# The Lestrade Logical Framework and its Software: Computer Assisted Reasoning driven by Philosophy of Mathematics

M. Randall Holmes

March 26, 2018

## 1 Introduction

The purpose of this paper is to introduce a logical framework, which we call Lestrade, and its accompanying software, and to place it in some context in relation to other systems.

The development of Lestrade was driven in the first instance by a set of ideas in the philosophy of mathematics. It is a consequence of the motivating set of ideas that the objects we actually interact with in mathematical work should be concretely given and finite: a natural way to enforce this is to require that our objects be representable on a computer, from which it followed naturally that we developed the framework in parallel with the software.

A further important component was the set of ideas was that mathematical proofs should themselves be among the objects with which the mathematician has direct contact as concrete finite presentations. The particular way in which we understand proofs as objects is via the Curry-Howard isomorphism, which has the consequence that our software has a distinct resemblance to Automath, and for example to Automath's descendant, Coq. This is entirely unaccidental: we regard Lestrade as in a sense a dialect of Automath, and explaining Lestrade will require (or at least is assisted by) a brief account of Automath.

In the course of the preparation of this paper, we had occasion to think about Automath and in fact ended up writing our own implementation of

Automath. We learned a good deal in the course of building our implementation about the exact relationship between Lestrade and Automath.

The notion of mathematical *function* is a primitive (perhaps in a sense *the* primitive) of Lestrade. Our philosophical motivation would be frustrated by an understanding of a function as a possibly infinite table of pairs of arguments and values (particularly since functions play an important role in the Curry-Howard isomorphism). We are more inclined to regard a function intuitively as a (finitely given) gadget which will take any input of appropriate sort that you give it and output something of appropriate sort. But it is not supposed that we are given all these outputs at once as a completed infinite totality. It is worth noting that a *predicate* is for us a species of function, from the sort of object to which it applies to propositions. It might be said that the philosophy of Lestrade is Aristotelean: all infinities are viewed as potential. Yet an account of classical nonconstructive set theory can be given within the Lestrade framework: this is charming to us.

Like Automath, Lestrade presents a view of the universe in which each entity considered has a sort. We use the word sort because Lestrade makes a particular internal use of the alternative word *type* which is often used. A Lestrade type is a particular kind of Lestrade sort. The sort system of Lestrade is similar to the type system of Automath, but we draw more careful distinctions between sorts which have the effect of notably weakening the logic of Lestrade (the weaknesses being removable by declaration of suitable axioms).

As befits a system with a motivation which was in the first instance philosophical rather than practical (though we do not disclaim practical intentions), Lestrade has a strictly minimal set of primitives. The logical framework of Lestrade is weaker than that of Automath (or at least, weaker than the final Automath systems), which allows for a more precise description of mathematical theories. We will explain how the Automath framework (at least in its later forms) makes it difficult to distinguish (for example) between first-order and second-order comprehension axioms in a set theory. Lestrade easily draws these distinctions. The metaphysical assumptions of Lestrade are extremely bare; much which would normally be viewed as built-in features of a logic must be explicitly declared in Lestrade (and so can readily be declared differently, to give different basic logics supported by the Lestrade framework).

## 2 A brief overview of Automath

We briefly review the system Automath as historical and intellectual background for Lestrade. We describe the dialect AUT-QE of Automath in which the one large Automath implementation of a mathematical text was carried out. We will find that Lestrade is more closely related to an earlier dialect of Automath.

### 2.1 Automath as logical framework

The Automath logical framework (like that of Lestrade) has the construction of dependently typed functions as almost its only primitive construction.

There are three species of things considered in the Automath scheme, entities, sorts, and metasorts (this is our own nonce terminology). The type of an entity term is a sort; the type of a sort term is a metasort. Entity or sort terms are either atomic, function application terms $\langle x \rangle\, f$ ($x$ an entity term); this *is* written backwards), construction application terms $f(t_1, \ldots, t_n)$, where $f$ is an atomic entity or sort term and the $t_i$'s are entity or sort terms, or abstraction terms $[x : s]T$, where $x$ is a bound entity variable of sort $s$, and $T$ is a term. Terms representing metasorts are restricted to the special constants `prop` and `type`, and terms $[x : s]T$ where $x$ is a bound entity variable of sort $s$, and $T$ is a metasort term. It should be noted that the metasort $[x : s]$`prop` is a subtype of `prop`, $[x : s]$`type` is a subtype of `type`, and if $T$ is a subtype of $U$, $[x : s]T$ is a subtype of $[x : s]U$. This is the only subtyping in Automath.

The variables $x$ in terms $[x : s]T$ are bound, with the usual effects on substitution.

If $f$ has type (sort or metasort) $[x : s]T$ and $a$ is of sort $s$, then $\langle x \rangle\, f$ is of type $T[a/x]$. The type of $[x : s]T$ is $[x : s]U$. where $U$ is the type of $T$. As one would expect, $\langle a \rangle\, [x : s]T$ evaluates to $T[a/x]$ if $a$ is of sort $s$.

