example of such a derivation.

We then describe how the rules for our sequent calculus are implemented in Marcel, commenting also on the notational issues which arise from the requirements of the programming language ML in which Marcel is written, as well as the limitations of ASCII. We then walk through three sample proofs using Marcel. The first is a basic proof in propositional logic, which is followed by a traditional proof of the completeness of propositional logic as implemented in Marcel. Our second example introduces the use of quantifiers in Marcel. Our final example was carried out within the context of our proof that every positive real number has a square root. This example involves the use of some of the axioms for an ordered field and also uses theorems we had already proved using those axioms.

Following the examples in Marcel, we describe the axioms for an ordered field as seen in a typical analysis book, then go on to discuss how these axioms were declared in Marcel. We also touch on some difficulties we faced declaring these axioms. These difficulties arose from the fact that Marcel is based on a logical system strong enough to show that the universe is larger than just the set of real numbers, yet our proof involves statements concerning only real numbers. After the axioms for an ordered field have been declared, we walk through a traditional proof of the fact that every positive real number has a square root. Our next task is to adapt this traditional proof into one that can be used with Marcel. We then give a brief outline of how the proof was completed using the theorem prover.

Lastly, we discuss some of the possible benefits, as well as the shortcomings, of using Marcel as an educational tool. Proofs completed using Marcel have the advantage that they are usually cleaner than hand-written proof trees, and there is the benefit that a such a proof, if completed, is almost guaranteed to be a valid proof. Marcel does require more rigor than is usually required for hand-written proofs, but it also exposes students to the rigor required to complete a formal proof in logic. With continued use of Marcel, we hope that students will more thoroughly understand when and how to use the rules of formal logic.

Chapter 2

A GENTZEN-STYLE SEQUENT CALCULUS

A sequent is an expression $\Gamma \vdash \Delta$, where Γ and Δ are (possibly empty) finite sets of formulas. A sequent is said to be valid if under any interpretation of the nonlogical symbols in which all formulas in Γ are satisfied, at least one formula in Δ is satisfied. If we are working within a given theory, its axioms and definitions commit us to fixed interpretations of some nonlogical symbols. We note that the sequent $\Gamma \vdash \emptyset$ is valid only if it is not possible for all formulas in Γ to be satisfied. If the sequent $\emptyset \vdash B$ is valid for a formula B, then we can take B to be a theorem. We note that we abuse notation here by writing B instead of $\{B\}$.

Given a sequent, suppose we wish to show that it is valid (or invalid). A sequent calculus is a collection of rules that allow us to simplify the sequent. The application of some rules, called branching rules, result in two sequents, each of which is simpler than the original sequent. When using a sequent calculus to show the validity of a sequent, one is generally working backwards. We start with the sequent whose validity is in question, and use the rules from our sequent calculus to simplify the sequent into one or more sequents that are readily seen to be valid or invalid. We note that the calculus is set up so that the validity of each simpler sequent implies the validity of the more complicated sequent at each step, and as we introduce our rules for a Gentzen-style sequent calculus we give intuitive explanations as to why this is the case for each rule.

This sequent calculus is set up slightly differently than a usual system of logical reasoning because there are multiple conclusions. We point out that these are alternative conclusions: if our premises are satisfied we need only to satisfy one conclusion to have a valid sequent.

2.1 Primitive Rules

We describe first the basic rules of our sequent calculus. We assume Γ and Δ are finite (possibly empty) sets of formulas, and A and B are formulas. These rules come in two forms: most have one or two sequents above the line and one sequent below. These mean that if the sequents above the line are valid, then the sequent below the line is valid. Two rules, including the first rule, have no sequent above the line, only a sequent below. These mean that the sequent below the line is always a valid sequent. Our first rule is the Assumption Rule.

(**Assm**):

$$\frac{}{\Gamma, A \vdash A, \Delta}$$

If Γ and A are satisfied, then clearly A or some formula in Δ is satisfied, so the sequent is valid.

2.1.1 Negation Rules

(¬**A**):

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$$

We assume $\Gamma \vdash A, \Delta$ is valid. Now suppose all formulas in Γ are satisfied and $\neg A$ is satisfied. By the validity of the premise, either A is satisfied or some formula in Δ is satisfied. But since $\neg A$ is satisfied, A cannot be satisfied. Therefore some formula in Δ is satisfied and our conclusion, $\Gamma, \neg A \vdash \Delta$ is valid.

(¬**S**):

$$\frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$$

We assume $\Gamma, A \vdash \Delta$ is valid. If we assume all formulas in Γ are satisfied, we have two possible scenarios depending on the satisfaction of A. If A is satisfied, then some formula in Δ is satisfied by the validity of the premise. If A is not satisfied, then $\neg A$ is satisfied. Therefore some formula on the right of the conclusion is satisfied in either case, and the conclusion is therefore valid.

2.1.2 Disjunction Rules

(∨**A**):

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$$

Assume $\Gamma, A \vdash \Delta$ and $\Gamma, B \vdash \Delta$ are valid. Now if all formulas in Γ are satisfied and $A \vee B$ is satisfied, then A is satisfied or B is satisfied. If A is satisfied, then since all formulas in Γ are satisfied the validity of the left-hand premise implies some formula in Δ is satisfied. Similarly, if B is satisfied then some formula in Δ must be satisfied by the validity of the right-hand premise. Therefore our conclusion is valid.

($\vee$**S**):

$$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$$

Assume $\Gamma \vdash A, B, \Delta$ is valid. If all formulas in Γ are satisfied, then A is satisfied, B is satisfied, or some formula in Δ is satisfied. That is, $A \vee B$ is satisfied or some formula in Δ is satisfied. Therefore the conclusion $\Gamma \vdash A \vee B, \Delta$ is valid.

2.1.3 Existential Quantifier Rules

Prior to discussing rules for dealing with existential quantifiers, we define a notation for substitution: For a formula ϕ, the expression $\phi\frac{t}{x}$ represents the formula ϕ with each occurrence of the variable x replaced by the term t. If ϕ contains quantifiers there are some subtleties involved in defining $\phi\frac{t}{x}$. We avoid discussing the details, which essentially come down to renaming bound variables before making substitutions [3].

($\exists$**S**):

$$\frac{\Gamma \vdash \phi\frac{t}{x}, \exists x\phi, \Delta}{\Gamma \vdash \exists x\phi, \Delta}$$

for any term t.

In this and one other quantifier rule, ($\forall$**A**), we adopt the practice of preserving the quantified formula in the premise of the rule. This lets us avoid having to "copy" a formula if we wish to instantiate it more than once during a proof. To see the validity of this rule, assume $\Gamma \vdash \phi\frac{t}{x}, \exists x\phi, \Delta$, and assume all formulas in Γ are satisfied. Then by the validity of the premise, $\phi\frac{t}{x}$, $\exists x\phi$, or some formula in Δ must be satisfied. The last two pose no problem, so we consider the first. If $\phi\frac{t}{x}$ is satisfied, then t witnesses the existence of an x for which ϕ is satisfied, therefore the conclusion $\exists x\phi$ is true.

($\exists$**A**):

$$\frac{\Gamma, \phi\frac{y}{x} \vdash \Delta}{\Gamma, \exists x\phi \vdash \Delta}$$

for variable y not free in Γ, $\exists x\phi$, or Δ.

To see the validity of this rule, assume $\Gamma, \phi\frac{y}{x} \vdash \Delta$ is valid and that all formulas in Γ are satisfied. If $\exists x\phi$ is satisfied, there must be some t for which ϕ is true. So $\phi\frac{y}{x}$ is satisfied in any interpretation under which the variable y represents this object t. By the validity of the premise we can conclude that some formula in Δ is satisfied, and because Δ does not mention y, this formula was satisfied before we assigned an interpretation for y. Therefore our rule is valid.

2.1.4 Equality Rules

($=$ **S**):

$$\frac{}{\Gamma \vdash (t = t), \Delta}$$

We note first that this rule of reflexivity of equality is the only rule other than the assumption rule that is an axiom in our sequent calculus. This rule is clearly valid:

$t = t$ is always satisfied, so if all formulas in Γ are satisfied, it is clear that $t = t$ or some formula in Δ is satisfied, so our conclusion is valid.

($=$**A**):

$$\frac{\Gamma, (t = t') \vdash \phi\frac{t}{x}, \Delta}{\Gamma, (t = t') \vdash \phi\frac{t'}{x}, \Delta}$$

We assume all formulas in Γ and $t = t'$ are satisfied. Then by the validity of the premise, $\phi\frac{t}{x}$ is satisfied. Because $t = t'$, we can conclude that $\phi\frac{t'}{x}$ is satisfied as well, so our rule is valid.

2.2 Derivable Rules

These rules in this section can all be derived from the primitive rules, if the connectives involved are assumed to be defined in terms of $\neg$ and $\vee$. An example of such a derivation is given at the end of this section. Although these rules are mathematically derivable from the primitive rules, Marcel sees all the rules given in this section and the previous section as having equivalent status.

2.2.1 Conjunction Rules

($\wedge$**A**):

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$$

Assume $\Gamma, A, B \vdash \Delta$ is valid. To have all formulas in Γ and $A \wedge B$ satisfied is the same as having all formulas in Γ, A, and B satisfied. By the validity of the premise some formula in Δ must then be satisfied. Hence our conclusion is valid.

