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

Randall Holmes

4/17/2020: annotating for talk. If I like it, this might be the permanent version. 4/15/2020: I have enabled abstract iteration without rewriting. Further corrections to the final section.
4/14/2020: The text is revised (largely) to conform with the new implementation, though it does not exhaustively survey new features.
4/13/2020: revised to run under the reimplementation. Rewriting cannot be used, so the block about abstract iteration is not run. Substitution isn't working correctly for some reason. Only Lestrade code is executed: remarks about Lestrade still presume the old version.
The reason substitution didn't work had to do with the order in which its arguments were declared: the new version puts implicit arguments in in the actual order in which they appear, and this can force a need for a different order.
10/31/2017 10:30 am: Systematic introduction of terms with bound variables.

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.

```
begin Lestrade execution

   >>> declare A prop


   A : prop


   {move 1}

   >>> declare B prop


   B : prop
```

```
   {move 1}
end Lestrade execution
```

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 `>>>` followed by Lestrade command names such as `declare`, `postulate`, `define` are entered by the user. Other lines in these blocks are Lestrade responses to commands typed by the user.

```
begin 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}
end Lestrade execution
```

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.

```
begin Lestrade execution

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


   proptest : [(A_1 : prop), (B_1 : prop) =>
       ({def} (A_1 & B_1) -> A_1 : prop)]


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


   {move 0}
end Lestrade execution
```

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 infix (or mixfix) operations have the same precedence and group to the right if explicit parentheses are not provided, and unary operations bind more tightly than binary or infix operations.

```
begin Lestrade execution

   >>> open
```

```
      {move 2}

      >>> 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


   {move 1}
end Lestrade execution
```

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.

```
begin 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) =>
       ({def} ((C_1 & D_1) proptest D_1) ->
       C_1 : prop)]


   zorch : [(C_1 : prop), (D_1 : prop) =>
       (--- : prop)]


   {move 0}

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

```
   zorch2 : [(C_1 : prop), (D_1 : prop) =>
       ({def} (((C_1 & D_1) & D_1) ->
       C_1 & D_1) -> C_1 : prop)]


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


   {move 0}
end Lestrade execution
```

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.

```
begin Lestrade execution

   >>> declare aa that A


   aa : that A


   {move 1}

   >>> declare bb that B


   bb : that B
```

```
   {move 1}
end Lestrade execution
```

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.

```
begin 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), (.B_1
       : prop), (aa_1 : that .A_1), (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) =>
       ({def} aa_1 Andproof aa_1 : that .A_1
       & .A_1)]


   Selfand : [(.A_1 : prop), (aa_1 : that
       .A_1) => (--- : that .A_1 & .A_1)]


   {move 0}
end Lestrade execution
```

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. In fact, `Selfand` is a rule of inference, not a proof of a conditional, but it can be used to prove the conditional as we will see below.

```
begin 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}
end Lestrade execution
```

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.

```
begin Lestrade execution

   >>> declare cc that A -> B
```

```
   cc : that A -> B


   {move 1}

   >>> postulate Mp aa cc : that B


   Mp : [(.A_1 : prop), (.B_1 : prop), (aa_1
       : that .A_1), (cc_1 : that .A_1
       -> .B_1) => (--- : that .B_1)]


   {move 0}
end Lestrade execution
```

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.

```
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}


end Lestrade execution
```

Above is the old style implementation of the proof of the deduction theorem without variable binding terms.

```
begin Lestrade execution

   >>> declare aaa1 that A


   aaa1 : that A


   {move 1}

   >>> declare ded [aaa1 => that B]


   ded : [(aaa1_1 : that A) => (--- : that
       B)]


   {move 1}

   >>> postulate Deduction ded : that A -> \
       B


   Deduction : [(.A_1 : prop), (.B_1
       : prop), (ded_1 : [(aaa1_2 : that
          .A_1) => (--- : that .B_1)]) =>
       (--- : that .A_1 -> .B_1)]
```

```
   {move 0}
end Lestrade execution
```

This piece of code implements a standard strategy for proving implications, in the more compact style which variable binding terms allow (it is not necessary to open a new move to declare `ded` as in the code given above): 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).

```
open

     declare aaa that A

>>       aaa: that A {move 2}


     define selfand aaa : Andproof aaa aaa

>>       selfand: [(aaa_1:that A) => ((aaa_1
>>              Andproof aaa_1):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}

end Lestrade execution
```

A proof given in the old style, discussed below.

```
begin Lestrade execution

   >>> define Selfand2 A : Deduction [aaa1 \
          => Andproof aaa1 aaa1]


   Selfand2 : [(A_1 : prop) =>
       ({def} Deduction ([(aaa1_2 : that
          A_1) =>
          ({def} aaa1_2 Andproof aaa1_2 : that
          A_1 & A_1)]) : that A_1 -> A_1
       & A_1)]


   Selfand2 : [(A_1 : prop) => (--- : that
       A_1 -> A_1 & A_1)]


   {move 0}
end Lestrade execution
```

Here we actually prove the theorem $A \to (A \wedge A)$ for any proposition A, in a very compact form allowed by use of the term `[aaa1 => Andproof aaa1 aaa1]` for the function declared as `selfand` in the open/close block in the original proof. 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 $A \rightarrow A \wedge A$.

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 The Curry Howard isomorphism: defining type constructors analogous to propositional connectives

We recapitulate the basic declarations for the propositional connectives of conjunction and implication, and in parallel implement the type constructors which build Cartesian products and function spaces, along with the basic operations on the complex types. The analogy between the propositional constructions on the one hand and the type constructions on the other is known as the Curry-Howard isomorphism.

```
declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}

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}

end Lestrade execution
```

We declare the Cartesian product and function space constructors.

```
begin Lestrade execution
```

```
   >>> declare At type


   At : type


   {move 1}

   >>> declare Bt type


   Bt : type


   {move 1}

   >>> postulate X At Bt : type


   X : [(At_1 : type), (Bt_1 : type) =>
       (--- : type)]


   {move 0}

   >>> postulate ->> At Bt : type


   ->> : [(At_1 : type), (Bt_1 : type) =>
       (--- : type)]


   {move 0}
end Lestrade execution


declare aa that A
```

```
>> aa: that A {move 1}


declare bb that B

>> bb: that B {move 1}

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}

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}

end Lestrade execution
```

We introduce the pair operation and projection functions, which we see are formally analogous to the logical rules of conjunction and simplification.

```
begin Lestrade execution

   >>> declare aat in At


   aat : in At


   {move 1}

   >>> declare bbt in Bt


   bbt : in Bt


   {move 1}

   >>> postulate Pair aat bbt in At X Bt


   Pair : [(.At_1 : type), (.Bt_1 : type), (aat_1
       : in .At_1), (bbt_1 : in .Bt_1) =>
       (--- : in .At_1 X .Bt_1)]


   {move 0}

   >>> declare xxt in At X Bt


   xxt : in At X Bt


   {move 1}
```

```
   >>> postulate proj1 xxt in At


   proj1 : [(.At_1 : type), (.Bt_1 : type), (xxt_1
       : in .At_1 X .Bt_1) => (--- : in
       .At_1)]


   {move 0}

   >>> postulate proj2 xxt in Bt


   proj2 : [(.At_1 : type), (.Bt_1 : type), (xxt_1
       : in .At_1 X .Bt_1) => (--- : in
       .Bt_1)]


   {move 0}
end 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}

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}

end Lestrade execution
```

We introduce function application and the formation of function objects from functions (lambda abstraction), which we see are formally analogous to modus ponens and the deduction theorem. The arguments of function application are supplied in converse order to those of modus ponens (not because there is any virtue to this but because existing text below written earlier would have had to be extensively revised to reverse the order of the arguments of `Mp`)!)

```
begin Lestrade execution

   >>> declare cct in At ->> Bt


   cct : in At ->> Bt
```

```
{move 1}

>>> declare aat2 in At


aat2 : in At


{move 1}

>>> postulate Apply cct aat2 in Bt


Apply : [(.At_1 : type), (.Bt_1 : type), (cct_1
    : in .At_1 ->> .Bt_1), (aat2_1 : in
    .At_1) => (--- : in .Bt_1)]


{move 0}

>>> declare dedt [aat2 => in Bt]


dedt : [(aat2_1 : in At) => (--- : in
    Bt)]


{move 1}

>>> postulate Lambda dedt in At ->> Bt


Lambda : [(.At_1 : type), (.Bt_1
    : type), (dedt_1 : [(aat2_2 : in
       .At_1) => (--- : in .Bt_1)]) =>
    (--- : in .At_1 ->> .Bt_1)]
```

```
   {move 0}
end Lestrade execution
```

In the section on equality, we will introduce more primitives for the case of types, which would have analogues for propositions if we were working in a constructive logic and wanted to carry out formal operations on proofs.

3 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.[1]

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

[1]A constructivist might presume that to suppose that X is true is the same thing as to suppose that X has been proved, but we do not presume a constructivist view here.

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 sort 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.

4 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 at the consequences of allowing environments to be named and saved in an appended subsection).

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" (elsewhere sometimes "the current 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.[2]

The `declare` command takes a fresh identifier and an object or function 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). Object sorts are represented by `prop`, `obj`, `type`, or by `that` p when p is of sort `prop`, or `in` τ when τ is of sort `type`. Function sorts are represented by terms $[x_1, \ldots, x_n \Rightarrow \tau]$ where the x_i's are variables declared in the next move and τ is an object term.

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

[2]In this simple model: saved moves make this possible, but Lestrade manages it with a minimum of fuss.

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

(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 (which can cause the namespace to get rather cluttered!)

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, postulating 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 postulating `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.

4.1 Namespace management refined: saving and retrieving environments

With the limited environment handling given above, there is no way to remove or revise declarations of variables and variable expressions in move 1 other than clearing all of them. After a while, it is quite hard to remember what sorts have been assigned to parameters and variable expressions, and for that matter what order they appear in (recalling that parameters in `postulate` and `define` commands must appear in order of declaration). We have already noted that the `clearcurrent` command will clear all declarations at the next move.

More intelligent namespace management is supported by the full specification of the `open`, `clearcurrent`, and `save` commands.

Each move is assigned a name. The default name is its numeral index (the j such that it is move j). The command `save envname` will save the next move with the name `envname`, associated with the list of names of preceding moves at the time it is saved (a saved move is actually identified by the sequence of names of all moves at the time it is saved, and this is how it is identified internally; this means that moves saved in different contexts can quite safely be tagged with the same name). The command `open envname` will open an already existing move (of the right index, wth the same preceding moves) with the name `envname` or if there is no such move, or create a new blank move with that name. The command `clearcurrent envname` will clear the net move and replace it with a move named `envname` if there is such a move with the appropriate preceding names of moves associated with it or replace it with a blank move of that name otherwise. A move cannot be saved or opened with its default numeral name: the reason for this is that we do not want the parameterless `open` or `clearcurrent` command to unexpectedly invoke declarations from a saved environment. For the same reason, no move other than move 0 in the sequence of moves associated with a named move created by an application of `save`, `open`, or `clearcurrent` may have its default numeral name.

If identifiers have been declared in earlier moves which conflict with those in a reopened saved move, Lestrade will interpret the identifier when entered by the user as referring to the instance from the latest move in which it appears, but will handle instances of earlier uses of the same identifier correctly in declarations from earlier moves (and will display them unhelpfully without additional annotation, at present).

The effect of all of this is that that instead of having a linear sequence of moves, which we can think of as times or possible worlds, we have a tree structure.[3]

5 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 $A \wedge B \rightarrow B \wedge A$.

[3]which can still be thought of using a temporal metaphor as working out the consequences of different choices on alternative timelines, as it were.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> 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


{move 1}

>>> define Andconj A B : Deduction uu


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


Andconj : [(A_1 : prop), (B_1 : prop) =>
    (--- : that (A_1 & B_1) -> B_1 & A_1)]


{move 0}
end Lestrade execution
```

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.

6 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.

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. In addition, there are λ-terms of quite general form, $[x_1, \ldots, x_n \Rightarrow T]$ where the

x_i's are variables declared at the next move and T is an object term.

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 (in earlier versions it was sometimes needed if the final preceding argument was a function identifier; it is now (we believe) always optional); 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.

7 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 but not discussed here.

```
begin Lestrade execution

   >>> declare T type


   T : type
```

```
   {move 1}

   >>> declare t in T


   t : in T


   {move 1}

   >>> declare u in T


   u : in T


   {move 1}

   >>> declare tt1 in T


   tt1 : in T


   {move 1}

   >>> declare tpred [tt1 => prop]


   tpred : [(tt1_1 : in T) => (--- : prop)]


   {move 1}
end Lestrade execution
```

We introduce a general object type T which will be a hidden parameter of our notions of equality.[4] We then introduce a predicate `tpred` of objects

[4]The implicit argument feature in effect allows overloading of equality as an operation

of type T (i.e, of sort `in` T).

```
begin Lestrade execution

   >>> 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}
end Lestrade execution
```

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

```
begin Lestrade execution

   >>> declare tpredev that tpred t


   tpredev : that tpred (t)
```

on each type; it was very annoying in earlier versions without this feature that equality on typed objects was a ternary relation and so had quite unexpected syntax. Genuine overloading could be achieved by for example declaring an addition operator `+` in every type without exception and providing for its properties by axioms in each type where it is to be used independently. Things that one expects to be uniformly true of all `+` operations (commutativlty, for example) could be stipulated as axioms.

```
{move 1}

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


Substitution0 : [(.T_1 : type), (.t_1
    : in .T_1), (.u_1 : in .T_1), (tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (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), (.tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (eqev_1
    : that .t_1 = .u_1), (tpredev_1
    : that .tpred_1 (.t_1)) =>
    ({def} Substitution0 (.tpred_1, eqev_1, tpredev_1) : that
    .tpred_1 (.u_1))]


Substitution : [(.T_1 : type), (.t_1
    : in .T_1), (.u_1 : in .T_1), (.tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (eqev_1
    : that .t_1 = .u_1), (tpredev_1
    : that .tpred_1 (.t_1)) => (---
    : that .tpred_1 (.u_1))]
```

```
   {move 0}
end Lestrade execution
```

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. We note that declaration of t and u before `tpred` is essential for implicit argument inference to work correctly for `Substitution`: one is faced with an expression for `tpred t` embedded in the type of `tpredev`, and one needs to know `t` to be able to guess what `tpred` is. The old implementation handled order of implicit arguments differently and was less sensitive to the actual order of declaration of implicit arguments than the new version, but the new version's approach has its own advantages.

```
begin Lestrade execution

   >>> postulate Reflexeq t : that t = t


   Reflexeq : [(.T_1 : type), (t_1 : in
        .T_1) => (--- : that t_1 = t_1)]


   {move 0}
end Lestrade execution
```

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.

```
begin Lestrade execution

   >>> open


      {move 2}
```

```
>>> declare t17 in T


t17 : in T


{move 2}

>>> declare u17 in T


u17 : in T


{move 2}

>>> declare v17 in T


v17 : in T


{move 2}

>>> declare eqev17 that t17 = u17


eqev17 : that t17 = u17


{move 2}

>>> define eqsymm0 eqev17 : Substitution0 \
    [v17 => v17 = t17], 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


{move 1}

>>> 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) =>
    ({def} Substitution0 ([(v17_2 : in
       .T_1) =>
       ({def} v17_2 = .t_1 : prop)], eqev_1, Reflexeq
    (.t_1)) : that .u_1 = .t_1)]


Eqsymm : [(.T_1 : type), (.t_1 : in
    .T_1), (.u_1 : in .T_1), (eqev_1
    : that .t_1 = .u_1) => (--- : that
    .u_1 = .t_1)]


{move 0}
end Lestrade execution
```

We present the proof of symmetry of equality. Notice the use of a variable binding term for the predicate in the crucial substitution step.

