Lestrade Tutorial

Randall Holmes

March 30, 2024

Contents

1 Introduction

This document is intended to introduce the Lestrade logical framework and the Lestrade Type Inspector (the LTI), the software which implements the framework.

The LaTeX source of this document is also an executable LTI script. This is an exercise in literate programming, and the way it works will be explained below.

The Lestrade framework is a dependent type theory, and the LTI could be construed as software for type checking declarations and definitions in a typed programming language. What it currently mostly lacks to be seen as a full programming language is means of execution, which an earlier version did have, and which we do intend to reinstall.

It is an insight which is becoming more widely known that a type checking environment is actually a theorem prover. The way this is achieved is by viewing mathematical propositions and proofs of (or more generally evidence for) mathematical propositions as being themselves mathematical entities of special types.

The first theorem prover of this kind was Automath, developed by de Bruijn and fellow workers in the 1970's. Lestrade is a descendant of Automath (this will be clear to anyone familiar with Automath). One of the primary current theorem proving systems, Coq, is a descendant of Automath. Coq is primarily intended to implement constructive mathematics, though it can handle classical mathematics. Automath was generally used to implement theories with classical logic (though one could do constructive mathematics in Automath).

Homotopy type theory (HoTT) is a system of dependent type theory which is currently a hot topic: it can be implemented in current theorem provers of the Automath family, such as Coq. HoTT (and the native type theories of current theorem provers of the Automath family) differ from Lestrade in being quite baroque with many constructors; Lestrade has a minimal set of primitives of its own (similarly to Automath); the more complex dependent type theories can be implemented by suitable type declarations in the Lestrade framework.

We want to implement classical mathematics, and theories with the full mathematical power of Cantorian set theory (such as the default system of set theory usually used) but we are guided in our work on Lestrade by a philosophy of mathematics which is not constructive *per se* but is Aristotelean: we are exploring the idea that all infinities can be viewed as potential (but nonetheless, sense can be made of the full panoply of modern classical mathematics based on Cantorian set theory).

1.1 Opening up the LTI

To start up the LTI, start up Poly/ML in a directory in which there is a subdirectory called `Ltxts`. Once Poly/ML is open, type

```
use "lestradespecificationpoly.sml";
```

and then type

```
Interface();
```

and the LTI will be up. Text commands are typed at the LTI prompt `>>>` and the LTI responds.

A LaTeX source file (`.tex`) can be made (in part) a Lestrade executable by enclosing Lestrade commands between the lines `Begin Lestrade execution` and `End Lestrade execution`. (the execution block should itself be enclosed in a `verbatim` environment). Commands should be preceded by the prompt `>>>`. The entire file should end with the line `quit`. When the file is run in the LTI using the `Readtex` command, the replies of the LTI to the commands in execution blocks will be inserted into the file. The reader will see this in this very document.

2 Things and their kinds in Lestrade

Things that we talk about in Lestrade theories are referred to at the most general level as *entities*. There are two kinds of entity, *objects* and *functions*. Any object or function has a *sort* (the types of the Lestrade framework). There are specific sorts (correlated with objects of a special sort) which we call *types*, as we will see. Types as such are sorts of mathematical objects of the usual kind as opposed to the sort of propositions and the sorts of their proofs [and as opposed to the sort of type labels], as we will see.

We begin with objects and their sorts.

2.1 Objects and their sorts

There are three singular sorts of object, `prop` (the sort inhabited by propositions), `obj` (the catchall sort inhabited by objects of an "untyped" theory like ZFC), and `type` (the sort inhabited by labels for the types of objects in a typed mathematical theory).

For a reader unfamiliar with distinctions between untyped and typed mathematical theories, we will present declarations of Lestrade theories implementing mathematical theories of both kinds below.

For each p of type `prop`, there is a sort `that` p inhabited by what might be called proofs of p, but we prefer to call objects of sort `that` p *evidence* for p. If you are given an object of sort `that` p you are committed to p being true: there is nothing probabilistic about this evidence.

We prefer to say "evidence for p" rather than "proofs of p", because for us adopting the hypothesis that p is true is the same as postulating an object of type `that` p, and it is on the face of it stronger to suppose that there is a proof of p than it is to suppose that p is true. A constructivist might feel that proofs are the only sort of evidence, but the framework here does not commit us to such a view. An explicitly constructed object of type `that` p, where neither the evidence object nor the proposition contains any variable parameters, we would call a proof.

For each object τ of sort `type` there is a sort `in` τ inhabited by objects of the sort labelled by τ. The object τ we call a "type label" and the sort `in` τ we might refer to as a type.

An example (to make it clear what we are talking about): a type label `Nat` might be present in a typed theory as the label for the type of natural numbers; objects $0, 1, 2, \ldots$ would then presumably be present, each of sort `in Nat`. We resist that idea that `Nat` is the set of natural numbers in this example: the sort `in Nat` is a feature we notice in each natural number when we encounter it: we do not need to know about all natural numbers at once to know what `Nat` is [this is suggested by our Aristotelean viewpoint].

It is useful to notice that there is no formal difference at all between `prop/that` on the one hand and `type/in` on the other. There is a well known analogy between constructions of proofs of propositions on the one hand and constructions of objects in mathematical types on the other, which we will explore below. The difference between propositions (associated with the sort of their proofs) and type labels (associated with the type they represent) is a difference of intention more than of basic mathematical form. Some approaches might not even make use of this distinction (all work might be done in `type/in` with propositions just a special case). Some approaches might declare general operations on propositions and their proofs which have no analogues on type labels and their associated types, or vice versa.

These are all the sorts of object in the current implementation. There is a plan to introduce an additional sort of object representing computational rules, which we might discuss later: the aim is to support interesting execution behavior and make the LTI into a typed programming language, in effect. For the moment, these are not implemented, though an approximation to them is implemented in an earlier version of the LTI.

