Marcel Lab Manual and Technical Reference

M. Randall Holmes

February 2, 2018

Contents

1 Introduction

This document is intended as a basic introduction to the use of the Marcel theorem prover which I have been developing for the last few years, in the Python version which I developed for delivery through Sage (but which is also usable as a freestanding Python program, available on my web page), with a full technical description appended.

The Python version of the Marcel theorem prover is maintained at

```
http://math.boisestate.edu/~holmes/marcelstuff/pythonmarcel.py
```

The way I usually run it is either to open it in Sage or to edit the file in IDLE amd "run module". One can then type commands at the prompt. It is useful to edit your own files in IDLE with

```
from pythonmarcel import *
```

at the head; one can then edit and otherwise play with a separate file without risking damaging the prover source.

2 Prover notation

For purposes of the first labs, we summarize the logical notation of Marcel.

propositional variables: A propositional variable (corresponding to a capital letter used to represent an unknown statement) is a string of lower case letters followed by a question mark. `p?` is a typical propositional variable. `abc?` is also a propositional variable. `P?` is not a propositional variable.

propositional operations: A negation $\neg P$ translates to `~p?`. A conjunction $P \wedge Q$ translates to `p? & q?`. A disjunction $P \vee Q$ translates to `p? V q?` (note that a capital V denotes disjunction). An implication $P \rightarrow Q$ translates to `p? -> q?`. A biconditional $P \leftrightarrow Q$ translates to `p? == q?`

spacing: Where operators are adjacent to one another, one needs to insert a space. $P \wedge \neg Q$ translates to `p? & ~q?`, not `p? &~q?`.This is the only case where explicit spacing is actually needed: Marcel parses `p?&q?` or `p?& ~q?` perfectly happily.

parentheses and order of operations: You can use parentheses (or brackets or braces) to force desired grouping. Marcel understands that negation binds most tightly, followed by conjunction, followed by disjunction, followed by the biconditional. Marcel will supply spaces in its display only when it determines that they are necessary, so its display may have fewer parentheses than yours (and will only use parentheses for grouping of propositions in its display). Conjunction and disjunction group to the left (e.g., `p? & q? & r?` is read as `(p? & q?) & r?`); implication and biconditional group to the right.

Our advice to users is to use as much extra explicit grouping with parentheses, braces and/or brackets as you need until you are comfortable with how Marcel groups things.

object variables and constants: An object bound variable is either a single lower-case letter or a string of one or more lower-case letters followed by a numeral (a string of digits). The integer value of the numeral component determines the type of the variable. A single letter variable such as `x` is of type 0 and is in fact synonymous for the prover with `x0`.

The types of bound variables are not important in the initial labs on propositional and quantifier logic, and indeed the single letter variables will suffice for all our purposes in the first labs. The basic idea of the types is that type 0 variables stand for objects not understood as sets or functions, while type $n+1$ objects stand for sets of type n objects or functions taking type n objects to type n objects. Constant expressions or free variables representing objects do not however have type in our logic.

Object constants are strings of more than one lower case letter not followed by a numeral or strings of digits. String constants must be declared to be used. Numerals are understood by Marcel as predeclared constants (eventually we intend to install arithmetic knowledge into Marcel but for the moment it merely recognizes the constants).

Object free variables are of two kinds. They consist of a string of lower case letters followed by either `_` or `$` then by a string of digits (an index). The ones with underscores are "arbitrary objects"; the ones with dollars signs are "instantiables". `a_3` is a typical arbitrary object; `b$5` is a typical instantiable. Note again that these terms do not have type: their final digit has a different function.

operators: An operator is of one of two forms, either a string of capital letters or a string of special characters, either of which may optionally be followed by an underscore and a string of digits (an index). Operators must be declared. Only nonindexed operators can be declared (using various commands illustrated where needed); once a nonindexed operator is declared, indexed operators with the same initial component can be used freely and will have the same type as their nonindexed component (types of operators to be discussed later). Indexed operators are variable operators (theorems about them are general facts about all operators of that type). Operators may be syntactically unary or binary (this is part of their type information) and their operator precedence may be set by the user. Additional operators can be declared or defined needed. See the technical reference for full details.

Predeclared operators are $\sim$ (negation), `&` (conjunction), `V`, (disjunction) `->` (implication), `==` (biconditional), `A` (universal quantifier), `E` (existential quantifier), `=` (equality), ‘ (infix operator of function application), `IN` (membership) and `THE` (the definite description operator).

Full information about types and precedences of these operators will be given in the technical reference. The symbols `:` (set abstractor) and `:>` (function abstractor) behave syntactically like operators.

quantifier notation: The quantifiers $\forall$ and $\exists$ are denoted by `A` and `E`, respectively. ($\forall x.x = x$) translates to `Ax : x = x`. The notation `x : x = x` is actually notation for the set $\{x \mid x = x\}$ and the quantified sentence is read as applying the operator $\forall$ to this set, but everything is designed so that this can be ignored. The unary operators `A` and `E` and the binary operator `:` have the lowest possible precedence and group to the right (it is probably best to simply use parentheses to mark the scope of each quantifier, though Marcel may remove some of them). Marcel will only accept a single variable left of the operator `:`.

notation for sets and functions: A bound variable followed by `:` followed by a sentence is a set notation; a bound variable followed by `:>` followed by an object expression is a function notation. Examples of these notations will appear later.

3 Sample propositional logic lab with extended examples

In this section we provide a sample Lab I on propositional logic. We open with extended examples illustrating the main points of reasoning in propositional logic with Marcel.

3.1 Example 1

```
>>>Start('(p?Vq?)&(r?V~p?)&(s?V~q?)->(r?Vs?)')

Line 1:



------------------

1:  (p? V q?) & (r? V ~p?) & (s? V ~q?) -> r? V s?
```

We introduce the first Marcel command, the `Start` or `s` command. This is a Marcel function which takes a string as an argument, parses it as a sentence in the logical language of Marcel and sets up an environment in which to prove it. Of course it will give various errors if the string does not parse correctly. Notice that Python expects the string to be enclosed in single quotes (the usual double quotes will also work).

```
>>> r()
Line 2
prove
(p? V q?) & (r? V ~p?) & (s? V ~q?) -> r? V s?
by assuming
(p? V q?) & (r? V ~p?) & (s? V ~q?)
and deducing
r? V s?:

1:  (p? V q?) & (r? V ~p?) & (s? V ~q?)


------------------

1:  r? V s?
```

The next Marcel command we introduce is the **Right()** or **r()** command. This is a Python function without an input (but Python wants the empty parentheses).