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.

```
begin Lestrade execution

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


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


   <-> : [(A_1 : prop), (B_1 : prop) =>
       (--- : prop)]


   {move 0}

   >>> declare ppred [A => prop]


   ppred : [(A_1 : prop) => (--- : prop)]


   {move 1}

   >>> 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 : [(.A_1 : prop), (.B_1
    : prop), (ppred_1 : [(A_2 : prop) =>
       (--- : 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), (.ppred_1 : [(A_2 : prop) =>
       (--- : prop)]), (iffev_1 : that
    .A_1 <-> .B_1), (ppredev_1 : that
    .ppred_1 (.A_1)) =>
    ({def} Substitutionp0 (.ppred_1, iffev_1, ppredev_1) : that
    .ppred_1 (.B_1))]


Substitutionp : [(.A_1 : prop), (.B_1
    : prop), (.ppred_1 : [(A_2 : prop) =>
       (--- : prop)]), (iffev_1 : that
    .A_1 <-> .B_1), (ppredev_1 : that
```

```
    .ppred_1 (.A_1)) => (--- : that
    .ppred_1 (.B_1))]


{move 0}

>>> define Reflexp0 A : Deduction [aaa1 \
       => aaa1]


Reflexp0 : [(A_1 : prop) =>
    ({def} Deduction ([(aaa1_2 : that
       A_1) =>
       ({def} aaa1_2 : that A_1)]) : that
    A_1 -> A_1)]


Reflexp0 : [(A_1 : prop) => (--- : 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) =>
    ({def} afix_1 : that A_1)]
```

```
propfixform : [(A_1 : prop), (afix_1
    : that A_1) => (--- : that A_1)]


{move 0}

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


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


Reflexp : [(A_1 : prop) => (--- : that
    A_1 <-> A_1)]


{move 0}

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


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


Reflexp1 : [(A_1 : prop) => (--- : that
    (A_1 -> A_1) & A_1 -> A_1)]


{move 0}
```

```
end Lestrade execution
```

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:

```
begin Lestrade execution

   >>> declare proofpred [aaa1 => prop]


   proofpred : [(aaa1_1 : that A) => (---
       : prop)]


   {move 1}

   >>> 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), (.aa_1
       : that .A_1), (.proofpred_1 : [(aaa1_2
          : that .A_1) => (--- : prop)]), (proofpredev_1
       : that .proofpred_1 (.aa_1)), (aax_1
       : that .A_1) => (--- : that .proofpred_1
       (aax_1))]


   {move 0}
end Lestrade execution
```

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.

7.1 Equality and type constructions

In this section we complete the primitives needed for Cartesian products and function spaces. Analogous constructions for propositions would be wanted in a constructive logic in which one wanted to extract information from proofs.

We implement the identities $\pi_1(x, y) = x; \pi_2(x, y) = y; (\pi_1(x), \pi_2(x)) = x$.

```
begin Lestrade execution

   >>> postulate Proj1 aat bbt : that proj1 \
       (Pair aat bbt) = aat


   Proj1 : [(.At_1 : type), (.Bt_1 : type), (aat_1
       : in .At_1), (bbt_1 : in .Bt_1) =>
       (--- : that proj1 (aat_1 Pair bbt_1) = aat_1)]


   {move 0}

   >>> postulate Proj2 aat bbt : that proj2 \
       (Pair aat bbt) = bbt


   Proj2 : [(.At_1 : type), (.Bt_1 : type), (aat_1
       : in .At_1), (bbt_1 : in .Bt_1) =>
       (--- : that proj2 (aat_1 Pair bbt_1) = bbt_1)]


   {move 0}

   >>> postulate Proj3 xxt : that Pair (proj1 \
       xxt, proj2 xxt) = xxt


   Proj3 : [(.At_1 : type), (.Bt_1 : type), (xxt_1
       : in .At_1 X .Bt_1) => (--- : that
```

```
        proj1 (xxt_1) Pair proj2 (xxt_1) = xxt_1)]


   {move 0}
end Lestrade execution
```

We implement the identities $(\lambda x.T)(a) = T[a/x]$ (β-reduction) and extensionality for function objects.

```
begin Lestrade execution

   >>> declare aat3 in At


   aat3 : in At


   {move 1}

   >>> postulate Betared dedt, aat3 : that \
       Apply (Lambda dedt, aat3) = dedt aat3


   Betared : [(.At_1 : type), (.Bt_1
       : type), (dedt_1 : [(aat2_2 : in
          .At_1) => (--- : in .Bt_1)]), (aat3_1
       : in .At_1) => (--- : that Lambda
       (dedt_1) Apply aat3_1 = dedt_1 (aat3_1))]


   {move 0}

   >>> declare dedt2 [aat3 => in Bt]


   dedt2 : [(aat3_1 : in At) => (---
       : in Bt)]
```

```
   {move 1}

   >>> declare fnext [aat3 => that dedt \
          aat3 = dedt2 aat3]


   fnext : [(aat3_1 : in At) => (---
       : that dedt (aat3_1) = dedt2 (aat3_1))]


   {move 1}

   >>> postulate Fnext fnext that (Lambda \
       dedt) = Lambda dedt2


   Fnext : [(.At_1 : type), (.Bt_1 : type), (.dedt_1
       : [(aat2_2 : in .At_1) => (---
          : in .Bt_1)]), (.dedt2_1 : [(aat3_2
          : in .At_1) => (--- : in .Bt_1)]), (fnext_1
       : [(aat3_2 : in .At_1) => (---
          : that .dedt_1 (aat3_2) = .dedt2_1
          (aat3_2))]) => (--- : that
       Lambda (.dedt_1) = Lambda (.dedt2_1))]


   {move 0}
end Lestrade execution
```

8 Natural numbers introduced

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

```
begin 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}
end Lestrade execution
```

The primitive notions of arithmetic are introduced. These are the type of natural numbers, the number zero, and the successor operation. We will next define iteration of a function a number of times, and we will see later that addition and multiplication are then definable.

```
begin Lestrade execution

   >>> declare nnn2 in Nat


   nnn2 : in Nat


   {move 1}

   >>> declare Tt [nnn2 => type]


   Tt : [(nnn2_1 : in Nat) => (--- : type)]


   {move 1}

   >>> open


      {move 2}

      >>> declare nn2 in Nat


      nn2 : in Nat


      {move 2}

      >>> declare ttt1 in Tt nn2
```

```
   ttt1 : in Tt (nn2)


   {move 2}

   >>> postulate F ttt1 in Tt (Succ nn2)


   F : [(.nn2_1 : in Nat), (ttt1_1
       : in Tt (.nn2_1)) => (--- : in
       Tt (Succ (.nn2_1)))]


   {move 1}

   >>> close


{move 1}

>>> 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 : [(nnn2_2 : in
       Nat) => (--- : type)]), (F_1
    : [(.nn2_2 : in Nat), (ttt1_2
       : in .Tt_1 (.nn2_2)) => (---
       : in .Tt_1 (Succ (.nn2_2)))]), (init_1
    : in .Tt_1 (0)), (n_1 : in Nat) =>
    (--- : in .Tt_1 (n_1))]


{move 0}

>>> define Iterate0 Tt, F, init n : Iterate \
    F, init n


Iterate0 : [(Tt_1 : [(nnn2_2 : in
       Nat) => (--- : type)]), (F_1
    : [(.nn2_2 : in Nat), (ttt1_2
       : in Tt_1 (.nn2_2)) => (---
       : in Tt_1 (Succ (.nn2_2)))]), (init_1
    : in Tt_1 (0)), (n_1 : in Nat) =>
    ({def} Iterate (F_1, init_1, n_1) : in
    Tt_1 (n_1))]


Iterate0 : [(Tt_1 : [(nnn2_2 : in
       Nat) => (--- : type)]), (F_1
    : [(.nn2_2 : in Nat), (ttt1_2
       : in Tt_1 (.nn2_2)) => (---
       : in Tt_1 (Succ (.nn2_2)))]), (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 : [(nnn2_2 : in
          Nat) => (--- : type)]), (F_1
       : [(.nn2_2 : in Nat), (ttt1_2
          : in .Tt_1 (.nn2_2)) => (---
          : in .Tt_1 (Succ (.nn2_2)))]), (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 : [(nnn2_2 : in
          Nat) => (--- : type)]), (F_1
       : [(.nn2_2 : in Nat), (ttt1_2
          : in .Tt_1 (.nn2_2)) => (---
          : in .Tt_1 (Succ (.nn2_2)))]), (init_1
       : in .Tt_1 (0)), (n_1 : in Nat) =>
       (--- : that Iterate (F_1, init_1, Succ
       (n_1)) = F_1 (n_1, Iterate (F_1, init_1, n_1)))]


   {move 0}
end Lestrade execution
```

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.

Note that the function variable `F` really needs to be declared in an open/close block if it is to have the syntax with which it is presented, because it has an implicit argument; there is no provision for function variables declared in one-line declarations to have implicit arguments.

```
begin Lestrade execution

   >>> declare nn99 in Nat


   nn99 : in Nat


   {move 1}

   >>> declare tt99 in T


   tt99 : in T


   {move 1}

   >>> declare F99 [tt99 => in T]


   F99 : [(tt99_1 : in T) => (--- : in
       T)]


   {move 1}

   >>> declare init98 in T


   init98 : in T
```

```
{move 1}

>>> declare n98 in Nat


n98 : in Nat


{move 1}

>>> define Simpleiter F99, init98 n98 \
    : Iterate [nn99, tt99 => F99 tt99], init98 \
    n98


Simpleiter : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1), (n98_1
    : in Nat) =>
    ({def} Iterate ([(nn99_2 : in Nat), (tt99_2
       : in .T_1) =>
       ({def} F99_1 (tt99_2) : in .T_1)], init98_1, n98_1) : in
    .T_1)]


Simpleiter : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1), (n98_1
    : in Nat) => (--- : in .T_1)]


{move 0}

>>> comment define Simpleiter2 F99, init98 \
    n98 : Iterate [nn99, tt99 => F99 tt99], init98 \
    n98
```

```
{move 1}

>>> define Simpleiter2 F99, init98 n98 \
    : Iterate [nn99, tt99 => F99 tt99], init98 \
    n98


Simpleiter2 : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1), (n98_1
    : in Nat) =>
    ({def} Iterate ([(nn99_2 : in Nat), (tt99_2
       : in .T_1) =>
       ({def} F99_1 (tt99_2) : in .T_1)], init98_1, n98_1) : in
    .T_1)]


Simpleiter2 : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1), (n98_1
    : in Nat) => (--- : in .T_1)]


{move 0}

>>> define Simpleinit F99, init98 : Initialize \
    [nn99, tt99 => F99 tt99], init98


Simpleinit : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1) =>
    ({def} Initialize ([(nn99_2 : in
       Nat), (tt99_2 : in .T_1) =>
       ({def} F99_1 (tt99_2) : in .T_1)], init98_1) : that
    Iterate ([(nn99_3 : in Nat), (tt99_3
```

```
        : in .T_1) =>
        ({def} F99_1 (tt99_3) : in .T_1)], init98_1, 0) = init98_1)]


Simpleinit : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1) =>
    (--- : that Iterate ([(nn99_3 : in
       Nat), (tt99_3 : in .T_1) =>
       ({def} F99_1 (tt99_3) : in .T_1)], init98_1, 0) = init98_1)]


{move 0}

>>> define Simpleiterstep F99, init98, n98 \
    : Iterstep [nn99, tt99 => F99 tt99], init98 \
    n98


Simpleiterstep : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1), (n98_1
    : in Nat) =>
    ({def} Iterstep ([(nn99_2 : in
       Nat), (tt99_2 : in .T_1) =>
       ({def} F99_1 (tt99_2) : in .T_1)], init98_1, n98_1) : that
    Iterate ([(nn99_3 : in Nat), (tt99_3
       : in .T_1) =>
       ({def} F99_1 (tt99_3) : in .T_1)], init98_1, Succ
    (n98_1)) = F99_1 (Iterate ([(nn99_4
       : in Nat), (tt99_4 : in .T_1) =>
       ({def} F99_1 (tt99_4) : in .T_1)], init98_1, n98_1)))]


Simpleiterstep : [(.T_1 : type), (F99_1
    : [(tt99_2 : in .T_1) => (--- : in
       .T_1)]), (init98_1 : in .T_1), (n98_1
    : in Nat) => (--- : that Iterate
```

```
        ([(nn99_3 : in Nat), (tt99_3
           : in .T_1) =>
           ({def} F99_1 (tt99_3) : in .T_1)], init98_1, Succ
        (n98_1)) = F99_1 (Iterate ([(nn99_4
           : in Nat), (tt99_4 : in .T_1) =>
           ({def} F99_1 (tt99_4) : in .T_1)], init98_1, n98_1)))]


   {move 0}
end Lestrade execution
```

We define simple iteration over a single type in terms of the more complex notion of iteration which we take as primitive. The alternative version `Simpleiter2` will be set up for automatic rewriting in an example below. Note the use of a bound variable term to refer to the form of `F99` which has an additional dummy natural number argument.

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.

`Fp` is declared in an open/close block so that it can have an implicit argument.

```
begin Lestrade execution

   >>> declare nn2 in Nat


   nn2 : in Nat


   {move 1}

   >>> declare Pp [nn2 => prop]
```

```
Pp : [(nn2_1 : in Nat) => (--- : prop)]


{move 1}

>>> open


   {move 2}

   >>> 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


{move 1}

>>> 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 : [(nn2_2 : in
       Nat) => (--- : prop)]), (Fp_1
    : [(.n2_2 : in Nat), (t1_2 : that
       .Pp_1 (.n2_2)) => (--- : that
       .Pp_1 (Succ (.n2_2)))]), (initp_1
    : that .Pp_1 (0)), (np_1 : in
    Nat) => (--- : that .Pp_1 (np_1))]


{move 0}
end Lestrade execution
```

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 $\tau(u,t) = \tau_1(g(u,t))$, where $g(u,t)$ is of the same constant sort as u. Now we want to define `Iterate f, init, n` so that `Iterate f, init, 0` is `init` and `Iterate f, init, Succ n` is `f(Iterate f, init, n)`. The sorts of $f(\texttt{Iterate}(\texttt{f},\texttt{init},\texttt{n}))$ and $\texttt{Iterate}(\texttt{f},\texttt{init},\texttt{Succ}\,\texttt{n})$ have to match. The first must take the form $\tau(g(u,\texttt{Iterate}(\texttt{f},\texttt{init},\texttt{n}))$, where the sort of `Iterate(f,init,n))` is $\tau_1(u)$. This tells us something about the output type of `Iterate f,init,n`: its output type must be a fixed function τ_1 of a parameter u extractible from the argument list `f,init,n`: write this `U(f,init,n)`. From this it follows that the type of `Iterate f, init, Succ n` is $\tau_1(\texttt{U}(\texttt{f},\texttt{init},\texttt{Succ}\,\texttt{n}))$. So $\tau_1(\texttt{U}(\texttt{f},\texttt{init},\texttt{Succ}\,\texttt{n})$ must match $\tau(g(\texttt{U}(\texttt{f},\texttt{init},\texttt{n}),\texttt{Iterate}(\texttt{f},\texttt{init},\texttt{n}))$. It can readily be seen that there is no actual dependence on `Iterate(f,init,n)` in the second term, since there is none in the first term. It appears in fact that the only way to achieve this compatible with the type matching facilities we have so far, which are entirely based on literal matching of terms supplemented with definitional expansion, is $\tau = \tau_1$, $g(f,\texttt{init},n) = \texttt{Succ}\,\texttt{n}$, whence $u = n$, which yields the form of `Iterate` given above.

Under the rewriting facilities of Lestrade not yet described, it may be possible to implement a more general form of iteration; we will revisit this later. In fact it seems pretty clear to us that the rewrite facility would handle iteration of a function $f(u,t)$ with t of type $\tau(u)$ and output type $\tau(g(u))$, where the type of $f^n(u,t)$ would be $\tau(g^n(u))$, for general τ and g: rewriting would allow the matching of types $\tau(g(\texttt{Simpleiter}(\texttt{g},\texttt{init},\texttt{n}))$ and $\tau(\texttt{Simpleiter}(\texttt{g},\texttt{init},\texttt{Succ}\,\texttt{n}))$, and considerations above indicate that this is the most general form of iteration we can expect to support. It appears that this would not require any additional primitives: the more general iteration would be definable in terms of the primitives already given. Additional primitives would be needed, precisely analogous to the ones we have, if we wanted to iterate functions applied to the sorts `prop`, `type`, or `obj` (the last being a case we would be very likely to want).

Here, where we are considering a version of Lestrade which does not support rewriting, we propose a different (though related) solution. Rewriting enables us to substitute equals for equals in types during the process of type inference. We cannot do this directly without rewriting, but we can provide axioms whicih indicate that substitutions of equals for equals in a type induce isomorphic types, then use the isomorphisms to get types to line up correctly.

