Introduction to the foundations of mathematics, using the Lestrade Type Inspector

Randall Holmes

April 15, 2020

Contents

The purpose of this document is to introduce a reader to the foundations of logic and mathematics using the Lestrade Type Inspector, a piece of software designed to allow the specification of mathematical objects in a very general way. It could also be used as an introduction to the software for someone familiar with the foundational subject matter.

Lestrade implements a particular very general framework for the implementation of mathematical objects, statements, and proofs of statements. Part of the underpinning of the approach is that in this framework the statements and their proofs are viewed as particular kinds of mathematical object themselves.

The actual implementation of foundational concepts of logic and mathematics here is not dictated by Lestrade: there is considerable latitude for different design decisions in the implementation of logic and mathematics in the framework. We may sometimes indicate alternative approaches.

1 Initial examples. Conjunction, implication, and their rules.

We begin with the implementation of the very simple concepts of logical conjunction, the use of the word "and" to link sentences, and logical implication, the use of "if…then…" to link sentences.

```
Lestrade execution:


declare A prop

>> A: prop {move 1}



declare B prop

>> B: prop {move 1}
```

Here is a bit of initial dialogue with Lestrade. Here we use the `declare` command to introduce two variables, A and B, of type `prop`, the type inhabited by mathematical statements.

Lines starting with Lestrade command names such as `declare`, `postulate`, `define` are entered by the user. Lines starting with `>>` are Lestrade responses to commands typed by the user.

```
Lestrade execution:



postulate & A B : prop

>> &: [(A_1:prop),(B_1:prop) => (---:prop)]
>>   {move 0}




postulate -> A B : prop

>> ->: [(A_1:prop),(B_1:prop) => (---:prop)]
>>   {move 0}
```

Here we declare the operations of conjunction and implication. At the moment, they look just the same: the only thing Lestrade knows about them so far is that they are operations taking two proposition inputs to a proposition output. Details of the input and output of Lestrade itself (the things the user enters and the replies that Lestrade produces) will be analyzed more carefully as we go forward.

```
Lestrade execution:



define proptest A B : (A & B) -> A

>> proptest: [(A_1:prop),(B_1:prop) => (((A_1
>>      & B_1) -> A_1):prop)]
```

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

We illustrate another Lestrade command, using `define` to introduce a defined operation. The main point here is to notice that Lestrade supports infix use of the conjunction and implication operators, though the Lestrade declaration commands requires their use in prefix position when they are newly declared. The Lestrade user should get used to typing lots of parentheses, though she does not need to use as many as are displayed in the output: she does need to be aware that in general terms all operations have the same precedence and group to the right if explicit parentheses are not provided.

```
Lestrade execution:


open

   declare A1 prop

>>    A1: prop {move 2}



   declare B1 prop

>>    B1: prop {move 2}



   define proptest2 A1 B1 : (A1 & B1) -> \
      A1

>>    proptest2: [(A1_1:prop),(B1_1:prop) =>
>>         (---:prop)]
>>      {move 1}
```

```
close
```

Here we do something subtle in the Lestrade declaration environment which we don't explain fully for now: the `open`...`close` environment creates a separate little Lestrade context. The alternative version `proptest2` of our defined notion will behave a little differently as we see at once.

```
Lestrade execution:


declare C prop

>> C: prop {move 1}



declare D prop

>> D: prop {move 1}



define zorch C D: proptest C & D, D -> C


>> zorch: [(C_1:prop),(D_1:prop) => (((C_1 &
>>      D_1) proptest (D_1 -> C_1)):prop)]
>>   {move 0}



define zorch2 C D: proptest2 C & D, D -> \
   C

>> zorch2: [(C_1:prop),(D_1:prop) => ((((C_1
>>      & D_1) & (D_1 -> C_1)) -> (C_1 & D_1)):
```

```
>>      prop)]
>>   {move 0}
```

Here we use `proptest` and `proptest2` to define new operations `zorch` and `zorch2`. The interesting thing which happens is that the operation `proptest2` which was defined in its own little local context gets expanded when it is used, while `proptest` (which "means" the same thing) is left unexpanded. Expansion of definitions is the main kind of "calculation" that Lestrade does, though we may detect it doing more complex things as we go forward.

Now we will return to our main line of development, introducing the machinery of proof in Lestrade.

```
Lestrade execution:


declare aa that A

>> aa: that A {move 1}



declare bb that B

>> bb: that B {move 1}
```

We declare new variables `aa` and `bb`. The sorts of these variables require special explanation. With each proposition p of sort `prop`, we associate a new sort `that` p inhabited by proofs of p, or, perhaps better, evidence that p is true.

```
Lestrade execution:
```

```
postulate Andproof0 A B aa bb:that A & B


>> Andproof0: [(A_1:prop),(B_1:prop),(aa_1:that
>>      A_1),(bb_1:that B_1) => (---:that (A_1
>>      & B_1))]
>>   {move 0}
```

```
postulate Andproof aa bb:that A & B

>> Andproof: [(.A_1:prop),(aa_1:that .A_1),(.B_1:
>>      prop),(bb_1:that .B_1) => (---:that (.A_1
>>      & .B_1))]
>>   {move 0}
```

```
define Selfand aa : Andproof aa aa

>> Selfand: [(.A_1:prop),(aa_1:that .A_1) =>
>>      ((aa_1 Andproof aa_1):that (.A_1 & .A_1))]
>>   {move 0}
```

And now we introduce a rule of proof: if we have evidence that A and evidence that B, we can conclude $A \wedge B$: to conclude $A \wedge B$ is equivalent to postulateing or defining an object of sort `that A & B`. The symbol $\wedge$ is the standard representation of "and" in formal logic; Lestrade uses `&` because of the limitations of the typewriter keyboard.

The fully verbose version `Andproof0` takes the arguments A, B, `aa` and `bb`, and the Lestrade framework requires these arguments officially. Notice though that from the arguments `aa` and `bb` we can deduce what A and B have to be: the second version `Andproof` uses the "implicit argument inference" feature of Lestrade to allow the user to enter just the names of the proofs, deducing the names of the propositions proved. The declaration that

Lestrade gives as a response makes it clear that it knows about the hidden arguments.

Selfand is a defined operation on proofs: from a proof of A it generates a proof of $A \wedge A$. We might think that this is a proof of "If A then $A \wedge A$, or $A \rightarrow (A \wedge A)$: the fact that we might think this is a hint as to how Lestrade represents proofs of implications.

```
Lestrade execution:


declare xx that A & B

>> xx: that (A & B) {move 1}



postulate Simplification1 xx : that A

>> Simplification1: [(.A_1:prop),(.B_1:prop),
>>      (xx_1:that (.A_1 & .B_1)) => (---:that
>>      .A_1)]
>>   {move 0}



postulate Simplification2 xx : that B

>> Simplification2: [(.A_1:prop),(.B_1:prop),
>>      (xx_1:that (.A_1 & .B_1)) => (---:that
>>      .B_1)]
>>   {move 0}
```

For completeness, we introduce the other two (quite obvious) rules of conjunction: from evidence xx for $A \wedge B$, we can extract evidence for A and evidence for B. We introduce them in forms which hide implicit arguments.

```
Lestrade execution:
```

```
declare cc that A->B

>> cc: that (A -> B) {move 1}



postulate Mp aa cc: that B

>> Mp: [(.A_1:prop),(aa_1:that .A_1),(.B_1:prop),
>>      (cc_1:that (.A_1 -> .B_1)) => (---:that
>>      .B_1)]
>>   {move 0}
```

This snippet of code embodies the traditional rule of *modus ponens*: given evidence for A and evidence for $A \to B$, we have evidence for B. We have only given the version with implicit arguments. It is interesting to note that the order of the arguments of `Mp` is probably not what we would choose if we were writing all the arguments explicitly: but it works.

```
Lestrade execution:


open

   declare aaa that A

>>    aaa: that A {move 2}



   postulate ded aaa that B

>>    ded: [(aaa_1:that A) => (---:that B)]
>>      {move 1}
```

```
   close

postulate Deduction ded : that A -> B

>> Deduction: [(.A_1:prop),(.B_1:prop),(ded_1:
>>      [(aaa_2:that .A_1) => (---:that .B_1)])
>>      => (---:that (.A_1 -> .B_1))]
>>   {move 0}
```

This piece of code implements a standard strategy for proving implications: if assuming A allows us to deduce B, we can conclude $A \rightarrow B$. What is quite tricky is how Lestrade represents this. We open a little environment in which we postulate the function `ded` which takes evidence `aaa` for A to evidence for B: we close this environment, and the symbol `ded` remains as a variable representing a function of this type. We are then able to postulate a function which takes any such function to evidence for $A \rightarrow B$. We *will* in due course have a careful discussion of Lestrade environments. For the moment, we will content ourselves with giving an example of how this is used.

A side remark to those in the know: it is important to notice that a proof of an implication is not identified with a function from proofs of its antecedent to proofs of its consequent, but obtained from such a function by applying a constructor casting from a function sort to an object sort (see the next section on metaphysics of Lestrade for a discussion of object vs. function sorts).

```
Lestrade execution:


open

   declare aaa that A
```

```
>>    aaa: that A {move 2}



   define selfand aaa : Andproof aaa aaa


>>    selfand: [(aaa_1:that A) => (---:that
>>         (A & A))]
>>      {move 1}




   close

define Selfand2 A : Deduction selfand

>> Selfand2: [(A_1:prop) => (Deduction([(aaa_2:
>>         that A_1) => ((aaa_2 Andproof aaa_2):
>>         that (A_1 & A_1))])
>>      :that (A_1 -> (A_1 & A_1)))]
>>   {move 0}
```

Here we actually prove the theorem $A \to (A \wedge A)$ for any proposition A. It is interesting to observe that this is actually a function of the proposition A rather than of a proof of A: whether A itself is true or not, this theorem is true, and the definition of the function `Selfand2` encapsulates reasoning justifying this: from a proposition A, we can postulate evidence for the proposition $A \wedge A$.

An interesting feature of the Lestrade output is that it contains a mathematical expression

$$[(\mathtt{A}_1 : \mathtt{prop}) => (\mathtt{Deduction}([(\mathtt{aaa}_2 : \mathtt{that}\, \mathtt{A}_1) => ((\mathtt{aaa}_2\, \mathtt{Andproof}\, \mathtt{aaa}_2) : \mathtt{that}(\mathtt{A}_1 \& \mathtt{A}_1))])$$

standing for the proof as a mathematical object. Lestrade allows itself output notation significantly more complex that the user input notation, but with experience we will be able to read this.

2 A brief discussion of the metaphysics of Lestrade

Probably we should explain ourselves a bit more.

The most general word used for things we talk about in Lestrade is *entity*. Entities are further partitioned into *objects* and *functions*.

Entities have sorts: the sort indicates what kind of thing we are talking about.

The sorts of object can be reviewed quickly:

1. `prop` is the sort of propositions, i.e., mathematical statements.

2. For each proposition p, we provide a sort `that` p inhabited by evidence that p is true. A proof of p is such evidence, and explicitly constructed objects of sort `that` p will be referred to as "proofs of p"; but to suppose that p is true (to postulate an object of the sort `that` p) is not the same thing as to suppose that p has actually been proved or even can be proved.

3. `obj` is a sort inhabited by untyped mathematical objects.

4. `type` is a sort inhabited by "type labels". An example of an object of sort `type` would be the label `Nat` for the sort "natural number".

5. For each τ of sort `type` we provide a sort `in` τ inhabited by objects of type τ. If n is a natural number, it might be construed as of sort `in Nat`. Something of sort `in` τ may more briefly be said to be of type τ.

If the reader notices an analogy between `prop`/`that` and `type`/`in`, she is perceptive.

Functions are more complicated and their sorts are more complicated. A Lestrade function takes a list of arguments of a fixed length, each item of which is of a sort possibly determined by earlier arguments in the list, and yields output of a sort which may depend on its arguments. A lot of the logical power of this framework comes from the fact that the sort of an argument of a function may depend on the values of earlier arguments, and the sort of the output may depend on the values of the inputs. One mechanism which makes such dependencies possible is the fact that the object sorts of the form `that` p and `in` τ may contain quite complex expressions abbreviated here by

p, τ; we have already seen this in Lestrade output above; function sorts also have complex internal structure which supports such dependencies.

The general notation for a function sort is

$$(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (-, \tau)$$

The variables x_i representing the arguments are dummy variables (they are "bound" in this expression) Distinct function sort expressions (including ones which might appear as τ_i's or parts of τ_i's) have different dummy variables. Each τ_i is an expression representing the sort of x_i, which may be an object or a function sort and is allowed to include x_j's only for $j < i$. The output sort τ will be an object sort, not a function sort, and may include any of the x_i.

A species of notation for a function used in Lestrade output is

$$(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (y, \tau),$$

where y is an expression for the value of the function which may of course include any or all of the x_i's (and which must be of the object sort τ): the formation rules for such an expression are the same as for function sorts: the function sort expression $(x_1 : \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (-, \tau)$ must be well-formed for the expression above to be well-formed.

