Documentation for `marcel.sml`

M. Randall Holmes

December 2, 2015

Contents

1 Version Notes

Sept. 8, 2011: This is the manual of December 15 2010 with version notes section added. The September 8th version of the Marcel source modifies the behavior of OneConclusion mode so that it actually has no conclusion when proving a contradiction, but the display shows an illusory $\perp$ (the absurd) as the conclusion. In this version (for class use) OneConclusion is the default mode.

2 Introduction

This file is the manual for the software implemented in the file `marcel.sml` which is in turn a reimplementation of the older version of the same concept found in `marcelsequent.sml`.

The program is a proof editor and checker implementing an extension of a sequent calculus first brought to my attention by Marcel Crabbé (after whom it was named) in his paper [**?**], in which he gives a semantic proof of cut-elimination for it. The sequent calculus is an implementation of Quine's set theory New Foundations with no extensionality. The consistency of New Foundations remains an open problem, but the consistency of New Foundation without the axiom of extensionality was shown by Jensen in [**?**], in which he actually showed the consistency of the stratified comprehension axiom of NF with the weak form of extensionality which requires that two objects with elements must be equal if they have the same elements. Marcel Crabbé himself showed that NF with no extensionality axiom at all (which he calls SF) interprets NFU, so SF and NFU have the same strength. This is the same level of strength as Zermelo set theory with bounded quantification in set definitions; this is more than adequate for all of mathematics except the higher reaches of set theory (and can readily be made much stronger).

We found it easier to understand the point of the work of Crabbé on sequent calculus with a concrete implementation of the sequent calculus in question at hand. In the process of implementing it, we adjoined equality (basically by defining $x = y$ as $(\forall z.x \in z \equiv y \in z)$) and adjoined the weak extensionality of NFU. We do not know if the system augmented with the weak extensionality rule enjoys cut elimination.

At the same time, we discovered the charm of the implementation of mere logic found in sequent calculus. We have now used the logical component of

the theorem prover (touching very briefly if at all on the set theory) to teach logic at the undergraduate level and beginning graduate level several times, with noticeable success. We have directed a master's thesis in which an elementary result of real analysis was shown using the prover. We wrote an unsuccessful grant proposal seeking support for a research project investigating the application of this system to education, and will attempt this again.

The original implementation `marcelsequent` was not designed to have quite the scope that its latest version attempts to cover, and the structure of the code was beginning to resemble patches on patches, so we decided to reimplement it from the bottom up in an organized fashion, and, at the same time as producing a better structured piece of software, produce adequate documentation. While in general terms the new prover is quite similar to the old one, it does not run its script files; however, it is possible to save theory files (including detailed proofs) to the new prover which were made under the latest version of the old prover.

3 The Language: Parsing, Display, Declarations

The original version `marcelsequent` of this prover had a rather limited notation. It had natural notation for propositional connectives, quantifiers, and set-builder notation $\{x \mid \phi\}$. There was provision for user-declared and defined predicates and functions (the latter including constants): to make the parser simple, all predicates were strings of lower-case letters starting with `#` and all functions (and constants) were strings of lower-case letters starting with `*`. Predicates and functions were always prefix (as `#predicate(arg1,arg2)`, `*function(arg1,arg2,arg3)`,`*constant`). During the fall of 2006, when I was using the prover with graduate students in the logic and set theory course, I updated the prover to support a notation similar to what is found here, but this was done by installing a postprocessor on top of the original notation, and the resulting parsing process was rather unstable.

Here we describe the language of the new version `marcel.sml`.

3.1 Tokens

A string to be parsed is broken into tokens.

These can be categorized as follows:

parentheses and brackets: `(`, `)`, `[`, `]`, { ,}. It is useful to note that parentheses are used to enclose terms denoting propositions and to enclose argument lists, while brackets are used to enclose terms denoting objects for purposes of grouping. `[a+b]+c` is well formed and `(a+b)+c` is not (if `+` is declared as an infix operator on objects (a binary function). Braces are used only in set notation.

other special characters: $\sim$ (the sign of negation); `.` (separates the head from the rest of the expression in expressions with variable binding); `|`, same as `.` but only in set notation, `,` (the comma), separates items in argument lists and pair notation. These characters have only the indicated role and cannot appear as part of any identifier (except the quoted strings, not yet implemented).

identifiers: These are of two shapes. An alphanumeric identifier consists of 0 or 1 capital letters, followed by any number of lower case letters (0 is possible), followed by any number of digits (0 is possible). Identifiers consisting of a single letter (upper or lower case) followed by 1 or more digits have reserved meaning (cannot be declared). An identifier cannot consist of two letters (the first being either upper or lower case) followed by 1 or more digits; the parser breaks this into the initial letter followed by a term of the reserved form. Two letter identifiers not followed by digits are all right. Identifiers can also be strings of special characters other than the ones reserved above. The empty string is not an identifier. A further kind of special identifier, not yet implemented, will be the *theorem name*: this will be an arbitrary string beginning and ending with `"` (and not containing any intermediate occurrences of this character). This is the only kind of identifier which can (will) contain the double quote.

note on whitespace: spaces and carriage returns are construed as whitespace, which has no role except to prevent adjacent identifiers from being concatenated.

The meanings of the single-character special tokens of the first two classes are fixed. The meanings of some identifiers are reserved. Some reserved identifiers have overloaded uses; no user declared identifier can be overloaded except as noted under the first heading.

`p1, p2:` The projection operators for the pair.

`x1,x2,x3...:` Bound object variables.

`a1,a2,a3...:` Free object variables.

`U1,U2,U3...:` Unknown object variables.

`P1,P2,P3...:` Predicate and propositional variables.

`R1,R2,R3...:` Predicate and propositional variables (binary infix form).

`F1,F2,F3...:` Function variables (not yet implemented).

`I1,I2,I3...:` Infix variables (binary infix form of function variables, not yet implemented).

`&, v, ->, ==, <-, =/=:` Propositional connectives.

`=, E:` Binary predicates of equality and membership.

`A, E:` Universal and existential quantifiers (these are binders). Notice that `E` is overloaded: users cannot similarly overload declared or defined notations.

`::` This is a special infix operator which enables restriction of complex binders to sets. It can be used only in very restricted ways (it can only appear as the top level infix of the head of a variable binding expression). It is intended that this will eventually be freely usable inside expressions built with variable binding operations.

There are two types of term: terms denoting propositions and terms denoting objects. Identifiers whose meaning is not reserved can be declared as operators or as binders. Operators and binders are further subdivided semantically by the types of their inputs and outputs and syntactically in that those of arity 0 are constants (and so show no argument lists) and those of arity 2 are expressed using infix notation (with user-defined operator precedence).

Operators are of three kinds, connectives, predicates and functions. No overloading other than that already found among the reserved identifiers is allowed.

Connectives take propositional input and give propositional output: there is no provision for user definition or declaration of connectives, so the binary propositional connectives shown above (and the special negation operator) are the only connectives. Infix notation is the only notation supported.

Predicates take object input and give propositional output. There are reserved predicates `=` and `E` of equality and membership as noted above. Any predicate has fixed arity. Users can both declare and define new predicates.

Functions take object input and give object output. There are reserved functions `p1` and `p2` declared above. The pair operation is in internal respects a reserved function of arity 2 but has different syntax. Similar remarks apply to the projected but not yet implemented operation of function application. Numerals are reserved constants (functions of arity 0). Quoted strings (theorem names) are reserved constants as well. Users can both declare and define new functions.

Binders are of three kinds, quantifiers, set binders, and function binders. It will be possible for users to define binders of all three kinds, but this is not yet supported.

The quantifiers `A` (universal quantifier) and `E` (existential quantifier) are primitives of the prover. Sentences are of the form `(B term.prop)` where `B` is the binder, `term` stands for an object term (usually but not always simply a bound variable) and `prop` stands for a proposition term.

Set binders take proposition input and give object output `[B term.prop]` is an object term if `B` is a set binder). The set binder {`term`|`term`} is internally a set binder, but has special syntax.

Function binders take proposition input and give object output `[B term.term]` is an object term if `B` is a function binder; note the use of brackets). The lambda binder `[L term . term]` is implemented in the latest version.

3.2 Syntax

Syntax of argument lists: A string obtained by concatenating a sequence of one or more object terms, separating them with commas `,`, prepending `(`, and postpending `)`, is an argument list.

