Computer implementation of a philosophy of mathematics: reflections on the Lestrade dependent type theory and theorem prover

M. Randall Holmes

virtual logic seminar 3/20/2020; some errors in the original slides are corrected. The extended proof at the end might be interesting additional reading.
preparing part 2 for the seminar of 4/9/2020.

Abstract

This is a topic I have talked about before. During my sabbatical I have been reimplementing my dependent type theory prover from the ground up and working on a paper about it. I'll talk about what the dependent type theory is, exhibit something about what the implementation looks like, and discuss the grounds for the assertion implicit in the title that I am in fact implementing some ideas about philosophy of mathematics in the type theory and the software.

An overview

There are three levels to what I am talking about:

1. A view of the philosophy of mathematics

2. A dependent type theory, which I call the Lestrade* logic.

3. Computer software for development and display of Lestrade declarations, which I call the Lestrade Type Inspector.

*I am already guilty of calling a theorem prover Watson, and I hope I may be forgiven for another literary play on my name.

A philosophical issue, with perhaps dubious proposals

I shall begin at the top level, with a philosophical question. This is the issue of whether we are dealing in mathematics with actual infinities.

The current style is to assume flatly that we are postulating actual infinities. For example, the function $f(x) = x^2+1$ (for x a real variable) is implemented for us by the set

$$\{(x, y) : x \in \mathbb{R} \wedge y = x^2\},$$

a table of all input and output values of the function of (uncountable!) infinite size.

A hint at a different view is found in the representation of f as $(\lambda x : x^2 + 1)$, an expression with a hole in it. This expression is of course a surveyable finite object, in which x can be replaced by any notation r for a real number to give the function output $r^2 + 1$ associated with r as input.

At the risk of withering scorn directed at us from beyond the grave by Bertrand Russell, we propose to think of the input r as an arbitary real number, with which we can associate a still "arbitrary" number $r^2 + 1$ (again, a real number of dubious ontological status, not an expression), and with the partially arbitrary $r^2 + 1$ we can further associate the function $(r \mapsto r^2 + 1)$, which is a definite object without the taint of vagueness associated with the input and output expressions. How is sense to be made of such an account?

We propose to justify this by traffic with a scheme of possible worlds. Following Lestrade parlance, we refer to the worlds as "moves". In move 0, we find everything fixed and immutable which we have encountered so far, including such items as 3, the operation of addition, and indeed $(r \mapsto r^2 + 1)$. In move 1, we can find arbitrary objects of any sort we have postulated, such as the arbitrary real number r (its sort `Real` is found at move 0), and objects like $r^2 + 1$ with more complicated representations. The modality here might be taken to be temporal: the arbitrary real number r may be thought of as "any real number we may encounter in the future".

Now we provide a basic construction taking an object in move 1 depending on completely arbitrary variables in move 1 to the function implementing it. If we ever encounter a real number r, we can further construct r^2+1, itself a move 1 variable entity, but not a completely arbitrary one. The way in which r^2+1 depends on r, which we may write $((r : \texttt{Real}) \mapsto r^2+1)$, is a definite function entity in move 0: the process of abstraction eliminates its variable character.

I did mention withering criticism from the shade of Bertrand Russell. Bertie would point out now that my arbitrary real number r is a quite incoherent object? Is it positive? negative? Greater than or less than 1? Rational or irrational?

I address this question indirectly by giving an account of the humble operation of division. I pull an arbitrary real r and an arbitrary real s out of the inchoate sea of move 1. I cannot immediately postulate $\frac{r}{s}$, because s might be zero. But suppose in addition that I have evidence e that $s \neq 0$. I can postulate an operation of division:

$$\texttt{Divide}(r : \texttt{Real}, s : \texttt{Real}, e : \texttt{that}\, s \neq 0)$$

can be postulated.

What is this $s \neq 0$? Of course it is a composite expression, $\neg s = 0$, of type $\texttt{prop}$, a proposition. It is a move 1 entity, depending on the completely arbitrary real s. It is obtained by applying negation (whatever it may be) to $s = 0$. So we see $\neg : ((p : \texttt{prop}) \Rightarrow \texttt{prop})$ and $=: ((r : \texttt{Real}), (s : \texttt{Real}) \Rightarrow \texttt{prop})$ as presupposed basic notions (themselves entities at move 0, though here we have instances of their use at move 1).

The truly interesting move here is the introduction of the variable/arbitrary entity e of type $\texttt{that}\, s \neq 0$. With any proposition p we are associating a type $\texttt{that}\, p$ of evidence for p (I almost said that e is a proof of p, and I almost said that e inhabits $\texttt{that}\ p$; I may yet say such things but there are reasons to resist these manners of speaking).