When a function is declared in Lestrade and explicitly defined, the type reported for it is actually this notation for it. The user will always refer to it using its name (the identifier declared with this type): users do not enter function sort notations or function notations.

The account given here should allow the reader to make a stab at interpreting details of Lestrade responses to user commands which we skated over above.

3 The care and feeding of declarations: the system of possible worlds or "moves"

We have to give an account of the declaration environments of Lestrade. We'll do this in the simplest way (in which all declared environments are anonymous and in a sense ephemeral: we will look later at the consequences of allowing environments to be named and saved).

In the simplest model of what we are doing, the Lestrade user is working in a finite sequence of environments indexed by natural numbers, called "move 0", "move 1",…,"move i", "move $i+1$". Move i is called "the last move" and move $i+1$ is called "the next move". There are always at least two moves, so all four explicitly given items are present, though they may not all be distinct. Each move contains an ordered list of declarations of identifiers as representing entities of given sorts. The sort of an identifier declared at a given sort will not mention identifiers declared at moves of higher index or declared at the same move but later in the list of declarations. Entities declared at the last move or earlier moves are to be thought of as constant; entities declared at the next move are to be thought of as variable.

By a fresh identifier we mean an identifier not declared at the moment in any move. It will never be the case that the same identifier is declared more than once.

The `declare` command takes a fresh identifier and an object sort as its two arguments (in that order) and declares the identifier as a variable of the given sort in the next move (placed last in the order on the move). A function variable cannot be declared in this way because, as we will see, the user cannot write a function sort.

The `postulate` command takes a fresh identifier followed by zero or more arguments (variables declared previously at the next move, appearing in the order in which their declarations appear in the next move), followed by an object sort [optionally separated from the previous arguments by a colon "`:`"; this is sometimes mandatory for the sake of the parser]. If there are zero arguments, the identifier is declared as being of the given sort, but at (the end of) the last move rather than the next move. This can be thought of as declaring a constant (relatively speaking, as we will see). If there are arguments x_i of types τ_i and the output type is τ, the identifier is declared at the last move (not at the next move!) and appearing finally in the order on the last move, as a function of sort

$$(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (-, \tau)$$

(with the refinement that the names of the parameters, since they become bound, are systematically changed).

The `postulate` command can be thought of as declaring axioms and primitive notions, when it is used when $i = 0$. At higher indexed moves, what it is doing is subtler, but will become evident with experience: we will see that in combination with the `open` and `close` commands it allows declaration of function variables.

The `define` command is a sister command of the `postulate` command: the keyword is followed by an identifier, then by zero or more arguments, variables x_i of type τ_i appearing in the same order in which they were declared, then by a Lestrade expression y of an object type [always separated from the previous arguments by a colon :]. The identifier is defined at the last move (not the next move), and finally in the order on the last move, as

$$(x_1 : \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (y, \tau),$$

as long as sort checking reports that this is possible [in the case where there are no arguments, it is just defined as y]. Identifiers declared in this way are not eligible to serve as arguments of functions (they are not variables).

The `open` command introduces a new move with the index $i+2$: as it were, the parameter i is incremented, so that the old "next move", move $i+1$, becomes the new "last move", and the new move $i+2$ is the new next move. We call this action "opening move $i+2$".

The `close` command erases all information in move $i+1$ and decrements the parameter i, if $i > 1$; it is not possible to close move 1. The old "last move" move i becomes the new next move, and move $i-1$ becomes the new last move. We call this action "closing move $i+1$".

The `clearcurrent` command removes all declarations from move $i+1$ but does not decrement the counter: at the end of this action, move i is unchanged and move $i+1$ is empty. This amounts to clearing accumulated variable declarations; it is needed because there is no other way to remove declarations from move 1. It will be a while before we see uses of this command: over a large initial segment of the document, we will suppose that the program remembers all previous move 1 declarations.

There are devices whereby moves can be saved and then reopened after being closed, which lead to some complexities, but these can be ignored for the present.

It may seem that we cannot create a function variable (recall that we said above that functions can have functions as arguments) but we can and in fact we have already illustrated this in an example above. One creates a function variable in move $i+1$ by opening move $i+2$, declaring desired variable parameters, postulateing a function of the desired type in move $i+1$ in its then role as the last move, then closing move $i+2$ whereupon the constructed function is now a variable. We did this (and the reader may now review the example to see that it conforms with our account) in postulateing `Deduction` above, which needed the function parameter `ded`.

Functions found in the next move which were introduced by the `define` command when there were more moves do not become variables: they are as it were "variable expressions", and a distinctive point about these is that where they are used in the final argument of a `define` command they must be expanded out (as `proptest2` was in an example above) as a defined identifier at move $i+1$ cannot appear in a declaration at move i. Where a defined operator declared at the last move is used in applied position, its application is carried out (suitable substitutions are made) as in the example above; where it appears as an argument it is replaced by its anonymous formal notation.

A further point about declarations of functions which must be noted, though its details are nasty, is the permission we give ourselves to not give all arguments of a function under certain circumstances. In fact, any non-defined identifier declared at the next move appearing in the sort of a variable appearing as an argument of a function must itself be an earlier argument of that function: the input/output mechanism of Lestrade itself allows us to hide this, omitting arguments when their presence can be deduced. If we did not do this, we would have a lot of arguments in argument lists which "felt" redundant, like A and B as arguments of `Andproof0` (it being evident from the sorts of `aa` and `bb` what A and B must be).

We make a philosophical remark at this point. The currently popular view of the nature of functions is that they are as it were actually infinite tables containing all their values. We resist this. We regard a function as determined by a specification of how a value is to be obtained (or, in the case of a primitive notion, simply *that* a value of given sort can be obtained) from *any given* sequence of inputs of appropriate sorts which may happen to be presented now or in the future, not from all possible such sequences in a way given all at once. The arbitrary objects used as inputs in a function definition can each be viewed as a single object drawn from a "possible world" ("the next move") accessible from the world which is our current standing point ("the last move"). Another metaphor which might be helpful is that objects at the next move are things to be chosen in the future; we do not know anything about them except what is given in their sort. When we declare a function as a primitive, we declare that there is a construction principle which for any given inputs of given types will give an output of that type: we do not presume that we have given such outputs for all possible inputs (such outputs are produced on demand when we apply the constructed function to specific inputs). In this way we preserve the possibility of the view that all infinities are potential, never completely realized. Nonetheless, the mathematical consequences of the particular Lestrade theory we present are fully classical.

4 A proof as an example $A \wedge B \rightarrow B \wedge A$.

We give the proof of a simple theorem of propositional logic, then present the proof in the form of Lestrade declarations.

Theorem: $A \wedge B \rightarrow B \wedge A$

> Assume $A \wedge B$ for the sake of argument: our goal is to show that $B \wedge A$ follows.
>
> B follows from $A \wedge B$ by simplification. A follows from $A \wedge B$ by simplification.
>
> The local conclusion $B \wedge A$ follows by conjunction from B and A.
>
> By deduction, we can conclude $A \wedge B \rightarrow B \wedge A$.

```
Lestrade execution:



open

   declare yy that A & B

>>    yy: that (A & B) {move 2}




   define zz yy : Simplification1 yy

>>    zz: [(yy_1:that (A & B)) => (---:that
>>         A)]
>>      {move 1}
```

```
   define ww yy : Simplification2 yy

>>    ww: [(yy_1:that (A & B)) => (---:that
>>         B)]
>>      {move 1}



   define uu yy : Andproof (ww yy, zz yy)


>>    uu: [(yy_1:that (A & B)) => (---:that
>>         (B & A))]
>>      {move 1}



   close

define Andconj A B: Deduction uu

>> Andconj: [(A_1:prop),(B_1:prop) => (Deduction([(yy_2:
>>         that (A_1 & B_1)) => ((Simplification2(yy_2)
>>         Andproof Simplification1(yy_2)):that
>>         (B_1 & A_1))])
>>      :that ((A_1 & B_1) -> (B_1 & A_1)))]
>>   {move 0}
```

The Lestrade declarations given embody the proof given. One very subtle point is that the functions `ww` and `yy` are distinct from `Simplification1` and `Simplification2`, because the latter functions take additional arguments which are not visible.

A point to note is that the argument under the hypothesis $A \wedge B$, assumed for the sake of argument, corresponds to the introduction of a new environment by the `open` command in which the variable `yy` of sort `that` $(A\&B)$ is declared.

5 The Lestrade user input language

We discuss practical details of entering mathematical expressions in the language of Lestrade. This section concentrates on what users can enter at the keyboard. For good or ill (later versions may modify this) users cannot enter notations for function sorts or the related notation for functions. This might change in future versions. For now the user notation is entirely "applicative" and does not allow variable binding.

Lestrade identifiers are the first detail of the syntax. An identifier is a string of characters of positive length, consisting of zero or one capital letters, followed by zero or more lower case letters, followed by zero or more numerals.

A Lestrade object expression is either an identifier declared of an object sort, or an application expression $f(t_1, \ldots, t_n)$ where f is an identifier declared as of function type with n arguments, and $t_1, \ldots, t_n$ are expressions of the correct sorts (some may be function expressions). A mixfix expression $(t_1\, f\, t_2, \ldots, t_n)$ is well-formed under the same conditions and has the same referent.

The parentheses and commas in these expressions may be omitted under some circumstances. All infix and mixfix operators have the same precedence and group to the right (in the absence of restrictive punctuation they will take as many arguments as they can). A function symbol used as an argument must be followed by a comma or parenthesis to avoid it attempting to take the next expression as an argument. A parenthesis following a function symbol will always be taken as opening an argument list (so if one wants to enclose the first argument in parentheses one must also enclose the entire argument list in parentheses). A function symbol representing a function taking more than one argument must be preceded by a comma when it might otherwise take a preceding object expression as a first argument [reading a mixfix expression]. A function symbol appearing as the first argument of a mixfix expression must be enclosed in parentheses to avoid the function symbol trying to eat the mixfix.

Function expressions include identifiers declared as of function type, and expressions $f(t_1, \ldots, t_m)$ where $m < n$, the number of arguments taken by

f. Such expressions are understood as functions of $(x_{m+1}, \ldots, x_n)$, and may only appear as arguments, not function or mixfix symbols. The parentheses around the argument list in such a function expression are mandatory.

An additional important punctuation device is the use of a colon : to separate the final argument of a `postulate` or `define` command from the preceding arguments. The colon is optional in the `postulate` command (though it may be needed if the final preceding argument is a function identifier); it is mandatory in the `define` command. (The colon is neither needed nor allowed in the `declare` command).

Lestrade output will use infix form for functions of two arguments where the first argument is not of function type. Lestrade output will never use mixfix notation for functions of more than two arguments.

In general, problems with parsing of input notation can be solved by explicitly writing more parentheses and commas. In Lestrade output, all parentheses and commas are shown.

6 We begin considering ontology: equality primitives introduced. The biconditional as equality on propositions. Identification of proofs of the same proposition.

We are by no means through with logic, but we will begin to consider the treatment of objects. In this section we introduce the notion of equality and its basic primitives. Equality is defined for typed mathematical objects: related notions applying to propositions and their proofs are discussed, and defining equality for untyped mathematical objects of sort `obj` is straightforward.

```
Lestrade execution:


declare T type

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

```
open

   declare t1 in T

>>    t1: in T {move 2}



   postulate tpred t1 : prop

>>    tpred: [(t1_1:in T) => (---:prop)]
>>      {move 1}



   close
```

We introduce a general object type T which will be a hidden parameter of our notions of equality. We then introduce a predicate `tpred` of objects of type T (i.e, of sort `in` T).

```
Lestrade execution:


declare t in T

>> t: in T {move 1}



declare u in T

>> u: in T {move 1}



postulate = t u : prop
```

```
>> =: [(.T_1:type),(t_1:in .T_1),(u_1:in .T_1)
>>      => (---:prop)]
>>   {move 0}
```

```
declare eqev that t=u

>> eqev: that (t = u) {move 1}
```

We introduce the primitive notion of equality and evidence of equality $t = u$.

Lestrade execution:

```
declare tpredev that tpred t

>> tpredev: that tpred(t) {move 1}
```

```
postulate Substitution0 tpred, eqev tpredev \
   that tpred u

>> Substitution0: [(.T_1:type),(tpred_1:[(t1_2:
>>         in .T_1) => (---:prop)]),
>>      (.t_1:in .T_1),(.u_1:in .T_1),(eqev_1:
>>      that (.t_1 = .u_1)),(tpredev_1:that tpred_1(.t_1))
>>      => (---:that tpred_1(.u_1))]
>>   {move 0}
```

```
define Substitution eqev tpredev : Substitution0 \
```

```
   tpred, eqev tpredev

>> Substitution: [(.T_1:type),(.t_1:in .T_1),
>>      (.u_1:in .T_1),(eqev_1:that (.t_1 = .u_1)),
>>      (.tpred_1:[(t1_2:in .T_1) => (---:prop)]),
>>      (tpredev_1:that .tpred_1(.t_1)) => (Substitution0(.tpred_1,
>>      eqev_1,tpredev_1):that .tpred_1(.u_1))]
>>   {move 0}
```

We introduce the substitution rule of equality, whose type is perhaps the most complex yet introduced. There are two different versions with different choices of explicitly given arguments.

