A concrete approach to mathematics?

M. Randall Holmes

November 7, 2021

Contents

1 Introduction

In this essay we consider a view of the modern mathematics of the continuous and the infinite which views the objects and the arguments of mathematics as objects which can be concretely presented as inhabitants of data types representable in practice on a computer. We illustrate this using actual general purpose software with this aim.

The software we use is called the Lestrade Type Inspector, implementing a formal system called Lestrade. We will initially present Lestrade code in contexts where it is reasonably clear what it does. A formal description will eventually appear.

The ontology for mathematics (and mathematical arguments) that we employ is typed. Mathematical objects do come in kinds. Both new objects and new sorts of objects can be introduced in Lestrade theory development, as we will see.

2 Propositions

"Proposition" is a fancy word for a declarative sentence. A proposition is something we can say which is either true or false.

For us, propositions are among the entities that mathematics studies. In the Lestrade software which we use as a tool in this work, the kind of entity "proposition" is a built-in (as most logical concepts are not: we will introduce those by explicit declaration in the sections that follow, and Lestrade supports approaches to logic different from the one we have taken).

```
begin Lestrade execution

   >>> declare A prop

   A : prop

   {move 1}

   >>> declare B prop

   B : prop
   {move 1}
end Lestrade execution
```

This document is among other things a Lestrade script. The commands behind `>>>` were typed in by the author, but the responses were generated by the Inspector. Here we have declared variable parameters A and B of type `prop`. The sort `prop` is one of the few Lestrade built-ins, the sort inhabited by propositions.

There will be operations on propositions which yield other propositions. We provide an example declaration.

```
begin Lestrade execution

   >>> postulate & A B prop

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

   {move 0}
end Lestrade execution
```

We postulate an operation & which takes two proposition arguments and has proposition output. This is not enough of course to tell us that this operation is conjunction ("and"), as the symbol for it suggests might be our intention.

Propositions are true or false. A true proposition has proofs, or other evidence that it is true. We think that these proofs (or other evidence) are themselves entities in the purview of our study.

```
begin Lestrade execution

   >>> declare aa that A

   aa : that A
   {move 1}

   >>> declare bb that B

   bb : that B

   {move 1}

   >>> declare cc that A & B

   cc : that A & B

   {move 1}
end Lestrade execution
```

With each proposition p (object of type `prop`) we associate a new type `that` p, which is inhabited by evidence for p (one could say, by proofs of p, but we have a philosophical objection to this which we will state below). In any case, if there is an object of type `that` p, it is to be understood that p is true. The `that` operation on propositions is a built-in of Lestrade.

Here we have declared parameters aa, bb, and cc which range over evidence for the propositions $A, B, A \wedge B$ respectively.

One should also notice that Lestrade understands infix notation $A\&B$.

Propositions have subjects and predicates (and direct and indirect objects, if we think in grammatical terms).

```
begin Lestrade execution

   >>> declare x obj

   x : obj

   {move 1}

   >>> declare y obj
```

```
y : obj

{move 1}

>>> postulate Blue x prop

Blue : [(x_1 : obj) => (--- : prop)]

{move 0}

>>> postulate Sam obj

Sam : obj

{move 0}

>>> declare blueev that Blue Sam

blueev : that Blue (Sam)

{move 1}

>>> postulate Taller x y prop

Taller : [(x_1 : obj), (y_1 : obj) =>
    (--- : prop)]

{move 0}

>>> postulate Betty obj

Betty : obj

{move 0}

>>> declare blueev2 that Taller Sam Betty
```

```
   blueev2 : that Sam Taller Betty

   {move 1}
end Lestrade execution
```

In this block, we introduce specific objects Sam and Betty and postulate a property Blue and a relation Taller (notice that a property is understood as a function from objects to propositions, and a (binary) relation is a function taking two objects to a proposition; also notice that Lestrade chose to use infix notation for the binary relation Taller.)

There is a basic difference between parameters (introduced by declare) and specific constants or functions introduce by postulate. We don't introduce a primitive object x by the command declare x: we provide an object parameter which can be used in introducing the predicates Blue and Taller as predicates of whatever arbitrarily chosen objects may be presented to us.

We haven't actually supposed that Sam is blue or that Sam is taller than Betty, either: we have introduced parameters which would represent evidence for these statements if there were any, not postulated an actual primitive constant which is evidence for either. The propositions A and B are also parameters: notice how they are used to introduce the notion of conjunction of arbitrary propositions.

3 Propositional logic

We begin with logic.

3.1 Conjunction

The basic rules for conjunction are: from $A\&B$ deduce A (one form of the rule of simplification); from $A\&B$ deduce B (the other form of the rule of simplification); from A and B, deduce $A\&B$ (the rule of conjunction. These rules are implemented by the Lestrade code below.

```
begin Lestrade execution

   >>> postulate Simp1 cc that A
   Simp1 : [(.A_1 : prop), (.B_1 : prop), (cc_1
       : that .A_1 & .B_1) => (--- : that
       .A_1)]

   {move 0}

   >>> postulate Simp2 cc that B

   Simp2 : [(.A_1 : prop), (.B_1 : prop), (cc_1
       : that .A_1 & .B_1) => (--- : that
       .B_1)]

   {move 0}

   >>> postulate Conj aa bb that A & B

   Conj : [(.A_1 : prop), (.B_1 : prop), (aa_1
       : that .A_1), (bb_1 : that .B_1) =>
       (--- : that .A_1 & .B_1)]

   {move 0}
end Lestrade execution
```

We claim that the text above (which requires a little reading) tells us that & actually is something like logical conjunction.

The function `Simp1` takes an argument which is evidence for$A\wedge B$ (switching over to more usual notation for "and") and returns a value which is evidence for A. This is one of the versions of the rule of simplification. `Simp2`, the other flavor of simplification, takes evidence for $A \wedge B$ as input and outputs evidence for B.

The function `Conj` takes as arguments evidence for A and evidence for B and returns evidence for $A \wedge B$: this can reasonably be taken to implement the rule of conjunction.

Our description is an oversimplification which Lestrade supports: if you look at the type information about `Conj` for example, it actually takes four arguments, A of type `prop`, B of type `prop`, `aa` of type `that` A and `bb` of type `that` B. The underlying type system of Lestrade is a system of dependent types, in which the type of an argument to a function may depend on the types of earlier arguments. The arguments A and B are needed in a formal sense, but deducible from the explicitly given arguments, and so can be left implicit.

We exhibit some evidence that we have captured at least part of the logic of “and”.

```
begin Lestrade execution

   >>> define Reverseconj cc : Conj Simp2 \
       cc Simp1 cc

   Reverseconj : [(.A_1 : prop), (.B_1
       : prop), (cc_1 : that .A_1 & .B_1) =>
       ({def} Simp2 (cc_1) Conj Simp1 (cc_1) : that
       .B_1 & .A_1)]

   Reverseconj : [(.A_1 : prop), (.B_1
       : prop), (cc_1 : that .A_1 & .B_1) =>
       (--- : that .B_1 & .A_1)]

   {move 0}
end Lestrade execution
```

We introduced objects and operations before by declaration of parameters and postulation of operations with given input and output types. Here we introduce a further way to introduce objects or (as in this case) operations, by explicit definition. Lestrade accepts the name of the defined operation, the parameters of the operation (already defined, so we know their types) and the expression for the body of the definition. Lestrade checks that the body of the definition is well typed and computes its type.

In this case, the object we have defined is the rule which allows us to deduce $B \wedge A$ from $A \wedge B$.