Syntax of object terms: brackets: If `term` is a term then `[term]` is a term (with the same denotation; this is used for grouping). If `term1`,...,`termn` are terms, then `[term1,...,termn]` is a term

(ordered n-tuple). The n-tuple `[term1,...,termn]` for $n > 2$ is actually the same as the pair `[term1,[term2,...,termn]]`.

function binder term: If `term1` is an object term (often just a single bound variable) and `term` is an object term, and `B` is a function binder token, then `[B term1.term]` is an object term.

set term: If `term1` is an object term (often just a single bound variable) and `prop` is a proposition term, then {`term1 | term`} is an object term.

set binder term: If `B` is a set binder token, `term1` is an object term (often just a single bound variable) and `prop` is a proposition term, then `[B term1.term]` is an object term.

function variable term: (not yet implemented) If `n` is a numeral and `(...)` is an argument list, then `Fn(...))` is an object term. It is intended that `Fn(t1,t2)` will be displayed as `t1 I1 t2` (which last will also be a parsable form).

prefix term: If `f` is a function token of arity 0, then `f` is an object term (a constant). If `f` is a function token of arity n and `(t1,...,tn)` is an argument list of length n then `f(t1,...,tn)` is an object term. If $n = 2$, the parser will accept the term, `f(t1,t2)`, but it will be displayed as `t1 f t2`.

pair: If the parentheses in an argument list are replaced by `[` and `]`, the result is a tuple term. Tuples of the form `[x1,[x2,x3]]` are displayed as `[x1,x2,x3]` (as noted above under brackets).

infix term: If `term1` is a term and `term2` is a term, and `i` is a function token of arity 2, then `term1 f term2` is an object term.

variables: If `n` is a numeral, then `xn`, `an` and `Un` are variables of various sorts (bound, free, and unknown, respectively).

operator precedence: Every infix operator (connective or function, but they act in separate domains) is assigned an integer precedence, which can be freely set by the user using commands explained below (and can be harmlessly reset during prover sessions). The default precedence is 0, and precedences are always non-negative. Higher precedences bind more tightly than lower ones; even precedence group to the right and odd ones to the left. The user commands are set up so that users do not need to be aware of these numbers.

Syntax of proposition terms:

Parenthesis: If `prop` is a proposition term, then `(prop)` is a proposition term with the same denotation. This is used for grouping.

Quantifier term: If `term1` is an object term (often a single bound variable) and `prop` is a proposition term, and `B` is a quantifier token, then `(B term1.prop)` is a proposition term. At the moment `A` and `E` are the only supported quantifiers, but user-defined binders will be implemented.

Negation: If `prop` is a proposition term, then ∼`prop` is a proposition term.

Connective: If `o` is a connective token, and `prop1` and `prop2` are proposition terms, then `prop1 o prop2` is a proposition term. No provision is made for declaration or definition of additional connectives.

Note on operator precedence: Precedence for the connectives is preset, and though it can be reset by the user this is not recommended.

Propositional and predicate variables: If `n` is a numeral, then `Pn` is a propositional term (propositional variable). If `(...)` is an argument list, then `Pn(...)` is a propositional term (with variable predicate). `Pn(t1,t2)` is accepted by the parser but displayed as `t1 Rn t2`, which is also a parsable form.

Prefix: If `q` is a predicate token of arity 0, then `q` is a propositional term (defined constant proposition). If `q` is a predicate token of arity n and `(t1,...,tn)` is an argument list of length n, then `q(t1,...,tn)` is a propositional term. If $n = 2$, `q(t1,t2)` is accepted by the parser but stored and displayed as the infix term `t1 q t2`.

Infix: If `term1` is an object term, `term2` is an object term, and `q` is a predicate token of arity 2, it follows that `term1 q term2` is a proposition term.

Restriction term: If `term2` is an object term and `term3` is an object term, it follows that `term2 : term3` is an object term, but such a term may only appear in the role `term1` in the definitions of quantifiers, set binder terms, sets, and function binder terms: in all other positions the colon is treated as an undeclared identifier. It is intended to allow `:` to appear in more general contexts but only in heads of binder terms.

It is important to notice that the parser relies in handling identifiers on declarations of their type and arity, so there is no separate process of declaration checking: what is not defined or declared is also not parsable. This was not the case with the old prover, though it began to be true in the latest versions.

4 Set Theory and Defined Notions

We next consider the declaration and definition of identifiers. This requires some understanding of the set theory in which we work.

The default set theory for this prover (it will be possible to set it to alternative theories, including something more like the standard ZFC) is the version of Quine's New Foundations with weak extensionality (we are permitted to have many objects with no elements; objects with some elements are equal if they have the same elements).

New Foundations is not actually a typed theory but it is best understood initially via a typed theory. The types in the related type theory are indexed by the natural numbers. Type 0 is inhabited by unanalyzed individuals. Type 1 is inhabited by sets of type 0 objects. Type 2 is inhabited by sets of type 1 objects. In general, type $n+1$ is inhabited by sets of type n objects.

Further, we stipulate that the two projections of an ordered pair are the same type as the pair (which means that our pair is not the usual Kuratowski pair, which does make sense in this type theory but is two types higher than its projections).

Objects in NFU do not actually have types. In NFU, the requirement is more subtle: any definition of an object (by set builder notation or by the definition commands) should have the property that all variables in the definition can be consistently assigned types (constants are allowed to be assigned multiple types, because a constant is presumed to have analogues in each type).

A predicate declaration is of the form

`DeclarePredicate q [t1,...,tn]` where `[t1,...,tn]` is a list of n integers: the predicate declared has the name `q` and has arity n, with its first argument assigned type `t1`, its second argument type `t2`, and so forth. (In a particular context, the arguments will be typed with a fixed offset i: the type of the nth argument will be t_n+i). For example, a declaration of membership `E` (it is predeclared) is of the form

`DeclarePredicate E [0,1]`

which tells us that `E` is a binary predicate whose second argument is always one type higher than its first argument (but the types of the arguments in a specific context might be 16 and 17 rather than 0 and 1).

A function declaration is of the form

`DeclareFunction f [t0,t1,...,tn]`. Here `t0` is the type to be assigned to the whole term when the arguments are assigned types `t1,...,tn`, and again all of these types may be offset by a fixed amount. If `Sing(x1)` is the singleton operation, then the type list would be `[1,0]`, because the singleton is one type higher than its sole element (its argument in the syntax). But the type of a particular singleton term might be 5 and the type of its argument 4.

Definition commands are of the following forms.

`DefinePredicate n "q" "q(x1,...xn)" P`

and

`DefineFunction n "f" "f(x1,...xn)" T`

The arguments are, in order, the arity of the token to be declared, the token to be declared, the left side of the definition with bound variable parameters, and the right side of the definition. The prover will type check the definition; if the term type checks the prover will automatically generate the type information for this token (we do not have to supply it as we did for declarations); if it does not type check the definition still succeeds and can be used in certain contexts, but it cannot be used inside any set expression or set binder or function binder expression (at least, not effectively). The prover will say something like `Stratification error` in this case.

The best approach for the naive user is probably to take the position that we are really reasoning in type theory and so that a definition that does not type correctly (is not stratified) is a failed definition.

An example of a definition which does not type correctly is the definition $n \cup \{n\}$ of the successor in the usual set theory, in which n appears with two different types (as an element of the given set and as an element of an element of the given set: so if the set were assigned type i, then n would be assigned type $i-1$ and type $i-2$, which is a conflict).

When user defined binders are introduced, some commentary will be added here. Binders do have a relationship to stratification: in the current implementation, the only term binder construction is the set construction, and it is sufficient to note that a set notation is one type higher than the variable bound in it.

4.1 The Stratification Algorithm

In this section, we describe the stratification (typability) algorithm of the prover. The stratification algorithm assigns a type to each object term which contains a bound variable. If it is unable to assign types, it issues an error message. We will discuss below the occasions on which it is invoked.

The types are represented internally by pairs of integers (m, n) which may best be read as $t_m + n$ where t_m is an unknown type. To add a contant i to $t_m + n$ yields $t_m + (n + i)$. If we discover that $t_m + n$ and $t_r + s$ are the same type, where $r < m$, then we record the identification of t_m with $t_r + (s - n)$, and retype the term with type $t_m + n$ to have type $t_r + s$. In the future, any term with type $t_m + k$ is automatically retyped with type $t_r + (k + (s - n))$. These identifications may be chained: the type assigned to any term has the minimum possible unknown type index. If $m = r$ and $n = s$, nothing is done; if $m = r$ and $n \neq s$ a type error is reported.