```
Lestrade execution:


postulate Reflexeq t : that t=t

>> Reflexeq: [(.T_1:type),(t_1:in .T_1) => (---:
>>      that (t_1 = t_1))]
>>   {move 0}
```

The other primitive rule of equality is the reflexivity rule $t = t$. We will see that other familiar rules of equality such as symmetry and transitivity can be proved.

```
Lestrade execution:


open

   declare t17 in T

>>    t17: in T {move 2}
```

```
   declare u17 in T

>>    u17: in T {move 2}



   open

      declare v17 in T

>>       v17: in T {move 3}



      define noncepred v17 : v17=t17

>>       noncepred: [(v17_1:in T) => (---:prop)]
>>         {move 2}



      close

   declare eqev17 that t17=u17

>>    eqev17: that (t17 = u17) {move 2}



   define eqsymm0 eqev17: Substitution0 noncepred, \
      eqev17 Reflexeq t17

>>    eqsymm0: [(.t17_1:in T),(.u17_1:in T),
>>         (eqev17_1:that (.t17_1 = .u17_1)) =>
>>         (---:that (.u17_1 = .t17_1))]
```

```
>>      {move 1}



   close

define Eqsymm eqev : eqsymm0 eqev

>> Eqsymm: [(.T_1:type),(.t_1:in .T_1),(.u_1:
>>      in .T_1),(eqev_1:that (.t_1 = .u_1)) =>
>>      (Substitution0([(v17_2:in .T_1) => ((v17_2
>>         = .t_1):prop)]
>>      ,eqev_1,Reflexeq(.t_1)):that (.u_1 = .t_1))]
>>   {move 0}
```

We present the proof of symmetry of equality. Notice the use of an extra layer of environment so that the nonce predicate `noncepred` in the proof really is nonce: its declaration is at move 2 and so it is expanded out when `Eqsymm` is declared at move 0 (basically by copying the move 1 predicate `eqsymm0`). This is a useful strategy to keep the namespace from being cluttered.

Other notions of equality for sorts of functions may be introduced, as well as equality for untyped objects of sort `obj`.

We introduce the biconditional, which plays the role of equality for propositions.

```
Lestrade execution:


define <-> A B : (A -> B) & (B -> A)

>> <->: [(A_1:prop),(B_1:prop) => (((A_1 ->
>>      B_1) & (B_1 -> A_1)):prop)]
>>   {move 0}
```

```
open

   declare A1 prop

>>    A1: prop {move 2}



   postulate ppred A1 : prop

>>    ppred: [(A1_1:prop) => (---:prop)]
>>      {move 1}



   close

declare iffev that A <-> B

>> iffev: that (A <-> B) {move 1}



declare ppredev that ppred A

>> ppredev: that ppred(A) {move 1}



postulate Substitutionp0 ppred, iffev ppredev: \
   that ppred B

>> Substitutionp0: [(ppred_1:[(A1_2:prop) =>
>>         (---:prop)]),
>>      (.A_1:prop),(.B_1:prop),(iffev_1:that
>>      (.A_1 <-> .B_1)),(ppredev_1:that ppred_1(.A_1))
>>      => (---:that ppred_1(.B_1))]
```

```
>>   {move 0}



define Substitutionp iffev ppredev : Substitutionp0 \
   ppred, iffev ppredev

>> Substitutionp: [(.A_1:prop),(.B_1:prop),(iffev_1:
>>      that (.A_1 <-> .B_1)),(.ppred_1:[(A1_2:
>>         prop) => (---:prop)]),
>>      (ppredev_1:that .ppred_1(.A_1)) => (Substitutionp0(.ppred_1,
>>      iffev_1,ppredev_1):that .ppred_1(.B_1))]
>>   {move 0}



open

   declare aa1 that A

>>    aa1: that A {move 2}



   define pid aa1 : aa1

>>    pid: [(aa1_1:that A) => (---:that A)]
>>      {move 1}



   close

define Reflexp0 A : Deduction pid

>> Reflexp0: [(A_1:prop) => (Deduction([(aa1_2:
>>         that A_1) => (aa1_2:that A_1)])
>>      :that (A_1 -> A_1))]
```

```
>>   {move 0}



declare afix that A

>> afix: that A {move 1}



define propfixform A afix : afix

>> propfixform: [(A_1:prop),(afix_1:that A_1)
>>      => (afix_1:that A_1)]
>>   {move 0}



define Reflexp A : propfixform (A<->A,Andproof(Reflexp0 \
   A,Reflexp0 A))

>> Reflexp: [(A_1:prop) => (((A_1 <-> A_1) propfixform
>>      (Reflexp0(A_1) Andproof Reflexp0(A_1))):
>>      that (A_1 <-> A_1))]
>>   {move 0}



define Reflexp1 A: Andproof(Reflexp0 A,Reflexp0 \
   A)

>> Reflexp1: [(A_1:prop) => ((Reflexp0(A_1)
>>      Andproof Reflexp0(A_1)):that ((A_1 ->
>>      A_1) & (A_1 -> A_1)))]
>>   {move 0}
```

We make some observations about the biconditional development. A primitive `Substitutionp` is needed to justify substitution of logically equivalent propositions in general contexts, but the reflexivity property `Reflexp` is a theorem derivable from primitives we have already. Notice the use of `propfixform` to force the type of the output of `Reflexp` into the correct form: what happens if we don't use it is exhibited in the declaration of `Reflexp1`. The Lestrade matching facility is good enough that in fact `Reflexp1` would be usable for exactly the same purposes as `Reflexp`; the two functions match in type because Lestrade recognizes that the type of one is a definitional expansion of the type of the other. The pragmatic advantages of `Reflexp` for user understanding of what is going on are clear.

A notion of equality for objects of sorts `that` p (proofs or evidence) could be defined by analogy with what is given above for objects of sorts `in` p, and such a development could be given. A radical alternative (not appropriate for example for a constructive logic) is the following:

```
Lestrade execution:


open

   declare aa1 that A

>>    aa1: that A {move 2}



   postulate proofpred aa1 : prop

>>    proofpred: [(aa1_1:that A) => (---:prop)]
>>      {move 1}



   close

declare proofpredev that proofpred aa
```

```
>> proofpredev: that proofpred(aa) {move 1}



declare aax that A

>> aax: that A {move 1}



postulate Indifference proofpredev aax : \
   that proofpred aax

>> Indifference: [(.A_1:prop),(.proofpred_1:
>>      [(aa1_2:that .A_1) => (---:prop)]),
>>      (.aa_1:that .A_1),(proofpredev_1:that
>>      .proofpred_1(.aa_1)),(aax_1:that .A_1)
>>      => (---:that .proofpred_1(aax_1))]
>>   {move 0}
```

The primitive `Indifference` takes a proof that a first proof of p satisfies a predicate of proofs, and another proof of p, to a proof that the second proof of p satisfies the same predicate. In other words, `Indifference` witnesses the fact that each type `that` p is in effect inhabited by no more than one object.

To assume such an axiom is optional. If a constructive logic were preferred, in which information could be extracted from proofs, one would certainly not want such an axiom. It should be noted in general that Lestrade is a very flexible framework in which many different logical approaches can be implemented: our particular development of logical and mathematical concepts is in no way dictated by the framework.

7 Natural numbers introduced

In this section, we introduce the natural numbers, via the concept of iterated application of functions.

```
Lestrade execution:


postulate Nat type

>> Nat: type {move 0}



postulate 0 in Nat

>> 0: in Nat {move 0}



declare n1 in Nat

>> n1: in Nat {move 1}



postulate Succ n1 in Nat

>> Succ: [(n1_1:in Nat) => (---:in Nat)]
>>   {move 0}
```

The primitive notions of arithmetic are introduced. These are the type of natural numbers, the number zero, and the successor operation. We will see that the operations of addition and multiplication can be defined in terms of these, later.

```
Lestrade execution:


open
```

```
   declare n2 in Nat

>>    n2: in Nat {move 2}



   postulate Tt n2 type

>>    Tt: [(n2_1:in Nat) => (---:type)]
>>      {move 1}



   close

open

   declare n2 in Nat

>>    n2: in Nat {move 2}



   declare t1 in Tt n2

>>    t1: in Tt(n2) {move 2}



   postulate F t1 in Tt (Succ n2)

>>    F: [(.n2_1:in Nat),(t1_1:in Tt(.n2_1))
>>         => (---:in Tt(Succ(.n2_1)))]
>>      {move 1}



   close
```

```
declare init in Tt 0

>> init: in Tt(0) {move 1}



declare n in Nat

>> n: in Nat {move 1}



postulate Iterate F, init n : in Tt n

>> Iterate: [(.Tt_1:[(n2_2:in Nat) => (---:type)]),
>>      (F_1:[(.n2_3:in Nat),(t1_3:in .Tt_1(.n2_3))
>>         => (---:in .Tt_1(Succ(.n2_3)))]),
>>      (init_1:in .Tt_1(0)),(n_1:in Nat) => (---:
>>      in .Tt_1(n_1))]
>>   {move 0}



postulate Initialize F, init : that (Iterate \
   F, init 0) = init

>> Initialize: [(.Tt_1:[(n2_2:in Nat) => (---:
>>         type)]),
>>      (F_1:[(.n2_3:in Nat),(t1_3:in .Tt_1(.n2_3))
>>         => (---:in .Tt_1(Succ(.n2_3)))]),
>>      (init_1:in .Tt_1(0)) => (---:that (Iterate(F_1,
>>      init_1,0) = init_1))]
>>   {move 0}



postulate Iterstep F, init n : that \
```

```
   (Iterate F, init (Succ n)) = F(Iterate F, \
   init n)

>> Iterstep: [(.Tt_1:[(n2_2:in Nat) => (---:
>>          type)]),
>>       (F_1:[(.n2_3:in Nat),(t1_3:in .Tt_1(.n2_3))
>>          => (---:in .Tt_1(Succ(.n2_3)))]),
>>       (init_1:in .Tt_1(0)),(n_1:in Nat) => (---:
>>       that (Iterate(F_1,init_1,Succ(n_1)) =
>>       (n_1 F_1 Iterate(F_1,init_1,n_1))))]
>>    {move 0}
```

We introduce the basic equations governing iterated application of a function. The fact that the type of the output can depend on a numerical argument will be used below in exhibiting the proof of the principle of mathematical induction. The type valued function `Tt` can be taken to be constant and the function `F` to be not dependent on the numerical argument to support simple iteration.

```
Lestrade execution:


open

   declare n99 in Nat

>>    n99: in Nat {move 2}



   declare t99 in T

>>    t99: in T {move 2}
```

```
   postulate F99 t99 in T

>>    F99: [(t99_1:in T) => (---:in T)]
>>      {move 1}



   define F98 n99 t99: F99 t99

>>    F98: [(n99_1:in Nat),(t99_1:in T) => (---:
>>         in T)]
>>      {move 1}



   close

declare init98 in T

>> init98: in T {move 1}



declare n98 in Nat

>> n98: in Nat {move 1}



define Simpleiter F99, init98 n98 : Iterate \
   F98, init98 n98

>> Simpleiter: [(.T_1:type),(F99_1:[(t99_2:in
>>         .T_1) => (---:in .T_1)]),
>>      (init98_1:in .T_1),(n98_1:in Nat) => (Iterate([(n99_4:
>>         in Nat),(t99_4:in .T_1) => (F99_1(t99_4):
>>         in .T_1)]
>>      ,init98_1,n98_1):in .T_1)]
```

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

```
define Simpleinit F99,init98 : Initialize \
   F98,init98

>> Simpleinit: [(.T_1:type),(F99_1:[(t99_2:in
>>          .T_1) => (---:in .T_1)]),
>>       (init98_1:in .T_1) => (Initialize([(n99_4:
>>          in Nat),(t99_4:in .T_1) => (F99_1(t99_4):
>>          in .T_1)]
>>       ,init98_1):that (Iterate([(n99_6:in Nat),
>>          (t99_6:in .T_1) => (F99_1(t99_6):in
>>          .T_1)]
>>       ,init98_1,0) = init98_1))]
>>   {move 0}
```

```
define Simpleiterstep F99,init98,n98 : Iterstep \
   F98, init98 n98

>> Simpleiterstep: [(.T_1:type),(F99_1:[(t99_2:
>>          in .T_1) => (---:in .T_1)]),
>>       (init98_1:in .T_1),(n98_1:in Nat) => (Iterstep([(n99_4:
>>          in Nat),(t99_4:in .T_1) => (F99_1(t99_4):
>>          in .T_1)]
>>       ,init98_1,n98_1):that (Iterate([(n99_6:
>>          in Nat),(t99_6:in .T_1) => (F99_1(t99_6):
>>          in .T_1)]
>>       ,init98_1,Succ(n98_1)) = F99_1(Iterate([(n99_8:
>>          in Nat),(t99_8:in .T_1) => (F99_1(t99_8):
>>          in .T_1)]
>>       ,init98_1,n98_1))))]
>>   {move 0}
```

We define simple iteration over a single type.

The very similar declarations which support the principle of mathematical induction follow. These are entirely analogous to the declarations for iteration of a function through a sequence of types above, but working with types of proofs or evidence rather than types of object, and analogues of `Initialize` and `Iterstep` do not seem to be required as we do not generally consider equations between proofs.

