College Mathematics with Lestrade

M. Randall Holmes

October 4, 2016

A note: To make this look more like an undergraduate discrete math text, it needs much more substantial comment on the Lestrade text given in later sections. There should be enough discussion that the Lestrade text can be seen to implement the English discussion. I am quite interested in so adorning it: it is to be expected that this will happen.

1 Introduction

This book is intended to introduce college level discrete and foundational mathematics to the Reader with the help of a companion, the Lestrade Type Inspector, whom we will familiarly call Inspector Lestrade.[1] Lestrade is a piece of software, a "dependent type checker". The Reader doesn't need to know this, although she will discover something about this in the course of her encounters with him. Dialogues with Lestrade will be embedded in the text to illustrate our points, and the Reader will be told in due course how to initiate dialogues with Lestrade herself.

[1] I am being disingenuous here, of course. My previous published book pretends to be an elementary set theory textbook, though it is nothing of the sort. It could be used as such, and it would further my aims if this book could be refined to the point where it could actually be used for the imagined purpose, as well.

2 Natural Numbers

We will begin in a very sensible place for discrete mathematics: we will introduce the usual natural numbers, which, following the ancient Greeks, for us start with 1 and continue from there in single steps:

$$1, 2, 3, 4, 5, \ldots$$

You should always be suspicious of a mathematician when they write ....
You should suspect that they are cheating.

We attempt to be more honest, and introduce our friend the Inspector.

```
>> Inspector Lestrade says:
>>   Welcome to the Lestrade Type Inspector,
>>    full version of 8/20/2016 (update to rewrited to be tested)
>>   2:30 pm Boise time


% Lestrade text supporting discrete math text.

construct Nat type

>> Nat: type {move 0}


construct 1 in Nat

>> 1: in Nat {move 0}


declare n in Nat

>> n: in Nat {move 1}


construct Succ n in Nat

>> Succ: [(n_1:in Nat) => (---:in Nat)]
```

```
>>   {move 0}
```

In this dialogue with the Inspector, we first tell him that there is a type of mathematical object which we call `Nat` (the natural numbers). We tell the Inspector that there is a natural number called `1`. We then make a different move (the use of the `declare` command instead of the `construct` command signals this) and introduce a natural number variable `n`. We do this in order to state that there is a function `Succ` which takes any natural number `n` to a natural number `Succ n` (its successor). We may write $\sigma(n)$ instead of `Succ n` when talking to the Reader rather than to the Inspector.

There are some obvious things that we can do.

```
define 2 :  Succ 1

>> 2: [(Succ(1):in Nat)]
>>   {move 0}


define 3 : Succ 2

>> 3: [(Succ(2):in Nat)]
>>   {move 0}


define 4 : Succ 3

>> 4: [(Succ(3):in Nat)]
>>   {move 0}


define 5 : Succ 4

>> 5: [(Succ(4):in Nat)]
>>   {move 0}
```

The Inspector knows how to implement the time-honored mathematical practice of defining new symbols. These will do to go on with for now.

The Reader should have no illusions. At this point Lestrade knows very little about natural numbers. They could, for all he knows all be the same object `1` with `Succ 1 = 1`. Or they could tidily close up into the little circle of five objects, with `Succ 5 = 1`. Or more confusingly, `1, 2, 3` might all be different but `Succ 5 = 4`. We will succeed in briefing the Inspector so that he can rule out these curious possibilities, but this requires a little discussion of logic.

3 A first introduction to objects and functions

The Inspector sees a world inhabited by objects and functions. Objects and functions are of more or less complicated sorts.

The sorts of objects are best discussed first: an object is either a proposition (of sort `prop`) or the proof of some proposition `p` (of sort `that p`) or a type label (`Nat` is a type label) of sort `type` or an untyped mathematical object (of sort `obj`) or an object of a mathematical type labelled by a type label `tau` (such an object is of sort `in tau`). The mathematical objects we introduce to the Inspector in this narrative will usually be typed, but we may make some use of `obj`.

A function will take a fixed number of inputs (which will be objects or functions of stated sorts) and produce an object output of a stated object sort. The sort of a function (these can be quite complex) is determined by the sorts of its input and output. For example, `Succ` belongs to a sort inhabited by functions which take an input of type `Nat` and output an object of type `Nat`.

We can define new functions just as we defined new objects above:

```
define Addtwo n : Succ Succ n

>> Addtwo: [(n_1:in Nat) => (Succ(Succ(n_1)):in Nat)]
>>   {move 0}
```

The function `Addtwo` applies `Succ` twice, and you can see that as expected it has the same sort as `Succ`.

Our method of defining functions parallels the very familiar way of defining functions in algebra: $f(x) = x^2 + 1$, for example. In some sense Lestrade is entirely driven by the way it handles variables and function definitions, and with this in mind we give examples which bring out features we will use.

```
declare m in Nat

>> m: in Nat {move 1}
```

For what follows, we need another natural number variable, so we declare it. Notice the annotation `move` 1. At any point in a Lestrade session, one is at a particular "move" (indexed by a non-negative integer). If one is in move n, everything declared at move m for $m \leq n$ is regarded as fixed, and anything declared at move $n+1$ is regarded as variable (in essence, these are "arbitrary" or "hypothetical" objects which may be presented in the future, about which we know nothing except their sorts). Nothing is declared at move $n+2$ or higher. [2]

If we are at move n, the `declare` command is used to introduce objects (not functions) at move $n+1$. This is how we declare object variables.

The `construct` command is used to introduce objects at `move` n (new constants) in two different ways: one can construct an object declared at `move` n in basically the same way one declares a variable (the declaration of 1 above is an example).

One can construct a function at move n by a command `construct f x1 ... xn : sort`, in which f is a fresh identifier which will represent the function and the names `x1, ... xn` are declared at move $n+1$: this is intended to postulate a primitive function f which will take any list of arguments $t_1, \ldots, t_n$ with types matching those of the given parameters to an output value of type `sort`.

There are some formal restrictions. None of `x1,...,xn` should be defined objects. A variable `xi` may appear in the sorts of later `xj`'s or in `sort`: this is what gives Lestrade a lot of its power. If some `xi` or `sort` depends on some move $n+1$ object not appearing in the argument list, the missing object may be supplied as an argument implicitly, as we will see below; if Lestrade cannot fill in the argument list in such a way that any dependencies of `sort`

[2]The temporal metaphor can be used to address an objection raised, for example, by Bertrand Russell. He found it absurd to speak of an arbitrary triangle concerning which one did not know whether it was acute, obtuse, right, etc. We think of mathematical objects as eternal things given (to God?) in every detail. The metaphor used here suggests that at least the view we take implicitly in Lestrade is not logically absurd. A name at move $n+1$ (of an object of a given sort) is the name of an object we have not yet constructed (or selected) or which is not yet given to us. We might not admit any actual temporality in the realm of mathematical objects, but the ordinary logic of dealing with objects in the real world can show us that at least we will not be led into logical absurdities. The house we will build tomorrow will be green or not green, though we do not presently have evidence for either condition, and at present it is arguably neither. The triangle we posit need not be posited with all its details; these may be revealed bit by bit in subsequent moves.

appear in the argument list and any dependencies of the sort of any variable in the argument list appear earlier in the argument list, he will decline to postulate the function.

The `construct` command is our vehicle for introducting axioms and primitive (undefined) notions.

One can define an object or function at `move` n: defining an object is straightforward (see the definitions of 2–5 above).

Defining a function has the format `define f x1 ... xn : T` where `T` is an object term. This implements precisely the maneuver already familiar to you in function definitions like $f(x) = x^2 + 1$ or $f(x, y) = x^2 + y^2$.

Here are the formal restrictions. The `xi`'s are just as in the `construct` command. The effect of such a definition is just as one would imagine from algebra: the work of Lestrade is to determine that the term `T` can be assigned a sort `Tsort`, then in effect execute `construct f x1 ... xn : Tsort`, and insert the extra information that this function has the specific output signalled by `T` (of course varying as the inputs vary, in the way indicated by appearances of the inputs in the term `T`).

```
construct + n m : in Nat

>> +: [(n_1:in Nat),(m_1:in Nat) => (---:in Nat)]
>>   {move 0}


construct * n m : in Nat

>> *: [(n_1:in Nat),(m_1:in Nat) => (---:in Nat)]
>>   {move 0}


construct ^ n m : in Nat

>> ^: [(n_1:in Nat),(m_1:in Nat) => (---:in Nat)]
>>   {move 0}
```

```
define f n : (n^2) + 1

>> f: [(n_1:in Nat) => (((n_1 ^ 2) + 1):in Nat)]
>>   {move 0}
```

We introduce some declarations and define the function $f(n) = n^2 + 1$ of a natural number variable n. Notice how the body of the definition of f appears in its type information (in the same place where — appears in the declaration of a primitive function).

Additional operations allow us to temporarily fix our current variables, or to declare function variables.