The immediately following block, which is executed, implements abstract iteration mod the need to apply isomorphisms to get from a sort to a sort which ought to be equal to it. The block following that is the more economical implementation using rewriting, which is disabled (and exhibits the hideous error messages generated on the first pass).

```
begin Lestrade execution

   >>> declare T73 type


   T73 : type


   {move 1}

   >>> declare t73 in T73


   t73 : in T73


   {move 1}

   >>> declare u73 in T73


   u73 : in T73


   {move 1}

   >>> declare v73 in T73
```

```
v73 : in T73


{move 1}

>>> declare tau73 [t73 => type]


tau73 : [(t73_1 : in T73) => (---
    : type)]


{move 1}

>>> declare eqev73 that t73 = u73


eqev73 : that t73 = u73


{move 1}

>>> declare eqev72 that t73 = t73


eqev72 : that t73 = t73


{move 1}

>>> declare eqev75 that u73 = v73


eqev75 : that u73 = v73
```

```
{move 1}

>>> declare eqev76 that t73 = v73


eqev76 : that t73 = v73


{move 1}

>>> declare x73 in tau73 t73


x73 : in tau73 (t73)


{move 1}

>>> declare y73 in tau73 u73


y73 : in tau73 (u73)


{move 1}

>>> comment These declarations implement \
    the ability to substitute equals for equals \
    in types, by creating isomorphisms between \
    types related in this way


{move 1}

>>> postulate iso eqev73 x73 in tau73 \
    u73
```

```
iso : [(.T73_1 : type), (.t73_1 : in
    .T73_1), (.u73_1 : in .T73_1), (.tau73_1
    : [(t73_2 : in .T73_1) => (---
       : type)]), (eqev73_1 : that
    .t73_1 = .u73_1), (x73_1 : in .tau73_1
    (.t73_1)) => (--- : in .tau73_1
    (.u73_1))]


{move 0}

>>> postulate Isorefl eqev72 x73 that \
    x73 = iso eqev72 x73


Isorefl : [(.T73_1 : type), (.t73_1
    : in .T73_1), (.tau73_1 : [(t73_2
       : in .T73_1) => (--- : type)]), (eqev72_1
    : that .t73_1 = .t73_1), (x73_1
    : in .tau73_1 (.t73_1)) => (---
    : that x73_1 = eqev72_1 iso x73_1)]


{move 0}

>>> postulate Isocomp eqev73 eqev75 eqev76 \
    x73 that (iso eqev76 x73) = iso eqev75 \
    (iso eqev73 x73)


Isocomp : [(.T73_1 : type), (.t73_1
    : in .T73_1), (.u73_1 : in .T73_1), (.v73_1
    : in .T73_1), (.tau73_1 : [(t73_2
       : in .T73_1) => (--- : type)]), (eqev73_1
    : that .t73_1 = .u73_1), (eqev75_1
    : that .u73_1 = .v73_1), (eqev76_1
    : that .t73_1 = .v73_1), (x73_1
    : in .tau73_1 (.t73_1)) => (---
```

```
    : that eqev76_1 iso x73_1 = eqev75_1
    iso eqev73_1 iso x73_1)]


{move 0}

>>> declare g73 [t73 => in T73]


g73 : [(t73_1 : in T73) => (--- : in
    T73)]


{move 1}

>>> declare n73 in Nat


n73 : in Nat


{move 1}

>>> define Ourtt t73 tau73 g73 n73 : tau73 \
    (Simpleiter g73, t73 n73)


Ourtt : [(.T73_1 : type), (t73_1
    : in .T73_1), (tau73_1 : [(t73_2
       : in .T73_1) => (--- : type)]), (g73_1
    : [(t73_2 : in .T73_1) => (---
       : in .T73_1)]), (n73_1 : in
    Nat) =>
    ({def} tau73_1 (Simpleiter (g73_1, t73_1, n73_1)) : type)]


Ourtt : [(.T73_1 : type), (t73_1
    : in .T73_1), (tau73_1 : [(t73_2
```

```
        : in .T73_1) => (--- : type)]), (g73_1
    : [(t73_2 : in .T73_1) => (---
        : in .T73_1)]), (n73_1 : in
    Nat) => (--- : type)]
```

{move 0}

```
>>> declare F73 [t73 x73 => in tau73 \
       g73 t73]
```

```
F73 : [(t73_1 : in T73), (x73_1 : in
    tau73 (t73_1)) => (--- : in tau73
    (g73 (t73_1)))]
```

{move 1}

```
>>> Showdec Iterate
```

```
Iterate : [(.Tt_1 : [(nnn2_2 : in
       Nat) => (--- : type)]), (F_1
    : [(.nn2_2 : in Nat), (ttt1_2
       : in .Tt_1 (.nn2_2)) => (---
       : in .Tt_1 (Succ (.nn2_2)))]), (init_1
    : in .Tt_1 (0)), (n_1 : in Nat) =>
    (--- : in .Tt_1 (n_1))]
```

{move 0}

```
>>> comment the argument Tt is the function \
    Ourtt above
```

{move 1}

```
>>> comment build the argument F


{move 1}

>>> open


   {move 2}

   >>> declare m73 in Nat


   m73 : in Nat


   {move 2}

   >>> declare ttt73 in tau73 (Simpleiter \
       g73, t73, m73)


   ttt73 : in tau73 (Simpleiter (g73, t73, m73))


   {move 2}

   >>> define Fa73 m73 ttt73 : F73 (Simpleiter \
       (g73, t73, m73), ttt73)


   Fa73 : [(m73_1 : in Nat), (ttt73_1
       : in tau73 (Simpleiter (g73, t73, m73_1))) =>
       (--- : in tau73 (g73 (Simpleiter
       (g73, t73, m73_1))))]
```

```
   {move 1}

   >>> define Fb73 m73 ttt73 : iso (Eqsymm \
       (Simpleiterstep (g73, t73, m73)), Fa73 \
       m73 ttt73)


   Fb73 : [(m73_1 : in Nat), (ttt73_1
       : in tau73 (Simpleiter (g73, t73, m73_1))) =>
       (--- : in tau73 (Iterate ([(nn99_3
          : in Nat), (tt99_3 : in T73) =>
          ({def} g73 (tt99_3) : in T73)], t73, Succ
       (m73_1))))]


   {move 1}

   >>> close


{move 1}

>>> test Iterate0 ([n73 => tau73 (Simpleiter \
       (g73, t73, n73))], Fb73, iso \
    ((Eqsymm (Simpleinit g73, t73)), x73))


[(n_7 : in Nat) =>
    ({def} Iterate0 ([(n73_19 : in
       Nat) =>
       ({def} tau73 (Simpleiter (g73, t73, n73_19)) : type)], Fb73, Eqsymm
    (Simpleinit (g73, t73)) iso x73, n_7) : in
    tau73 (Simpleiter (g73, t73, n_7)))]


{move 1}

>>> comment define Abstractiter Fb73 x73 \
```

```
    n73 : Iterate0 ([n73 => tau73 (Simpleiter \
       (g73, t73, n73))], Fb73, iso \
    ((Eqsymm (Simpleinit g73, t73)), x73), n73)


{move 1}

>>> define Abstractiter Fb73 x73 n73 : Iterate \
    (Fb73, iso ((Eqsymm (Simpleinit g73, t73)), x73), n73)


Abstractiter : [(.T73_1 : type), (.t73_1
    : in .T73_1), (.tau73_1 : [(t73_2
       : in .T73_1) => (--- : type)]), (x73_1
    : in .tau73_1 (.t73_1)), (.g73_1
    : [(t73_2 : in .T73_1) => (---
       : in .T73_1)]), (n73_1 : in
    Nat), (.F73_1 : [(t73_2 : in .T73_1), (x73_2
       : in .tau73_1 (t73_2)) => (---
       : in .tau73_1 (.g73_1 (t73_2)))]) =>
    ({def} Iterate ([(m73_2 : in Nat), (ttt73_2
       : in .tau73_1 (Simpleiter (.g73_1, .t73_1, m73_2))) =>
       ({def} Eqsymm (Simpleiterstep
       (.g73_1, .t73_1, m73_2)) iso
       Simpleiter (.g73_1, .t73_1, m73_2) F73
       ttt73_2 : in .tau73_1 (Iterate
       ([(nn99_4 : in Nat), (tt99_4
          : in .T73_1) =>
          ({def} .g73_1 (tt99_4) : in
          .T73_1)], .t73_1, Succ (m73_2))))], Eqsymm
    (Simpleinit (.g73_1, .t73_1)) iso
    x73_1, n73_1) : in .tau73_1 (Simpleiter
    (.g73_1, .t73_1, n73_1)))]


Abstractiter : [(.T73_1 : type), (.t73_1
    : in .T73_1), (.tau73_1 : [(t73_2
       : in .T73_1) => (--- : type)]), (x73_1
```

```
      : in .tau73_1 (.t73_1)), (.g73_1
      : [(t73_2 : in .T73_1) => (---
         : in .T73_1)]), (n73_1 : in
      Nat), (.F73_1 : [(t73_2 : in .T73_1), (x73_2
         : in .tau73_1 (t73_2)) => (---
         : in .tau73_1 (.g73_1 (t73_2)))]) =>
      (--- : in .tau73_1 (Simpleiter (.g73_1, .t73_1, n73_1)))]


   {move 0}
end Lestrade execution

% begin Lestrade execution

   >>> declare U30 type


   U30 : type


   {move 1}

   >>> declare u30 in U30


   u30 : in U30


   {move 1}

   >>> declare n30 in Nat


   n30 : in Nat


   {move 1}
```

```
>>> open

   {move 2}

   >>> declare u31 in U30


   u31 : in U30


   {move 2}

   >>> postulate g30 u31 in U30


   g30 : [(u31_1 : in U30) => (---
       : in U30)]


   {move 1}

   >>> postulate tau30 u31 type


   tau30 : [(u31_1 : in U30) => (---
       : type)]


   {move 1}

   >>> declare t31 in tau30 u31


   t31 : in tau30 (u31)


   {move 2}
```

```
   >>> postulate f30 u31 t31 in tau30 \
       g30 u31


   f30 : [(u31_1 : in U30), (t31_1
       : in tau30 (u31_1)) => (---
       : in tau30 (g30 (u31_1)))]


   {move 1}

   >>> close


{move 1}

>>> declare t30 in tau30 u30


t30 : in tau30 (u30)


{move 1}

>>> rewritep Iterwrite u30, n30, g30, Simpleiter2 \
    g30, u30, Succ n30, g30 (Simpleiter2 \
    g30, u30, n30)


{move 1}

>>> rewritep Iterwrite2 u30, g30, Simpleiter2 \
    g30, u30, 0, u30


>>> declare init32 in tau30 u30
```

```
init32 : in tau30 (u30)


{move 1}

>>> declare n32 in Nat


n32 : in Nat


{move 1}

>>> open


   {move 2}

   >>> declare n33 in Nat


   n33 : in Nat


   {move 2}

   >>> define Tt30 n33 : (Simpleiter2 \
       g30, u30, n33)


   Tt30 : [(n33_1 : in Nat) => (---
       : in U30)]


   {move 1}

   >>> define Tt31 n33 : tau30 (Tt30 \
```

```
    n33)


Tt31 : [(n33_1 : in Nat) => (---
    : type)]


{move 1}

>>> declare n35 in Nat


n35 : in Nat


{move 2}

>>> declare t35 in tau30 (Tt30 n35)


t35 : in tau30 (Tt30 (n35))


{move 2}

>>> define f35 t35 : f30 (Tt30 n35, t35)


f35 : [(.n35_1 : in Nat), (t35_1
    : in tau30 (Tt30 (.n35_1))) =>
    (--- : in tau30 (g30 (Tt30 (.n35_1))))]


{move 1}

>>> close
```

```
   {move 1}

   >>> define Abstractiter f30, init32, n32 \
       : Iterate f35, init32, n32

% end Lestrade execution
```

9 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.

```
begin Lestrade execution

   >>> postulate Forall tpred : prop


   Forall : [(.T_1 : type), (tpred_1
       : [(tt1_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
    : [(tt1_2 : in .T_1) => (--- : prop)]), (univev_1
    : that Forall (.tpred_1)), (ttt_1
    : in .T_1) => (--- : that .tpred_1
    (ttt_1))]


{move 0}

>>> declare ugen [ttt => that tpred ttt]


ugen : [(ttt_1 : in T) => (--- : that
    tpred (ttt_1))]


{move 1}

>>> postulate Ugen ugen : that Forall \
    tpred


Ugen : [(.T_1 : type), (.tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (ugen_1
    : [(ttt_2 : in .T_1) => (--- : that
       .tpred_1 (ttt_2))]) => (---
    : that Forall (.tpred_1))]
```

```
   {move 0}
end Lestrade execution
```

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.

```
begin Lestrade execution

   >>> declare natpred [nn2 => prop]


   natpred : [(nn2_1 : in Nat) => (---
       : prop)]


   {move 1}

   >>> declare ind that Forall [nn2 => (natpred \
          nn2) -> natpred (Succ nn2)]


   ind : that Forall ([(nn2_2 : in Nat) =>
       ({def} natpred (nn2_2) -> natpred
```

```
      (Succ (nn2_2)) : prop)])


   {move 1}

   >>> declare basis that natpred 0


   basis : that natpred (0)


   {move 1}
end Lestrade execution
```

Here are familiar prerequisites for mathematical induction, the basis step, evidence for `natpred`(0), and the induction step, evidence for

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

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> 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 natpred (n2_1) -> natpred
       (Succ (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


{move 1}

>>> declare nq in Nat


nq : in Nat


{move 1}
```

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


Induction1 : [(.natpred_1 : [(nn2_2
       : in Nat) => (--- : prop)]), (ind_1
    : that Forall ([(nn2_3 : in Nat) =>
       ({def} .natpred_1 (nn2_3) ->
       .natpred_1 (Succ (nn2_3)) : prop)])), (basis_1
    : that .natpred_1 (0)), (nq_1
    : in Nat) =>
    ({def} Iteratep ([(n2_2 : in Nat), (indhyp_2
       : that .natpred_1 (n2_2)) =>
       ({def} indhyp_2 Mp ind_1 Uinst
       n2_2 : that .natpred_1 (Succ (n2_2)))], basis_1, nq_1) : that
    .natpred_1 (nq_1))]


Induction1 : [(.natpred_1 : [(nn2_2
       : in Nat) => (--- : prop)]), (ind_1
    : that Forall ([(nn2_3 : in Nat) =>
       ({def} .natpred_1 (nn2_3) ->
       .natpred_1 (Succ (nn2_3)) : prop)])), (basis_1
    : that .natpred_1 (0)), (nq_1
    : in Nat) => (--- : that .natpred_1
    (nq_1))]


{move 0}

>>> define Inductiona1 natpred, ind basis \
    nq : Iteratep step2, basis, nq


Inductiona1 : [(natpred_1 : [(nn2_2
       : in Nat) => (--- : prop)]), (ind_1
    : that Forall ([(nn2_3 : in Nat) =>
```

```
      ({def} natpred_1 (nn2_3) -> natpred_1
      (Succ (nn2_3)) : prop)])), (basis_1
   : that natpred_1 (0)), (nq_1 : in
   Nat) =>
   ({def} Iteratep ([(n2_2 : in Nat), (indhyp_2
      : that natpred_1 (n2_2)) =>
      ({def} indhyp_2 Mp ind_1 Uinst
      n2_2 : that natpred_1 (Succ (n2_2)))], basis_1, nq_1) : that
   natpred_1 (nq_1))]


Inductiona1 : [(natpred_1 : [(nn2_2
      : in Nat) => (--- : prop)]), (ind_1
   : that Forall ([(nn2_3 : in Nat) =>
      ({def} natpred_1 (nn2_3) -> natpred_1
      (Succ (nn2_3)) : prop)])), (basis_1
   : that natpred_1 (0)), (nq_1 : in
   Nat) => (--- : that natpred_1 (nq_1))]


{move 0}

>>> declare nq2 in Nat


nq2 : in Nat


{move 1}

>>> define Inductiona natpred, ind basis \
    : Ugen ([nq2 => Inductiona1 natpred, ind, basis, nq2])


Inductiona : [(natpred_1 : [(nn2_2
      : in Nat) => (--- : prop)]), (ind_1
   : that Forall ([(nn2_3 : in Nat) =>
      ({def} natpred_1 (nn2_3) -> natpred_1
```

```
      (Succ (nn2_3)) : prop)])), (basis_1
    : that natpred_1 (0)) =>
    ({def} Ugen ([(nq2_2 : in Nat) =>
      ({def} Induction1 (natpred_1, ind_1, basis_1, nq2_2) : that
      natpred_1 (nq2_2))]) : that
    Forall ([(tt1'_2 : in Nat) =>
      ({def} natpred_1 (tt1'_2) : prop)]))]


Inductiona : [(natpred_1 : [(nn2_2
      : in Nat) => (--- : prop)]), (ind_1
    : that Forall ([(nn2_3 : in Nat) =>
      ({def} natpred_1 (nn2_3) -> natpred_1
      (Succ (nn2_3)) : prop)])), (basis_1
    : that natpred_1 (0)) => (--- : that
    Forall ([(tt1'_2 : in Nat) =>
      ({def} natpred_1 (tt1'_2) : prop)]))]


{move 0}

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


Induction : [(.natpred_1 : [(nn2_2
      : in Nat) => (--- : prop)]), (ind_1
    : that Forall ([(nn2_3 : in Nat) =>
      ({def} .natpred_1 (nn2_3) ->
      .natpred_1 (Succ (nn2_3)) : prop)])), (basis_1
    : that .natpred_1 (0)) =>
    ({def} Ugen ([(nq_2 : in Nat) =>
      ({def} Induction1 (ind_1, basis_1, nq_2) : that
      .natpred_1 (nq_2))]) : that
    Forall ([(tt1'_2 : in Nat) =>
      ({def} .natpred_1 (tt1'_2) : prop)]))]
```

```
   Induction : [(.natpred_1 : [(nn2_2
          : in Nat) => (--- : prop)]), (ind_1
      : that Forall ([(nn2_3 : in Nat) =>
         ({def} .natpred_1 (nn2_3) ->
         .natpred_1 (Succ (nn2_3)) : prop)])), (basis_1
      : that .natpred_1 (0)) => (---
      : that Forall ([(tt1'_2 : in Nat) =>
         ({def} .natpred_1 (tt1'_2) : prop)]))]


   {move 0}
end Lestrade execution
```

Here is the proof of a standard form of mathematical induction. `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.