2.2 Declarations of parameters, constants and functions

Proceeding directly to a description of the other major kind of entity, functions, would involve digesting a complicated abstraction all at once. We approach this more gently by introducing some declarations and definitions of objects and functions, and introducing the important topic of *parameters* or *variables*.

We introduce the definition of the logical notion of *conjunction* (and) and its basic rules. It is important to notice that conjunction is not a primitive of Lestrade: Lestrade in fact has no primitive operations on propositions at all! These are declarations in a specific Lestrade theory (and so universally useful that this is likely to be in the background library of almost all theories a user will construct).

```
begin Lestrade execution

   >>> declare p prop

   p : prop

   {move 1}

   >>> declare q prop

   q : prop

   {move 1}
end Lestrade execution
```

The only lines that I typed in the block above were the ones beginning with the prompt `>>>`. The rest of the text is replies from the LTI.

The commands above declare two variable parameters of type `prop`.

```
begin Lestrade execution

   >>> postulate False prop

   False : prop

   {move 0}

   >>> postulate & p q prop

   & : [(p_1 : prop), (q_1 : prop) =>
       (--- : prop)]

   {move 0}
end Lestrade execution
```

The first command above declares a proposition constant `False` (demonstrating the difference between declaring a constant and declaring a variable parameter). The annotation `move 0` tells us that this is a constant declaration, as opposed to the annotation `move 1` below the parameter declarations in the block above.

The second declaration requires more attention. The symbol `&` is declared as a function taking two proposition parameters and returning a proposition. The output notation below gives rather verbose notation for the sort of this operation. Please note that we will feel free to use the now more usual notation $\wedge$ for conjunction; in Lestrade we must accommodate the limits of the typewriter keyboard.

This does not tell Lestrade that `&` has the intended meaning "and": all it says is that this is a binary propositional connective.

And now we will exhibit further declarations which will give `&` the intended meaning.

```
begin Lestrade execution

   >>> declare pp that p

   pp : that p

   {move 1}

   >>> declare qq that q

   qq : that q

   {move 1}

   >>> postulate Conj pp qq that p & q

   Conj : [(.p_1 : prop), (.q_1 : prop), (pp_1
       : that .p_1), (qq_1 : that .q_1) =>
       (--- : that .p_1 & .q_1)]

   {move 0}
end Lestrade execution
```

Since we are doing something really interesting here. . .several things happen at once which need discussion.

We first declare variables *pp* and *qq* of types `that` p and `that` q respectively. *pp* is a parameter varying over evidence for p, and *qq* is a parameter varying over evidence for q.

The declaration we write for `Conj` says that it takes evidence for p and evidence for q and returns evidence for $p \wedge q$ (notice that the prover will use infix notation for a function with two arguments). And this embodies the logical rule of conjunction introduction (which I call simply conjunction).

The type declaration which the LTI returns reveals that `Conj` is actually a function of four arguments: the arguments p and q can safely be left implicit because they can be deduced from the types of the explicit arguments *pp* and *qq*. The implicit argument facility is a very subtle aspect of the LTI.

We complete the declaration of the rules of deduction for conjunction. The rule(s) of conjunction elimination we are in the habit of calling "simplification".

```
begin Lestrade execution

   >>> declare rr that p & q

   rr : that p & q

   {move 1}

   >>> postulate Simp1 rr that p

   Simp1 : [(.p_1 : prop), (.q_1 : prop), (rr_1
       : that .p_1 & .q_1) => (--- : that
       .p_1)]

   {move 0}

   >>> postulate Simp2 rr that q

   Simp2 : [(.p_1 : prop), (.q_1 : prop), (rr_1
       : that .p_1 & .q_1) => (--- : that
       .q_1)]

   {move 0}
end Lestrade execution
```

From evidence for $p \wedge q$ we can extract evidence for p and evidence for q. Notice that again the propositions p and q are hidden arguments of the functions `Simp1` and `Simp2`.

We illustrate development of a derived rule of inference.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare p prop

   p : prop

   {move 1}

   >>> declare q prop

   q : prop

   {move 1}

   >>> declare rr that p & q

   rr : that p & q

   {move 1}

   >>> open

      {move 2}

      >>> define rr1 : Simp2 rr

      rr1 : that q

      {move 1}

      >>> define rr2 : Simp1 rr
```

```
      rr2 : that p

      {move 1}

      >>> define rr3 : Conj rr1 rr2

      rr3 : that q & p

      {move 1}

      >>> close

   {move 1}

   >>> define Conjsymm rr : rr3

   Conjsymm : [(.p_1 : prop), (.q_1
       : prop), (rr_1 : that .p_1 & .q_1) =>
       ({def} Simp2 (rr_1) Conj Simp1 (rr_1) : that
       .q_1 & .p_1)]

   Conjsymm : [(.p_1 : prop), (.q_1
       : prop), (rr_1 : that .p_1 & .q_1) =>
       (--- : that .q_1 & .p_1)]