Notice that the format of the Marcel display is a list of numbered assumptions above and a conclusion to be proved below. The effect of the `r()` command is to apply an appropriate logical rule to the conclusion. Notice that the prover generates a note in English describing the proof strategy that it is using.

```
>>> r()
Line 3
prove
r? V s?
by denying
s?
and showing
r?:

1:  (p? V q?) & (r? V ~p?) & (s? V ~q?)

2*:  ~s?

-------------------

1:  r?
```

Here we use the `r()` command again. What is actually going on here is that either the conclusion `r` or the conclusion `s` will hold if the assumption 1 holds. But the display handles the second and subsequent alternative conclusions by assuming that they are false and attempting to deduce the first conclusion. Internally, Marcel has a second alternative conclusion `s`, but it displays it as a negative assumption ∼`s`. Marcel does have a display mode which would show multiple conclusions, but it does not sell as well to introductory logic students.

```
>>> l()
Line 4
use
(p? V q?) & (r? V ~p?) & (s? V ~q?)
by breaking it into its parts
(p? V q?) & (r? V ~p?)
and
s? V ~q?:

1:  (p? V q?) & (r? V ~p?)
2:  s? V ~q?

2*:  ~s?

------------------

1:  r?
```

Here we introduce another user command `Left()` or `l()`. This comand acts on the first assumption in a way dictated by its logical form; in this case it breaks a conjunction apart.

The names `Right()` and `Left()` are dictated by the habit of writing arguments in which a conclusion C is to be deduced from premises $P_1, \ldots, P_n$ on a single line in the format $P_1, \ldots, P_n \vdash C$ (such an expression is called a sequent). The premises are on the left and the conclusion is on the right (and there is an extended format in which one might have many alternative conclusions on the right).

```
>>> l()
Line 5
use
(p? V q?) & (r? V ~p?)
by breaking it into its parts
p? V q?
and
r? V ~p?:

1:  p? V q?
2:  r? V ~p?
3:  s? V ~q?

2*:  ~s?

------------------

1:  r?
```

We apply `Left()` again. It should be clear why we want conjunction to group to the left , so that each application of `Left` moves a single conjunct to the second position.

```
>>> gl(2)
Line 5
use
(p? V q?) & (r? V ~p?)
by breaking it into its parts
p? V q?
and
r? V ~p?:

1:  r? V ~p?
2:  s? V ~q?
3:  p? V q?

2*:  ~s?

------------------

1:  r?
```

Since our aim is to prove $\sim$`r?`, we want to manipulate the second assumption rather than the first. The command `gl` (for "get left") brings the assumption numbered by its argument to the front. We can then apply `Left()`.

This command does not generate comments.

```
>>> l()
Line 6
using hypothesis
r? V ~p?
first part: assume case 1,
r?:

1:  r?
2:  s? V ~q?
3:  p? V q?

2*:  ~s?

------------------

1:  r?
```

We use a disjunctive hypothesis, and we should expect a proof by cases. In fact, the proof breaks into two parts: the first one is presented here, and the second one will appear when this part of the proof has been finished, and we will be able to recognize it from the initial notes.

```
>>> Done()
Line 7
using hypothesis
r? V ~p?
second part: assume case 2,
~p?:

1:  ~p?
2:  s? V ~q?
3:  p? V q?

2*:  ~s?

------------------

1:  r?
```

Notice on the previous page that the conclusion to be proved is the same as the first premise. In this case we can issue the command `Done()` and the prover moves to the next part of the proof it needs you to complete, in this case the second case.

```
>>> l()
Line 8
use
~p?
by denying conclusion and proving
p?:

1:  s? V ~q?
2:  p? V q?

2*:  ~r?
3*:  ~s?

-------------------

1:  p?
```

Notice that the left rule for negation creates a new conclusion (for the sake of a contradiction, as it were) and so increases the number of alternative conclusions.

```
>>> gl(2)
Line 8
use
~p?
by denying conclusion and proving
p?:

1:  p? V q?
2:  s? V ~q?

2*:  ~r?
3*:  ~s?

-------------------

1:  p?
```

Now we bring the hypothesis which mentions `p?` to the front.

```
>>> l()
Line 9
using hypothesis
p? V q?
first part: assume case 1,
p?:

1:  p?
2:  s? V ~q?

2*:  ~r?
3*:  ~s?

------------------

1:  p?
>>> Done()
```

We use the disjunctive hypothesis, getting a proof by cases, the first case of which is trivial (so apply `Done()`).

```
Line 10
using hypothesis
p? V q?
second part: assume case 2,
q?:

1:  q?
2:  s? V ~q?

2*:  ~r?
3*:  ~s?

-------------------

1:  p?
```

Now we have the second case.

```
>>> gl(2)
Line 10
using hypothesis
p? V q?
second part: assume case 2,
q?:

1:  s? V ~q?
2:  q?

2*:  ~r?
3*:  ~s?

------------------

1:  p?
```

We don't have any assumption mentioning the conclusion, so we bring forward the assumption mentioning the first hypothesis.

```
>>> l()
Line 11
using hypothesis
s? V ~q?
first part: assume case 1,
s?:

1:  s?
2:  q?

2*:  ~r?
3*:  ~s?

------------------

1:  p?

>>> gr(3)
Line 11
using hypothesis
s? V ~q?
first part: assume case 1,
s?:

1:  s?
2:  q?

2*:  ~s?
3*:  ~p?

------------------

1:  r?
```

The negative assumption (disguising an alternative conclusion) 2* seems the only thing we can work with, but how do we get at it?

```
Line 11
using hypothesis
s? V ~q?
first part: assume case 1,
s?:

1:  s?
2:  q?

2*:  ~p?
3*:  ~r?

--------------------

1:  s?
```

```
>>> Done()
```

The `gr` (for "get right") command performs a contrapositive maneuver, negating the current conclusion and pulling out the alternative conclusion numbered by its input. The resulting goal is trivial so we apply `Done()`.

We now have all the commands strictly needed in propositional proofs.

```
Line 12
using hypothesis
s? V ~q?
second part: assume case 2,
~q?:

1:  ~q?
2:  q?

2*:  ~r?
3*:  ~s?

-------------------

1:  p?
>>> l()
Line 13
use
~q?
by denying conclusion and proving
q?:

1:  q?

2*:  ~p?
3*:  ~r?
4*:  ~s?

-------------------

1:  q?
>>> Done()
Q. E. D.
```

When the `Done()` command is executed and there is no further goal to prove, the prover displays `Q. E. D.`, signalling that the proof of the initially entered theorem is complete.

The `Showall()` command will show the entire state of the proof in progress. In this case, run at the end of the whole process, it gives a theoretically readable argument for the theorem we have proved. It can also be run during an incomplete proof, and will then show which lines we have not finished proving. The order in which the lines are presented in this output is sometimes unexpected.