10 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.

```
begin 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) =>
       ({def} Simpleiter (Succ, N1_1, N2_1) : in
       Nat)]


   + : [(N1_1 : in Nat), (N2_1 : in
       Nat) => (--- : in Nat)]


   {move 0}
end Lestrade execution
```

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.

```
begin Lestrade execution

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


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


   Addid : [(N1_1 : in Nat) => (--- : 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) =>
       ({def} (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))]
```

```
   Additer : [(N1_1 : in Nat), (N2_1
       : in Nat) => (--- : that (N1_1 + Succ
       (N2_1)) = Succ (N1_1 + N2_1))]


   {move 0}
end Lestrade execution
```

Here the usual Peano axioms for addition are proved as instances of the basic equations governing simple iteration.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> 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


{move 1}

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


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


* : [(N1_1 : in Nat), (N2_1 : in
    Nat) => (--- : in Nat)]


{move 0}

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


Multzero : [(N1_1 : in Nat) =>
    ({def} (N1_1 * 0) = 0 propfixform
    Simpleinit ([(n3_3 : in Nat) =>
       ({def} n3_3 + N1_1 : in Nat)], 0) : that
    (N1_1 * 0) = 0)]
```

```
   Multzero : [(N1_1 : in Nat) => (---
       : 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) =>
       ({def} (N1_1 * Succ (N2_1)) = (N1_1
       * N2_1) + N1_1 propfixform Simpleiterstep
       ([(n3_3 : in Nat) =>
          ({def} n3_3 + N1_1 : in Nat)], 0, N2_1) : that
       (N1_1 * Succ (N2_1)) = (N1_1 * N2_1) + N1_1)]


   Multiter : [(N1_1 : in Nat), (N2_1
       : in Nat) => (--- : that (N1_1 * Succ
       (N2_1)) = (N1_1 * N2_1) + N1_1)]


   {move 0}
end Lestrade execution
```

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` (when this was first written: we could replace it with `[nn2 => nn2 + N1]`, but we see no compelling reason to do so).

11 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.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> declare M3 in Nat


      M3 : in Nat


      {move 2}

      >>> declare N3 in Nat


      N3 : in Nat


      {move 2}

      >>> define commuteswithall M3 : Forall \
          [N3 => (M3 + N3) = N3 + M3]


      commuteswithall : [(M3_1 : in Nat) =>
          (--- : prop)]


      {move 1}
```

```
      >>> close


   {move 1}
end Lestrade execution
```

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.

```
begin Lestrade execution

   >>> comment The basis step 0 + 0 = 0 + 0 .


   {move 1}

   >>> define zerocommuteswithzero : Reflexeq \
       (0 + 0)


   zerocommuteswithzero : [
       ({def} Reflexeq (0 + 0) : that (0 + 0) = 0 + 0)]


   zerocommuteswithzero : that (0 + 0) = 0 + 0


   {move 0}

   >>> open


      {move 2}
```

```
>>> declare M3 in Nat


M3 : in Nat


{move 2}

>>> open


   {move 3}

   >>> comment Now the induction step \
       : we need to show that if 0 + m = m + 0 then \
       0 + S (m) = S (m) + 0


   {move 3}

   >>> 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}

>>> declare M4 in Nat


M4 : in Nat
```

```
{move 3}

>>> define commzero5 indhyp : Substitution0 \
    ([M4 => (0 + Succ M3) = M4], Eqsymm \
    (Addid (Succ M3)), commzero4 \
    indhyp)


commzero5 : [(indhyp_1 : that
    (0 + M3) = M3 + 0) => (---
    : that (0 + Succ (M3)) = Succ
    (M3) + 0)]


{move 2}

>>> close
```

```
{move 2}

>>> 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
```

```
{move 1}

>>> define commzerobasisindstep : Ugen \
```

```
    indstep1


commzerobasisindstep : [
    ({def} Ugen ([(M3_2 : in Nat) =>
       ({def} Deduction ([(indhyp_3
          : that (0 + M3_2) = M3_2 + 0) =>
          ({def} Substitution0 ([(M4_4
             : in Nat) =>
             ({def} (0 + Succ (M3_2)) = M4_4
             : prop)], Eqsymm (Addid
          (Succ (M3_2))), Addid (M3_2) Substitution
          indhyp_3 Substitution 0 Additer
          M3_2) : that (0 + Succ (M3_2)) = Succ
          (M3_2) + 0)]) : that (0 + M3_2) = M3_2
       + 0 -> (0 + Succ (M3_2)) = Succ
       (M3_2) + 0)]) : that Forall
    ([(tt1'_2 : in Nat) =>
       ({def} (0 + tt1'_2) = tt1'_2
       + 0 -> (0 + Succ (tt1'_2)) = Succ
       (tt1'_2) + 0 : prop)]))]


commzerobasisindstep : that Forall ([(tt1'_2
    : in Nat) =>
    ({def} (0 + tt1'_2) = tt1'_2 + 0 ->
    (0 + Succ (tt1'_2)) = Succ (tt1'_2) + 0 : prop)])


{move 0}

>>> define commzerobasis : Induction commzerobasisindstep \
    zerocommuteswithzero


commzerobasis : [
    ({def} commzerobasisindstep Induction
    zerocommuteswithzero : that Forall
```

```
      ([(tt1'_2 : in Nat) =>
         ({def} (0 + tt1'_2) = tt1'_2
         + 0 : prop)]))]


   commzerobasis : that Forall ([(tt1'_2
       : in Nat) =>
       ({def} (0 + tt1'_2) = tt1'_2 + 0 : prop)])


   {move 0}
end Lestrade execution
```

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

```
begin Lestrade execution

   >>> declare M3 in Nat


   M3 : in Nat


   {move 1}

   >>> open


      {move 2}

      >>> declare commindhyp that commuteswithall \
          M3


      commindhyp : that commuteswithall (M3)
```

```
{move 2}

>>> comment We provide evidence that \
    for all x, M3 + x = x + M3


{move 2}

>>> open


   {move 3}

   >>> declare N3 in Nat


   N3 : in Nat


   {move 3}

   >>> comment Our aim here is to show \
       that S (M3) + N3 = N3 + S (M3)


   {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}
```

```
end Lestrade execution
```

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.

```
begin Lestrade execution

         >>> define commindlemma1 : Addid \
             Succ M3


         commindlemma1 : that (Succ (M3) + 0) = Succ
          (M3)


         {move 2}

         >>> declare N4 in Nat


         N4 : in Nat


         {move 3}

         >>> define commindlemma2 : Substitution0 \
             ([N4 => (Succ M3 + 0) = Succ \
                N4], Eqsymm (Addid M3), commindlemma1)


         commindlemma2 : that (Succ (M3) + 0) = Succ
          (M3 + 0)


         {move 2}
end Lestrade execution
```

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

```
begin Lestrade execution

          >>> open


             {move 4}

             >>> 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


{move 3}

>>> 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 ([(tt1'_2
    : in Nat) =>
    ({def} (Succ (M3) + tt1'_2) = Succ
    (M3 + tt1'_2) -> (Succ (M3) + Succ
    (tt1'_2)) = Succ (M3 + Succ
    (tt1'_2)) : prop)])


{move 2}

>>> define commindlemma : Induction \
    commindlemma7, commindlemma2


commindlemma : that Forall ([(tt1'_2
    : in Nat) =>
    ({def} (Succ (M3) + tt1'_2) = Succ
    (M3 + tt1'_2) : prop)])


{move 2}

>>> declare M4 in Nat


M4 : in Nat


{move 3}

>>> define commind2 N3 : Substitution0 \
    ([M4 => (Succ M3 + N3) = M4], 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


{move 2}

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

```
   commind5 : [(commindhyp_1 : that
       commuteswithall (M3)) => (---
       : that commuteswithall (Succ (M3)))]


   {move 1}

   >>> close


{move 1}

>>> define commind6 M3 : Deduction commind5


commind6 : [(M3_1 : in Nat) =>
    ({def} Deduction ([(commindhyp_2
       : that Forall ([(N3_4 : in Nat) =>
          ({def} (M3_1 + N3_4) = N3_4
          + M3_1 : prop)])) =>
       ({def} Forall ([(N3_4 : in Nat) =>
          ({def} (Succ (M3_1) + N3_4) = N3_4
          + Succ (M3_1) : prop)]) propfixform
       Ugen ([(N3_4 : in Nat) =>
          ({def} Eqsymm (N3_4 Additer
          M3_1) Substitution commindhyp_2
          Uinst N3_4 Substitution Substitution0
          ([(M4_7 : in Nat) =>
             ({def} (Succ (M3_1) + N3_4) = M4_7
             : prop)], Ugen ([(N3_10
             : in Nat) =>
             ({def} Deduction ([(commindlemmaindhyp_11
                : that (Succ (M3_1) + N3_10) = Succ
                (M3_1 + N3_10)) =>
                ({def} Eqsymm (M3_1 Additer
                N3_10) Substitution commindlemmaindhyp_11
```

```
                Substitution Succ (M3_1) Additer
                N3_10 : that (Succ (M3_1) + Succ
                (N3_10)) = Succ (M3_1
                + Succ (N3_10)))]) : that
             (Succ (M3_1) + N3_10) = Succ
             (M3_1 + N3_10) -> (Succ
             (M3_1) + Succ (N3_10)) = Succ
             (M3_1 + Succ (N3_10)))]) Induction
          Substitution0 ([(N4_10 : in
             Nat) =>
             ({def} (Succ (M3_1) + 0) = Succ
             (N4_10) : prop)], Eqsymm
          (Addid (M3_1)), Addid (Succ
          (M3_1))) Uinst N3_4, Reflexeq
          (Succ (M3_1) + N3_4)) : that
          (Succ (M3_1) + N3_4) = N3_4
          + Succ (M3_1))]) : that
       Forall ([(N3_3 : in Nat) =>
          ({def} (Succ (M3_1) + N3_3) = N3_3
          + Succ (M3_1) : prop)]))]) : that
    Forall ([(N3_3 : in Nat) =>
       ({def} (M3_1 + N3_3) = N3_3 + M3_1
       : prop)]) -> Forall ([(N3_3
       : in Nat) =>
       ({def} (Succ (M3_1) + N3_3) = N3_3
       + Succ (M3_1) : prop)]))]


commind6 : [(M3_1 : in Nat) => (---
    : that Forall ([(N3_3 : in Nat) =>
       ({def} (M3_1 + N3_3) = N3_3 + M3_1
       : prop)]) -> Forall ([(N3_3
       : in Nat) =>
       ({def} (Succ (M3_1) + N3_3) = N3_3
       + Succ (M3_1) : prop)]))]


{move 0}
```

```
>>> define commind7 : Ugen commind6


commind7 : [
    ({def} Ugen (commind6) : that Forall
    ([(tt1'_2 : in Nat) =>
       ({def} Forall ([(N3_4 : in Nat) =>
          ({def} (tt1'_2 + N3_4) = N3_4
          + tt1'_2 : prop)]) -> Forall
       ([(N3_4 : in Nat) =>
          ({def} (Succ (tt1'_2) + N3_4) = N3_4
          + Succ (tt1'_2) : prop)]) : prop)]))]


commind7 : that Forall ([(tt1'_2 : in
    Nat) =>
    ({def} Forall ([(N3_4 : in Nat) =>
       ({def} (tt1'_2 + N3_4) = N3_4
       + tt1'_2 : prop)]) -> Forall
    ([(N3_4 : in Nat) =>
       ({def} (Succ (tt1'_2) + N3_4) = N3_4
       + Succ (tt1'_2) : prop)]) : prop)])


{move 0}

>>> define Addcomm : Induction commind7 \
    commzerobasis


Addcomm : [
    ({def} commind7 Induction commzerobasis
    : that Forall ([(tt1'_2 : in Nat) =>
       ({def} Forall ([(N3_3 : in Nat) =>
          ({def} (tt1'_2 + N3_3) = N3_3
          + tt1'_2 : prop)]) : prop)]))]
```

```
Addcomm : that Forall ([(tt1'_2 : in
    Nat) =>
    ({def} Forall ([(N3_3 : in Nat) =>
       ({def} (tt1'_2 + N3_3) = N3_3
       + tt1'_2 : 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) =>
    ({def} Addcomm Uinst term1_1 Uinst
    term2_1 : that (term1_1 + term2_1) = term2_1
    + term1_1)]


Addcomm2 : [(term1_1 : in Nat), (term2_1
```

```
      : in Nat) => (--- : that (term1_1
      + term2_1) = term2_1 + term1_1)]


   {move 0}
end Lestrade execution
```

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.

```
% begin Lestrade execution

   >>> open

      >>> declare M3 in Nat

      >>> declare N3 in Nat

      >>> define commuteswithall M3 : Forall \
          [N3 => (M3 + N3) = N3 + M3]

% end Lestrade execution
```

What follows is the same proof in a considerably compressed format. This format could probably be achieved by a setting of the prover: what I did is preserved command lines, comments, and reported types of defined concepts only.

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.

```
% begin Lestrade execution

   >>> comment The basis step 0 + 0 = 0 + 0 .