   {move 0}
end Lestrade execution
```

The `clearcurrent` command clears declarations of parameters, so p and q are declared as parameters again.

The effect of the block beginning with `open` and ending with `close` will be discussed further. In this context, the effect is that the parameter `rr` is treated as a constant, not a parameter in that block, so the definitions of `rr1`, `rr2`, `rr3` are in effect definitions of constants locally and do not require `rr` as a parameter. Once the block is closed, `rr3` is seen as an expression depending on the variable parameter `rr` in a way which makes the definition work. Notice that `Conjsymm` is actually a function of three variables: p, q, rr, with the first two deducible from the last and so safely left implicit.

2.3 Variable management and "moves"

The reader may have noticed the annotation of each declaration with a *move*.

Move 0 objects are the objects actually declared in a theory (its constants). In our current theory, the move 0 objects are (thus far) the primitives `&`, `False`, `Conj`, `Simp1`, `Simp2`, and the defined `Conjsymm`.

Move 1 objects include parameters used for definitions of move 0 functions, and also variable expressions at this level, such as `rr1, rr2, rr3` above.

These may be the only moves. But if we use the `open` command we can introduce new moves. In any given context, the highest indexed move is the "next move", currently inhabited by parameters; the next-to-highest indexed move is called the "last move", and entities declared in the last and all lower indexed moves are treated as constants for the moment. The `open` command increments the index of the next move; the `close` command decrements the index of the next move and discards everything defined or declared in the prior next move, which may force expansions of definitions (we will see this happen). The `clearcurrent` command discards every declaration in the next move without decrementing; this is particularly useful when the next move is move 1, since we cannot decrement at that point. There are some variations involving saved hierarchies of declarations which may be discussed later.

The `declare` command is designed to introduce parameters at the next move (so at move 1 in the default state of the LTI). The `postulate` command and the `define` command introduce objects or functions at the last move (by declaration or definition respectively), and take parameters from the next move. This can be seen in examples above.

All of this may seem mysterious but it is actually easy to exhibit declarations based on things we see in high school and undergraduate mathematics which exhibit the gradations of variability which the move facility supports.

```
begin Lestrade execution

   >>> postulate Real type

   Real : type
```

```
{move 0}

>>> declare a in Real

a : in Real

{move 1}

>>> declare b in Real

b : in Real

{move 1}

>>> postulate + a b in Real

+ : [(a_1 : in Real), (b_1 : in Real) =>
    (--- : in Real)]

{move 0}

>>> postulate * a b in Real

* : [(a_1 : in Real), (b_1 : in Real) =>
    (--- : in Real)]

{move 0}

>>> open

   {move 2}

   >>> declare x in Real

   x : in Real

   {move 2}
```

```
   >>> declare y in Real

   y : in Real

   {move 2}

   >>> define testfun x y : (a * x) + (b * y)

   testfun : [(x_1 : in Real), (y_1
       : in Real) => (--- : in Real)]

   {move 1}

   >>> close

{move 1}

>>> Showdec testfun

testfun : [(x_1 : in Real), (y_1
    : in Real) =>
    ({def} (a * x_1) + b * y_1 : in
    Real)]

testfun : [(x_1 : in Real), (y_1
    : in Real) => (--- : in Real)]

{move 1}

>>> postulate 1 in Real

1 : in Real

{move 0}

>>> postulate 2 in Real

2 : in Real
```

```
   {move 0}

   >>> define test2 a b : testfun 1 2

   test2 : [(a_1 : in Real), (b_1 : in
       Real) =>
       ({def} (a_1 * 1) + b_1 * 2 : in
       Real)]

   test2 : [(a_1 : in Real), (b_1 : in
       Real) => (--- : in Real)]

   {move 0}
end Lestrade execution
```

The declarations above introduce the declaration of a general linear function of two variables $ax+by$ in a quite ordinary sense. The difference between x and y, parameters of this function, and a, b ("unknown constants", but just as much variables) can be expressed exactly in Lestrade terms: a, b are move 1 variables and x, y are move 2 variables.

`testfun`$(x, y) = ax + by$ is then a function at move 1 (because it depends on the move 1 parameters a, b being treated as constants). The function `test2` we define at move 0 at the end is really a bit of fun.

2.4 Implication and quantifiers in Lestrade

Before moving to the description of functions in full abstraction, we introduce implication in Lestrade, which requires more sophisticated work with functions, and quantifiers in Lestrade, which show the need for "dependent types" (we will explain in context what this means).

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare p prop

   p : prop

   {move 1}

   >>> declare q prop

   q : prop

   {move 1}

   >>> postulate -> p q prop

   -> : [(p_1 : prop), (q_1 : prop) =>
       (--- : prop)]

   {move 0}

   >>> declare rr that p -> q

   rr : that p -> q

   {move 1}

   >>> declare pp that p
```

```
   pp : that p

   {move 1}

   >>> postulate Mp rr pp that q

   Mp : [(.p_1 : prop), (.q_1 : prop), (rr_1
       : that .p_1 -> .q_1), (pp_1 : that
       .p_1) => (--- : that .q_1)]

   {move 0}
end Lestrade execution
```

In the block of text above we declare the operation of implication on propositions, and the familiar rule of *modus ponens* for use of conditional statements which are given. The rule for proving a conditional statement, usually called the deduction theorem, is of a more complicated nature.

```
begin Lestrade execution

   >>> open

      {move 2}

      >>> declare pp2 that p

      pp2 : that p

      {move 2}

      >>> postulate qq2 pp2 that q

      qq2 : [(pp2_1 : that p) => (---
          : that q)]
```

```
   {move 1}

   >>> close

{move 1}

>>> postulate Deduction qq2 that p -> \
    q

Deduction : [(.p_1 : prop), (.q_1
    : prop), (qq2_1 : [(pp2_2 : that
       .p_1) => (--- : that .q_1)]) =>
    (--- : that .p_1 -> .q_1)]

{move 0}
end Lestrade execution
```

A first thing to notice is that the `postulate` command introducing `qq2`, since it is executed at move 2, declares a function from evidence for p to evidence for q at move 1, which is then usable as a parameter once the next move is again move 1.

This subtlety once grasped, what `Deduction` does is take p, q as implicit arguments, and a function sending evidence for p to evidence for q as an explicit argument, and return evidence for $p \to q$.