The rule is perfectly general in its applicability as we illustrate.

```
begin Lestrade execution

   >>> declare C prop

   C : prop

   {move 1}

   >>> declare dd that A & (B & C)

   dd : that A & B & C

   {move 1}

   >>> define test dd : Reverseconj dd

   test : [(.A_1 : prop), (.B_1 : prop), (.C_1
       : prop), (dd_1 : that .A_1 & .B_1
       & .C_1) =>
       ({def} Reverseconj (dd_1) : that
       (.B_1 & .C_1) & .A_1)]

   test : [(.A_1 : prop), (.B_1 : prop), (.C_1
       : prop), (dd_1 : that .A_1 & .B_1
       & .C_1) => (--- : that (.B_1 & .C_1) & .A_1)]

   {move 0}
end Lestrade execution
```

3.2 Implication

Having shown the validity of the rule "from $A \wedge B$ deduce $B \wedge A$" is not the same thing as proving the statement "if $A \wedge B$ then $B \wedge A$", for which our notation is $A \wedge B \rightarrow B \wedge A$, though one feels there is a relationship. In fact, our way of implementing implication will bring out the relationship.

```
begin Lestrade execution

   >>> postulate -> A B prop

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

   {move 0}

   >>> declare imp that A -> B

   imp : that A -> B

   {move 1}

   >>> declare aa3 that A

   aa3 : that A

   {move 1}

   >>> postulate Mp imp aa3 that B

   Mp : [(.A_1 : prop), (.B_1 : prop), (imp_1
       : that .A_1 -> .B_1), (aa3_1 : that
       .A_1) => (--- : that .B_1)]

   {move 0}
end Lestrade execution
```

In the block of Lestrade code above, we declare implication, an operation on two propositions which returns a proposition, and declare the possibly familiar rule of *modus ponens*, "from $A \rightarrow B$ and A, deduce B". We had to introduce a new variable `aa3` of type `that` A because we wanted the argument which is evidence for $A \rightarrow B$ to appear before the argument which is evidence for A, and Lestrade insists that explicitly given arguments appear in the order in which they are declared.

The other rule of implication requires more attention because it brings in essentially new Lestrade features.

```
begin Lestrade execution

   >>> open

      {move 2}

      >>> declare aa2 that A

      aa2 : that A

      {move 2}

      >>> postulate ded1 aa2 that B

      ded1 : [(aa2_1 : that A) => (---
          : that B)]

      {move 1}

      >>> close

   {move 1}

   >>> postulate Ded1 ded1 that A -> B

   Ded1 : [(.A_1 : prop), (.B_1 : prop), (ded1_1
       : [(aa2_2 : that .A_1) => (---
         : that .B_1)]) => (--- : that
      .A_1 -> .B_1)]

   {move 0}
end Lestrade execution
```

Ded1 implements the rule of deduction: if we can show that assuming A allows us to prove B, then we have proved $A \to B$.

In terms of the metaphysics of Lestrade, we have expressed this by postulating a function which takes implicit arguments A, B and an explicit argument which is a function taking evidence for A to evidence for B. If I suppose A I am postulating evidence for it, to which I can apply this function to get evidence for B.

The way in which the parameter ded1 is declared is a bit esoteric, our first introduction to local environments in Lestrade, of which more examples will be given soon with more comment, and a full explanation a bit later.

Briefly, ded1 is postulated in a local environment, and functions or objects postulated in a local environment become parameters when that local environment is closed, in the environment in which the local environment was originally opened. Parameters declared in a local environment disappear (pass out of scope) when the environment is closed. Objects or functions defined in a local environment persist when it is closed, but become in effect abbreviations: their definitions are always expanded when they are used, because they have a nonce quality: they will go out of scope if we further close the environment in which they are abbreviations, so we do not want their names used in anything that will persist.

There is a briefer way to make the same declaration.

```
begin Lestrade execution

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

   ded : [(aa_1 : that A) => (--- : that
       B)]

   {move 1}
   >>> postulate Ded ded that A -> B

   Ded : [(.A_1 : prop), (.B_1 : prop), (ded_1
       : [(aa_2 : that .A_1) => (--- : that
          .B_1)]) => (--- : that .A_1
       -> .B_1)]

   {move 0}
end Lestrade execution
```

Here we just declare the parameter `ded` of the type which takes an argument which is a function taking vidence for A to evidence for B and postulate `Ded`, which takes a function of that type as argument and returns evidence for $A \rightarrow B$.

Here we can state our objection to thinking that "evidence for A" is synonymous with "proof of A": when we are applying the rule of deduction, we assume that A is true, not that it is formally provable. So the object of type `that` A that we introduce as a token of our assumption should really not be thought of as necessarily a proof. But a proof of A is certainly evidence for A.

```
begin Lestrade execution

   >>> open

      {move 2}

      >>> declare cc2 that A & B

      cc2 : that A & B

      {move 2}

      >>> define andcommev cc2 : Reverseconj \
          cc2
      andcommev : [(cc2_1 : that A & B) =>
          ({def} Reverseconj (cc2_1) : that
          B & A)]

      andcommev : [(cc2_1 : that A & B) =>
          (--- : that B & A)]

      {move 1}

      >>> close

   {move 1}

   >>> define Andrev A B : Ded andcommev

   Andrev : [(A_1 : prop), (B_1 : prop) =>
       ({def} Ded ([(cc2_2 : that A_1
          & B_1) =>
          ({def} Reverseconj (cc2_2) : that
          B_1 & A_1)]) : that (A_1 & B_1) ->
       B_1 & A_1)]

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

   {move 0}

   >>> define test2 A B C : Andrev (A & B, B & C)

   test2 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) =>
       ({def} (A_1 & B_1) Andrev B_1 & C_1
       : that ((A_1 & B_1) & B_1 & C_1) ->
       (B_1 & C_1) & A_1 & B_1)]

   test2 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) => (--- : that ((A_1 & B_1) & B_1
        & C_1) -> (B_1 & C_1) & A_1 & B_1)]

   {move 0}
end Lestrade execution
```

Here we prove the theorem $A \wedge B \rightarrow B \wedge A$ which we saw through a glass darkly when we defined `Reverseconj`, and illustrate that assignment of values to its propositional parameters allow proof of specific instances of the general theorem.

The line on which `andcommev` is introduced may be confusing: what does `andcommev` do that is different from what `Reverseconj` does? The answer can be divined by looking at the type information for the two. Reverseconj has not only the explicit argument `cc2`, but also the implicit arguments A, B. In the local environment (block between an `open` statement and a matching `close` statement) A, B are treated as known constants, not parameters, so `andcommev` is defined as a function only of the parameter `cc2` defined in that local environment, which is fortunate because `andcommev` is thus of the right type to feed to `Ded` to get a proof of the theorem $A \wedge B \rightarrow B \wedge A$, as `Reverseconj`, with its hidden additional arguments, is not.

3.3 Proving a substantial logical theorem

In order to see what we are really doing (and in order to further illustrate what local environments do in Lestrade) we prove a substantial theorem of propositional logic.

```
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}
end Lestrade execution
```

We define the biconditional operation on propositions (if and only if).

The theorem we aim to prove is

$$(A \rightarrow (B \rightarrow C)) \leftrightarrow ((A \wedge B) \rightarrow C).$$

The proof of a biconditional breaks down into the proof of two conditionals.