```
Lestrade execution:



open

   declare n2 in Nat

>>    n2: in Nat {move 2}



   postulate Pp n2 prop

>>    Pp: [(n2_1:in Nat) => (---:prop)]
>>      {move 1}



   close

open

   declare n2 in Nat

>>    n2: in Nat {move 2}



   declare t1 that Pp n2
```

```
>>    t1: that Pp(n2) {move 2}



   postulate Fp t1 that Pp (Succ n2)

>>    Fp: [(.n2_1:in Nat),(t1_1:that Pp(.n2_1))
>>         => (---:that Pp(Succ(.n2_1)))]
>>      {move 1}



   close

declare initp that Pp 0

>> initp: that Pp(0) {move 1}



declare np in Nat

>> np: in Nat {move 1}



postulate Iteratep Fp, initp np : that Pp \
   np

>> Iteratep: [(.Pp_1:[(n2_2:in Nat) => (---:
>>         prop)]),
>>      (Fp_1:[(.n2_3:in Nat),(t1_3:that .Pp_1(.n2_3))
>>         => (---:that .Pp_1(Succ(.n2_3)))]),
>>      (initp_1:that .Pp_1(0)),(np_1:in Nat)
>>      => (---:that .Pp_1(np_1))]
>>   {move 0}
```

Technical note: We discuss the question of the most general form an iteration operator can take in the Lestrade sort system. If f takes an argument t of type τ_1 to type $\tau(t)$, there is no latitude for $\tau(t)$ to be anything but τ_1 for iteration to be possible. Suppose that f actually takes an additional hidden argument, so its actual form is $f(u,t)$, where t is of type $\tau_1(u)$ and the output is of type $\tau(u,t)$. For iteration to be possible, it must be the case that $\tau(u,t) = \tau_1(g(u))$, where $g(u)$ is of the same constant sort as u. So $f^n(t)$ in this case is of type $\tau_1(g^n(u))$, and this rather more abstract iteration framework might be thought to implementable in terms of our more concrete approach taking the type of u to be `Nat` and the operation g to be `Succ`, but the type checker cannot handle this in its basic form (the rewriting feature of Lestrade, not yet discussed, may enable this more abstract scheme to be implemented using the primitives given here and type check correctly, by allowing a simple computation on the type index to be carried out automatically during type checking: in effect, the type checker would need to be able to match $\tau_1(g^{n+1}(u))$ with $\tau_1(g(g^n(u)))$ for this to work, and the rewrite feature may support this).

8 The universal quantifier. Principle of mathematical induction.

In this section we introduce the notion of universal quantification (over types of mathematical object) and develop the familiar form of the principle of mathematical induction.

```
Lestrade execution:


postulate Forall tpred : prop

>> Forall: [(.T_1:type),(tpred_1:[(t1_2:in .T_1)
>>          => (---:prop)])
>>       => (---:prop)]
>>   {move 0}
```

```
declare univev that Forall tpred

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



declare ttt in T

>> ttt: in T {move 1}



postulate Uinst univev ttt : that tpred ttt


>> Uinst: [(.T_1:type),(.tpred_1:[(t1_2:in .T_1)
>>         => (---:prop)]),
>>      (univev_1:that Forall(.tpred_1)),(ttt_1:
>>      in .T_1) => (---:that .tpred_1(ttt_1))]
>>   {move 0}



open

   declare ttt1 in T

>>    ttt1: in T {move 2}



   postulate ugen ttt1 that tpred ttt1

>>    ugen: [(ttt1_1:in T) => (---:that tpred(ttt1_1))]
>>      {move 1}
```

```
   close

postulate Ugen ugen : that Forall tpred

>> Ugen: [(.T_1:type),(.tpred_1:[(t1_2:in .T_1)
>>          => (---:prop)]),
>>       (ugen_1:[(ttt1_3:in .T_1) => (---:that
>>          .tpred_1(ttt1_3))])
>>       => (---:that Forall(.tpred_1))]
>>    {move 0}
```

Here is the development of the universal quantifier (over a type) and its basic rules. The usual notation for `Forall(tpred)` in mathematical text is $(\forall x \in T : \texttt{tpred}(x))$, where $\texttt{tpred}(x)$ may be expanded out. This is read "for all x in T, $\texttt{tpred}(x)$". We should note that we are being bad here, conflating x being of type T with x belonging to a set T. Our excuse for this is that mathematical reasoning is usually done in an officially untyped language, where actual types of mathematical object are usually referred to via sets.

In contrast with Automath and other dependent type provers, evidence for a universal statement is not identified with a suitable dependently typed function, but is obtained by applying a suitable constructor to such a function to get an object of the appropriate object type. This means that Lestrade, unlike Automath, does not automatically support quantification over all sorts. This weakness of the framework will turn out to be useful in the formulation of an ambiguous version of the simple theory of types below.

```
Lestrade execution:


open

   declare n2 in Nat

>>    n2: in Nat {move 2}
```

```
   postulate natpred n2 prop

>>    natpred: [(n2_1:in Nat) => (---:prop)]
>>      {move 1}



   close

open

   declare n2 in Nat

>>    n2: in Nat {move 2}



   define indimp n2 : (natpred n2) -> natpred \
      (Succ n2)

>>    indimp: [(n2_1:in Nat) => (---:prop)]
>>      {move 1}



   close

declare ind that Forall indimp

>> ind: that Forall(indimp) {move 1}



declare basis that natpred 0

>> basis: that natpred(0) {move 1}
```

Here are familiar prerequisites for mathematical induction, the basis step, evidence for $\texttt{natpred}(0)$, and the induction step, evidence for

$$(\forall n \in \texttt{Nat} : \texttt{natpred}(n) \rightarrow \texttt{natpred}(n+1)).$$

The way in which the induction step is formulated is a consequence of the fact that we reference abstractions in user input only by names, not "anonymously" with variable binding expressions.

```
Lestrade execution:


open

   declare n2 in Nat

>>     n2: in Nat {move 2}



   declare indhyp that natpred n2

>>     indhyp: that natpred(n2) {move 2}



   define step1 n2 : Uinst ind n2

>>     step1: [(n2_1:in Nat) => (---:that indimp(n2_1))]
>>       {move 1}



   define step2 n2 indhyp : Mp (indhyp,step1 \
      n2)
```

```
>>    step2: [(n2_1:in Nat),(indhyp_1:that natpred(n2_1))
>>         => (---:that natpred(Succ(n2_1)))]
>>      {move 1}
```

```
   close

declare nq in Nat

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

```
define Induction1 ind basis nq : Iteratep \
   step2, basis, nq

>> Induction1: [(.natpred_1:[(n2_2:in Nat) =>
>>         (---:prop)]),
>>      (ind_1:that Forall([(n2_3:in Nat) => ((.natpred_1(n2_3)
>>         -> .natpred_1(Succ(n2_3))):prop)]))
>>      ,(basis_1:that .natpred_1(0)),(nq_1:in
>>      Nat) => (Iteratep([(n2_4:in Nat),(indhyp_4:
>>         that .natpred_1(n2_4)) => ((indhyp_4
>>         Mp (ind_1 Uinst n2_4)):that .natpred_1(Succ(n2_4)))]
>>      ,basis_1,nq_1):that .natpred_1(nq_1))]
>>   {move 0}
```

```
define Induction ind basis : Ugen(Induction1 \
   (ind, basis))

>> Induction: [(.natpred_1:[(n2_2:in Nat) =>
>>         (---:prop)]),
>>      (ind_1:that Forall([(n2_3:in Nat) => ((.natpred_1(n2_3)
>>         -> .natpred_1(Succ(n2_3))):prop)]))
>>      ,(basis_1:that .natpred_1(0)) => (Ugen([(nq_4:
```

```
>>          in Nat) => (Induction1(ind_1,basis_1,
>>          nq_4):that .natpred_1(nq_4))])
>>       :that Forall(.natpred_1))]
>>    {move 0}
```

Here is the proof of a standard form of mathematical induction. Notice that the term Forall(indimp) representing the result of the induction step is expanded out into

$$\mathtt{Forall}([(\mathtt{n2}_3 : \mathtt{inNat}) => ((.\mathtt{natpred}_1(\mathtt{n2}_3) - > .\mathtt{natpred}_1(\mathtt{Succ}(\mathtt{n2}_3))) : \mathtt{prop})])).$$

in the declaration of Induction1, where the name indimp has passed out of scope: this is much closer to the more usual way of writing this as

$$(\forall n \in \mathtt{Nat} : \mathtt{natpred}(n) \rightarrow \mathtt{natpred}(\mathtt{Succ}(n))).$$

Induction1 generates instances of theorems proved by induction: Induction generates universally quantified theorems derived by induction. The meat of the proof lies in showing that the existence of a proof of Forall(indimp) yields a function taking proofs of natpred(n) to proofs of natpred(Succ(n)), which is what is required as input to Iteratep. The declaration of Induction is a nice example of the use as an argument of a function defined by giving another function a truncated argument list.

We think that it is interesting to contemplate the mathematical object presented as the referent of Induction1 in the Lestrade reply to its declaration.

It may seem odd that the induction step is the first argument rather than the basis step: the reason for this is that Lestrade can reliably read the hidden argument natpred from the induction step, but not so reliably from the basis step.

9 Definitions and basic axioms for addition and multiplication

In this section we define the notions of addition and multiplication and prove the usual Peano "axioms" governing these operations. No new axioms are actually required: addition and multiplication are defined by iterating suitable functions, and here natural numbers are entirely defined in terms of iteration of abstract functions.

```
Lestrade execution:


declare N1 in Nat

>> N1: in Nat {move 1}



declare N2 in Nat

>> N2: in Nat {move 1}



define + N1 N2 : Simpleiter Succ, N1 N2

>> +: [(N1_1:in Nat),(N2_1:in Nat) => (Simpleiter(Succ,
>>      N1_1,N2_1):in Nat)]
>>   {move 0}
```

The sum `N1 + N2` is defined as the result of iterating successor `N2` times starting at `N1`. The function `Succ1` is needed because the function iterated in the fully abstract case has an additional natural number argument which can qualify types. Note that Lestrade does not need to be told that the function `Tt` from natural numbers to types which is a hidden parameter of `Iterate` is here the constant function whose value is `Nat`: its type inference is smart enough to figure this out.

```
Lestrade execution:


define Addid N1: propfixform ((N1+0)=N1,Simpleinit \
```

```
   Succ, N1)

>> Addid: [(N1_1:in Nat) => ((((N1_1 + 0) =
>>       N1_1) propfixform Simpleinit(Succ,N1_1)):
>>       that ((N1_1 + 0) = N1_1))]
>>    {move 0}



define Additer N1 N2 : propfixform   ((N1 \
   + Succ N2)=Succ(N1 + N2), Simpleiterstep \
   Succ, N1 N2)

>> Additer: [(N1_1:in Nat),(N2_1:in Nat) =>
>>       ((((N1_1 + Succ(N2_1)) = Succ((N1_1 +
>>       N2_1))) propfixform Simpleiterstep(Succ,
>>       N1_1,N2_1)):that ((N1_1 + Succ(N2_1))
>>       = Succ((N1_1 + N2_1))))]
>>    {move 0}
```

Here the usual Peano axioms for addition are proved as instances of `Initialize` and `Iterstep`, the basic equations governing iteration. Note the use of `propfixform` to get the output types here to take the right surface form. Lestrade's matching facility is smart enough to recognize the actual expansions of the cases of `Initialize` and `Iterstep` as being equivalent by definition to the forms explicitly given as arguments.

```
Lestrade execution:



open

   declare n2 in Nat

>>     n2: in Nat {move 2}
```

```
   declare n3 in Nat

>>    n3: in Nat {move 2}



   define addenone n3: n3+N1

>>    addenone: [(n3_1:in Nat) => (---:in Nat)]
>>      {move 1}



   close

define * N1 N2 : Simpleiter addenone, 0, \
   N2

>> *: [(N1_1:in Nat),(N2_1:in Nat) => (Simpleiter([(n3_2:
>>         in Nat) => ((n3_2 + N1_1):in Nat)]
>>      ,0,N2_1):in Nat)]
>>   {move 0}



define Multzero N1 : propfixform ((N1*0)=0, \
   Simpleinit addenone, 0)

>> Multzero: [(N1_1:in Nat) => ((((N1_1 * 0)
>>      = 0) propfixform Simpleinit([(n3_2:in
>>         Nat) => ((n3_2 + N1_1):in Nat)]
>>      ,0)):that ((N1_1 * 0) = 0))]
>>   {move 0}
```

```
define Multiter N1 N2 : propfixform \
   ((N1*Succ N2)=(N1*N2)+N1, Simpleiterstep \
   addenone,0,N2)

>> Multiter: [(N1_1:in Nat),(N2_1:in Nat) =>
>>      ((((N1_1 * Succ(N2_1)) = ((N1_1 * N2_1)
>>      + N1_1)) propfixform Simpleiterstep([(n3_2:
>>         in Nat) => ((n3_2 + N1_1):in Nat)]
>>      ,0,N2_1)):that ((N1_1 * Succ(N2_1)) =
>>      ((N1_1 * N2_1) + N1_1)))]
>>   {move 0}
```

The development of multiplication is very similar to that of addition, subject to the additional complication that the operation "add `N1`" which is iterated has to be given a nonce name `addenone`, which has a dummy first natural number argument just as `Succ1` does above.