The typing of construction terms requires special attention. Each construction is initially declared as $f(x_1, \ldots, x_n)$, where each variable $x_i$ has type $s_i$ which may not contain variables $x_j$ for $j \geq i$. If $f$ has type $T$, then $f(t_1, \ldots, t_n)$ will be well-typed iff each $t_i$ has the type obtained by replacing all $x_j$'s with $t_j$'s (necessarily for $j < i$) in $s_i$, and the type of $f(t_1, \ldots, t_n)$ will be obtained from $T$ by replacing each $x_j$ with $t_j$ in $T$. The letter $f$ does not denote the function $[x_1 : s_1] \ldots [x_n : s_n]f(x_1, \ldots, x_n)$ as one might suppose: it represents $f(x_1, \ldots, x_n)$ in a context extending the one in which $f$ was declared. Note also that since arguments of constructions may be sorts, this alternative proposal for semantics of $f$ as a function rather than a variable expression might not make sense. The notation $f(t_1, \ldots, t_m)$, for $m < n$, represents $f(x_1, \ldots, x_{n-m}, t_1, \ldots, t_m)$, in any context with $(x_1, \ldots, x_{n-m})$, an initial segment of the context in which $f$ was declared, as an initial segment. In a context which does not share an initial segment with the context in which $f$ was defined, $f$ must have all of its arguments. This is to our mind an oddity of Automath: the basic construction declarations introduce variable expressions rather than functions, and supplying arguments to constructions represents substitution rather than function application. But it should also be noted that some parameters of constructions may be sorts, and the Automath type scheme does not support application of functions to sorts.

## 2.2 Automath in operation

An Automath command line consists of four components: a *context*, which is either a null context (for which we use the notation 00 in our implementation) or an identifier (atomic term), an *identifier*, the object or sort being declared, a *value*, which is a term or one of the special expressions `EB` or `PN`, and a *type*, which is a sort or metasort term which is the type to be assigned to the identifier.

A context $x$ has the same name as a variable $x$ but it actually represents a list $(x_1, \ldots, x_n)$ of variables with $x_n = x$, and the type of each $x_i$ depending only on variables $x_j$ with $j < i$. The reserved context name `00` represents the empty context ().

A line $(x, y, \texttt{EB}, \tau)$ sets $y$ as a variable of type $\tau$ and sets the context $y$ to the list obtained by appending $x$ to the context $x$. The type $\tau$ may not depend on variables not found in the context.

A line $(x, f, \texttt{PN}, \tau)$ declares a primitive construction $f(x_1, \ldots, x_n)$ ($x_n = x$ of course) of type $\tau$. The type $\tau$ may not depend on variables not found in the context.

A line $(x, d, T, \tau)$ declares a defined construction $d(x_1, \ldots, x_n) = T$ (where $T$ is a term of type $\tau$ containing no free variable not in the context $x$). A term $d(t_1, \ldots, t_n)$ types as above, and evaluates to the result of replacing each $x_i$ with $t_i$ in $T$, if it is well-typed. Neither $T$ nor $\tau$ may depend on any variable not found in the context.

When a context is interpreted, it is read as referring to the most recently declared variable (and so context) with that name.

Automath name management is shockingly liberal (as we found in the course of implementing Automath and attempting to check the one large example text). Nothing prevents the reuse of names already defined. One cannot define a construction with the same name as one of the variables in the context in which it is defined; that seems to be the only hard and fast rule. One can bind variables which have the same shape as context variables or constructions in the current environment. One can declare context variables with the names of constructions one has declared (the interpretation as a context variable takes precedence). Where multiple definitions of an identifier exist, one can use the Automath paragraph system (not described here) to clarify which meaning of the identifier one intends.

