A logical framework, Lestrade, and its parent, Automath

M. Randall Holmes

starting to write, 3/26/2018

1 Brief overview

We report here on two closely connected activities. We developed a logical framework, Lestrade, based on some ideas in the philosophy of mathematics informed by some acquaintance with Automath (as well as other ingredients which should become evident). Lestrade is implemented by a piece of software which we will describe, the Lestrade Type Inspector. We do hope that we may be forgiven for the literary pun connected with our name and the name of the system.

In the course of describing Lestrade and its associated software, we revisited Automath. This paper contains a brief description of Automath, including an account of a set theoretical semantics for Automath which implements oddities of its type system. We also reimplemented Automath ourselves, and will describe some additional insights which this work afforded us.

We are indebted to the original Automath workers, whose work is mostly collected in the yellow Automath book [?], and to Freek Wiedijk, who is the author of the other modern Automath implementation ([?],[?]) and who raises a number of interesting questions about Automath in connection with his discussion of his implementation.

For each of the systems discussed (Lestrade and Automath) we provide a section on semantics (what the notations of the theory mean, that is, what mathematical universe it is talking about) and a section on pragmatics (how the system is used, that is, what it is like to talk to it).

2 Lestrade: semantics

Our intention in designing Lestrade was to create a system in which the user interacted with significant mathematical concepts as concretely, finitely given things, as far as possible. Infinities should as far as possible be potential rather than actual. Part of our intention was that mathematical proofs should be viewed as a species of mathematical object (we had the Curry-Howard isomorphism in mind). This means that the notion of *function* is central (it could be viewed as *the* primitive notion of Lestrade). This means that we do not want to view a function as a completely given infinite table of values, as is the usual set theoretical view: this would defeat our attempt to avoid reliance on actual infinities. Our guiding metaphor in dealing with functions is to view them as immediately given gadgets which will give output of a specified type when input(s) of specified type(s) are given; for functions which are defined rather than primitive (a function such as $f(x, y) = x^2 + y^2$) we have the description of a function by a formula or rule as in calculus as a model: we are talking about functions in intension rather than functions in extension.

The things we talk about in Lestrade are partitioned into two species, entities and Functions. A reason that Function is capitalized is that among the entities we may have some functions, indicated as also being entities by the lower-case letter. Each species is further partitioned into *sorts*. The word *type* is reserved for a certain kind of sort. We enumerate the sorts of entity. There is a primitive sort `prop` of propositions. For each p of sort `prop` there is a sort (`that` p) inhabited by "proofs of p" (we prefer to say "evidence for p" as use of *proof* here appears to presume a constructive view, which we do not necessarily take). There is a primitive sort *type* of "types of mathematical object". We prefer to call these "type labels", because we naturally wish to resist viewing a type as a completely given collection of objects, instead viewing it as a feature encountered in each object of the type. With each τ of sort `type`, we associate a sort (`in` τ) inhabited by the objects of type τ: this is another reason to call τ a type *label*, as it is not itself the type! If one wishes to have mathematical objects "without type", an additional sort `obj` is provided for these. An implementation of ZFC in Lestrade might have the sets as of type `obj`: the implementation would not be unsorted, however, as there would be plenty of other sorts inhabited by propositions and proofs of propositions. We have enumerated all the sorts of entity in Lestrade.

Each Function sort is of the form

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

where the x_i's are variables bound in this notation, each τ_i is a notation for an entity or Function sort, and τ is an entity sort. No x_i may appear in τ_j for $j \leq i$. The variable x_i may appear in τ_j for $j > i$ or in τ: these are dependently sorted Functions. Note that entity sorts `that` p or `in` τ with p or τ complicated terms may quite naturally contain variables.

That completes the account of Lestrade sorts. We now give a compact account of Lestrade terms (of a theoretical flavor not quite the same as the term language of the software) and the computation of their sorts. An entity term is either atomic (with an entity sort presumably given by a declaration) or is of the form $f(t_1, \ldots, t_n)$, where f is an atomic term representing a Function of sort

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

(the coincidence of arities is required), each t_i is an entity or Function term, and t_1 is required to be a term of sort τ_1 (a necessary condition for this to be well-sorted). If $n = 1$ the sort of $f(t_1)$ is $\tau[t_1/x_1]$. If $n > 1$, the sort of $f(t_1, t_2 \ldots, t_n)$ is the same as the sort of $f^*(t_2, \ldots, t_n)$ where f^* is of sort

$$((x_2, \tau_1[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1]) \Rightarrow \tau[t_1/x_1])$$

(and the further requirements for this to be well-sorted must hold).