We exhibit a couple of examples.

```
begin Lestrade execution

   >>> open

      {move 2}

      >>> declare pp1 that p

      pp1 : that p

      {move 2}

      >>> define selfimp pp1 : pp1

      selfimp : [(pp1_1 : that p) => (---
          : that p)]

      {move 1}

      >>> close

   {move 1}

   >>> define Selfimp p : Deduction selfimp

   Selfimp : [(p_1 : prop) =>
       ({def} Deduction ([(pp1_2 : that
          p_1) =>
          ({def} pp1_2 : that p_1)]) : that
       p_1 -> p_1)]

   Selfimp : [(p_1 : prop) => (--- : that
       p_1 -> p_1)]

   {move 0}
```

```
   >>> open

      {move 2}

      >>> declare rr1 that p & q

      rr1 : that p & q

      {move 2}

      >>> define conjsymm rr1 : Conjsymm \
          rr1

      conjsymm : [(rr1_1 : that p & q) =>
          (--- : that q & p)]

      {move 1}

      >>> close

   {move 1}

   >>> define Conjsymm2 p q : Deduction conjsymm

   Conjsymm2 : [(p_1 : prop), (q_1 : prop) =>
       ({def} Deduction ([(rr1_2 : that
          p_1 & q_1) =>
          ({def} Conjsymm (rr1_2) : that
          q_1 & p_1)]) : that (p_1 & q_1) ->
       q_1 & p_1)]

   Conjsymm2 : [(p_1 : prop), (q_1 : prop) =>
       (--- : that (p_1 & q_1) -> q_1 & p_1)]

   {move 0}
end Lestrade execution
```

It is worth noticing in the second example, which derives from the rule of inference given above the closely related tautology, that `conjsymm` and `Conjsymm` in fact have quite different types, because the second concept has implicit arguments which the first does not.

Now we introduce the universal quantifier and its rules.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> open

      {move 2}

      >>> declare x1 obj

      x1 : obj

      {move 2}

      >>> postulate pred x1 prop

      pred : [(x1_1 : obj) => (--- : prop)]

      {move 1}

      >>> close

   {move 1}

   >>> postulate Forall pred prop

   Forall : [(pred_1 : [(x1_2 : obj) =>
          (--- : prop)]) => (--- : prop)]

   {move 0}
end Lestrade execution
```

Above we declare a variable `pred` which represents a predicate (a function from the type obj to the type of propositions), then declare Forall as a function mapping predicates to propositions.

```
begin Lestrade execution

   >>> declare x obj

   x : obj

   {move 1}

   >>> declare univev that Forall pred

   univev : that Forall (pred)

   {move 1}

   >>> postulate Ui univev x that pred x

   Ui : [(.pred_1 : [(x1_2 : obj) =>
          (--- : prop)]), (x_1 : obj), (univev_1
       : that Forall (.pred_1)) => (---
       : that .pred_1 (x_1))]

   {move 0}
end Lestrade execution
```

This is the rule of universal instantiation. Given evidence univev for $(\forall u : \texttt{pred}(u))$ and a particular x of type obj, the function Ui returns evidence for $\texttt{pred}(x)$. Notice that not only does the output depend on the values of the inputs, but the *sort* of the output depends on some of the inputs: this is what makes this a dependent type theory.

The rule of universal generalization is perhaps more subtle.

```
begin Lestrade execution

   >>> open

      {move 2}

      >>> declare x1 obj

      x1 : obj

      {move 2}

      >>> postulate univev2 x1 that pred \
          x1

      univev2 : [(x1_1 : obj) => (---
          : that pred (x1_1))]

      {move 1}

      >>> close

   {move 1}

   >>> postulate Ug pred, univev2 that Forall \
       pred

   Ug : [(pred_1 : [(x1_2 : obj) =>
          (--- : prop)]), (univev2_1
       : [(x1_2 : obj) => (--- : that
          pred_1 (x1_2))]) => (--- : that
       Forall (pred_1))]

   {move 0}
```

```
   >>> define Ug2 univev2 : Ug pred, univev2

   Ug2 : [(.pred_1 : [(x1_2 : obj) =>
          (--- : prop)]), (univev2_1
       : [(x1_2 : obj) => (--- : that
          .pred_1 (x1_2))]) =>
       ({def} Ug (.pred_1, univev2_1) : that
       Forall (.pred_1))]

   Ug2 : [(.pred_1 : [(x1_2 : obj) =>
          (--- : prop)]), (univev2_1
       : [(x1_2 : obj) => (--- : that
          .pred_1 (x1_2))]) => (---
       : that Forall (.pred_1))]