($\wedge$**S**):

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash (A \wedge B), \Delta}$$

Assume $\Gamma \vdash A, \Delta$ and $\Gamma \vdash B, \Delta$ are valid. If we assume all formulas in Γ are satisfied, we have two possibilities depending on whether something in Δ is satisfied. If something in Δ is satisfied, then our conclusion is valid. If nothing in Δ is satisfied, then by the validity of the premises A must be satisfied and B must be satisfied. Therefore $A \wedge B$ is satisfied, and our conclusion is valid. Hence our rule is valid in either case.

2.2.2 Implication Rules

($\rightarrow$ **A**):

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, (A \rightarrow B) \vdash \Delta}$$

Suppose $\Gamma \vdash A, \Delta$ and $\Gamma, B \vdash \Delta$ are valid. Now assume all formulas in Γ are satisfied and $A \rightarrow B$ is satisfied. We have two scenarios depending on the satisfaction of A. If A is satisfied then B is satisfied, since $A \rightarrow B$ is satisfied. Therefore some formula in Δ is satisfied, by the validity of the second premise. If A is not satisfied, some formula in Δ must be satisfied by the validity of the first premise, since all formulas in Γ are satisfied. Therefore $\Gamma, (A \rightarrow B) \vdash \Delta$ is a valid conclusion.

($\rightarrow$ **S**):

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash (A \rightarrow B), \Delta}$$

Assume $\Gamma, A \vdash B, \Delta$ is valid. If we assume all formulas in Γ are satisfied, we once again have two scenarios depending on the satisfaction of A. If A is not satisfied, then $A \rightarrow B$ is satisfied so our conclusion is valid. If A is satisfied, then by the validity of the premise, either B is satisfied (and therefore $A \rightarrow B$ is satisfied) or some formula in Δ is satisfied. Therefore our conclusion is valid.

($\leftrightarrow$ **A**):

$$\frac{\Gamma \vdash A, B, \Delta \qquad \Gamma, A, B \vdash \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}$$

We assume $\Gamma \vdash A, B, \Delta$ and $\Gamma, A, B \vdash \Delta$ are satisfied. Now if all formulas in Γ are satisfied and $A \leftrightarrow B$ is satisfied, we have two possibilities depending on how $A \leftrightarrow B$ is satisfied. If A and B are both satisfied, then some formula in Δ is satisfied by the validity of the second premise. If neither A nor B is satisfied, then by the validity of the first premise, some formula in Δ is satisfied. Our rule is valid in either case.

($\leftrightarrow$ **S**):

$$\frac{\Gamma, A \vdash B, \Delta \qquad \Gamma, B \vdash A, \Delta}{\Gamma \vdash A \leftrightarrow B, \Delta}$$

We assume $\Gamma, A \vdash B, \Delta$ and $\Gamma, B \vdash A, \Delta$ are valid and that all formulas in Γ are satisfied. We have several possible scenarios depending on the satisfaction of A and B. If both A and B are satisfied, then $A \leftrightarrow B$ is satisfied. Likewise, if neither A nor B is satisfied, $A \leftrightarrow B$ is satisfied. If A is satisfied but B is not satisfied, then by the validity of our first premise, some formula in Δ is satisfied. Similarly, if B is satisfied but A is not satisfied, then by the validity of our second premise some formula in Δ is satisfied. Therefore regardless of the satisfaction of A or B, if all formulas in Γ are satisfied then $A \leftrightarrow B$ or some formula in Δ is satisfied, so our rule is valid.

2.2.3 The Cut Rule

(**Cut**):

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma \vdash \Delta}$$

Assume $\Gamma, A \vdash \Delta$ and $\Gamma \vdash A, \Delta$ are valid. Now assume all formulas in Γ are satisfied. Then we have two possibilities, either A is satisfied or A is not satisfied. If A is satisfied, then some formula in Δ is satisfied by the validity of the first premise. If A is not satisfied then some formula in Δ is satisfied by the validity of the second premise and the fact that all formulas in Γ are satisfied. Therefore some formula in Δ is satisfied independent of the satisfaction of A, so our conclusion is valid.

In practice, the Cut Rule is used to introduce a lemma (the formula represented by A above) into a proof. If we can show that the lemma follows from our premise(s), we are then able to include the lemma as a premise in our proof. It is a technical result beyond the scope of this paper that the Cut Rule can be derived from the Primitive Rules stated here. As with any of the derivable rules, all proofs using the Cut Rule in propositional and first-order logic can also be done without the Cut Rule, although the proofs may become much longer [1].

2.2.4 The Antecedent Rule

(**Ant**):

$$\frac{\Gamma \vdash \Delta}{\Gamma' \vdash \Delta'}$$

where $\Gamma \subseteq \Gamma'$ and $\Delta \subseteq \Delta'$.

To see the validity of this rule, we assume $\Gamma \vdash \Delta$ is valid, and assume all formulas in Γ' are satisfied. Then all formulas in Γ are satisfied, therefore some formula in Δ must be satisfied. Since $\Delta \subseteq \Delta'$, at least one formula in Δ' must be satisfied. Therefore $\Gamma' \vdash \Delta'$ is a valid conclusion.

2.2.5 Universal Quantifier Rules

($\forall$**S**):

$$\frac{\Gamma \vdash \phi\frac{y}{x}, \Delta}{\Gamma \vdash \forall x\phi, \Delta}$$

for variable y not free in Γ, $\forall x\phi$, or Δ.

We suppose $\Gamma \vdash \phi\frac{y}{x}, \Delta$ is valid and all formulas in Γ are satisfied. Then because our premise is valid, we can conclude that $\phi\frac{y}{x}$ is satisfied or some formula in Δ is satisfied. If some formula in Δ is satisfied our conclusion is clearly valid. Therefore we consider the case that $\phi\frac{y}{x}$ is satisfied but no formula in Δ is satisfied. Because y has been given to us arbitrarily (that is, y does not appear free in Γ, $\forall x\phi$, or Δ) it can be assigned any interpretation without changing the satisfaction of Γ, $\forall x\phi$, or Δ. Therefore we can conclude that $\forall x\phi$ must be satisfied, because $\phi\frac{y}{x}$ must be satisfied for any y given to us arbitrarily, so our rule is valid in either case.

($\forall$**A**):

$$\frac{\Gamma, \phi\frac{t}{x}, \forall x\phi \vdash \Delta}{\Gamma, \forall x\phi \vdash \Delta}$$

for any term t.

We assume $\Gamma, \phi\frac{t}{x}, \forall x\phi \vdash \Delta$ is a valid sequent, and we assume all formulas in Γ and $\forall x\phi$ are satisfied. Because $\forall x\phi$ is satisfied, $\phi\frac{t}{x}$ is satisfied for any choice of t. Therefore by the validity of the premise, some formula in Δ is satisfied, and our conclusion is valid. As in ($\exists$**S**) we preserve the quantified formula in the premise of the rule so that it remains available for further instantiations without the need of explicitly copying it.

2.2.6 A Sample Derivation

We recall the rule for an implication in the antecedent:

($\rightarrow$ **A**):

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, (A \rightarrow B) \vdash \Delta}$$

We show a derivation for this rule using only primitive rules and the definition of implication.

(1)	$\Gamma \vdash A, \Delta$	Premise
(2)	$\Gamma, B \vdash \Delta$	Premise
(3)	$\Gamma, \neg A \vdash \Delta$	($\neg$**A**) applied to (1)
(4)	$\Gamma, \neg A \vee B \vdash \Delta$	($\vee$**A**) applied to (2) and (3)
(5)	$\Gamma, A \rightarrow B \vdash \Delta$	Definition of $\rightarrow$

Alternatively, we can present this derivation as a proof tree.

$$\dfrac{\dfrac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,(\neg\mathbf{A}) \qquad \Gamma, B \vdash \Delta}{\dfrac{\Gamma, \neg A \vee B \vdash \Delta}{\Gamma, A \rightarrow B \vdash \Delta}\,(\text{Definition of } \rightarrow)}\,(\vee\mathbf{A})$$

Chapter 3

MARCEL

3.1 Implementation of the Sequent Calculus

We present an outline of the differences between the usual notation for logic which we have been using without comment and the formal notation expected by Marcel, which has to adapt to the limitations of ASCII.

- Negation
  $\neg P$ is written `~P`
- Binary Connectives
  $P_1 \vee P_2$ is written `P1 v P2`
  $P_1 \wedge P_2$ is written `P1 & P2`
  $P_1 \rightarrow P_2$ is written `P1 -> P2`
  $P_1 \leftrightarrow P_2$ is written `P1 == P2`
- Binary Predicates
  $a_1 = a_2$ is still written `a1 = a2`
  $a_1 \in a_2$ becomes `a1 E a2`
- Quantifiers
  $\forall x_1 P_1(x_1)$ is written `(Ax1.P1(x1))`
  $\exists x_1 P_1(x_1)$ is written `(Ex1.P1(x1))`

  We make a technical note that the parentheses and dots in the above two examples are mandatory for quantifiers in Marcel.

Furthermore, when working with Marcel we use certain letters to denote certain types of variables. `P1,P2,P3,...` are used to represent predicate and propositional variables, `x1,x2,x3,...` represent bound variables, free variables are represented by `a1,a2,a3,...` , and `U1,U2,U3,...` are used for unknown variables.

The sequent calculus used in Marcel is mostly standard; its details are taken from [2], whose full system includes set theory operations not discussed here (though they are used a little in the main proof).