Each primitive predicate and most defined predicates have a prescribed template of types for their arguments: these are relative, in the sense that a fixed offset may be added to all of the types. If a sentence `predicate(t1,...tn)` is typed, and the type of argument i of `predicate` is x_i, and each argument `ti` has been assigned type y_i based on its internal structure, we choose a fresh unknown type t_j and unify the types y_i with the corresponding types $t_j + x_i$. A primitive predicate term is treated similarly, except that a type x_0 is assigned to the whole term `function(t1,...tn)` as well as to the arguments `ti` and the type $t_j + x_0$ needs to be unified with any type y_0 assigned to the whole term on the basis of its context, in addition to the unifications of the types y_i assigned to its arguments with the corresponding types $t_j + x_i$ for $1 \leq i \leq n$.

Defined predicates and functions are assigned vectors of types (for their arguments and for the term itself in the case of functions) if the definition of the term is stratified. If a definition cannot be stratified, it follows that the defined predicate or function cannot be used effectively in variable binding contexts.

Binders are treated similarly: there are fixed displacements between the types assigned to the head, body, and whole term of a binding expression in case any of these are terms. As always, types are not assigned to propositions.

The stratification algorithm is invoked when a sentence `t` $\in$ {`x | ...x...`}is to be converted to `...t...`: the prover only allows this substitution if the term {`x | ...x...`} is stratified.

`marcel.sml` has a much better and more easily understood stratification algorithm than `marcelsequent.sml` did: the implementation of the algorithm loosely described above is fairly simple.

5 Proof Trees and The Logic of Propositions

In this section we start to introduce the basic mechanics of our logic.

5.1 Sequents

A *sequent* is a pair of finite lists of propositions. A sequent is *valid* if any assignment of values to variables appearing in the sequent which makes all the propositions on the left true also makes at least one of the sequents on the right true. Another way of putting it is that any assignment of values to variables in the sequent makes something on the left false or something on the right true (or both).

A sequent is usually written

$$P_1, P_2, \ldots \vdash Q_1, Q_2, \ldots$$

. In the prover, this is displayed vertically:

```
1. P1

2. P1

...

|-

1. Q1

2. Q2

...
```

When operating the prover, one is usually viewing a sequent which one is attempting to show to be valid. What one proves in the end is that a sequent is valid. Notice that to say that P is a theorem is the same as to say that $\vdash P$ (a sequent with the empty list on the left and the one element list with just P on the right) is valid.

To begin a proof one can type

`StartSequent L M`

where `L` and `M` are lists of proposition terms, to prove the sequent with `L` as left list and `M` as right list, or

`Start P`

where `P` is a proposition term, which is the same as the StartSequent comand with `L` the empty list `nil` and `M` the list `[P]`.

For concreteness we present two examples:

`Start "a1=a1";`

and

`StartSequent ["P1","P1->P2"] ["P2"];.`

5.2 Proof Trees

A (partial) proof of a sequent S is one of three things:

a theorem reference: The pair of S and (the name of) a theorem (already

proved valid sequent) of which S is an instance is a proof of S. This is written in the form $(S, \texttt{ref}(thm))$.

a proof tree: The pair of S and a list of partial proofs of statements T_i such that validity of all of the T_i's entails validity of S immediately (by some rule of our logic).

triviality: This is actually a special case of the previous: the pair $(S, [])$ of S and the empty list is a proof of S if the validity of S is obvious (by some rule of our logic).

goal: $(T, \texttt{goal})$ is a partial proof of T; this is just a hope that we can find a proof.

Clearly a partial proof of S which contains no goals (in the natural sense) is a proof of S.

A partial proof of S is a tree structure (a proof tree) in which the root is S and the leaves are the theorem references, trivialities, and any remaining goals. In any proof under `marcel`, the user is viewing the leftmost leaf of a proof tree: the proof tree is always automatically reordered to put a goal in leftmost position if there is a goal anywhere in the tree (and if there is no goal we are done: we have a proof of the sequent originally entered).

The operations carried out on the proof tree are of the following kinds. A goal $(S, \texttt{goal})$ may be replaced with (S, L) where L is a list of goal proofs of statements from which S can be deduced by a rule of our logic. There will be more discussion below of the forms of sequent rules. A goal $(S, \texttt{goal})$ may be replaced with $(S, \texttt{nil})$ if S is "obviously" true or by $(S, \texttt{ref}(\texttt{thm}))$ if S is an instance of the theorem "thm": in either of these cases, the proof tree will then be automatically reordered to present another goal in leftmost position. The goal $(S, \texttt{goal})$ may be replaced with $(S', \texttt{goal})$ where S' is obtained from S by reordering and/or omitting propositions in the left and right lists of S. Finally, global substitutions can be made for the unknown variables `Un`. The last command differs from the others in having an effect on the entire proof, not a single goal. The `NextGoal` command allows the user to cycle through all the goals in the proof: to decide what global substitution might be best might require that more than one goal be examined; the `SwapGoals` command cycles through subgoals near the current goal (it is used where the default order in which a sequent rule presents subgoals is for some reason undesirable).

5.3 Sequent Rules in General

What is missing from the abstract account above is a description of the rules by which validity of sequents may be recognized or deduced from the postulated validity of other sequents.

The issue of when a sequent is an instance of a theorem will be discussed later.

Sequents are regarded as trivial under two circumstances: a sequent $A, P_2, P_3, \ldots \vdash A, Q_2, Q_3 \ldots$ is recognized as valid (when the user issues the `Done` command) and a sequent $P_1, P_2, \ldots \vdash A = A, Q_2, Q_3 \ldots$ is recognized as valid when the omnibus `r` command (the general command for applying a sequent rule on the right) is issued. Identity of terms up to renaming of bound variables is recognized.

It is a general feature of sequents that if $A \vdash B$ is a valid rule, so is $\Gamma \cup A \vdash B \cup \Delta$. Sequent rules inherit a similar feature: if the validity of $A \vdash B$ can be inferred from the validity of $A_1 \vdash B_1, \ldots, A_i \vdash B_i$, then the validity of $A \cup \Gamma \vdash B \cup \Delta$ can be inferred from the validity of $A_1 \cup \Gamma \vdash B_1 \cup \Delta, \ldots, A_i \cup \Gamma \vdash B_i \cup \Delta$, and we regard this as an application of the same rule. Most of the rules we use apply to a single proposition in the sequent, either the first on the left side or the first on the right, and the rest of the sequent is copied into the generated premise or into each of the generated premises. The exceptions are the triviality rule which compares the first terms on both sides and the rules for rewriting, which allow an equation in the first position on the left to rewrite either the second proposition on the left or the first proposition on the right.

So with each basic logical operation (connective or quantifier) two rules are associated, a left rule and a right rule. Some alternatives and refinements are provided, especially in connection with equality, and there are the additional rewrite rules as well.

5.4 Specific Sequent Rules: Connectives

We give left rules and right rules for the commonly used connectives, excluding converse implication and xor. The notations Γ and Δ represent arbitrary finite sets of propositions. The sentence closest to the turnstile on the right or left is the first sentence in the right or left list in the prover's presentation. The premise on the left is the one which is presented first by the prover when the rule is applied.

In the old prover `marcelsequent` the biconditional was handled by definitional expansion; here a little more work is done by the prover so a proof involving a biconditional has fewer steps.

In our experience the rule which it is somewhat difficult to get used to is the left rule for implication, though with a little thought it can be seen to express the rule of *modus ponens*.

5.5 Axiom

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

5.6 Left rules

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

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

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

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

$$\frac{\Gamma, A \to B, B \to A \vdash \Delta}{\Gamma, A \equiv B \vdash \Delta}$$

5.7 Right rules

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

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

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

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

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

6 Variables and Substitution

There are four kinds of variables in the language of `marcel`, and a fifth (function variables) may be introduced in the future. The predicate/proposition variables `Pn` and the unimplemented function variables `Fn` will not be mentioned in this section. The three sorts of variable that are mentioned in this section have in common that they refer to objects rather than propositions, predicates, or formal operations (whose ontological status is more vexed).

These three kinds of variable are the *bound variables* `x1,x2,x3...`, the *free variables* `a1,a2,a3...`, and the *unknown variables* `U1,U2,U3...`.

Our approach to bound and free variables in the old version `marcelsequent` was influenced by our pleasure with the discovery that the usual notion of a free occurrence of a bound variable in an expression could be entirely avoided in the treatment of substitution by aggressively renaming bound variables. However, `marcel` allows more complex variable binding expressions and it turns out that the concept of free occurrence of a bound variable in a subexpression becomes very difficult to avoid in this more complex context (and we broke down and used it).