   {move 0}
end Lestrade execution
```

The rule of universal generalization tells us that if we have a procedure to generate from any x of type `obj` evidence for $\texttt{pred}(x)$, then we have evidence for $(\forall x : \texttt{pred}(x))$.

The reason that a second version is given is that, as we will see, it is sometimes but not always possible for the implicit argument mechanism to deduce from `univev2` what the predicate `pred` is. It can be very convenient not to have to write out the predicate.

Here we have developed the universal quantifier for untyped mathematical objects (type `obj`). We are likely later to present the declarations introducing quantification over types (`in` τ).

A difference between Lestrade and other systems of this family is that in many of them an item of evidence for $p \to q$ *is* a function from evidence for p to evidence for q, and an item of evidence for $(\forall x : \texttt{pred}(x))$ *is* a function taking each appropriate x to evidence for $\texttt{pred}(x)$. We find it useful to separate objects from functions for reasons which we hope will be seen as we carry out developments. For those in the know (and for everyone once we look at mathematical functions) it will be seen that the same thing happens on the other side of the Curry-Howard isomorphism: evidence for an implication is an object in the Lestrade sense produced by applying a constructor

(`Deduction`) to a function in the Lestrade sense; mathematical objects in type theory which a mathematician (including this author) would describe as a function will actually be Lestrade objects which package Lestrade functions in the same way.

2.5 Functions and their sorts: a technical interlude

Here we formally describe functions and their sorts, completing the description of the entities of the Lestrade framework, mod user declarations.

An object sort, as noted above, is either `prop`, `type`, `obj`, or `that` p where p is a term of sort `prop` or `in` τ where τ is a term of sort `type`.

A function sort is of the shape $(x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow \tau)$, where each x_i is a variable of type τ_i bound in the expression and sorts τ and τ_j for $j > i$ may contain occurrences of the variable x_i (this is what makes this dependent type theory). The types τ_i may be either object or function sorts; τ is an object sort. n might be 1 in which case the shape is actually $(x_1 : \tau_1 \Rightarrow \tau)$

An object term is either atomic or an application term $f(t_1, \ldots, t_n)$, where f must be an atomic function term and the t_i's are general object or function terms. We note as seen above that binary function terms may be used as infixes; Lestrade output notation will use infix notation by preference.

A function term is either atomic or of the shape $(x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow t : \tau)$ (which is of sort $(x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow \tau)$), each x_i being of sort τ_i and t of sort τ). As above, n might be 1.

A term $f(t_1, \ldots, t_n)$ where f is of type $(x_1 : \tau_1, \ldots, x_m : \tau_m \Rightarrow \tau)$ is well-typed iff $m = n$ and (if $n = 1$) t_1 is of type τ_1 [in which case the term is of type $\tau[t_1/x_1]$] or (if $n > 1$) $g(t_2, \ldots, t_n)$ is well-typed where g is 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 the type of the original term is the type of this term]. The notation $X[t/v]$ denotes the result of substituting the term t for the variable v in the term or sort X.

The notion of substitution has an elaborate recursive definition, the interesting clause of which is that replacement of f with

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

in $f(t_1, \ldots, t_n)$ yields (if $n = 1$) $t[t_1/x_1]$ and otherwise the result of replacing g with $(x_2 : \tau_2[t_1/x_1], \ldots, x_n : \tau_n[t_1/x_1] \Rightarrow t : \tau[t_1/x_1])$ in $g(t_2, \ldots, t_n)$: we do not write function abstraction terms in applied position, but always carry through with the implied substitution (a feature of Russell and Whitehead's *Principia*.)

The output notation of Lestrade takes roughly the form described above (the attentive reader has seen samples). A function sort actually takes the shape $(x_1 : \tau_1, \ldots, x_n : \tau_n \Rightarrow$ `---` $: \tau)$, its computer representation being similar to that of an abstract function term, but with a dummy object as the body. The input notation takes advantage of the fact that atomic terms always have types declared earlier in the environment, and does not require [or even support] user entered types attached to variables or other terms.

It *is* possible to write function sorts and function abstraction terms in the form $[t_1, \ldots, t_n \Rightarrow \tau]$ or $[t_1, \ldots, t_n \Rightarrow t]$ as user input, where the types of the bound variables are read from the types of variables of the same shape in the next move [though they do *not* have the same semantics, being actually variables in further moves not explicitly introduced, if one looks closely]. Examples will be given. There will also be discussion of the forms that user input notation may take.

The reader should note that application terms are officially always of object type. If $f(t_1, \ldots, t_n)$ is well formed, the LTI does permit use of $f(t_1, \ldots, t_m)$ $[m < n]$ as shorthand for $[(x_{m+1}, \ldots, x_n \Rightarrow f(t_1, \ldots, t_m, x_{m+1}, \ldots, x_n)]$.

This is a fairly complete description of the Lestrade type framework. The rest is user declaration of objects of useful types.

Thus far we have considered mathematicial technicalities about the notion of function. We also have to consider metaphysical technicalities.

Our guiding philosophy in Lestrade is Aristotelean: we do not want to posit actual infinite objects given as completed wholes. So we do not view functions as complete infinite tables of values but as rules which tell us, for any input we encounter, how to compute the output. We can extend this to such things as universally quantified statements: evidence for $(\forall x : \mathtt{pred}(x)$ is not construed as involving evidence for each and every $\mathtt{pred}(a)$, but a recipe for evidence for $\mathtt{pred}(a)$ for any a that we may encounter. We may think of objects in the next move as arbitrary objects that we may encounter in the future. The machinery of Lestrade is designed to convince us that we really can manage with a notion of "arbitrary object" without having to suppose that we are given all objects at once. Impredicativity makes its appearance in the fact that we are able to consider arbitrary functions which may be introduced in the future, as well – without claiming that we know all possible functions at once. Of course, hypotheses about all functions that we may encounter will in general be a lot stronger than hypotheses about all objects we may encounter: but we can formulate them.

And, curiously enough, classical logic and Cantorian set theory make perfectly good sense in this framework, as we will see.

3 Definitions and proofs: logic

In this section we will build machinery for logic in Lestrade and prove some theorems, with both the usefulness of such a Lestrade theory in mind and the implicit illustration of how Lestrade works.

The theory of dependent type theorem proving might be taken to suggest a preference for constructive logic. We are actually fine with classical logic, but we are concerned to illustrate that it is compatible with our Aristotelean philosophical framework.

So, we will give independent constructive definitions for the connectives of propositional logic, but also provide the principle of double negation, so that classical equivalences obtain.

3.1 More propositional logic declarations

We introduce disjunction as a primitive.