And our answer to Bertie's shade is that we can postulate whatever partial information we need about arbitrary objects in move 1 (or any move of higher index) in the form of evidence for the truth value of propositions of interest about those arbitrary objects.

You may notice that I have introduced all objects as sorted. In the usual foundations of mathematics in set theory, we do of course talk about mathematical objects of various sorts, but we have been trained to think of all these objects as belonging to a common sort of sets, and further we tend to think of the sorts themselves as sets! This is not the basic view of the Lestrade logic (though Lestrade does support implementation of the usual set theoretic view of the world, as you will see if you have patience with me).

We have just introduced a subtle and powerful feature of the Lestrade logic's sorts/types: Lestrade is dependently typed, meaning that there are types which depend on variables, and there are functions in which the type of an argument later in its argument list (or of the output) may depend on an earlier argument, as in the case of our presentation of the division function.

There is no reason for the system of possible worlds (levels of vagueness or futurity) to be confined to moves 0 and 1. If we have an arbitrary real number r (move 1) we certainly have a function $(x \mapsto x+r)$ ("add r"). How is this to be understood in our scheme? Reach into move 1 and choose the arbitrary r. Reach further into move 2 (where r appears as an (unknown from our move 0 standpoint) constant!), introduce a variable object x at move 2 and form the expression $x+r$, a move 2 variable expression in which x is varying and r is supposed held fixed. Now the same procedure we have postulated for forming functions admits the formation of $(x \mapsto x+r)$, the function of adding r, at move 1.

The introduction of variable objects at different moves corresponds to phenomena which we expect our students to understand in undergraduate mathematics. Consider a function $z = ax + by + c$, whose graph is a plane. Can you see that this is a move 1 entity (depending on the unknown constants a, b, c which are move 1 real variables) formed by abstraction from the move 2 entity $ax + by + c$, in which x and y are move 2 variables? We quite often tell our students that letters are constants which are clearly (even to them) themselves variables. Here we formalize a relative notion of variable status.

(I could pause and actually declare the function $z = ax + by + c$ here)

We give a formal account of the Lestrade logic. Referents of general terms of our language are *entities* (further subdivided into objects and functions) and *sorts* (entities have sorts, which are the types of the Lestrade scheme).

Objects have object sorts. No specific objects are postulated by the Lestrade logic, postulation of specific objects being the privilege of Lestrade theories. There are specific object sorts and object sort constructions provided by Lestrade, which are exhaustive: all objects are of these sorts.

1. There is a sort `prop` of propositions.

2. There is a sort `type` of "type labels": an example would be `Real` in the examples above. We call objects of sort `type` "type labels" because we prefer to view a sort as a feature of an entity rather than a collection of entities (as one of our philosophical aims is to avoid the need to postulate actual infinities).

3. There is a sort `obj` of "untyped mathematical objects". In an implementation of ZFC, the sets might be of sort `obj`.

4. For p of sort `prop`, there is a sort `that` p. This is the sort of evidence for the truth of p. We could say "proofs" but this would commit us to a constructive view of logic,

and the Lestrade logic does not commit us to such a view (demonstrating that such a philosophical approach does not commit us to constructive mathematics is one of our philosophical aims).

5. For τ of sort `type`, there is a sort `in` τ. If `Real` is the type label of real numbers (an object of sort `type`), then the specific object π is of sort `in Real`. In the discussion above, the references to the sort of real numbers are actually references to `in Real`, but it is useful to have objects associated with sorts.

That is the complete scheme of object sorts. Of course, it is expandable by introducing propositions and type labels.

In declaring functions, we get into more fiddly issues, and I may skate over some technicalities unless challenged.

Each Lestrade function has a fixed arity n (it takes n arguments). The arguments may be of object or function sorts: the output is always of an object sort.

The shape of our notation for a Lestrade function sort is $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (-, \tau))$. Each τ_i is an object or function sort, and x_i is a variable of type τ_i. τ is an object sort. Each τ_i may depend on variables x_j with $j < i$. τ may depend on any or all of the x_i's. The variables x_i are bound in this notation.

If f is of type $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (-, \tau))$ then $f(t_1, \ldots, t_n)$ is well-formed iff τ_1 is the sort of t_1 and either $n = 1$ and $f(t_1)$ is of sort $\tau[t_1/x_1]$ or $n > 1$ and $f^*(t_2, \ldots, t_n)$ is well-formed, where f^* is of sort

$((x_1, \tau_1[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1]) \Rightarrow (-, \tau[t_1/x_1]))$:

$f(t_1, \ldots, t_n)$ has the same sort as $f^*(t_2, \ldots, t_n)$.

$[t_1/x_1]$ denotes substitution of t_1 for x_1, involving the usual formalities about bound variable renaming.