## 2.3 An example of use of Automath

Some declarations follow (on which we made remarks and our Automath checker also had comments).

```
00  p   EB   prop

% (p:prop)

# declare a proposition variable p

p     q    EB  prop

% (p:prop,q:prop)

# declare a proposition variable q,
# and create a context (p,q)

q     and  PN  prop

%Mon Mar 19 13:21:25 2018

%1. and(p:prop,q:prop) = PN : prop

# declare a primitive operation
# and which takes arguments (p,q)
# and returns a proposition.

q     pp    EB   p

% (p:prop,q:prop,pp:p)

# declare a variable pp representing
# a proof of p (p is a sort, with
# its inhabitants being its proofs.
# The context pp is (p,q,pp)

pp   qq    EB   q

% (p:prop,q:prop,pp:p,qq:q)

# declare a variable qq representing
# a proof of q.
# the context qq is (p,q,pp,qq)

qq   conj  PN  and(p,q)

%Mon Mar 19 13:21:25 2018

%2. conj(p:prop,q:prop,pp:p,qq:q) = PN : and

# declare a primitive operation
# conj which takes arguments
# (p,q,pp,qq) and returns
# a proof of and(p,q)
# [I could have just written
# and instead of and(p,q),
# but that would seem opaque]
# this is the rule of conjunction

q    rr    EB   and(p,q)

% (p:prop,q:prop,rr:and)
```

```
# in the context q=(p,q)
# declare a variable rr of type
# and(p,q) (a proof of p and q).
#the context rr = (p,q,rr)

rr   simp1  PN  p

%Mon Mar 19 13:21:25 2018

%3. simp1(p:prop,q:prop,rr:and) = PN : p

rr   simp2  PN  q

%Mon Mar 19 13:21:25 2018

%4. simp2(p:prop,q:prop,rr:and) = PN : q

# declare primitive notions simpl
# and simp2, both taking
# argument lists (p,q,rr)
# and returning proofs of p
# and q respectively:
# the rules of simplification.

q    implies  PN prop

%Mon Mar 19 13:21:25 2018

%5. implies(p:prop,q:prop) = PN : prop

# declare the primitive
# construction if(p,q).

pp   pthenq EB  implies(p,q)

% (p:prop,q:prop,pp:p,pthenq:implies)
```

```
# add a variable representing
# a proof of if(p,q)

pthenq  mp  PN  q

%Mon Mar 19 13:21:25 2018

%6. mp(p:prop,q:prop,pp:p,pthenq:implies) = PN : q

# This is the rule of modus
# ponens.  You should be able
# to provide your own analysis.

q    ded0   EB   [pp:p]q

% (p:prop,q:prop,ded0:[pp:p]q)

#  we define a variable ded0
# which is of type a function from
# proofs of p to proofs of q.
# The expression [pp:p]q
# represents this sort.

ded0  ded   PN  implies(p,q)

%Mon Mar 19 13:21:25 2018

%7. ded(p:prop,q:prop,ded0:[pp:p]q) = PN : implies

# this is the rule of deduction:
# from the context (p,q,ded0)
# where p and q are propositions
# and ded0 is a function from proofs
# of p to proofs of q, produce
# a proof of if(p,q)
# (again, the sort here could
# have been written just if,
# but that would be
# mysterious).
# Now we prove
# that and(p,q) implies
# and(q,p)

q   hyp1  EB  and(p,q)

% (p:prop,q:prop,hyp1:and)

# introduce a variable hyp1,
# a proof of and(p,q)
# [the context being
# (p,q, hyp1)]

hyp1  line1 simp1(hyp1)  p

%Mon Mar 19 13:21:25 2018

%8. line1(p:prop,q:prop,hyp1:and) = simp1 : p

# present proof of p

hyp1  line2  simp2(hyp1)  q

%Mon Mar 19 13:21:25 2018

%9. line2(p:prop,q:prop,hyp1:and) = simp2 : q

# present proof of q

hyp1  line3   conj(q,p,line2,line1)  and(q,p)

%Mon Mar 19 13:21:25 2018

%10. line3(p:prop,q:prop,hyp1:and) = conj(q,p,line2,line1) : and(q,p)

# present proof of and(q,p)
```

```
q   thm  ded(and(p,q),and(q,p),[hyp1:and(p,q)]line3(hyp1))       implies(and(p,

%Mon Mar 19 13:21:25 2018

%11. thm(p:prop,q:prop) = ded(and,and(q,p),[hyp1:and]line3(hyp1)) : implies(and

# Now present the function which actually
# witnesses this proof.  The environment is
# now just p and q.  the object thm
# is defined as the result of applying
# the primitive construction  ded
# to the function ([hyp1:and(p,q)]line3(hyp1)
# thus obtaining a proof of if(p,q)
# We are too used to our approach in Lestrade
# (or a more conservative older approach in Automath).
# The operation if does not have to be declared as a primitive.  The
# following alternative declarations work (they depend
# on the subtyping).

q  If   [pp:p]q  [pp:p]prop

%Mon Mar 19 13:21:25 2018

%12. If(p:prop,q:prop) = [pp:p]q : [pp:p]prop

# the computed type of [pp:p]q is actually
# [pp:p]prop, but this is regarded as
# a subtype of prop.

pp Pthenq  EB  If(p,q)

% (p:prop,q:prop,pp:p,Pthenq:If)

# introduce a proof of If(p,q)

Pthenq Mp <pp>Pthenq  q

%Mon Mar 19 13:21:25 2018
```

```
%13. Mp(p:prop,q:prop,pp:p,Pthenq:If) = <pp>Pthenq : q

# the environment is
# (p,q,pp,Pthenq):
# apply Pthenq to pp
# and a proof of q results.
# No declaration of ded is needed
# at all:  a function such as ded0 above
# just *is* a proof of If(p,q).

q Ded0 EB [pp:p]q

% (p:prop,q:prop,Ded0:[pp:p]q)

Ded0 Ded Ded0 If(p,q)

%Mon Mar 19 13:21:25 2018

%14. Ded(p:prop,q:prop,Ded0:[pp:p]q) = Ded0 : If

# the Automath lines above express this thought.
# Notice that the type checker needs to be able
# to use definitional expansion (and the bit
# of subtyping that we have) to check that
# types are actually the same.  Equivalence
# of function or function type terms
# [x:s]T will also need to take into account
# equivalence up to renaming of bound
# variables.
quit
```

I wrote a little Automath checker in Python which accepts the dialect I have described. The lines beginning with `%` are the checker's responses.

## 2.4 A semantics for Automath suggested

We briefly describe a semantics for Automath (working in the usual set theory ZFC plus existence of an inaccessible). Our model is composed of three sets $O$ (the objects), $S$ (the sorts), and $M$ (the metasorts) as one might suppose.

Let $\kappa$ be a suitable ordinal. We will see below how large $\kappa$ needs to be.

Each element $s$ of $S$ determines the type extension $\tau(s) = \{(x, s) \in O : x = x\}$, the collection of elements of $O$ with second projection $s$.