The `open` command increments the move counter, so that we treat what were variables at `move` $n+1$ as constants, and become able to declare variables at `move` $n + 2$.

The `close` command decrements the move counter, so that we discard all `move` $n + 1$ declarations and return to viewing `move` n declarations as declarations of variables. Notice that if we `open`, declare some variables, then construct a function, then `close`, we obtain a function at move $n + 1$, that is, a function variable. Defining instead of constructing the function would give us a variable expression. Functions can have both object and function inputs, though we allow them to have only object outputs. Reasons for this last restriction may be discussed later.

A subtle point to notice that that when functions are declared at move n, any defined move $n + 1$ notions which appear in their sort information must be expanded out. This can cause actual expansions when a defined function appears in applied position, or wherever a defined object appears; when a defined function appears as an argument to another function, it is replaced by its own sort information (which contains the body of its definition, so works as anonymous notation for the function).

```
construct Natfun2 type

>> Natfun2: type {move 0}
```

```
open

     declare x in Nat

>>       x: in Nat {move 2}


     declare y in Nat

>>       y: in Nat {move 2}


     construct F x y : in Nat

>>       F: [(x_1:in Nat),(y_1:in Nat) => (---:in Nat)]
>>         {move 1}


     close

construct natfun2 F : in Natfun2

>> natfun2: [(F_1:[(x_2:in Nat),(y_2:in Nat) => (---:in
>>              Nat)])
>>         => (---:in Natfun2)]
>>   {move 0}
```

The preceding block is a bit technical. We declare a type of functions with two natural number inputs and a natural number output: elements of this type are *objects* constructed from actual functions of the given type using the function `natfun2`. This is an indication of how we can get around any restrictions imposed by functions having only object outputs: we can now define a function with output in effect a function from two natural numbers to a natural number by packaging the output with the operator `natfun2`.

The block also illustrates how to construct a function variable.

```
declare a in Nat

>> a: in Nat {move 1}


declare b in Nat

>> b: in Nat {move 1}


open

     declare x in Nat

>>       x: in Nat {move 2}


     declare y in Nat

>>       y: in Nat {move 2}


     define g x y : (a * x) + b * y

>>       g: [(x_1:in Nat),(y_1:in Nat) => (((a * x_1) +
>>              (b * y_1)):in Nat)]
>>         {move 1}


     close

define Lincomb a b : natfun2 g

>> Lincomb: [(a_1:in Nat),(b_1:in Nat) => (natfun2([(x_2:
>>              in Nat),(y_2:in Nat) => (((a_1 * x_2) + (b_1
>>              * y_2)):in Nat)])
>>         :in Natfun2)]
```

```
>>   {move 0}
```

In this block we illustrate the Lestrade implementation of a subtle distinction between variables which we all learn about in algebra. We all know how to read a function definition $g(x, y) = ax + by$, once we are told that a and b are constants. But of course a and b are variables, just less variable than x and y. The distinction between the `move` 1 variables a and b and the `move` 2 variables in x and y in Lestrade implements this distinction.

Finally, the function `Lincomb` takes input a, b and returns the function $g(x, y) = ax + by$ "packaged" with the operation `natfun2`. Notice that the name g (defined at move 1) is replaced with the term

```
[(x_2:  in Nat),(y_2:in Nat) => (((a_1 * x_2) + (b_1 * y_2)):in
                                 Nat)],
```

an anonymous description of the function represented by g which makes no use of identifiers declared at move 1 (notice that the parameters x, y are replaced by indexed dummy variables). We may make some use of notations $(x, y \mapsto ax + by)$ when talking to our Reader; it is a notable feature of Lestrade that the user never has to type terms of this form: functions are always referenced by names, set by definitions analogous to $f(x) = x^2 + 1$, in Lestrade text written by the user.

4 A first introduction to propositions and proofs: the logic of "and"

The sort `prop` of propositions is inhabited by statements, the sort of things that we say, believe, conjecture, prove, disprove, etc. Any particular proposition `p` is associated with a sort `that p` which we briefly name "proofs of `p`" but which we might also reasonably call "evidence for `p`". When we assume a statement for the sake of argument, we are supposing that it is true (that there is evidence for it) not that we actually have a proof for it.

There are operations on propositions, which are probably familiar to the Reader, embodied in such humble English words as "and", "or", "if", and "not". We can introduce these operations to Lestrade, and explain how to construct proofs of propositions built with these operations or use evidence for propositions built with these operations in building proofs of other propositions.

```
declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


construct & p q : prop

>> &: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}
```

We introduce propositional variables `p` and `q` (notice the use of the `declare` command instead of the `construct` command), and use these to declare the primitive operation of *conjunction* (and). When we are talking to the Reader, we may say "p and q" or write in symbols $p \wedge q$ instead of using

the Lestrade notation `p & q`. Lestrade requires that an operation being constructed or defined appear before its arguments in the command introducing it, but Lestrade does support infix notation for operations with two arguments[3] as we will see in what follows. It does not support much order of operations: one-argument operators bind more tightly, but infixes all have the same precedence and group to the right, as in the old computer language APL.

Of course at this point all we know about the operation is that it takes two propositions as input and outputs a proposition. We need to be convinced that we really are talking about "and".

```
declare pp that p

>> pp: that p {move 1}


declare qq that q

>> qq: that q {move 1}
```

We introduce variables of new sorts, a piece of evidence `pp` for the truth of p and a piece of evidence `qq` for the proof of q.

```
construct Conjunction pp qq : that p & q

>> Conjunction: [(.p_1:prop),(pp_1:that .p_1),(.q_1:prop),
>>         (qq_1:that .q_1) => (---:that (.p_1 & .q_1))]
>>   {move 0}
```

We assert the existence of a function which takes evidence `pp` for p and evidence `qq` for q to evidence for $p \wedge q$. This implements a fact that we know about the logic of "and": we may really be getting somewhere here!

[3]when the first argument is not a function; Lestrade will display values of a two argument function whose first argument is not a function using infix notation. The user may enter terms in mixfix notation (`x1 f x2 ... xn` instead of `f x1 x2 ... xn`), for $n > 2$; Lestrade can parse such mixfix expressions but will never display them.

```
declare rr that p & q

>> rr: that (p & q) {move 1}


construct Simplification1 rr : that p

>> Simplification1: [(.p_1:prop),(.q_1:prop),(rr_1:that
>>        (.p_1 & .q_1)) => (---:that .p_1)]
>>   {move 0}


construct Simplification2 rr : that q

>> Simplification2: [(.p_1:prop),(.q_1:prop),(rr_1:that
>>        (.p_1 & .q_1)) => (---:that .q_1)]
>>   {move 0}
```

Now we introduce a variable `rr` which is evidence for $p \wedge q$ and provide a function (the first rule of simplification) which takes `rr` to evidence for p and a function (the second rule of simplification) which takes `rr` to evidence for q.

You will notice if you read the Lestrade dialogue carefully that each of these functions has `p` and `q` as additional hidden arguments. The reason that we do not have to supply them explicitly is that their values can be deduced from the sorts of the explicitly given arguments.

```
define Conjcomm rr : Conjunction (Simplification2 rr, Simplification1 rr)

>> Conjcomm: [(.p_1:prop),(.q_1:prop),(rr_1:
>>        that (.p_1 & .q_1)) => ((Simplification2(rr_1)
>>        Conjunction Simplification1(rr_1)):that
>>        (.q_1 & .p_1))]
>>   {move 0}
```

We illustrate the construction of a derived rule. From input `rr`, evidence for $p \wedge q$, we extract evidence for q by simplification (2) and evidence for p by simplification (1), which we put together using conjunction into evidence for $q \wedge p$. We appear to have shown the theorem that if $p \wedge q$, then $q \wedge p$, though we do not yet have a representation of if...then....

Notice that the hidden arguments of the conjunction and simplification functions which we did not have to write in our instructions to Lestrade are not shown in its description of the type of `Conjcomm`. Notice that the function `Conjunction` is treated as an infix operator because it has two explicit arguments.

5 The logic of implication

In this section, we introduce propositions of the form "if p then q", in symbolic form $p \to q$.

```
construct -> p q : prop

>> ->: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}


declare ss that p -> q

>> ss: that (p -> q) {move 1}


construct Mp pp ss : that q

>> Mp: [(.p_1:prop),(pp_1:that .p_1),(.q_1:prop),(ss_1:
>>        that (.p_1 -> .q_1)) => (---:that .q_1)]
>>   {move 0}
```

We inform the good Inspector that if we have evidence `pp` for p and evidence `ss` for $p \to q$, we can construct evidence for $p \to q$. This rule for using an assumption which is an implication is not so different from the rules of conjunction above, except for detail (it is after all a different logical operation). `Mp` abbreviates the usual Latin name *modus ponens* for this rule.

