Computer implementation of a philosophy of mathematics: the Lestrade dependent type system and the Lestrade Type Inspector which implements it

M. Randall Holmes

March 12, 2020

1 Introduction

We propose a dependent type system, which we call Lestrade[1], which is implemented by software which we call the Lestrade Type Inspector. We argue for a philosophical view under which Lestrade is seen to be a basic framework for the presentation of mathematical objects and proofs in general, and so the Type Inspector is seen as affording the opportunity to interact with general mathematical proofs and objects reliably on a computer.

There are three different levels which we need to serve in our exposition. We need to articulate a view of the philosophy of mathematics, we need to describe a specific dependent type theory intended to implement the philosophy, and we need to describe the software which implements the type theory. We will begin in the middle with an account of the dependent type theory, which will cast light on the other two matters, or which will force us to make points about the other two matters.

There is prior art. Lestrade is clearly a variant of the old system Automath and has simlar aims, though there are definitely differences. The type system of Lestrade is in a mathematical sense weaker than that of Automath

[1]The author already has a theorem proving system called Watson, and hopes that he may be forgiven for another literary play on his own name.

(which is a positive advantage when it comes to implementing mathematicial theories at lower levels of strength). We will compare name management in Lestrade and Automath (naturally, we do this because Lestrade has an advantage). Another system descended from Automath is Coq, and we may have something to say about differences between Coq and Lestrade. An obvious point is that Lestrade is (deliberately) a very minimal system; much more mathematics is built into Coq. Another point is that Lestrade (and Automath) have largely been used to implement theories with classical logic, while Coq in practice emphasizes constructive logic and is implemented with a constructive approach in mind.

2 Lestrade as a dependent type theory

In this section we discuss the specific dependent type theory of the Lestrade project.

Terms in the Lestrade language represent *sorts* and *entities*. The domain of entities is further subdivided into *objects* and *functions*. Entities (objects and functions) have sorts.

In a first indication of our prejudices, we note that it is important to think of the sort of an entity as a *feature* of the entity, not a collection of entities. We avoid consideration of infinite collections in our underlying philosophical viewpoint.

2.1 Object sorts

An outline of the sorts of object is easy enough to give.

1. Entities of sort `prop` are propositions.

2. Entities of sort `type` are type labels, entities correlated with sorts of typed mathematical object as we will see below.

3. Entities of sort `obj` may be thought of as untyped mathematical objects. In a Lestrade implementation of ZFC, all sets would be of the same sort `obj`.

4. Entities of sort `that` p, where p is a proposition, are proofs or evidence for the truth of p (we do not commit to an entity of sort `that` p being

a formal proof, which would be a constructivist view; we do say that if there is such an entity, p is true).

5. Entities of sort `in` τ, where τ is a type label, are objects of the type labelled by τ. For example, `Nat` of sort `type` (a label for the sort of natural numbers) would be used in the assertion that 3 is of sort `in Nat`.

It is important to notice our intention of treating mathematical proofs as objects: you should expect to see applications of the Curry-Howard isomorphism.

The basic theory of Lestrade does not provide us with any specific objects at all: these have to be postulated in a Lestrade theory.

2.2 Functions and function sorts

Any Lestrade function has a fixed arity (it always takes the same number of arguments). The arguments may be objects or functions. Each argument in a particular application of a function to a list of arguments is of a sort determined by the sort of the function and the specific preceding arguments. So the first argument of a specific function is always of the same sort, but the sort of the second argument may be computed from the first argument in a way dictated by the applied function. This is dependent type theory. The values of a Lestrade function are always objects (not functions).

Our philosophical viewpoint asserts itself again. A Lestrade function is not to be thought of as an infinite table of suitably sorted argument lists associated with object outputs. Instead it is to be thought of as an entity which responds when presented with appropriately typed inputs with an appropriately typed output. Such an entity does not need to be thought of as involving any kind of actual infinity.

We present this somewhat more formally. The notation for a Lestrade function sort is of the form $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (-, \tau))$, in which each x_i is a (bound) variable of sort τ_i. Each τ_i is a sort (either an object or a function sort) and τ is an object sort. Each τ_i may depend on variables x_j where $j < i$, and τ may depend on any of the x_i's.

A notation for a function of sort $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (-, \tau))$ may be atomic or may be of the form $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (\delta, \tau))$, where δ is a term of type τ. This last is a dependently typed λ-term.

2.3 Notation for objects: computation of types and beta reduction

Now we can describe notations for objects. These are either atomic (with one of the sorts indicated above) or applicative, notation of the form $f(t_1, \ldots, t_n)$, where f is an atomic notation for a function (lambda terms cannot be explicitly applied in Lestrade notation!), where f must be of a type

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

(yes the two n's must be the same), and the type of t_1 must be τ_1. If $n = 1$, the type of $f(t_1, \ldots, t_n)$ is τ. If $n > 1$, the type of $f(t_1, \ldots, t_n)$ is the same as the type of $f^*(t_2, \ldots, t_n)$, where f^* is of type

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

If we stipulate more specifically that f denotes the same function as $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (\delta, \tau))$, then f^* denotes the same function as

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

The notation $X[t_1/x_1]$ represents the result of substitution of t_1 for x_1 in the term X.