In `marcelsequent`, where an occurrence of a bound variable was simply a typographical occurrence, there was no requirement that terms entered to the prover have all bound variables actually bound in the usual sense: in `marcel` it is forbidden to enter a term at the top level which contains an occurrence of a bound variable which is free in the usual sense.

We look at the details of the definition of substitution of a term T for a variable v (free, bound or unknown) in a variable binding context $(\forall U.V)$ (the head binder U is permitted to be a complex term; there is nothing special about the universal quantifier here, which is just given as an example of a binding construction). Other clauses of the definition of substitution are uncomplicated and are not presented in detail here. The general notation for simultaneous replacement of variables v_i with terms T_i in W is $W[T_1/v_1, \ldots, T_n/v_n]$. $(\forall U.V)[T/v]$ is defined as
$(\forall U[y_1/x_1, \ldots, y_n/x_n].V[y_1/x_1, \ldots, y_n/x_n][T/v])$ where the x_i's are all the bound variables *distinct from* v which occur free in U (similarly if there are multiple T_i's and v_i's the x_i's are the bound variables free in U which do not occur among the v_i's) and the y_i's are new bound variables (not found anywhere in the context). Notice that the notion of free occurrence of a bound variable is only important here for occurrences of variables in binders, as other problems with variable capture are avoided by aggressive renaming

of binding variables before substitutions are carried out.

The odd thing here is that in a binder $(\forall x.T)$ *all* typographical occurrences of the variable x in T are bound by this binder, which will cause certainly oddly written quantifier or other variable binding expressions to have different meanings than expected. There is a reason for this: look at the example $(\forall x.|\{y + x \mid y \in A\}| = |A|)$. Here we want the variable x to be treated as a constant in the subexpression $\{y + x \mid y \in A\}$ (we want it to stand for all sums of elements of A and x, not all sums of two elements of A). The convention we have adopted here seems to be the simplest one that allows us to use the full expressive power of binding with complex terms as binders.

7 More Sequent Rules

For conventions on how rules are to be read in general, see the section on sequent rules for connectives above. It is also important to note that when rules are applied to equations, definitional expansions are carried out on both sides of the equation, and when rules are applied to membership statements, definitional expansions are carried out on the right side, and any further opportunities to apply rules after definitional expansion are taken immediately (including further definitional expansions!).

In the rules which follow, if ϕ is a formula, $\phi[t/x]$ is taken to represent the result of substituting the term t for the variable x in ϕ.

7.1 Rules for Quantifiers

7.1.1 Left Rules

$$\frac{\Gamma, \phi[t/x], (\forall x.\phi) \vdash \Delta}{\Gamma, (\forall x.\phi) \vdash \Delta}$$

where t is any term

$$\frac{\Gamma, \phi[a/x] \vdash \Delta}{\Gamma, (\exists x.\phi) \vdash \Delta}$$

where a is a variable not appearing in the conclusion

7.1.2 Right Rules

$$\frac{\Gamma \vdash \phi[a/x], \Delta}{\Gamma \vdash (\forall x.\phi), \Delta}$$

where a is a variable not appearing in the conclusion

$$\frac{\Gamma \vdash \phi[t/x], (\exists x.\phi), \Delta}{\Gamma \vdash (\exists x.\phi), \Delta}$$

where t is any term

7.1.3 Comments on Quantifier Rules

In the quantifier rules, we have retained the quantified sentence from the conclusion in the premise. This is so that we can avoid formalizing notions of copying and reordering formulas in sequents: a quantified formula may be reused several times in a proof, and if it were erased by the application of the rule we would need to copy it explicitly. Another advantage is that it preserves precise equivalence of the conclusion with the conjunction of all the premises, which is a feature of all the sequent rules of Marcel.

The rules requiring input of a new variable a supply a computer-generated variable. In the original version of this prover, the rules involving a new term t were implemented by separate commands with the term t as a parameter. In the current version, the computer supplies a new "unknown variable" which can subsequently be replaced by a term: the advantage is that the same command can then handle all basic sequent rules.

7.2 Rules for Membership

7.2.1 Left Rule

$$\frac{\Gamma, \phi[t/x] \vdash \Delta}{\Gamma, t \in \{x \mid \phi\} \vdash \Delta}$$

when ϕ is stratified

7.2.2 Right Rule

$$\frac{\Gamma \vdash \phi[t/x], \Delta}{\Gamma \vdash t \in \{x \mid \phi\}, \Delta}$$

when ϕ is stratified

7.3 Rules for Equality

7.3.1 Left Rule

$$\frac{\Gamma, \phi[u/x][t/y], t = u \vdash \psi[u/x][t/y], \Delta}{\Gamma, \phi[t/x][u/y], t = u \vdash \psi[t/x][u/y], \Delta}$$

7.3.2 Right Rule

$$\Gamma \vdash t = t, \Delta$$

This is an axiom: it requires no premises

$$\frac{\Gamma \vdash (\exists x.x \in t), t = u, \Delta \qquad \Gamma \vdash (Ax.x \in t \leftrightarrow x \in u), t = u, \Delta}{\Gamma \vdash t = u, \Delta}$$

7.3.3 Comments on Equality Rules

In the earliest versions of the prover, equality was handled by definitional expansion: $t = u$ was defined as $(\forall x.t \in x \leftrightarrow u \in x)$. This was used as the left rule; the right rule looked like the second right rule here but with $t = u$ replaced in the premises by $(\forall x.t \in x \leftrightarrow t \in y)$. This is sufficient to support rewriting (as in the left rule given) but it is somewhat awkward.

The second rule implements the weak extensionality of *NFU*. This is an extension of the logic of the prover not found in *SF*, and is the reason that we do not know whether this logic has cut elimination.

$t = u$ is retained in premises in all rules. In the left rule it is retained because the equation may be used to rewrite again. In the right rule it is retained because it may actually not be possible to prove $t = u$ by the weak extensionality rule.

The actual implementations of the equality rules do not necessarily exactly resemble what is given here, but the implementations are justified by these rules.

7.4 Global Substitution, manual and automatic

The left rule for the universal quantifier and the right rule for the existential quantifier require input of a specific term t. What the prover actually supplies is an unknown variable `Un` with a fresh index (belonging to the same series of indices as those on the free variables `an` introduced by the left existential and right universal rules).

The `SetUnknown` command allows the user to set the unknown variable to a particular term. There is a restriction on what terms can replace an unknown variable: the term must have been definable at the time that the unknown variable `Un` was generated, so it cannot contain any free or unknown variables with index higher than `n`.

The prover also automatically makes substitutions for `Un`'s under certain circumstances. Under most circumstances in which two terms are to be matched (as in checking whether a sequent is an axiom or whether a theorem or a rewrite rule applies) if the prover can make the match work by assigning specific terms to the `Un`'s (with the same restrictions indicated above) it will do so.

An advantage of this approach are that it enables one to delay the choice of a witness: the later progress of the proof may make it more evident what witness will work.

8 Proving and Using Theorems

When a sequent has been proved, one can view all sequents in the proof using the `Showall` command. Each sequent has a program-generated serial number. Each sequent in the proof is valid: the sequent with serial number n can be recorded as a theorem using the command `NameSequent (Prove) n <name>`.

Whenever a match can be established between a theorem and a subsequent of the current sequent, this means that the current sequent is valid. The command `UseThm <name> L1 L2`, where `L1` and `L2` are lists of numbers, invites the prover to test that the sequent made of the left formulas of the current sequent with indices listed in `L1` and the right formulas of the current sequent with indices listed in `L2`: if a match is found (which may be forced by setting unknown variables to specific terms) the current sequent is proved (and a reference to that theorem is placed in the proof tree).

9 Cut and "Theorem Cut"

The Cut Rule

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

is clearly valid. It is different from the other rules: the other rules involve simplification of the conclusion in some sense, while this one involves introduction of a completely new formula A.

Pragmatically, the Cut Rule is indispensible. It represents the process of introducing a lemma in order to prove a theorem. Theoretically, it is interesting that the Cut Rule is redundant in something very like our full logic (we do not know whether this remains true when the weak extensionality axiom is implemented).

A powerful extension of Cut is implemented by the prover's `ThmCut` command. This command takes the name of a theorem as a command and generates a number of sequents, one a copy of the theorem with all free variables replaced by unknown variables (so they can later be instantiated) which is of course valid and is immediately proved, and the others, one for each left formula of the modified theorem with that formula added to the current sequent on the right, and one for each right formula of the modified theorem with that formula added to the current sequent on the left.