3.2 Examples

3.2.1 Propositional Example

We demonstrate how to use Marcel to show that the sequent

$$\vdash ((P_1 \vee P_2) \wedge (P_1 \to P_3) \wedge (P_2 \to P_3)) \to P_3$$

where P_i is a propositional variable, is valid. We enter the proposition using the `start`, or `s`, command. This command takes as an argument a string, which will be the proposition we wish to begin proving.

This creates a sequent with nothing on the left and the proposition entered on the right. Alternatively we could use the `StartSequent`, or `ss`, command which takes as an argument two string lists, the first being the left side of the sequent we wish to prove, the second the right side.

```
 - Start "((P1vP2)&(P1->P3)&(P2->P3))->P3";
Line number 1:
 |-
 1:  (P1 v P2) & (P1 -> P3)
      & (P2 -> P3) -> P3
> val it = () : unit
 -
```

The line `> val it = () : unit` is a message from the ML interpreter that can be ignored. Continuing our proof, we use the `R`, or `r`, command which is short for "right" and will apply the relevant rule from our sequent calculus to the first proposition on the right side of the sequent. The `r` command has a variety of other uses, including expanding definitions and acting as a `done` command if reflexivity of equality applies. Here the rule ($\rightarrow$**S**) is applied. We note that the `r` command takes no arguments, but the formal "unit" argument, represented by (), is required by the ML programming language.

```
 - r();
Line number 2:
 1:  (P1 v P2) & (P1 -> P3)
      & (P2 -> P3)
 |-
 1:  P3
> val it = () : unit
 -
```

The `L`, or `l`, command, which stands for "left" works similarly to the `r` command, applying the appropriate rule to the first proposition on the left side of the sequent. Here ($\wedge$ **A**) is applied.

```
 - l();
Line number 3:
 1:  P1 v P2
 2:  (P1 -> P3) & (P2 -> P3)
 |-
 1:  P3
> val it = () : unit
 -
```

Because `l` works only on the first proposition on the left side of the sequent, our next step is to move the second proposition on the left into the first position. To do this we use the command `GetLeft`, or `gl`, which takes as an argument the line number of the proposition to be moved to the first position. We note that reordering the propositions in our sequent is mathematically valid because the left and right sides of these sequents are defined as sets of formulas and have therefore no intrinsic order. In other sequent calculi the left and right sides of sequents are defined as lists, and issues of reordering and copying of formulas require more attention.

```
 - GetLeft 2;
Line number 3:
 1:  (P1 -> P3) & (P2 -> P3)
 2:  P1 v P2
 |-
 1:  P3
> val it = () : unit
 -
```

Now `l` will expand the conjunction.

```
 - l();
Line number 4:
 1:  P1 -> P3
 2:  P2 -> P3
 3:  P1 v P2
 |-
 1:  P3
> val it = () : unit
 -
```

Again we use `l`. This time the appropriate rule is ($\rightarrow$ **A**). This is a branching rule, and we are given one branch to prove first. The prover saves the other branch which we will be asked to prove once the proof of the first branch is complete.

```
 - l();
Line number 5:
 1:  P2 -> P3
 2:  P1 v P2
 |-
 1:  P1
 2:  P3
> val it = () : unit
 -
```

Our next goal is to move the second proposition on the left to the first position, then apply `l`. This can be done in one step with the command `Gl`, which takes as an argument the line number of the proposition to be rotated to the first position.

```
 - Gl 2;
Line number 5:
 1:  P1 v P2
 2:  P2 -> P3
 |-
 1:  P1
 2:  P3

Line number 7:
 1:  P1
 2:  P2 -> P3
 |-
 1:  P1
 2:  P3
> val it = () : unit
 -
```

The command `l`, executed within `Gl`, caused Marcel to apply the rule ($\vee$**A**), which is another branching rule. The first branch we are given can be immediately seen to be valid, so we are done with this branch of the proof. To indicate to Marcel that this branch of the proof is complete, we issue the `Done`, or `d`, command.

```
 - d();
Line number 8:
 1:  P2
 2:  P2 -> P3
 |-
 1:  P1
 2:  P3
> val it = () : unit
 -
```

Now we must prove the second branch that resulted from our application of the rule ($\rightarrow$ **A**). Our first move in this second branch is to use `Gl` to expand the second proposition on the left. This command causes Marcel to apply the rule ($\rightarrow$ **A**) which will create two more branches in the proof.

```
 - Gl 2;
Line number 8:
 1:  P2 -> P3
 2:  P2
 |-
 1:  P1
 2:  P3

Line number 9:
 1:  P2
 |-
 1:  P2
 2:  P1
 3:  P3
> val it = () : unit
 -
```

We are again done with this branch right away.

```
 - d();
Line number 10:
 1:  P3
 2:  P2
 |-
 1:  P1
 2:  P3
> val it = () : unit
 -
```

This second branch is a valid sequent, but because `P3` appears in the second line on the right instead of the first the prover will not recognize the `done` command as valid. We could use `gr` (which works just like `gl`, only on the right) and then `done`. A shorter way is to use `Triv`, a command which takes two numeric arguments `m` and `n` and has the same net effect as `gl m`, `gr n`, `done`.

```
 - Triv 1 2;
Line number 10:
 1:  P3
 2:  P2
 |-
 1:  P1
 2:  P3

Line number 10:
 1:  P3
 2:  P2
 |-
 1:  P3
 2:  P1

Line number 6:
 1:  P3
 2: P2 -> P3
 3:  P1 v P2
 |-
 1:  P3
> val it = () : unit
 -
```

The final branch we are given is the second branch from when we applied the rule ($\vee$ **A**), and when we enter the `Done` command Marcel displays Q.E.D. to indicate that the proof of the original sequent is complete.

```
- Done();
Q. E. D.
> val it = () : unit
```

3.2.2 Proof of Completeness

Theorem: Our sequent calculus is complete for propositional logic. More specifically, we are able to use a finite combination of the commands `l`, `r`, `gl`, and `gr` in Marcel to prove any valid sequent involving only propositional letters P_i, the logical symbol $\neg$, and the logical connectives $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$.

Proof: Assume we have a valid sequent involving only propositional letters P_i, the logical symbol $\neg$, and the logical connectives $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. The proof of our theorem is easiest if we visualize the proof of this valid sequent as a proof tree. We then proceed by induction on the total number of logical symbols ($\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$) in the sequent. If our sequent has no logical symbols, only propositional letters, we claim that it is provable by the assumption rule. If it were not provable, it would be invalid. This is because the sequent consists only of propositional letters; if it was not provable it must be that none of the propositional letters appear on both the left and right sides of the sequent. We could assign all propositional letters on the left to be true and all propositional letters on the right to be false, a counterexample that would show the sequent to be invalid. As our induction hypothesis we assume any valid sequent involving only propositional letters and n logical symbols is provable. Now suppose we have a valid sequent with only propositional letters and $n + 1$ logical symbols. We use `gl` or `gr` as needed to bring a nontrivial formula to the top of our list on either the left or the right side of the sequent, then use the `l` or `r` command to apply an appropriate rule from our sequent calculus. Inspection of the sequent calculus rules for the individual logical connectives will show that applying this rule will produce premises with one fewer logical connective than the conclusion. This reduces the number of logical symbols in each sequent by one, though it may generate two sequents each with the n logical symbols. (If this happens repeatedly, the size of the proof could increase exponentially, although the proof would still terminate after a finite number of steps.) If the rule generates only one sequent with n logical symbols, we use our induction hypothesis to show that this sequent is provable. Otherwise, if the rule generates two sequents each with n logical symbols, we apply our induction hypothesis to each sequent to show both are provable. In either case, our sequent involving only propositional letters and $n+1$ logical symbols is provable.

3.2.3 Quantifier Example

We use the prover to show that the sequent

$$\vdash \exists x_1(\forall x_2(P_1(x_1) \to P_1(x_2)))$$

where P_1 is a unary predicate, is valid. Our first step is to issue the `r` command which will apply the ($\exists$**S**) rule. This replaces `x1` with the unknown variable `U1` to which we will later assign a value. We note that the rules for our sequent calculus require that we name our witness when we apply the rule ($\exists$**S**), but the prover gives us the freedom to name this witness later in the proof. This is convenient if we are unsure at this point in the proof what would be an appropriate witness.

```
 -  s "(Ex1.(Ax2.(P1(x1) ->P1(x2))))";
Line number 1:
 |-
 1: (Ex1.(Ax2.P1(x1) -> P1(x2)))
> val it = () : unit
 - r();

Line number 2:
 |-
 1:  (Ax3.P1(U1) -> P1(x3))
 2: (Ex1.(Ax2.P1(x1) -> P1(x2)))
> val it = () : unit
-
```

Applying ($\forall$**S**) by using the `r` command again replaces `x3` with the free variable `a2`.

```
 - r();
Line number 3:
 |-
 1:  P1(U1) -> P1(a2)
 2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
> val it = () : unit
-
```

We issue the `r` command yet again to apply ($\rightarrow$ **S**).

```
- r();
Line number 4:
 1:  P1(U1)
 |-
 1:  P1(a2)
 2:  (Ex1.(Ax2.P1(x1)-> P1(x2)))
> val it = () : unit
-
```