Terms for objects are always either atomic or application terms $f(t_1, \ldots, t_n)$ in which f is always an *atomic* function term and each t_i is an object or function term of appropriate type.

Terms for functions are either atomic or of the form $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (\delta, \tau))$ [this is a lambda term]. This term is of type

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

The result of replacing f with

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

in $f(t_1, \ldots, t_n)$, which only makes sense if f has the correct type and $f(t_1, \ldots, t_n)$ is well-formed as indicated above, is $\delta[t_1/x_1]$ if $n = 1$ and otherwise the same as the result of replacing f^* with

$$((x_1, \tau_1[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1])$$

$$\Rightarrow (\delta[t_1/x_1], \tau[t_1/x_1]))$$

in $f^*(t_2, \ldots, t_n)$.

This is the dependently typed version of the usual procedure of beta reduction for evaluating applications of lambda terms: and note that we follow Russell in his Principia in not allowing lambda terms* in applied position at all: substitution of a lambda term for an applied function variable triggers beta reduction.

We have now actually described the entire Lestrade logic, except for observations that terms with variable binding are equivalent where this can be established by renaming bound variables, and it is possible to define atomic object and function terms, and terms which can be shown to be equivalent by definitional expansion to the same form are equivalent. Computational equivalence of types is required for recognition that certain terms are well-typed; this is why this has to be mentioned.

*what he calls "propositional functions"

Some declarations in logic

In the last part of today's talk, we develop basic declarations on logic in the Lestrade Type Inspector. We have aims at two levels: one is to show what the Inspector is like, and the other is to demonstrate what it looks like to prove a theorem in this environment.

```
begin Lestrade execution

   >>> 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}
end Lestrade execution
```

We declare propositional variables p and q and use them as parameters in the declaration of conjunction (and).

```
begin Lestrade execution

   >>> declare pp that p


   pp : that p


   {move 1}

   >>> declare qq that q


   qq : that q


   {move 1}

   >>> postulate Conj0 p q pp qq that p & q


   Conj0 : [(p_1 : prop), (q_1 : prop), (pp_1
       : that p_1), (qq_1 : that q_1) =>
       (--- : that p_1 & q_1)]


   {move 0}

   >>> 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
```

We declare variables witnessing truth of p and q and define the rule of conjunction. This rule says that if p and q are true, $p \wedge q$ is true. So it has four arguments, p, q, pp, qq, but the first two can be deduced from the sorts of the last two, and we show that the prover can actually detect such implicit arguments. Notice that the second version `Conj` is declared with two parameters, but the system detects that it has two more (the implicit parameters being adorned with dots).

```
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
```

And here we declare the rules of simplification.

Declarations for implication

```
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}

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


   {move 0}

   >>> declare ss that p -> q


   ss : that p -> q


   {move 1}

   >>> postulate Mp pp ss that q


   Mp : [(.p_1 : prop), (.q_1 : prop), (pp_1
       : that .p_1), (ss_1 : that .p_1
       -> .q_1) => (--- : that .q_1)]


   {move 0}
end Lestrade execution
```

We declare the operation of implication and the rule of modus ponens (which you may notice has implicit arguments). The `clearcurrent` command clears the variable declarations in move 1; we could use a style in which we had continuously accumulating declared parameters, but this would lead to memory challenges.

The rule for proving implications (the familiar deduction theorem) is of quite a different kind.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> declare pp1 that p


      pp1 : that p


      {move 2}

      >>> postulate ded0 pp1 that q


      ded0 : [(pp1_1 : that p) => (---
          : that q)]


      {move 1}

      >>> close


   {move 1}

   >>> postulate Ded0 ded0 that p -> q
   Ded0 : [(.p_1 : prop), (.q_1 : prop), (ded0_1
       : [(pp1_2 : that .p_1) => (---
          : that .q_1)]) => (--- : that
       .p_1 -> .q_1)]


   {move 0}

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


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


   {move 1}

   >>> postulate Ded ded that p -> q


   Ded : [(.p_1 : prop), (.q_1 : prop), (ded_1
       : [(pp_2 : that .p_1) => (--- : that
          .q_1)]) => (--- : that .p_1
       -> .q_1)]


   {move 0}
end Lestrade execution
```

I present the declaration of rule of deduction in two different styles. The second is the one I would normally use. The first makes a philosophical point. Notice that in the second approach I declare the variable `ded0` in the function type $((pp : \texttt{that}\, p) \Rightarrow \texttt{that}\, q)$. This requires me as the user to write a term representing this function sort.

There is a style, illustrated here, in which the user can avoid ever writing function sorts or lambda terms (though they inevitably show up in output). In this case, we open a new environment (go into move 2) and declare `ded0` as a primitive (so it shows up in move 1); closing the environment leaves `ded0` as a variable of function type in move 1. We are trading explicit variable binding in terms for the implicit variable binding involved in the move system.

Now we prove a theorem!