```
begin Lestrade execution

   >>> declare p prop

   p : prop

   {move 1}

   >>> declare q prop

   q : prop

   {move 1}

   >>> postulate V p q prop

   V : [(p_1 : prop), (q_1 : prop) =>
       (--- : prop)]

   {move 0}
```

```
   >>> declare pp that p

   pp : that p

   {move 1}

   >>> declare qq that q

   qq : that q

   {move 1}

   >>> postulate Addition1 q pp that p V q

   Addition1 : [(.p_1 : prop), (q_1
       : prop), (pp_1 : that .p_1) =>
       (--- : that .p_1 V q_1)]

   {move 0}

   >>> postulate Addition2 p qq that p V q

   Addition2 : [(p_1 : prop), (.q_1
       : prop), (qq_1 : that .q_1) =>
       (--- : that p_1 V .q_1)]

   {move 0}
end Lestrade execution
```

It is interesting to note that the rules of addition need the other proposition as an argument. It is hardly a surprise, as the other proposition does not appear as a component of what you might think is the default input, but it is there.

The rule of disjunction elimination (proof by cases) is more complicated.

```
begin Lestrade execution

   >>> declare r prop

   r : prop

   {move 1}

   >>> open

      {move 2}

      >>> declare pp1 that p

      pp1 : that p

      {move 2}

      >>> postulate case1 pp1 that r

      case1 : [(pp1_1 : that p) => (---
          : that r)]

      {move 1}

      >>> declare qq1 that q

      qq1 : that q

      {move 2}

      >>> postulate case2 qq1 that r

      case2 : [(qq1_1 : that q) => (---
          : that r)]

      {move 1}
```

```
   >>> close

{move 1}

>>> postulate Cases case1, case2 that \
    p V q

Cases : [(.p_1 : prop), (.q_1 : prop), (.r_1
    : prop), (case1_1 : [(pp1_2 : that
       .p_1) => (--- : that .r_1)]), (case2_1
    : [(qq1_2 : that .q_1) => (---
       : that .r_1)]) => (--- : that
    .p_1 V .q_1)]

{move 0}
end Lestrade execution
```

A nice feature here is that we do not need a separate environment for each case: one environment with two hypothetical arguments works, because we can keep track in our declarations of which hypthoses are used.

Arguments to a function which are themselves functions should be separated by commas from what follows to avoid a misreading in which the function argument is *applied* to what follows. `case1` in the definition of `Cases` is an example.

The negation and the biconditional are introduced by definition.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare p prop

   p : prop

   {move 1}
```

```
>>> declare q prop

q : prop

{move 1}

>>> define ~ p : p -> False

~ : [(p_1 : prop) =>
    ({def} p_1 -> False : prop)]

~ : [(p_1 : prop) => (--- : prop)]

{move 0}

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

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

<-> : [(p_1 : prop), (q_1 : prop) =>
    (--- : prop)]

{move 0}

>>> declare maybe that ~ ~ p

maybe : that ~ (~ (p))

{move 1}

>>> postulate Dneg maybe that p

Dneg : [(.p_1 : prop), (maybe_1 : that
    ~ (~ (.p_1))) => (--- : that
```

```
       .p_1)]

   {move 0}
end Lestrade execution
```

We derive rules for the use of these defined operators. We start with negation introduction.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare p prop

   p : prop

   {move 1}

   >>> open

      {move 2}

      >>> declare pp1 that p

      pp1 : that p

      {move 2}

      >>> postulate contra1 pp1 that False

      contra1 : [(pp1_1 : that p) => (---
          : that False)]

      {move 1}
```

```
   >>> close

{move 1}

>>> define Negintro1 contra1 : Deduction \
    contra1

Negintro1 : [(.p_1 : prop), (contra1_1
    : [(pp1_2 : that .p_1) => (---
       : that False)]) =>
    ({def} Deduction (contra1_1) : that
    .p_1 -> False)]

Negintro1 : [(.p_1 : prop), (contra1_1
    : [(pp1_2 : that .p_1) => (---
       : that False)]) => (--- : that
    .p_1 -> False)]

{move 0}
end Lestrade execution
```

We would like the rule of negation introduction to prove a theorem with the negation symbol in it! There is a general technique to handle this.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> 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}
end Lestrade execution
```

The subtlety here is that `Fixform` will check by matching, which includes expansion of definitions, that its second argument `pp` is of a type equivalent to `that` p, and will return the literal type `that` p as its type.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare p prop

   p : prop

   {move 1}

   >>> declare pp that p

   pp : that p
```

```
   {move 1}

   >>> declare contra1 [pp => that False]

   contra1 : [(pp_1 : that p) => (---
       : that False)]

   {move 1}

   >>> define Negintro contra1 : Fixform \
       (~ p, Negintro1 contra1)

   Negintro : [(.p_1 : prop), (contra1_1
       : [(pp_2 : that .p_1) => (--- : that
          False)]) =>
       ({def} ~ (.p_1) Fixform Negintro1
       (contra1_1) : that ~ (.p_1))]

   Negintro : [(.p_1 : prop), (contra1_1
       : [(pp_2 : that .p_1) => (--- : that
          False)]) => (--- : that ~ (.p_1))]