It is not mathematically valid to set the unknown `U1` to `a2` because `a2` did not "exist" when we applied the ($\exists$**S**) rule. The prover recognizes this, and that the `a2` was introduced into the proof after the `U1` is witnessed by the higher index, which is a nice feature if we wish to use the prover for educational purposes. Our next step is to instantiate the existential conclusion again, followed by an application of the ($\rightarrow$ **S**) rule to get the following.

```
- Gr 2;
Line number 4:
1:  P1(U1)
|-
1:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
2:  P1(a2)

Line number 5:
1:  P1(U1)
|-
1:  (Ax3.P1(U3) -> P1(x3))
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
3:  P1(a2)
> val it = () : unit
- r();
```

```
Line number 6:
1:  P1(U1)
|-
1:  P1(U3) -> P1(a4)
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
3:  P1(a2)
> val it = () : unit
- r();

Line number 7:
1:  P1(U3)
2:  P1(U1)
|-
1:  P1(a4)
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
3:  P1(a2)
> val it = () : unit
-
```

Now we could use the `SetUnknown`, or `su`, command to set `U3` to the value `a2`. The `SetUnknown` command takes a numeric and string argument, the numeric argument provides the index of the unknown variable whose value we wish to set and the string argument gives the value to which the unknown variable is to be set. We note that the `SetUnknown` command fixes the value for a given unknown variable throughout the proof, not just the current sequent. This is in contrast to most prover commands which act only "locally" on the branch of the proof in question. However, the prover is designed to automatically set unknown variables in cases such as this. Here, for example, if the command `Triv 1 3` is issued, the propositions listed in positions 1 on the left and position 3 on the right are not the same, but could be made equal with appropriate values assigned to unknown variables. The prover recognizes this and issuing the command `Triv 1 3` will cause the unknown variable U3 to be automatically set to the value a2, and the sequent will be accepted as valid, thus completing the proof.

```
- Triv 1 3;
Line number 7:
 1:  P1(U3)
 2:  P1(U1)
 |-
 1:  P1(a2)
 2:  P1(a4)
 3:  (Ex1.(Ax2.P1(x1) -> P1(x2)))

Q. E. D.
> val it = () : unit
-
```

We note that there are other situations where the prover will automatically set unknowns: we will see two of these situations shortly. We also make a technical note that a step involving line reordering was omitted from the above screen-shot.

3.2.4 Example from the Main Proof

For our final example we consider the proof of a lemma that was completed within the context of our main proof, that every positive real number has a square root. In the process of this main proof we proved several necessary background lemmas, one of which was

$$\vdash 0 < a_1 \rightarrow 0 < \frac{1}{2}a_1$$

the proof of which we present here.

We input the sequent and use the `r` command to expand the implication on the right.

```
 -Start "Zero < a1 -> Zero < Half*a1";
 Line number 1:
 |-
 1:  Zero < a1 -> Zero < Half * a1

> val it = () : unit

 - r();
Line number 2:
 1:  Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
-
```

Intuitively our plan is obvious, if $0 < a_1$ multiplying both sides by $\frac{1}{2}$ will give the desired conclusion. A theorem concerning the monotonicity of multiplication exists in our theory and it is this theorem we introduce. We can view this theorem, which is called `MTIMES` using the `ThmDisplay` command, as shown below.

```
-ThmDisplay "MTIMES";

MTIMES:

|-

1:  Zero < a3 & a1 < a2 -> a1 * a3 < a2 * a3
```

This theorem states that multiplying both sides of an inequality by a positive number retains the validity of the inequality. To introduce this theorem we use the command `ThmCut` which invokes the cut rule on an instance of a theorem. Because the theorem has already been shown to be valid, or stated as an axiom as in this case, we are immediately able to include the theorem in our premises.

```
- ThmCut "MTIMES";
Line number 3:

|-
 1:  Zero < U2 & U3 < U4 -> U3 * U2 < U4
      * U2

Line number 4:
 1:  Zero < U2 & U3 < U4 -> U3 * U2 < U4
      * U2
 2:  Zero < a
 |-
 1:  Zero < Half * a1
> val it = () : unit
-
```

We note that when we invoke a theorem cut, the instance of the theorem to be used is automatically generated and proved by the system which then immediately presents an application of the theorem. Therefore it is easy to miss the actual statement of the theorem, seen in Line 3 above.

Our plan is to use the fact that $0 < \frac{1}{2}$ and $0 < a_1$ to conclude that $0 < \frac{1}{2}a_1$. So we could use `SetUnknown`, or `su`, to set `U2` to be `Half`, `U3` to be `Zero`, and `U4` to `a1`. Having Marcel set these values for us is generally a more stable course of action because it does not require referencing the indices of the unknown variables, which may change if we alter anything earlier in the proof that affects the variable counters.

We expand the theorem just introduced using `r` and `l`.

```
Line number 7:
 1:  Zero < a1
 |-
 1:  Zero < U2
 2:  Zero < Half * a1
> val it = () : unit
-
```

At this point we can use a theorem already established in our theory that $\frac{1}{2}$ is positive. This theorem is called `HALFPOS` and we can view the specific statement of this theorem using the `ThmDisplay` command.

```
- ThmDisplay "HALFPOS";

HALFPOS:

|-

1:  Zero < Half
```

We invoke the theorem `HALFPOS` with the command `UseThm`, which takes a string followed by two integer lists as arguments. The lists indicate the propositions on the left and right of the current sequent that match the theorem to be used, and this theorem we wish to use is named by the string. If the current sequent matches the named theorem, it is considered proved and we move to the next branch of the proof. To use `HALFPOS`, we could set `U2` to be `Half` using the `SetUnknown` command. This is not necessary however, as Marcel will recognize this implicitly if we instead just ask it to use the theorem `HALFPOS` on the first proposition on the right side.

```
 - UseThm "HALFPOS" [] [1];
Line number 8:
 1:  Zero < a1
 |-
 1:  U3 < U4
 2:  Zero < Half * a1
> val it = () : unit
-
```

Here, issuing the `Done` command will cause the prover to recognize that setting `U3` to `Zero` and `U4` to `a1` would create a valid sequent. We issue this command, therefore setting `U3` to `Zero` and `U4` to `a1` for the remainder of the proof, and we are done with this branch of the proof.

```
 - d();
Line number 6:
 1:  Zero * Half < a1 * Half
 2:  Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
-
```

The first step in getting the prover to see that this is a valid sequent is to introduce the theorem `MZERO2` which states that `Zero` times anything is `Zero`.

```
 - ThmCut "MZERO2";
Line number 9:
 |-
 1:  Zero * U3 = Zero

Line number 10:
 1:  Zero * U3 = Zero
 2:  Zero * Half < a1 * Half
 3:Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
```

Now we will implicitly set `U3` to `Half` and use the first proposition on the left to rewrite the second proposition on the left. The command for this rewrite is `Rwl`, or `rwl`, which takes an integer list as an argument. If the list is empty, denoted by `[]` or `nil`, each occurrence of the left side of the first equation is replaced by the right side. If the list is nonempty, only the occurrences indexed by the terms of the list are replaced. We note that the first proposition on the left side of the sequent must be an equation for this command to be used. The command `Crwl`, or `crwl`, works as `rwl`, but occurrences of the right side of the first equation are replaced by the left side of the equation. The command `Rwr`, or `rwr`, works in a similar manner to `rwl`, but uses the first proposition on the left to rewrite the first proposition on the right side of the sequent. Finally, the command `Crwr`, or `crwr`, works as `rwr` but rewrites the left side of the equation to the right side [1].

[1]For more on the rewrite rules, see Appendix A.

```
 - rwl [1];
Line number 11:
 1:  Zero * Half = Zero
 2:  Zero < a1 * Half
 3:  Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
-
```

Because we won't need the hypothesis `Zero * Half = Zero` again for this proof, we can discard it using `PruneLeft`, or `pl`, which takes as an argument the line number of the proposition to be pruned. There is also a `PruneRight`, or `pr`, command which acts similarly on the right side of a sequent. We note that the `PruneLeft` and `PruneRight` commands are the prover implementation of the Antecedent Rule from our sequent calculus.

```
- pl 1;
Line number 11:
 1:  Zero < a1 * Half
 2:  Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
-
```

Our final step is to use commutativity of multiplication, an axiom already declared in our theory, to rewrite the left side of the sequent. The appropriate theorem to introduce is `CTIMES`.

```
- ThmCut "CTIMES";
Line number 12:
 |-
 1:  U3 * U4 = U4 * U3

Line number 13:
 1:  U3 * U4 = U4 * U3
 2:  Zero < a1 * Half
 3: Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
```

We use `rwl` to rewrite the left side of the sequent and we have a sequent the prover will recognize as valid if issued the `Triv` command.

```
- rwl [1];
Line number 14:
 1:  a1 * Half = Half * a1
 2:  Zero < Half * a1
 3:  Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
- Triv 2 1;

Line number 14:
 1:  Zero < Half * a1
 2:  Zero < a1
 3:  a1 * Half = Half * a1
 |-
 1:  Zero < Half * a1

 Q. E. D.
> val it = () : unit
-
```

Because we will need this lemma later in our proof, we wish to save it as a theorem. To do this we use the `NameSequent` command, which takes as arguments the line number of the sequent to be named and the name we wish to give to the theorem. The command `Showall`, which takes no argument, allows the user to scroll through the lines of the proof if we wish to save a slightly different version of the sequent than the one we started with. Here, for example, we actually use line two of the proof as the sequent to be named.

```
- NameSequent 2 "MHALFPOS";
MHALFPOS:

 1:  Zero < a1
 |-
 1:  Zero < Half * a1
> val it = () : unit
-
```

Chapter 4

THE MAIN PROOF

4.1 Basic Setup

As a basis we are given the axioms for an ordered field. From these we wish to show that every positive real number has a square root. We recall the axioms for an ordered field as stated in a typical analysis book [5]. The objects of this universe are called "real numbers".