10 Addition is commutative

In this section, we prove from the axioms for addition given in the previous section that addition is commutative, narrating our motivations as we go.

```
Lestrade execution:


open

   declare M3 in Nat

>>    M3: in Nat {move 2}



   open

      declare N3 in Nat
```

```
>>       N3: in Nat {move 3}



      define commuteswithm N3 : (M3 + N3) \
         = N3 + M3

>>       commuteswithm: [(N3_1:in Nat) => (---:
>>            prop)]
>>         {move 2}



      close

   define commuteswithall M3 : Forall commuteswithm


>>    commuteswithall: [(M3_1:in Nat) => (---:
>>         prop)]
>>      {move 1}



   close
```

Open a working environment, in which we declare a natural number `M3`, and introduce the property of commuting with `M3`, and then the property of `M3` of commuting with every natural number.

We first show `commuteswithall 0` by induction.

```
Lestrade execution:

comment The basis step


define zerocommuteswithzero : Reflexeq (0+0)
```

```
>> zerocommuteswithzero: [(Reflexeq((0 + 0)):
>>      that ((0 + 0) = (0 + 0)))]
>>   {move 0}



open

   declare M3 in Nat

>>    M3: in Nat {move 2}



   open

      declare indhyp that (0 + M3) = M3 + \
         0

>>       indhyp: that ((0 + M3) = (M3 + 0))
>>         {move 3}



      define commzero1 : Additer 0 M3

>>       commzero1: [(---:that ((0 + Succ(M3))
>>            = Succ((0 + M3))))]
>>         {move 2}



      define commzero2 indhyp : Substitution \
         indhyp commzero1

>>       commzero2: [(indhyp_1:that ((0 + M3)
```

```
>>            = (M3 + 0))) => (---:that ((0 +
>>            Succ(M3)) = Succ((M3 + 0))))]
>>         {move 2}
```

```
      define commzero3 : Addid M3

>>       commzero3: [(---:that ((M3 + 0) = M3))]
>>         {move 2}
```

```
      define commzero4 indhyp : Substitution \
         commzero3 commzero2 indhyp

>>       commzero4: [(indhyp_1:that ((0 + M3)
>>            = (M3 + 0))) => (---:that ((0 +
>>            Succ(M3)) = Succ(M3)))]
>>         {move 2}
```

```
      open

         declare M4 in Nat

>>          M4: in Nat {move 4}
```

```
         define noncepred M4 : (0 + Succ \
            M3)=M4

>>          noncepred: [(M4_1:in Nat) => (---:
>>               prop)]
>>            {move 3}
```

```
        close

      define commzero5 indhyp: Substitution0 \
         (noncepred,Eqsymm Addid Succ M3, commzero4 \
         indhyp)

>>       commzero5: [(indhyp_1:that ((0 + M3)
>>            = (M3 + 0))) => (---:that ((0 +
>>            Succ(M3)) = (Succ(M3) + 0)))]
>>         {move 2}



      close

   define indstep1 M3 : Deduction commzero5


>>    indstep1: [(M3_1:in Nat) => (---:that
>>         (((0 + M3_1) = (M3_1 + 0)) -> ((0 +
>>         Succ(M3_1)) = (Succ(M3_1) + 0))))]
>>      {move 1}



   close

define commzerobasisindstep : Ugen indstep1


>> commzerobasisindstep: [(Ugen([(M3_1:in Nat)
>>         => (Deduction([(indhyp_2:that ((0 +
>>            M3_1) = (M3_1 + 0))) => (Substitution0([(M4_3:
>>               in Nat) => (((0 + Succ(M3_1))
>>               = M4_3):prop)]
>>            ,Eqsymm(Addid(Succ(M3_1))),(Addid(M3_1)
```

```
>>           Substitution (indhyp_2 Substitution
>>           (0 Additer M3_1)))):that ((0 + Succ(M3_1))
>>           = (Succ(M3_1) + 0)))])
>>        :that (((0 + M3_1) = (M3_1 + 0)) ->
>>        ((0 + Succ(M3_1)) = (Succ(M3_1) + 0))))])
>>     :that Forall([(M3_6:in Nat) => ((((0 +
>>        M3_6) = (M3_6 + 0)) -> ((0 + Succ(M3_6))
>>        = (Succ(M3_6) + 0))):prop)]))
>>     ]
>>  {move 0}
```

```
define commzerobasis : Induction commzerobasisindstep \
   zerocommuteswithzero

>> commzerobasis: [((commzerobasisindstep Induction
>>     zerocommuteswithzero):that Forall([(M3_2:
>>        in Nat) => (((0 + M3_2) = (M3_2 + 0)):
>>        prop)]))
>>     ]
>>  {move 0}
```

We have now proved the basis step (commutativity of addition with zero). We commence the induction step.

```
Lestrade execution:


declare M3 in Nat

>> M3: in Nat {move 1}



open
```

```
   declare commindhyp that commuteswithall \
      M3

>>    commindhyp: that commuteswithall(M3) {move
>>      2}
```

```
   open

      declare N3 in Nat

>>       N3: in Nat {move 3}
```

```
      define commind1 N3 : Reflexeq (Succ \
         M3 + N3)

>>       commind1: [(N3_1:in Nat) => (---:that
>>            ((Succ(M3) + N3_1) = (Succ(M3) +
>>            N3_1)))]
>>         {move 2}
```

At this point we pause and remark that we immediately need the lemma $\sigma(m) + m = \sigma(m + n)$. We prove the lemma inline right here.

```
Lestrade execution:
```

```
      define commindlemma1 : Addid Succ M3


>>       commindlemma1: [(---:that ((Succ(M3)
>>            + 0) = Succ(M3)))]
```

```
>>        {move 2}



      open

         declare N4 in Nat

>>          N4: in Nat {move 4}



         define noncepred N4 :( Succ M3 +0)= \
            Succ N4

>>          noncepred: [(N4_1:in Nat) => (---:
>>               prop)]
>>            {move 3}



         close

      define commindlemma2:         Substitution0(noncepred, \
         Eqsymm (Addid         M3),commindlemma1)


>>       commindlemma2: [(---:that ((Succ(M3)
>>            + 0) = Succ((M3 + 0))))]
>>         {move 2}
```

The object `commindlemma2` is evidence for the basis of the lemma.

```
Lestrade execution:
```

```
open

   declare commindlemmaindhyp \
      that (Succ M3 + N3) = Succ(M3 + \
      N3)

>>          commindlemmaindhyp: that ((Succ(M3)
>>            + N3) = Succ((M3 + N3))) {move 4}



   define commindlemma3 : Additer Succ \
      M3 N3

>>          commindlemma3: [(---:that ((Succ(M3)
>>               + Succ(N3)) = Succ((Succ(M3)
>>               + N3))))]
>>            {move 3}



   define commindlemma4 commindlemmaindhyp \
      : Substitution            commindlemmaindhyp \
      commindlemma3

>>          commindlemma4: [(commindlemmaindhyp_1:
>>               that ((Succ(M3) + N3) = Succ((M3
>>               + N3)))) => (---:that ((Succ(M3)
>>               + Succ(N3)) = Succ(Succ((M3 +
>>               N3)))))]
>>            {move 3}



   define commindlemma5 commindlemmaindhyp \
      : Substitution            (Eqsymm(Additer \
      M3 N3),            commindlemma4 \
      commindlemmaindhyp)
```

```
>>          commindlemma5: [(commindlemmaindhyp_1:
>>               that ((Succ(M3) + N3) = Succ((M3
>>               + N3)))) => (---:that ((Succ(M3)
>>               + Succ(N3)) = Succ((M3 + Succ(N3)))))]
>>            {move 3}
```

```
   close

define commindlemma6 N3 : Deduction \
   (commindlemma5)
```

```
>>       commindlemma6: [(N3_1:in Nat) => (---:
>>            that (((Succ(M3) + N3_1) = Succ((M3
>>            + N3_1))) -> ((Succ(M3) + Succ(N3_1))
>>            = Succ((M3 + Succ(N3_1))))))]
>>         {move 2}
```

```
define commindlemma7 : Ugen commindlemma6
```

```
>>       commindlemma7: [(---:that Forall([(N3_2:
>>               in Nat) => ((((Succ(M3) + N3_2)
>>               = Succ((M3 + N3_2))) -> ((Succ(M3)
>>               + Succ(N3_2)) = Succ((M3 + Succ(N3_2))))):
>>               prop)]))
>>            ]
>>         {move 2}
```

```
define commindlemma : Induction commindlemma7, \
   commindlemma2
```

```
>>        commindlemma: [(---:that Forall([(N3_2:
>>                in Nat) => (((Succ(M3) + N3_2)
>>                = Succ((M3 + N3_2))):prop)]))
>>             ]
>>          {move 2}
```

```
      open

         declare M4 in Nat
```

```
>>          M4: in Nat {move 4}
```

```
         define noncepred2 M4 : (Succ M3 \
            + N3)=M4
```

```
>>          noncepred2: [(M4_1:in Nat) => (---:
>>               prop)]
>>            {move 3}
```

```
         close

      define commind2 N3 : Substitution0(noncepred2, \
         Uinst commindlemma N3,commind1 N3)
```

```
>>        commind2: [(N3_1:in Nat) => (---:that
>>             ((Succ(M3) + N3_1) = Succ((M3 +
>>             N3_1))))]
>>          {move 2}
```

```
      define commind3 N3 : Substitution(Uinst \
         commindhyp N3,commind2 N3)

>>       commind3: [(N3_1:in Nat) => (---:that
>>            ((Succ(M3) + N3_1) = Succ((N3_1
>>            + M3))))]
>>         {move 2}



      define commind4 N3 : Substitution(Eqsymm(Additer \
         N3 M3),commind3 N3)

>>       commind4: [(N3_1:in Nat) => (---:that
>>            ((Succ(M3) + N3_1) = (N3_1 + Succ(M3))))]
>>         {move 2}



      close

   define commind5 commindhyp :      propfixform(commuteswithall \
      (Succ M3),      Ugen commind4)

>>    commind5: [(commindhyp_1:that commuteswithall(M3))
>>         => (---:that commuteswithall(Succ(M3)))]
>>      {move 1}



   close

define commind6 M3:Deduction commind5

>> commind6: [(M3_1:in Nat) => (Deduction([(commindhyp_4:
>>         that Forall([(N3_5:in Nat) => (((M3_1
>>            + N3_5) = (N3_5 + M3_1)):prop)]))
>>         => ((Forall([(N3_6:in Nat) => (((Succ(M3_1)
```

```
>>             + N3_6) = (N3_6 + Succ(M3_1))):prop)])
>>          propfixform Ugen([(N3_8:in Nat) =>
>>             ((Eqsymm((N3_8 Additer M3_1)) Substitution
>>             ((commindhyp_4 Uinst N3_8) Substitution
>>             Substitution0([(M4_12:in Nat) =>
>>                (((Succ(M3_1) + N3_8) = M4_12):
>>                prop)]
>>             ,((Ugen([(N3_16:in Nat) => (Deduction([(commindlemmaindhyp_17:
>>                   that ((Succ(M3_1) + N3_16)
>>                   = Succ((M3_1 + N3_16)))) =>
>>                   ((Eqsymm((M3_1 Additer N3_16))
>>                   Substitution (commindlemmaindhyp_17
>>                   Substitution (Succ(M3_1) Additer
>>                   N3_16))):that ((Succ(M3_1)
>>                   + Succ(N3_16)) = Succ((M3_1
>>                   + Succ(N3_16)))))])
>>                :that (((Succ(M3_1) + N3_16)
>>                = Succ((M3_1 + N3_16))) -> ((Succ(M3_1)
>>                + Succ(N3_16)) = Succ((M3_1 +
>>                Succ(N3_16))))))])
>>             Induction Substitution0([(N4_20:
>>                in Nat) => (((Succ(M3_1) + 0)
>>                = Succ(N4_20)):prop)]
>>             ,Eqsymm(Addid(M3_1)),Addid(Succ(M3_1))))
>>             Uinst N3_8),Reflexeq((Succ(M3_1)
>>             + N3_8))))):that ((Succ(M3_1) +
>>             N3_8) = (N3_8 + Succ(M3_1))))]))
>>          :that Forall([(N3_21:in Nat) => (((Succ(M3_1)
>>             + N3_21) = (N3_21 + Succ(M3_1))):
>>             prop)]))
>>          ])
>>       :that (Forall([(N3_22:in Nat) => (((M3_1
>>          + N3_22) = (N3_22 + M3_1)):prop)])
>>       -> Forall([(N3_23:in Nat) => (((Succ(M3_1)
>>          + N3_23) = (N3_23 + Succ(M3_1))):prop)]))
>>       )]
>>    {move 0}
```

```
define commind7 : Ugen commind6

>> commind7: [(Ugen(commind6):that Forall([(M3_4:
>>         in Nat) => ((Forall([(N3_5:in Nat)
>>            => (((M3_4 + N3_5) = (N3_5 + M3_4)):
>>            prop)])
>>         -> Forall([(N3_6:in Nat) => (((Succ(M3_4)
>>            + N3_6) = (N3_6 + Succ(M3_4))):prop)]))
>>         :prop)]))
>>      ]
>>   {move 0}
```

```
define Addcomm : Induction commind7 commzerobasis


>> Addcomm: [((commind7 Induction commzerobasis):
>>      that Forall([(M3_3:in Nat) => (Forall([(N3_4:
>>            in Nat) => (((M3_3 + N3_4) = (N3_4
>>            + M3_3)):prop)])
>>         :prop)]))
>>      ]
>>   {move 0}
```

```
declare term1 in Nat

>> term1: in Nat {move 1}
```

```
declare term2 in Nat

>> term2: in Nat {move 1}
```

```
define Addcomm2 term1 term2 : Uinst(Uinst \
   Addcomm term1,term2)

>> Addcomm2: [(term1_1:in Nat),(term2_1:in Nat)
>>      => (((Addcomm Uinst term1_1) Uinst term2_1):
>>      that ((term1_1 + term2_1) = (term2_1 +
>>      term1_1)))]
>>   {move 0}
```