The elements of $M$ include the special objects `prop` and `type`. We refer to these as the elements of $M$ with 0 iterated values. Further elements of $M$ are functions with domain $\tau(s)$ for some $s \in S$ and range the set of elements of $M$ with $n$ iterated values, for a fixed $n$; such a function is said to have $n+1$ iterated values. All elements of $M$ are constructed in this way. There is a subtyping relation defined on elements of $M$: `prop` and `type` are subtypes of every element of $M$, and if $f$ and $g$ have the same domain and each element of $f$ is a subtype of the corresponding element of $g$, $f$ is a subtype of $g$. All subtyping relations are determined in this way.

An element of $S$ is either a pair $((\alpha, \beta), x)$ with $\alpha, \beta < \kappa$, with $x$ one of `type`, `prop` [this is a base sort with $|\beta|$ elements], or a pair $(f, g)$ where $f$ is a function from a type $\tau(s)$ into $S$ with the property that $g \in M$ has the same domain as $f$ and $\pi_2(f(y))$ is a subtype of $g(y)$ for every $y \in \tau(s)$. Further, no $g'$ which is a proper supertype of $g$ should satisfy this condition. An element of $O$ is either a pair $(\alpha, ((\beta, \gamma), x))$ where $\beta, \gamma < \kappa$, $\alpha < \gamma$, and $(\beta, x) \in S$ (so $x$ is `prop` or `type`) or a pair $(f, (g, x))$ where $f$ and $g$ are functions with the same domain $\tau(s)$, $g(y) = \pi_2(f(y))$ for every $y \in \tau(s)$, and $(g, x) \in S$.

It should be noted that we are having fun here, making sense of the idea that that the type of a dependently typed function may actually be a function itself, which is implicit in the Automath syntax and contrary to the persistent intuition that a type is a set.

We build a model of this kind in stages. Initially, we are given some elements of $S$ of the form $((\alpha, \beta), x)$ with $\alpha, \beta < \kappa$. We do not have to have all of these.

Each of these types in $S$ is immediately inhabited with suitable elements in $O$.

We proceed through ordinal indexed stages, taking unions at limit stages as usual. The objects we add at each successor stage are new elements of $M$, determined as functions from newly discovered types at the previous stage to previously given elements of $M$, as well as functions from previously given elements of $S$ which have new elements of $M$ in their ranges, new elements of $S$, determined by functions from newly discovered types into $S$, as well as functions from already constructed elements of $S$ with new elements of $S$ in their ranges, and new elements of $O$ with second projections new elements of $S$ and first projections appropriately dependently typed functions.

Observe that if $\kappa$ is inaccessible, this process will terminate, so definitely the Automath logic has models. It seems to me that all that may be needed is $\kappa$ strong limit, stopping the construction at stage $\mathtt{cf}(\kappa)$. It is true that after stage $\mathtt{cf}(\kappa)$, the construction would add new sorts and objects, but these sorts and objects are likely not definable with the syntactical machinery of Automath.

Reasonably tame constructions other than dependently typed function constructions could be thrown in here.

Notice that the postulation of new constructions can introduce more logical power because the types of constructions may involve abstraction over types of sorts as well as types of objects; all we are supporting here is the basic logic of abstraction over object types.

Clearly anything that Automath syntax allows us to represent will be constructed.

# 3 The Lestrade Logical Framework

## 3.1 The primitive sorts and sort constructors

We use the word *entity* for the objects that fall under the sorts of Lestrade. No sort is actually an entity in Lestrade (it is possible to interpret the language of Automath as identifying certain sorts with entities).

Entities fall into two species, objects and Functions. We capitalize "Function" for reasons that should become clearer as we go along (in particular freeing the use of the word "function" for entities of appropriate object sorts). An approximation to our reasons might be that Functions are actually as it were *proper class* functions, not to be viewed as first-class objects of Lestrade theories, although as we shall see Functions have more privileges than proper classes do in set theories which allow them.

### 3.1.1 Sorts of object

Lestrade has the primitive object sorts `prop`, `type`, and `obj`. The intention is that entities of sort `prop` are propositions and entities of sort `type` are types of mathematical object (we have a tendency to call entities of sort `type` "type labels" rather than simply "types", so as to resist the idea that these are (possibly infinite) *collections*; another way of putting this is that we view the type of a mathematical object as a *feature* we can see in each (concretely, finitely) given instance of the type, and so itself concretely and finitely given.) Entities of sort `obj` are the objects of an "unsorted" mathematical theory. The sets of a Lestrade implementation of ZFC might be of sort `obj`. Such an implementation would not really be unsorted though, because its propositions and proofs would be of other sorts.

Lestrade has two object sort constructors, `that` and `in`. For each $p$ of sort `prop`, there is a sort (`that` $p$) which one might regard as inhabited by proofs of $p$, though we in general prefer to refer to objects of sort `that` $p$ as items of *evidence* for $p$. To view evidence for $p$ as restricted to proofs of $p$ is to presume a constructivist viewpoint. For each $\tau$ of sort `type`, there is a sort (`in` $\tau$) inhabited by what we might informally call objects of type $\tau$.

The treatment of the pairs `prop/that` and `type/in` in the Lestrade framework is almost exactly analogous and could be made exactly analogous.

This is a complete account of the sorts of object in the Lestrade framework.