A1: $a + (b + c) = (a + b) + c$ for all a, b, c.

A2: $a + b = b + a$ for all a, b.

A3: $a + 0 = a$ for all a.

A4: For each a, there is an element $-a$ such that $a + (-a) = 0$.

M1: $a(bc) = (ab)c$ for all a, b, c.

M2: $ab = ba$ for all a, b.

M3: $a \cdot 1 = a$ for all a.

M4: For each $a \neq 0$, there is an element a^{-1} such that $aa^{-1} = 1$.

DL: $a(b + c) = ab + ac$ for all a, b, c.

NT: The system of real numbers has more than one element.

O1: Given a and b, either $a \leq b$ or $b \leq a$.

O2: If $a \leq b$ and $b \leq a$, then $a = b$.

O3: If $a \leq b$ and $b \leq c$, then $a \leq c$.

O4: If $a \leq b$, then $a + c \leq b + c$.

O5: If $a \leq b$ and $0 \leq c$, then $ac \leq bc$.

CA: Every nonempty subset S of the real numbers that is bounded above has a least upper bound. In other words, $\sup(S)$ exists and is a real number.

In the prover we first declare several basic notions, such as the constants one and zero, the addition operation, the less than relation, and the supremum of a set. The command for these declarations is `DeclareFunction` to declare a function, or `DeclarePredicate` to declare a predicate, both of which are followed by the name of the command, then an integer list gives the relative type of the output of the function followed by the relative types of the inputs of the function [1]. This implicitly defines

[1]All we need to know about types here is that type $n + 1$ consists of sets of objects of type n, so for example, if we take real numbers to be of type zero, sets of real numbers have type one.

the number of arguments to be one less than the length of the list. We show the declaration of addition:

```
 - DeclareFunction "+" [0,0,0];
```

and we note the list indicates that there are two inputs, each of the same relative type as the output. This is not the case for the supremum operation, whose declaration is:

```
- DeclareFunction "Sup" [0,1];
```

Here the list indicates that the input has a type one higher than the output, which certainly is the case for the supremum operation as the input is a set of numbers and the output is a number.

After the addition and multiplication functions are declared, we use the `setprecrightabove` command to set the precedence of multiplication to be just above the precedence for addition, but below all higher precedences, and for multiplication to group to the right (addition groups to the right by default). The `setprecrightabove` command takes two string arguments, the first is the operation to be given precedence over the second. (Marcel has several other precedence-setting options as well.)

With the declarations complete, we are ready to define the axioms for an ordered field. This is done with the `Axiom` command, which is followed by the name of the axiom, then two lists of strings to be parsed as formulas, representing the left and right side of the sequent to be declared as an axiom. Because most axioms are simply propositions, the left side is usually an empty list denoted `[]` or `nil`, and the right side is a one-element list. We give as an example the axiom of associativity of addition:

```
- Axiom "APLUS" nil ["[a1 + a2] + a3 = a1 + a2 + a3"];
```

We note that Marcel requires brackets, not parentheses, to be used as grouping symbols for objects. Parentheses are used to group statements and for argument lists. We note also that because addition groups to the right by default, no grouping symbols are needed on the right side of the equation.

Because Marcel is based on a logical system strong enough to show that the universe is larger than just the set of real numbers, it is necessary to include additional axioms and add extra assumptions in certain places concerning which variables represent real numbers. We assume each of our basic operations can take any sort of object as an argument, not just real numbers, but that these operations "coerce" the input to a real number. Therefore the first axioms we define state that the output of all basic operations declared above are real numbers. When we state the axioms for an ordered field, this has the effect of minimizing the need for additional assumptions that objects are real numbers.

We also define the relation `Equal` to hold if two objects are "coerced to the same real", that is, we say `x1 Equal x2` if `x1 + 0 = x2 + 0`, as `x1 + 0` is the real number to which `x1` is coerced, and `x2 + 0` is the real to which `x2` is coerced. This again minimizes the need to insert the assumption that a given object is a real number.

After giving the axioms that the output of our operations are all real numbers, we state the axioms for a field, most of which are formulated as given above. The identity laws are the only axioms for which we need the assumption that the object in question is a real number. For example, the additive identity axiom takes the form:

```
Axiom "IPLUS" [nil] ["Real(a1) -> a1+0 = a1"];
```

The multiplicative inverse axiom uses the `Equal` relation and has the following form:

```
 Axiom "INV" nil ["a1 Equal Zero v a1*Inv(a1) = One"];
```

Following the field axioms we state the requirement that $0 \neq 1$, which, in the presence of the other axioms, is equivalent to the axiom NT stated above.

Our next step is to define the order axioms. We use the strict $<$ relation instead of the $\leq$ used in the standard axioms above, so our order axioms take on a slightly different form. We give the axiom of irreflexivity of $<$:

```
Axiom "IRR" nil ["~ a1 < a1"];
```

the trichotomy axiom:

```
Axiom "TRI" nil ["a1 Equal a2 v a1 < a2 v a2 < a1"];
```

the axiom of transitivity of $<$:

```
Axiom "TRANS" nil ["a1 < a2 & a2 < a3 -> a1 < a3"];
```

and the axioms of monotonicity of addition and multiplication:

```
Axiom "MPLUS0" nil ["a1<a2 == a1+a3 < a2+a3"];

Axiom "MTIMES" nil ["0<a3 & a1<a2 -> a1*a3 < a2*a3"];
```

Because of technicalities involving assumptions about real numbers, the axiom `MPLUS0` is stated as a biconditional. From this we show the validity of the statement

```
a1 < a2 -> a1 + a3 < a2 + a3
```

which we name `MPLUS` and we often use this form within proofs.

It is a relatively simple exercise to show that this formulation of the order axioms is equivalent to the formulation given above using $\leq$.

Our next step is to formally define $\leq$ using the `DefinePredicate` command, which was also used to define `Equal`. This command takes four arguments, the first of which is an integer and the last three of which are strings. The first argument gives the arity of the predicate, the second gives the name of the predicate, the third gives the left side of the predicate to be defined, and the fourth is the right side of the definition.

```
DefinePredicate 2 "<=" "x1<=x2" "x1 Equal x2 v x1<x2";
```

The argument `"<="`, which may seem redundant, is needed because the third and fourth arguments cannot be parsed until `<=` is declared with the arity indicated by the first argument.

Finally we give the completeness axiom in two parts:

```
Axiom "SUP1" nil

["(Ex1.x1 E a1) & (Ax1.x1Ea1 -> x1 <= a2) -> Sup(a1) <= a2"];

Axiom "SUP2" nil

["(Ax1.x1Ea1 -> x1 <= a2)->(Ax1.x1Ea1 -> x1 <= Sup(a1))"];
```

The first part states that if a set `a1` is nonempty and is bounded above by `a2`, then the supremum of `a1` is less than or equal to the upper bound `a2`. The second states that if a set `a1` is bounded above by `a2`, then the supremum of `a1` is an upper bound for `a1`.

4.2 Intuitive Outline of the Main Proof

In an analysis class we might define the square root of a given positive real number x to be $r := \sup(\{z : z^2 < x\})$. It then remains to prove that the expected condition $r^2 = x$ actually holds. In a typical proof of this fact we derive a contradiction from both the possibility that $r^2 < x$ and that $x < r^2$, thereby concluding $x = r^2$, and so $\sqrt{x} = r$. To see how these contradictions arise, assume $x \in \mathbb{R}$, $x > 0$, and let $r = \sup\{z : z^2 < x\}$. We first consider the case that $r^2 < x$.

Then for all $0 < \varepsilon < r$, we have $r < r + \varepsilon$. Therefore

$$r^2 < x \leq (r + \varepsilon)^2$$

where the second inequality comes from the fact that $r = \sup\{z : zz < x\}$, and if $(r+\varepsilon)^2 < x$, r would not be an upper bound for $\{z : zz < x\}$. Therefore we have

$$0 < x - r^2 \leq 2r\varepsilon + \varepsilon^2$$

but

$$2r\varepsilon + \varepsilon^2 < 3r\varepsilon$$

so

$$0 < x - r^2 < 3r\varepsilon$$

and hence

$$0 < \frac{x - r^2}{3r} < \varepsilon$$

for any $0 < \varepsilon < r$. From this we can conclude $\frac{x-r^2}{3r} = 0$, thus $x = r^2$. But this is a contradiction to our assumption that $r^2 < x$. We can show that the case $x < r^2$ leads to a similar contradiction. Therefore $r^2 = x$, and thus $r = \sqrt{x}$.

4.3 Outline of the Main Proof as carried out in Marcel

The proof above contains several algebraic lemmas that we understand implicitly to be true, but that we need to prove explicitly in Marcel before we actually begin our proof of the fact that every positive real number has a square root. These lemmas, which include the lemma discussed above that half of any positive number is positive, follow relatively easily from the axioms given but are necessary to prove the main result. Some other lemmas we proved included the results that $0x = 0$, $x < y$ if and only if $-y < -x$, and if x is a real number, then $0 \leq x^2$.