The latest updates "prune" the output of the `Showall` command, removing premises and conclusions which are not used in the argument. This makes the output shorter and perhaps more human-readable. You can see this if you run the commands given under a later version.

```
>>> Showall()
Line 1:



------------------

1:  (p? V q?) & (r? V ~p?) & (s? V ~q?) -> r? V s?




proved
by Line 2

Line 2
prove
(p? V q?) & (r? V ~p?) & (s? V ~q?) -> r? V s?
by assuming
(p? V q?) & (r? V ~p?) & (s? V ~q?)
and deducing
r? V s?:

1:  (p? V q?) & (r? V ~p?) & (s? V ~q?)


------------------
```

```
1:  r? V s?



proved
by Line 3

Line 3
prove
r? V s?
by denying
s?
and showing
r?:

1:  (p? V q?) & (r? V ~p?) & (s? V ~q?)

2*:  ~s?

-------------------

1:  r?



proved
by Line 4

Line 4
use
(p? V q?) & (r? V ~p?) & (s? V ~q?)
by breaking it into its parts
(p? V q?) & (r? V ~p?)
and
s? V ~q?:
```

```
1:  (p? V q?) & (r? V ~p?)
2:  s? V ~q?

2*:  ~s?

-------------------

1:  r?




proved
by Line 5

Line 5
use
(p? V q?) & (r? V ~p?)
by breaking it into its parts
p? V q?
and
r? V ~p?:

1:  r? V ~p?
2:  s? V ~q?
3:  p? V q?

2*:  ~s?

-------------------

1:  r?




proved
```

```
by Line 6
AND
Line 7

Line 6
using hypothesis
r? V ~p?
first part: assume case 1,
r?:

1:  r?
2:  s? V ~q?
3:  p? V q?

2*:  ~s?

-------------------

1:  r?




proved

trivial

Line 7
using hypothesis
r? V ~p?
second part: assume case 2,
~p?:

1:  ~p?
2:  s? V ~q?
3:  p? V q?

2*:  ~s?
```

```
------------------

1:  r?



proved
by Line 8

Line 8
use
~p?
by denying conclusion and proving
p?:

1:  p? V q?
2:  s? V ~q?

2*:  ~r?
3*:  ~s?

------------------

1:  p?



proved
by Line 9
AND
Line 10

Line 9
using hypothesis
p? V q?
```

```
first part: assume case 1,
p?:

1:  p?
2:  s? V ~q?

2*:  ~r?
3*:  ~s?

-------------------

1:  p?



proved

trivial

Line 10
using hypothesis
p? V q?
second part: assume case 2,
q?:

1:  s? V ~q?
2:  q?

2*:  ~r?
3*:  ~s?

-------------------

1:  p?
```

```
proved
by Line 11
AND
Line 12

Line 11
using hypothesis
s? V ~q?
first part: assume case 1,
s?:

1:  s?
2:  q?

2*:  ~p?
3*:  ~r?

-------------------

1:  s?




proved

trivial

Line 12
using hypothesis
s? V ~q?
second part: assume case 2,
~q?:

1:  ~q?
2:  q?
```

```
2*:  ~r?
3*:  ~s?

--------------------

1:  p?



proved
by Line 13

Line 13
use
~q?
by denying conclusion and proving
q?:

1:  q?

2*:  ~p?
3*:  ~r?
4*:  ~s?

--------------------

1:  q?



proved

trivial

>>>
```

3.2 Example 2

The example given above shows all the commands used. A couple of shorter examples are useful to bring out particular points.

The point of the second example is to illustrate how Marcel uses implications as assumptions. Our basic rule for using an implication as an assumption is *modus ponens*, which uses two different hypotheses, of forms A and $A \to B$, to deduce B.

Marcel's approach is modified because Marcel only wants to break down one hypothesis at a time. In outline, when it is trying to prove a conclusion C from a hypothesis $A \to B$, it first tries to prove either C or A from the other hypotheses; if it proves C it is of course done; if it proves A we get B by modus ponens. It then tries to prove C from the other hypotheses and B: if it succeeds in doing this, we see that C follows from $A \to B$, and there is an application of modus ponens hiding under the hood, as it were.

```
>>>Start('(p?->q?) & (q?->r?) -> p?->r?')

Line 1:



------------------

1:  (p? -> q?) & (q? -> r?) -> p? -> r?
```

The start command, already seen.

```
>>> r()
Line 2
prove
(p? -> q?) & (q? -> r?) -> p? -> r?
by assuming
(p? -> q?) & (q? -> r?)
and deducing
p? -> r?:

1:  (p? -> q?) & (q? -> r?)


------------------

1:  p? -> r?

>>> r()
Line 3
prove
p? -> r?
by assuming
p?
and deducing
r?:

1:  p?
2:  (p? -> q?) & (q? -> r?)
------------------
1:  r?
```

The right rule for implication twice already seen, which is our familiar strategy for proving an implication.

```
>>> gl(2)
Line 3
prove
p? -> r?
by assuming
p?
and deducing
r?:

1:  (p? -> q?) & (q? -> r?)
2:  p?
-------------------
1:  r?
>>> l()
Line 4
use
(p? -> q?) & (q? -> r?)
by breaking it into its parts
p? -> q?
and
q? -> r?:

1:  p? -> q?
2:  q? -> r?
3:  p?
-------------------
1:  r?
```

Bring the conjunction of implications to the first hypothesis position and break it up.

```
>>> l()
Line 5
use
p? -> q?
, first part, showing that
p?
or the desired conclusion holds:

1:  q? -> r?
2:  p?

2*:  ~r?

-------------------

1:  p?
```

We use the implication $P \to Q$: in the first part, we prove either P or the desired conclusion, and clearly we can prove P (trivially).

```
>>> gl(2); Done()
Line 5
use
p? -> q?
, first part, showing that
p?
or the desired conclusion holds:

1:  p?
2:  q? -> r?

2*:  ~r?
------------------
1:  p?

Line 6
use
p? -> q?
second part, show that the desired conclusion follows from
q?:

1:  q?
2:  q? -> r?
3:  p?
------------------
1:  r?
```

We point out to the prover that we already have the desired conclusion P among our hypotheses. Notice that Python allows us to issue more than one command on a line (separated by semicolons). The next goal comes up.

```
>>> gl(2)
Line 6
use
p? -> q?
second part, show that the desired conclusion follows from
q?:

1:  q? -> r?
2:  p?
3:  q?


------------------

1:  r?
```

Now we get the second part of the application of $P \to Q$, which is to prove our desired conclusion from Q. We move the hypothesis $Q \to R$ to the front.