### 3.1.2 Sorts of Function

The notations for Function sorts are of the form

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

The notations $\tau_i$ are object and/or Function sort notations. The notation $\tau$ is an object sort notation. The variables $x_i$ are bound in this notation, with the usual effects of variable binding on notions of substitution. Each variable $x_i$ is of sort $\tau_i$. A variable $x_j$ cannot occur in a sort $\tau_i$ with $i \leq j$. Dependencies of the output sort $\tau$ on the input variables $x_i$ is permitted, as are dependencies of the input sorts $\tau_j$ on input variables $x_i$ with $i < j$.

Notice that this is a full dependent type system, with the restriction that outputs of Functions must be of object types.

### 3.1.3 Object terms: Function application

Object terms are either atomic or function application terms $f(t_1, \ldots, t_n)$. An atomic object term is always assigned an object sort in context. In a function application term $f(t_1, \ldots, t_n)$, the function term $f$ will always be atomic, and will have a sort

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

The identity of the numerical parameter $n$ in the function application term and the sort term is required. The term $t_1$ must have the sort $\tau_1$ for this term to be well-sorted. If $n = 1$, the term $f(x_1)$ will have sort $\tau[t_1/x_1]$. If $n > 1$, there is the further requirement that $f^*(t_2, \ldots, t_n)$ must be well-sorted, where $f^*$ is assigned the sort

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

the sort of $f^*(t_2, \ldots, t_n)$ will be the same as the sort of $f(t_1, \ldots, t_n)$ if it is defined.

### 3.1.4 Function terms: $\lambda$-abstractions and curried terms

Function terms are either atomic, abstraction terms, or curried terms. Each atomic term is assigned a Function sort in context. Function terms which are not atomic appear only as arguments of function application terms.

An abstraction term is of the form

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

in which $T$ is a term of sort $\tau$ (so an object term) and the same constraints on appearances of $x_i$'s in $\tau_j$'s apply as in function sort terms. The variables $x_i$ may freely occur in $T$. The sort of

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

is

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

Abstraction terms do not appear in applied position. However, an atomic term $D$ may be defined as referring to

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

Definitional expansion of $D(t_1, \ldots, t_n)$ will yield $T[t_1/x_1]$ (of sort $\tau[t_1/x_1]$) if $n = 1$, and otherwise will yield the definitional expansion of $D^*(t_1, \ldots t_n)$, where $D^*$ is defined as

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

A curried term has the form $f(t_1, \ldots, t_m)$ and has the same denotation as

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

where $f$ has sort

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

and $m < n$. This is really a notational convenience, not adding essential strength to the framework.

### 3.1.5 Type matching

At crucial points in sort checking, it is necessary to match sorts. Function sorts will match if one can be obtained from the other by renaming of bound variables. Types will match if they can be reduced to the same term by definitional expansion. Further, sorts will match if they can be reduced to the same term by application of rewrite rules, of which we give an account below.

### 3.1.6 Rewriting

All instances of a pattern `pattern` may be rewritten to a target `target` if there is a Function sending any object of sort `that` $P$(`pattern`) to an object of sort `that` $P$(`target`), taking all variables appearing in `pattern` and `target` as parameters, taking $P$ as a parameter, where $P$ does not occur in `pattern` or `target`, and with the side constraint that no variable appearing in `target` fails to appear in `pattern`. It should be noted that this is the sort of Function from which one could construct a proof of `pattern` = `target` (suitably universally closed) in a Lestrade theory with a reasonable notion of equality (and quantification). We implemented the rewrite in this way to avoid having to add equality to the primitives of Lestrade.

Rewrite rules may be introduced by the user, either relying on the prior demonstration of the witnessing Function, or simply declaring the witnessing Function. It is the user's responsibility to ensure that rewrites terminate. This adds essential power to the framework because it is possible to establish sort matching by rewriting. Note that the ability of the Lestrade software to check the sorts of terms may depend on what rewrite rules have been declared by the user.

The execution of rewrite rules is defined in a way which enforces confluence but not termination: an instance of `pattern` cannot be rewritten with the indicated rule if a proper subterm of the instance can be rewritten (in such a way as to actually change it) which corresponds to a subterm of `pattern` other than a variable.

Symmetry between `prop`/`that` and `type`/`in` could be restored by allowing use of `type`/`in` in the declarations of Functions witnessing rewrite rules. We are not so far concerned to do this.

### 3.1.7 Completeness

This is a complete account of the framework. The further commitments of a Lestrade theory are completely determined by the objects and Functions postulated in its declarations, which are teased out using the definitions given here. The reader should notice that this is quite bare.

# 4 The Lestrade Environment and Commands

What Lestrade (or Automath) is like is not revealed by describing the type system. One wants to know how one talks to the system.

## 4.1 The system of moves

The state of the Lestrade environment at any given time is given by a system of declarations of identifiers, each declaration certainly involving a type and possibly involving an indication of how to definitionally expand an occurence fo the identifier applied to appropriate arguments. This is similar to Automath.