An interesting feature of this particular proof is that it was carried out on two different versions of Marcel. We were able to save the theory (the lemmas and the partial proof) from the older version and load it onto the new version to complete the proof. We note that any commands referenced apply to the new version of Marcel, as the old version is now obsolete. The versions of the prover differ slightly in how lemmas are proved and introduced into the proof. In the old prover, lemmas were proved on the side, and then introduced in the proof by using `ThmCut`. This occasionally led to a rather fragmented main proof, as one had to stop the main proof, prove a lemma, then run the entire proof again to get to where one needed the lemma in the first place. The new prover has a command called `CutLemma` that allows the user to prove a lemma within the main proof (any irrelevant premises and conclusions are temporarily removed). Once the lemma has been proved, the user is taken back to where he or she started, but the lemma is included as a hypothesis. This allows for a better flow within the main proof, but has the drawback that if a mistake is made in proving the lemma, one has to stop the main proof and re-run everything to be in a position to correctly prove the lemma.

With several lemmas at our disposal, we are ready to tackle the main proof. The exact statement we prove is introduced by the following command:

```
 - Start "(Ax1.Real(x1)&Zero<x1 ->(Ex2.x2*x2 = x1))";
```

and the outline of our proof follows the intuitive proof plan above very closely. Our first steps are to use `r` and `l` commands to apply appropriate rules from our sequent calculus. The prover provides the witness `a1` for `x1` and we provide the witness `Sup({x3 | Zero < x3 & x3*x3 < a1})` for `x2`. For ease of notation, we let `Sup(S)` denote `Sup({x3 | Zero < x3 & x3*x3 < a1})`.

Our next step in the proof is to introduce the trichotomy axiom using the `ThmCut` command, which give us three possibilities:

- `a1 Equal Sup(S)*Sup(S)`
- `Sup(S)*Sup(S) < a1`
- `a1 < Sup(S)*Sup(S)`

The first scenario is easily shown to be valid, and our next step is to derive a contradiction from the premise `Sup(S)*Sup(S) < a1`. To do this we use one of the axioms concerning suprema: `SUP2` states

```
(Ax1.x1Ea1->x1 <= a2) -> (Ax1.x1Ea1->x1 <= Sup(a1))
```

so if a set is bounded above by some `a2`, then the set is bounded above by its supremum. Our first task then is to show that this axiom applies to our set `S`, so we show

`S` is bounded above by `a1 + One`. Once we have done this, we have as a premise that any element in `S` is less than or equal to `Sup(S)` and we are back to deriving a contradiction. In our intuitive outline above, we showed that for any $0 < \varepsilon < r$,

$$0 < \frac{x - r^2}{3r} < \varepsilon$$

where $r = \sup\{z : zz < x\}$. To introduce this idea of an epsilon with the prover, we use the `Cut` command, and we cut on the statement

```
(Ax2.Zero<x2 & Real(x2) & x2 < Sup(S) ->
[a1 + Minus(Sup(S)*Sup(S))]*Inv(Three*Sup(S)) <= x2)
```

Because we use the cut rule, we must show we have a valid sequent with this statement added to our premises on the left and a valid sequent with this statement included in our conclusions on the right.

The prover first gives us the statement on the right. We use the `r` and `l` rules to expand the quantifier, implication, and conjunctions, wherein the prover instantiates `x2` with `a23`. This `a23` will play the role usually taken by epsilon. Referring to our intuitive proof, we see our next goal is to show that

```
a1 <= (Sup(S) + a23)*(Sup(S) + a23).
```

To do this we again introduce the trichotomy axiom using `ThmCut` and derive a contradiction from the case that `(Sup(S) + a23)*(Sup(S) + a23) < a1`. This contradiction comes from our premise that any element of `S` is less than or equal to `Sup(S)`.

We show that if `(Sup(S) + a23)*(Sup(S) + a23) < a1`, then `Sup(S) + a23` is in `S`, and therefore `Sup(S) + a23 <= Sup(S)`, a contradiction.

The next possibility we face is that `a1 = (Sup(S) + a23)*(Sup(S) + a23)`. We use lemmas we have proved to multiply out the right side and because `a23 < Sup(S)` we can conclude that `a1 < a23*Sup(S) + Two*a23*Sup(S) + Sup(S)*Sup(S)`. From here we use the axioms we were given and lemmas we have proved to show that

```
[a1 + Minus( Sup(S)* Sup(S))] * Inv(Three* Sup(S)) < a23
```

so our cut was valid if `a1 = (Sup(S) + a23)(Sup(S) + a23)`. We follow a similar path and show that our cut is valid in the case that

```
a1 < (Sup(S) + a23)(Sup(S) + a23).
```

Now we are given the statement

```
(Ax2.Zero<x2 & Real(x2) & x2<Sup(S) ->
[a1 + Minus(Sup(S)*Sup(S))] * Inv(Three* Sup(S)) <= x2)
```

as a premise, and our task is to show that our original conclusion,

```
Sup(S)*Sup(S) = a1
```

is valid. As a technical note, it was at this point in the proof that we began using the new version of Marcel. From here we use several algebraic lemmas stating such facts as if a quotient is zero, its numerator is zero, and a lemma concerning epsilon, which is shown below, to complete the proof for the case `Sup(S)*Sup(S) < a1`.

```
- ThmDisplay "EPSLEMMA";
 EPSLEMMA:
 1:  Zero <= a1
 2:  Real(a1)
 3:  Zero < a2
 4:  Real(a2)
 5:(Ax1.Real(x1) & Zero < x1 & x1 < a2 -> a1 <= x1)
 |-
 1:  Zero = a1
```

This lemma, a version of which is at the heart of almost all proofs involving epsilon, says that if we have a non-negative real number a_1, and for *any* positive real number x_1 (which for technical reasons is here bounded above by the positive real number a_2), we have that $a_1 \leq x_1$, then we can conclude that $a_1 = 0$.

The case `a1 < Sup(S)*Sup(S)` follows very closely to the first case, with a few minor changes. The process of proving the second case went much more quickly than the first, perhaps partially due to already having proved lemmas that applied to both cases. While we did prove several new lemmas for the second case, they all either were closely analogous to lemmas needed in the first case or were lemmas we really *should* have proved during the first case, as they made the proof more concise.

Chapter 5

CONCLUSIONS

5.1 Prover Use Commentary

Marcel was designed as an educational tool, and as such has many possible benefits. One of the main concerns is how “pencil and paper” proofs compare to proofs completed using Marcel. Proofs generated through the use of a theorem prover are much “cleaner” than hand-written proof trees. Furthermore, the use of a computer allows students to prove much more complicated sequents than it would be feasible to prove using hand-written proof trees. Marcel also requires a rigor that is difficult to achieve when proofs are done using the traditional “pencil and paper”. Not only does the prover require such rigor, but it “grades” the students’ work as they go along. If a student successfully proves a sequent, he or she is rewarded with a “QED” from the prover. As long as there are no bugs in the theorem prover, a student receives immediate feedback as to whether or not his proof is correct. Of course, the possibility of bugs in the software is a real issue, but one that is beyond the scope of this paper. In fact, we found a few bugs during our main proof, all of which were quickly fixed by Holmes.

Using a computer to assist in proving mathematical statements may seem questionable at first, and indeed there are some areas in which the proving process using Marcel is noticeably different than the process of writing a proof-tree by hand. The first is in working with quantifier rules. Marcel's automation guides the student through the process, providing a witness automatically or requiring the user to name a specific witness, where appropriate. With sufficient practice using Marcel, a student should be more likely to take the correct action when dealing with quantifier rules in a pencil and paper proof. After all, it is these hand-written proofs we hope the prover will eventually benefit, as they make up the majority of a math student's proving experience.

One concept that is more difficult to keep track of during a proof using Marcel is the results of applying branching rules. Here pencil and paper proofs have the benefit that when a branching rule is applied, each branch is always visible on paper, although the branches are usually dealt with individually. When a branching rule is applied using the prover, one branch is presented for the user to work on, while the other branch is completely out of sight until the first has been proved. Especially within larger proofs, students are likely to forget that the second branch is still "waiting in the wings" and may not realize why it appears when they have completed the proof of the first branch. Marcel does provide tools for switching between branches of a proof, although they are a bit cumbersome to use.

The prover has a few other drawbacks, as well. There is a learning curve associated with working with Marcel for the first time, and there is a real risk that students will struggle with learning prover commands when the goal is really for them to learn the logic that is implemented by the prover. Another drawback is the "working backwards" proof-style associated with the sequent calculus. Students are often taught that in a proof we never start with what we are trying to prove. But using the sequent calculus that is exactly what we do: the rules are set up so that this is a well-founded style of proving that a sequent is valid. There is the risk that students will not recognize when this form of proof is acceptable and when it is not. Ideally one would like to introduce the use of a theorem prover in a basic analysis class to help students learn the structure of proofs. However, analysis has a learning curve of its own and asking students to learn to use a system like Marcel while learning the math implemented by the system may be more detrimental than helpful. It seems a theorem prover would be more suitable to use in a higher-level course in logic, indeed, that is where Marcel has been used here at Boise State.

Using Marcel in higher level classes could help students to better understand the underpinnings of analysis. While some may argue that the prover requires rigor that is unreasonable when working through a proof by hand, the prover provides exposure to this sort of rigor and we hope this will make the student realize just how much we "slide under the rug" during a typical proof in a class such as analysis. This could be especially useful for anyone who will later teach analysis, as the more understanding one has of a subject, the more tools they have to teach that subject.