   {move 0}
end Lestrade execution
```

There is an important feature of Lestrade which is introduced briefly here to save time. One can declare `contra1` without opening a block as shown above. It is possible to avoid ever entering abstraction terms like `[pp => that False]` by suitable use of blocks, but it is very convenient to do so. Notice that the LTI uses a declaration in the next move to determine the type of the bound occurrence of `pp`, though (as you can see in the previous proof using a block) if we did this using a block this would not be a variable in the next move but in a further move which we do not admit opening here.

And as noted we use `Fixform` to coerce the type of the output of `Negintro` to the form desired.

We develop the standard strategy for proving a biconditional.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare p prop

   p : prop

   {move 1}

   >>> declare q prop

   q : prop

   {move 1}

   >>> declare pp that p

   pp : that p

   {move 1}

   >>> declare qq that q

   qq : that q

   {move 1}

   >>> declare part1 [pp => that q]

   part1 : [(pp_1 : that p) => (--- : that
       q)]

   {move 1}

   >>> declare part2 [qq => that p]
```

```
   part2 : [(qq_1 : that q) => (--- : that
       p)]

   {move 1}

   >>> define Bicondintro part1, part2 : Fixform \
       (p <-> q, Conj (Deduction part1, Deduction \
       part2))

   Bicondintro : [(.p_1 : prop), (.q_1
       : prop), (part1_1 : [(pp_2 : that
          .p_1) => (--- : that .q_1)]), (part2_1
       : [(qq_2 : that .q_1) => (--- : that
          .p_1)]) =>
       ({def} (.p_1 <-> .q_1) Fixform Deduction
       (part1_1) Conj Deduction (part2_1) : that
       .p_1 <-> .q_1)]

   Bicondintro : [(.p_1 : prop), (.q_1
       : prop), (part1_1 : [(pp_2 : that
          .p_1) => (--- : that .q_1)]), (part2_1
       : [(qq_2 : that .q_1) => (--- : that
          .p_1)]) => (--- : that .p_1
       <-> .q_1)]

   {move 0}
end Lestrade execution
```

This is work in a much more succinct style, taking advantage of the ability to write variable binding expressions for function types and declare variables of such types without opening a block, and then (in the proof of `Bicondintro`) taking an algebraic approach of just writing down the proof as a mathematical expression.

It is worth noting in passing that Lestrade's default view of display of most functions of two explicit arguments is to use them as infixes; for example, `Conj` is used as an infix in Lestrade output above.

3.2 Proving the exportation theorem

In this section I present a proof in something like my favored natural deduction style for paper proofs of the theorem $P \rightarrow (Q \rightarrow R) \leftrightarrow (P \wedge Q) \rightarrow R$. This should give an idea of a strategy for proof development in Lestrade.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare p prop

   p : prop

   {move 1}

   >>> declare q prop

   q : prop

   {move 1}

   >>> declare r prop

   r : prop

   {move 1}

   >>> open

      {move 2}

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

      aa : that p -> q -> r

      {move 2}
```

```
>>> declare bb that (p & q) -> r

bb : that (p & q) -> r

{move 2}

>>> comment we start by proving p & q -> \
    r using aa

{move 2}

>>> open

   {move 3}

   >>> declare cc that p & q

   cc : that p & q

   {move 3}

   >>> open

      {move 4}

      >>> define dd : Simp1 cc

      dd : that p

      {move 3}

      >>> define ee : Simp2 cc

      ee : that q

      {move 3}
```

```
      >>> define ff : Mp aa dd

      ff : that q -> r

      {move 3}

      >>> define gg : Mp ff ee

      gg : that r

      {move 3}

      >>> close

   {move 3}

   >>> define hh cc : gg

   hh : [(cc_1 : that p & q) =>
       (--- : that r)]

   {move 2}

   >>> close

{move 2}

>>> define ii aa : Deduction hh

ii : [(aa_1 : that p -> q -> r) =>
    (--- : that (p & q) -> r)]

{move 1}

>>> declare jj that (p & q) -> r

jj : that (p & q) -> r
```

```
{move 2}

>>> open

   {move 3}

   >>> declare kk that p

   kk : that p

   {move 3}

   >>> open

      {move 4}

      >>> declare ll that q

      ll : that q

      {move 4}

      >>> open

         {move 5}

         >>> define mm : Conj kk ll

         mm : that p & q

         {move 4}

         >>> define nn : Mp jj mm

         nn : that r

         {move 4}
```

```
        >>> close

      {move 4}

      >>> define oo ll : nn

      oo : [(ll_1 : that q) => (---
          : that r)]

      {move 3}

      >>> define rr : Deduction oo

      rr : that q -> r

      {move 3}

      >>> close

   {move 3}

   >>> define ss kk : rr

   ss : [(kk_1 : that p) => (---
       : that q -> r)]

   {move 2}

   >>> define tt : Deduction ss

   tt : that p -> q -> r

   {move 2}

   >>> close

{move 2}
```

```
   >>> define uu jj : tt

   uu : [(jj_1 : that (p & q) -> r) =>
       (--- : that p -> q -> r)]

   {move 1}

   >>> close

{move 1}

>>> define Export p q r : Bicondintro \
    ii, uu

Export : [(p_1 : prop), (q_1 : prop), (r_1
    : prop) =>
    ({def} Bicondintro ([(aa_2 : that
       p_1 -> q_1 -> r_1) =>
       ({def} Deduction ([(cc_3 : that
          p_1 & q_1) =>
          ({def} (aa_2 Mp Simp1 (cc_3)) Mp
          Simp2 (cc_3) : that r_1)]) : that
       (p_1 & q_1) -> r_1)], [(jj_2
       : that (p_1 & q_1) -> r_1) =>
       ({def} Deduction ([(kk_3 : that
          p_1) =>
          ({def} Deduction ([(ll_4
             : that q_1) =>
             ({def} jj_2 Mp kk_3 Conj
             ll_4 : that r_1)]) : that
          q_1 -> r_1)]) : that p_1 ->
       q_1 -> r_1)]) : that (p_1 ->
    q_1 -> r_1) <-> (p_1 & q_1) -> r_1)]