We do not comment the rest of this proof, which repeats the same maneuvers for the hypothesis $Q \to R$. Examination of our analysis of the strategy above and this example should give an idea of how to approach `modus ponens` arguments under Marcel.

```
>>> l()
Line 7
use
q? -> r?
, first part, showing that
q?
or the desired conclusion holds:

1:  p?
2:  q?

2*:  ~r?

-------------------

1:  q?
```

```
>>> gl(2)
Line 7
use
q? -> r?
, first part, showing that
q?
or the desired conclusion holds:

1:  q?
2:  p?

2*:  ~r?

-------------------

1:  q?
```

```
>>> Done()
Line 8
use
q? -> r?
second part, show that the desired conclusion follows from
r?:

1:  r?
2:  p?
3:  q?


-------------------

1:  r?




>>> Done()
Q. E. D.
>>>
```

3.3 Example 3

The next example illustrates the proof of a biconditional (which takes the expected form) and issues to do with using and proving negations, including a feature of the display which has not yet occurred in our examples.

```
>>>Start('p?->q? == ~q? -> ~p?')

Line 1:



------------------

1:  p? -> q? == ~q? -> ~p?
```

```
>>> r()
Line 2
proving biconditional
p? -> q? == ~q? -> ~p?
Part I =>
:

1:  p? -> q?


-------------------

1:  ~q? -> ~p?
```

There should be nothing surprising about the appearance of this goal here. You should expect to see the converse goal later.

```
>>> r()
Line 4
prove
~q? -> ~p?
by assuming
~q?
and deducing
~p?:

1:  ~q?
2:  p? -> q?


-------------------

1:  ~p?
```

No surprises here.

```
>>> r()
Line 5
prove
~p?
by assuming
p?
and deducing a contradiction or alternative conclusion:

1:  p?
2:  ~q?
3:  p? -> q?

------------------

_|_
```

This is a feature we haven't seen yet. When a negative conclusion is proved and no alternative conclusion can slide into its place, we get (formally speaking) no conclusion. This means that our goal is to show that the hypotheses lead to a contradiction; Marcel displays this in a way which suggests this, using $\bot$ as a symbol for the absurd conclusion.

```
>>> gl(2)
Line 5
prove
~p?
by assuming
p?
and deducing a contradiction or alternative conclusion:

1:  ~q?
2:  p? -> q?
3:  p?

------------------

_|_
```

```
>>> l()
Line 6
use
~q?
by denying conclusion and proving
q?:

1:  p? -> q?
2:  p?


-------------------

1:  q?
```

Acting on a negative hypothesis gives us a new conclusion.

```
>>> l()
Line 7
use
p? -> q?
, first part, showing that
p?
or the desired conclusion holds:

1:  p?

2*:  ~q?

------------------

1:  p?

>>> Done()
```

We are arguing by modus ponens here.

```
Line 8
use
p? -> q?
second part, show that the desired conclusion follows from
q?:

1:  q?
2:  p?


-------------------

1:  q?

>>> Done()
```

Completing a trivial example of m.p.

```
Line 3
proving biconditional
p? -> q? == ~q? -> ~p?
Part II <=
:

1:  ~q? -> ~p?


-------------------

1:  p? -> q?
```

The second half of the biconditional pops up when the first is completed.

```
>>> r()
Line 9
prove
p? -> q?
by assuming
p?
and deducing
q?:

1:  p?
2:  ~q? -> ~p?


-------------------

1:  q?
```

Expected.

```
>>> gl(2)
Line 9
prove
p? -> q?
by assuming
p?
and deducing
q?:

1:  ~q? -> ~p?
2:  p?


-------------------

1:  q?
```

Bring the implication we want to use to the front.

```
>>> l()
Line 10
use
~q? -> ~p?
, first part, showing that
~q?
or the desired conclusion holds:

1:  p?

2*:  ~q?

------------------

1:  ~q?
```

This looks superficially like a `Done()` situation, but it isn't.

```
>>> r()
Line 12
prove
~q?
by assuming
q?
and deducing a contradiction or alternative conclusion:

1:  q?
2:  p?


-------------------

1:  q?

>>> Done()
```

But it easily becomes one. Acting on the negative conclusion brings its positive component into the hypothesis, whereupon the last alternative conclusion slides into place, and the first hypothesis and conclusion are the same as desired.

```
Line 11
use
~q? -> ~p?
second part, show that the desired conclusion follows from
~p?:

1:  ~p?
2:  p?


-------------------

1:  q?
```

From contradictory hypotheses, anything, even Q, follows.

```
>>> l()
Line 13
use
~p?
by denying conclusion and proving
p?:

1:  p?

2*:  ~q?

--------------------

1:  p?




>>> Done()
Q. E. D.
>>>
```

Action on the negated statement from the contradictory pair of hypotheses moves its positive component into the conclusion, which gives the desired `Done()` condition.

3.4 Exercises for Lab I on propositional logic (lots more could be added here)

1. Here are some examples with commands set up.

```
#Examples for you to try out

#Start('p?&(p?->q?)->q?')

#Start('((p?&q?)->r?)==(p?->(q?->r?))')

#Start('p?Vq?== ~(~p?& ~q?)')
```

2. Set up the other deMorgan law in Marcel and prove it.

3. Show the validity of the rule of **constructive dilemma**: from $P \vee Q$, $P \to R$ and $Q \to S$, derive $R \vee S$. To do this, you not only need to translate the notations for the individual propositions, but also write a single larger proposition to prove.

4. Write out the theorem of Example 1 in standard propositional logic notation and write a paper proof.

5. (optional) `Start('((a? == b?) == c?) == (a? == (b? == c?))')`

4 Examples and exercises for a quantifier lab

We begin with some declarations. Propositional logic doesn't need these, but for predicate logic we need to declare some operators as properties and relations.

```
declareproperty ('P')

declareproperty('Q')

declareproperty('S')
```

These commands ensure that the parser understands `P`, `Q` and `S` as predicates (of a single object). In fact the parser would also understand `P1`, `Q1` and `S1` as predicates and if we were really proving this theorem for further use we would use the indexed forms (because they could then be replaced with other predicates as needed). But we will keep things simple.