The next rule is quite different, revealing additional capabilities of the Inspector.

```
open

     declare pp1 that p

>>       pp1: that p {move 2}


     construct Ded pp1 : that q

>>       Ded: [(pp1_1:that p) => (---:that q)]
>>         {move 1}


     close

construct Deduction Ded : that p -> q

>> Deduction: [(.p_1:prop),(.q_1:prop),(Ded_1:[(pp1_2:
>>              that .p_1) => (---:that .q_1)])
>>         => (---:that (.p_1 -> .q_1))]
>>   {move 0}
```

Here we make use of the more sophisticated machinery of function definitions to introduce a variable function `Ded` from evidence for p to evidence for q and define a function `Deduction` which takes such a function `Ded` to a proof of $p \to q$.

And this makes sense: if we have a way to construct a proof of q given a proof of p, we have proved $p \to q$.

```
open

     declare cc1 that p & q

>>       cc1: that (p & q) {move 2}


     define conjproof cc1 : Conjcomm cc1

>>       conjproof: [(cc1_1:that (p & q)) =>
>>              (Conjcomm(cc1_1):that (q & p))]
>>         {move 1}


     close

define Conjthm p q : Deduction conjproof

>> Conjthm: [(p_1:prop),(q_1:prop) => (Deduction([(cc1_2:
>>              that (p_1 & q_1)) => (Conjcomm(cc1_2):
>>              that (q_1 & p_1))])
>>         :that ((p_1 & q_1) -> (q_1 & p_1)))]
>>    {move 0}
```

Here we show how to use the `Deduction` construction of proofs of implications. Recall at the end of the previous section that we thought we had proved that $p \wedge q$ implies $q \wedge p$: here we show that we can in fact prove this. `Conjthm` is a function which takes propositions p and q to a proof of $p \wedge q \rightarrow q \wedge p$.

There is a subtle point which might be useful to bring up here: notice that `conjproof` really is needed. The object `cc1` is evidence for p, and `Conjcomm cc1` is evidence for q, but `Conjcomm` is not a function taking evidence for p to evidence for q: `Conjcomm` has two additional hidden arguments p, q so is a function of three variables of a rather complex type, not a function of one

variable of the type needed.

It might also be a useful exercise to read the sort information of `Conjthm` and identify the expanded version of `conjproof`.

6 An extensive proof using conjunction and implication alone

We give an extensive proof written (more or less) in English using the same formal rules we have implemented in Lestrade, then convert it to a definition of a proof object in Lestrade.

Main Goal: Prove $((P \wedge Q) \rightarrow R) \leftrightarrow (P \rightarrow (Q \rightarrow R))$

This statement is a biconditional, so its proof will have two parts.

Part I:

Assume (1): $((P \wedge Q) \rightarrow R)$

Goal (of part I): $(P \rightarrow (Q \rightarrow R))$

The form of the goal tells us what to do!

Assume (2): P

Goal: $Q \rightarrow R$

and again!

Assume (3): Q

Goal: R

Now we have "unpacked" everything...we should pause and take stock of what we have. Line (1), $((P \wedge Q) \rightarrow R)$, would give us our goal R...if we had $P \wedge Q$. And we can have this.

(4): $P \wedge Q$ by conjunction, lines 2,3

(5): R by modus ponens, lines 1,4 [I do not care which order the line numbers are given in]

(6): $Q \rightarrow R$ by deduction lines 3-5. [this is by the entire block of lines, I do mean a hyphen not a comma]

(7): $(P \rightarrow (Q \rightarrow R))$ by deduction lines 2-6.

This completes our work for Part I. We could state an implication as proved here, but we do not have to. Part II is on the next page.

Part II:

Assume (8): $(P \to (Q \to R))$

note that while I numbered this line 8, to avoid conflict with the part of the proof already given, I in fact have no lines available to me but line 8 for my argument at this point: the argument of Part I is over, and everything in lines 1-7 depends on the assumption in line 1 which we are no longer making here.

Goal (of part II): $((P \wedge Q) \to R)$

The form of the goal tells us what to do!

Assume (9): $P \wedge Q$

Goal: R

We are now as unpacked as we can get, so we have to look at our resources. If we had P, we could apply modus ponens with line 8 and get $Q \to R$. And we can have P...

(10): P simplification line 9

(11): $Q \to R$ m.p. lines 9,10 (m.p. is an allowed abbreviation for "modus ponens".)
Since we have $Q \to R$, it is natural to think about whether we have Q. And we do...

(12): Q simplification line 9 [we could have unpacked this at the same time we unpacked P above: the line order here is a little flexible]

(13): R by m.p. lines 11,12.

(14): $((P \wedge Q) \to R)$ by deduction lines 9-13.

This completes the argument for Part II.

(15): The Main Goal to be proved follows by biconditional introduction, lines 1-7, 8-14.

I could say "$((P \wedge Q) \to R) \leftrightarrow (P \to (Q \to R))$ follows by biconditional introduction, lines 1-7, 8-14", but this is a situation where it is natural (since the statement of the Main Goal is long) to refer to it by name.

The supporting Lestrade text follows.

```
define <-> p q : (p -> q) & q -> p

>> <->: [(p_1:prop),(q_1:prop) => (((p_1 ->
>>        q_1) & (q_1 -> p_1)):prop)]
>>   {move 0}
```

We define the biconditional, exactly as one would expect.

```
declare uu that p->q

>> uu: that (p -> q) {move 1}


declare vv that q->p

>> vv: that (q -> p) {move 1}


define fixprop p pp : pp

>> fixprop: [(p_1:prop),(pp_1:that p_1) => (pp_1:
>>        that p_1)]
>>   {move 0}
```

The function `fixprop` is a technical device to prevent Lestrade from giving a definitionally equivalent but undesired form to a proposition for which we have evidence in stored and displayed sort information. Lestrade does know that things equivalent by application of definitions are the same.

```
define Biconditional uu vv : fixprop (p<->q,Conjunction uu vv)

>> Biconditional: [(.p_1:prop),(.q_1:prop),(uu_1:
>>        that (.p_1 -> .q_1)),(vv_1:that (.q_1
>>        -> .p_1)) => (((.p_1 <-> .q_1) fixprop
>>        Conjunction((.p_1 -> .q_1),uu_1,(.q_1
>>        -> .p_1),vv_1)):that (.p_1 <-> .q_1))]
```

```
>>   {move 0}
```

We have defined the biconditional and given a rule generating a proof of a biconditional from the proofs of its supporting implications.

```
declare r prop

>> r: prop {move 1}

declare r prop

>> r: prop {move 1}

open

     declare line1 that (p & q) -> r

>>      line1: that ((p & q) -> r) {move 2}


     open

          declare line2 that p

>>           line2: that p {move 3}


          open

               declare line3 that q

>>                line3: that q {move 4}


               define line4 line3 : Conjunction line2 line3

>>                line4: [(line3_1:that q) =>
>>                       ((line2 Conjunction line3_1):
```

```
>>                      that (p & q))]
>>                  {move 3}


              define line5 line3 : Mp (line4 line3, line1)

>>                line5: [(line3_1:that q) =>
>>                       ((line4(line3_1) Mp line1):
>>                       that r)]
>>                  {move 3}


              close

          define line6 line2 : Deduction line5

>>             line6: [(line2_1:that p) => (Deduction([(line3_2:
>>                       that q) => (((line2_1
>>                       Conjunction line3_2)
>>                       Mp line1):that r)])
>>                  :that (q -> r))]
>>              {move 2}


          close

     define line7 line1 :  Deduction line6

>>       line7: [(line1_1:that ((p & q) -> r))
>>              => (Deduction([(line2_2:that p)
>>                   => (Deduction([(line3_3:that
>>                       q) => (((line2_2 Conjunction
>>                       line3_3) Mp line1_1):
>>                       that r)])
>>                  :that (q -> r))])
>>              :that (p -> (q -> r)))]
>>         {move 1}


     close
```

```
define Part1 p q r:  Deduction line7

>> Part1: [(p_1:prop),(q_1:prop),(r_1:prop)
>>        => (Deduction([(line1_2:that ((p_1 &
>>             q_1) -> r_1)) => (Deduction([(line2_3:
>>                  that p_1) => (Deduction([(line3_4:
>>                       that q_1) => (((line2_3
>>                       Conjunction line3_4)
>>                       Mp line1_2):that r_1)])
>>                  :that (q_1 -> r_1))])
>>             :that (p_1 -> (q_1 -> r_1)))])
>>        :that (((p_1 & q_1) -> r_1) -> (p_1
>>        -> (q_1 -> r_1))))]
>>   {move 0}


open

     declare line8 that p -> (q -> r)

>>      line8: that (p -> (q -> r)) {move 2}


     open

          declare line9 that p&q

>>           line9: that (p & q) {move 3}


          define line10 line9 : Simplification1 line9

>>           line10: [(line9_1:that (p & q))
>>                  => (Simplification1(line9_1):
>>                  that p)]
>>             {move 2}


          define line11 line9 :  Mp (line10 line9, line8)
```

```
>>           line11: [(line9_1:that (p & q))
>>                  => ((line10(line9_1) Mp line8):
>>                  that (q -> r))]
>>             {move 2}


          define line12 line9 :  Simplification2 line9

>>           line12: [(line9_1:that (p & q))
>>                  => (Simplification2(line9_1):
>>                  that q)]
>>             {move 2}


          define line13 line9 :  Mp (line12 line9, line11 line9)

>>           line13: [(line9_1:that (p & q))
>>                  => ((line12(line9_1) Mp line11(line9_1)):
>>                  that r)]
>>             {move 2}


          close

     define line14 line8 :  Deduction line13

>>      line14: [(line8_1:that (p -> (q -> r)))
>>             => (Deduction([(line9_2:that (p
>>                  & q)) => ((Simplification2(line9_2)
>>                  Mp (Simplification1(line9_2)
>>                  Mp line8_1)):that r)])
>>             :that ((p & q) -> r))]
>>        {move 1}


     close

define Part2 p q r :  Deduction line14
```

```
>> Part2: [(p_1:prop),(q_1:prop),(r_1:prop)
>>        => (Deduction([(line8_2:that (p_1 ->
>>             (q_1 -> r_1))) => (Deduction([(line9_3:
>>                  that (p_1 & q_1)) => ((Simplification2(line9_3)
>>                  Mp (Simplification1(line9_3)
>>                  Mp line8_2)):that r_1)])
>>             :that ((p_1 & q_1) -> r_1))])
>>        :that ((p_1 -> (q_1 -> r_1)) -> ((p_1
>>        & q_1) -> r_1)))]
>>   {move 0}


define Export p q r : Biconditional (Part1 p q r,Part2 p q r)

>> Export: [(p_1:prop),(q_1:prop),(r_1:prop)
>>        => ((Part1(p_1,q_1,r_1) Biconditional
>>        Part2(p_1,q_1,r_1)):that (((p_1 & q_1)
>>        -> r_1) <-> (p_1 -> (q_1 -> r_1))))]
>>   {move 0}
```