First, prove

$$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \wedge B) \rightarrow C).$$

To do this, we assume $A \rightarrow (B \rightarrow C)$ for the sake of argument and deduce $(A \wedge B) \rightarrow C$.

```
begin Lestrade execution

   >>> clearcurrent
```

```
{move 1}

   >>> declare A prop

   A : prop

   {move 1}

   >>> declare B prop

   B : prop

   {move 1}

   >>> declare C prop

   C : prop

   {move 1}

   >>> open

      {move 2}

      >>> declare hyp1 that A -> (B -> C)

      hyp1 : that A -> B -> C

      {move 2}
end Lestrade execution
```

The command `clearcurrent` cleared the variable parameters declared earlier; we want our declarations to be close at hand so we can read them.

The process of supposing for the sake of argument that $A \rightarrow (B \rightarrow C)$ is represented by opening a local environment and introducing a local parameter which is evidence for this proposition.

Our aim that is to prove $(A \wedge B) \rightarrow C$, which we prove in the same way, opening another environment, supposing $A \wedge B$ (and adopting the new goal of proving C).

```
begin Lestrade execution

      >>> open

         {move 3}

         >>> declare hyp2 that A & B

         hyp2 : that A & B

         {move 3}
end Lestrade execution
```

Now we have the meat of the proof, where we prove C fron the hypotheses introduced so far, in a quite standard way.

```
begin Lestrade execution

         >>> define line1 hyp2 : Simp1 hyp2

         line1 : [(hyp2_1 : that A & B) =>
             ({def} Simp1 (hyp2_1) : that
             A)]

         line1 : [(hyp2_1 : that A & B) =>
             (--- : that A)]

         {move 2}

         >>> define line2 hyp2 : Mp (hyp1, line1 \
             hyp2)
         line2 : [(hyp2_1 : that A & B) =>
             ({def} hyp1 Mp line1 (hyp2_1) : that
             B -> C)]

         line2 : [(hyp2_1 : that A & B) =>
             (--- : that B -> C)]

         {move 2}

         >>> define line3 hyp2 : Simp2 hyp2

         line3 : [(hyp2_1 : that A & B) =>
             ({def} Simp2 (hyp2_1) : that
             B)]

         line3 : [(hyp2_1 : that A & B) =>
             (--- : that B)]

         {move 2}

         >>> define line4 hyp2 : Mp (line2 \
             hyp2, line3 hyp2)

         line4 : [(hyp2_1 : that A & B) =>
             ({def} line2 (hyp2_1) Mp line3
             (hyp2_1) : that C)]

         line4 : [(hyp2_1 : that A & B) =>
             (--- : that C)]

         {move 2}
end Lestrade execution
```

We do the reasoning steps to get to C. Now we close the local environment and use the function `line4` as input to `Ded` to prove a first implication goal. When the local environment is closed, its variable parameters go out of scope, but functions defined or postulated in that environment remain.

```
begin Lestrade execution

         >>> close

      {move 2}

      >>> define line5 hyp1 : Ded line4

      line5 : [(hyp1_1 : that A -> B ->
          C) =>
          ({def} Ded ([(hyp2_2 : that
             A & B) =>
             ({def} hyp1_1 Mp Simp1 (hyp2_2) Mp
             Simp2 (hyp2_2) : that C)]) : that
          (A & B) -> C)]

      line5 : [(hyp1_1 : that A -> B ->
          C) => (--- : that (A & B) ->
          C)]

      {move 1}
end Lestrade execution
```

and then we prove the first direction of our biconditional target.

```
begin Lestrade execution

      >>> close

   {move 1}

   >>> define line6 A B C : Ded line5

   line6 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) =>
       ({def} Ded ([(hyp1_2 : that A_1
          -> B_1 -> C_1) =>
          ({def} Ded ([(hyp2_3 : that
             A_1 & B_1) =>
             ({def} hyp1_2 Mp Simp1 (hyp2_3) Mp
             Simp2 (hyp2_3) : that C_1)]) : that
          (A_1 & B_1) -> C_1)]) : that
       (A_1 -> B_1 -> C_1) -> (A_1 & B_1) ->
       C_1)]

   line6 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) => (--- : that (A_1 -> B_1
       -> C_1) -> (A_1 & B_1) -> C_1)]

   {move 0}
end Lestrade execution
```

This completes the first half of the proof. Notice that the objects defined in the local environments which are closed have been expanded out so they are not mentioned.

For example `hyp1_2 Mp Simp1 (hyp2_3) Mp Simp2 (hyp2_3)` is the representation of the proof of C in the innermost local environment, with all the functions whose names suggest line numbers expanded out. Note that `Mp` is weirdly being used as an infix operator: Lestrade does this with most operations with two arguments.

The function `line5` is still accessible but would be eliminated by the `clearcurrent` command. The objects declared or defined in the innermost local environment are gone beyond recall.

We present the second half of the proof as a single block of Lestrade code.

```
begin Lestrade execution

   >>> open

      {move 2}
```

```
>>> declare hyp3 that (A & B) -> \
    C

hyp3 : that (A & B) -> C

{move 2}

>>> open

   {move 3}

   >>> declare hyp4 that A

   hyp4 : that A

   {move 3}

   >>> open

      {move 4}

      >>> declare hyp5 that B

      hyp5 : that B

      {move 4}

      >>> define line7 hyp5 : Conj \
          hyp4 hyp5

      line7 : [(hyp5_1 : that B) =>
          ({def} hyp4 Conj hyp5_1 : that
          A & B)]

      line7 : [(hyp5_1 : that B) =>
          (--- : that A & B)]

      {move 3}
      >>> define line8 hyp5 : Mp hyp3 \
          line7 hyp5

      line8 : [(hyp5_1 : that B) =>
          ({def} hyp3 Mp line7 (hyp5_1) : that
          C)]

      line8 : [(hyp5_1 : that B) =>
          (--- : that C)]

      {move 3}

      >>> close

   {move 3}

   >>> define line9 hyp4 : Ded line8

   line9 : [(hyp4_1 : that A) =>
       ({def} Ded ([(hyp5_2 : that
          B) =>
          ({def} hyp3 Mp hyp4_1 Conj
          hyp5_2 : that C)]) : that
       B -> C)]

   line9 : [(hyp4_1 : that A) =>
       (--- : that B -> C)]

   {move 2}

   >>> close

{move 2}

>>> define line10 hyp3 : Ded line9

line10 : [(hyp3_1 : that (A & B) ->
       C) =>
       ({def} Ded ([(hyp4_2 : that
          A) =>
          ({def} Ded ([(hyp5_3 : that
             B) =>
             ({def} hyp3_1 Mp hyp4_2 Conj
             hyp5_3 : that C)]) : that
          B -> C)]) : that A -> B ->
       C)]

   line10 : [(hyp3_1 : that (A & B) ->
       C) => (--- : that A -> B -> C)]

   {move 1}

   >>> close

{move 1}

>>> define line11 A B C : Ded line10

line11 : [(A_1 : prop), (B_1 : prop), (C_1
    : prop) =>
    ({def} Ded ([(hyp3_2 : that (A_1
       & B_1) -> C_1) =>
       ({def} Ded ([(hyp4_3 : that
          A_1) =>
          ({def} Ded ([(hyp5_4 : that
             B_1) =>
             ({def} hyp3_2 Mp hyp4_3 Conj
             hyp5_4 : that C_1)]) : that
          B_1 -> C_1)]) : that A_1 ->
       B_1 -> C_1)]) : that ((A_1
    & B_1) -> C_1) -> A_1 -> B_1 -> C_1)]

line11 : [(A_1 : prop), (B_1 : prop), (C_1
    : prop) => (--- : that ((A_1 & B_1) ->
    C_1) -> A_1 -> B_1 -> C_1)]
   {move 0}

   >>> define line12 A B C : Conj (line6 \
       A B C, line11 A B C)

   line12 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) =>
       ({def} line6 (A_1, B_1, C_1) Conj
       line11 (A_1, B_1, C_1) : that ((A_1
       -> B_1 -> C_1) -> (A_1 & B_1) ->
       C_1) & ((A_1 & B_1) -> C_1) ->
       A_1 -> B_1 -> C_1)]

   line12 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) => (--- : that ((A_1 ->
       B_1 -> C_1) -> (A_1 & B_1) -> C_1) & ((A_1
       & B_1) -> C_1) -> A_1 -> B_1 -> C_1)]

   {move 0}
end Lestrade execution
```