Our example has the merit of being obvously true.

```
>>>Start('(Ax : Px -> Qx) & (Ay: Qy-> Sy) -> (Ax : Px -> Sx)')

Line 1:



------------------

1:  (Ax : Px -> Qx) & (Ay : Qy -> Sy) -> (Ax : Px -> Sx)
```

We've seen a Start command before.

```
>>> r()
Line 2
prove
(Ax : Px -> Qx) & (Ay : Qy -> Sy) -> (Ax : Px -> Sx)
by assuming
(Ax : Px -> Qx) & (Ay : Qy -> Sy)
and deducing
Ax : Px -> Sx:

1:  (Ax : Px -> Qx) & (Ay : Qy -> Sy)


------------------

1:  Ax : Px -> Sx
```

Standard strategy for proving an implication.

```
>>> r()
Line 3
prove the universal
Ax : Px -> Sx
by proving an arbitrary instance
:

1:  (Ax : Px -> Qx) & (Ay : Qy -> Sy)


-------------------

1:  Px_1 -> Sx_1
```

The strategy of universal generalization: strip off the quantifier from the universal conclusion and replace the bound variable with a fresh arbitrary object (Marcel automatically generates new indices, ensuring that each arbitrary object or instantiable it introduces is fresh).

```
>>> r()
Line 4
prove
Px_1 -> Sx_1
by assuming
Px_1
and deducing
Sx_1:

1:  Px_1
2:  (Ax : Px -> Qx) & (Ay : Qy -> Sy)


-------------------

1:  Sx_1
```

Standard strategy for proving an implication.

```
>>> gl(2)
Line 4
prove
Px_1 -> Sx_1
by assuming
Px_1
and deducing
Sx_1:

1:  (Ax : Px -> Qx) & (Ay : Qy -> Sy)
2:  Px_1


------------------

1:  Sx_1
```

Bring the complex hypothesis to the front.

```
>>> l()
Line 5
use
(Ax : Px -> Qx) & (Ay : Qy -> Sy)
by breaking it into its parts
Ax : Px -> Qx
and
Ay : Qy -> Sy:

1:  Ax : Px -> Qx
2:  Ay : Qy -> Sy
3:  Px_1


------------------

1:  Sx_1
```

Break up the complex (conjunctive) hypothesis.

```
>>> l()
Line 6
use the universal hypothesis
Ax : Px -> Qx
by creating an instance to be further specified and used:

1:  Px$2 -> Qx$2
2:  Ax : Px -> Qx
3:  Ay : Qy -> Sy
4:  Px_1


-------------------

1:  Sx_1
```

Use the universal hypothesis. The quantifier is stripped off and the variable can then be replaced by anything – Marcel allows us to delay our choice of example of the universal statement by replacing the variable with an instantiable, which can later be replaced by any desired witness **which could have be written at this point** throughout the proof.

```
>>> Inst('x_1','x$2')
Line 6
use the universal hypothesis
Ax : Px -> Qx
by creating an instance to be further specified and used:

1:  Px_1 -> Qx_1
2:  Ax : Px -> Qx
3:  Ay : Qy -> Sy
4:  Px_1


------------------

1:  Sx_1
```

A new user command is introduced. `Inst` takes two arguments, an expression for an object and an instantiable, and replaces the instantiable with the object expression throughout the current proof (including inside other parts of the proof which we cannot see). Any free variables occurring in the object expression (whether arbitrary objects or instantiables) must have lower index than the instantiable for which we are making the substition (the object expression needs to be one we could have intended as a witness as the time the instantiable was created).

```
>>> l()
Line 7
use
Px_1 -> Qx_1
, first part, showing that
Px_1
or the desired conclusion holds:

1:  Ax : Px -> Qx
2:  Ay : Qy -> Sy
3:  Px_1

2*:  ~ Sx_1

-------------------

1:  Px_1
```

```
>>> gl(3)
Line 7
use
Px_1 -> Qx_1
, first part, showing that
Px_1
or the desired conclusion holds:

1:  Ay : Qy -> Sy
2:  Px_1
3:  Ax : Px -> Qx

2*:  ~ Sx_1

-------------------

1:  Px_1
```

Modus ponens at work

```
Line 7
use
Px_1 -> Qx_1
, first part, showing that
Px_1
or the desired conclusion holds:

1:  Px_1
2:  Ax : Px -> Qx
3:  Ay : Qy -> Sy

2*:  ~ Sx_1

-------------------

1:  Px_1



>>> Done()
```

Modus ponens finished

```
Line 8
use
Px_1 -> Qx_1
second part, show that the desired conclusion follows from
Qx_1:

1:  Qx_1
2:  Ax : Px -> Qx
3:  Ay : Qy -> Sy
4:  Px_1


-------------------

1:  Sx_1
```

We repeat the same maneuver

```
>>> gl(3)
Line 8
use
Px_1 -> Qx_1
second part, show that the desired conclusion follows from
Qx_1:

1:  Ax : Px -> Qx
2:  Ay : Qy -> Sy
3:  Px_1
4:  Qx_1


-------------------

1:  Sx_1
```

gl(3) actually works by two rotations of the premises

```
Line 8
use
Px_1 -> Qx_1
second part, show that the desired conclusion follows from
Qx_1:

1:  Ay : Qy -> Sy
2:  Px_1
3:  Qx_1
4:  Ax : Px -> Qx


------------------

1:  Sx_1
```

We have the desired premise in first position

```
>>> l()
Line 9
use the universal hypothesis
Ay : Qy -> Sy
by creating an instance to be further specified and used:

1:  Qy$3 -> Sy$3
2:  Ay : Qy -> Sy
3:  Px_1
4:  Qx_1
5:  Ax : Px -> Qx


-------------------

1:  Sx_1
```

Introduce an instantiable as before

```
>>> Inst('x_1','y$3')
Line 9
use the universal hypothesis
Ay : Qy -> Sy
by creating an instance to be further specified and used:

1:  Qx_1 -> Sx_1
2:  Ay : Qy -> Sy
3:  Px_1
4:  Qx_1
5:  Ax : Px -> Qx


------------------

1:  Sx_1
```

Set the instantiable to x_1 as before, and the rest is propositional logic.

```
>>> l()
Line 10
use
Qx_1 -> Sx_1
, first part, showing that
Qx_1
or the desired conclusion holds:

1:  Ay : Qy -> Sy
2:  Px_1
3:  Qx_1
4:  Ax : Px -> Qx

2*:  ~ Sx_1

-------------------

1:  Qx_1
```

```
>>> gl(3)
Line 10
use
Qx_1 -> Sx_1
, first part, showing that
Qx_1
or the desired conclusion holds:

1:  Px_1
2:  Qx_1
3:  Ax : Px -> Qx
4:  Ay : Qy -> Sy

2*:  ~ Sx_1

-------------------

1:  Qx_1
```

```
Line 10
use
Qx_1 -> Sx_1
, first part, showing that
Qx_1
or the desired conclusion holds:

1:  Qx_1
2:  Ax : Px -> Qx
3:  Ay : Qy -> Sy
4:  Px_1

2*:  ~ Sx_1

--------------------

1:  Qx_1



>>> Done()
```

```
Line 11
use
Qx_1 -> Sx_1
second part, show that the desired conclusion follows from
Sx_1:

1:  Sx_1
2:  Ay : Qy -> Sy
3:  Px_1
4:  Qx_1
5:  Ax : Px -> Qx


------------------

1:  Sx_1




>>> Done()
Q. E. D.
>>>
```