The Lestrade calculation implementing the proof above is formally very close to the paper version in structure. This is emphasized here by giving names to declared items corresponding exactly to the lines and parts in the proof above. What is present in the paper proof and absent in the Lestrade proof is the statement of goals: and this could be supplied by suitably commenting the Lestrade text, which one would very likely do if developing the proof incrementally in a Lestrade session.

Notice that the Lestrade device of moves neatly implements what happens to the structure of an argument when one introduces a hypothesis for the sake of argument.

7 The logic of negation, contrapositives, and disjunction

In this section we introduce the sentence construction "It is not the case that p", symbolically $\neg p$. It is useful to introduce a primitive false statement, which Lestrade calls `??`, but which we will typeset as $\bot$, familiarly known as "the absurd". We continue with exploration of the interaction of implication and negation, and define the last major logical operation of disjunction (and/or) in terms of implication and negation, and develop its basic rules.

```
construct ?? prop

>> ??: prop {move 0}
```

```
define ~ p : p -> ??

>> ~: [(p_1:prop) => ((p_1 -> ??):prop)]
>>   {move 0}
```

We declare the absurd and use it to define $\neg p$ as $p \rightarrow \bot$.

```
declare maybe that ~ ~ p

>> maybe: that ~(~(p)) {move 1}

construct Dneg maybe that p

>> Dneg: [(.p_1:prop),(maybe_1:that ~(~(.p_1))) => (---:
>>        that .p_1)]
>>   {move 0}
```

We introduce the rule of double negation, by declaring a primitive function sending evidence for $\neg\neg p$ to evidence for p.

```
open

     declare pp1 that p

>>       pp1: that p {move 2}


     construct Negded pp1 that ??

>>       Negded: [(pp1_1:that p) => (---:that
>>               ??)]
>>         {move 1}


     close

define Negproof Negded :  fixprop (~p,Deduction Negded)

>> Negproof: [(.p_1:prop),(Negded_1:[(pp1_2:
>>              that .p_1) => (---:that ??)])
>>         => ((~(.p_1) fixprop Deduction(Negded_1)):
>>         that ~(.p_1))]
>>   {move 0}
```

Here we develop the standard strategy for proving a negative statement $\neg p$: if we have a function taking proofs of p to proofs of $\bot$, we can get a proof of $\neg p$. The function `fixprop` is used here to tell Lestrade that the output of `Negproof` is to be recorded as evidence for $\neg p$ rather than evidence for $p \rightarrow \bot$ (at bottom Lestrade does recognize that these are the same thing). The Lestrade type checker does recognize terms as being the same which are equivalent by definition in a suitable sense.

It is also worth noting here that Lestrade views every two argument function whose first argument is not a function as an infix operator by default.

`fixprop` is thus rather surprisingly used as an infix. Of course we have been seeing this all along with the binary propositional operators, and with operations like addition and multiplication on natural numbers.

```
define v p q : ~p -> q

>> v: [(p_1:prop),(q_1:prop) => ((~(p_1) -> q_1):prop)]
>>   {move 0}
```

We define disjunction in terms of implication and negation.

```
declare indev that ~q -> ~p

>> indev: that (~(q) -> ~(p)) {move 1}

open

     declare pp1 that p

>>       pp1: that p {move 2}


     open

          declare notq that ~q

>>           notq: that ~(q) {move 3}


          define line1 notq : Mp notq indev

>>           line1: [(notq_1:that ~(q)) => ((notq_1
>>                  Mp indev):that ~(p))]
>>             {move 2}
```

```
          define line2 notq : Mp (pp1, line1 notq)

>>           line2: [(notq_1:that ~(q)) => ((pp1
>>                  Mp line1(notq_1)):that ??)]
>>             {move 2}


          close

     define qq1 pp1 : Negproof line2

>>      qq1: [(pp1_1:that p) => (Negproof([(notq_2:
>>                  that ~(q)) => ((pp1_1 Mp (notq_2
>>                  Mp indev)):that ??)])
>>              :that ~(~(q)))]
>>        {move 1}


     define qq2 pp1:  Dneg qq1 pp1

>>      qq2: [(pp1_1:that p) => (Dneg(qq1(pp1_1)):
>>             that q)]
>>        {move 1}


     close

define Contrapositive indev : Deduction qq2

>> Contrapositive: [(.q_1:prop),(.p_1:prop),
>>        (indev_1:that (~(.q_1) -> ~(.p_1)))
>>        => (Deduction([(pp1_2:that .p_1) =>
>>             (Dneg(Negproof([(notq_3:that ~(.q_1))
>>                  => ((pp1_2 Mp (notq_3 Mp indev_1)):
>>                  that ??)]))
>>             :that .q_1)])
>>        :that (.p_1 -> .q_1))]
```

```
>>   {move 0}

>>   {move 0}
```

We develop a function `Contrapositive` which given a proof of $\neg q \to \neg p$ (as its only explicit argument) returns a proof of $p \to q$.

Reversing the order of development of the last large text proof, we develop a text proof of the rule "from $\neg Q \to \neg P$ deduce $P \to Q$" from the Lestrade text above.

Assume (indev): $\neg Q \to \neg P$

Goal: $P \to Q$

> **Assume (pp1):** P
>
> **Goal:** Q
>
> > **Assume (notq):** $\neg Q$
> >
> > **Goal:** $\bot$
> >
> > **line 1:** $\neg P$ modus ponens (indev, pp1)
> >
> > **line 2:** $\bot$ contradiction (pp1, line 1)
>
> **qq1:** $\neg\neg Q$ negation introduction(notq–line 2)
>
> **qq2:** Q double negation (qq1)

contrapositive: $P \to Q$ deduction (pp1–qq2)

```
open

     declare notp that ~p

>>      notp: that ~(p) {move 2}


     construct excalt1 notp that q
```

```
>>      excalt1: [(notp_1:that ~(p)) => (---:
>>             that q)]
>>        {move 1}


     close

define Orproof1 excalt1 :  fixprop (p v q,Deduction excalt1)

>> Orproof1: [(.p_1:prop),(.q_1:prop),(excalt1_1:
>>        [(notp_2:that ~(.p_1)) => (---:that
>>             .q_1)])
>>        => (((.p_1 v .q_1) fixprop Deduction(excalt1_1)):
>>        that (.p_1 v .q_1))]
>>   {move 0}
```

We develop our basic rule of disjunction introduction: if we can deduce q from $\neg p$, we get $p \vee q$. Of course, there should be a symmetric rule, since disjunction is commutative: to derive it we will need the converse double negation rule and an application of `Contrapositive`.

```
open

     declare notp that ~p

>>      notp: that ~(p) {move 2}


     define dneg2 notp : Mp pp notp

>>      dneg2: [(notp_1:that ~(p)) => ((pp Mp
>>             notp_1):that ??)]
>>        {move 1}


     close
```

```
define Dneg2 pp : fixprop (~ ~p,Deduction dneg2)

>> Dneg2: [(.p_1:prop),(pp_1:that .p_1) => ((~(~(.p_1))
>>        fixprop Deduction([(notp_2:that ~(.p_1))
>>             => ((pp_1 Mp notp_2):that ??)]))
>>        :that ~(~(.p_1)))]
>>   {move 0}
```