Now we have proved the biconditional theorem, if you look at the definition of the biconditional, but not in the form we would like. We can fix this.

```
begin Lestrade execution

   >>> declare p prop

   p : prop

   {move 1}
   >>> declare pp that p

   pp : that p

   {move 1}

   >>> define Fixform p pp : pp

   Fixform : [(p_1 : prop), (pp_1 : that
       p_1) =>
       ({def} pp_1 : that p_1)]

   Fixform : [(p_1 : prop), (pp_1 : that
       p_1) => (--- : that p_1)]

   {move 0}

   >>> define line13 A B C : Fixform ((A -> \
       (B -> C)) <-> (A & B) -> C, line12 \
       A B C)

   line13 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) =>
       ({def} ((A_1 -> B_1 -> C_1) <->
       (A_1 & B_1) -> C_1) Fixform line12
       (A_1, B_1, C_1) : that (A_1 ->
       B_1 -> C_1) <-> (A_1 & B_1) -> C_1)]

   line13 : [(A_1 : prop), (B_1 : prop), (C_1
       : prop) => (--- : that (A_1 -> B_1
       -> C_1) <-> (A_1 & B_1) -> C_1)]

   {move 0}
end Lestrade execution
```

`Fixform` is a bit of black magic. Lestrade can recognize that `line12` actually is a proof of the biconditional theorem, because Lestrade matching includes definition expansion. Application of `Fixform` forces the type of the output of the proof to the form given explicitly in the first argument, if Lestrade agrees that this form is actually proved by the second argument.

We present the exact same proof in structured English and math symbolism with line labels indicating the relationship to the Lestrade proof.

To be proved: $A \rightarrow (B \rightarrow C) \leftrightarrow (A \wedge B) \rightarrow C$

Part I: Assume (`hyp1`): $A \rightarrow (B \rightarrow C)$

  Goal: $(A \wedge B) \rightarrow C$

    Assume (`hyp2`): $A \wedge B$

    Goal: C

      (`line1 hyp2`) : A simplification, `hyp2`

      (`line2 hyp2`): $B \rightarrow C$ modus ponens `line1 hyp2`, `hyp1`

      (`line3 hyp2`): B simplification, `hyp2`

      (`line4 hyp2`): C modus ponens `line2 hyp2`, `line3 hyp2`

    (`line5 hyp1`): $(A \wedge B) \rightarrow C$ deduction `hyp2`–`line4`

  `line6`: $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \wedge B) \rightarrow C)$ deduction `hyp1` – `line5`

Part II: Assume (`hyp3`): $A \wedge B \rightarrow C$

  Goal: $A \rightarrow (B \rightarrow C)$

    Assume (`hyp4`): A

    Goal: $B \rightarrow C$

      Assume (`hyp5`): B

      Goal: C

        (`line7 hyp5`): $A \wedge B$ conjunction `hyp4`, `hyp5`

        (`line8 hyp5`): C modus ponens `hyp3`, `line7 hyp5`

      (`line9 hyp4`): $B \rightarrow C$ deduction `hyp5` – `line8`

    (`line10 hyp3`): $A \rightarrow (B \rightarrow C)$ deduction `hyp4`–`line9`

  (`line11`): $(A \wedge B \rightarrow C) \rightarrow (A \rightarrow (B \rightarrow C))$ deduction `hyp3`–`line10`

  The theorem: the theorem as stated above, biconditional introduction `line6`, `line11`

3.4 Negation and disjunction

In the presentation given here, we have one more basic primitive logical operation, the constant `false`, a statement known to be false, one additional primitive axiom governing this concept, and some definitions.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare A prop

   A : prop

   {move 1}

   >>> declare B prop

   B : prop

   {move 1}

   >>> postulate False prop

   False : prop

   {move 0}

   >>> define ~ A : A -> False

   ~ : [(A_1 : prop) =>
       ({def} A_1 -> False : prop)]

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

   {move 0}
   >>> define v A B : ~ A -> B

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

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

   {move 0}

   >>> 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}
end Lestrade execution
```

Here we have introduced the special statement `False` (which we might write $\perp$ in LaTeX), the notion of negation and the notion of disjunction, introduced by definition, and another axiom of propositional logic, the principle of double negation.

3.5 Equality

In this section, we present a suite of implementations of notions of equality in different types.

3.5.1 Equality of propositions as the biconditional

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare A prop

   A : prop

   {move 1}

   >>> declare B prop

   B : prop

   {move 1}

   >>> declare iffev that A <-> B

   iffev : that A <-> B

   {move 1}

   >>> declare proppred [A => prop]

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

   {move 1}

   >>> declare predev that proppred A

   predev : that proppred (A)

   {move 1}

   >>> postulate Propsub iffev, proppred, predev \
       that proppred B
   Propsub : [(.A_1 : prop), (.B_1 : prop), (iffev_1
       : that .A_1 <-> .B_1), (proppred_1
       : [(A_2 : prop) => (--- : prop)]), (predev_1
       : that proppred_1 (.A_1)) => (---
       : that proppred_1 (.B_1))]

   {move 0}

   >>> define Propsub2 iffev predev : Propsub \
       iffev, proppred, predev

   Propsub2 : [(.A_1 : prop), (.B_1
       : prop), (iffev_1 : that .A_1 <->
       .B_1), (.proppred_1 : [(A_2 : prop) =>
          (--- : prop)]), (predev_1
       : that .proppred_1 (.A_1)) =>
       ({def} Propsub (iffev_1, .proppred_1, predev_1) : that
       .proppred_1 (.B_1))]

   Propsub2 : [(.A_1 : prop), (.B_1
       : prop), (iffev_1 : that .A_1 <->
       .B_1), (.proppred_1 : [(A_2 : prop) =>
          (--- : prop)]), (predev_1
       : that .proppred_1 (.A_1)) => (---
       : that .proppred_1 (.B_1))]

   {move 0}
end Lestrade execution
```

It is in no way a consequence of adopting the Lestrade logical framework that propositions should be identified with truth values, making the biconditional the equality relation on propositions, It would be equally implementable in Lestrade to view propositions as. to go to an opposite extreme, character strings (or maybe parse trees), in which case there would be natural operations on propositions which would not respect the biconditional.