At this point the commutativity of addition is proved. The method of proof is entirely standard. Moreover, it is not nearly as verbose as the length of the text above would seem to suggest: the correct measure is the length of the text consisting only of user-entered lines. These lines are closely analogous to the lines in a usual proof of this result from the axioms of Peano arithmetic, complicated by a fine-grained approach to application of rules and careful notation of dependencies and levels of hypothesis.

We shall probably clean up this proof, with attention to better use of namespace and better mnemonics for proof line objects.

11 Power set types introduced

```
Lestrade execution:


postulate setsof T: type

>> setsof: [(T_1:type) => (---:type)]
>>   {move 0}



postulate setof tpred: in setsof T
```

```
>> setof: [(.T_1:type),(tpred_1:[(t1_2:in .T_1)
>>         => (---:prop)])
>>      => (---:in setsof(.T_1))]
>>   {move 0}
```

A more usual notation for `setsof T` might be $\mathcal{P}(T)$, the "power set type" of T. The terminology here relates to the conceptual abuse confusing a type T with the set of its elements. The more usual mathematical notation for `setof tpred` would be $\{x \in T : \mathtt{tpred}(x)\}$, subject to the same remark about abuse of terminology for types and sets.

```
Lestrade execution:


declare t6 in T

>> t6: in T {move 1}



declare s6 in setsof T

>> s6: in setsof(T) {move 1}



postulate E t6 s6 : prop

>> E: [(.T_1:type),(t6_1:in .T_1),(s6_1:in setsof(.T_1))
>>      => (---:prop)]
>>   {move 0}
```

We declare the membership relation.

```
Lestrade execution:


declare elementev1 that tpred t6

>> elementev1: that tpred(t6) {move 1}



declare elementev2 that t6 E setof tpred


>> elementev2: that (t6 E setof(tpred)) {move
>>   1}



postulate Comprehension10 tpred, t6 elementev1 \
   that t6 E setof tpred

>> Comprehension10: [(.T_1:type),(tpred_1:[(t1_2:
>>         in .T_1) => (---:prop)]),
>>      (t6_1:in .T_1),(elementev1_1:that tpred_1(t6_1))
>>      => (---:that (t6_1 E setof(tpred_1)))]
>>   {move 0}



define Comprehension11 tpred, elementev1 \
   : Comprehension10 tpred, t6 elementev1

>> Comprehension11: [(.T_1:type),(tpred_1:[(t1_2:
>>         in .T_1) => (---:prop)]),
>>      (.t6_1:in .T_1),(elementev1_1:that tpred_1(.t6_1))
>>      => (Comprehension10(tpred_1,.t6_1,elementev1_1):
>>      that (.t6_1 E setof(tpred_1)))]
>>   {move 0}
```

```
define Comprehension12 t6 elementev1 : Comprehension10 \
   tpred, t6 elementev1

>> Comprehension12: [(.T_1:type),(t6_1:in .T_1),
>>      (.tpred_1:[(t1_2:in .T_1) => (---:prop)]),
>>      (elementev1_1:that .tpred_1(t6_1)) =>
>>      (Comprehension10(.tpred_1,t6_1,elementev1_1):
>>      that (t6_1 E setof(.tpred_1)))]
>>   {move 0}
```

```
postulate Comprehension2 elementev2 that \
   tpred t6

>> Comprehension2: [(.T_1:type),(.t6_1:in .T_1),
>>      (.tpred_1:[(t1_2:in .T_1) => (---:prop)]),
>>      (elementev2_1:that (.t6_1 E setof(.tpred_1)))
>>      => (---:that .tpred_1(.t6_1))]
>>   {move 0}
```

We implement the comprehension axiom, the equivalence of

$$a \in \{x \in T : \texttt{tpred}(x)\}$$

and $\texttt{tpred}(a)$, via the declaration of the functions `Comprehension1x` (where `x` is 0,1,2) and `Comprehension2`.

`Lestrade execution:`

```
open

   declare t5 in T
```

```
>>    t5: in T {move 2}



   postulate tpred1 t5 prop

>>    tpred1: [(t5_1:in T) => (---:prop)]
>>      {move 1}



   postulate tpred2 t5 prop

>>    tpred2: [(t5_1:in T) => (---:prop)]
>>      {move 1}



   declare tpredev1 that tpred1 t5

>>    tpredev1: that tpred1(t5) {move 2}



   declare tpredev2 that tpred1 t5

>>    tpredev2: that tpred1(t5) {move 2}



   postulate ext1 tpredev1 : that tpred2 \
      t5

>>    ext1: [(.t5_1:in T),(tpredev1_1:that tpred1(.t5_1))
>>         => (---:that tpred2(.t5_1))]
>>      {move 1}
```

```
   postulate ext2 tpredev2 : that tpred1 \
      t5

>>    ext2: [(.t5_1:in T),(tpredev2_1:that tpred1(.t5_1))
>>         => (---:that tpred1(.t5_1))]
>>      {move 1}



   close

postulate Extensionality ext1, ext2 : \
   that (setof tpred1) = setof tpred2

>> Extensionality: [(.T_1:type),(.tpred1_1:[(t5_2:
>>         in .T_1) => (---:prop)]),
>>      (.tpred2_1:[(t5_3:in .T_1) => (---:prop)]),
>>      (ext1_1:[(.t5_4:in .T_1),(tpredev1_4:that
>>         .tpred1_1(.t5_4)) => (---:that .tpred2_1(.t5_4))]),
>>      (ext2_1:[(.t5_5:in .T_1),(tpredev2_5:that
>>         .tpred1_1(.t5_5)) => (---:that .tpred1_1(.t5_5))])
>>      => (---:that (setof(.tpred1_1) = setof(.tpred2_1)))]
>>   {move 0}



declare s7 in setsof T

>> s7: in setsof(T) {move 1}



open

   declare t5 in T

>>    t5: in T {move 2}
```

```
   define elementpred t5 : t5 E s7

>>    elementpred: [(t5_1:in T) => (---:prop)]
>>      {move 1}



   close

postulate Extensionality2 s7 that s7 = setof \
   elementpred

>> Extensionality2: [(.T_1:type),(s7_1:in setsof(.T_1))
>>      => (---:that (s7_1 = setof([(t5_2:in .T_1)
>>         => ((t5_2 E s7_1):prop)]))
>>      )]
>>   {move 0}
```

The functions Extensionality1 and Extensionality2 implement the axiom of extensionality. There is something to note about how this is done (and we ought to prove some theorems later to show equivalence of this approach to other possible approaches). In effect, we postulate equivalence of $\{x \in T : \texttt{tpred}(x)\} = \{x \in T : \texttt{tpred}(x)\}$ and $(\forall x : \texttt{tpred}(x) \leftrightarrow \texttt{tpred}(x))$: this is what Extensionality1 does. To get full extensionality in the usual sense, we also postulate $S = \{x \in T : x \in S\}$ (this is what Extensionality2 does): for each S of type $\mathcal{P}(T)$: this prevents existence of additional objects of type $\mathcal{P}(T)$ with the same extension as sets defined in the usual way using set builder notation from predicates, but not themselves defined using set builder notation.

We have a philosophical reason for taking this approach. We have general metaphysical reasons for avoiding conflation of functions and objects, on which we may expand later. The function setof enables implementation of predicates of objects of type T (functions from T to prop) as objects of type $\mathcal{P}(T)$: `Extensionality1` thus expresses identity criteria for predicates (indirectly). It can be further noted that it is perfectly possible to define an equality predicate directly on the function sort of predicates of type T objects, and explicitly state extensional identity criteria for such functions, and we may do this later. But in any event, we regard the assertion of identity criteria for predicates implemented as objects of a power set type as distinguishable from the assertion that all objects of the power set type actually are implementations of predicates.

A theory of sets as untyped mathematical objects (in sort `obj`) could be implemented similarly, and we may present this later.

12 Naive set theory and Russell's paradox (without even using negation!)

In this section we develop naive set theory (in which any property of untyped mathematical objects defines a set, and sets are untyped mathematical objects) and develop something like the paradox of Russell. The way in which we do this is a little strange since we do not have negation yet, but implication is enough: the function `Russell` which is our final product takes any proposition A and returns a proof of A: the existence of a such a function would at the very least make mathematics uninteresting.

```
Lestrade execution:


declare ao obj

>> ao: obj {move 1}



declare bo obj

>> bo: obj {move 1}
```

```
open

   declare xo obj

>>    xo: obj {move 2}



   postulate opred xo prop

>>    opred: [(xo_1:obj) => (---:prop)]
>>      {move 1}



   close

postulate osetof opred obj

>> osetof: [(opred_1:[(xo_2:obj) => (---:prop)])
>>      => (---:obj)]
>>   {move 0}
```

We introduce the set builder operation `osetof` which takes a predicate of untyped objects to an untyped object.

```
Lestrade execution:


postulate Eo ao bo prop

>> Eo: [(ao_1:obj),(bo_1:obj) => (---:prop)]
>>   {move 0}
```

```
declare oelementev1 that ao Eo osetof opred


>> oelementev1: that (ao Eo osetof(opred)) {move
>>   1}



declare oelementev2 that opred ao

>> oelementev2: that opred(ao) {move 1}



postulate Ocomp1 oelementev1 that opred ao


>> Ocomp1: [(.ao_1:obj),(.opred_1:[(xo_2:obj)
>>         => (---:prop)]),
>>      (oelementev1_1:that (.ao_1 Eo osetof(.opred_1)))
>>      => (---:that .opred_1(.ao_1))]
>>   {move 0}



postulate Ocomp2 ao opred, oelementev2 that \
   ao Eo osetof opred

>> Ocomp2: [(ao_1:obj),(opred_1:[(xo_2:obj)
>>         => (---:prop)]),
>>      (oelementev2_1:that opred_1(ao_1)) =>
>>      (---:that (ao_1 Eo osetof(opred_1)))]
>>   {move 0}
```

We introduce the membership relation `Eo` and the two functions implementing its comprehension axiom, which are precisely analogous to the functions implementing the comprehension scheme in typed set theory above.

Lestrade execution:

```
open

   declare yo obj

>>    yo: obj {move 2}



   define R yo : (yo Eo yo) -> A

>>    R: [(yo_1:obj) => (---:prop)]
>>      {move 1}



   close

define r A : osetof R

>> r: [(A_1:prop) => (osetof([(yo_2:obj) =>
>>          (((yo_2 Eo yo_2) -> A_1):prop)])
>>      :obj)]
>>   {move 0}
```

This is our paradoxical set `r(A)` , which we would write in ordinary notation as $\{x : x \in x \to A\}$.