We develop the function `Dneg2` which takes a proof of p to a proof of $\neg\neg p$.

```
open

     declare notq that ~q

>>      notq: that ~(q) {move 2}


     construct excalt2 notq that p

>>      excalt2: [(notq_1:that ~(q)) => (---:
>>             that p)]
>>        {move 1}


     define line1 notq : Dneg2 excalt2 notq

>>      line1: [(notq_1:that ~(q)) => (Dneg2(excalt2(notq_1)):
>>             that ~(~(p)))]
>>        {move 1}


     close

define Orproof2 excalt2: fixprop(p v q,Contrapositive (Deduction line1))
```

```
>> Orproof2: [(.q_1:prop),(.p_1:prop),(excalt2_1:
>>        [(notq_2:that ~(.q_1)) => (---:that
>>             .p_1)])
>>        => (((.p_1 v .q_1) fixprop Contrapositive(Deduction([(notq_3:
>>             that ~(.q_1)) => (Dneg2(excalt2_1(notq_3)):
>>             that ~(~(.p_1)))])))
>>        ):that (.p_1 v .q_1))]
>>   {move 0}
```

and we develop the symmetrical disjunction introduction rule.

```
open

     declare notq that ~q

>>      notq: that ~(q) {move 2}


     define ppconst notq : pp

>>      ppconst: [(notq_1:that ~(q)) => (pp:
>>             that p)]
>>        {move 1}


     close

define Addition1 q pp : Orproof2 ppconst

>> Addition1: [(q_1:prop),(.p_1:prop),(pp_1:
>>        that .p_1) => (Orproof2([(notq_2:that
>>             ~(q_1)) => (pp_1:that .p_1)])
>>        :that (.p_1 v q_1))]
>>   {move 0}
```

```
open

     declare notp that ~p

>>       notp: that ~(p) {move 2}


     define qqconst notp : qq

>>       qqconst: [(notp_1:that ~(p)) => (qq:
>>              that q)]
>>         {move 1}


     close

define Addition2 p qq : Orproof1 qqconst

>> Addition2: [(p_1:prop),(.q_1:prop),(qq_1:
>>         that .q_1) => (Orproof1([(notp_2:that
>>              ~(p_1)) => (qq_1:that .q_1)])
>>         :that (p_1 v .q_1))]
>>    {move 0}
```

We present the rules of addition (the constructive disjunction introduction rules).

```
open

     declare notq that ~q

>>       notq: that ~(q) {move 2}
```

```
     open

          declare pp1 that p

>>           pp1: that p {move 3}


          define line3 pp1 :  Mp pp1 ss

>>           line3: [(pp1_1:that p) => ((pp1_1
>>                  Mp ss):that q)]
>>             {move 2}


          define line4 pp1 : Mp (line3 pp1,notq)

>>           line4: [(pp1_1:that p) => ((line3(pp1_1)
>>                  Mp notq):that ??)]
>>             {move 2}


          close

     define pcantbe notq : Negproof line4

>>      pcantbe: [(notq_1:that ~(q)) => (Negproof([(pp1_2:
>>                  that p) => (((pp1_2 Mp ss)
>>                  Mp notq_1):that ??)])
>>             :that ~(p))]
>>        {move 1}


     close

define Contrapositive2 ss : Deduction pcantbe
```

```
>> Contrapositive2: [(.p_1:prop),(.q_1:prop),
>>        (ss_1:that (.p_1 -> .q_1)) => (Deduction([(notq_2:
>>             that ~(.q_1)) => (Negproof([(pp1_3:
>>                  that .p_1) => (((pp1_3 Mp
>>                  ss_1) Mp notq_2):that ??)])
>>             :that ~(.p_1))])
>>        :that (~(.q_1) -> ~(.p_1)))]
>>   {move 0}
```

We present the converse contrapositive rule, taking evidence for $p \to q$ to evidence for $\neg q \to \neg p$. We suggest that the Reader write her own text proof of the rule "from $P \to Q$, deduce $\neg Q \to \neg P$" from the Lestrade text above, using our development for the other contrapositive rule as a model.

```
declare ss2 that q -> r

>> ss2: that (q -> r) {move 1}


open

     declare pp1 that p

>>      pp1: that p {move 2}


     define line5 pp1 : Mp pp1 ss

>>      line5: [(pp1_1:that p) => ((pp1_1 Mp
>>             ss):that q)]
>>        {move 1}


     define line6 pp1 : Mp (line5 pp1,ss2)
```

```
>>      line6: [(pp1_1:that p) => ((line5(pp1_1)
>>             Mp ss2):that r)]
>>        {move 1}


     close

define Transimp ss ss2:  Deduction line6

>> Transimp: [(.p_1:prop),(.q_1:prop),(ss_1:
>>        that (.p_1 -> .q_1)),(.r_1:prop),(ss2_1:
>>        that (.q_1 -> .r_1)) => (Deduction([(pp1_2:
>>             that .p_1) => (((pp1_2 Mp ss_1)
>>             Mp ss2_1):that .r_1)])
>>        :that (.p_1 -> .r_1))]
>>   {move 0}
```

We present the rule for transitivity of implication: given evidence for $p \to q$ and $q \to r$, get evidence for $p \to r$.

```
declare alts that p v q

>> alts: that (p v q) {move 1}


declare case1 that p -> r

>> case1: that (p -> r) {move 1}


declare case2 that q -> r

>> case2: that (q -> r) {move 1}
```

```
open

     declare notr that ~r

>>      notr: that ~(r) {move 2}


     define linea7 :  Contrapositive2 case1

>>      linea7: [(Contrapositive2(case1):that
>>             (~(r) -> ~(p)))]
>>        {move 1}


     define line8 notr : Mp notr linea7

>>      line8: [(notr_1:that ~(r)) => ((notr_1
>>             Mp linea7):that ~(p))]
>>        {move 1}


     define line9 notr : Mp (line8 notr, alts)

>>      line9: [(notr_1:that ~(r)) => ((line8(notr_1)
>>             Mp alts):that q)]
>>        {move 1}


     define line10 notr : Mp (line9 notr, case2)

>>      line10: [(notr_1:that ~(r)) => ((line9(notr_1)
>>             Mp case2):that r)]
>>        {move 1}


     define line11 notr : Mp (line10 notr, notr)
```

```
>>      line11: [(notr_1:that ~(r)) => ((line10(notr_1)
>>             Mp notr_1):that ??)]
>>        {move 1}


     close

define Cases alts case1 case2 : Dneg (Negproof line11)

>> Cases: [(.p_1:prop),(.q_1:prop),(alts_1:that
>>        (.p_1 v .q_1)),(.r_1:prop),(case1_1:
>>        that (.p_1 -> .r_1)),(case2_1:that (.q_1
>>        -> .r_1)) => (Dneg(Negproof([(notr_2:
>>             that ~(.r_1)) => (((((notr_2 Mp
>>             Contrapositive2(case1_1)) Mp alts_1)
>>             Mp case2_1) Mp notr_2):that ??)]))
>>        :that .r_1)]
>>   {move 0}
```

We derive the rule of proof by cases: if we have evidence for $p \vee q$, evidence for $p \to r$ and evidence for $q \to r$, we get evidence for r. This is the constructive rule of disjunction elimination (even though we proved it classically).

```
declare ppx that ~p

>> ppx: that ~(p) {move 1}


define Ds1 alts ppx : Mp ppx alts

>> Ds1: [(.p_1:prop),(.q_1:prop),(alts_1:that
>>        (.p_1 v .q_1)),(ppx_1:that ~(.p_1))
```

```
>>        => ((ppx_1 Mp alts_1):that .q_1)]
>>   {move 0}


declare qqx that ~q

>> qqx: that ~(q) {move 1}


define Ds2 alts qqx : Dneg (Mp (qqx,Contrapositive2 alts))

>> Ds2: [(.p_1:prop),(.q_1:prop),(alts_1:that
>>        (.p_1 v .q_1)),(qqx_1:that ~(.q_1))
>>        => (Dneg((qqx_1 Mp Contrapositive2(alts_1))):
>>        that .p_1)]
>>   {move 0}
```

We develop the rules of disjunctive syllogism, additional classical disjunction elimination rules. That is enough propositional logic to illustrate that we have enough power to prove what we need to in propositional logic, and to give an indication of how we go about it.

All of the rules above are good bases for text proof writing exercises.