10 The Theorem List and Reading Saved Proofs

The prover allows one to view proofs: it converts the internal representation of proof trees into a theoretically human readable (though intensely boring) format. The `Showall` command will show the proof of a theorem just proved (or indeed an incomplete proof in progress).

Where a proof reference is involved, the prover in its default state will present the proof of the referenced theorem (proofs are stored on the theorem list), so that the `Showall` command will show the complete proof from first principles. The `nolemmas` command will turn off the display of proofs of theorems. An option to refrain from storing proofs of theorems might be appropriate if membory were very limited, but is not currently provided.

11 Alternative Logics

Some alternative logics are provided and more are projected.

The `Constructive` command implements constructive (intuitionistic) logic. We found it very interesting how easy it was to implement this. The implementation amounts to ensuring that whenever a rule is applied, all formulas

on the right after the first one are eliminated (if a sequent generates more than one alternative conclusion, all are preserved so that the user can move the desired one to the first position before applying the next rule, which will eliminate the second and later alternatives). In addition, implications and negations are preserved on the left when the left rules for these operators are applied (so that they can be expanded multiple times). This precisely implements a constructive logic.

A logic equivalent to the usual logic of the prover is turned on by the `OneConclusion` command (and turned off by the `ManyConclusions` command. This option eliminates the alternative conclusions in a nondestructive way: each alternative conclusion after the first is negated and moved to the front, after each rule is applied. The difference between this and the usual handling of sequents by the prover is purely one of style, but may be appreciated by some users.

A planned alternative implementation is one which uses something more like standard set theory (Zermelo or Zermelo-Frankel set theory).

Other set theories, such as positive set theory or the double extension set theory of Kisielewicz could be implemented.

12 Future Developments

13 Examples of Use of Marcel

We give some sample proofs. The `Start` command takes as its parameter notation for a theorem to be proved.

```
- Start "P1&P2->P3==P1->P2->P3";

Line number 1:



|-

1:  P1 & P2 -> P3 == P1
      -> P2 -> P3
```

```
> val it = () : unit
```

What is displayed at each step of a Marcel proof is a sequent. The display is vertical rather than horizontal: the formulas "left" of the turnstile are listed above and those "right" of it below.

```
r();


Line number 2:

1:  P1 & P2 -> P3

|-

1:  P1 -> P2 -> P3
```

The `r` command applies an appropriate rule to the first formula in the list below the turnstile. The sequent displayed here is not the only sequent generated: another sequent is also generated, and when the proof of this sequent is complete Marcel will automatically present the other one.

```
- r();

Line number 4:

1:  P1
2:  P1 & P2 -> P3

|-

1:  P2 -> P3
```

```
- r();


Line number 5:

1:  P2
2:  P1
3:  P1 & P2 -> P3

|-

1:  P3
```

The right rule is applied twice (in both cases the quite natural rule for proving an implication). No complex proposition remains in first position in either list. So we change the order.

```
- gl 3;

Line number 5:

1:  P1 & P2 -> P3
2:  P2
3:  P1

|-

1:  P3


Line number 6:

1:  P2
2:  P1
```

```
|-

1:  P1 & P2

2:  P3
```

The command `gl 3` brings the third proposition in the left list to the front. The `l` command applies an appropriate sequent rule determined by the first formula in the left list. In this case as in the case of the first rule application, two sequents are generated and the other is reserved by the prover to be displayed for proof later.

```
- r();



Line number 8:

1:  P2
2:  P1

|-

1:  P1
2:  P3
```

The rule application here creates two sequents. We will very shortly see the other one.

```
- gl 2;

Line number 8:

1:  P1
2:  P2
```

```
|-

1:  P1
2:  P3

- Done();

Line number 9:

1:  P2
2:  P1

|-

1:  P2
2:  P3
```

When the formula `P1` is brought to the head of the left list by the `gl 2` command, the sequent becomes trivially valid by the Axiom rule, since the first formula on each list is now the same as the first formula on the other list. The `Done` command causes Marcel to acknowledge this. The formula which now appears is the other sequent produced by the recent application of the conjunction rule, to which `Done` can immediately be applied:

```
- Done();

Line number 7:

1:  P3
2:  P2
3:  P1

|-

1:  P3
```

The sequent which appears now is deferred from the application of the left rule of implication earlier. We can apply `Done`.

```
- Done();

Line number 3:

1:  P1 -> P2 -> P3

|-

1:  P1 & P2 -> P3
```

The converse implication which we still need to prove to complete the proof of the biconditional theorem is now displayed.

```
- r(); l();

Line number 10:

1:  P1 & P2
2:  P1 -> P2 -> P3

|-

1:  P3

Line number 11:

1:  P1
2:  P2
3:  P1 -> P2 -> P3

|-

1:  P3
```

If it is clear what to do we can as here issue several commands at once. Again, we need to bring line 3 on the left to the front of the list.

```
- gl 3; l();

Line number 11:

1:  P1 -> P2 -> P3
2:  P1
3:  P2

|-

1:  P3

Line number 12:

1:  P1
2:  P2

|-

1:  P1
2:  P3
```

We apply `Done` and are served another sequent generated by the above application of the left rule for implication.

```
- Done();

Line number 13:

1:  P2 -> P3
2:  P1
3:  P2

|-
```

```
1:  P3
```

We apply `l` to the only complex proposition we see.

```
- l();

Line number 14:

1:  P1
2:  P2

|-

1:  P2
2:  P3
```

The rest is proof of simple sequents using `Done`.

```
- gl 2; Done();

Line number 14:

1:  P2
2:  P1

|-

1:  P2
2:  P3

Line number 15:

1:  P3
```

```
2:  P1
3:  P2

|-

1:  P3

Done();

Q. E. D.
```

The final message `Q. E. D.` tells us that Marcel finds no unproved sequent goals in the current proof tree, so the proof of the original theorem is complete.

Now we give an example from the logic of quantifiers.

```
- Start "(Ex1.(Ax2.P1(x1)->P1(x2)))";

Line number 1:

|-

1:  (Ex1.(Ax2.P1(x1) -> P1(x2)))

- r();

Line number 2:

|-

1:  (Ax3.P1(U1) -> P1(x3))
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
```

The right rule for the existential quantifier allows the substitution of any term for the bound variable in the quantified sentence. Marcel introduces a special "unknown variable" `U1` here: the user may immediately or later

select a value for an unknown variable (which will replace all occurrences of that variable throughout the proof, not just in the sequent currently being proved). Oddly, `U1` is never replaced by anything in this particular proof.

```
- r();

Line number 3:

|-

1:  P1(U1) -> P1(a2)
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
```

The right rule for the universal quantifier simply replaces the bound variable by a fresh free variable.

```
- r();

Line number 4:

1:  P1(U1)

|-

1:  P1(a2)
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))

- su 1 "a2";

Circularity error

<hit enter>
```

Here an attempt is made to replace the unknown variable `U1` with the fresh variable `a2`. This is not logically valid (thus the error message). `U1` can only be replaced with a term which is meaningful in the context at which the unknown variable was originally introduced: no free variable or unknown variable with a higher index than $n - 1$ can appear in a term which is to replace `Un`.

```
- 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)
```

The command `Gr n` is a macro, equivalent to `gr n; r();` Note that we use the existential conclusion again, giving an example of re-use of existential conclusions justifying the fact that they are retained when the appropriate rule is applied.

```
- r();

Line number 6:

1:  P1(U1)

|-

1:  P1(U3) -> P1(a4)
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
3:  P1(a2)
```

```
- r();

Line number 7:

1:  P1(U3)
2:  P1(U1)

|-

1:  P1(a4)
2:  (Ex1.(Ax2.P1(x1) -> P1(x2)))
3:  P1(a2)

- su 3 "a2";

Line number 7:

1:  P1(a2)
2:  P1(U1)

|-

1:  P1(a2)
2:  P1(a4)
3:  (Ex3.(Ax4.P1(x3) -> P1(x4)))

- Done();

Q. E. D.
```

It is possible to replace `U3` with `a2` as it was not possible to replace `U1`. This makes it possible to convert the sequent to a form which is trivially valid.

14 Command Reference

14.1 Commands Listed Alphabetically

AutoPrune: No arguments. Remove redundant propositions from the current proof (propositions not used in the proofs of completed parts).