Lestrade execution:

```
open
```

```
   declare rhyp that (r A) Eo r A

>>    rhyp: that (r(A) Eo r(A)) {move 2}
```

```
   define rstep1 rhyp: Ocomp1 rhyp

>>    rstep1: [(rhyp_1:that (r(A) Eo r(A)))
>>         => (---:that ((r(A) Eo r(A)) -> A))]
>>      {move 1}
```

```
   define rstep2 rhyp: Mp rhyp (rstep1 rhyp)


>>    rstep2: [(rhyp_1:that (r(A) Eo r(A)))
>>         => (---:that A)]
>>      {move 1}
```

```
   define rstep3 rhyp: Deduction rstep2

>>    rstep3: [(rhyp_1:that (r(A) Eo r(A)))
>>         => (---:that ((r(A) Eo r(A)) -> A))]
>>      {move 1}
```

```
   define rstep4 rhyp: Mp rhyp rstep3 rhyp


>>    rstep4: [(rhyp_1:that (r(A) Eo r(A)))
>>         => (---:that A)]
>>      {move 1}
```

```
   close

define Russell1 A : Deduction rstep4

>> Russell1: [(A_1:prop) => (Deduction([(rhyp_2:
>>         that (r(A_1) Eo r(A_1))) => ((rhyp_2
>>         Mp Deduction([(rhyp_3:that (r(A_1)
>>            Eo r(A_1))) => ((rhyp_3 Mp Ocomp1(rhyp_3)):
>>            that A_1)]))
>>         :that A_1)])
>>      :that ((r(A_1) Eo r(A_1)) -> A_1))]
>>   {move 0}



define Ocomp22 ao oelementev2 : Ocomp2 ao \
   opred, oelementev2

>> Ocomp22: [(ao_1:obj),(.opred_1:[(xo_2:obj)
>>         => (---:prop)]),
>>      (oelementev2_1:that .opred_1(ao_1)) =>
>>      (Ocomp2(ao_1,.opred_1,oelementev2_1):that
>>      (ao_1 Eo osetof(.opred_1)))]
>>   {move 0}



define Russell2 A: propfixform ((r   A) \
   Eo r A,Ocomp22 ((r A),(Russell1 A)))

>> Russell2: [(A_1:prop) => (((r(A_1) Eo r(A_1))
>>      propfixform (r(A_1) Ocomp22 Russell1(A_1))):
>>      that (r(A_1) Eo r(A_1)))]
>>   {move 0}
```

```
define Russell A: Mp (Russell2 A, Russell1 \
   A)

>> Russell: [(A_1:prop) => ((Russell2(A_1) Mp
>>      Russell1(A_1)):that A_1)]
>>   {move 0}
```

The argument here is perfectly mad, of course. We review it since this is not the form usually given.

Let R denote the set $\{x : x \in x \rightarrow A\}$.

Our goal is to prove $R \in R$. To prove $R \in R$, that is $R \in \{x \in x \rightarrow A\}$, it suffices to prove $R \in R \rightarrow A$.

Suppose $R \in R$ for the sake of argument. Our goal is A. $R \in R$ as already noted is equivalent to $R \in R \rightarrow A$. Modus ponens gives us our goal A, so we have established $R \in R \rightarrow A$ by deduction, and so we have established $R \in R$, as already discussed.

Since we have both $R \in R$ and $R \in R \rightarrow A$, we have A by modus ponens.

But A was any proposition at all.

A Lestrade technicality to note is that it was convenient to introduce a version `Ocomp22` of `Ocomp2` which did not take an explicit predicate argument.

One should always have something philosophical to say after introducing something reputed to be a paradox, a threat to the foundations of reason. Our remark is that one should look carefully at the hypotheses before concluding that the foundations of reason are threatened. The Lestrade framework does nothing to encourage us to think it likely that the function sort of predicates of objects of sort `obj` can be embedded into the sort `obj` itself. The proof simply shows that this cannot be done (in the presence of implication, at any rate).

13 Constructive forms of negation, disjunction, and the existential quantifier

We resume the development of logical primitives. Here we give the constructive rules for negation, disjunction and existential quantification.

```
Lestrade execution:


postulate ?? prop

>> ??: prop {move 0}



declare absurd that ??

>> absurd: that ?? {move 1}



declare Dd prop

>> Dd: prop {move 1}



postulate Panic absurd Dd that Dd

>> Panic: [(absurd_1:that ??),(Dd_1:prop) =>
>>      (---:that Dd_1)]
>>   {move 0}
```

We introduce the false statement `??` and introduce the rule that any proposition may be deduced from a false statement.

Lestrade execution:

```
define ~ Dd : Dd -> ??

>> ~: [(Dd_1:prop) => ((Dd_1 -> ??):prop)]
>>   {move 0}
```

We define negation.

Lestrade execution:

```
postulate v A B prop

>> v: [(A_1:prop),(B_1:prop) => (---:prop)]
>>   {move 0}



postulate Addition1 B aa that A v B

>> Addition1: [(B_1:prop),(.A_1:prop),(aa_1:
>>      that .A_1) => (---:that (.A_1 v B_1))]
>>   {move 0}



postulate Addition2 A bb that A v B

>> Addition2: [(A_1:prop),(.B_1:prop),(bb_1:
>>      that .B_1) => (---:that (A_1 v .B_1))]
>>   {move 0}
```

```
declare cases that A v B

>> cases: that (A v B) {move 1}



open

   declare aa1 that A

>>    aa1: that A {move 2}



   declare bb1 that B

>>    bb1: that B {move 2}



   postulate case1 aa1 that Dd

>>    case1: [(aa1_1:that A) => (---:that Dd)]
>>      {move 1}



   postulate case2 bb1 that Dd

>>    case2: [(bb1_1:that B) => (---:that Dd)]
>>      {move 1}



   close

postulate Cases cases, case1, case2 that \
   Dd
```

```
>> Cases: [(.A_1:prop),(.B_1:prop),(cases_1:
>>      that (.A_1 v .B_1)),(.Dd_1:prop),(case1_1:
>>      [(aa1_2:that .A_1) => (---:that .Dd_1)]),
>>      (case2_1:[(bb1_3:that .B_1) => (---:that
>>         .Dd_1)])
>>      => (---:that .Dd_1)]
>>   {move 0}
```

We introduce disjunction and its basic rules, addition and proof by cases.

```
Lestrade execution:


postulate Exists tpred prop

>> Exists: [(.T_1:type),(tpred_1:[(t1_2:in .T_1)
>>         => (---:prop)])
>>      => (---:prop)]
>>   {move 0}



declare existsev that tpred t

>> existsev: that tpred(t) {move 1}



postulate Egen0 tpred, t existsev : that \
   Exists tpred

>> Egen0: [(.T_1:type),(tpred_1:[(t1_2:in .T_1)
>>         => (---:prop)]),
>>      (t_1:in .T_1),(existsev_1:that tpred_1(t_1))
>>      => (---:that Exists(tpred_1))]
```

```
>>   {move 0}



define Egen1 t existsev : Egen0 tpred, t \
   existsev

>> Egen1: [(.T_1:type),(t_1:in .T_1),(.tpred_1:
>>      [(t1_2:in .T_1) => (---:prop)]),
>>      (existsev_1:that .tpred_1(t_1)) => (Egen0(.tpred_1,
>>      t_1,existsev_1):that Exists(.tpred_1))]
>>   {move 0}



define Egen2 tpred, existsev : Egen0 tpred, \
   t existsev

>> Egen2: [(.T_1:type),(tpred_1:[(t1_2:in .T_1)
>>         => (---:prop)]),
>>      (.t_1:in .T_1),(existsev_1:that tpred_1(.t_1))
>>      => (Egen0(tpred_1,.t_1,existsev_1):that
>>      Exists(tpred_1))]
>>   {move 0}



declare existsev2 that Exists tpred

>> existsev2: that Exists(tpred) {move 1}



open

   declare witness in T

>>    witness: in T {move 2}
```

```
   declare witnessev that tpred witness

>>    witnessev: that tpred(witness) {move 2}



   postulate witnessprf witnessev that Dd


>>    witnessprf: [(.witness_1:in T),(witnessev_1:
>>         that tpred(.witness_1)) => (---:that
>>         Dd)]
>>      {move 1}



   close

postulate Einst existsev2, witnessprf that \
   Dd

>> Einst: [(.T_1:type),(.tpred_1:[(t1_2:in .T_1)
>>         => (---:prop)]),
>>      (existsev2_1:that Exists(.tpred_1)),(.Dd_1:
>>      prop),(witnessprf_1:[(.witness_3:in .T_1),
>>         (witnessev_3:that .tpred_1(.witness_3))
>>         => (---:that .Dd_1)])
>>      => (---:that .Dd_1)]
>>   {move 0}
```

We introduce the existential quantifier and its basic rules. At this point we have introduced all operations and rules of constructive (intuitionist) logic.

Note that two different additional versions of existential instantiation with different choices of explicit arguments are given.

14 Classical logic completed with double negation. Proofs of some classical theorems.

```
Lestrade execution:


declare maybe that ~ ~ A

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



postulate Dneg maybe that A

>> Dneg: [(.A_1:prop),(maybe_1:that ~(~(.A_1)))
>>      => (---:that .A_1)]
>>   {move 0}



open

   declare nega1 that ~Dd

>>    nega1: that ~(Dd) {move 2}



   define howler nega1 :absurd

>>    howler: [(nega1_1:that ~(Dd)) => (---:
>>         that ??)]
>>      {move 1}
```

```
   close

define Panic0 absurd Dd: Dneg(Deduction howler)


>> Panic0: [(absurd_1:that ??),(Dd_1:prop) =>
>>      (Dneg(Deduction([(nega1_2:that ~(Dd_1))
>>         => (absurd_1:that ??)]))
>>      :that Dd_1)]
>>   {move 0}
```

We introduce the rule of double negation $\neg\neg P \vdash P$, and we show that the constructive rule `Panic` can be implemented using `Dneg`.

What follows below is the full proof of the classically valid equivalence of $\neg A \to B$ and $A \vee B$, which we ought to comment line by line with a parallel proof in English. Notice how indentation in Lestrade output signals the depth of the nest of environments one is working in.

```
Lestrade execution:


open

   declare side1 that (~A) -> B

>>    side1: that (~(A) -> B) {move 2}
```

Suppose that $\neg A \to B$. Our aim is to prove $A \vee B$.

```
Lestrade execution:
```

```
   open

      declare contrahyp that ~(A v B)

>>       contrahyp: that ~((A v B)) {move 3}
```

Our strategy for proving $A \vee B$ is to suppose $\neg(A \vee B)$ and reason to a contradiction.

```
Lestrade execution:


      open

         declare howabouta that A

>>          howabouta: that A {move 4}



         define noa1 howabouta : Mp (Addition1 \
            B howabouta,contrahyp)

>>          noa1: [(howabouta_1:that A) => (---:
>>               that ??)]
>>            {move 3}



         close

      define thusnota contrahyp: propfixform(~A, \
         Deduction noa1)

>>       thusnota: [(contrahyp_1:that ~((A v
```

```
>>            B))) => (---:that ~(A))]
>>         {move 2}
```

In the block of text above we prove $\neg A$ from the local hypotheses. The strategy is to suppose that A, deduce $A \vee B$ from this by the rule of addition, then note the contradiction with the assumption $\neg A \vee B$ made above. To follow this, it is useful to recall that the deduction of a contradiction when we have both X and $\neg X$ is actually an instance of *modus ponens*, since $\neg X$ is defined as $X \rightarrow \perp$.

```
Lestrade execution:


      define thusb contrahyp: Mp (thusnota \
         contrahyp,side1)

>>       thusb: [(contrahyp_1:that ~((A v B)))
>>            => (---:that B)]
>>         {move 2}



      define thusaorb contrahyp: Addition2 \
         A thusb contrahyp

>>       thusaorb: [(contrahyp_1:that ~((A v
>>            B))) => (---:that (A v B))]
>>         {move 2}



      define thuscontra1 contrahyp: Mp (thusaorb \
         contrahyp,contrahyp)

>>       thuscontra1: [(contrahyp_1:that ~((A
>>            v B))) => (---:that ??)]
```

```
>>         {move 2}
```

In the three lines above we deduce a contradiction: we first deduce B by modus ponens from previous lines $\neg A$ and $\neg A \rightarrow B$, then we deduce $A \vee B$ from B by the rule of addition, then we obtain a contradiction.

Lestrade execution:

```
      close

   define classicalor1 side1 : Dneg(Deduction \
      thuscontra1)

>>    classicalor1: [(side1_1:that (~(A) ->
>>         B)) => (---:that (A v B))]
>>      {move 1}
```

Applying `Deduction` to the function `thuscontra1` above gives a proof that $\neg\neg(A \vee B)$. Applying `Dneg` to this gives a proof of $A \vee B$. What we have actually done is constructed a function from the original assumption that $\neg A \rightarrow B$ to evidence that $A \vee B$.