3.5.2 Equality of untyped and typed mathematical objects

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare x obj

   x : obj

   {move 1}

   >>> declare y obj

   y : obj

   {move 1}

   >>> postulate = x y prop

   = : [(x_1 : obj), (y_1 : obj) =>
       (--- : prop)]

   {move 0}

   >>> postulate Refleq x that x = x

   Refleq : [(x_1 : obj) => (--- : that
       x_1 = x_1)]

   {move 0}
end Lestrade execution
```

So far we have declared the equality relation on untyped objects and the reflexive property of equality.

Now we work on the substitution property of equality. This should remind us of the development of propositional substitutions using the biconditional above.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare x obj

   x : obj

   {move 1}

   >>> declare y obj

   y : obj

   {move 1}

   >>> declare eqev that x = y

   eqev : that x = y

   {move 1}

   >>> declare pred [x => prop]

   pred : [(x_1 : obj) => (--- : prop)]

   {move 1}

   >>> declare predev that pred x

   predev : that pred (x)
   {move 1}

   >>> postulate Subs eqev, pred, predev \
       that pred y

   Subs : [(.x_1 : obj), (.y_1 : obj), (eqev_1
       : that .x_1 = .y_1), (pred_1 : [(x_2
          : obj) => (--- : prop)]), (predev_1
       : that pred_1 (.x_1)) => (--- : that
       pred_1 (.y_1))]

   {move 0}

   >>> define Subs2 eqev predev : Subs eqev, pred, predev

   Subs2 : [(.x_1 : obj), (.y_1 : obj), (eqev_1
       : that .x_1 = .y_1), (.pred_1 : [(x_2
          : obj) => (--- : prop)]), (predev_1
       : that .pred_1 (.x_1)) =>
       ({def} Subs (eqev_1, .pred_1, predev_1) : that
       .pred_1 (.y_1))]

   Subs2 : [(.x_1 : obj), (.y_1 : obj), (eqev_1
       : that .x_1 = .y_1), (.pred_1 : [(x_2
          : obj) => (--- : prop)]), (predev_1
       : that .pred_1 (.x_1)) => (---
       : that .pred_1 (.y_1))]

   {move 0}
end Lestrade execution
```

The substitution axiom `Subs` has an alternative form `Subs2` which does not require the predicate as an explicit argument. Basically, this enables substitution of y for *all* occurrences of x: Lestrade can infer the predicate argument in this case.

The symbol `==` will be declared as equality in each type `in tau`. The implicit argument feature allows us to overload this symbol, as we will see.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare tau type

   tau : type

   {move 1}

   >>> declare x in tau

   x : in tau

   {move 1}

   >>> declare y in tau

   y : in tau

   {move 1}

   >>> postulate == x y prop

   == : [(.tau_1 : type), (x_1 : in
       .tau_1), (y_1 : in .tau_1) => (---
       : prop)]

   {move 0}

   >>> postulate Refleqt x that x == x

   Refleqt : [(.tau_1 : type), (x_1
       : in .tau_1) => (--- : that x_1 ==
       x_1)]
```

```
{move 0}

>>> declare eqev that x == y

eqev : that x == y

{move 1}

>>> declare pred [x => prop]

pred : [(x_1 : in tau) => (--- : prop)]

{move 1}

>>> declare predev that pred x

predev : that pred (x)

{move 1}

>>> postulate Subst eqev, pred, predev \
    that pred y

Subst : [(.tau_1 : type), (.x_1 : in
    .tau_1), (.y_1 : in .tau_1), (eqev_1
    : that .x_1 == .y_1), (pred_1 : [(x_2
       : in .tau_1) => (--- : prop)]), (predev_1
    : that pred_1 (.x_1)) => (--- : that
    pred_1 (.y_1))]

{move 0}

>>> define Subst2 eqev predev : Subst \
    eqev, pred, predev

Subst2 : [(.tau_1 : type), (.x_1
    : in .tau_1), (.y_1 : in .tau_1), (eqev_1
    : that .x_1 == .y_1), (.pred_1 : [(x_2
           : in .tau_1) => (--- : prop)]), (predev_1
        : that .pred_1 (.x_1)) =>
        ({def} Subst (eqev_1, .pred_1, predev_1) : that
        .pred_1 (.y_1))]

   Subst2 : [(.tau_1 : type), (.x_1
        : in .tau_1), (.y_1 : in .tau_1), (eqev_1
        : that .x_1 == .y_1), (.pred_1 : [(x_2
           : in .tau_1) => (--- : prop)]), (predev_1
        : that .pred_1 (.x_1)) => (---
        : that .pred_1 (.y_1))]

   {move 0}
end Lestrade execution
```

The relation == is actually ternary, but the first argument is the common type of the second and third, and this can be left implicit. In all other respects, these definitions are quite parallel to the definition for type `obj`, and a nice advertisement of what the implicit argument mechanism can do for us.

3.5.3 Equality of types: univalence

We implement something like the Univalence Axiom of HOTT in our entirely classical context. We think it is important to notice that there is nothing intensional or constructive about this axiom.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare tau1 type

   tau1 : type
```

```
{move 1}

>>> declare tau2 type

tau2 : type

{move 1}

>>> postulate === tau1 tau2 prop

=== : [(tau1_1 : type), (tau2_1 : type) =>
    (--- : prop)]

{move 0}

>>> declare x1 in tau1

x1 : in tau1

{move 1}

>>> declare F [x1 => in tau2]

F : [(x1_1 : in tau1) => (--- : in
    tau2)]

{move 1}

>>> declare x2 in tau2

x2 : in tau2

{move 1}

>>> declare G [x2 => in tau1]

G : [(x2_1 : in tau2) => (--- : in
    tau1)]
```

```
{move 1}

>>> postulate bijection F prop

bijection : [(.tau1_1 : type), (.tau2_1
    : type), (F_1 : [(x1_2 : in .tau1_1) =>
       (--- : in .tau2_1)]) => (---
    : prop)]

{move 0}

>>> declare bijectionev [x1 => that x1 \
       == G (F x1)]

bijectionev : [(x1_1 : in tau1) =>
    (--- : that x1_1 == G (F (x1_1)))]

{move 1}

>>> postulate Bijectionproof bijectionev \
    that bijection F

Bijectionproof : [(.tau1_1 : type), (.tau2_1
    : type), (.F_1 : [(x1_2 : in .tau1_1) =>
       (--- : in .tau2_1)]), (.G_1
    : [(x2_2 : in .tau2_1) => (---
       : in .tau1_1)]), (bijectionev_1
    : [(x1_2 : in .tau1_1) => (---
       : that x1_2 == .G_1 (.F_1 (x1_2)))]) =>
    (--- : that bijection (.F_1))]

{move 0}

>>> postulate Univalence F, G, bijectionev \
    that tau1 === tau2

Univalence : [(.tau1_1 : type), (.tau2_1
    : type), (F_1 : [(x1_2 : in .tau1_1) =>
       (--- : in .tau2_1)]), (G_1
    : [(x2_2 : in .tau2_1) => (---
       : in .tau1_1)]), (bijectionev_1
    : [(x1_2 : in .tau1_1) => (---
       : that x1_2 == G_1 (F_1 (x1_2)))]) =>
    (--- : that .tau1_1 === .tau2_1)]

{move 0}

>>> define Typerefleq tau1 : Univalence \
    [x1 => x1] [x1 => x1], [x1 => Refleqt \
       x1]

Typerefleq : [(tau1_1 : type) =>
    ({def} Univalence ([(x1_2 : in
       tau1_1) =>
       ({def} x1_2 : in tau1_1)], [(x1_2
       : in tau1_1) =>
       ({def} x1_2 : in tau1_1)], [(x1_2
       : in tau1_1) =>
       ({def} Refleqt (x1_2) : that
       x1_2 == x1_2)]) : that tau1_1
    === tau1_1)]

Typerefleq : [(tau1_1 : type) => (---
    : that tau1_1 === tau1_1)]

{move 0}

>>> declare typeqev that tau1 === tau2

typeqev : that tau1 === tau2

{move 1}

>>> declare typepred [tau1 => prop]
```

```
typepred : [(tau1_1 : type) => (---
    : prop)]

{move 1}

>>> declare typepredev that typepred tau1

typepredev : that typepred (tau1)

{move 1}

>>> postulate Typeeqsub typeqev, typepred, typepredev \
    that typepred tau2

Typeeqsub : [(.tau1_1 : type), (.tau2_1
    : type), (typeqev_1 : that .tau1_1
    === .tau2_1), (typepred_1 : [(tau1_2
       : type) => (--- : prop)]), (typepredev_1
    : that typepred_1 (.tau1_1)) =>
    (--- : that typepred_1 (.tau2_1))]

{move 0}

>>> define Typeeqsub2 typeqev typepredev \
    : Typeeqsub typeqev, typepred, typepredev

Typeeqsub2 : [(.tau1_1 : type), (.tau2_1
    : type), (typeqev_1 : that .tau1_1
    === .tau2_1), (.typepred_1 : [(tau1_2
       : type) => (--- : prop)]), (typepredev_1
    : that .typepred_1 (.tau1_1)) =>
    ({def} Typeeqsub (typeqev_1, .typepred_1, typepredev_1) : that
    .typepred_1 (.tau2_1))]

Typeeqsub2 : [(.tau1_1 : type), (.tau2_1
    : type), (typeqev_1 : that .tau1_1
    === .tau2_1), (.typepred_1 : [(tau1_2
       : type) => (--- : prop)]), (typepredev_1
       : that .typepred_1 (.tau1_1)) =>
       (--- : that .typepred_1 (.tau2_1))]

   {move 0}
end Lestrade execution
```