The system of declarations is a list of lists. The lists in the master list are called *moves*. The exact significance of the system of moves is best understood by exhibiting how it operates. At any point there are at least two moves, move 0, move 1, ... move $i$ and move $i + 1$. The parameter $i$ (the length of the list of moves minus 2) is important. Move $i$ is referred to as "the last move" and move $i + 1$ is referred to as "the next move". It is useful to think of all objects in move $i + 1$ as permitted to vary arbitrarily (subject to conditions that some of them may have sorts depending on earlier variables in move $i + 1$ and some may be defined in terms of earlier items in move $i + 1$).

## 4.2 The basic Lestrade command set

We describe a basic set of commands for Lestrade. We will describe extensions of this functionality as we go on.

One may at any time open a new move by issuing the `open` command. This has the effect of incrementing $i$, creating a new empty move $i + 1$, and in effect viewing all previously declared items as given constants for the moment. One may close the current move by issuing the `close` command (as long as $i > 0$). This causes $i$ to be decremented and all declarations in the former move $i + 1$ to be discarded; objects in the former move $i$ are now viewed again as freely varying. One may clear all declarations in move $i + 1$ without decrementing $i$ by issuing the `clearcurrent` command (this is useful as otherwise one could not remove declarations from move 1).

One may declare a variable of an object sort. The command

```
declare ident tau
```

declares the new identifier `ident` as having object sort `tau`. This declaration will be placed last in the move $i+1$ declaration list.

One may declare a primitive construction with a list of variable arguments taken from move $i+1$, given in the order in which they were declared, and a sort term indicating the object sort of the output. The requirement that the variables appear in order of declaration ensures that any dependencies of their types are appropriate, and is much easier to verify than the condition that the types of variables appearing earlier in the argument list cannot depend on variables appearing later in the argument list. The command

```
construct f t1 ...  tn tau
```

where $f$ is a new identifier, each argument $t_i$ is a variable of sort $\tau_i$ (which means that it is an identifier declared in move $i+1$, not introduced by the `define` command), and the variables $t_i$ appear in the order in which they were declared, and any variable on which the sort of a $t_i$ depends appears as an earlier $t_j$, introduces a new function $f$ with sort $(t_1 : \tau_1, \ldots, t : \tau_n \Rightarrow \tau)$ and places its declaration at the end of move $i$ (not move $i+1$!). We are not declaring $f$ as a variable expression but as a function, and we are not introducing it as a new variable but as a constant (at least, constant relative to the current move system).

A function introduced by the `construct` command will become a Function variable if the `close` command is issued (and then can be used as an argument to a Function introduced by the `construct` command or the `define` command introduced next).

One may define a construction by the command

```
define f t1 ...  tn :  T
```

in which $T$ is an object term. The conditions on `f t1 ...  tn` are exactly as for the `construct` command, with the additional condition that any variable on which $T$ depends must appear in the argument list. If the type of $T$ is $\tau$, this introduces the identifier $f$ as representing the Function

$$(t_1 : \tau_1, \ldots, t_n : \tau_n \Rightarrow T : \tau).$$

Any appearance of a defined identifier $f$ declared in move $i+1$ in the type of a Function being declared in move $i$ must be definitionally expanded: we cannot have the declarations in move $i$ depending on the declarations in move $i+1$, which can be discarded at any time using the `close` command.

Where $f$ is applied to arguments, definitional expansion will be carried out as described above. Where $f$ appears as an argument, it will be replaced by anonymous notation parallel in structure to the notation we use in the paper. Curried Function arguments will be expanded by first converting to a $\lambda$-term as indicated above then expanding out the defined operator.

We should note that both the `construct` and `define` commands can appear with empty argument lists. The effect of `construct x tau` is to declare an atomic identifier $x$ of type $\tau$ in move $i$. This is the only way to declare an atomic identifier of sort $\tau$ in move 0 (a genuine primitive constant). The effect of `define x T` is to introduce an atomic identifier $x$ in move $i$ which definitionally expands to $T$. Its declaration takes the form $(T : \tau)$ as if it were a 0-ary function, but it does not behave this way. Definied constants can be introduced in move 0 in this way.

The basic command set has an interesting feature. The user does not at any time have to enter a term involving bound variables (a representation of the notation for a Function sort or abstraction term). Automath has a subset Pal which makes no use of abstraction terms `[x:s]T`: the logical strength of Pal is less than that of Automath. We *have* added the ability to have the user enter Function sort and abstraction terms: this adds convenience but not logical power to the basic Lestrade functionality.

## 4.3 The user language

The Lestrade input language is not the same as the formalized language of the Lestrade framework as presented above.

A Lestrade object term is either atomic (a declared or defined atomic identifier introduced by one of the commands above) or an application term `f t1 ... tn`, where `f` is an atomic identifier representing an $n$-ary Function and the $t_i$'s are object and Function terms of appapriate sorts. Parentheses may enclose the argument list and arguments may be separated by commas. This term may also be written in the form `t1 f t2 ... tn` if $n$ is at least 2. Because this mixfix notation is allowed, it is necessary to follow an identifier representing a Function with a comma in an argument list, and sometimes to precede it with a comma as well, to avoid unintended mixfix readings (enclosing a Function identifier argument in parentheses also works). If the first item in an argument list is enclosed in parentheses, the entire argument list must be enclosed in parentheses as well. Lestrade will always display Functions with arity 2 applied to arguments in infix form, as long as the first argument is not of Function sort. Lestrade will never automatically display infix forms with more than two arguments. One is always safe if one uses all parentheses and commas, which is what Lestrade mostly does in its display.