Lestrade execution:

```
   declare side2 that A v B

>>    side2: that (A v B) {move 2}
```

Now we assume that $A \vee B$ and argue to the conclusion $\neg A \rightarrow B$.

Lestrade execution:

```
   open

      declare ahyp1 that ~A

>>       ahyp1: that ~(A) {move 3}
```

We assume $\neg A$ and our goal is now B. Our strategy is to prove this by cases on our hypothesis $A \vee B$, first showing that B follows from A, then showing that B follows from B.

Lestrade execution:

```
      open

         declare ifa2 that A

>>          ifa2: that A {move 4}



         define ifa21 ifa2 : Mp ifa2 ahyp1


>>          ifa21: [(ifa2_1:that A) => (---:
>>               that ??)]
>>            {move 3}



         define ifa22 ifa2 : Panic (ifa21 \
            ifa2,B)
```

```
>>          ifa22: [(ifa2_1:that A) => (---:
>>               that B)]
>>            {move 3}
```

A function from proofs of A to proofs of B is defined: from a proof of A we get a proof of $\bot$ because we have a constant proof of $\neg A$ given. From a proof of $\bot$ we get a proof of anything, in particular B.

```
Lestrade execution:



         declare ifb2 that B

>>          ifb2: that B {move 4}




         define ifb21 ifb2 : ifb2

>>          ifb21: [(ifb2_1:that B) => (---:
>>               that B)]
>>            {move 3}
```

The identity function takes proofs of B to proofs of B.

```
Lestrade execution:



         close

      define thusb2 ahyp1 : Cases side2, \
         ifa22, ifb21

>>       thusb2: [(ahyp1_1:that ~(A)) => (---:
```

```
>>            that B)]
>>         {move 2}
```

We complete the proof of the conclusion B from the hypothesis $\neg A$ by cases outlined above.

```
Lestrade execution:


      close

   define classicalor2 side2 : Deduction \
      thusb2

>>    classicalor2: [(side2_1:that (A v B))
>>         => (---:that (~(A) -> B))]
>>      {move 1}



   close

define Classicalor1 A B : Deduction classicalor1


>> Classicalor1: [(A_1:prop),(B_1:prop) => (Deduction([(side1_2:
>>         that (~(A_1) -> B_1)) => (Dneg(Deduction([(contrahyp_3:
>>            that ~((A_1 v B_1))) => (((A_1 Addition2
>>            ((~(A_1) propfixform Deduction([(howabouta_4:
>>               that A_1) => (((B_1 Addition1
>>               howabouta_4) Mp contrahyp_3):
>>               that ??)]))
>>            Mp side1_2)) Mp contrahyp_3):that
>>            ??)]))
>>         :that (A_1 v B_1))])
>>      :that ((~(A_1) -> B_1) -> (A_1 v B_1)))]
```

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

```
define Classicalor2 A B : Deduction classicalor2
```

```
>> Classicalor2: [(A_1:prop),(B_1:prop) => (Deduction([(side2_2:
>>         that (A_1 v B_1)) => (Deduction([(ahyp1_3:
>>            that ~(A_1)) => (Cases(side2_2,[(ifa2_4:
>>               that A_1) => (((ifa2_4 Mp ahyp1_3)
>>               Panic B_1):that B_1)]
>>            ,[(ifb2_5:that B_1) => (ifb2_5:that
>>               B_1)])
>>            :that B_1)])
>>         :that (~(A_1) -> B_1))])
>>      :that ((A_1 v B_1) -> (~(A_1) -> B_1)))]
>>   {move 0}
```

```
define Classicalor A B: propfixform \
   (((~A)->B)<->(A v B), Andproof (Classicalor1 \
   A B,Classicalor2 A B))

>> Classicalor: [(A_1:prop),(B_1:prop) => ((((~(A_1)
>>      -> B_1) <-> (A_1 v B_1)) propfixform ((A_1
>>      Classicalor1 B_1) Andproof (A_1 Classicalor2
>>      B_1))):that ((~(A_1) -> B_1) <-> (A_1
>>      v B_1)))]
>>   {move 0}
```

Finally we exit to the outermost environment and prove our three theorems, two conditionals and a biconditional. The conditionals are proved by applying `Deduction` to the appropriate functions developed above, and the biconditional is proved using `Andproof`.

The following block of so far uncommented text proves the equivalence of $\neg(A \rightarrow B)$ and $A \wedge \neg B$ in the same style.

```
Lestrade execution:


open

   declare side1 that ~(A -> B)

>>    side1: that ~((A -> B)) {move 2}



   open

      declare nota that ~A

>>       nota: that ~(A) {move 3}



      open

         declare buta that A

>>          buta: that A {move 4}



         define step10 buta : Mp buta nota


>>          step10: [(buta_1:that A) => (---:
>>               that ??)]
>>            {move 3}
```

```
      define step20 buta : Panic (step10 \
         buta, B)

>>         step20: [(buta_1:that A) => (---:
>>              that B)]
>>           {move 3}



      close

   define athenb nota : Deduction step20


>>      athenb: [(nota_1:that ~(A)) => (---:
>>           that (A -> B))]
>>        {move 2}



   define iscontra nota : Mp (athenb nota, \
      side1)

>>      iscontra: [(nota_1:that ~(A)) => (---:
>>           that ??)]
>>        {move 2}



   close

define yesa side1 : Dneg(Deduction iscontra)


>>   yesa: [(side1_1:that ~((A -> B))) => (---:
>>        that A)]
>>      {move 1}
```

```
   open

      declare butb that B

>>       butb: that B {move 3}



      open

         declare supposea that A

>>          supposea: that A {move 4}



         define indeedb supposea : butb

>>          indeedb: [(supposea_1:that A) =>
>>               (---:that B)]
>>            {move 3}



         close

      define ahenceb butb : Deduction indeedb


>>       ahenceb: [(butb_1:that B) => (---:that
>>            (A -> B))]
>>         {move 2}
```

```
      define iscontra2 butb : Mp (ahenceb \
         butb,side1)

>>       iscontra2: [(butb_1:that B) => (---:
>>            that ??)]
>>         {move 2}



      close

   define notob side1 : propfixform(~B,Deduction \
      iscontra2)

>>    notob: [(side1_1:that ~((A -> B))) =>
>>         (---:that ~(B))]
>>      {move 1}



   define negimp1 side1 : Andproof(yesa side1, \
      notob side1)

>>    negimp1: [(side1_1:that ~((A -> B))) =>
>>         (---:that (A & ~(B)))]
>>      {move 1}



   declare side2 that A & ~B

>>    side2: that (A & ~(B)) {move 2}



   open

      declare ifathenb that A -> B
```

```
>>       ifathenb: that (A -> B) {move 3}


      define step11 ifathenb : Mp(Simplification1 \
         side2,ifathenb)

>>       step11: [(ifathenb_1:that (A -> B))
>>            => (---:that B)]
>>         {move 2}


      define step21 ifathenb :         Mp(step11 ifathenb,Simplification2

>>       step21: [(ifathenb_1:that (A -> B))
>>            => (---:that ??)]
>>         {move 2}


      close

   define negimp2 side2: propfixform(~(A \
      -> B),Deduction step21)

>>    negimp2: [(side2_1:that (A & ~(B))) =>
>>         (---:that ~((A -> B)))]
>>      {move 1}


   close

define Negimp1 A B : Deduction negimp1

>> Negimp1: [(A_1:prop),(B_1:prop) => (Deduction([(side1_2:
```

```
>>         that ~((A_1 -> B_1))) => ((Dneg(Deduction([(nota_3:
>>            that ~(A_1)) => ((Deduction([(buta_4:
>>               that A_1) => (((buta_4 Mp nota_3)
>>               Panic B_1):that B_1)])
>>            Mp side1_2):that ??)]))
>>         Andproof (~(B_1) propfixform Deduction([(butb_5:
>>            that B_1) => ((Deduction([(supposea_6:
>>               that A_1) => (butb_5:that B_1)])
>>            Mp side1_2):that ??)]))
>>         ):that (A_1 & ~(B_1)))])
>>      :that (~((A_1 -> B_1)) -> (A_1 & ~(B_1))))]
>>   {move 0}
```

```
define Negimp2 A B : Deduction negimp2

>> Negimp2: [(A_1:prop),(B_1:prop) => (Deduction([(side2_2:
>>         that (A_1 & ~(B_1))) => ((~((A_1 ->
>>         B_1)) propfixform Deduction([(ifathenb_3:
>>            that (A_1 -> B_1)) => (((Simplification1(side2_2)
>>            Mp ifathenb_3) Mp Simplification2(side2_2)):
>>            that ??)]))
>>         :that ~((A_1 -> B_1)))])
>>      :that ((A_1 & ~(B_1)) -> ~((A_1 -> B_1))))]
>>   {move 0}
```

```
define Negimp A B : propfixform((~(A   -> B))<->A & ~B, Andproof(Negimp1   A B,

>> Negimp: [(A_1:prop),(B_1:prop) => (((~((A_1
>>      -> B_1)) <-> (A_1 & ~(B_1))) propfixform
>>      ((A_1 Negimp1 B_1) Andproof (A_1 Negimp2
>>      B_1))):that (~((A_1 -> B_1)) <-> (A_1
>>      & ~(B_1))))]
>>   {move 0}
```

15 Basic declarations for a version of Quine's New Foundations

Lestrade execution:

```
postulate V type

>> V: type {move 0}



open

   declare Tt3 type

>>    Tt3: type {move 2}



   postulate typepred Tt3 prop

>>    typepred: [(Tt3_1:type) => (---:prop)]
>>      {move 1}



   close

declare typepredev1 that typepred V

>> typepredev1: that typepred(V) {move 1}
```

```
postulate Ambiguity typepredev1 that typepred \
   setsof V

>> Ambiguity: [(.typepred_1:[(Tt3_2:type) =>
>>         (---:prop)]),
>>      (typepredev1_1:that .typepred_1(V)) =>
>>      (---:that .typepred_1(setsof(V)))]
>>   {move 0}
```

This is a conjectural formulation of the simple theory of types with Specker's axiom scheme of Ambiguity, which is equiconsistent with Quine's New Foundations.

We first declare a type `V` as a primitive notion: this is type 0 in a model of the simple theory of types.

The idea is that we declare a function `Ambiguity` which will send evidence `typepredev1` that a predicate `typepred` of types holds of `V` to evidence that the same predicate holds of `setsof V`, type 1 of the same model.

We would want an inverse operation for `Ambiguity` as well if we did not have double negation.

The reason that it appears that this might work is that the primitives we have given seem to allow formulation of predicates of types only under very limited circumstances: basically the predicates of types that can be formulated are limited to assertions that formulas of the usual first order language of TST hold in the model of TST with `V` as type 0 (with the additional point that the universal applicability of our natural number type for indexing functions on different types may imply that consequences of the Axiom of Counting hold in our ambiguous type theory). We suspect that adding equality of types and quantification over types to this theory would lead to contradiction (and so it is important that quantification over the sort of type labels is not automatically supported by our framework). We intend to supply proofs of this point if we are able to postulate them.

Another point worth noting is that the "Axiom" of Infinity is provable in this system (without any use of Ambiguity) by use of the fact that our notion of iteration is applicable to any type using the same type of natural numbers. I'll supply a proof of this at some point.

16 The third and fourth Peano axioms

For the moment, just an outline. The third Peano axiom can be proved using the operation $A \mapsto A \cup V$ in any double power type. Applying this function 0 times to the empty set gives the empty set, and applying this function $n+1$ times for any n will give V, which is provably nonempty in a double power type, so $0 = n + 1$ is false.

The fourth Peano axiom is best shown by considering the type of natural numbers and the Frege natural numbers over the power type of the natural numbers. If numbers $1, \ldots, n$ are distinct, the Frege natural number containing $\{\{1\}, \ldots, \{n\}\}$ is the result of iterating the Frege successor operation n times on the Frege zero in the appropriate type. Now, the Frege natural number containing $\{\{1\}, \ldots, \{n\}, \emptyset\}$ is a new one, and the result of iterating the Frege successor operation $n+1$ times on the Frege zero, which establishes that $n \neq n+1$. This establishes that Infinity holds in the model of type theory based on the natural numbers, which is enough to show that Axiom 4 holds.

These are going to be tricky arguments with lots of preliminaries under Lestrade.

In the interpretation of NF, if the size of V is the result of applying the Frege successor operation n times to the Frege zero, than the same is true of $\mathcal{P}(V)$, and this is readily shown not to be true. The fact that the natural numbers are type free in this interpretation of NF (being defined in a way independent of the Frege natural numbers in each high enough type) suggests that the stratified consequences of the axiom of counting ought to hold.