We were initially exposed to Marcel during the Logic and Set Theory class, Math 502, in the fall of 2006. We personally found the prover helpful in reinforcing concepts in logic, and were intrigued enough to want to investigate further uses of Marcel. Indeed, most classmates with whom we talked felt it benefited their understanding of the course material. The class as a whole seemed to enjoy using Marcel, and what better way to motivate students than with an educational tool they find to be fun?

REFERENCES

[1] Samuel R. Buss, editor. *Handbook of Proof Theory*, volume 137, pages 10,16. Elsevier, 1998.

[2] Marcel Crabbé. The Hauptsatz for stratified comprehension: a semantic proof. *Mathematical Logic Quarterly*, 40:481 – 489.

[3] H. D. Ebbinghaus, W. Thomas, and J. Flum. *Mathematical Logic*, pages 52 – 57. Springer, 1994.

[4] Randall Holmes. Marcel source
`(http://math.boisestate.edu/∼holmes/marcelstuff/marcel.sml)`
and documentation
`(http://math.boisestate.edu/∼holmes/marcelstuff/marcelmanual.pdf).`

[5] Kenneth A. Ross. *Elementary Analysis: The Theory of Calculus*, pages 10 – 17. Springer, 1980.

[6] `http://math.boisestate.edu/∼holmes/marcelstuff/guildfiles`
Master files for the main proof.

Appendix A

THE REWRITE RULES

To see the validity of the rewrite rules, we note that `rwr` and `crwr` are direct implementations of the (= **A**) rule from our sequent calculus. The `rwl` and `crwl` rules are implementations of both (= **A**) and the negation rules from our sequent calculus. To see this, take the second proposition on the left side of our sequent (the proposition we wish to rewrite using the first proposition on the left side) and use (¬**S**) in reverse to put its negation in the first position on the right (the (¬**S**) rule is reversible, although we don't use it that way in Marcel). We then apply `rwr` or `crwr`, which are implementations of (= **A**), to rewrite the negated proposition. We can then use the (¬**A**) rule to move the rewritten original proposition to the left side of the sequent. If we then move the rewritten proposition to the second position on the left side of the sequent, we have executed the `rwl` or `crwl` command.

Next we give some simple examples of how the rewrite rules can be used. The first is `rwr`:

```
Line number 2:
1:  a1 = a2
|-
1:  a1 + a3 = a2 + a3
> val it = () : unit
- rwr [1];

Line number 3:
1:  a1 = a2
|-
1:  a2 + a3 = a2 + a3
> val it = () : unit
-
```

We see that the the first (and only) occurrence of `a1` (the left side of the first proposition on the left side of the sequent) in the first proposition on the right side of the sequent was replaced by `a2`.

The `crwr` command works similarly, but occurrences of the right side of the first equation are replaced by the left side:

```
Line number 2:
1:  a1 = a2
|-
1:  a1 + a3 = a2 + a3
> val it = () : unit
- crwr [1];
```

```
Line number 3:
1:  a1 = a2
|-
1:  a1 + a3 = a1 + a3
> val it = () : unit
-
```

Here, the first (and only) occurrence of `a2` (the right side of the first proposition on the left side of the sequent) in the first proposition on the right side of the sequent was replaced by `a1`. The `rwl` command works much like the `rwr` command, but uses the first proposition on the left side of the sequent to rewrite the second proposition on the left side of the sequent.

```
Line number 3:
1:  a1 = a2
2:  a1 + a3 < a2 + a4
|-
1:  a3 < a4
> val it = () : unit
- rwl [1];

Line number 4:
1:  a1 = a2
2:  a2 + a3 < a2 + a4
|-
1:  a3 < a4
> val it = () : unit
-
```

Likewise, the command `crwl` replaces occurrences of the right side of the first proposition on the left side of the sequent by the left side of the proposition.

```
Line number 3:
1:  a1 = a2
2:  a1 + a3 < a2 + a4
|-
1:  a3 < a4
> val it = () : unit
- crwl [1];

Line number 4:
1:  a1 = a2
2:  a1 + a3 < a1 + a4
|-
1:  a3 < a4
> val it = () : unit
-
```

To see how the integer list argument works, we consider a few ways to rewrite the sequent below:

```
1:  a1 = a2
|-
1:  a1 + a1 + a3 = a1 + a2 + a3
```

We could use the left side of the sequent to rewrite the first occurrence of `a1` on the right side.

```
Line number 2:
1:  a1 = a2
|-
1:  a1 + a1 + a3 = a1 + a2 + a3
> val it = () : unit
- rwr [1];

Line number 3:
1:  a1 = a2
|-
1:  a2 + a1 + a3 = a1 + a2 + a3
> val it = () : unit
-
```

Or we could rewrite the second occurrence of `a1`.

```
Line number 2:
1:  a1 = a2
|-
1:  a1 + a1 + a3 = a1 + a2 + a3
> val it = () : unit
- rwr [2];

Line number 3:
1:  a1 = a2
|-
1:  a1 + a2 + a3 = a1 + a2 + a3
> val it = () : unit
-
```

Or we could rewrite all occurrences of `a1` on the right side of the sequent.

```
Line number 2:
1:  a1 = a2
|-
1:  a1 + a1 + a3 = a1 + a2 + a3
> val it = () : unit
- rwr [];

Line number 3:
1:  a1 = a2
|-
1:  a2 + a2 + a3 = a2 + a2 + a3
> val it = () : unit
-
```

Or, of course, we could rewrite the one occurrence of `a2` on the right side of the sequent using `crwr`.

```
Line number 2:
1:  a1 = a2
|-
1:  a1 + a1 + a3 = a1 + a2 + a3
> val it = () : unit
- crwr [1];

Line number 3:
1:  a1 = a2
|-
1:  a1 + a1 + a3 = a1 + a1 + a3
> val it = () : unit
-
```

Appendix B

DECLARATIONS AND LEMMAS FOR THE MAIN PROOF

We first give the declarations and axioms for an ordered field, noting that most of these were actually defined using the old prover and have been changed to use the notation of the current prover.

Primitive Notions:

```
DeclarePredicate "Real" [0];

DeclareFunction "Zero" [0];

DeclareFunction "One" [0];

DeclareFunction "+" [0,0,0];

DeclareFunction "*" [0,0,0];

setprecrightabove "*" "+";

DeclarePredicate "<" [0,0];

DeclareFunction "Minus" [0,0];

DeclareFunction "Inv" [0,0];

DeclareFunction "Sup" [0,1];
```

Axioms:

First we define the outputs of all operations to be real numbers.

```
Axiom "RPLUS" nil ["Real(a1+a2)"];

Axiom "RTIMES" nil ["Real(a1*a2)"];

Axiom "RMINUS" nil ["Real(Minus(a1))"];

Axiom "RINV" nil ["Real(Inv(a1))"];

Axiom "RSUP" nil ["Real(Sup(a1))"];
```

The Commutative Laws:

```
Axiom "CPLUS" nil ["a1+a2=a2+a1"];

Axiom "CTIMES" nil ["a1*a2=a2*a1"];
```

The Associative Laws:

```
Axiom "APLUS" nil ["[a1+a2]+a3=a1+a2+a3"];

Axiom "ATIMES" nil ["[a1*a2]*a3=a1*a2*a3"];
```

The Distributive Law:

```
Axiom "DIST" nil ["a1*[a2+a3]=a1*a2+a1*a3"];
```

The Identity Laws:

We recall that these do not take quite the expected form due to the coercion of arbitrary arguments to real numbers.

```
Axiom "IPLUS" nil ["Real(a1)->a1+0=a1"];

Axiom "ITIMES" nil ["Real(a1)-> a1*1=a1"];
```

We define the relation “coerced to the same real”:

```
DefinePredicate 2 "Equal" "x1 Equal x2" "x1+0=x2+0";
```

The Inverse Laws:

```
Axiom "MINUS" nil ["a1+Minus(a1)=0"];

Axiom "INV" nil ["a1 Equal 0 v a1*Inv(a1)=1"];
```

We ensure that our model has more than one inhabitant:

```
Axiom "NONTRIV" nil ["~0 Equal 1"];
```

We give the basic properties of a strict linear order:

```
Axiom "IRR" nil ["~a1 < a1"];

Axiom "TRI" nil ["a1 Equal a2 v a1 < a2 v a2<a1"];

Axiom "TRANS" nil ["a1 < a2 & a2 < a3 -> a1 < a3"];
```

The monotonicity properties of order:

```
Axiom "MPLUSO" nil ["a1<a2==a1+a3<a2+a3"];

Axiom "MTIMES" nil ["0<a3 & a1<a2 -> a1*a3<a2*a3"];
```

We define less than or equal to:

```
DefinePredicate 2 "<=" "x1<=x2" "x1 Equal x2 v x1 < x2";
```

We give the least upper bound property in two parts:

```
Axiom "SUP1" nil
["(Ex1.x1 E a1) & (Ax1.x1Ea1->x1 <= a2)-> Sup(a1) <= a2"];

Axiom "SUP2" nil
["(Ax1.x1Ea1->x1 <= a2) -> (Ax1.x1Ea1->x1 <= Sup(a1))"];
```

Additional terms we defined:

```
DefineFunction 0 "Two" "Two" "One + One";
DefineFunction 0 "Half" "Half" "Inv(Two)";
DefineFunction 0 "Three" "Three" "One + One + One";
```

Additional theorems we proved:

We note that a few of the first theorems were proved by Holmes as a guide for the theorems we proved.