8 Equality

We need some atomic statements to link with our propositional connectives. We'll start with equations.

```
declare tau type

>> tau: type {move 1}
```

```
declare x in tau

>> x: in tau {move 1}
```

```
declare y in tau

>> y: in tau {move 1}
```

```
construct = x y : prop

>> =: [(.tau_1:type),(x_1:in .tau_1),(y_1:in .tau_1) =>
>>        (---:prop)]
>>   {move 0}
```

We declare the equality relation. Note that it has a hidden third parameter, the type of the objects asserted to be equal.

```
construct Refleq x that x=x

>> Refleq: [(.tau_1:type),(x_1:in .tau_1) =>
>>        (---:that (x_1 = x_1))]
>>   {move 0}
```

This is the first of the two axiomatic properties of equality: everything of whatever type is equal to itself.

```
open

     declare x1 in tau

>>       x1: in tau {move 2}


     construct Pred x1 : prop

>>       Pred: [(x1_1:in tau) => (---:prop)]
>>         {move 1}


     close

declare eqev that x=y

>> eqev: that (x = y) {move 1}


declare subsev that Pred x

>> subsev: that Pred(x) {move 1}


construct Substitution Pred, eqev subsev : that Pred y

>> Substitution: [(.tau_1:type),(Pred_1:[(x1_2:
>>              in .tau_1) => (---:prop)]),
>>         (.x_1:in .tau_1),(.y_1:in .tau_1),(eqev_1:
>>         that (.x_1 = .y_1)),(subsev_1:that Pred_1(.x_1))
>>         => (---:that Pred_1(.y_1))]
>>    {move 0}
```

This is the rule of substitution of equals for equals: if $P(x)$ is true and $x = y$ then $P(y)$ is true.

Commas between arguments in Lestrade parameter lists are usually optional, but after a function which appears as an argument (and before it if it has two or more arguments and the first is not a function) we may need to place a comma so that it is not misinterpreted as a function applied in either prefix or infix/mixfix position.

```
open

     declare y1 in tau

>>       y1: in tau {move 2}


     define sympred y1 : y1 = x

>>       sympred: [(y1_1:in tau) => ((y1_1 =
>>              x):prop)]
>>         {move 1}


     close

define Symmeq eqev:Substitution(sympred,eqev,Refleq x)

>> Symmeq: [(.tau_1:type),(.x_1:in .tau_1),(.y_1:
>>         in .tau_1),(eqev_1:that (.x_1 = .y_1))
>>         => (Substitution([(y1_2:in .tau_1) =>
>>              ((y1_2 = .x_1):prop)]
>>         ,eqev_1,Refleq(.x_1)):that (.y_1 = .x_1))]
>>   {move 0}
```

Here is the proof of the symmetry rule for equality: from evidence for $x = y$ get evidence for $y = x$ (by substitution for the leftmost occurrence of x in the tautology $x = x$)

Here is a text proof of the symmetry property.

Assume (1): $x = y$

Goal: $y = x$

Define (2) $P(z)$ as $z = x$.

(3): $x = x$ reflexivity of equality.

(4): $P(x)$ by 2,3

(5): $P(y)$ by substitution using (1) into (4)

(6): $y = x$ by (5), (2)

```
declare z in tau

>> z: in tau {move 1}


declare eqev2 that y=z

>> eqev2: that (y = z) {move 1}


open

     declare x1 in tau

>>       x1: in tau {move 2}


     define transpred x1 : x1 = z

>>       transpred: [(x1_1:in tau) => ((x1_1
```

```
>>              = z):prop)]
>>        {move 1}


     close

define Transeq eqev eqev2 :  Substitution(transpred,Symmeq eqev,eqev2)

>> Transeq: [(.tau_1:type),(.x_1:in .tau_1),
>>        (.y_1:in .tau_1),(eqev_1:that (.x_1
>>        = .y_1)),(.z_1:in .tau_1),(eqev2_1:that
>>        (.y_1 = .z_1)) => (Substitution([(x1_2:
>>             in .tau_1) => ((x1_2 = .z_1):prop)]
>>        ,Symmeq(eqev_1),eqev2_1):that (.x_1
>>        = .z_1))]
>>   {move 0}
```

Here is a proof of the transitive property of equality.

We recommend as an exercise writing your own text proof of this theorem using the text proof of symmetry of equality above as a model.

9 The logic of quantifiers

In this section we introduce the universal and existential quantifiers.

```
construct Forall Pred : prop

>> Forall: [(.tau_1:type),(Pred_1:[(x1_2:in
>>              .tau_1) => (---:prop)])
>>        => (---:prop)]
>>   {move 0}
```

We define the universal quantifier as an operation on functions from type `tau` to propositions (i.e, predicates of type `tau` objects).

```
open

     declare x1 in tau

>>       x1: in tau {move 2}


     define Notpred x1 : ~ Pred x1

>>       Notpred: [(x1_1:in tau) => (~(Pred(x1_1)):
>>              prop)]
>>         {move 1}


     close

define Exists Pred: ~ Forall Notpred

>> Exists: [(.tau_1:type),(Pred_1:[(x1_2:in
>>              .tau_1) => (---:prop)])
>>         => (~(Forall([(x1_3:in .tau_1) => (~(Pred_1(x1_3)):
>>              prop)]))
>>         :prop)]
>>    {move 0}
```

We define the existential quantifier $(\exists x : P(x))$ as $\neg(\forall x : \neg P(x)\}$. Notice that we have to assign a name to $(x \to \neg P(x))$ $[\neg P(x)$ as a predicate of $x]$.

```
declare univev that Forall Pred

>> univev: that Forall(Pred) {move 1}


declare u in tau
```

```
>> u: in tau {move 1}


construct Ui univev, u : that Pred u

>> Ui: [(.tau_1:type),(.Pred_1:[(x1_2:in .tau_1)
>>             => (---:prop)]),
>>        (univev_1:that Forall(.Pred_1)),(u_1:
>>        in .tau_1) => (---:that .Pred_1(u_1))]
>>   {move 0}
```

We introduce the rule of universal instantiation. From $(\forall x \in \tau : P(x))$ deduce $P(a)$ for any specific $a \in \tau$.[4]

```
open

open

     declare x1 in tau

>>      x1: in tau {move 2}


     construct univ x1 that Pred x1

>>      univ: [(x1_1:in tau) => (---:that Pred(x1_1))]
>>        {move 1}


     close

construct Ug univ : that Forall Pred

>> Ug: [(.tau_1:type),(.Pred_1:[(x1_2:in .tau_1)
>>             => (---:prop)]),
```

[4]The use of $\in$ here is admittedly an abuse of notation. Types are not sets.

```
>>        (univ_1:[(x1_3:in .tau_1) => (---:that
>>             .Pred_1(x1_3))])
>>        => (---:that Forall(.Pred_1))]
>>   {move 0}
```

We introduce the rule of universal generalization. If I have a general method to prove $P(a)$ for any $a \in \tau$ (a function taking a in type `tau` to a proof of $P(a)$, I postulate a proof for $(\forall x \in \tau : P(x))$.

```
open

     declare x1 in tau

>>      x1: in tau {move 2}


     define selfproof x1 : Refleq x1

>>      selfproof: [(x1_1:in tau) => (Refleq(x1_1):
>>             that (x1_1 = x1_1))]
>>        {move 1}


     close

define Univexample tau: Ug selfproof

>> Univexample: [(tau_1:type) => (Ug([(x1_3:
>>             in tau_1) => (Refleq(x1_3):that
>>             (x1_3 = x1_3))])
>>        :that Forall([(x1_4:in tau_1) => ((x1_4
>>             = x1_4):prop)]))
>>        ]
>>   {move 0}
```

We give an example, proving $(\forall x \in \tau : x = x)$. It is useful to notice here that Lestrade *can* deduce the hidden predicate argument here (the predicate $(x \mapsto x = x)$ which we have not even named).

```
declare xx in tau

>> xx: in tau {move 1}


declare predev that Pred xx

>> predev: that Pred(xx) {move 1}


open

     declare line15 that Forall Notpred

>>      line15: that Forall(Notpred) {move 2}


     define line16 line15 : Ui line15,xx

>>      line16: [(line15_1:that Forall(Notpred))
>>             => ((line15_1 Ui xx):that Notpred(xx))]
>>        {move 1}


     define line17 line15 : Mp predev, line16 line15

>>      line17: [(line15_1:that Forall(Notpred))
>>             => ((predev Mp line16(line15_1)):
>>             that ??)]
>>        {move 1}


     close
```

```
define Ei Pred, predev : fixprop (Exists Pred,Negproof line17)

>> Ei: [(.tau_1:type),(Pred_1:[(x1_2:in .tau_1)
>>             => (---:prop)]),
>>        (.xx_1:in .tau_1),(predev_1:that Pred_1(.xx_1))
>>        => ((Exists(Pred_1) fixprop Negproof([(line15_4:
>>             that Forall([(x1_5:in .tau_1) =>
>>                  (~(Pred_1(x1_5)):prop)]))
>>             => ((predev_1 Mp (line15_4 Ui .xx_1)):
>>             that ??)]))
>>        :that Exists(Pred_1))]
>>   {move 0}
```