Axiom: A string argument followed by two string list arguments. The first argument is the name of the axiom to be recorded. The sequent to be declared an axiom has as its left side the list of terms represented by the strings in the first list and as its right side the list of terms represented by the strings on the right side.

b: No arguments. Abbreviation for Undo.

bookmark: One string argument. Assigns the string identifier which is the argument as a name for the sequent being viewed in the current proof. Bookmarks are not yet remembered by b/undo, saved by savetheory, or recorded in proof logs. Eventually they will be so remembered, saved and recorded.

Constructive: No arguments. Puts the prover in constructive logic mode (by restricting the use of more than one conclusion).

Crwl (crwl): An integer list argument. Rewrite the second proposition on the left side of the current sequent with the first proposition on the left side of the current sequent (which needs to be an equation). If the argument is the empty list, all occurrences of the right side of the rewriting equation are to be replaced with occurrences of the left side. If the argument is a nonempty list, the occurrences of the right side of the equation indexed by the terms of that list are written.

Crwr (crwr): An integer list argument. Rewrite the first proposition on the right side of the current sequent with the first proposition on the left side of the current sequent (which needs to be an equation). If the argument is the empty list, all occurrences of the right side of the rewriting equation are to be replaced with occurrences of the left side. If the argument is a nonempty list, the occurrences of the right side of the equation indexed by the terms of that list are written.

Cut (Cutr): A single string argument. Applies the cut rule to the current sequent. The argument is the cut proposition. The sequent with the argument on the right is to be proved first.

Cut2 (Cutl): A single string argument. Applies the cut rule to the current sequent. The argument is the cut proposition. The sequent with the argument on the left is to be proved first.

CutLemma: One string argument. Applies the cut rule to the current sequent with modifications. The argument is a single proposition. The two sequents produced are (presented first) a sequent with just the argument appearing on the right and (presented second) a sequent looking like the original sequent but with the argument proposition added on the right. The intention is to support proving a lemma while embedded in a larger proof; if the lemma is to be put on the theorems list, we suggest immediately using the `bookmark` command.

Declarefunction (DeclareFunction): a string argument followed by an integer list argument. Declare a primitive function or constant. The string is the name of the new function or constant. The integer list consists of the relative type of the output of the function followed by the relative types of its inputs (and so implicitly gives the number of arguments, which is one less than the length of the list).

Declarepredicate (DeclarePredicate): a string argument followed by an integer list argument. Declare a primitive predicate. The string is the name of the new predicate and the list gives the relative types of its arguments for stratification (implicitly giving the number of arguments).

Definefunction (DefineFunction): an integer argument followed by three string arguments. Define a function or constant. The first argument is the arity of the new operation (0 for a constant). The second argument is the name of the new operation. The third argument is the left side of the definition (the name followed by a list of distinct bound variables; an infix expression will work if there are two arguments). The fourth argument is the right side of the definition. Stratification information is generated automatically (or failure is reported, in which case the operation is still defined but its use is restricted).

Definepredicate (DefinePredicate): an integer argument followed by three string arguments. Define a predicate or sentence. The first argument is the arity of the new predicate (0 for a sentence). The second argument is the name of the new predicate. The third argument is the left side of the definition (the name followed by a list of distinct bound variables; an infix expression will work if there are two arguments). The fourth argument is the right side of the definition. Stratification information is generated automatically (or failure is reported, in which case the predicate is still defined but its use is restricted).

DefSent: Two string arguments: the first string is the name of the new sentence and the second is its intended meaning.

Done (d): No arguments. If the first formulas on the left and right sides of the current sequent are equivalent up to renaming of bound variables, prove the current sequent (and automatically pass to another branch of the proof if it is not complete). In `marcel387`, experimental version of March 3, 2009, also works for reflexivity of equality: this upgrade should be expected for the official version as well.

GetLeft (gl): One integer argument. Rotate the proposition on the left side of the current sequent indexed by the argument to the first position.

GetLeft2 (gl2): One integer argument. Rotate the proposition on the left side of the current sequent indexed by the argument to the second position, leaving the first proposition in place.

GetProof: One string argument. Set the current proof to the recorded proof of the theorem named by the argument. The application is to extract sequents as new theorems (since the proof of a theorem is complete we can't change it).

GetRight (gr): One integer argument. Rotate the proposition on the right side of the current sequent indexed by the argument to the first position.

GetRight2 (gr2): One integer argument. Rotate the proposition on the right side of the current sequent indexed by the argument to the second position, leaving the first proposition in place.

Gl: One numerical argument. Equivalent to `GetLeft n` followed by `l()`.

Gr: One numerical argument. Equivalent to `GetRight n` followed by `l()`.

L (l): No arguments. Apply the left rule appropriate to the form of the leading proposition on the left to the current sequent.

loadtheory: One string argument `name`. Load the theory saved in `name.thy1` and `name.thy2`.

LogProof: One string argument. Record the proof of the theorem named by the argument to the current log file.

LogTheProof: No arguments. Record the current partial proof in the log file.

Look: No arguments. Show the current sequent.

ManyConclusions: No arguments. Turns off the mode introduced by `OneConclusion`.

NameSequent (namesequent,Prove,prove): Integer argument followed by string argument. The sequent in the current proof with the given index is recorded as a theorem with the given string as its name.

NextGoal (ng, nextgoal): No arguments. Go to the next goal in the proof tree in a rather complex order guaranteed to traverse the entire tree but in which goals are sometimes repeated before traversal is complete.

nolemmas: No arguments. Turn off the display of proofs of theorems used in displayed proofs.

NoRemember (noremember): No arguments. Turn off the history feature.

OneConclusion: No arguments. Turns on a mode where the logic remains classical but multiple conclusions are not used (they are negated and moved to the left).

Prove (prove): This is an alias for `NameSequent`.

ProveMarked: Two string arguments. The first argument is a bookmark (see `bookmark`). The sequent referenced by the bookmark argument is recorded as a theorem if it has in fact been proved, with name given by the second argument.

PruneLeft (pl): One integer argument. Rotate the left proposition indexed by the argument to the front then remove it.

PruneRight (pr): One integer argument. Rotate the right proposition indexed by the argument to the front then remove it.

Remember (remember): No arguments. Turn on the history feature.

R (r): No arguments. Apply the right rule appropriate to the form of the leading proposition on the left to the current sequent.

RewriteFree (rf): The leading proposition on the left should be an equation with a free variable on one or both sides. If the two sides of the equation are the same free variable, drop the proposition. If they are both free variables, rewrite the higher indexed one to the lower indexed one through the entire sequent, then drop the proposition. If only one side is a free variable, and the other side does not contain that variable, rewrite all occurrences of the free variable to the other side of the equation, then drop the equation. If the free variable is contained in the other side of the equation, issue an error message.

runscript: One string argument. Adds the extension `.mlg` to the string argument and runs this script file. Will look in a working directory set by `SetDir`.

runtext: One string argument. Adds the extension `.txt` to the string argument and runs this script file. Will look in a working directory set by `SetDir`.

Rwl (rwl): One integer list argument. As `Crwl`, but rewrite the left side of the equation to the right side.

Rwr (rwr): One integer list argument. As `Crwr`, but rewrite the left side of the equation to the right side.

savetheory: One string argument `name`. Store the current theory to files `name.thy1` and `name.thy2`.

SetDir: One string argument. Sets a working directory (in which `runscript` and `runtext` will look for files).

SetMargin: One integer argument. Set the margin at which the display breaks lines.

setprecleftabove (Spla): Two string arguments. Set the precedence of the infix named by the first string to be just above the precedence of the infix named by the second string and below all higher precedences, and set it to group to the left.

setprecleftbelow (Splb): Two string arguments. Set the precedence of the infix named by the first string to be just below the precedence of the infix named by the second string and above all lower precedences, and set it to group to the left.

setprecleftmax (Splx): One string argument. Set the precedence of the infix named by the first string to be maximal, and set it to group to the left.

setprecleftmin (Spln): One string argument. Set the precedence of the infix named by the first string to be minimal and set it to group to the left.

setprecrightabove (Spra): Two string arguments. Set the precedence of the infix named by the first string to be just above the precedence of the infix named by the second string and below all higher precedences, and set it to group to the right.

setprecrightbelow (Sprb): Two string arguments. Set the precedence of the infix named by the first string to be just below the precedence of the infix named by the second string and above all lower precedences, and set it to group to the right.

setprecrightmax (Sprx): One string argument. Set the precedence of the infix named by the first string to be maximal, and set it to group to the left.

setprecrightmin (Sprn): One string argument. Set the precedence of the infix named by the first string to be minimal and set it to group to the left.

setprecsame (Sps): Two string arguments. Set the precedence of the infix named by the first string to be the same as the precedence of the infix named by the second string.