It is worth pointing out that order matters here. Arbitrary objects must be introduced before instantiables which might need to depend on them. We repeat the opening of the previous proof with a subtle change.

```
>>>Start('(Ax1 : Px1 -> Qx1) & (Ay1: Qy1-> Sy1) -> (Ax1 : Px1 -> Sx1)')

Line 1:



------------------
```

```
1:  (Ax1 : Px1 -> Qx1) & (Ay1 : Qy1 -> Sy1) -> (Ax1 : Px1 -> Sx1)
```

```
>>> r()
Line 2
prove
(Ax1 : Px1 -> Qx1) & (Ay1 : Qy1 -> Sy1) -> (Ax1 : Px1 -> Sx1)
by assuming
(Ax1 : Px1 -> Qx1) & (Ay1 : Qy1 -> Sy1)
and deducing
Ax1 : Px1 -> Sx1:

1:  (Ax1 : Px1 -> Qx1) & (Ay1 : Qy1 -> Sy1)


------------------

1:  Ax1 : Px1 -> Sx1
```

As before

```
>>> l()
Line 3
use
(Ax1 : Px1 -> Qx1) & (Ay1 : Qy1 -> Sy1)
by breaking it into its parts
Ax1 : Px1 -> Qx1
and
Ay1 : Qy1 -> Sy1:

1:  Ax1 : Px1 -> Qx1
2:  Ay1 : Qy1 -> Sy1


-------------------

1:  Ax1 : Px1 -> Sx1
```

As before

```
>>> l()
Line 4
use the universal hypothesis
Ax1 : Px1 -> Qx1
by creating an instance to be further specified and used:

1:  Px$1 -> Qx$1
2:  Ax1 : Px1 -> Qx1
3:  Ay1 : Qy1 -> Sy1


------------------

1:  Ax1 : Px1 -> Sx1
```

We try using the universal hypothesis before setting up the proof of the universal conclusion ...

```
>>> r()
Line 5
prove the universal
Ax1 : Px1 -> Sx1
by proving an arbitrary instance
:

1:  Px$1 -> Qx$1
2:  Ax1 : Px1 -> Qx1
3:  Ay1 : Qy1 -> Sy1


-------------------

1:  Px_2 -> Sx_2
```

As before, though belated

```
>>> Inst('x_2','x$1')
substitution fails because term is newer than instantiable
Line 5
prove the universal
Ax1 : Px1 -> Sx1
by proving an arbitrary instance
:

1:  Px$1 -> Qx$1
2:  Ax1 : Px1 -> Qx1
3:  Ay1 : Qy1 -> Sy1


------------------

1:  Px_2 -> Sx_2




>>>
```

And the `Inst` command fails. The problem is that x_2 is an object that one could not have chosen as a witness when the instantiable `x$1` was created.

The following is a truly strange example.

```
>>>Start('Ex:Ay:Py -> Px')

Line 1:



------------------

1:  Ex : Ay : Py -> Px
```

```
>>> r()
Line 2
prove the existential
Ex : Ay : Py -> Px
by introducing an instance to be further specified then proved:


2*:  ~ Ex : Ay : Py -> Px

--------------------

1:  Ay : Py -> Px$1
```

```
>>> r()
Line 3
prove the universal
Ay : Py -> Px$1
by proving an arbitrary instance
:


2*:  ~ Ex : Ay : Py -> Px

-------------------

1:  Py_2 -> Px$1
```

```
>>> r()
Line 4
prove
Py_2 -> Px$1
by assuming
Py_2
and deducing
Px$1:

1:  Py_2

2*:  ~ Ex : Ay : Py -> Px

-------------------

1:  Px$1
```

```
>>> gr(2)
Line 4
prove
Py_2 -> Px$1
by assuming
Py_2
and deducing
Px$1:

1:  Py_2

2*:  ~ Px$1

-------------------

1:  Ex : Ay : Py -> Px
```

```
>>> r()
Line 5
prove the existential
Ex : Ay : Py -> Px
by introducing an instance to be further specified then proved:

1:  Py_2

2*:  ~ Ex : Ay : Py -> Px
3*:  ~ Px$1

-------------------

1:  Ay : Py -> Px$3
```

```
>>> r()
Line 6
prove the universal
Ay : Py -> Px$3
by proving an arbitrary instance
:

1:  Py_2

2*:  ~ Ex : Ay : Py -> Px
3*:  ~ Px$1

------------------

1:  Py_4 -> Px$3
```

```
>>> r()
Line 7
prove
Py_4 -> Px$3
by assuming
Py_4
and deducing
Px$3:

1:  Py_4
2:  Py_2

2*:  ~ Ex : Ay : Py -> Px
3*:  ~ Px$1

------------------

1:  Px$3
```

```
>>> Inst('y_2','x$3')
Line 7
prove
Py_4 -> Px$3
by assuming
Py_4
and deducing
Px$3:

1:  Py_4
2:  Py_2

2*:  ~ Ex : Ay : Py -> Px
3*:  ~ Px$1

------------------

1:  Py_2
```

```
>>> gl(2)
Line 7
prove
Py_4 -> Px$3
by assuming
Py_4
and deducing
Px$3:

1:  Py_2
2:  Py_4

2*:  ~ Ex : Ay : Py -> Px
3*:  ~ Px$1

-------------------

1:  Py_2




>>> Done()
Q. E. D.
>>>
```

A feature which we quite deliberately do not use in these examples is that the prover will automatically instantiate dollar-sign variables (instantiables) when the `Done` command is issued if it can do so in a way which will cause the initial premise and the conclusion to be identical. In proofs in pure quantifier logic, this means that explicit use of the `Inst` command can be avoided. We will see this style in later sections. The danger is that the prover will quite happily make substitutions which turn out to be not the ones that you want.

4.1 Quantifier lab exercises (lots more could be added here)

1. Prove the following theorems under Marcel. Write proofs in mathematical English of the same results.

   (a)
   ```
   declarerelation('R')  #this command declares a
        #(binary infix) relation predicate R

   Start('(Ax : Ey : x R y) & (Ax : Ay : x R y -> y R x) \
       & (Ax : Ay : Az : x R y & y R z -> x R z) -> (Ax : x R x)')
   ```

   (b)
   ```
   declareproperty('F')
   declareproperty('P')
   declarerelation('R')
   declareproperty('S')
   declareproperty('T')
   Start('(Ax:Fx -> Ay:Sy -> x R y)&(Ax:Px-> \
      (Ay:xRy -> Ty))->(Ex:Fx & Px) -> (Ay:Sy->Ty)')
   ```

   (c)
   ```
   declarerelation('L')
   declarerelation('M')
   Start('(Ey:Ax:xLy -> xMy)->(Ex:Ay:xLy) -> (Ey:Ex:xMy)')
   ```

2. Write an English proof of the last example before the exercises.

5 Technical Reference

5.1 Introduction

This file is the manual for the software originally implemented in the file `marcel.sml` and now implemented in `pythonmarcel.py`.