Lestrade Function terms are either atomic (declared with some Function sort) or of the form

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

(with sort

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

where T is an entity term of sort τ (computed under the assumption that the type of each x_i is τ_i). If the letter f denotes the Function

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

the term $f(t_1, \ldots, t_n)$ will denote $T[t_1/x_1]$ if $n = 1$ and otherwise will have the same denotation as $f^*(t_2, \ldots, t_n)$, where f^* denotes

$$((x_2, \tau_1[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1]) \Rightarrow T[t_1/x_1], \tau[t_1/x_1])$$

We put matters in this seemingly indirect way because non-atomic Function terms do not appear in applied position in Lestrade notation: they appear only as top-level notation or as arguments. The definition just given of the denotation of $f(t_1, \ldots, t_n)$ if f denotes

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

is also our definition of $f(t_1, \ldots, t_n)[((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow T, \tau)/f]$, a clause of the definition of substitution. We do not comment further on the formal definiton of substitution except to observe that the binding of $x_1, \ldots, x_n$ in the notations $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (T,)\tau)$ for Function sorts and Functions has the usual effects on substitution.

Note that entity sort terms are `prop`, `type`, `that` p for each term p of sort `prop`, `in` τ for each term τ of sort `type`, and `obj`.

3 Lestrade: pragmatics

Everything about Lestrade is dictated by the desire to be able to define a Function in the parametric form exemplified by $f(x,y) = x^2 + y^2$, with proper attention to all sorts involved.

Thus, the basic commands of Lestrade are declarations of variables (that is parameters for Function definitions) as being of given sorts, declarations of primitive notions (axioms and undefined operations) taking parameters of given input sorts to a given output sort, and finally definitions of Functions in the target format, defining a Function taking parameters of given sorts to a given term value (whose sort is computed by Lestrade rather than supplied by the user).

Things are ramified by a scheme of relative variation. The declarations in a Lestrade theory at any particular point are organized into a sequence of lists called "moves", of which there are always at least two, the first two being move 0 and move 1, and the last two being move i and move $i+1$ (i being an important parameter of the Lestrade state). The declarations in each move are ordered. Move i is referred to as "the last move". Move $i+1$ is referred to as the "next move". Terms declared at the next move should be thought of as variable (parameters of Function definitions live there); terms declared at earlier moves are (for the moment) constant.

3.1 The basic functionality of Lestrade

We describe the basic functionality of Lestrade commands; some of these have additional options which we may discuss later. The command `open` creates a new next move, empty of declarations: the parameter i is incremented, so that what was formerly move $i+1$ is now move i. The command `close` discards the next move and decrements i: all declarations in the former move $i+1$ are discarded and the former move i becomes move $i+1$. If $i = 0$, the `close` command cannot be issued. The `clearcurrent` command clears all declarations from move $i+1$ and does not change the parameter i: this is the only way to clear declarations from move 1. These are the basic commands managing the move system.

The command `declare` x τ, where x is a fresh identifier and τ is an entity sort term, introduces a new declaration of the identifier x as a variable of sort τ, which is introduced at the last position in the order on move $i+1$, the next move.

The command `construct` x τ, where x is a fresh identifier and τ is an entity sort term, introduces a new declaration of the identifier x as a primitive constant of sort τ, which is introduced at the last position in the order on move i, the last move. A constant declared in move $i > 0$ might become a variable after the `close` command is issued: only a constant declared in move 0 is unconditionally a constant.

The command `construct` f $x_1, \ldots, x_n$ (:) τ (colon optional), where f is a fresh identifier and the x_i's are previously declared (not defined) in world $i+1$

in the order in which they are given in the argument list, each x_i declared with type τ_i, and each variable appearing in the each τ_i or τ appearing in the list, and τ is an entity sort term, declares f as a primitive Function constant of type

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

the declaration being recorded at the end of move i. The basic Lestrade functionality does not provide an explicit way to declare a variable of Function type, but note that a Function declared as a primitive using the `construct` command in move i becomes a Function variable if the `close` command is issued later from that move (being not defined and declared in world $i+1$ after the `close` command decrements i) .