   >>> define zerocommuteswithzero : Reflexeq \
       (0 + 0)

   >>> open

      >>> declare M3 in Nat

      >>> open

         >>> comment Now the induction step \
             : we need to show that if 0 + m = m + 0 then \
             0 + S (m) = S (m) + 0

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

         >>> define commzero1 : Additer 0 M3


         commzero1 : that (0 + Succ (M3)) = Succ
          (0 + M3)


         >>> define commzero2 indhyp : Substitution \
             (indhyp, commzero1)


         commzero2 : [(indhyp_1 : that
             (0 + M3) = M3 + 0) => (---
             : that (0 + Succ (M3)) = Succ
             (M3 + 0))]

         >>> define commzero3 : Addid M3
```

```
   commzero3 : that (M3 + 0) = M3

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


   commzero4 : [(indhyp_1 : that
       (0 + M3) = M3 + 0) => (---
       : that (0 + Succ (M3)) = Succ
       (M3))]

   >>> declare M4 in Nat

   >>> define commzero5 indhyp : Substitution0 \
       ([M4 => (0 + Succ M3) = M4], Eqsymm \
       (Addid (Succ M3)), commzero4 \
       indhyp)


   commzero5 : [(indhyp_1 : that
       (0 + M3) = M3 + 0) => (---
       : that (0 + Succ (M3)) = Succ
       (M3) + 0)]

   >>> 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)]

>>> close
```

```
>>> define commzerobasisindstep : Ugen \
    indstep1

commzerobasisindstep : that Forall ([(tt1'_2
    : in Nat) =>
    ({def} (0 + tt1'_2) = tt1'_2 + 0 ->
    (0 + Succ (tt1'_2)) = Succ (tt1'_2) + 0 : prop)])

>>> define commzerobasis : Induction commzerobasisindstep \
    zerocommuteswithzero

commzerobasis : that Forall ([(tt1'_2
    : in Nat) =>
    ({def} (0 + tt1'_2) = tt1'_2 + 0 : prop)])
```

```
% end Lestrade execution
```

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

```
% begin Lestrade execution

   >>> declare M3 in Nat

   >>> open

      >>> declare commindhyp that commuteswithall \
          M3

      >>> comment We provide evidence that \
          for all x, M3 + x = x + M3

      >>> open

         >>> declare N3 in Nat

         >>> comment Our aim here is to show \
```

```
    that S (M3) + N3 = N3 + S (M3)

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

commind1 : [(N3_1 : in Nat) =>
    (--- : that (Succ (M3) + N3_1) = Succ
    (M3) + N3_1)]

%  end Lestrade execution
```

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.

```
% begin Lestrade execution

>>> define commindlemma1 : Addid \
    Succ M3

commindlemma1 : that (Succ (M3) + 0) = Succ
 (M3)


>>> declare N4 in Nat

>>> define commindlemma2 : Substitution0 \
    ([N4 => (Succ M3 + 0) = Succ \
       N4], Eqsymm (Addid M3), commindlemma1)

commindlemma2 : that (Succ (M3) + 0) = Succ
 (M3 + 0)

%end Lestrade execution
```

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

```
% begin Lestrade execution

         >>> open

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

            commindlemmaindhyp : that (Succ
             (M3) + N3) = Succ (M3 + N3)

            >>> define commindlemma3 : Additer \
                Succ M3 N3

            commindlemma3 : that (Succ (M3) + Succ
             (N3)) = Succ (Succ (M3) + N3)

            >>> 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)))]

            >>> 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)))]

            >>> 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)))]

>>> define commindlemma7 : Ugen \
    commindlemma6

commindlemma7 : that Forall ([(tt1'_2
    : in Nat) =>
    ({def} (Succ (M3) + tt1'_2) = Succ
    (M3 + tt1'_2) -> (Succ (M3) + Succ
    (tt1'_2)) = Succ (M3 + Succ
    (tt1'_2)) : prop)])

>>> define commindlemma : Induction \
    commindlemma7, commindlemma2

commindlemma : that Forall ([(tt1'_2
    : in Nat) =>
    ({def} (Succ (M3) + tt1'_2) = Succ
    (M3 + tt1'_2) : prop)])

>>> declare M4 in Nat

>>> define commind2 N3 : Substitution0 \
    ([M4 => (Succ M3 + N3) = M4], Uinst \
    commindlemma N3, commind1 N3)

commind2 : [(N3_1 : in Nat) =>
    (--- : that (Succ (M3) + N3_1) = Succ
    (M3 + N3_1))]

>>> define commind3 N3 : Substitution \
```

```
        (Uinst commindhyp N3, commind2 \
        N3)

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

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

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

    >>> close

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

  commind5 : [(commindhyp_1 : that
      commuteswithall (M3)) => (---
      : that commuteswithall (Succ (M3)))]

  >>> close

>>> define commind6 M3 : Deduction commind5

commind6 : [(M3_1 : in Nat) => (---
    : that Forall ([(N3_3 : in Nat) =>
      ({def} (M3_1 + N3_3) = N3_3 + M3_1
      : prop)]) -> Forall ([(N3_3
      : in Nat) =>
      ({def} (Succ (M3_1) + N3_3) = N3_3
      + Succ (M3_1) : prop)]))]
```

```
   >>> define commind7 : Ugen commind6

   commind7 : that Forall ([(tt1'_2 : in
       Nat) =>
       ({def} Forall ([(N3_4 : in Nat) =>
          ({def} (tt1'_2 + N3_4) = N3_4
          + tt1'_2 : prop)]) -> Forall
       ([(N3_4 : in Nat) =>
          ({def} (Succ (tt1'_2) + N3_4) = N3_4
          + Succ (tt1'_2) : prop)]) : prop)])


   >>> define Addcomm : Induction commind7 \
       commzerobasis



   Addcomm : that Forall ([(tt1'_2 : in
       Nat) =>
       ({def} Forall ([(N3_3 : in Nat) =>
          ({def} (tt1'_2 + N3_3) = N3_3
          + tt1'_2 : prop)]) : prop)])


   >>> declare term1 in Nat

   >>> declare term2 in Nat


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

   Addcomm2 : [(term1_1 : in Nat), (term2_1
       : in Nat) => (--- : that (term1_1
       + term2_1) = term2_1 + term1_1)]

% end Lestrade execution
```

12 Power set types introduced

```
begin 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
       : [(tt1_2 : in .T_1) => (--- : prop)]) =>
       (--- : in setsof (.T_1))]


   {move 0}
end Lestrade execution
```

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 $\{x \in T : \texttt{tpred}(x)\}$, subject to the same remark about abuse of terminology for types and sets.

```
begin 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}
end Lestrade execution
```

We declare the membership relation.

```
begin 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
    : [(tt1_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
    : [(tt1_2 : in .T_1) => (--- : prop)]), (.t6_1
    : in .T_1), (elementev1_1 : that
    tpred_1 (.t6_1)) =>
    ({def} Comprehension10 (tpred_1, .t6_1, elementev1_1) : that
    .t6_1 E setof (tpred_1))]


Comprehension11 : [(.T_1 : type), (tpred_1
    : [(tt1_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 Comprehension12 t6 elementev1 \
       : Comprehension10 tpred, t6 elementev1


   Comprehension12 : [(.T_1 : type), (.tpred_1
       : [(tt1_2 : in .T_1) => (--- : prop)]), (t6_1
       : in .T_1), (elementev1_1 : that
       .tpred_1 (t6_1)) =>
       ({def} Comprehension10 (.tpred_1, t6_1, elementev1_1) : that
       t6_1 E setof (.tpred_1))]


   Comprehension12 : [(.T_1 : type), (.tpred_1
       : [(tt1_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}

   >>> postulate Comprehension2 elementev2 \
       that tpred t6


   Comprehension2 : [(.T_1 : type), (.tpred_1
       : [(tt1_2 : in .T_1) => (--- : prop)]), (.t6_1
       : in .T_1), (elementev2_1 : that
       .t6_1 E setof (.tpred_1)) => (---
       : that .tpred_1 (.t6_1))]


   {move 0}
end Lestrade execution
```

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`.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> 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


{move 1}

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


Extensionality : [(.T_1 : type), (.tpred1_1
    : [(t5_2 : in .T_1) => (--- : prop)]), (.tpred2_1
    : [(t5_2 : in .T_1) => (--- : prop)]), (ext1_1
    : [(.t5_2 : in .T_1), (tpredev1_2
       : that .tpred1_1 (.t5_2)) =>
       (--- : that .tpred2_1 (.t5_2))]), (ext2_1
    : [(.t5_2 : in .T_1), (tpredev2_2
       : that .tpred1_1 (.t5_2)) =>
       (--- : that .tpred1_1 (.t5_2))]) =>
    (--- : that setof (.tpred1_1) = setof
    (.tpred2_1))]


{move 0}

>>> declare s7 in setsof T


s7 : in setsof (T)


{move 1}

>>> declare t5 in T


t5 : in T
```

```
   {move 1}

   >>> postulate Extensionality2 s7 that \
       s7 = setof [t5 => t5 E s7]


   Extensionality2 : [(.T_1 : type), (s7_1
       : in setsof (.T_1)) => (--- : that
       s7_1 = setof ([(t5_3 : in .T_1) =>
          ({def} t5_3 E s7_1 : prop)]))]


   {move 0}
end Lestrade execution
```

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.

13 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.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> declare A1 prop


      A1 : prop


      {move 2}

      >>> declare ao obj


      ao : obj
```

```
{move 2}

>>> declare bo obj


bo : obj


{move 2}

>>> open


   {move 3}

   >>> declare xo obj


   xo : obj


   {move 3}

   >>> postulate opred xo prop


   opred : [(xo_1 : obj) => (---
       : prop)]


   {move 2}

   >>> close


{move 2}
```

```
      >>> postulate osetof opred obj


      osetof : [(opred_1 : [(xo_2 : obj) =>
             (--- : prop)]) => (--- : obj)]


      {move 1}
end Lestrade execution
```

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

```
begin Lestrade execution

      >>> postulate Eo ao bo prop


      Eo : [(ao_1 : obj), (bo_1 : obj) =>
          (--- : prop)]


      {move 1}

      >>> declare oelementev1 that ao Eo \
          osetof opred


      oelementev1 : that ao Eo osetof (opred)


      {move 2}

      >>> declare oelementev2 that opred \
          ao
```

```
      oelementev2 : that opred (ao)


      {move 2}

      >>> 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 1}

      >>> 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 1}
end Lestrade execution
```

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.

```
begin Lestrade execution

      >>> open
```

```
    {move 3}

    >>> declare yo obj


    yo : obj


    {move 3}

    >>> define R yo : (yo Eo yo) -> \
        A1


    R : [(yo_1 : obj) => (--- : prop)]


    {move 2}

    >>> close


  {move 2}

  >>> define r A1 : osetof R


  r : [(A1_1 : prop) => (--- : obj)]


  {move 1}
end Lestrade execution
```

This is our paradoxical set `r(A1)` , which we would write in ordinary notation as $\{x : x \in x \to A1\}$.

```
begin Lestrade execution

      >>> open


         {move 3}

         >>> declare rhyp that (r A1) Eo \
             r A1


         rhyp : that r (A1) Eo r (A1)


         {move 3}

         >>> define rstep1 rhyp : Ocomp1 \
             rhyp


         rstep1 : [(rhyp_1 : that r (A1) Eo
             r (A1)) => (--- : that (r (A1) Eo
             r (A1)) -> A1)]


         {move 2}

         >>> define rstep2 rhyp : Mp rhyp \
             (rstep1 rhyp)


         rstep2 : [(rhyp_1 : that r (A1) Eo
             r (A1)) => (--- : that A1)]


         {move 2}

         >>> define rstep3 rhyp : Deduction \
```

```
    rstep2


  rstep3 : [(rhyp_1 : that r (A1) Eo
      r (A1)) => (--- : that (r (A1) Eo
      r (A1)) -> A1)]


  {move 2}

  >>> define rstep4 rhyp : Mp rhyp \
      rstep3 rhyp


  rstep4 : [(rhyp_1 : that r (A1) Eo
      r (A1)) => (--- : that A1)]


  {move 2}

  >>> close


{move 2}

>>> define Russell1 A1 : Deduction \
    rstep4


Russell1 : [(A1_1 : prop) => (---
    : that (r (A1_1) Eo r (A1_1)) ->
    A1_1)]


{move 1}

>>> 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)) => (---
        : that ao_1 Eo osetof (.opred_1))]


    {move 1}

    >>> define Russell2 A1 : propfixform \
        ((r A1) Eo (r A1), Ocomp22 ((r A1), (Russell1 \
        A1)))


    Russell2 : [(A1_1 : prop) => (---
        : that r (A1_1) Eo r (A1_1))]


    {move 1}

    >>> define Russell A1 : Mp (Russell2 \
        A1, Russell1 A1)


    Russell : [(A1_1 : prop) => (---
        : that A1_1)]


    {move 1}

    >>> close


   {move 1}
end Lestrade execution
```

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

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

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

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

Since we have both $R \in R$ and $R \in R \to A$, 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).

The observant reader may notice that we packed the whole preceding argument in an extra Lestrade environment, so that we do not actually have primitives at move 0 which allow us to deduce that any proposition A is true. What we can prove at move 0 we now unveil (if objects with types of the primitives in the development above exist, contradiction follows). It is also a frightening example of the effects of definition unpacking!

```
begin Lestrade execution

   >>> define Russellthm A, osetof, Eo, Ocomp1, Ocomp2 \
       : Russell A


   Russellthm : [(A_1 : prop), (osetof_1
       : [(opred_2 : [(xo_3 : obj) =>
             (--- : prop)]) => (--- : obj)]), (Eo_1
       : [(ao_2 : obj), (bo_2 : obj) =>
```

```
   (--- : prop)]), (Ocomp1_1
: [(.ao_2 : obj), (.opred_2 : [(xo_3
      : obj) => (--- : prop)]), (oelementev1_2
   : that .ao_2 Eo osetof_1 (.opred_2)) =>
   (--- : that .opred_2 (.ao_2))]), (Ocomp2_1
: [(ao_2 : obj), (opred_2 : [(xo_3
      : obj) => (--- : prop)]), (oelementev2_2
   : that opred_2 (ao_2)) => (---
   : that ao_2 Eo osetof_1 (opred_2))]) =>
({def} ((osetof_1 ([(yo_5 : obj) =>
   ({def} (yo_5 Eo yo_5) -> A_1
   : prop)]) Eo osetof_1 ([(yo_5
   : obj) =>
   ({def} (yo_5 Eo yo_5) -> A_1
   : prop)])) propfixform Ocomp2_1
(osetof_1 ([(yo_5 : obj) =>
   ({def} (yo_5 Eo yo_5) -> A_1
   : prop)]), [(xo_4 : obj) =>
   ({def} (xo_4 Eo xo_4) -> A_1
   : prop)], Deduction ([(rhyp_5
   : that osetof_1 ([(yo_8 : obj) =>
      ({def} (yo_8 Eo yo_8) -> A_1
      : prop)]) Eo osetof_1 ([(yo_8
      : obj) =>
      ({def} (yo_8 Eo yo_8) -> A_1
      : prop)])) =>
   ({def} rhyp_5 Mp Deduction ([(rhyp_7
      : that osetof_1 ([(yo_10 : obj) =>
         ({def} (yo_10 Eo yo_10) ->
         A_1 : prop)]) Eo osetof_1
      ([(yo_10 : obj) =>
         ({def} (yo_10 Eo yo_10) ->
         A_1 : prop)])) =>
      ({def} rhyp_7 Mp Ocomp1_1 (osetof_1
      ([(yo_10 : obj) =>
         ({def} (yo_10 Eo yo_10) ->
         A_1 : prop)]), [(yo_9
         : obj) =>
```

```
          ({def} (yo_9 Eo yo_9) ->
          A_1 : prop)], rhyp_7) : that
       A_1)]) : that A_1)]))) Mp
 Deduction ([(rhyp_3 : that osetof_1
    ([(yo_6 : obj) =>
       ({def} (yo_6 Eo yo_6) -> A_1
       : prop)]) Eo osetof_1 ([(yo_6
       : obj) =>
       ({def} (yo_6 Eo yo_6) -> A_1
       : prop)])) =>
    ({def} rhyp_3 Mp Deduction ([(rhyp_5
       : that osetof_1 ([(yo_8 : obj) =>
          ({def} (yo_8 Eo yo_8) ->
          A_1 : prop)]) Eo osetof_1
       ([(yo_8 : obj) =>
          ({def} (yo_8 Eo yo_8) ->
          A_1 : prop)])) =>
       ({def} rhyp_5 Mp Ocomp1_1 (osetof_1
       ([(yo_8 : obj) =>
          ({def} (yo_8 Eo yo_8) ->
          A_1 : prop)]), [(yo_7
          : obj) =>
          ({def} (yo_7 Eo yo_7) ->
          A_1 : prop)], rhyp_5) : that
       A_1)]) : that A_1)]) : that
 A_1)]


Russellthm : [(A_1 : prop), (osetof_1
    : [(opred_2 : [(xo_3 : obj) =>
          (--- : prop)]) => (--- : obj)]), (Eo_1
    : [(ao_2 : obj), (bo_2 : obj) =>
       (--- : prop)]), (Ocomp1_1
    : [(.ao_2 : obj), (.opred_2 : [(xo_3
          : obj) => (--- : prop)]), (oelementev1_2
       : that .ao_2 Eo osetof_1 (.opred_2)) =>
       (--- : that .opred_2 (.ao_2))]), (Ocomp2_1
    : [(ao_2 : obj), (opred_2 : [(xo_3
```

```
          : obj) => (--- : prop)]), (oelementev2_2
       : that opred_2 (ao_2)) => (---
       : that ao_2 Eo osetof_1 (opred_2))]) =>
    (--- : that A_1)]


   {move 0}
end Lestrade execution
```

14 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.