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 (in `pythonmarcel.py` the extensionality rule is weaker than in the usual formulations 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.

It should be noted that as we write this manual for `pythonmarcel.py` we

have lifted the math sections straight from the manual for `marcel.sml`, and the rules actually implemented in the Python version may be found to have technical differences from the ones given here. This will be checked over in due course.

5.2 Sequent Rules in General

We give a mathematical description of the rules by which validity of sequents may be recognized or deduced from the postulated validity of other sequents.

We should initially say that a sequent is a notation $P_1, P_2, P_3, \ldots \vdash Q_1, Q_2, Q_3 \ldots$ (in which the set of premises P_i or conclusions Q_1 may have 0 or 1 elements with appropriate changes in the notation) which we say is valid if all assignments of semantics to variable components in the sequent which make all the premises true also make at least one of the conclusions true. Marcel will generally display a sequent which it thinks of internally as $P_1, P_2, P_3, \ldots \vdash Q_1, Q_2, Q_3 \ldots$ in the form $P_1, P_2, P_3, \ldots, \neg Q_2, \neg Q_3, \ldots \vdash Q_1$ with one conclusion (or none), though it does support a multiple-conclusion display format.

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.

5.3 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 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*.

5.4 Axiom

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

5.5 Left rules

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

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

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

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

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

5.6 Right rules

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

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

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

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

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

5.7 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.

5.8 Rules for Quantifiers

5.8.1 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

5.8.2 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

5.8.3 Comments on Quantifier Rules

In some 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.

5.9 Rules for Membership

We add the remark here to be made good later that the prover supports not only set abstraction but also function abstraction (λ-terms) and rules for this feature should also be stated.

5.9.1 Left Rule

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

when ϕ is stratified

5.9.2 Right Rule

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

when ϕ is stratified

5.10 Rules for Equality

5.10.1 Left Rule

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

5.10.2 Right Rule

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

This is an axiom: it requires no premises

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

5.10.3 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.

5.11 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 (recognizable because it contains a dollar sign) with a fresh index (belonging to the same series of indices as those on the free variables introduced by the left existential and right universal rules).

The `Inst` 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 unknown variables 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.

5.12 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 `Usethm` 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.

5.13 Command Reference

All commands are Python functions. A function without arguments will need to be supplied with a null argument (a pair of parentheses) as in `Left()`. Arguments which are Marcel identifiers or bits of Marcel text are strings as far as Python is concerned, and should be enclosed in single or double quotes, as in `Start('p? ->?p')` or `Start("p? ->?p")`. Quotes do not appear in Marcel text, so which quotes you use should not be an issue. Integer arguments do not need to be quoted. Arguments need to be enclosed in parentheses and separated by commas when there are two or more of them. It should be noted that Python is case sensitive, and so is Marcel itself.

setlog: Takes one string argument `<filename>`: opens the log file `<filename>logfile.py`. It is really important to issue another setlog command to **close** the log file you are working with when you are done: my habit is to issue `setlog('done')` at the end of any session where I am producing log files.

addtolog: As **setlog**, but appending to the file rather than initializing it. The same remark about closing the file applies: it is really important to issue another setlog command to **close** the log file you are working with when you are done: my habit is to issue `setlog('done')` at the end of any session where I am producing log files.

Demo: Turns demo mode on or off. In this mode, user commands are echoed and the system pauses until the user hits return. This is a handy mode to run log files under (insert `Demo()` manually at the beginning of the file).

Back(): Moves the state of the current proof back to that before the execution of the last logged user command. Note that the Start command clears all history. This command is entered in log files but does not participate in demo mode nor is it entered in the history it consults (it is semi-logged).

Forward(): Undoes immediately preceding Back() commands: the forward history is cleared as soon as any logged user command other than Back or Forward is executed. This command is entered in log files but does not participate in demo mode nor is it entered in the history it consults (it is semi-logged).

Forget(): Clear back and forward history. This command is entered in log files but does not participate in demo mode nor is it entered in the history it consults (it is semi-logged).

setprecsame: Two arguments, binary operator names: sets the precedence of the first one to be the same as the precedence of the second one. For all precedence commands, notice that if you set the precedence of an operator higher than the minimum possible for its type, the system will not allow you to set its precedence again.

setprecevenabove: Two arguments, binary operator names: sets the precedence of the first one just higher than that of the second one, grouping to the right. For all precedence commands, notice that if you set the precedence of an operator higher than the minimum possible for its type, the system will not allow you to set its precedence again.

setprecoddabove: Two arguments, binary operator names: sets the precedence of the first one just higher than that of the second one, grouping to the left. For all precedence commands, notice that if you set the precedence of an operator higher than the minimum possible for its type, the system will not allow you to set its precedence again.

setprecevenbelow: Two arguments, binary operator names: sets the precedence of the first one just lower than that of the second one, grouping to the right. For all precedence commands, notice that if you set the precedence of an operator higher than the minimum possible for its type, the system will not allow you to set its precedence again.

setprecoddbelow: Two arguments, binary operator names: sets the precedence of the first one just lower than that of the second one, grouping to the left. For all precedence commands, notice that if you set the precedence of an operator higher than the minimum possible for its type, the system will not allow you to set its precedence again.

declareproperty: One argument, an operator name. Declare a unary operator with object input and proposition output.

declarefunction: One argument, an operator name. Declare a unary operator with object input and output of the same relative type.

declarescfunction: One argument, an operator name. Declare a unary operator with object input and output bounded in a strongly cantorian set.

declaretypedfunction: Two arguments, an operator name and an integer. Declare a unary operator with object input and relative type of the output minus the relative type of the input equal to the numeral argument.

declarerelation: One argument, an operator name, a binary operator taking two inputs of the same relative type.

declareoperation: One argument, an operator name, a binary operator taking two inputs of the same relative type as the object output.

declarescoperation: One argument, an operator name, a binary operator taking two inputs of the same relative type and returning an output bounded in a strongly cantorian set.

declaretypedrelation: Two arguments, an operator name and a numeral, a binary operator taking two inputs with the relative type of the second differing from the relative type of the first by the numeral argument, and output a proposition.

declaretypedoperation: Three arguments, an operator name and two numerals: the first is the name of a binary operator; the second is the displacement of the type of its first input from the type of the output; the second is the displacement of the type of its second input from the type of the output.

declaretypedscoperation: Two arguments, an operator name and a numeral: the first is the name of a binary operator; the second is the displacement of the type of the second input from the displacement of the first input; the output is bounded in a strongly cantorian set.