Export : [(p_1 : prop), (q_1 : prop), (r_1
    : prop) => (--- : that (p_1 -> q_1
    -> r_1) <-> (p_1 & q_1) -> r_1)]
```

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

This proof has the exact structure of a natural deduction proof with nested proofs of implications. At certain points we went one move deeper than strictly necessary so that definitions which amount to lines in the proof would be locally constant expressions rather than functions with hypotheses as explicit parameters. The shallower style is perfectly manageable. Another thing to notice is that definitions made in moves below move 0 are expanded out (any definition made in a move is expanded out everywhere when that move is closed) so reading the term defining `Export` will actually reveal the complete proof in an unaccustomed, more algebraic style. This expansion of the length of a proof term does not occur when lemmas are proved at move 0.

In large proofs, displayed proof terms can get quite large. Your devoted author actually *read* the term defining `Export`, and it does say the right thing (reading it requires awareness that `Mp` and `Conj` are unexpectedly being used as infixes, and that all infix operators have the same precedence and grouping is to the right).

It is also worth noticing here that the feedback from Lestrade includes both the abstraction term representing `Export` as a function (with the large embedded term embodying its definition) and the shorter term which is an abstraction term representing the sort of `Export`. The latter term is easier to read to see what has been proved rather than how to prove it.

This proof is large enough to exhibit Lestrade's automatic indentation of text to indicate both depth in `open/close` blocks and depth in abstraction terms (these two depths being intellectually very closely related).

It should be also noted that the settings of Lestrade under which this is run do not display bodies of definitions except in move 0 declarations; this is a setting designed to minimize clutter but turning if off might be interesting for our present purposes.

4 Translation of an important mathematical text: Zermelo set theory

In this section I use Lestrade to analyze an index text of the foundations of mathematics, Zermelo's original paper on the axiomatics of set theory, and at the same time exhibit what Lestrade can do and how a Lestrade book can relate to an underlying test.

Zermelo's paper is written in numbered paragraphs. I paraphrase essentials of each paragraph and follow with any relevant Lestrade declarations.

1. The text says that set theory is concerned with a domain $\mathcal{B}$ of individuals which we will call simply objects, some of which are sets. If two objects a, b are the same we write $a = b$, and otherwise we write $a \neq b$. We say that an object exists if it belongs to $\mathcal{B}$; we say of a class $\mathcal{R}$ of objects that there exist objects of this class iff $\mathcal{R}$ contains at least one individual of this class.

   An important observation here is that the class $\mathcal{B}$ and the classes $\mathcal{R}$ cannot be interpreted as sets in $\mathcal{B}$, and the notion of being contained in $\mathcal{B}$ or a class $\mathcal{R}$ mentioned here cannot be the relation $\in$ introduced in the next paragraph.

   There is no need to declare $\mathcal{B}$: it can be construed as the Lestrade sort `obj`. A class in the sense mentioned here can be viewed as a Lestrade function from `obj` to `prop`.

```
begin Lestrade execution

   >>> clearcurrent
{move 1}

   >>> declare a obj

   a : obj

   {move 1}

   >>> declare b obj
```

```
   b : obj

   {move 1}

   >>> declare c obj

   c : obj

   {move 1}

>>>postulate isaset a prop

   >>> postulate = a b prop

   = : [(a_1 : obj), (b_1 : obj) =>
       (--- : prop)]

   {move 0}

   >>> postulate Refleq a that a = a

   Refleq : [(a_1 : obj) => (--- : that
       a_1 = a_1)]

   {move 0}

   >>> declare eqev that a = b

   eqev : that a = b

   {move 1}

   >>> declare R [c => prop]

   R : [(c_1 : obj) => (--- : prop)]

   {move 1}
```

```
>>> declare rev that R a

rev : that R (a)

{move 1}

>>> postulate Subs eqev R, rev that R b

Subs : [(.a_1 : obj), (.b_1 : obj), (eqev_1
    : that .a_1 = .b_1), (R_1 : [(c_2
       : obj) => (--- : prop)]), (rev_1
    : that R_1 (.a_1)) => (--- : that
    R_1 (.b_1))]

{move 0}

>>> define Subs2 eqev rev : Subs eqev, R, rev

Subs2 : [(.a_1 : obj), (.b_1 : obj), (eqev_1
    : that .a_1 = .b_1), (.R_1 : [(c_2
       : obj) => (--- : prop)]), (rev_1
    : that .R_1 (.a_1)) =>
    ({def} Subs (eqev_1, .R_1, rev_1) : that
    .R_1 (.b_1))]

Subs2 : [(.a_1 : obj), (.b_1 : obj), (eqev_1
    : that .a_1 = .b_1), (.R_1 : [(c_2
       : obj) => (--- : prop)]), (rev_1
    : that .R_1 (.a_1)) => (--- : that
    .R_1 (.b_1))]

{move 0}

>>> postulate Exists R prop

Exists : [(R_1 : [(c_2 : obj) =>
       (--- : prop)]) => (--- : prop)]
```

```
{move 0}

>>> postulate Ei R, a rev that Exists \
    R

Ei : [(a_1 : obj), (R_1 : [(c_2
       : obj) => (--- : prop)]), (rev_1
    : that R_1 (a_1)) => (--- : that
    Exists (R_1))]

{move 0}

>>> declare p prop

p : prop

{move 1}

>>> declare existsev that Exists R

existsev : that Exists (R)

{move 1}

>>> declare wev [a rev => that p]

wev : [(a_1 : obj), (rev_1 : that
    R (a_1)) => (--- : that p)]

{move 1}

>>> postulate Witnessintro existsev, wev \
    that p

Witnessintro : [(.R_1 : [(c_2 : obj) =>
       (--- : prop)]), (.p_1 : prop), (existsev_1
    : that Exists (.R_1)), (wev_1
```

```
      : [(a_2 : obj), (rev_2 : that
         .R_1 (a_2)) => (--- : that .p_1)]) =>
      (--- : that .p_1)]

   {move 0}
end Lestrade execution
```

Here I declare the basic properties of equality (reflexivity and substitution), expressing the stated meaning of $a = b$.

I need to define or declare inequality. I am not certain as I write whether I will exploit the logic above or extract logic declarations local to this development from what Zermelo does. How I declare inequality depends on the decision about this.

I introduce the basic declarations for existence of something in a class (which is actually the existential quantifier).