```
begin Lestrade execution

   >>> comment We use the goal command to \
       display what we want to prove .We have \
       found this useful in practice for developing \
       large proofs .


   {move 1}

   >>> goal that (p & q) -> q & p


   that (p & q) -> q & p


   {move 1}

   >>> open


      {move 2}

      >>> declare thehyp that p & q


      thehyp : that p & q


      {move 2}

      >>> goal that q & p
```

```
that q & p


{move 2}

>>> open


   {move 3}

   >>> define line1 : Simp2 thehyp


   line1 : that q


   {move 2}

   >>> define line2 : Simp1 thehyp


   line2 : that p


   {move 2}

   >>> close


{move 2}

>>> define conjsymm thehyp : Conj line1 \
    line2
   conjsymm : [(thehyp_1 : that p & q) =>
       (--- : that q & p)]


   {move 1}

   >>> close


{move 1}

>>> define Conjsymm p q : Ded conjsymm


Conjsymm : [(p_1 : prop), (q_1 : prop) =>
    ({def} Ded ([(thehyp_2 : that p_1
       & q_1) =>
       ({def} Simp2 (thehyp_2) Conj
       Simp1 (thehyp_2) : that q_1 & p_1)]) : that
    (p_1 & q_1) -> q_1 & p_1)]


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


{move 0}

>>> declare thehyp2 that p & q


thehyp2 : that p & q


{move 1}
```

```
   >>> define Conjsymm2 p q : Ded [thehyp2 \
          => Conj (Simp2 thehyp2, Simp1 thehyp2)]


   Conjsymm2 : [(p_1 : prop), (q_1 : prop) =>
       ({def} Ded ([(thehyp2_2 : that
          p_1 & q_1) =>
          ({def} Simp2 (thehyp2_2) Conj
          Simp1 (thehyp2_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}

   >>> comment The same proof in an alarmingly \
       compressed format, achieved by explicitly \
       presenting the proof object as a term \
       .


   {move 1}
end Lestrade execution
```

Here we present a proof of the theorem $(p \wedge q) \rightarrow (q \wedge p)$ of propositional logic. This is presented in two styles, one an extended proof using the move system to emulate reasoning under a hypothesis, and the other a brief presentation of the exact object which serves as the proof as an explicit lambda term.

In this case, I would say the extended style emulates what we do as mathematicians, though the more telegraphic style might have its uses.

A philosophical point to note here is that for us a proof is a mathematical object, and we introduce it by defining it. A proof of $(p \wedge q) \rightarrow (q \wedge p)$ is an object of sort
`that (p & q) -> q & p`, and our formal proof is the careful definition (in either style) of an object of this type.

The exact scheme of identification of proofs with mathematical objects is a version of the Curry-Howard isomorphism. Typically, a proof of $p \wedge q$ is regarded as a pair of a proof of p and a proof of q: `Conj` could be regarded as a pair construction and `Simp1` and `Simp2` as its projections (and indeed the declarations of a primitive pair operation of ordinary objects would be isomorphic: we could present this in the second talk, or actually do the declarations live to make this point). A proof of $p \rightarrow q$ is usually described as a function from proofs of p to proofs of q: we regard a proof of $p \rightarrow q$ as produced from such a function by a constructor `Ded`. This isn't purely an accident of our system: briefly, I'll say that we avoid identifying Lestrade objects and functions and we give the former a more central ontological status. Mathematical functions will actually be packaged as objects in the same way in Lestrade theories which manipulate them as sets; a sophisticated way of expressing this is that we actually view Lestrade's functions as proper class functions rather than sets. In a second talk where we present an axiomatization of Zermelo set theory, reasons for this view will be presented.

```
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}

   >>> 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 pp that p


pp : that p


{move 1}

>>> comment this function can be used \
    to force displayed types into desired \
    forms .