```
begin 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}
end Lestrade execution
```

We introduce the false statement `??` (typeset notation for this is $\perp$) and introduce the rule that any proposition may be deduced from a false statement.

```
begin Lestrade execution

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


   ~ : [(Dd_1 : prop) =>
       ({def} Dd_1 -> ?? : prop)]


   ~ : [(Dd_1 : prop) => (--- : prop)]


   {move 0}
end Lestrade execution
```

We define negation.

```
begin 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 : [(.A_1 : prop), (B_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

   {move 2}

   >>> 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


   {move 1}

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


   Cases : [(.A_1 : prop), (.B_1 : prop), (.Dd_1
       : prop), (cases_1 : that .A_1 v .B_1), (case1_1
       : [(aa1_2 : that .A_1) => (---
          : that .Dd_1)]), (case2_1 : [(bb1_2
          : that .B_1) => (--- : that .Dd_1)]) =>
       (--- : that .Dd_1)]


   {move 0}
end Lestrade execution
```

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

```
begin Lestrade execution

   >>> postulate Exists tpred prop


   Exists : [(.T_1 : type), (tpred_1
       : [(tt1_2 : in .T_1) => (--- : prop)]) =>
       (--- : prop)]


   {move 0}

   >>> declare t3 in T
```

```
t3 : in T


{move 1}

>>> declare existsev that tpred t3


existsev : that tpred (t3)


{move 1}

>>> postulate Egen0 tpred, t3 existsev \
    : that Exists tpred


Egen0 : [(.T_1 : type), (tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (t3_1
    : in .T_1), (existsev_1 : that tpred_1
    (t3_1)) => (--- : that Exists (tpred_1))]


{move 0}

>>> define Egen1 t3 existsev : Egen0 tpred, t3 \
    existsev


Egen1 : [(.T_1 : type), (.tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (t3_1
    : in .T_1), (existsev_1 : that .tpred_1
    (t3_1)) =>
    ({def} Egen0 (.tpred_1, t3_1, existsev_1) : that
    Exists (.tpred_1))]
```

```
Egen1 : [(.T_1 : type), (.tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (t3_1
    : in .T_1), (existsev_1 : that .tpred_1
    (t3_1)) => (--- : that Exists (.tpred_1))]


{move 0}

>>> define Egen2 tpred, existsev : Egen0 \
    tpred, t3 existsev


Egen2 : [(.T_1 : type), (tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (.t3_1
    : in .T_1), (existsev_1 : that tpred_1
    (.t3_1)) =>
    ({def} Egen0 (tpred_1, .t3_1, existsev_1) : that
    Exists (tpred_1))]


Egen2 : [(.T_1 : type), (tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (.t3_1
    : in .T_1), (existsev_1 : that tpred_1
    (.t3_1)) => (--- : that Exists
    (tpred_1))]


{move 0}

>>> declare existsev2 that Exists tpred


existsev2 : that Exists (tpred)


{move 1}
```

```
>>> declare witness in T


witness : in T


{move 1}

>>> declare witnessev that tpred witness


witnessev : that tpred (witness)


{move 1}

>>> declare witnessprf [witness, witnessev \
       => that Dd]


witnessprf : [(witness_1 : in T), (witnessev_1
    : that tpred (witness_1)) => (---
    : that Dd)]


{move 1}

>>> postulate Einst existsev2, witnessprf \
    that Dd


Einst : [(.T_1 : type), (.tpred_1
    : [(tt1_2 : in .T_1) => (--- : prop)]), (.Dd_1
    : prop), (existsev2_1 : that Exists
    (.tpred_1)), (witnessprf_1 : [(witness_2
       : in .T_1), (witnessev_2 : that
       .tpred_1 (witness_2)) => (---
       : that .Dd_1)]) => (--- : that
```

```
    .Dd_1)]


   {move 0}
end Lestrade execution
```

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 three different additional versions of existential instantiation with different choices of explicit arguments are given.

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

```
begin 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
```

```
   {move 2}

   >>> declare nega1 that ~ Dd


   nega1 : that ~ (Dd)


   {move 2}

   >>> define howler nega1 : absurd


   howler : [(nega1_1 : that ~ (Dd)) =>
       (--- : that ??)]


   {move 1}

   >>> close


{move 1}

>>> define Panic0 absurd Dd : Dneg (Deduction \
    howler)


Panic0 : [(absurd_1 : that ??), (Dd_1
    : prop) =>
    ({def} Dneg (Deduction ([(nega1_3
       : that ~ (Dd_1)) =>
       ({def} absurd_1 : that ??)])) : that
    Dd_1)]
```

```
   Panic0 : [(absurd_1 : that ??), (Dd_1
       : prop) => (--- : that Dd_1)]


   {move 0}
end Lestrade execution
```

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 $\neg A \rightarrow B$ 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.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> declare side1 that (~ A) -> B


      side1 : that ~ (A) -> B


      {move 2}
end Lestrade execution
```

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

```
begin Lestrade execution

      >>> open
```

```
        {move 3}

        >>> declare contrahyp that ~ (A v B)


        contrahyp : that ~ (A v B)


        {move 3}
end Lestrade execution
```

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

```
begin Lestrade execution

        >>> open


           {move 4}

           >>> 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


{move 3}

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


thusnota : [(contrahyp_1 : that
    ~ (A v B)) => (--- : that
    ~ (A))]


{move 2}
end Lestrade execution
```

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 $X \rightarrow \perp$.

```
begin 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}
end Lestrade execution
```

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.

```
begin Lestrade execution

        >>> close


      {move 2}

      >>> define classicalor1 side1 : Dneg \
```

```
        (Deduction thuscontra1)


    classicalor1 : [(side1_1 : that ~ (A) ->
        B) => (--- : that A v B)]


    {move 1}
end Lestrade execution
```

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 $\neg A \to B$ to evidence that $A \vee B$.

```
begin Lestrade execution

    >>> declare side2 that A v B


    side2 : that A v B


    {move 2}
end Lestrade execution
```

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

```
begin Lestrade execution

    >>> open


      {move 3}

      >>> declare ahyp1 that ~ A
```

```
        ahyp1 : that ~ (A)


        {move 3}
end Lestrade execution
```

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.

```
begin Lestrade execution

        >>> open


           {move 4}

           >>> 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}
end Lestrade execution
```

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

```
begin Lestrade execution

            >>> declare ifb2 that B


            ifb2 : that B


            {move 4}

            >>> define ifb21 ifb2 : ifb2


            ifb21 : [(ifb2_1 : that B) =>
                (--- : that B)]


            {move 3}
end Lestrade execution
```

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

```
begin Lestrade execution

            >>> close


         {move 3}

         >>> define thusb2 ahyp1 : Cases \
             side2, ifa22, ifb21


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


         {move 2}
end Lestrade execution
```

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

```
begin Lestrade execution

         >>> close


      {move 2}

      >>> define classicalor2 side2 : Deduction \
          thusb2


      classicalor2 : [(side2_1 : that A v B) =>
          (--- : that ~ (A) -> B)]


      {move 1}
```

```
   >>> close


{move 1}

>>> define Classicalor1 A B : Deduction \
    classicalor1


Classicalor1 : [(A_1 : prop), (B_1
    : prop) =>
    ({def} Deduction ([(side1_2 : that
       ~ (A_1) -> B_1) =>
       ({def} Dneg (Deduction ([(contrahyp_4
          : that ~ (A_1 v B_1)) =>
          ({def} A_1 Addition2 (~ (A_1) propfixform
          Deduction ([(howabouta_9 : that
             A_1) =>
             ({def} B_1 Addition1 howabouta_9
             Mp contrahyp_4 : that ??)])) Mp
          side1_2 Mp contrahyp_4 : that
          ??)])) : that A_1 v B_1)]) : that
    (~ (A_1) -> B_1) -> A_1 v B_1)]


Classicalor1 : [(A_1 : prop), (B_1
    : prop) => (--- : 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) =>
    ({def} Deduction ([(side2_2 : that
       A_1 v B_1) =>
       ({def} Deduction ([(ahyp1_3
          : that ~ (A_1)) =>
          ({def} Cases (side2_2, [(ifa2_4
             : that A_1) =>
             ({def} ifa2_4 Mp ahyp1_3
             Panic B_1 : that B_1)], [(ifb2_4
             : that B_1) =>
             ({def} ifb2_4 : that B_1)]) : that
          B_1)]) : that ~ (A_1) ->
       B_1)]) : that (A_1 v B_1) ->
    ~ (A_1) -> B_1)]


Classicalor2 : [(A_1 : prop), (B_1
    : prop) => (--- : 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) =>
    ({def} ((~ (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)]


Classicalor : [(A_1 : prop), (B_1
```

```
      : prop) => (--- : that (~ (A_1) ->
      B_1) <-> A_1 v B_1)]


   {move 0}
end Lestrade execution
```

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 $\neg(A \to B)$ and $A \wedge \neg B$ in the same style.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> declare side1 that ~ (A -> B)


      side1 : that ~ (A -> B)


      {move 2}

      >>> open


         {move 3}

         >>> declare nota that ~ A
```

```
nota : that ~ (A)


{move 3}

>>> open


   {move 4}

   >>> 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


   {move 3}

   >>> 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


{move 2}

>>> define yesa side1 : Dneg (Deduction \
    iscontra)


yesa : [(side1_1 : that ~ (A ->
    B)) => (--- : that A)]
```

```
{move 1}

>>> open

   {move 3}

   >>> declare butb that B

   butb : that B

   {move 3}

   >>> open

      {move 4}

      >>> declare supposea that A

      supposea : that A

      {move 4}

      >>> define indeedb supposea : butb

      indeedb : [(supposea_1 : that
          A) => (--- : that B)]

      {move 3}

      >>> close
```

```
   {move 3}

   >>> 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


{move 2}

>>> 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


   {move 3}

   >>> 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 \
          side2)


      step21 : [(ifathenb_1 : that A ->
          B) => (--- : that ??)]


      {move 2}

      >>> close


   {move 2}

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


   negimp2 : [(side2_1 : that A & ~ (B)) =>
       (--- : that ~ (A -> B))]


   {move 1}

   >>> close


{move 1}

>>> define Negimp1 A B : Deduction negimp1
```

```
Negimp1 : [(A_1 : prop), (B_1 : prop) =>
    ({def} Deduction ([(side1_2 : that
       ~ (A_1 -> B_1)) =>
       ({def} Dneg (Deduction ([(nota_5
          : that ~ (A_1)) =>
          ({def} Deduction ([(buta_7
             : that A_1) =>
             ({def} buta_7 Mp nota_5 Panic
             B_1 : that B_1)]) Mp side1_2
          : that ??)])) Andproof ~ (B_1) propfixform
       Deduction ([(butb_5 : that B_1) =>
          ({def} Deduction ([(supposea_7
             : that A_1) =>
             ({def} butb_5 : that B_1)]) Mp
          side1_2 : that ??)]) : that
       A_1 & ~ (B_1))]) : that ~ (A_1
    -> B_1) -> A_1 & ~ (B_1))]


Negimp1 : [(A_1 : prop), (B_1 : prop) =>
    (--- : that ~ (A_1 -> B_1) -> A_1
    & ~ (B_1))]


{move 0}

>>> define Negimp2 A B : Deduction negimp2


Negimp2 : [(A_1 : prop), (B_1 : prop) =>
    ({def} Deduction ([(side2_2 : that
       A_1 & ~ (B_1)) =>
       ({def} ~ (A_1 -> B_1) propfixform
       Deduction ([(ifathenb_4 : that
          A_1 -> B_1) =>
          ({def} Simplification1 (side2_2) Mp
          ifathenb_4 Mp Simplification2
          (side2_2) : that ??)]) : that
```

```
        ~ (A_1 -> B_1))]) : that (A_1
      & ~ (B_1)) -> ~ (A_1 -> B_1))]


   Negimp2 : [(A_1 : prop), (B_1 : prop) =>
       (--- : that (A_1 & ~ (B_1)) ->
       ~ (A_1 -> B_1))]


   {move 0}

   >>> define Negimp A B : propfixform ((~ (A -> \
       B)) <-> A & ~ B, Andproof (Negimp1 \
       A B, Negimp2 A B))


   Negimp : [(A_1 : prop), (B_1 : prop) =>
       ({def} (~ (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))]


   Negimp : [(A_1 : prop), (B_1 : prop) =>
       (--- : that ~ (A_1 -> B_1) <-> A_1
       & ~ (B_1))]


   {move 0}
end Lestrade execution
```

We note that the more sophisticated namespace management made possible by the ability to save environments (moves) has applications relative to our philosophical motivations. If in world j we have not decided the truth value of a proposition p, we can in different moves with index $j+1$ postulate objects of sorts `that` p and `that` $\neg p$, and develop further declarations and definitions in these two contexts, switching back and forth at will betweeen the two developments. If one of p or $\neg p$ leads to contradiction, we will then have proved the other (if we are using classical logic) in the original world j, and we will be able to import any definitions with that premise from the appropriate world $j+1$ down into the original world j. Even more interesting, perhaps, is what happens if the question cannot be decided: we can continue to develop two different alternative pictures of the world, and anything that we can prove in both, we can import into the original world j (again, on the assumption that we have implemented classical logic). Our philosophical view supports classical logic, but it does not support the view that there is a fact of the matter with respect to (for example) the Continuum Hypothesis (a statement in set theory which is known to be undecidable); it suggests that we can explore mathematical universes in which CH holds, and mathematical universes in which ¬CH holds, and anything which follows from both hypotheses we can conclude is true in any universe satisfying our basic assumptions apart from CH, without presuming that we can or even should decide the question of CH one way or the other. We are not thereby supposing that there is a God's-eye view in which every question is resolved, though in some sense we may be providing support for the coherence of the latter view.

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

```
begin Lestrade execution

   >>> postulate V type


   V : type


   {move 0}

   >>> open
```

```
   {move 2}

   >>> declare Tt3 type


   Tt3 : type


   {move 2}

   >>> postulate typepred Tt3 prop


   typepred : [(Tt3_1 : type) => (---
       : prop)]


   {move 1}

   >>> close


{move 1}

>>> 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}
end Lestrade execution
```

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.

There is a further issue. The Ambiguity rule as stated would allow a rewrite rule to be declared with `rewrited` which would allow a rewrite of `setsof V` to `V`, which is disastrous. There are two approaches: we could continue to investigate the theory articulated above but with the constraint that we cannot freely use the rewrite logic, or formulate this a little differently.