The command `define` x T, where x is a fresh identifier and T is an entity term containing no variables, introduces the definition of x as T (recorded by Lestrade as if x were declared as the nullary Function $(() \Rightarrow T, \tau)$, where τ is the sort of T computed by Lestrade, though x as a term always is understood by the system as simply meaning T). This declaration is recorded at the end of move i (the last move). The command `define` f $x_1, \ldots, x_n$: T (colon obligatory), where f is a fresh identifier and the x_i's are previously declared (not defined) in world $i+1$ in the order in which they are given in the argument list, each x_i declared with type τ_i, and each variable appearing in each τ_i or in T or in the type of T appearing in the list, and T is an entity term, declares f as a defined Function

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

of type $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau)$, the declaration being recorded at the end of move i. The basic Lestrade functionality does not provide an explicit way to declare a variable of Function type, but note that a Function declared as a primitive using the `construct` command in move i becomes a variable if the `close` command is issued later from that move (being not defined and declared in world $i+1$ after the `close` command decrements i). Expressions involving Functions f declared at move $i+1$ may be referred to as "variable expressions" (f by itself as an argument has a similar character); f is not eligible to be a variable because it is defined, but it has the same evanescent character as a variable parameter, as we will see in discussion of what Lestrade has to do with variable expressions when declarations are recorded.

When sort checking a term, Lestrade will carry out definitional expansions to determine whether required equations between sorts hold.

The basic functionality of Lestrade that we describe here supports a limited palette of terms that the user can enter. The user may enter atomic entity terms, entity terms of the form $f(t_1, \ldots, t_n)$, *atomic* Function terms (such as those introduced by the `construct` and `define` commands) and entity sort terms. The user does not need to enter terms with bound variables (Function sort or Function terms), though it proves convenient to add this capability later. It is interesting when thinking about what bound variables do for us to notice that the full logical power of Lestrade is in principle available without the user entering terms with bound variables.

It must further be noted that when a declaration is recorded in world i, any appearances of identifiers defined in world $i+1$ ("variable expressions") in the types and values recorded must be expanded out, since execution of the `close` command would discard the information needed to interpret them. *Variables* declared in world $i+1$ appear as parameters of the Function being declared, and their type information is reported as part of its type. Occurrences of defined entity terms or of defined Function terms in applied position are expanded out in the obvious way (formally described above). Occurrences of Function terms in argument position are replaced by the formal notation for the Function given in the previous section, that is, by a term containing bound variables. So, while the user never has to write such terms under the basic functionality of Lestrade, they do occur in dialogue with Lestrade. This limits the still interesting sense in which the basic functionality of Lestrade is bound-variable-free.

We note that in the term language of the Lestrade Type Inspector, there are some refinements of our notation. The parentheses and commas in $f(t_1, \ldots, t_n)$ are often but not always optional. Mixfix notation $t_1 f t_2, \ldots, t_n$ is supported; commas may be obligatory to prevent Function terms appearing as arguments from being confused with Function terms appearing as mixfixes. Lestrade will always display binary Functions with first argument an entity as infix, and otherwise will not display mixfix notation; the display notation uses as many parentheses as it can. It is worth noting that if a first argument in an argument list is enclosed in parentheses, the entire argument list must be as well.

3.2 A brief example

We present some declarations from algebra, which might give the reader some insights into what "moves" are for.

```
construct Number type

construct 1 in Number

declare a in Number

declare b in Number

construct + a b in Number

define 2:  1+1

construct * a b in Number

open

declare x in Number
```

```
define f x : (a * x) + (b * x) +1

close

define Test a b: f(2)
```

3.3 An enhancement: rewriting

Lestrade supports rewriting as part of its logic.

The command `rewritec` *id* $x_1, \ldots, x_n$, `source`, `target` where *id* is a fresh identifier, $x_1, \ldots, x_n$ are an argument list as in the `construct` or `define` command, and `source` and `target` are terms of the same sort, with every variable contained in `source` or its type appearing in the argument list and every variable appearing in `target` or its type appearing in `source` or its type, will declare *id* as a Function taking arguments $x_1, \ldots, x_n, P, y$ where P is a fresh variable of a type the reader can deduce from the type of y and y is a fresh variable of type `that` $P($`source`$)$ to output of type `that` $P($`target`$)$. This is in effect a proof that `source` is equal to `target` for all values of the x_i's. In addition, a rewrite rule is recorded, so that terms of the form `source` will always be rewritten to the form `target` whenever they are encountered (as long as *id* remains in scope). There are subtleties in the exact forms that `source` and `target` can take and in the exact conditions under which rewrites are applied which enforce confluence. Lestrade will use rewrite rules to expand defined terms and to justify equations of types. When a term is defined, Lestrade will actually apply rewrite rules to its displayed form; rewrite rules will not be visibly applied to types, but Lestrade will use rewriting as well as definitional expansion in attempting to show equality of sort terms during sort checking. This has been used for example in a partial simulation of the type checking algorithm of homotopy type theory.