It should be noted that these declarations are in no way required by the Lestrade framework. The idea here is that all that matters about a type (of the form `in tau`) is its cardinality: two types which are the same size can be supposed to be exactly the same domain with different functions accessing it. This might have a paradoxical air: it implies for example that if we have a type of natural numbers and a type of rational numbers, they will be the same type. But of course every rational number can be implemented as a natural number via some enumeration of the rationals, and all operations on rationals can be defined on such numerical implementations of rationals and output numerical implementations of rationals. This is a subject which can be revisited when we have example types to look at.

Note that we didn't provide the reflexive property of equality of types as a primitive: we proved it as a consequence of univalence (the identity map is a bijection from a type to itself).

There are also practical things which we might like to do which are ruled out by univalence. It would be possible to make addition (for example) an overloaded operation working on all types. But this would then force the naturals and the rationals to be unequal types, because propositions about these types (on reasonable assumptions) would express the quite different properties of the addition operation on the two types.

3.5.4 Equality of proofs/evidence: proof irrelevance

We regard all proofs of the same proposition as equal (again, in no way a consequence of the Lestrade framework, which quite naturally implements constructive logics in which information can be extracted from proofs: for example, one might suppose that from a proof of an existentially quantified statement one could extract a witness, which might be different for different proofs).

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare A prop

   A : prop

   {move 1}

   >>> declare aa1 that A

   aa1 : that A

   {move 1}

   >>> declare aa2 that A

   aa2 : that A

   {move 1}

   >>> declare proofpred [aa1 => prop]

   proofpred : [(aa1_1 : that A) => (---
       : prop)]

   {move 1}

   >>> declare proofpredev that proofpred \
       aa1

   proofpredev : that proofpred (aa1)

   {move 1}

   >>> postulate Proofirrelevance aa2 proofpredev \
       that proofpred aa2

   Proofirrelevance : [(.A_1 : prop), (.aa1_1
       : that .A_1), (aa2_1 : that .A_1), (.proofpred_1
       : [(aa1_2 : that .A_1) => (---
          : prop)]), (proofpredev_1 : that
       .proofpred_1 (.aa1_1)) => (---
       : that .proofpred_1 (aa2_1))]

   {move 0}
end Lestrade execution
```

This code declares that any predicate that holds of one proof of a proposition A also holds of every other proof of that proposition: there is only one proof of A.

3.5.5 A remark on generalizing over types in Lestrade

This section gives an example of what happens when one wants to define operations on all object sorts of Lestrade. The object sorts come in five flavors. Sometimes it is useful to treat them differently: sometimes one ends up repeating developments for more than one kind of object sort. There is a temptation to allow object sort parameters which I have considered. In this case the five families are treated differently (though the treatment of untyped and typed mathematical objects is very similar).

4 Predicates and sets

Here we introduce sets of untyped objects (which are themselves untyped objects) and sets of typed objects, which belong to power types.

First, we introduce the notion of extensional equality of predicates.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare x obj

   x : obj

   {move 1}

   >>> declare pred [x => prop]

   pred : [(x_1 : obj) => (--- : prop)]

   {move 1}

   >>> declare predpred [pred => prop]

   predpred : [(pred_1 : [(x_2 : obj) =>
          (--- : prop)]) => (--- : prop)]

   {move 1}

   >>> declare pred2 [x => prop]

   pred2 : [(x_1 : obj) => (--- : prop)]

   {move 1}

   >>> declare predev that pred x
```

```
predev : that pred (x)

{move 1}

>>> declare predev2 that pred2 x

predev2 : that pred2 (x)

{move 1}

>>> declare extensev1 [x, predev => \
       that pred2 x]

extensev1 : [(x_1 : obj), (predev_1
    : that pred (x_1)) => (--- : that
    pred2 (x_1))]

{move 1}

>>> declare extensev2 [x, predev2 => \
       that pred x]

extensev2 : [(x_1 : obj), (predev2_1
    : that pred2 (x_1)) => (--- : that
    pred (x_1))]

{move 1}

>>> declare predpredev that predpred pred

predpredev : that predpred (pred)

{move 1}

>>> postulate Predext extensev1, extensev2, predpredev \
    that predpred pred2
```

```
   Predext : [(.pred_1 : [(x_2 : obj) =>
          (--- : prop)]), (.predpred_1
       : [(pred_2 : [(x_3 : obj) => (---
             : prop)]) => (--- : prop)]), (.pred2_1
       : [(x_2 : obj) => (--- : prop)]), (extensev1_1
       : [(x_2 : obj), (predev_2 : that
          .pred_1 (x_2)) => (--- : that
          .pred2_1 (x_2))]), (extensev2_1
       : [(x_2 : obj), (predev2_2 : that
          .pred2_1 (x_2)) => (--- : that
          .pred_1 (x_2))]), (predpredev_1
       : that .predpred_1 (.pred_1)) =>
       (--- : that .predpred_1 (.pred2_1))]

   {move 0}
end Lestrade execution
```

The intention of the function `Predext` is that it witnesses that predicates of untyped objects with the same extension are equal.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare tau type

   tau : type

   {move 1}

   >>> declare x in tau

   x : in tau

   {move 1}
```

```
>>> declare pred [x => prop]

pred : [(x_1 : in tau) => (--- : prop)]

{move 1}

>>> declare predpred [pred => prop]

predpred : [(pred_1 : [(x_2 : in tau) =>
       (--- : prop)]) => (--- : prop)]

{move 1}

>>> declare pred2 [x => prop]

pred2 : [(x_1 : in tau) => (--- : prop)]

{move 1}

>>> declare predev that pred x

predev : that pred (x)

{move 1}

>>> declare predev2 that pred2 x

predev2 : that pred2 (x)

{move 1}

>>> declare extensev1 [x, predev => \
       that pred2 x]

extensev1 : [(x_1 : in tau), (predev_1
    : that pred (x_1)) => (--- : that
    pred2 (x_1))]
```

```
   {move 1}

   >>> declare extensev2 [x, predev2 => \
          that pred x]

   extensev2 : [(x_1 : in tau), (predev2_1
       : that pred2 (x_1)) => (--- : that
       pred (x_1))]

   {move 1}

   >>> declare predpredev that predpred pred

   predpredev : that predpred (pred)

   {move 1}

   >>> postulate Predextt extensev1, extensev2, predpredev \
       that predpred pred2