```
begin Lestrade execution

   >>> declare V1 type


   V1 : type


   {move 1}

   >>> declare V2 type


   V2 : type


   {move 1}

   >>> declare V3 type


   V3 : type


   {move 1}

   >>> postulate << V1 V2 prop


   << : [(V1_1 : type), (V2_1 : type) =>
       (--- : prop)]


   {move 0}
```

```
>>> postulate Order1 V1 that V1 << setsof \
    V1


Order1 : [(V1_1 : type) => (--- : that
    V1_1 << setsof (V1_1))]


{move 0}

>>> declare order1 that V1 << V2


order1 : that V1 << V2


{move 1}

>>> declare order2 that V2 << V3


order2 : that V2 << V3


{move 1}

>>> postulate Ordertrans order1 order2 \
    that V1 << V3


Ordertrans : [(.V1_1 : type), (.V2_1
    : type), (.V3_1 : type), (order1_1
    : that .V1_1 << .V2_1), (order2_1
    : that .V2_1 << .V3_1) => (--- : that
    .V1_1 << .V3_1)]


{move 0}
```

```
>>> postulate maxtype V1 V2 type


maxtype : [(V1_1 : type), (V2_1 : type) =>
    (--- : type)]


{move 0}

>>> postulate Max1 V1 V2 that V1 << maxtype \
    V1 V2


Max1 : [(V1_1 : type), (V2_1 : type) =>
    (--- : that V1_1 << V1_1 maxtype V2_1)]


{move 0}

>>> postulate Max2 V1 V2 that V2 << maxtype \
    V1 V2


Max2 : [(V1_1 : type), (V2_1 : type) =>
    (--- : that V2_1 << V1_1 maxtype V2_1)]


{move 0}

>>> declare order3 that V1 << V3


order3 : that V1 << V3


{move 1}
```

```
>>> declare order4 that V2 << V3


order4 : that V2 << V3


{move 1}

>>> postulate Max3 order3 order4 that \
    (maxtype V1 V2) << V3


Max3 : [(.V1_1 : type), (.V2_1 : type), (.V3_1
    : type), (order3_1 : that .V1_1
    << .V3_1), (order4_1 : that .V2_1
    << .V3_1) => (--- : that (.V1_1
    maxtype .V2_1) << .V3_1)]


{move 0}

>>> postulate ambtype typepred type


ambtype : [(typepred_1 : [(Tt3_2 : type) =>
       (--- : prop)]) => (--- : type)]


{move 0}

>>> declare evid1 that typepred (ambtype \
    typepred)


evid1 : that typepred (ambtype (typepred))


{move 1}
```

```
>>> declare evid2 that (ambtype typepred) << \
    V1


evid2 : that ambtype (typepred) << V1


{move 1}

>>> postulate Ambiguity2 evid1 evid2 that \
    typepred V1


Ambiguity2 : [(.typepred_1 : [(Tt3_2
       : type) => (--- : prop)]), (.V1_1
    : type), (evid1_1 : that .typepred_1
    (ambtype (.typepred_1))), (evid2_1
    : that ambtype (.typepred_1) << .V1_1) =>
    (--- : that .typepred_1 (.V1_1))]


{move 0}
end Lestrade execution
```

The development above is consistent with Lestrade logic with rewriting and proves existence of types ambiguous for any desired concrete finite set of properties of types. The idea is that there is a transitive relation on types under which a type is below its power type and two types always have a maximum, and for every predicate of types there is an associated type such that the predicate has the same value on every type above the given type. Now for any collection of predicates of types (sentences true of those types) we can go to the maximum of their associated types, and above that point we have ambiguity for those sentences. We do not incur rewriting opportunities because we do not assert that any two types satisfy exactly the same predicates.

17 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.

18 A note on polymorphic typing

We provide the standard function type constructor.

```
begin Lestrade execution

   >>> declare tau8 type


   tau8 : type
```

```
{move 1}

>>> declare tau9 type


tau9 : type


{move 1}

>>> open


   {move 2}

   >>> declare x8 in tau8


   x8 : in tau8


   {move 2}

   >>> postulate f8 x8 in tau9


   f8 : [(x8_1 : in tau8) => (---
       : in tau9)]


   {move 1}

   >>> close


{move 1}
```

```
>>> postulate Arrowtype tau8 tau9 type


Arrowtype : [(tau8_1 : type), (tau9_1
    : type) => (--- : type)]


{move 0}

>>> declare ff8 in Arrowtype tau8 tau9


ff8 : in tau8 Arrowtype tau9


{move 1}

>>> declare xx8 in tau8


xx8 : in tau8


{move 1}

>>> postulate Arrowapp ff8 xx8 in tau9


Arrowapp : [(.tau8_1 : type), (.tau9_1
    : type), (ff8_1 : in .tau8_1 Arrowtype
    .tau9_1), (xx8_1 : in .tau8_1) =>
    (--- : in .tau9_1)]


{move 0}

>>> postulate Fun f8 in Arrowtype tau8 \
    tau9
```

```
   Fun : [(.tau8_1 : type), (.tau9_1
       : type), (f8_1 : [(x8_2 : in .tau8_1) =>
          (--- : in .tau9_1)]) => (---
       : in .tau8_1 Arrowtype .tau9_1)]


   {move 0}

   >>> postulate Arrowcomp f8, xx8 that \
       Arrowapp (Fun f8, xx8) = f8 xx8


   Arrowcomp : [(.tau8_1 : type), (.tau9_1
       : type), (f8_1 : [(x8_2 : in .tau8_1) =>
          (--- : in .tau9_1)]), (xx8_1
       : in .tau8_1) => (--- : that Fun
       (f8_1) Arrowapp xx8_1 = f8_1 (xx8_1))]


   {move 0}
end Lestrade execution
```

The following is a set of tentative Lestrade declarations supporting a popular brand of polymorphic type. We supply the arrow type above because any application of this polymorphic type scheme requires some prior type constructors to work with. This section may be expanded with some actual constructions to illustrate this.

The declarations here are very bare: for example, no extensionality principles are given. The arrow type declarations above are made at move 0 (they may be postulated and used later); the polymorphism constructors are kept in a sandbox environment, declared at move 1.

8/16/2017 entirely new polymorphism declarations

```
begin Lestrade execution
```

```
>>> open

   {move 2}

   >>> open

      {move 3}

      >>> declare tau50 type

      tau50 : type

      {move 3}

      >>> postulate taufun50 tau50 type

      taufun50 : [(tau50_1 : type) =>
          (--- : type)]

      {move 2}

      >>> postulate polyfun50 tau50 in \
          taufun50 tau50

      polyfun50 : [(tau50_1 : type) =>
          (--- : in taufun50 (tau50_1))]

      {move 2}

      >>> close
```

```
{move 2}

>>> postulate Polytype taufun50 type


Polytype : [(taufun50_1 : [(tau50_2
       : type) => (--- : type)]) =>
    (--- : type)]


{move 1}

>>> postulate Polyfun polyfun50 in \
    Polytype taufun50


Polyfun : [(.taufun50_1 : [(tau50_2
       : type) => (--- : type)]), (polyfun50_1
    : [(tau50_2 : type) => (---
       : in .taufun50_1 (tau50_2))]) =>
    (--- : in Polytype (.taufun50_1))]


{move 1}

>>> declare polyobj50 in Polytype taufun50


polyobj50 : in Polytype (taufun50)


{move 2}

>>> declare tau51 type
```

```
tau51 : type


{move 2}

>>> postulate Polyapp polyobj50 tau51 \
    : in taufun50 tau51


Polyapp : [(.taufun50_1 : [(tau50_2
       : type) => (--- : type)]), (polyobj50_1
    : in Polytype (.taufun50_1)), (tau51_1
    : type) => (--- : in .taufun50_1
    (tau51_1))]


{move 1}

>>> postulate Polycomp polyfun50, tau51 \
    that ((Polyfun polyfun50) Polyapp \
    tau51) = polyfun50 tau51


Polycomp : [(.taufun50_1 : [(tau50_2
       : type) => (--- : type)]), (polyfun50_1
    : [(tau50_2 : type) => (---
       : in .taufun50_1 (tau50_2))]), (tau51_1
    : type) => (--- : that Polyfun
    (polyfun50_1) Polyapp tau51_1
    = polyfun50_1 (tau51_1))]


{move 1}

>>> open


   {move 3}
```

```
>>> declare tau50 type


tau50 : type


{move 3}

>>> define taufuntest tau50 : tau50 \
    Arrowtype tau50


taufuntest : [(tau50_1 : type) =>
    (--- : type)]


{move 2}

>>> open


   {move 4}

   >>> declare x50 in tau50


   x50 : in tau50


   {move 4}

   >>> define testid x50 : x50


   testid : [(x50_1 : in tau50) =>
       (--- : in tau50)]
```

```
      {move 3}

      >>> close


   {move 3}

   >>> define polyfuntest tau50 : Fun \
       testid


   polyfuntest : [(tau50_1 : type) =>
       (--- : in tau50_1 Arrowtype
       tau50_1)]


   {move 2}

   >>> close


{move 2}

>>> define Polyfuntest : Polyfun polyfuntest


Polyfuntest : in Polytype ([(tau50'_2
    : type) =>
    ({def} tau50'_2 Arrowtype tau50'_2
    : type)])


{move 1}

>>> define Polyapptest : (Polyfuntest \
    Polyapp Nat) Arrowapp 0
```

```
Polyapptest : in Nat


{move 1}

>>> define Polycomptest : Polycomp \
    polyfuntest, Nat


Polycomptest : that Polyfun ([(tau50_4
    : type) =>
    ({def} Fun ([(x50_5 : in tau50_4) =>
       ({def} x50_5 : in tau50_4)]) : in
    tau50_4 Arrowtype tau50_4)]) Polyapp
 Nat = Fun ([(x50_3 : in Nat) =>
    ({def} x50_3 : in Nat)])


{move 1}

>>> open


   {move 3}

   >>> declare Y50 in Nat Arrowtype \
       Nat


   Y50 : in Nat Arrowtype Nat


   {move 3}

   >>> define noncepred Y50 : ((Polyfuntest \
       Polyapp Nat) Arrowapp 0) = Y50 \
       Arrowapp 0
```

```
   noncepred : [(Y50_1 : in Nat Arrowtype
       Nat) => (--- : prop)]


   {move 2}

   >>> close


{move 2}

>>> define Polycomptest2 : Substitution0 \
    noncepred, Polycomptest, Reflexeq \
    ((Polyfuntest Polyapp Nat) Arrowapp \
    0)


Polycomptest2 : that Polyfuntest Polyapp
 Nat Arrowapp 0 = Fun ([(x50_4 : in
    Nat) =>
    ({def} x50_4 : in Nat)]) Arrowapp
 0


{move 1}

>>> open


   {move 3}

   >>> declare n50 in Nat


   n50 : in Nat
```

```
   {move 3}

   >>> define natid n50 : n50


   natid : [(n50_1 : in Nat) =>
       (--- : in Nat)]


   {move 2}

   >>> close


{move 2}

>>> declare qqq in Nat


qqq : in Nat


{move 2}

>>> define Polycomptest3 : Substitution \
    (Arrowcomp (natid, 0), Polycomptest2)