The variant command `rewrited` *id* $x_1, \ldots, x_n$, `source`, `target` differs in requiring that *id* be an identifier already declared as a Function taking arguments $x_1, \ldots, x_n, P, y$ where P is a fresh variable of a type the reader can deduce from the type of y and y is a fresh variable of type `that` $P($`source`$)$ to output of type `that` $P($`target`$)$. In this way, theorems proved within a Lestrade theory can be used to extend the type checking algorithm with new rewrite rules.

The use of rewrite rules to expand the displayed forms of defined terms gives Lestrade some of the character of a programming language: we have for example written a Lestrade book in which the algorithms for addition and multiplication of binary numerals and the recursive algorithm for computation of Fibonacci numbers are "proved" as a system of rewrite rules from basic algebra axioms, and quite large Fibonacci numbers can be computed as a side effect of defining them (rewrites being properly memoized).

3.4 Another enhancement: implicit arguments

The basic declarations for the propositional connective $\wedge$ and the rule of conjunction might look like this in Lestrade:

```
declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


construct & p q prop

>> &: [(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}


construct Conj p q 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}
```

An odd point here is that wherever the rule of conjunction is used, one needs (if one only has the basic functionality) to use four arguments, two of which, p and q, can be deduced from the types of the others and which might be quite complicated sentences in an extended example. A further feature of Lestrade permits the declaration of Functions (primitive or defined) with implicit arguments. In this style, the last declaration would take the shape

```
construct Conj2 pp qq : that p & q

>> Conj2: [(.p_1:prop),(pp_1:that .p_1),(.q_1:
>>        prop),(qq_1:that .q_1) => (---:that
>>        (.p_1 & .q_1))]
>>   {move 0}
```

The declaration functions of Lestrade identify the arguments p and q which are missing [one can note that the automatically generated order of arguments is not exactly as above], and in any particular instance of application of `Conj2`, matching will be used to determine the appropriate values of p and q. Lestrade can use matching to deduce (or sometimes guess) Function arguments as well as entity arguments (it has some higher order matching capability). Such deductions of implicit arguments can sometimes fail, and it is useful to declare versions of Functions with all arguments explicitly given for such situation (with explicitly given arguments, sort checking will always succeed if it is possible). Implicit argument deduction using higher order matching is very useful for bearable implementation of quantifier and equality rules, for example.

3.5 A final enhancement: user-entered terms with bound variables

It is theoretically interesting that users never need to write Function or Function sort terms with variable binding, but it is quite useful to allow this. Function sorts can be used in commands `declare` f τ to declare Function variables. The form of a Function sort term in the term language of the Lestrade Type Inspector is `[x1,...,xn => tau]`, where the `xi`'s are variables declared at the next move and `tau` is an entity term. It is important to note that these bound occurrences of `x1,...,xn` do *not* have the same reference as the variables of the same shape declared in the next move: what they share is their sorts (with the subtlety that occurrences of x_i for $i < j$ in the type of a bound x_j refer to the bound x_i preceding it, not to the ambient x_i in the last move). Observe that if this declaration were avoided, the variables appearing bound in the sort would actually be declared in a further move not actually opened: these bound variables are in effect clones of variables in the current "next move" standing in for variables notionally declared in a further move which we have avoided actually opening explicitly.

Two kinds of complex Function terms are supported in addition to the atomic Function terms provided in the basic functionality. If f has type

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

and $m < n$, the term $f(t_1, \ldots, t_m)$ represents

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

a curried term. In the Type Inspector's term language, it is worth noting that the parentheses enclosing the argument list of a curried term are mandatory (so that the parser doesn't attempt to absorb the missing arguments from any following material). A more general form of complex Function term is `[x1,...,xn => T]`, which represents $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow T, \tau)$, where each x_i is a variable declared at the next move (with the subtlety noted above that any reference to x_j with $j < i$ in its type is a reference to the bound x_j appearing in this term, not to the ambient x_j in the next move), and τ is the sort computed by Lestrade for the entity term T. Again, it is interesting to note that the bound variables are actually clones of the variables x_i in the next move which would have to be declared in a further move if we did not have the device of user-entered variable binding terms.

3.6 Another enhancement briefly noted: saved moves

In the basic functionality, when a move is closed or cleared, all declarations in that move are simply discarded. It is easy to imagine occasions on which it might be useful to reopen such declarations: Lestrade provides the ability to save moves, so that a tree structure of saved moves with explicit names may appear in addition to the anonymous linear sequence of numbered moves in which most work is usually done.

4 Automath: semantics

5 Automath: pragmatics