A Lestrade Function term is either atomic (declared or defined using commands above) or curried (an application term with too few arguments, with the qualification that the argument list *must* be enclosed in parentheses), or is a variable binding term. The form of these is

```
[t1, t2,..., tn => T],
```

where commas are required between arguments if there is more than one argument. Unlike the Lestrade display notation or our formal notation above, this does not include indications of sort for the bound variables $t_i$: the types of these variables are read from move $i+1$ (with the qualification that these really are dummy variables, not the same as the $t_i$'s exhibited in move $i+1$; the types of later $t_i$'s are understood to depend on earlier dummy $t_i$'s, not on the ambient $t_i$'s in move $i+1$, and of course $T$ and its type refer only to the dummies). Curried terms and variable-binding terms may be used only as arguments in function application terms. The variable binding terms are not part of the basic Lestrade functionality, and use of the variable-binding terms can always be avoided by defining an identifier with the desired meaning in an new `open...close` block prior to the place where the variable-binding terms is to be used. It is similarly possible to avoid using the curried terms, which in fact abbreviate variable-binding terms. But both are very useful in practice.

A Lestrade object sort term is either `prop`, `type`, `obj`, `that p` where $p$ is a term of sort `prop` or `that tau` where $\tau$ is a term of sort `type`.

The basic Lestrade functionality admits no user-entered Function sort terms at all, but an extension permits notation

```
[t1, t2,..., tn => tau]
```

for these, where the term `tau` is an object sort term and the typing of the variables is handled just as in variable binding Function terms. The only use of these terms is as arguments to the `declare` command, permitting definition of Function variables without having to open and close a new move.

## 4.4 Implicit arguments

Functions defined using the basic Lestrade functionality frequently have arguments which are annoying because apparently redundant. For example, the logical operation of conjunction and rule of conjunction in propositional logic might be introduced as follows:

```
declare p prop

declare q prop

construct And p q: prop

declare pp that p

declare qq that q

construct Conj p q pp qq: that p And q

construct Conj p q pp qq
```

Now every occurrence of the operator `Conj` will be attended by the arguments `p` and `q`, whose identity can be read off the types of the arguments `pp` and `qq`, respectively. In fact, Lestrade allows the declaration

```
construct Conj pp qq: that And p q
```

This will declare a function of exactly the same type as the last line of the previous block, with the qualification that its first two arguments are understood to be implicit. The user will always write a `Conj` term with two arguments, and the other two will be deduced by the implicit argument feature, which attempts to fill in the missing arguments by a matching process on types of the explicitly given arguments.

The implicit argument feature is rather subtle: it will attempt to deduce implicit arguments which are Functions as well as ones which are objects. In the case of Functions, it will sometimes guess an appropriate Function which may fail to be the correct one. Users will sometimes want to declare versions of operations with and without implicit arguments in cases where subtleties are involved in determining them.

The added convenience is considerable. In the example of the rule of conjunction, one need only consider examples where the propositions whose conjunction is to be proved are themselves quite complicated. In rules of substitution or quantification, the implicit argument feature can often deduce the predicate involved in the quantification or substitution, so that it does not have to be laboriously entered by the user.

## 4.5 Rewriting commands

The command

```
rewritec Id x1 ... xn pattern target
```

with each $x_i$ having type $\tau_i$, will declare a Function Id taking inputs of types $\tau_1, \ldots, \tau_n$, `that` $P$(`pattern`) to output of type `that` $P$(`target`) (where $P$ is a new variable of suitable type; of course the types of the complex terms `pattern` and `target` must agree). As usual in an argument list, no item in the list may have type depending on an item later in the list, and all types required by items later in the list must appear earlier in the list (subject to the fact that implicit arguments for this command are supported). Further, every variable appearing `target` must also appear in `pattern`.

In addition to declaring this Function, the command introduces a rewrite rule which will take instances of `pattern` to instances of `target`. In expanding definitions, rewriting is applied aggressively (with a technical refinement which ensures confluence). Rewriting is not applied overtly to types, but types are identified for purposes of type matching if they can be rewritten to the same form: rewriting is added to definition expansion for the purpose of type matching.

The command

```
rewrited Id x1 ... xn pattern target
```

is similar to the above, except that it requires Id to be a Function which has already been declared of the appropriate type: a theorem already proved can be used to justify a rewrite rule.

It is entirely the responsibility of the user to be sure that their scheme of rewrite rules is terminating: infinite loops are possible! Rewrite rules are declared in moves (the new declarations from either of these commands are posted at the end of move $i$). The order in which rewrite rules are applied is that the most recently declared rewrite rules are applied first (this facilitates redeclaration of rules to modify execution order: the `rewrited` command can be repeated, or applied to the argument of a previous `rewritec` command).

The technical refinement which enforces confluence is that a rewrite rule cannot be applied to a subterm of an instance of the pattern of another rewrite rule unless it is a subterm of the term corresponding to a variable in that pattern.