Polycomptest3 : that Polyfuntest Polyapp
 Nat Arrowapp 0 = 0


{move 1}

>>> close
```

```
   {move 1}
end Lestrade execution
```

19 Introduction to Lestrade

This section contains a formal discussion of Lestrade (the framework and the software), versions of parts of which are already embedded in the discussion of the development of particular Lestrade theories which precedes this. The discussion here is generally more detailed and can be consulted for reference. Note that the phrase "the current move" is used for move i in this section, where "the last move" is used above. These usages are equivalent: in terms of our temporal metaphor, "the present" is right after the last declaration in move i and right before the first declaration in move $i + 1$.

19.1 Introduction

Lestrade is a general purpose logical framework for mathematics. It is motivated by a philosophical premise: contrary to the statements of its founders, practitioners, and detractors (when they say anything about philosophical matters at all), modern foundational mathematics, with classical logic and including Cantorian set theory, does not depend on actual infinities. All activities in mathematics can be viewed as finitary, or at least as involving no more than potential infinities, and this does not make classical logic or the Cantorian transfinite mathematics illegitimate.

Such a claim might be taken to be rather startling. We support it with the development of a computer implementation of the framework (the Lestrade Type Inspector) in which one can actually conduct mathematical investigations in the style suggested by the underlying philosophical view. Some aspects of the formal framework have been strongly shaped by what might seem accidental consequences of the way we have implemented the software. Some of these features are not in our opinion accidental. Of course, some features of the software and of the framework as formally presented are the results of design choices which could have been made differently: we will indicate some choices we have made which were not essential.

Lestrade uses the idea that mathematical propositions and their proofs are among the objects of mathematics. It uses a version of the Curry-Howard isomorphism, which is usually associated with constructive logic (which can be implemented in Lestrade) but can, as here, be used to motivate classical logic. As the mention of the Curry-Howard isomorphism should indicate, Lestrade is a dependent type system.

The view taken of functions in Lestrade is not the standard one. We do not view functions as completely given infinite tables of values (this would quite defeat our basic philosophical premise!); instead, we regard a function as a machine which will return an output (of a type given in advance, possibly depending on input values) given a sequence of inputs (later ones possibly of types depending on earlier ones). When a function is given by an expression, we can reasonably say that we have finitely described the action of the function given any inputs that may later be presented to us. We do not, however, assume that an arbitrarily given function is determined by some unknown expression; our language is not regarded as constraining what functions are possible. Theorems about functions provable by induction on the structure of the expressions we are able to define so far may suggest themselves as further axioms, but our framework does not presume that such principles are true.

Lestrade can be viewed as a version of Automath, though there are considerable differences. Since it is related to Automath, it is also more distantly related to other systems such as Coq with this genealogy.

19.2 Metaphysics of Lestrade

The things we talk about in Lestrade we will refer to as *entities* when we are being completely general. The entities fall into two large categories, *objects* and *functions*. Each object or function has a sort (the word *type* is reserved for a specific variety of sert, as we will see shortly).

The sorts of object can be reviewed quickly.

1. There is a sort `prop` intended to be inhabited by mathematical propositions.

2. For each p of sort `prop`, there is a sort `that` p inhabited by evidence for p. A proof of p is evidence for p, of course, but we do not hold that evidence for p must be a formal proof as such. If `that` p is inhabited, we do take p to be true: there is no probable evidence here.

3. There is a sort `obj` intended to be inhabited by "untyped mathematical objects". In an implementation of ZFC, for example, the sets would

be of sort `obj`. Note that other sorts would be in use, in spite of the fact that ZFC is an untyped theory, because its propositions and their proofs (or evidence) would be objects of other sorts.

4. There is a sort `type` inhabited by "type labels". A typical example would be `Nat`, the type of natural numbers. We find it useful to call these objects type labels in order to resist the temptation to view them as (necessarily usually infinite) collections containing all the objects of the indicated type, which would tell against our philosophical premise.

5. For each type (label) τ, there is a sort `in` τ inhabited by the objects of type τ. A natural number n would have sort `in Nat`. We say that an object of sort `in` τ is of type τ.

And that is all. But it turns out to be quite a lot, once the additional apparatus of functions is introduced.

The analogy between `prop/that` and `type/in` is almost perfect in Lestrade: the analogy is perfect in the core functions of the Type Inspector, though the symmetry is likely to be collapsed by postulates in particular theories, where we may not want to treat proofs/evidence for propositions in the same way as mathematical objects of various types. The rewriting feature of the old implementation of Lestrade did break the symmetry, treating propositions differently from type labels: this feature is not yet present in the new implementation, and could easily be implemented in an entirely symmetrical manner.

A function in Lestrade takes a fixed positive number of arguments. The sort of a function is determined by the sorts of its arguments and its output, with the further subtlety that the sort of each argument may depend on the values of earlier arguments and the sort of the output may depend on the values of the arguments.

The general notation for a function sort is

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

The variables x_i are bound in this notation, and bound variables in distinct function sort notations are viewed as distinct. Each τ_i is the sort of x_i, and may contain x_j's only for $j < i$. The τ_i's may be object or function sorts. The notation τ stands for the output sort, which must be an object sort and may contain any of the x_i's.

A user of the Lestrade Type Inspector never writes a function sort notation (or the kind of notation for specific functions presented just below): Lestrade does present such notations in output. The purpose of presenting the notation at this point is to explain what sorts of function there are.

A general notation for a function given by an explicit definition $y = f(x_1, \ldots, x_n)$ is

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

where of course each x_i has type τ_i and y must have sort τ and

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

will be its sort (which imposes conditions on the τ_i's and τ which are described above).

That is the complete sort system of Lestrade. The rest is user postulation of objects and functions of different types, and user development of useful definitions from the primitive constructions, where "user" may indifferently mean "user of the Lestrade Type Inspector" or "developer of a mathematical theory in the Lestrade framework".

In the new implementation, there is a new feature (invisible for the most part in its default settings) allowing formal arguments in abstractions which are in effect `let` expressions. We do not discuss this aspect here, except to note that it makes the analogy between the list of binders in a Lestrade lambda term or function sort and the list of declarations in a move perfect, which was not the case in the old implementation: let clauses in abstractions correspond to definitions in moves.

19.3 The Lestrade Environment: a metaphor for mathematical activity

The Lestrade environment is in concrete terms a finite sequence of finite lists of declarations of identifiers. Each declaration is a pair whose first component is an identifier and whose second component is a function sort notation or a function notation[5]. This sequence always has at least two lists of declarations

[5]The second component of the form $[(y, \tau)]$ in the declaration of a defined object can be thought of as a notation for a constant function to be applied to an empty argument list, though Lestrade does not explicitly treat it that way.

in it, and we refer to its length as $i+2$. The jth list will be referred to as "move j" (because we have a temporal metaphor for what is going on, though we are certainly not talking about physical time). Move $i+1$ is called "the next move" and move i is called "the current move" (or "the last move", in some documents; the idea is that the "present" is right after the end of the current/last move, and just before all the declarations in the next move).

We should think of each move as a possible world in some modality. We use a temporal metaphor: the current move and the previous moves are as it were "past" and the items declared there are constant: the next move is "future" and the items declared there are variable. The concrete mathematical actions we carry out should make it clearer what is going on.

All identifiers declared in all the moves are distinct (under normal circumstances: see below for the way this can be subverted). We think of the entities represented by those identifiers as having been discovered in the order of the moves, and within each move in the order in which they are listed. This is useful for getting dependencies to work correctly without rather laborious checks.

We introduce the six core commands of Lestrade, the declaration commands `declare`, `postulate`, and `define`, and the environment handling commands `open`, `close`, and `clearcurrent`.[6]

declare: An instance of the `declare` command is of the form `declare` i τ, where i is a fresh identifier and τ is an expression for an object or function sort (which of course may not contain any identifier which has not been declared previously). The effect of the command is to introduce a variable i of sort τ, an item in the next move.

postulate: An instance of the `postulate` command is of the form

`postulate` f `args` (:) τ

The component f is a fresh identifier.

The component `args` is an argument list, which may be empty, or may be of the form $x_1, \ldots, x_n$, where the x_i's are identifiers previously declared in the next move, none of which were introduced using the

[6]In the new implementation, `declare`, `postulate`, and `define` are implemented in terms of two new commands `Declare` and `Posit`, which are also available to users but which we do not describe here. Anything that can be done with the new commands can be done with the original set.

`define` command, appearing in the order in which they appear in the next move (if they do not, they will be reordered). This order restriction automatically enforces dependency relations on the x_i's.

The component τ must be an expression for an object sort.

All non-defined identifiers declared at the next move on which τ or the sort τ_i of each x_i depends must be either among the x_k's in `args` or should appear in the sort of some x_k. Lestrade will insert any non-defined identifiers found in τ or some τ_i into its internal representation of the argument list of f at the point indicated by order of declaration; when Lestrade evaluates f at particular lists of arguments, it will attempt to deduce the values of these implicit arguments [and may fail: this is not a failure of the type system, but a failure of Lestrade input/output, as it were, and such failures can always be avoided by re-declaring the function with more arguments given explicitly]. The details of the implicit argument scheme are a complication here, but must be mentioned as the implicit argument scheme is very useful and is used immediately in examples.

The effect of the command is to enhance the argument list to an argument list $x'_1, \ldots, x'_m$, appearing in the order in which they are declared in the next move and with every identifier declared at the next move and appearing in τ or any of the τ_i's (or indeed any of the sorts τ'_i of x'_i's) appearing as an x'_j, and then declare the fresh identifier f as having sort

$$(x'_1, \tau'_1), \ldots, (x'_m, \tau) \Rightarrow (-, \tau),$$

appending this declaration to *the current move*.

If `args` is empty, the effect of the command is to declare f of type τ at the current move rather than the next move: this is a declaration of a constant.

The colon before the τ is now (we believe) always optional: earlier it was sometimes required for things to parse correctly.

When executed when the next move is move 1, the `postulate` command should be thought of as introducing axioms and primitive notions. When used at later moves in combination with the `open` and `close` commands to be discussed below, it is used to introduce function variables (and for other purposes, as for example to introduce hypothetical primitives or axioms).

define: An instance of the `define` command is of the form

`define` f `args` : y

The component f is a fresh identifier.

The component `args` is an argument list, which may be empty, or may be of the form $x_1, \ldots, x_n$, where the x_i's are identifiers previously declared in the next move, none of which were introduced using the `define` command [we will see here that such declarations have distinctive features], appearing in the order in which they appear in the next move (if they do not, they will be reordered). This order restriction automatically enforces dependency relations on the x_i's.

The component y must be an expression representing an object, whose sort we denote by τ.

All non-defined identifiers declared at the next move on which y, τ or the sort τ_i of each x_i depend must be either among the x_k's in `args` or should appear in the sort of some x_k. Lestrade will insert any non-defined identifiers found in y, τ, or some τ_i and not found among the arguments into its internal representation of the argument list of f at the point indicated by order of declaration; when Lestrade evaluates f at particular lists of arguments, it will attempt to deduce the values of these implicit arguments [and may fail: implicit argument handling is not a failure of the type system, but a failure of Lestrade input/output, as it were, and such failures can always be avoided by re-declaring the function with more arguments given explicitly]. The details of the implicit argument scheme are a complication here, but must be mentioned as the implicit argument inference scheme is very useful and is used immediately in examples.

The effect of the command is to enhance the argument list to an argument list $x'_1, \ldots, x'_m$, appearing in the order in which they are declared in the next move and with every identifier declared at the next move and appearing in y, τ, or any of the τ_i's (or indeed any of the sorts τ'_i of x'_i's) appearing as an x'_j, and then declare the fresh identifier f as having sort

$$(x'_1, \tau'_1), \ldots, (x'_m, \tau) \Rightarrow (y, \tau),$$

appending this declaration to *the current move*.

Of course, f is actually being declared as of sort

$$(x'_1, \tau'_1), \ldots, (x'_m, \tau) \Rightarrow (-, \tau),$$

with y serving as an annotation that it is a specific function.

If `args` is empty, the effect of the command is to declare f of type (y, τ) [really, as being of type τ with the additional data of its particular identity as y] at the current move rather than the next move: this is a declaration of a defined constant.

The colon before y is required.

remarks on postulate and define commands: In either the `postulate` or `define` commands, in terms of the metaphor, f is declared as a presently given object, which, whatever x_i's of type τ_i may be given in the future, will return an output of type τ. In the case of the `define` command, we are specifically told what object will be returned in each case (we have a template into which to insert the arguments); in the case of the `postulate` command we suppose that the values will be presented on demand on presentation of appropriately sorted inputs. In neither case are we obliged to suppose that we know all the values at once. Even in the case of the `define` command, executing the definition with a particular list of arguments may cause us to request as yet unknown values of primitive functions introduced by the `postulate` command (indeed, this will almost certainly be the case).

Part of the commitment is that if we are given $x_1, \ldots, x_n$, we can deduce the implied values of $x'_1, \ldots x'_m$ from the sorts of the explicitly given arguments: it should be noted that this can fail at run-time as it were, as we may see in examples. The implicit argument inference feature is actually a feature of the input/output of Lestrade: as far as the sort checker (which is the heart of Lestrade) is concerned, all functions have all of their required arguments. A problem with implicit argument inference can always be solved by using a version of the offending function with all arguments given explicitly.

remarks on the environment commands: The commands `open`, `close`, and `clearcurrent` manipulate the environment. Only their core behavior is described here. There is a system for saving and restoring moves, and in this context some of these commands may appear with names of saved moves as arguments: these uses are not described here.

open: The `open` command adds a new empty list of declarations to the end of the environment. This has the effect of incrementing the parameter i and changing the identities of the current move and the next move: the old next move becomes the new current move, and the new empty move is the new next move.

close, clearcurrent: The `close` command can only be executed if $i > 0$. It simply deletes the next move. The environment is shortened by one, the old current move becomes the next move and the old move $i-1$ becomes the current move (again; it must have been at some time before). The `clearcurrent` command is a variant of `close`: it replaces the next move with an empty list (clearing all variable declarations, as it were) but leaves the length of the environment unchanged. The command `clearcurrent` is required as an independent command because `close` cannot be used to clear declarations in move 1.

function variables: Now we can explain how to generate function parameters. To introduce a function parameter f, execute the `open` command, introduce parameters for f of the desired types, then use the `postulate` command to declare f, followed by the `close` command, which leaves us with f of the desired type declared at the next move. If a function parameter itself requires parameters of function sorts, this may require repeated use of `open` and `close`, but it can be done.

variable expressions: Defined identifiers declared at the next move are as it were complex variable expressions. When they are used in the expression y in a `define` command, they must be expanded out: where defined constants appear, they are expanded in the obvious way; where defined functions appear in applied position, their application is carried out; where defined functions appear as arguments they are replaced with their function notation with bound variables given above: this must be done because a declaration at the current move cannot depend on a defined identifier at the next move, whose declaration disappears when the `close` command is executed. The variable parameters of a `postulate` command are, as we will see, replaced by bound variables [differentiated by applying a fresh numerical index to each variable, preserving the condition on Lestrade expressions that identical bound variables are always associated with the same instance of that variable as a binder] and their types are supplied as part of the information in the function sort reported for the constructed identifier: in any case, no declaration in a particular move can depend on a declaration appearing in a later move or later in the same move. Defined identifiers declared in the current move or previous moves do not need to be expanded out.

19.3.1 Namespace management refined: saving and retrieving environments

With the limited environment handling given above, there is no way to remove or revise declarations of variables and variable expressions in move 1 other than clearing all of them. After a while, it is quite hard to remember what sorts have been assigned to parameters and variable expressions, and for that matter what order they appear in (recalling that parameters in `postulate` and `define` commands must appear in order of declaration). We have already noted that the `clearcurrent` command will clear all declarations at the next move.

More intelligent namespace management is supported by the full specification of the `open`, `clearcurrent`, and `save` commands.

Each move is assigned a name. The default name is its numeral index (the j such that it is move j). The command `save envname` will save the next move with the name `envname`, associated with the list of names of preceding moves at the time it is saved (a saved move is actually identified by the sequence of names of all moves at the time it is saved, and this is how it is identified internally; this means that moves saved in different contexts can quite safely be tagged with the same name). The command `open envname` will open an already existing move (of the right index, with the same preceding moves) with the name `envname` or if there is no such move, or create a new blank move with that name. The command `clearcurrent envname` will clear the net move and replace it with a move named `envname` if there is such a move with the appropriate preceding names of moves associated with it or replace it with a blank move of that name otherwise (and will always replace it with a blank move with the default numeral name if it has no argument). Lestrade discards extensions of saved moves when they are overwritten and discards saved moves with their default numeral name immediately after they are reopened.

It is possible that identifiers declared in a saved move may conflict with identifiers declared after the move is saved and closed in moves with smaller index: there will then be a possible conflict of meaning when the saved move is reopened. In such cases, user-entered instances of that identifier will be interpreted as taken from the latest move possible, but occurrences of the identifier with other meanings will be handled correctly. Such masking of declarations is probably best avoided but we believe everything will work correctly if it occurs.

This means that instead of having a linear sequence of moves, which we can think of as times or possible worlds, we have a tree structure. Each node in the tree of moves has a name; different nodes may actually have the same name, since the identity of a node is determined by the sequence of names of the nodes on the branch leading to it (including its own name), not the name attached to it by itself.

19.4 Lestrade Notation

In this section we describe the notation which the user can enter for objects, functions, and object and function sorts.

An identifier is a string of characters of positive length which consists of zero or one upper case letters followed by zero or more lower case characters followed by zero or more digits then by zero or more single quotes, or consists of zero or one special characters taken from a list

```
~!@#$%^&*-+=/<>|?
```

followed by zero or more digits then by zero more more single quotes.

An object notation is either an identifier declared of object type or an object notation enclosed in parentheses or a notation $f(t_1, \ldots, t_n)$ or $t_1\,f\,t_2, \ldots, t_n$ (synonymous) [in the mixfix notation t_1 must be enclosed in parentheses if it is a function identifier and $n > 1$ is required] where f is declared as a function identifier of appropriate type, or a variant of the latter two notations obtained by dropping some of the parentheses and commas. The rule is that the parser will read as long an expression as possible before performing any sort checking. Thus, commas (or close parentheses, or colons) must appear after function identifiers used as arguments to keep them from being applied to following terms, and preceded with commas [or sometimes other punctuation] to keep them from absorbing previous terms as an infix. The parser will read a parenthesis following a non-infix function identifier as starting an argument list, so if it is desired to enclose a first argument in parentheses, it is also necessary to enclose the entire argument list. It should be remembered that in effect all operations have the same precedence and group to the right as in the ancient language APL. Problems with user input can always be avoided by putting in more parentheses and commas. Lestrade output will show as many parentheses and commas as possible, and will use infix notation for operators of arity 2 which have first arguments which are objects. Lestrade output will never use mixfix notation with more than one argument after the function symbol. The new implementation generalizes the treatment of prefix unary operators as having higher binding strength by requiring that any prefix term in which the argument list is not enclosed by parentheses has final argument not a mixfix term.

Lestrade user function notations are either identifiers declared as functions or or function notations enclosed in parentheses or notations $f(t_1, \ldots, t_m)$ where m is less than the number of arguments which f requires: such a term represents the function

$$(x_{m+1}, \ldots, x_n) \Rightarrow f(t_1, \ldots, t_m, x_{m+1}, \ldots, x_n).$$

, can appear only as an argument (not in applied or infix position), and the parentheses around the argument list are required, or, finally, an actual lambda term `[x1 ... xn => t]` in which the bound variables x_i must be declared in the next move and t is an object term: again, this lambda term can only appear in argument position, not in applied position or as an infix.

Lestrade user object sort notations are then `prop`, `that` p where p is an expression for an object of sort `prop`, `obj`, `type`, or `in` τ where τ is an expression for an object of sort `type`. Lestrade function sort notations are of the form `[x1 ... xn => tau]` in which the bound variables x_i must be declared in the next move and τ is an object term. Function sort notation can only appear in the `declare` command (of the commands considered here: it can appear in a `Declare` or `Posit` command).

In a Lestrade declaration line (with any of the first three commands) the identifier is always the second component of the command: declarations always present the declared identifier in prefix notation. The arguments may be separated with commas as required, and the argument list (which is never enclosed in parentheses) may be terminated with a colon `:` if required. This is mandatory in a `define` command and we believe now always optional (but allowed) in a `postulate` command. A colon will never appear in a `declare` command, which does not contain an argument list.

19.5 Lestrade Sort Checking and Definition Expansion

An identifier will have a sort determined by lookup in the environment.

We use the notation $T[t/x]$ for the result of replacing the variable x with the term t in the term T. Of course a formal definition of substitution in the presence of variable binding constructions requires care (and is not given here).

A function application term $f(t_1, \ldots, t_n)$ or $t_1\, f\, t_2, \ldots, t_n$, for simplicity supposed supplied with all needed arguments (we suppose that the implicit argument inference algorithm has already been carried out), where f has type

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

will sort check if t_1 is of type τ_1 and if either $n = 1$ (in which case its sort is $\tau[t_1/x_1]$) or if $n > 1$ and if we introduce f', a new function symbol of type

$$(x_2, \tau_2[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1]) \Rightarrow (-, \tau[t_1/x_1])$$

and $f'(t_2, \ldots, t_n)$ is well-typed [in this case, we return the sort of $f'(t_2, \ldots, t_n)$ as the sort of the original expression].

Where f is defined as

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

we evaluate $f(t_1, \ldots, t_n)$ very similarly. If $n = 1$ we return $y[t_1, x_1]$. If $n > 1$, we define a new function symbol f' for

$$(x_2, \tau_2[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1]) \Rightarrow (y[t_1, x_1], \tau[t_1/x_1])$$

and return $f'(t_2, \ldots, t_n)$.

Expansion of definitions is employed in two different contexts. Whenever an identifier passes out of scope (when a defined identifier in the next move is used in the definition of an entity introduced at the current move), this identifier will be expanded. If it is a defined object, the identifier is replaced with its definition. If it is a defined function applied to arguments, the appropriate values of the arguments are substituted into its definition. If it is a function appearing as an argument, it is replaced by the anonymous notation for that function used in its declaration (something similar happens if a function with a truncated argument list appears as an argument). The second use of definitional expansion is in matching (determining when two expressions are the same): the matching facility will check by carrying out definitional expansions behind the scenes to determine whether two expressions being compared have the same meaning. This use of definitional expansion does not lead to the expansion of defined terms in Lestrade output.