   Predextt : [(.tau_1 : type), (.pred_1
       : [(x_2 : in .tau_1) => (--- : prop)]), (.predpred_1
       : [(pred_2 : [(x_3 : in .tau_1) =>
             (--- : prop)]) => (--- : prop)]), (.pred2_1
       : [(x_2 : in .tau_1) => (--- : prop)]), (extensev1_1
       : [(x_2 : in .tau_1), (predev_2
          : that .pred_1 (x_2)) => (---
          : that .pred2_1 (x_2))]), (extensev2_1
       : [(x_2 : in .tau_1), (predev2_2
          : that .pred2_1 (x_2)) => (---
          : that .pred_1 (x_2))]), (predpredev_1
       : that .predpred_1 (.pred_1)) =>
       (--- : that .predpred_1 (.pred2_1))]

   {move 0}
end Lestrade execution
```

At this point, any statement we make about a predicate of typed or untyped mathematical objects is guaranteed to be determined entirely by the extension of the predicate. We will use this principle to prove extensionality for sets.

Sets, whether of typed or untyped mathematical objects, are as it were reified predicates, objects which we postulate in correspondence with predicates.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare x obj

   x : obj

   {move 1}

   >>> declare y obj

   y : obj

   {move 1}

   >>> postulate E x y prop

   E : [(x_1 : obj), (y_1 : obj) =>
       (--- : prop)]

   {move 0}

   >>> declare A obj

   A : obj

   {move 1}
```

```
>>> declare pred [x => prop]

pred : [(x_1 : obj) => (--- : prop)]

{move 1}

>>> postulate Setof A pred obj

Setof : [(A_1 : obj), (pred_1 : [(x_2
       : obj) => (--- : prop)]) =>
    (--- : obj)]

{move 0}

>>> declare thatxina that x E A

thatxina : that x E A

{move 1}

>>> declare predev that pred x

predev : that pred (x)

{move 1}

>>> declare insetev that x E A Setof pred

insetev : that x E A Setof pred

{move 1}

>>> postulate Comp1 thatxina predev that \
    x E A Setof pred

Comp1 : [(.x_1 : obj), (.A_1 : obj), (.pred_1
    : [(x_2 : obj) => (--- : prop)]), (thatxina_1
    : that .x_1 E .A_1), (predev_1 : that
    .pred_1 (.x_1)) => (--- : that
    .x_1 E .A_1 Setof .pred_1)]

{move 0}

>>> postulate Comp2 insetev that x E A

Comp2 : [(.x_1 : obj), (.A_1 : obj), (.pred_1
    : [(x_2 : obj) => (--- : prop)]), (insetev_1
    : that .x_1 E .A_1 Setof .pred_1) =>
    (--- : that .x_1 E .A_1)]

{move 0}

>>> postulate Comp3 insetev that pred \
    x

Comp3 : [(.x_1 : obj), (.A_1 : obj), (.pred_1
    : [(x_2 : obj) => (--- : prop)]), (insetev_1
    : that .x_1 E .A_1 Setof .pred_1) =>
    (--- : that .pred_1 (.x_1))]

{move 0}

>>> postulate Ext0 thatxina that A = A Setof \
    [x => x E A]

Ext0 : [(.x_1 : obj), (.A_1 : obj), (thatxina_1
    : that .x_1 E .A_1) => (--- : that
    .A_1 = .A_1 Setof [(x_3 : obj) =>
       ({def} x_3 E .A_1 : prop)])]

{move 0}

>>> postulate Ext1 x y that (x Setof \
    [x => ~ (x = x)]) = (y Setof [x => \
       ~ (x = x)])
```

```
   Ext1 : [(x_1 : obj), (y_1 : obj) =>
       (--- : that (x_1 Setof [(x_3 : obj) =>
          ({def} ~ (x_3 = x_3) : prop)]) = y_1
       Setof [(x_3 : obj) =>
          ({def} ~ (x_3 = x_3) : prop)])]

   {move 0}
end Lestrade execution
```

These are the basic comprehension principles for sets of untyped objects, with the weak extensionality principle that objects *with elements* that have the same elements are equal. If we write `pred x` as $\phi(x)$, `x E y` as $x \in y$, and `A Setof pred` as $\{x : \phi(x)\}$ the reader may have an easier time recognizing this.

We provide that if we have $x \in A$ and $\phi(x)$, then we have $x \in \{u \in A : \phi(u)\}$ (avoiding confusion of free and bound variables by our choice of letters).

We provide that if we have $x \in \{u \in A : \phi(u)\}$, we get $x \in A$ and we get $\phi(x)$.

Finally, we provide that if we have $x \in A$ then we have $A = \{u : u \in A\}$, from which we can prove using extensionality of predicates that if A and B have the same elements, and actually have elements, then they are equal (proof to be supplied), and a further principle that all empty sets are equal.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare tau type

   tau : type

   {move 1}
```

```
>>> declare x in tau

x : in tau

{move 1}

>>> declare pred [x => prop]

pred : [(x_1 : in tau) => (--- : prop)]

{move 1}

>>> postulate Pow tau type

Pow : [(tau_1 : type) => (--- : type)]

{move 0}

>>> declare y in Pow tau

y : in Pow (tau)

{move 1}

>>> postulate Et x y prop

Et : [(.tau_1 : type), (x_1 : in
    .tau_1), (y_1 : in Pow (.tau_1)) =>
    (--- : prop)]

{move 0}

>>> postulate Setoft pred in Pow tau

Setoft : [(.tau_1 : type), (pred_1
    : [(x_2 : in .tau_1) => (--- : prop)]) =>
    (--- : in Pow (.tau_1))]
```

```
{move 0}

>>> declare predev that pred x

predev : that pred (x)

{move 1}

>>> declare insetev that x Et Setoft pred

insetev : that x Et Setoft (pred)

{move 1}

>>> postulate Compt1 predev that x Et \
    Setoft pred

Compt1 : [(.tau_1 : type), (.x_1
    : in .tau_1), (.pred_1 : [(x_2
       : in .tau_1) => (--- : prop)]), (predev_1
    : that .pred_1 (.x_1)) => (---
    : that .x_1 Et Setoft (.pred_1))]

{move 0}

>>> postulate Compt2 insetev that pred \
    x

Compt2 : [(.tau_1 : type), (.x_1
    : in .tau_1), (.pred_1 : [(x_2
       : in .tau_1) => (--- : prop)]), (insetev_1
    : that .x_1 Et Setoft (.pred_1)) =>
    (--- : that .pred_1 (.x_1))]

{move 0}

>>> declare A in Pow tau
```

```
   A : in Pow (tau)

   {move 1}

   >>> postulate Extt0 A that A == Setoft \
       [x => x Et A]

   Extt0 : [(.tau_1 : type), (A_1 : in
       Pow (.tau_1)) => (--- : that A_1
       == Setoft ([(x_3 : in .tau_1) =>
          ({def} x_3 Et A_1 : prop)]))]