This exhibits the process of beta-reduction of "applied" lambda terms. In our notation (as was the case in the notation of *Principia Mathematica*) lambda terms do not appear in applied position: the notion of substitution is defined in such a way that beta reduction is carried out whenever a lambda term replaces an applied function variable.

The device of currying may be used to interpret terms $f(x_1, \ldots, x_i)$ where i is of arity less than n as a function of arity $n - i$; such terms can be used in input to the Type Inspector for convenience but are immediately transformed into lambda terms.

At this point, we have almost given a complete account of the basic Lestrade logic. The additional elements are that function sorts and lambda terms differing only up to bound variable renaming are equivalent and that atomic terms may be defined as denoting lambda terms, and that terms obtained by definitional expansion are equivalent to their originals. Finally, any atomic term f of sort $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (-, \tau))$ is equivalent to the lambda term $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (f(x_1, \ldots, x_n), \tau))$ (the principle of eta

reduction). Identity of terms plays an essential role because identity of types is involved in determining whether applicative terms are well-typed.

Persnickety details of the definition of substitution and of computable identity of terms can be discovered in the code for the Type Inspector. It is important to note that for a term to be well-typed, the equality of appropriate types must be computable from their given forms: there is no question of a term being possibly well-formed on the basis of an equation whose truth value we are unable to determine.

3 The Lestrade proof environment: parameters as arbitrary objects

We advocate an unpopular view of what is going on when we define a function with parameters or conider the semantics of a lambda term. A naive way of considering the definition $f(x) = x^2 + 1$ (where x is a real variable) is to say that we define f by considering squaring an arbitrary real number and adding 1. This is actually how we view this definition in Lestrade.

The mathematical environment for us is a finite system of possible worlds (which we actually call "moves") indexed by natural numbers starting with 0. There are at least two moves (so there are always move 0 and move 1). If $i + 2$ is the number of moves, we term move i the last move or the current move, and we call move $i + 1$ the next move. If we declare something in the next move, we declare it as a parameter, a completely arbitrary object. A useful modality to consider here is temporal: an object or function of a given sort declared at move 0 may be thought of as any object of that sort one might encounter in the future. If we have posited a real number x (a variable x of sort `in Real`, in Lestrade parlance, and we have already posited multiplication, 1 and addition, we can posit the expression x^2+1 at move $i+1$, but further we can postulate the function f such that $f(x) = x^2 + 1$ at move i. Since $f(x) = x^2 + 1$ involves definition notions (other than the variable x itself), it might be supposed declared at move 1. The need for further moves can be illustrated with natural ideas from algebra: declare addition, multiplication, 1, the type of reals, all at move 0 (where our definite ideas live). Declare a real parameter a at move 1. Declare a real parameter b at move 1. Open move 2 and declare a real parameter at move 2. The expression $ax + b$ makes sense at move 2, and the function $(x \mapsto ax + b)$ can be declared

at move 1. This example provides a framework to talk about variables such as x and (relative) constants like a, b which are clearly also variables.

Here is some actual dialogue with the Type Inspector making the points of the previous paragraph and some related points.'

```
begin Lestrade execution

   >>> postulate Real type


   Real : type


   {move 0}

   >>> postulate 1 in Real


   1 : in Real


   {move 0}
end Lestrade execution
```

We start in the default state of Lestrade, with two moves. The `postulate` command is used to introduce notions in move 0 (the primitive and defined notions of our theory without any arbitrary, hypothetical, or variable aspect. What we introduce is the type of reals and the constant real number 1.

```
begin Lestrade execution

   >>> declare a in Real


   a : in Real
```

```
   {move 1}

   >>> declare b in Real


   b : in Real


   {move 1}
end Lestrade execution
```

We introduce the real variables a and b.

```
begin Lestrade execution

   >>> 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}

   >>> define sq a : a * a
```

```
   sq : [(a_1 : in Real) =>
       ({def} a_1 * a_1 : in Real)]


   sq : [(a_1 : in Real) => (--- : in
       Real)]


   {move 0}
end Lestrade execution
```

Here we introduce new notions at move 0. These notions are functions, so they need parameters from move 1. The notions of addition and multiplication are introduced as primitives and the squaring function is defined.

```
begin Lestrade execution

   >>> define f a : sq a + 1


   f : [(a_1 : in Real) =>
       ({def} sq (a_1) + 1 : in Real)]


   f : [(a_1 : in Real) => (--- : in
       Real)]


   {move 0}
end Lestrade execution
```

We introduce the function $f(a) = a^2 + 1$ with which we started our discussion.

```
begin Lestrade execution

   >>> open


      {move 2}

      >>> declare x in Real


      x : in Real


      {move 2}
end Lestrade execution
```

The `open` command adds another move. We then declare a real variable x – notice that a and b, which were variables at move 1, are unknown constants at move 2.

```
begin Lestrade execution

      >>> define g x : a * x + b


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


      g : [(x_1 : in Real) => (--- : in
          Real)]


      {move 1}
end Lestrade execution
```

We define the function $g(x) = ax + b$; the move machinery makes the different roles of the letters clear.

```
begin Lestrade execution

      >>> close


   {move 1}

   >>> define h a b : g 1


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


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


   {move 0}
end Lestrade execution
```

We discard move 2, losing the declaration of x but retaining the definition of g. The metaphysical status of g is now the same as that of a or b: it is a variable expression, as it were.

We define a new function $h(a, b) = g(1) = a(1) + b$.