SetPredVar (sp): Integer argument followed by string argument. Where `n` is the integer argument and `S` is the string argument, globally replace propositions of the form `Pn(t1,...,tn)` with propositions `[t1,...,tn] E S`. In other words, globally interpret the predicate variable `Pn` as the relation with extension `S`.

SetUnknown (su): Integer argument followed by string argument. Where `n` is the numeral argument, globally replace `Un` with the term represented by the string argument as long as it refers to no `Um` or `am` with $m \geq n$. This affects the entire current proof, not just the sequent you are looking at.

Showall (showall): No arguments. Show the current partial proof of the current sequent. Hit return between sequents.

Showalltheorems: No arguments. Show all theorems; hit return between theorems.

Showaxioms: No arguments. Show all axioms; hit return between theorems.

Showcurrent: No arguments. Show the list of theorems in use in the current proof; hit return between theorems.

showlemmas: No arguments. Turn on the feature which displays proofs of theorems used in the current proof as part of the whole proof (on by default).

ShowMarked: One string argument. The argument is a bookmark (see `bookmark`). The command allows one to view the part of the proof at and above the bookmarked sequent.

Showpropdefs: No arguments. Show all predicate and sentence definitions.

Showtermdefs: No arguments. Show all function and constant definitions.

Start (s, start): One string argument. Reset the current sequent to the sequent with nothing on the left and the proposition represented by the string on the right.

startdemo: No arguments. Start demo mode (displays commands in a file and their effects with pauses).

startlogging: One string argument `name`. Start logging prover commands and error messages to the file `name.mlg`.

StartSequent (ss): Two string list arguments. Set the current sequent to the lists of propositions represented by the lists of string arguments: the first argument is the left side of the sequent and the right argument is the right side of the sequent.

stopdemo: No arguments. Stop demo mode.

stoplogging: Stop recording to the log file started by `startlogging`.

SwapGoals (swapgoals, sg): No arguments. Interchange the order of the two or more goals just created by a rule application. This allows proof using the conclusion of an implication before the proof of its hypothesis, for example. If there are more than two goals, they are rotated. If the command is used not in the immediate aftermath of a rule application, it will cycle through a few goals in a local area of the proof.

ThmCut: One argument, the name of a theorem. Creates a new sequent for each proposition in the theorem, replacing all free variables in the theorem with unknowns. Each hypothesis needs to be proved and each conclusion can then be used. The hypotheses are proved first. SwapGoals can be used to rotate through the list of goals created.

ThmCut2: One argument, the name of a theorem. Creates a new sequent for each proposition in the theorem, replacing all free variables in the theorem with unknowns. Each hypothesis needs to be proved and each conclusion can then be used. The conclusions are used first. SwapGoals can be used to rotate through the list of goals created.

ThmDisplay (td): One string argument. Display the theorem named by the argument.

Triv: Two integer arguments. `Triv m n` has the effect of `GetLeft m; GetRight n; Done()`.

Undo (undo, b): No arguments. Back up one step in the current proof. The steps are not always whole user commands: for example it steps through individual substitutions for `Ui`'s when these have been made automatically.

UseThm: A string argument followed by two integer list arguments. The string argument names a theorem; the integer lists indicate the propositions on the left and right which match it: if the match is verified, the current sequent is proved.

Witness (w): Integer argument followed by string argument. As `SetUnknown` except that the index of the unknown variable replaced is the global free/unknown variable counter, plus one, minus the integer argument. `r()` or `l()` followed by `w 1 term` has roughly the effect of `w term` in `marcelsequent`.

14.2 Commands in Categories by Function

14.2.1 Prover Settings

These are settings of the prover which change the logic. Currently these are not handled carefully by the scripting functions.

Constructive: No arguments. Puts the prover in constructive logic mode (by restricting the use of more than one conclusion).

ManyConclusions: No arguments. Turns off the mode introduced by `OneConclusion`.

OneConclusion: No arguments. Turns on a mode where the logic remains classical but multiple conclusions are not used (they are negated and moved to the left).

14.2.2 Declaring Primitives and Syntax

These commands declare primitive identifiers (without definitions) and set operator precedences.

Declarefunction (DeclareFunction): a string argument followed by an integer list argument. Declare a primitive function or constant. The string is the name of the new function or constant. The integer list consists of the relative type of the output of the function followed by the relative types of its inputs (and so implicitly gives the number of arguments, which is one less than the length of the list).

Declarepredicate (DeclarePredicate): a string argument followed by an integer list argument. Declare a primitive predicate. The string is the name of the new predicate and the list gives the relative types of its arguments for stratification (implicitly giving the number of arguments).

setprecleftabove (Spla): Two string arguments. Set the precedence of the infix named by the first string to be just above the precedence of the infix named by the second string and below all higher precedences, and set it to group to the left.

setprecleftbelow (Splb): Two string arguments. Set the precedence of the infix named by the first string to be just below the precedence of the infix named by the second string and above all lower precedences, and set it to group to the left.

setprecleftmax (Splx): One string argument. Set the precedence of the infix named by the first string to be maximal, and set it to group to the left.

setprecleftmin (Spln): One string argument. Set the precedence of the infix named by the first string to be minimal and set it to group to the left.

setprecrightabove (Spra): Two string arguments. Set the precedence of the infix named by the first string to be just above the precedence of the infix named by the second string and below all higher precedences, and set it to group to the right.

setprecrightbelow (Sprb): Two string arguments. Set the precedence of the infix named by the first string to be just below the precedence of the infix named by the second string and above all lower precedences, and set it to group to the right.

setprecrightmax (Sprx): One string argument. Set the precedence of the infix named by the first string to be maximal, and set it to group to the left.

setprecrightmin (Sprn): One string argument. Set the precedence of the infix named by the first string to be minimal and set it to group to the left.

setprecsame (Sps): Two string arguments. Set the precedence of the infix named by the first string to be the same as the precedence of the infix named by the second string.

14.2.3 Axioms, Definitions, and Theorems

These commands define things, whether predicates, functions or theorems. `bookmark` is a bit of a stretch but it was unclear where else to put it.

bookmark: One string argument. Assigns the string identifier which is the argument as a name for the sequent being viewed in the current proof. Bookmarks are not yet remembered by b/undo, saved by savetheory, or recorded in proof logs. Eventually they will be so remembered, saved and recorded.

Axiom: A string argument followed by two string list arguments. The first argument is the name of the axiom to be recorded. The sequent to be declared an axiom has as its left side the list of terms represented by the strings in the first list and as its right side the list of terms represented by the strings on the right side.

Definefunction (DefineFunction): an integer argument followed by three string arguments. Define a function or constant. The first argument is the arity of the new operation (0 for a constant). The second argument is the name of the new operation. The third argument is the left side of the definition (the name followed by a list of distinct bound variables; an infix expression will work if there are two arguments). The fourth argument is the right side of the definition. Stratification information is generated automatically (or failure is reported, in which case the operation is still defined but its use is restricted).

Definepredicate (DefinePredicate): an integer argument followed by three string arguments. Define a predicate or sentence. The first argument is the arity of the new predicate (0 for a sentence). The second argument is the name of the new predicate. The third argument is the left side of the definition (the name followed by a list of distinct bound variables; an infix expression will work if there are two arguments). The fourth argument is the right side of the definition. Stratification information is generated automatically (or failure is reported, in which case the predicate is still defined but its use is restricted).

DefSent: Two string arguments: the first string is the name of the new sentence and the second is its intended meaning.

NameSequent (namesequent,Prove,prove): Integer argument followed by string argument. The sequent in the current proof with the given index is recorded as a theorem with the given string as its name.

Prove (prove): This is an alias for `NameSequent`.

ProveMarked: Two string arguments. The first argument is a bookmark (see `bookmark`). The sequent referenced by the bookmark argument is recorded as a theorem if it has in fact been proved, with name given by the second argument.

14.2.4 Proof Manipulations

These commands perform manipulations of the proof structure rather than of individual sequents.

AutoPrune: No arguments. Remove redundant propositions from the current proof (propositions not used in the proofs of completed parts).

GetProof: One string argument. Set the current proof to the recorded proof of the theorem named by the argument. The application is to extract sequents as new theorems (since the proof of a theorem is complete we can't change it).

NextGoal (ng, nextgoal): No arguments. Go to the next goal in the proof tree in a rather complex order guaranteed to traverse the entire tree but in which goals are sometimes repeated before traversal is complete.