{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}

>>> goal that (p -> q -> r) <-> (p & q) -> \
    r


that (p -> q -> r) <-> (p & q) ->
 r


{move 1}

>>> open


   {move 2}

   >>> declare hyp1 that p -> q -> r


   hyp1 : that p -> q -> r


   {move 2}

   >>> goal that (p & q) -> r


   that (p & q) -> r


   {move 2}

   >>> open
```

```
{move 3}

>>> declare hyp2 that p & q


hyp2 : that p & q


{move 3}

>>> open


   {move 4}

   >>> define line1 : Simp1 hyp2


   line1 : that p


   {move 3}

   >>> define line2 : Simp2 hyp2


   line2 : that q


   {move 3}

   >>> define line3 : Mp line1, hyp1


   line3 : that q -> r
```

```
      {move 3}

      >>> define linea4 : Mp line2, line3


      linea4 : that r


      {move 3}

      >>> close


   {move 3}

   >>> define line4 hyp2 : linea4


   line4 : [(hyp2_1 : that p & q) =>
       (--- : that r)]


   {move 2}

   >>> close


{move 2}

>>> define line5 hyp1 : Ded line4


line5 : [(hyp1_1 : that p -> q ->
    r) => (--- : that (p & q) ->
    r)]


{move 1}

>>> define line6 : Ded line5


line6 : that (p -> q -> r) -> (p & q) ->
 r


{move 1}

>>> open


   {move 3}

   >>> declare hyp3 that (p & q) -> \
       r


   hyp3 : that (p & q) -> r


   {move 3}

   >>> goal that p -> q -> r


   that p -> q -> r


   {move 3}
```

```
>>> open

   {move 4}

   >>> declare hyp4 that p


   hyp4 : that p


   {move 4}

   >>> goal that q -> r


   that q -> r


   {move 4}

   >>> open


      {move 5}

      >>> declare hyp5 that q


      hyp5 : that q


      {move 5}

      >>> open
   {move 6}

   >>> define line7 : Conj \
       hyp4 hyp5


   line7 : that p & q


   {move 5}

   >>> define linea8 : Mp \
       line7, hyp3


   linea8 : that r


   {move 5}

   >>> close


{move 5}

>>> define line8 hyp5 : linea8


line8 : [(hyp5_1 : that
    q) => (--- : that r)]


{move 4}

>>> close
```

```
      {move 4}

      >>> define line9 hyp4 : Ded line8


      line9 : [(hyp4_1 : that p) =>
          (--- : that q -> r)]


      {move 3}

      >>> close


   {move 3}

   >>> define line10 hyp3 : Ded line9


   line10 : [(hyp3_1 : that (p & q) ->
       r) => (--- : that p -> q ->
       r)]


   {move 2}

   >>> close


{move 2}

>>> define line11 : Ded line10
```

```
   line11 : that ((p & q) -> r) ->
    p -> q -> r


   {move 1}

   >>> define line12 : Fixform ((p -> \
       q -> r) <-> (p & q) -> r, Conj \
       line6 line11)


   line12 : that (p -> q -> r) <-> (p & q) ->
    r


   {move 1}

   >>> close


{move 1}

>>> define Exportation p q r : line12


Exportation : [(p_1 : prop), (q_1
    : prop), (r_1 : prop) =>
    ({def} ((p_1 -> q_1 -> r_1) <->
    (p_1 & q_1) -> r_1) Fixform Ded
    ([(hyp1_4 : that p_1 -> q_1 -> r_1) =>
       ({def} Ded ([(hyp2_5 : that
          p_1 & q_1) =>
          ({def} Simp2 (hyp2_5) Mp Simp1
          (hyp2_5) Mp hyp1_4 : that r_1)]) : that
       (p_1 & q_1) -> r_1)]) Conj
    Ded ([(hyp3_4 : that (p_1 & q_1) ->
          r_1) =>
          ({def} Ded ([(hyp4_5 : that
             p_1) =>
             ({def} Ded ([(hyp5_6 : that
                q_1) =>
                ({def} hyp4_5 Conj hyp5_6
                Mp hyp3_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)]


   Exportation : [(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
```

Quantification

We introduce the universal quantifier and its associated rules. Here we introduce a universal quantifier over the type `obj` of "untyped mathematical objects" (which we will use to implement sets).

```
begin Lestrade execution

   >>> clearcurrent

{move 1}

   >>> declare x obj


   x : obj


   {move 1}

   >>> declare pred [x => prop]


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


   {move 1}

   >>> postulate Forall pred : prop
```

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


{move 0}

>>> declare univev that Forall pred


univev : that Forall (pred)


{move 1}

>>> declare y obj


y : obj


{move 1}

>>> postulate Ui univev y that pred y


Ui : [(.pred_1 : [(x_2 : obj) =>
       (--- : prop)]), (univev_1
    : that Forall (.pred_1)), (y_1
    : obj) => (--- : that .pred_1 (y_1))]


{move 0}

>>> declare generalev [y => that pred \
        y]


generalev : [(y_1 : obj) => (--- : that
    pred (y_1))]


{move 1}

>>> postulate Ug generalev that Forall \
    pred


Ug : [(.pred_1 : [(x_2 : obj) =>
       (--- : prop)]), (generalev_1
    : [(y_2 : obj) => (--- : that .pred_1
       (y_2))]) => (--- : that Forall
    (.pred_1))]


{move 0}

>>> 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}

   >>> define Ugtest : Ug Refleq


   Ugtest : [
       ({def} Ug (Refleq) : that Forall
       ([(x''_2 : obj) =>
          ({def} x''_2 = x''_2 : prop)]))]


   Ugtest : that Forall ([(x''_2 : obj) =>
       ({def} x''_2 = x''_2 : prop)])


   {move 0}
end Lestrade execution
```