The rewrite rule system has two interesting effects on Lestrade. It allows much more flexible type checking, if applied with care (we are using it to implement the typing rules of homotopy type theory, which involve judgements of equality between types which are not definitions), and it also in effect makes Lestrade into a programming language. We have written a Lestrade book which is in effect a program for computing Fibonacci numbers in binary notation, whose primitives are just basic axioms for addition and multiplication: the algorithms for addition and multiplication of binary numerals and for the computation of the Fibonacci numbers themselves are proved as theorems and turned into rewrite rules using the `rewrited` command. It is not especially efficient (though we did introduce caching so that it is not *monstrously* inefficient, but it is fun that it can be done.

## 4.6 Elaboration of the system of moves

The system of moves can be elaborated into a tree structure rather than a linear structure. The idea is that one can *save* a move before closing it (or clearing it) which gives it a name. One can then open a saved move again if one is in the right context. Both the `open` and `clearcurrent` commands have a saved move name as an optional parameter: one either opens the move of that name as a new next move, or clears the current next move and opens the saved move of that name. A saved move can of course only be opened if the current sequence of preceding moves agrees with the sequence of preceding moves when it was saved. It is possible that some variable name conflicts may have been created in the interim: in this case, there is a scheme for systematically renaming variables in the saved move which is being opened.

There is a general restriction that a move cannot be saved with its default numeral name, nor can a move be saved if the previous move has its default numeral name, unless the previous move is move 0. This avoids various confusions. This means that in a tree of saved moves, none will have the numeral name associated with their level. Move 0 is never a saved move, and will appear at the root of the tree.

## 5 Comparison of Lestrade and Automath

We have been considerably assisted in evaluating the relationship between Lestrade and Automath by the fact that we decided to write our own Automath checker while we were writing the section reviewing the logic of Automath above in this paper.

As far as the logic goes, Lestrade's logic is in a very definite sense a strict subset of the logic of Automath (at least of the dialect described above, which is basically what is called AUT-QE). To clone a Lestrade book in Automath, declare a type called `Prop` and a type called `Type`, then declare type constructors `That` and `In` so that `That p` is a type if $p$ is of type `Prop` and `In p` is a type if $q$ is of type `Type`. One can also declare a type `Obj`. All Functions which one might declare in Lestrade (and their sorts) then have precise analogues which can be declared in Automath.

Where Lestrade obviously shines is in the treatment of argument lists and the declaration of functions. When one declares a function in Lestrade with a given argument list, one is free to use any variables one wants from the context, as long as they appear in order and the variables on which the type of an argument depends also appear as arguments (and the implicit argument feature relaxes the last requirement). In Automath, one might have to clone each variable, since the exact arguments used must be built up as a context in the way described above, and there is nothing corresponding to the Lestrade implicit argument feature. It should be noted that there is nothing in Lestrade corresponding directly to the ability to abbreviate an Automath term if an initial segment of its arguments coincides with an initial segment of the context. But in Lestrade the system of moves can be used to curtail the number of arguments one must carry around for constructions: Automath contexts can get quite long.

When a function has been declared, the identifier that is declared does not represent that function, but the value of that function in the context in which it is declared with the default values of the arguments taken from that context. To introduce a name for the function, the user must write out an abstraction term representing it. We did add a feature to our Automath checker which automatically declares functions implementing its declared and defined constructions when this is legal: this could be used to advantage when replicating a Lestrade book in Automath.

Nothing in the Automath context corresponds to the subtler articulation of the variable context afforded by the system of moves in Lestrade.

We further remark that we consider the fact that Lestrade is a weaker logical system than Automath to be, or at least to reflect, a virtue of Lestrade. Both Lestrade and Automath are not intended as full foundational theories of mathematics on their own: they are logical frameworks. A logical framework should be weak in logical power, but strong in its means of expression. Automath (or at least the dialect AUT-QE described above: the older version AUT-68 was closer to Lestrade in what we now describe) has an identifiable excess of logical power. Because of the subtyping of metasorts, Automath allows the definition of a universal quantifier over every type. The logical operation of implication is a special case of this. Lestrade requires one to declare the implication operator, and to declare the universal quantifier over each type which one wishes to quantify over. It *is* possible to systematically declare a single universal quantifier which works for all object types, but one might not want to. For example, this makes it hard to write declarations for a true first-order set theory in Automath. If one writes the separation or replacement axiom in a natural way, associating sets with predicates of a type `set`, the fact that Automath immediately gives the power type of `set` and the universal quantifier over it means that one has almost inevitably supported second-order separation and replacement, not just the first-order separation and replacement that one intended. In Lestrade, it is not even the case that there will be a power type of `set` (as an object type), so even if one had a global universal quantifier over object types one might not be committed to second order set theory after making the natural declarations for Zermelo set theory or ZFC.

Our study of the type system of Automath in the process of building our checker allows us to state an important qualification of the point made in the previous paragraph. Predicates of elements of `set` actually have type `[x:set]prop` which is a metasort, not a sort, and this cannot be automatically quantified over. What would force second-order (and even higher-order) set theory given the natural primitives for an Automath implementation of first order set theory would be the presence of a type `bool` and the ability to map propositions to booleans. This would not be likely to happen in a Coq implementation, but if one was working in a classical, nonconstructive theory, as the Automath workers often did, the introduction of the type `bool` would be quite natural, and and soon as it was present one would in effect be able to quantify over all predicates of sets by exploting the analogy with the object type `[x:set]bool`. Any kind of general construction for defining objects by cases om a proposition would have this effect.

## 6 Examples