SetPredVar (sp): Integer argument followed by string argument. Where `n` is the integer argument and `S` is the string argument, globally replace propositions of the form `Pn(t1,...,tn)` with propositions `[t1,...,tn] E S`. In other words, globally interpret the predicate variable `Pn` as the relation with extension `S`.

SetUnknown (su): Integer argument followed by string argument. Where `n` is the numeral argument, globally replace `Un` with the term represented by the string argument as long as it refers to no `Um` or `am` with $m \geq n$. This affects the entire current proof, not just the sequent you are looking at.

SwapGoals (swapgoals, sg): No arguments. Interchange the order of the two or more goals just created by a rule application. This allows proof using the conclusion of an implication before the proof of its hypothesis, for example. If there are more than two goals, they are rotated. If the command is used not in the immediate aftermath of a rule application, it will cycle through a few goals in a local area of the proof.

Witness (w): Integer argument followed by string argument. As `SetUnknown` except that the index of the unknown variable replaced is the global free/unknown variable counter, plus one, minus the integer argument. `r()` or `l()` followed by `w 1 term` has roughly the effect of `w term` in `marcelsequent`.

14.2.5 Starting and Finishing Sequents

These commands initiate or terminate the handling of a sequent.

Done (d): No arguments. If the first formulas on the left and right sides of the current sequent are equivalent up to renaming of bound variables, prove the current sequent (and automatically pass to another branch of the proof if it is not complete). In `marcel387`, experimental version of March 3, 2009, also applies to reflexivity of equality: expect this in the official version as well.

Start (s, start): One string argument. Reset the current sequent to the sequent with nothing on the left and the proposition represented by the string on the right.

StartSequent (ss): Two string list arguments. Set the current sequent to the lists of propositions represented by the lists of string arguments: the first argument is the left side of the sequent and the right argument is the right side of the sequent.

UseThm: A string argument followed by two integer list arguments. The string argument names a theorem; the integer lists indicate the propositions on the left and right which match it: if the match is verified, the current sequent is proved.

14.2.6 Sequent Manipulations

These commands handle sequents in various non-initial and non-terminal ways, including reducing the sequent to one or more simpler sequents.

Crwl (crwl): An integer list argument. Rewrite the second proposition on the left side of the current sequent with the first proposition on the left side of the current sequent (which needs to be an equation). If the argument is the empty list, all occurrences of the right side of the rewriting equation are to be replaced with occurrences of the left side. If the argument is a nonempty list, the occurrences of the right side of the equation indexed by the terms of that list are written.

Crwr (crwr): An integer list argument. Rewrite the first proposition on the right side of the current sequent with the first proposition on the left side of the current sequent (which needs to be an equation). If the argument is the empty list, all occurrences of the right side of the rewriting equation are to be replaced with occurrences of the left side. If the argument is a nonempty list, the occurrences of the right side of the equation indexed by the terms of that list are written.

Cut (Cutr): A single string argument. Applies the cut rule to the current sequent. The argument is the cut proposition. The sequent with the argument on the right is to be proved first.

Cut2 (Cutl): A single string argument. Applies the cut rule to the current sequent. The argument is the cut proposition. The sequent with the argument on the left is to be proved first.

CutLemma: One string argument. Applies the cut rule to the current sequent with modifications. The argument is a single proposition. The two sequents produced are (presented first) a sequent with just the argument appearing on the right and (presented second) a sequent looking like the original sequent but with the argument proposition added on the right. The intention is to support proving a lemma while embedded in a larger proof; if the lemma is to be put on the theorems list, we suggest immediately using the `bookmark` command.

GetLeft (gl): One integer argument. Rotate the proposition on the left side of the current sequent indexed by the argument to the first position.

GetLeft2 (gl2): One integer argument. Rotate the proposition on the left side of the current sequent indexed by the argument to the second position, leaving the first proposition in place.

GetRight (gr): One integer argument. Rotate the proposition on the right side of the current sequent indexed by the argument to the first position.

GetRight2 (gr2): One integer argument. Rotate the proposition on the right side of the current sequent indexed by the argument to the second position, leaving the first proposition in place.

Gl: One numerical argument. Equivalent to `GetLeft n` followed by `l()`.

Gr: One numerical argument. Equivalent to `GetRight n` followed by `l()`.

L (l): No arguments. Apply the left rule appropriate to the form of the leading proposition on the left to the current sequent.

PruneLeft (pl): One integer argument. Rotate the left proposition indexed by the argument to the front then remove it.

PruneRight (pr): One integer argument. Rotate the right proposition indexed by the argument to the front then remove it.

R (r): No arguments. Apply the right rule appropriate to the form of the leading proposition on the left to the current sequent.

RewriteFree (rf): The leading proposition on the left should be an equation with a free variable on one or both sides. If the two sides of the

equation are the same free variable, drop the proposition. If they are both free variables, rewrite the higher indexed one to the lower indexed one through the entire sequent, then drop the proposition. If only one side is a free variable, and the other side does not contain that variable, rewrite all occurrences of the free variable to the other side of the equation, then drop the equation. If the free variable is contained in the other side of the equation, issue an error message.

Rwl (rwl): One integer list argument. As `Crwl`, but rewrite the left side of the equation to the right side.

Rwr (rwr): One integer list argument. As `Crwr`, but rewrite the left side of the equation to the right side.

ThmCut: One argument, the name of a theorem. Creates a new sequent for each proposition in the theorem, replacing all free variables in the theorem with unknowns. Each hypothesis needs to be proved and each conclusion can then be used. The hypotheses are proved first. SwapGoals can be used to rotate through the list of goals created.

ThmCut2: One argument, the name of a theorem. Creates a new sequent for each proposition in the theorem, replacing all free variables in the theorem with unknowns. Each hypothesis needs to be proved and each conclusion can then be used. The conclusions are used first. `SwapGoals` can be used to rotate through the list of goals created.

Triv: Two integer arguments. `Triv m n` has the effect of `GetLeft m; GetRight n; Done()`.

14.2.7 Lists and Displays

These commands manipulate prover displays. A large subcategory is displays of theorem lists.

Look: No arguments. Show the current sequent.

nolemmas: No arguments. Turn off the display of proofs of theorems used in displayed proofs.

SetMargin: One integer argument. Set the margin at which the display breaks lines.

Showall (showall): No arguments. Show the current partial proof of the current sequent. Hit return between sequents.

Showalltheorems: No arguments. Show all theorems; hit return between theorems.

Showaxioms: No arguments. Show all axioms; hit return between theorems.

Showcurrent: No arguments. Show the list of theorems in use in the current proof; hit return between theorems.

showlemmas: No arguments. Turn on the feature which displays proofs of theorems used in the current proof as part of the whole proof (on by default).

ShowMarked: One string argument. The argument is a bookmark (see `bookmark`). The command allows one to view the part of the proof at and above the bookmarked sequent.

Showpropdefs: No arguments. Show all predicate and sentence definitions.

Showtermdefs: No arguments. Show all function and constant definitions.

startdemo: No arguments. Start demo mode (displays commands in a file and their effects with pauses).

stopdemo: No arguments. Stop demo mode.

ThmDisplay (td): One string argument. Display the theorem named by the argument.

14.2.8 History

Commands related to the undo feature. I should add a forward command which allows undo to be undone if no other intervening command has been performed.

b: No arguments. Abbreviation for Undo.

NoRemember (noremember): No arguments. Turn off the history feature.

Remember (remember): No arguments. Turn on the history feature.

Undo (undo, b): No arguments. Back up one step in the current proof. The steps are not always whole user commands: for example it steps through individual substitutions for `Ui`'s when these have been made automatically.

14.2.9 Loading and Saving

These commands handle loading and saving. There are commands here which manage scripting (generating scripts as you work and running them) and commands handling a theory saving system.

loadtheory: One string argument `name`. Load the theory saved in `name.thy1` and `name.thy2`.

LogProof: One string argument. Record the proof of the theorem named by the argument to the current log file.

LogTheProof: No arguments. Record the current partial proof in the log file.

runscript: One string argument. Adds the extension `.mlg` to the string argument and runs this script file. Will look in a working directory set by `SetDir`.

runtext: One string argument. Adds the extension `.txt` to the string argument and runs this script file. Will look in a working directory set by `SetDir`.

savetheory: One string argument `name`. Store the current theory to files `name.thy1` and `name.thy2`.

SetDir: One string argument. Sets a working directory (in which `runscript` and `runtext` will look for files).

startlogging: One string argument `name`. Start logging prover commands and error messages to the file `name.mlg`.

stoplogging: Stop recording to the log file started by `startlogging`.