Some sorting out

We do some analysis of typing used here. To begin with, the parameter pred (for "predicate") is quite naturally a function which takes an object of sort `obj` as input and gives a proposition (a truth value) as output. The universal quantifier itself is a gadget which takes predicates as input and returns propositions as output. The intention of course that that we believe `Forall pred` to be the case if we believe that `pred x` is true for any `x`.

Our philosophical motivation disinclines us to claim that evidence for `Forall pred` consists of evidence for `pred x` for each `x` of type `obj` individually: this would seem to entail having an infinite body of evidence.

The rule `Ui` is a function which given (implicitly) a predicate `pred` and (explicitly) evidence for `Forall pred` and an object `y` of sort `obj`, returns evidence for `pred y`.

The typing of `Ui` and general consideration of how we think about functions in Lestrade gives the idea of what is happening: if we believe $(\forall y : \phi(y))$ this means not that we believe each $\phi(y)$ individually (that would be a lot of beliefs) but that we have a machine which, presented with an object x, spits back at us evidence for $\phi(x)$

The rule of universal generalization is similarly motivated as a converse operation. The parameter `generalev` is a function which takes an object y and returns evidence for $\phi(y)$: the function `Ug` takes any such function as input and returns evidence for $(\forall x : \phi(x))$. A difference between the treatment here and the treatment in Automath is that the function `Ug` would in effect be the identity: in Automath the function `generalev` would simply be evidence for $(\forall x : \phi(x))$ itself; our typing scheme precludes this identification.

In our next block of Lestrade code, we declare the other basic rule of equality (substitution).

```
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 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 Subsfull pred eqev predev : Subs \
    eqev predev


Subsfull : [(.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} eqev_1 Subs predev_1 : that
      pred_1 (.y_1))]


   Subsfull : [(.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
```

Our reasons for supplying two different versions of `Subs` should be revealed here (along with limitations of the implicit argument inference mechanism).

```
begin Lestrade execution

   >>> clearcurrent

{move 1}

   >>> declare x obj


   x : obj


   {move 1}

   >>> declare y obj


   y : obj


   {move 1}

   >>> open


      {move 2}

      >>> declare eqev that x = y
```

```
eqev : that x = y


{move 2}

>>> declare z obj


z : obj


{move 2}

>>> define dudeqsymm eqev : Subs eqev \
    (Refleq x)


dudeqsymm : [(eqev_1 : that x = y) =>
    (--- : that y = y)]


{move 1}

>>> define eqsymm eqev : Subsfull eqev \
    [z => z = x] (Refleq x)


eqsymm : [(eqev_1 : that x = y) =>
    (--- : that y = x)]


{move 1}

>>> close
```

```
   {move 1}

   >>> define Eqsymm x y : Ded eqsymm


   Eqsymm : [(x_1 : obj), (y_1 : obj) =>
       ({def} Ded ([(eqev_2 : that x_1
          = y_1) =>
          ({def} Subsfull (eqev_2, [(z_3
             : obj) =>
             ({def} z_3 = x_1 : prop)], Refleq
          (x_1)) : that y_1 = x_1)]) : that
       (x_1 = y_1) -> y_1 = x_1)]


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


   {move 0}
end Lestrade execution
```

The effect of `Subs eqev predev`, where `predev` is evidence for `pred x`, is to replace `x` with `y` in `pred x`. An important feature of Lestrade is that we do not have to write a lambda term representing `pred` on every occasion: the implicit argument inference mechanism can often deduce what predicate of x is intended. But here we want to consider $x = x$ as a predicate of only the first occurrence of x, so we need to use the form `Subsfull` which actually gives the intended predicate as an explicit argument.

The Russell error

It's an error, not a paradox, and it isn't Russell's error: he pointed it out and tried to fix it. The Frege error?