We develop the rule of existential instantiation. Given an `xx` of type τ and evidence that `Pred(xx)`, we can conclude $(\exists x \in \tau : Pred(x))$. The argument `xx` is implicit.

```
declare existsev that Exists Pred

>> existsev: that Exists(Pred) {move 1}


declare wgoal prop

>> wgoal: prop {move 1}


open

     declare w1 in tau

>>       w1: in tau {move 2}
```

```
     declare exev that Pred w1

>>      exev: that Pred(w1) {move 2}


     construct wproof exev that wgoal

>>      wproof: [(.w1_1:in tau),(exev_1:that
>>             Pred(.w1_1)) => (---:that wgoal)]
>>        {move 1}


     declare notwgoal that ~ wgoal

>>      notwgoal: that ~(wgoal) {move 2}


     open

          declare w2 in tau

>>           w2: in tau {move 3}


          open

               declare exev2 that Pred w2

>>                exev2: that Pred(w2) {move
>>                  4}


               define line25 exev2 : wproof exev2

>>                line25: [(exev2_1:that Pred(w2))
>>                       => (wproof(exev2_1):that
>>                       wgoal)]
```

```
>>                  {move 3}


               define line26 exev2 : Mp (line25 exev2, notwgoal)

>>                line26: [(exev2_1:that Pred(w2))
>>                       => ((line25(exev2_1)
>>                       Mp notwgoal):that ??)]
>>                  {move 3}


               close

          define line27 w2 : Negproof line26

>>             line27: [(w2_1:in tau) => (Negproof([(exev2_2:
>>                         that Pred(w2_1)) => ((wproof(exev2_2)
>>                         Mp notwgoal):that ??)])
>>                    :that ~(Pred(w2_1)))]
>>               {move 2}


          close

     define line28 notwgoal : Ug line27

>>       line28: [(notwgoal_1:that ~(wgoal))
>>              => (Ug([(w2_3:in tau) => (Negproof([(exev2_4:
>>                         that Pred(w2_3)) => ((wproof(exev2_4)
>>                         Mp notwgoal_1):that ??)])
>>                    :that ~(Pred(w2_3)))])
>>              :that Forall([(w2_5:in tau) =>
>>                    (~(Pred(w2_5)):prop)]))
>>              ]
>>         {move 1}


     define line29 notwgoal : Mp (line28 notwgoal,existsev)
```

```
>>      line29: [(notwgoal_1:that ~(wgoal))
>>             => ((line28(notwgoal_1) Mp existsev):
>>             that ??)]
>>        {move 1}


     close

define Witness existsev wproof : Dneg(Negproof line29)

>> Witness: [(.tau_1:type),(.Pred_1:[(x1_2:in
>>             .tau_1) => (---:prop)]),
>>        (existsev_1:that Exists(.Pred_1)),(.wgoal_1:
>>        prop),(wproof_1:[(.w1_3:in .tau_1),(exev_3:
>>             that .Pred_1(.w1_3)) => (---:that
>>             .wgoal_1)])
>>        => (Dneg(Negproof([(notwgoal_4:that
>>             ~(.wgoal_1)) => ((Ug([(w2_7:in
>>                  .tau_1) => (Negproof([(exev2_8:
>>                       that .Pred_1(w2_7)) =>
>>                       (((w2_7 wproof_1 exev2_8)
>>                       Mp notwgoal_4):that ??)])
>>                  :that ~(.Pred_1(w2_7)))])
>>             Mp existsev_1):that ??)]))
>>        :that .wgoal_1)]
>>   {move 0}
```

We develop the witness introduction rule. If we can show that introduction of a witness `w` and evidence that `Pred(w)` will yield a proof of `wgoal` via application of a suitable function `wproof`, then from evidence `existsev` that $(\exists x : \texttt{Pred}(x))$ and the function `wproof` (in which the goal `wgoal` is implicit) we get a proof of `wgoal`. In other words, a hypothesis that there is a witness to an existential statement may be introduced as soon as the existential statement is postulated to be true.

```
open

     declare w1 in tau

>>      w1: in tau {move 2}


     define Testpred w1 : (Pred w1) & Pred w1

>>      Testpred: [(w1_1:in tau) => ((Pred(w1_1)
>>             & Pred(w1_1)):prop)]
>>        {move 1}


     declare exev that Pred w1

>>      exev: that Pred(w1) {move 2}


     define test exev: Conjunction exev exev

>>      test: [(.w1_1:in tau),(exev_1:that Pred(.w1_1))
>>             => ((exev_1 Conjunction exev_1):
>>             that (Pred(.w1_1) & Pred(.w1_1)))]
>>        {move 1}


     define zorch exev : Ei (Testpred, test exev)

>>      zorch: [(.w1_1:in tau),(exev_1:that
>>             Pred(.w1_1)) => (Ei(Testpred,test(exev_1)):
>>             that Exists(Testpred))]
>>        {move 1}


     close
```

```
define Witnesstest existsev : Witness(existsev,zorch)

>> Witnesstest: [(.tau_1:type),(.Pred_1:[(x1_2:
>>             in .tau_1) => (---:prop)]),
>>        (existsev_1:that Exists(.Pred_1)) =>
>>        ((existsev_1 Witness [(.w1_4:in .tau_1),
>>             (exev_4:that .Pred_1(.w1_4)) =>
>>             (Ei([(w1_5:in .tau_1) => ((.Pred_1(w1_5)
>>                  & .Pred_1(w1_5)):prop)]
>>             ,(exev_4 Conjunction exev_4)):that
>>             Exists([(w1_6:in .tau_1) => ((.Pred_1(w1_6)
>>                  & .Pred_1(w1_6)):prop)]))
>>             ])
>>        :that Exists([(w1_7:in .tau_1) => ((.Pred_1(w1_7)
>>             & .Pred_1(w1_7)):prop)]))
>>        ]
>>   {move 0}
```

We give a simple example of use of the existential rules. We show that if $(\exists x : P(x))$, then $(\exists x : P(x) \wedge P(x))$.

10 More about the natural numbers: induction and recursion principles

```
open

     declare n1 in Nat

>>      n1: in Nat {move 2}


     construct Predn n1 : prop

>>      Predn: [(n1_1:in Nat) => (---:prop)]
```

```
>>        {move 1}


     close

open

     declare n1 in Nat

>>      n1: in Nat {move 2}


     declare indhyp that Predn n1

>>      indhyp: that Predn(n1) {move 2}


     construct indstep indhyp that Predn Succ n1

>>      indstep: [(.n1_1:in Nat),(indhyp_1:that
>>             Predn(.n1_1)) => (---:that Predn(Succ(.n1_1)))]
>>        {move 1}


     close

declare basis that Predn 1

>> basis: that Predn(1) {move 1}


declare k in Nat

>> k: in Nat {move 1}


construct Induction indstep, basis, k : that Predn k
```

```
>> Induction: [(.Predn_1:[(n1_2:in Nat) => (---:
>>             prop)]),
>>        (indstep_1:[(.n1_3:in Nat),(indhyp_3:
>>             that .Predn_1(.n1_3)) => (---:that
>>             .Predn_1(Succ(.n1_3)))]),
>>        (basis_1:that .Predn_1(1)),(k_1:in Nat)
>>        => (---:that .Predn_1(k_1))]
>>   {move 0}
```

We begin by building the familiar principle of mathematical induction. The function `Induction` takes as arguments a function taking proofs of `Pred`(k) for arbitrary k to proofs of `Pred`$(k+1)$ and a proof of `Pred`(1) and a natural number K and returns a proof of `Pred`(K). This is quite deliberately phrased in a way which doesn't involve any quantifiers, but a version with the usual quantifiers could easily be derived. [5]

```
open

     declare n1 in Nat

>>      n1: in Nat {move 2}


     construct typefun n1 : type

>>      typefun: [(n1_1:in Nat) => (---:type)]
>>        {move 1}


     close

open
```

[5]The induction step is given as an argument before the basis step for a technical reason: Lestrade can deduce the implicit argument `Pred` from the induction step term but not from the basis step term.

```
     declare n1 in Nat

>>       n1: in Nat {move 2}


     declare m1 in typefun n1

>>       m1: in typefun(n1) {move 2}


     construct repeatfun n1 m1 in typefun Succ n1

>>       repeatfun: [(n1_1:in Nat),(m1_1:in typefun(n1_1))
>>              => (---:in typefun(Succ(n1_1)))]
>>         {move 1}


     close

declare initval in typefun 1

>> initval: in typefun(1) {move 1}


declare K in Nat

>> K: in Nat {move 1}


construct Iter repeatfun, initval, K : in typefun K

>> Iter: [(.typefun_1:[(n1_2:in Nat) => (---:
>>             type)]),
>>        (repeatfun_1:[(n1_3:in Nat),(m1_3:in
>>             .typefun_1(n1_3)) => (---:in .typefun_1(Succ(n1_3)))]),
>>        (initval_1:in .typefun_1(1)),(K_1:in
>>        Nat) => (---:in .typefun_1(K_1))]
```

```
>>   {move 0}
```

```
construct Iterinit repeatfun, initval :
    that (Iter repeatfun, initval, 1) = initval

>> Iterinit: [(.typefun_1:[(n1_2:in Nat) =>
>>             (---:type)]),
>>        (repeatfun_1:[(n1_3:in Nat),(m1_3:in
>>             .typefun_1(n1_3)) => (---:in .typefun_1(Succ(n1_3)))]),
>>        (initval_1:in .typefun_1(1)) => (---:
>>        that (Iter(repeatfun_1,initval_1,1)
>>        = initval_1))]
>>   {move 0}
```

```
construct Iternext repeatfun, initval, K :
    that (Iter repeatfun, initval, Succ K)
    = repeatfun K (Iter repeatfun, initval K)

>> Iternext: [(.typefun_1:[(n1_2:in Nat) =>
>>             (---:type)]),
>>        (repeatfun_1:[(n1_3:in Nat),(m1_3:in
>>             .typefun_1(n1_3)) => (---:in .typefun_1(Succ(n1_3)))]),
>>        (initval_1:in .typefun_1(1)),(K_1:in
>>        Nat) => (---:that (Iter(repeatfun_1,
>>        initval_1,Succ(K_1)) = (K_1 repeatfun_1
>>        Iter(repeatfun_1,initval_1,K_1))))]
>>   {move 0}
```