Declareconstant: One argument, an object constant.

Oneconclusion: No arguments. Sets display to have conclusions after the first displayed as negated hypotheses, but with underlying structure unchanged.

Manyconclusions: No arguments. Sets display to have multiple conclusions.

Oneconclusion2: No arguments. Sets system to automatically negate conclusions after the first and make them premises.

Manyconclusions2: No arguments. Turns off `Oneconclusion2` mode.

Start: One argument, a proposition. Create a new proof with no premises and the argument as the conclusion. The macro **s** abbreviates the same command.

Nextgoal: No arguments. Change which goal is viewed.

Look: No arguments. View the current goal. If the proof is complete, `Q. E. D` will be displayed followed by the theorem which has been proved.

Done: No arguments. Marks the current goal as proved if the first premise and the first conclusion match (either are identical or can be unified by suitable proof-wide substitutions for instantiables and propositional and operator variables; be aware that automatic unification will not act on variables appearing in the theorem to be proved). It will then display the next unproved goal. If the proof is complete, `Q. E. D` will be displayed followed by the theorem which has been proved.

Left: No arguments. Applies the appropriate logical rule to the first premise of the current goal. The macro **l()** abbreviates the same command.

Right: No arguments. Applies the appropriate logical rule to the first conclusion of the current goal. The macro **r()** abbreviates the same command.

Pruneleft: Removes the first premise of the current goal.

Pruneright: Removes the first conclusion of the current goal.

Define: One argument, a Marcel binary expression with the illegal operator `:=`, the left input being a constant or unary or binary expression with variable inputs (the latter two enclosed in parentheses) and the right input being the body of the definition. Marcel will automatically compute the correct type for unary or binary operators. The body

must contain no indexed constants, instantiables, free occurrences of bound variables other than the inputs of the operation to be defined, or propositional or operator variables.

Getleft: Rotate premises of current goal by one, moving second to first position. Macro **gl** takes a numeral argument, which is the index of the premise it brings to the front by repeating this command. In the latest versions, `gl` is also a user command (and so appears in log files), which invokes "interal" versions of `Getleft`, but `Getleft` remains a user command.

GetleftE: Rotate premises of current goal after the first by one, moving third to second position. Used for equational reasoning. Macro **gle** takes a numeral argument, which is the index of the premise it brings to the front by repeating this command. This macro is also a user command.

Getleft2E: Rotate premises after the first by one, moving last to second position. Used for equational reasoning.

Getright2: Rotate conclusions after the first, moving last to first position.

Getleft2: Rotate premises after the first by one, moving last to first position.

Getright: Rotate conclusions by one, moving second to first position. Macro **gr** takes a numeral argument, which is the index of the conclusion it brings to the front by repeating this command. In the latest versions, `gr` is also a user command (and so appears in log files), which invokes "interal" versions of `Getleft`, but `Getleft` remains a user command.

Cut: One argument, a proposition: replace current goal by two goals, one with the argument added to premises and one with it added to the conclusions.

Equaldr0: An equation command. The equation or biconditional is applied by replacing left side with right side. The equation or biconditional applied is the first premise, applied to the first conclusion. All occurrences rewritten without unification.

Equaldl0: An equation command. The equation or biconditional is applied by replacing left side with right side. The equation or biconditional applied is the first premise, applied to the second premise. All occurrences rewritten without unification.

Equalcr0: An equation command. The equation or biconditional is applied by replacing right side with left side. The equation or biconditional applied is the first premise, applied to the first conclusion. All occurrences rewritten without unification.

Equalcl0: An equation command. The equation or biconditional is applied by replacing right side with left side. The equation or biconditional applied is the first premise, applied to the second premise. All occurrences rewritten without unification.

Equaldr1: An equation command. The equation or biconditional is applied by replacing left side with right side. The equation or biconditional applied is the first premise, applied to the first conclusion. Leftmost occurrence rewritten, with unification.

Equaldl1: An equation command. The equation or biconditional is applied by replacing left side with right side. The equation or biconditional applied is the first premise, applied to the second premise. Leftmost occurrence rewritten, with unification.

Equalcr1: An equation command. The equation or biconditional is applied by replacing right side with left side. The equation or biconditional applied is the first premise, applied to the first conclusion. Leftmost occurrence rewritten, with unification.

Equalcl1: An equation command. The equation or biconditional is applied by replacing right side with left side. The equation or biconditional applied is the first premise, applied to the second premise. Leftmost occurrence rewritten, with unification.

Equaldr2: An equation command. The equation or biconditional is applied by replacing left side with right side. The equation or biconditional applied is the first premise, applied to the first conclusion. Rightmost occurrence rewritten, with unification.

Equaldl2: An equation command. The equation or biconditional is applied by replacing left side with right side. The equation or biconditional applied is the first premise, applied to the second premise. Rightmost occurrence rewritten, with unification.

Equalcr2: An equation command. The equation or biconditional is applied by replacing right side with left side. The equation or biconditional applied is the first premise, applied to the first conclusion. Rightmost occurrence rewritten, with unification.

Equalcl2: An equation command. The equation or biconditional is applied by replacing right side with left side. The equation or biconditional applied is the first premise, applied to the second premise. Rightmost occurrence rewritten, with unification.

Inst: Two arguments, an object term and an instantiable. The instantiable is replaced by the term everywhere in the current proof, if the maximum index appearing in the term is less than the index of the instantiable.

SU: One argument, a string, a Marcel term with operator `:=`: the right input term replaces the left input instantiable everywhere in the proof. A variant of Inst.

PropInst: Two arguments, a term and a propositional variable. The term is to replace the propositional variable throughout the current proof, subject to type conditions.

OpInst: Two arguments, an operator and an operator variable. The operator is to replace the operator variable throughout the current proof, subject to type conditions.

Showall: Show all goals in the current proof. In the latest versions, the proof will be automatically pruned of all premises and conclusions not used in the proof of the sequent they inhabit once the entire theorem is completely proved.

Saveproof: One argument, a string name under which the current proof is to be saved.

Loadproof: One argument, a string name of the saved proof to be retrieved.

Savetheorem: Two arguments, a line name and a name for a new theorem. If the line is proved, save the line with its proof as a theorem under the name given.

Savethetheorem: One argument, the name of the theorem to be proved. The top level sequent of the current proof is recorded as a theorem if it has indeed been proved.

Axiom: Two arguments, a string name for an axiom and a proposition term. Subject to type conditions, the proposition is saved as an axiom.

Usethm: One argument, a name of a theorem to be used in effect as a rule in the current proof.

Showdef: One argument, the name of an operator whose definition is displayed.

Showthm: One argument, the name of a theorem which is displayed.

Showdec: One argument, the name of an operator for which internal type and precedence information are displayed.