It gives us an example of work with mathematical objects rather than proofs.

```
begin Lestrade execution

   >>> clearcurrent

{move 1}

   >>> declare x obj


   x : obj


   {move 1}

   >>> declare y obj


   y : obj


   {move 1}
```

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


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


{move 1}

>>> postulate E x y prop


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


{move 0}

>>> postulate Setof pred obj


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


{move 0}

>>> declare inev that x E Setof pred


inev : that x E Setof (pred)


{move 1}

>>> postulate Comp1 inev that pred x
```

```
   Comp1 : [(.x_1 : obj), (.pred_1 : [(x_2
          : obj) => (--- : prop)]), (inev_1
       : that .x_1 E Setof (.pred_1)) =>
       (--- : that .pred_1 (.x_1))]


   {move 0}

   >>> declare predev that pred x


   predev : that pred (x)


   {move 1}

   >>> postulate Comp2 x predev that x E Setof \
       pred


   Comp2 : [(x_1 : obj), (.pred_1 : [(x_2
          : obj) => (--- : prop)]), (predev_1
       : that .pred_1 (x_1)) => (--- : that
       x_1 E Setof (.pred_1))]


   {move 0}
end Lestrade execution
```

If you are beginning to understand the Lestrade idiom, you should sense disaster looming. Notice that the Lestrade logic in itself gives us no reason to believe the powerful assumptions `Comp1` and `Comp2`: its power and merit is that it allows us to formulate these assumptions and deduce their consequences.

We haven't introduced a final tiny bit of logic yet. After declaring the False and defining negation, we carry out the proof that this is all untenable. Our logic so far is constructive, by the way.

```
begin Lestrade execution

   >>> clearcurrent

{move 1}

   >>> declare p prop


   p : prop


   {move 1}

   >>> postulate False prop


   False : prop


   {move 0}

   >>> define ~ p : p -> False


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


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


   {move 0}

   >>> declare absurd that False
```

```
absurd : that False


{move 1}

>>> declare q prop


q : prop


{move 1}

>>> postulate Panic absurd q that q


Panic : [(absurd_1 : that False), (q_1
    : prop) => (--- : that q_1)]


{move 0}

>>> declare x obj


x : obj


{move 1}

>>> define russell x : ~ (x E x)


russell : [(x_1 : obj) =>
    ({def} ~ (x_1 E x_1) : prop)]
```

```
russell : [(x_1 : obj) => (--- : prop)]


{move 0}

>>> define Russell : Setof russell


Russell : [
    ({def} Setof (russell) : obj)]


Russell : obj


{move 0}

>>> open


   {move 2}

   >>> declare dangerhyp that Russell \
       E Russell


   dangerhyp : that Russell E Russell


   {move 2}

   >>> define line1 dangerhyp : Comp1 \
       dangerhyp
```

```
line1 : [(dangerhyp_1 : that Russell
    E Russell) => (--- : that russell
    (Russell))]


{move 1}

>>> define linea1 dangerhyp : Fixform \
    (~ (Russell E Russell), line1 dangerhyp)


linea1 : [(dangerhyp_1 : that Russell
    E Russell) => (--- : that ~ (Russell
    E Russell))]


{move 1}

>>> define lineb1 dangerhyp : Fixform \
    ((Russell E Russell) -> False, line1 \
    dangerhyp)


lineb1 : [(dangerhyp_1 : that Russell
    E Russell) => (--- : that (Russell
    E Russell) -> False)]


{move 1}

>>> define line2 dangerhyp : Mp dangerhyp \
    (line1 dangerhyp)


line2 : [(dangerhyp_1 : that Russell
        E Russell) => (--- : that False)]


   {move 1}

   >>> close


{move 1}

>>> define unfortunate : Fixform (~ (Russell \
    E Russell), Ded line2)


unfortunate : [
    ({def} ~ (Russell E Russell) Fixform
    Ded ([(dangerhyp_3 : that Russell
       E Russell) =>
       ({def} dangerhyp_3 Mp Comp1 (dangerhyp_3) : that
       False)]) : that ~ (Russell E Russell))]


unfortunate : that ~ (Russell E Russell)


{move 0}

>>> define moreunfortunate : Comp2 Russell \
    unfortunate


moreunfortunate : [
    ({def} Russell Comp2 unfortunate : that
    Russell E Setof ([(x'_3 : obj) =>
       ({def} ~ (x'_3 E x'_3) : prop)]))]
   moreunfortunate : that Russell E Setof
    ([(x'_3 : obj) =>
       ({def} ~ (x'_3 E x'_3) : prop)])