   {move 0}
end Lestrade execution
```

This is the analogous set of declarations for sets of typed objects, where a set of objects of type τ belongs to a power set type $\mathcal{P}(\tau)$, and we do not allow for atoms as we did in the case of untyped objects: it seems reasonable to suppose that all elements of a power type are actually sets, where one might doubt that all elements of an undifferentiated universe of objects supporting construction of sets are actually sets.

5 Lestrade as a logical framework and the Type Inspector as a program

In reading what comes before (and in teaching it) it may be useful to look ahead at this section, which tries to give an account of the logical framework underlying our approach, but which is also menacingly dense.

Lestrade has a very specific metaphysics (as it were, a built-in philosophy).

The most general term for the things that Lestrade talks about is *entities*. The realm of entities is subdivided into the realm of *objects* and the realm of *functions*. Each of these realms is subdivided into sorts (types in the most general sense; the word "type" itself is reserved for particular special sorts.

5.1 Objects and object sorts

Propositions are objects, belonging to the sort `prop`. Lestrade is a logical framework rather than a particular logic: there are no Lestrade primitive propositions or propositional constructions: these have to be introduced by user declaration, as they are for example in the first section. Different developments are possible, and even developments which lead to different logics (constructive logic is quite naturally presented under Lestrade).

For each proposition p defined in a Lestrade theory, there is an object sort `that` p, inhabited by evidence for p. We are skeptical of the view that all inhabitants of these types are proofs, but terms of type `that` p which do not depend on arbitrary parameters may be reasonably be taken to coincide with proofs of p.

There is a sort `obj` which we describe as inhabited by "untyped" mathematical objects. This sort is provided to support the kind of set theoretical foundations which are now usual, in which all objects discussed are at bottom *sets* and there is no essential distinction between different kinds of object (though there are certainly practical distinctions). A theory whose domain of discourse is `obj` is not going to be strictly speaking an untyped theory as implemented in Lestrade, because the objects of the theory may all be of the same sort, but the types of propositions and proofs will still be used (and function types, which we have not yet touched on, will probably play a role).

There is a sort `type` which we think of as inhabited by *labels* for different types of objects in a typed mathematical theory. We view the objects of sort `type` as *labels* because the common alternative is to view them as (possibly infinite) sets of objects, and we want to preserve the insight that Lestrade formulation even of classical Cantorian mathematics of the infinite does not actually commit us to contemplation of any infinities which are not potential: at any stage, we are only considering finitely many entities.

For each object τ of the sort `type`, there is a sort `in` τ which is inhabited by objects of type τ in the usual sense. So if τ were `Nat`, a label for the type of natural numbers, `in Nat` would be the sort of natural numbers.

There is an entirely non-accidental parallelism between `prop` and the `that` sort constructor on the one hand and `type` and the `in` sort constructor on the other hand.

This is a complete account of the object sorts in Lestrade. It is expandable by expanding the specific sorts `prop` and `type`, which can be done by user declarations of constants of these types and functions returning values of these types.

5.2 Functions and function sorts

A function in Lestrade is specified by a list of parameters of specified object and function sorts followed by an object term containing no other parameters (the difference between parameters and other identifiers is relative to context and will be discussed below), along with the type of that term.

The type of a parameter in the list may contain parameters, but only ones which appear earlier in the list. We have seen this in the typing of `Conj` above, for example.

A function sort in Lestrade is specified by a list of parameters of specified object and function sorts followed by an object sort term containing no other parameters. The type of a parameter in the list may contain parameters, but only ones which appear earlier in the list.

Notation for a function is $[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow d : \tau]$, where the sorts τ_i are attributed to the atomic parameters x_i, the only parameters which can occur in a τ_i are x_j's with $j < i$ and the only parameters which can occur in d or τ are x_i's, and τ is the object sort of the object term d.

Notation for a function sort is $[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow - : \tau]$, with the same formal restrictions, `-` being a blank placeholder for a definition body d not supplied.

The sort of $[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow d : \tau]$ is $[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow - : \tau]$ if the first term is well-formed. There are atomic terms of each type

$$[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow - : \tau].$$

Object terms of our language are atomic terms with declared sorts (the context in which an atomic term is introduced will supply its sort) and function application terms. A function application term is of the form $f(t_1, \ldots, t_n)$, where f is an atomic term of a sort $[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow - : \tau]$ (yes, that is the same n).

We describe the (recursive) way to determine whether $f(t_1, \ldots, t_n)$ is well-typed. If $n = 1$, the type of $f(t_1)$ is $\tau[t_1/x]$ if t_1 is of type τ_1, and otherwise is ill typed. If $n > 1$, the type of $f(t_1, t_2, \ldots, t_n)$ is the same as the type of $g(t_2, \ldots, t_n)$, where g is an atomic term new in the context of type

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

if this term is well-typed, and otherwise is ill-typed.

Throughout the above, $T[t/x]$ is the result of replacing atomic term x with term t in T. The definition of replacement is complicated by the fact that the variables x_i are bound in the function and function sort terms: one may suppose all bound variables given fresh names before any substitutions are made, enforcing the constraint in the actual substitutions described above that no parameter x_i will occur in t_1.

There is a special note to be made about replacing the atomic f in a term $f(t_1, \ldots, t_n)$ with a term $[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow d : \tau]$: the result of this substitution will be recursively defined in much the same way that sorts of such terms are defined above: if $n = 1$, $f(t_1)[[x_1 : \tau_1 \Rightarrow d : \tau]/f]$ is $d[t_1/x_1]$; if $n > 1$, $f(t_1, t_2, \ldots, t_n)[[x_1 : \tau_1, x_2 : \tau_2 \ldots, x_n : \tau_n \Rightarrow d : \tau]/f]$ is $g(t_2, \ldots, t_n)[[x_2 : \tau_2[t_1/x_1], \ldots, x_n : \tau_n[t_1/x_1] \Rightarrow d[t_1/x_1] : \tau[t_1/x_1]]/g]$ where g is new in the context and of the same type stated above,

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

if this term is well-typed, and otherwise is ill-typed. The idea here is that we preserve the condition that all function terms in applied position are atomic by expanding out defined functions whenever a substitution is made.

There is a special provision allowing terms of the form $f(t_1, \ldots, t_m)$ where f is of type

$$[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow - : \tau]$$

and $m < n$. The intention is that such a term represents a function

$$[x_{m+1} : \tau^*_{m+1}, \ldots, x_n : \tau^*_n \Rightarrow f(t_1, \ldots, t_n, x_{m+1}, \ldots, x_n)],$$

where the types τ^*_i are deduced from the typing of $f(t_1, \ldots, t_n, x_{m+1}, \ldots, x_n)$.

5.3 Actual Lestrade input notation

A Lestrade identifier is either a string of digits, or a string of lower case characters with the first optionally capitalized, optionally followed by a string of digits, or a string of special characters. Any of these can optionally be followed by one or more single quotes.

An atomic term will be an identifier. An application term $f(t_1, \ldots, t_n)$ can take that literal shape but some variations are allowed. It can be written without the parentheses. It can be written $t_1\, f\, t_2, \ldots, t_n$ if $n > 1$ (making f a mixfix). If t_1 is a function term, this must be written $(t_1)\, f\, t_2, \ldots, t_n$ to avoid t_1 as being read as a function rather than an argument. Commas are only required after an argument if it is an atomic function term; commas are required before an argument if it is an atomic function of arity ≥ 2. A function term or function sort term $[x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow d : \tau]$ (in which d might be $-$) is written either $[x_1, \ldots, x_n => d]$ (if d is not $-$ or $[x_1, \ldots, x_n => \tau]$ (if d is $-$). This works because Lestrade requires the types of the parameters to have been explicitly stated earlier. Commas may be omitted from the argument list under the same conditions as above.

Order of operations is, mixfix operators of arity ≥ 1 are all of the same precedence and group to the right. Unary operators bind more tightly than these. Terms may be enclosed in additional parentheses freely, with the one proviso that if the first argument of a prefix function term is written in a form which begins with a parentheses, the entire argument list must also be enclosed in parentheses.

Lestrade output includes type labels never entered by the user as in the notation of the previous subsection. Lestrade always treats operators of arity 2 as infix, and always supplies as many parentheses as possible.