In this block, we develop the ability to define functions by recursion (as a primitive, though with some additional postulates it could be derived from induction). The function `Iter` is precisely analogous to `Induction` in its form, replacing `prop` with `type` and `that` with `in`. The functions `Iterinit` and `Iternext` give more precise information about the function that is defined.

In the simple case where `typefun(n)` is a constant type `tau` and `repeatfun` ignores its initial natural number argument, `repeatfun` can be thought of as a function from type `tau` to type `tau` and `Iter(repeatfun,iterval,n)` will be the result of applying `repeatfun` $n-1$ times to `initval` (so it sends 1 to `initval`). The apparently odd definition of iteration is a consequence of starting the natural numbers at 1 instead of 0, which we believe has certain advantages in the construction of the larger number systems.

We will use `Iter` to define addition and multiplication of natural numbers shortly.

It can be noted that we are giving priority to the use of natural numbers to measure how many times an action is performed rather than the use of natural numbers to measure the sizes of sets (which we will also soon define). This is natural in a theory whose basic primitive concept is function rather than set.

```
define oneisone :  Refleq 1

>> oneisone: [(Refleq(1):that (1 = 1))]
>>   {move 0}


open

     declare n1 in Nat

>>      n1: in Nat {move 2}


     open

          declare m1 in Nat

>>           m1: in Nat {move 3}


          define ispred m1:  n1 = Succ m1
```

```
>>           ispred: [(m1_1:in Nat) => ((n1
>>                  = Succ(m1_1)):prop)]
>>             {move 2}


          close

     define Haspred n1: Exists ispred

>>      Haspred: [(n1_1:in Nat) => (Exists([(m1_2:
>>                  in Nat) => ((n1_1 = Succ(m1_2)):
>>                  prop)])
>>              :prop)]
>>        {move 1}


     close

define basisexample :  Addition1 (Haspred 1, oneisone)

>> basisexample: [((Exists([(m1_1:in Nat) =>
>>              ((1 = Succ(m1_1)):prop)])
>>        Addition1 oneisone):that ((1 = 1) v
>>        Exists([(m1_2:in Nat) => ((1 = Succ(m1_2)):
>>             prop)]))
>>        )]
>>   {move 0}


open

     declare n1 in Nat

>>      n1: in Nat {move 2}


     declare indhypexample that (n1 = 1) v (Haspred n1)
```

```
>>      indhypexample: that ((n1 = 1) v Haspred(n1))
>>        {move 2}


     define line37 n1: Refleq Succ n1

>>      line37: [(n1_1:in Nat) => (Refleq(Succ(n1_1)):
>>             that (Succ(n1_1) = Succ(n1_1)))]
>>        {move 1}


     open

          declare m1 in Nat

>>           m1: in Nat {move 3}


          define ispred2 m1 :  Succ n1 = Succ m1

>>           ispred2: [(m1_1:in Nat) => ((Succ(n1)
>>                  = Succ(m1_1)):prop)]
>>             {move 2}


          close

     define line38 n1: fixprop(Haspred Succ n1,Ei ispred2, line37 n1)

>>      line38: [(n1_1:in Nat) => ((Haspred(Succ(n1_1))
>>             fixprop Ei([(m1_2:in Nat) => ((Succ(n1_1)
>>                  = Succ(m1_2)):prop)]
>>             ,line37(n1_1))):that Haspred(Succ(n1_1)))]
>>        {move 1}


     define line39 indhypexample: Addition2(Succ n1 = 1,line38 n1)
```

```
>>      line39: [(.n1_1:in Nat),(indhypexample_1:
>>             that ((.n1_1 = 1) v Haspred(.n1_1)))
>>             => (((Succ(.n1_1) = 1) Addition2
>>             line38(.n1_1)):that ((Succ(.n1_1)
>>             = 1) v Haspred(Succ(.n1_1))))]
>>        {move 1}


     close

define Inductionexample K:Induction line39, basisexample K

>> Inductionexample: [(K_1:in Nat) => (Induction([(.n1_4:
>>             in Nat),(indhypexample_4:that ((.n1_4
>>             = 1) v Exists([(m1_5:in Nat) =>
>>                  ((.n1_4 = Succ(m1_5)):prop)]))
>>             ) => (((Succ(.n1_4) = 1) Addition2
>>             (Exists([(m1_7:in Nat) => ((Succ(.n1_4)
>>                  = Succ(m1_7)):prop)])
>>             fixprop Ei([(m1_8:in Nat) => ((Succ(.n1_4)
>>                  = Succ(m1_8)):prop)]
>>             ,Refleq(Succ(.n1_4))))):that ((Succ(.n1_4)
>>             = 1) v Exists([(m1_9:in Nat) =>
>>                  ((Succ(.n1_4) = Succ(m1_9)):
>>                  prop)]))
>>             )]
>>        ,basisexample,K_1):that ((K_1 = 1) v
>>        Exists([(m1_10:in Nat) => ((K_1 = Succ(m1_10)):
>>             prop)]))
>>        )]
>>   {move 0}
```

We give the unique induction proof which doesn't actually use its induction hypothesis (if you review the structure of the proof, you can see this), the proof that each natural number is either 1 or a successor.

10.1 Operations of arithmetic

In this subsection we define the operations of addition and multiplication of natural numbers.

```
open

     declare n1 in Nat

>>       n1: in Nat {move 2}


     declare t1 in tau

>>       t1: in tau {move 2}


     construct f1 t1 : in tau

>>       f1: [(t1_1:in tau) => (---:in tau)]
>>         {move 1}


     define f2 n1 t1 :  f1 t1

>>       f2: [(n1_1:in Nat),(t1_1:in tau) =>
>>               (f1(t1_1):in tau)]
>>         {move 1}


     close

declare tinit in tau

>> tinit: in tau {move 1}


declare nt in Nat
```

```
>> nt: in Nat {move 1}
```

```
define iter f1, tinit nt : Iter f2, tinit Succ nt
```

```
>> iter: [(.tau_1:type),(f1_1:[(t1_2:in .tau_1)
>>             => (---:in .tau_1)]),
>>        (tinit_1:in .tau_1),(nt_1:in Nat) =>
>>        (Iter([(n1_4:in Nat),(t1_4:in .tau_1)
>>             => (f1_1(t1_4):in .tau_1)]
>>        ,tinit_1,Succ(nt_1)):in .tau_1)]
>>   {move 0}
```

If f is a function from type τ to type τ, `iter f x n` will represent $f^n(x)$, the result of applying the function f to x n times. Notice that `Iter` is applied to `Succ n` in the definition of `iter`.

```
define ++ n m :  iter Succ, n m
```

```
>> ++: [(n_1:in Nat),(m_1:in Nat) => (iter(Succ,
>>        n_1,m_1):in Nat)]
>>   {move 0}
```

The introduction of `iter` enables us to define $n + m$ as $\sigma^m(n)$.

```
open
```

```
     declare k1 in Nat
```

```
>>      k1: in Nat {move 2}
```

```
     declare k2 in Nat

>>       k2: in Nat {move 2}


     define Addn k1 k2: k2++n

>>       Addn: [(k1_1:in Nat),(k2_1:in Nat) =>
>>              ((k2_1 ++ n):in Nat)]
>>         {move 1}


     close

define ** n m :  Iter Addn, n, m

>> **: [(n_1:in Nat),(m_1:in Nat) => (Iter([(k1_3:
>>              in Nat),(k2_3:in Nat) => ((k2_3
>>              ++ n_1):in Nat)]
>>         ,n_1,m_1):in Nat)]
>>   {move 0}
```

The definition of multiplication is in terms of `Iter`. A nice definition as $(\sigma^n)^m(0)$ using `iter` would be possible if we started the natural numbers at 0, but we are quite interested in starting them at 1.