   {move 0}

   >>> define moreunfortunate2 : Fixform \
       (Russell E Russell, moreunfortunate)


   moreunfortunate2 : [
       ({def} (Russell E Russell) Fixform
       moreunfortunate : that Russell E Russell)]


   moreunfortunate2 : that Russell E Russell


   {move 0}

   >>> define mostunfortunate : Mp moreunfortunate \
       unfortunate


   mostunfortunate : [
       ({def} moreunfortunate Mp unfortunate
       : that False)]


   mostunfortunate : that False


   {move 0}
end Lestrade execution
```

And that about wraps it up for naive set theory.

An amusing technical point here is that we actually made no use whatsoever of the logic of negation. No special properties of `False` are used anywhere in this development: this could be adapted into a proof of any proposition at all, using only implication. (We presented the rule `Panic`, but we never used it).

Here we have presented enough logic and ontology in terms of Lestrade to get to a theory which was once regarded as adequate for mathematics. The revisions needed to formulate Zermelo set theory or type theory should be imaginable at this point.

The fact that things that we would usually regard as logical primitives (logical connectives, quantifiers and their rules) need to be explicitly declared (and can be declare in alternative forms if you prefer) is sometimes expressed by saying that systems like Lestrade are "logical frameworks" rather than being logics themselves.

Mathematics with typed objects

We present some basic declarations of familiar concepts using typed objects. We begin with ordered pairs in a typed context and cartesian product types.

```
begin Lestrade execution

   >>> clearcurrent

{move 1}

   >>> declare sigma type


   sigma : type


   {move 1}

   >>> declare tau type


   tau : type


   {move 1}

   >>> declare x in sigma


   x : in sigma


   {move 1}
```

```
>>> declare y in tau


y : in tau


{move 1}

>>> postulate * sigma tau type


*: [(sigma_1 : type), (tau_1 : type) =>
    (--- : type)]


{move 0}

>>> postulate ; x y in sigma * tau


;: [(.sigma_1 : type), (.tau_1 : type), (x_1
    : in .sigma_1), (y_1 : in .tau_1) =>
    (--- : in .sigma_1 * .tau_1)]


{move 0}

>>> declare z in sigma * tau


z : in sigma * tau


{move 1}
```

```
   >>> postulate pi1 z in sigma


   pi1 : [(.sigma_1 : type), (.tau_1
       : type), (z_1 : in .sigma_1 * .tau_1) =>
       (--- : in .sigma_1)]


   {move 0}

   >>> postulate pi2 z in tau


   pi2 : [(.sigma_1 : type), (.tau_1
       : type), (z_1 : in .sigma_1 * .tau_1) =>
       (--- : in .tau_1)]


   {move 0}
end Lestrade execution
```

This should look suspiciously familiar: in fact, it is parallel in every detail to the declaration of the logical operation of conjunction and its rules, an instance of the Curry Howard isomorphism. In the context of objects rather than proofs, we would like to say a little more.

```
begin Lestrade execution

   >>> declare x2 in sigma


   x2 : in sigma


   {move 1}

   >>> postulate == x x2 prop


   ==: [(.sigma_1 : type), (x_1 : in
       .sigma_1), (x2_1 : in .sigma_1) =>
       (--- : prop)]


   {move 0}

   >>> postulate Refleqt x that x == x


   Refleqt : [(.sigma_1 : type), (x_1
       : in .sigma_1) => (--- : that x_1
       == x_1)]


   {move 0}

   >>> declare eqevt that x == x2


   eqevt : that x == x2
```

```
{move 1}

>>> declare predt [x => prop]


predt : [(x_1 : in sigma) => (---
    : prop)]


{move 1}

>>> declare predevt that predt x


predevt : that predt (x)


{move 1}

>>> postulate Subst eqevt predevt that \
    predt x2


Subst : [(.sigma_1 : type), (.x_1
    : in .sigma_1), (.x2_1 : in .sigma_1), (eqevt_1
    : that .x_1 == .x2_1), (.predt_1
    : [(x_2 : in .sigma_1) => (---
       : prop)]), (predevt_1 : that
    .predt_1 (.x_1)) => (--- : that
    .predt_1 (.x2_1))]


{move 0}

>>> postulate Proj1 x y that pi1 (x ; y) == \
      x


   Proj1 : [(.sigma_1 : type), (.tau_1
      : type), (x_1 : in .sigma_1), (y_1
      : in .tau_1) => (--- : that pi1 (x_1
      ; y_1) == x_1)]


   {move 0}

   >>> postulate Proj2 x y that pi2 (x ; y) == \
      y


   Proj2 : [(.sigma_1 : type), (.tau_1
      : type), (x_1 : in .sigma_1), (y_1
      : in .tau_1) => (--- : that pi2 (x_1
      ; y_1) == y_1)]


   {move 0}
end Lestrade execution
```

Above we introduced the notion of equality for typed objects. Notice that the fact that this is a dependently typed ternary relation is hidden: the implicit argument inference mechanism allows us to treat it as in effect an overloaded binary relation which has an instance on each type. We are then able to supply crucial additonal information about the projection operators which we do not usually want about their analogues in the propositional realm, the rules of simplification. We *could* install similar additional rules on the proof side if we had a use for them.