MPLUS:

```
|-

1:  a1 < a2 -> a1 + a3 < a2 + a3
```

NULLADD:

```
1:  Real(a1)

2:  Real(a2)

|-

1:  a1 + a2 = a1 == a2 = Zero
```

IPLUSa:

```
1:  Real(a1)

|-

1:  a1 + Zero = a1
```

TRIV:

```
|-

1:  a1 = a1
```

MP:
```
1:  P1 -> P2

2:  P1

|-

1:  P2
```

RZERO:
```
|-

1:  Real(Zero)
```

MZERO:
```
|-

1:  a1 * Zero = Zero
```

UNIQUEINV:
```
|-

1:
      Real(a2) & a1 + a2 = Zero -> a2 = Minus(a1)
```

UINV:
```
1:  Real(a2)

2:  a1 + a2 = Zero

|-

1:  a2 = Minus(a1)
```

IPLUSb:
```
1:  Real(a1)

|-

1:  Zero + a1 = a1
```

MINUS2:
```
|-

1:  Minus(a1) + a1 = Zero
```

MINUSLESS:
```
|-

1:  a1 < a2 == Minus(a2) < Minus(a1)
```

EQUALITY:
```
1:  a1 Equal a2

2:  Real(a1)

3:  Real(a2)

|-

1:  a1 = a2
```

NEGINEQ:
```
1:  a1 < Zero

|-

1:  Zero < Minus(a1)
```

NEGCOMM:
```
|-

1:  a1 * Minus(a2) = Minus(a1 * a2)
```

DOUBLENEG:

```
1:  Real(a1)

|-

1:  Minus(Minus(a1)) = a1
```

TIMESDOUBNEG:

```
|-

1:  Minus(a1) * Minus(a2) = a1 * a2
```

MTIMESNEG:

```
1:  a3 < Zero

2:  a1 < a2

|-

1:  a2 * a3 < a1 * a3
```

SQUARENONNEG:

```
1:  Real(a1)

|-

1:  Zero <= a1 * a1
```

MZERO2:

```
|-

1:  Zero * a1 = Zero
```

EqualSymm:
```
1:  a1 Equal a2

|-

1:  a2 Equal a1
```

REALONE:
```
|-

1:  Real(One)
```

NOTBOTH:
```
1:  a2 < a1

2:  a1 < a2

|-
```

NOTBOTH2:
```
1:  a1 Equal a2

2:  a1 < a2

|-
```

NOTBOTH3:
```
1:  a2 < a1

2:  a1 Equal a2

|-
```

ZEROLESSONE:
```
|-

1:  Zero < One
```

SubLemma:

```
|-

1:  a1 * a2 = [a1 + Zero] * a2
```

SquareOrder1:

```
1:  a1 < a2

2:  Zero < a1

3:  Zero < a2

|-

1:  a1 * a1 < a2 * a2
```

SquareOrder:

```
|-

1:  Zero < a1 & Zero < a2
      -> (a1 < a2 ==
     a1 * a1 < a2 * a2)
```

LESSTYPE:

```
|-

1:  a1 + Zero < a2 + Zero -> a1 < a2
```

LESSTYPEa:

```
1:  a1 + Zero < a2 + Zero

|-

1:  a1 < a2
```

RETRACT:

```
|-

1:  [a1 + Zero] + Zero = a1 + Zero
```

DIST2:
```
|-

1:  [a1 + a2] * a3 = a1 * a3
      + a2 * a3
```

TYPECONVERT:
```
|-

1:  a1 * One = a1 + Zero
```

MPLUSCONV:
```
1:  a1 + a3 < a2 + a3

|-

1:  a1 < a2
```

POSINV:
```
1:  Zero < a1

|-

1:  Zero < Inv(a1)
```

SomethingLess:
```
|-

1:  a1 + Minus(One) < a1
```

SOMETHINGBIGGER:
```
|-

1:  a1 < a1 + One
```

RTWO:

```
|-

1:  Real(Two)
```

RHALF:

```
|-

1:  Real(Half)
```

TWOPOS:

```
|-

1:  Zero < Two
```

HALFPOS:

```
|-

1:  Zero < Half
```

TWOHALVES:

```
|-

1:  Half + Half = One
```

MHALFPOS:

```
1:  Zero < a1

|-

1:  Zero < Half * a1
```

HalfLess:
```
1:  Zero < a1

2:  Real(a1)

|-

1:  Half * a1 < a1
```

SQPOS:
```
1:  Zero < a1

|-

1:  Zero < a1 * a1
```

SquareOrder2:
```
1:  a1 * a1 < a2 * a2

2:  Zero < a1

3:  Zero < a2

|-

1:  a1 < a2
```

SqLemma:
```
1:  Zero < a1

2:  Real(a1)

|-

1:  a1 < [a1 + One] * [a1 + One]
```

EPSLEMMA:

```
    1:  Zero <= a1

    2:  Real(a1)

    3:  Zero < a2

    4:  Real(a2)

    5:  (Ax1.
         Real(x1) & Zero < x1 & x1 < a2 -> a1 <= x1)

    |-

    1:  Zero = a1
```

FOIL:

```
    |-

    1:  [a1 + a2] * [a1 + a2] = a1 * a1 +
         Two * a1 * a2
          + a2 * a2
```

SupLemma:

```
    1:  Real(a1)

    2:  Zero < a1

    |-

    1:  (Ex2.x2 E {x3|Zero < x3 & x3 * x3 < a1})
```

HALFTWO:

```
    |-

    1:  Half * Two = One
```

ORDERLEMMA1:

```
1:  Zero < a1

2:  a1 < a2

|-

1:  a1 * a1 + Two * a1 * a2 + a2
      * a2 < a1 * a2
      + Two * a1
      * a2 + a2 * a2
```

THREEPOS:

```
|-

1:  Zero < Three
```

TIMESPOS:

```
1:  Zero < a1

2:  Zero < a2

|-

1:  Zero < a1 * a2
```

LIKETERMS1:

```
1:  Real(a1)

|-

1:  a1 + Two * a1 = Three * a1
```

BDED:

```
1:  Real(a1)

2:  Zero < a1

|-

1:  (Ax5.
     x5 E {x7|Zero < x7 & x7 * x7 < a1}
      -> x5 <= a1 + One)
```

SUPPOS:

```
1:  Real(a1)

2:  Zero < a1

|-

1:  Zero < Sup({x77|Zero < x77 & x77 * x77 < a1})
```

ZPROD:

```
1:  Real(a2)

2:  a1 * a2 = Zero

3:  Real(a1)

|-

1:  a1 = Zero

2:  a2 = Zero
```

ZFRAC:

```
1:  Zero < a1

2:  Zero =
     [a1 + Minus(
    Sup({x77|Zero < x77 & x77 * x77 < a1})
     * Sup({x78|
    Zero < x78 & x78 * x78 < a1}))] * Inv(Three
     * Sup({x79|
    Zero < x79 & x79 * x79 < a1}))

3:  Real(a1)

|-

1:  Zero = a1 + Minus(
    Sup({x77|Zero < x77 & x77 * x77 < a1})
     * Sup({x78|
    Zero < x78 & x78 * x78 < a1}))
```

EQSUPa1:

```
1:  Real(a1)

2:  Zero = a1 + Minus(
    Sup({x78|Zero < x78 & x78 * x78 < a1})
     * Sup({x79|
    Zero < x79 & x79 * x79 < a1}))

|-

1:  a1 =
    Sup({x78|Zero < x78 & x78 * x78 < a1})
     * Sup({x79|
    Zero < x79 & x79 * x79 < a1})
```

NEGINEQ2:

```
     1:  Zero < a181

     |-

     1:  Minus(a181) < Zero
```

SUPMINUSPOS:

```
     1:  a181 < Sup({x81|Zero < x81 & x81 * x81 < a1})

     |-

     1:  Zero <
          Sup({x81|Zero < x81 & x81 * x81 < a1})
           + Minus(a181)
```

SUBLESS:

```
     1:  Zero < a181 * a181

     |-

     1:  Two *
          Sup({x106|Zero < x106 & x106 * x106 < a1})
           * a181
           + Minus(a181 * a181) < Two *
          Sup({x107|Zero < x107 & x107
           * x107 < a1}) * a181
```

ADDEQ:

```
     1:  a1 = a2

     |-

     1:  a1 + a3 = a2 + a3
```

ZFRAC2:

```
1:  Zero < a1

2:  Zero = [
     Sup({x151|Zero < x151 & x151 * x151 < a1})
      * Sup({x152|
     Zero < x152 & x152 * x152 < a1})
      + Minus(a1)] * Inv(Two
      * Sup({x153|Zero < x153 &
     x153 * x153 < a1}))

3:  Real(a1)

|-

1:  Zero =
     Sup({x151|Zero < x151 & x151 * x151 < a1})
      * Sup({x152|
     Zero < x152 & x152 * x152 < a1})
      + Minus(a1)
```

EQSUPa12:

```
1:  Real(a1)

2:  Zero =
     Sup({x151|Zero < x151 & x151 * x151 < a1})
      * Sup({x152|
     Zero < x152 & x152 * x152 < a1})
      + Minus(a1)

|-

1:
     Sup({x151|Zero < x151 & x151 * x151 < a1})
      * Sup({x152|
     Zero < x152 & x152 * x152 < a1}) = a1
```