# A logical framework, Lestrade, and its parent, Automath

M. Randall Holmes

crunched version with citations indicated, 3/30/18

## 1 Brief overview

We report here on a logical framework, Lestrade, based on some ideas in the philosophy of mathematics informed by some acquaintance with Automath. Lestrade is implemented by a piece of software which we will describe, the Lestrade Type Inspector (source at [4], manual at [5]). We hope to be forgiven for the literary pun.

While preparing this report, we revisited Automath. We briefly describe Automath in order to put Lestrade in context. We also reimplemented Automath ourselves and may allude to some insights coming out of this experience.

We are indebted to the original Automath workers, whose work is mostly collected in the yellow Automath book [12], and to Freek Wiedijk, who is the author of the other modern Automath implementation ([14],[15]) and who raises interesting questions about Automath in his comments on 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 an environment in which one can interact with "abstract mathematical objects" as concretely, finitely given things. Infinities should be potential rather than actual (we are, as it were, Aristotelean, though not necessarily constructivist). Mathematical proofs are to be viewed as a species of mathematical object (via the Curry-Howard isomorphism, which was independently discovered by de Bruijn during the development of Automath, so the natural reference to give is [1]). Thus the notion of *function* is central (it could be viewed as *the* primitive notion of Lestrade) 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 frustrate our intention of avoiding actual infinities. Our guiding metaphor in dealing with functions is to view them as directly

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.[1]

The things we talk about in Lestrade are of two species, entities and Functions. [2] 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 $p$" [3]. There is a primitive sort `type` of "types of mathematical object". We actually call these "type labels", deprecating treatment of a type as a collection of objects (with the temptation that it might be viewed as infinite and given all at once), and preferring instead to view a type as a feature encountered in each object of the type. With each $\tau$ of sort `type`, we associate a sort (`in` $\tau$) inhabited by the objects of type $\tau$: this is another reason to call $\tau$ a type *label*, as it is not itself the type! An additional sort `obj` is provided for "untyped" mathematical objects. An implementation of ZFC in Lestrade might have the sets as of type `obj`: the implementation would not be unsorted, though, since other sorts would be 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 $\tau_i$ is a notation for the entity or Function sort of $x_i$, and $\tau$ is an entity sort. No $x_i$ may appear in $\tau_j$ for $j \leq i$. The variable $x_i$ may appear in $\tau_j$ for $j > i$ or in $\tau$: these are dependently sorted Functions. Note that entity sorts `that` $p$ or `in` $\tau$ with $p$ or $\tau$ complicated terms may quite naturally contain variables.

That completes the account of Lestrade sorts. We describe Lestrade terms and the computation of their sorts. An entity term is either atomic (with sort given by a declaration) or is of the form $f(t_1, \ldots, t_n)$, with $f$ an atomic Function term of sort

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

(the coincidence of arities is required), each $t_i$ an entity or Function term, and $t_1$ a term of sort $\tau_1$ (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 $f^*(t_2, \ldots, t_n)$ [and the one is well-sorted iff the other is] where $f^*$ is of sort

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

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

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

[1] We are considering functions in intension rather than in extension.

[2] A reason that Function is capitalized is that among the entities we may have some functions, identified as entities by the use of a lower-case letter.

[3] or "evidence for $p$", since use of *proof* here appears to presuppose a constructive point of view, which we might not take.

(with sort

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

where $T$ is an entity term of sort $\tau$. If $f$ denotes

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

then $f(t_1, \ldots, t_n)$ will denote $T[t_1/x_1]$ if $n = 1$ and otherwise will have the same denotation (or lack of 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 remind the reader 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` $\tau$ for each term $\tau$ 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).

The declarations in a Lestrade theory at any particular point are organized into a sequence of lists called "moves"[4], 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) or "hypothetical"; terms declared at the last move or earlier moves are (for the moment) constant, or "given". Things declared at move 0 are the actual primitive and defined notions of a Lestrade theory.

[4]In earlier versions, we called the moves "worlds": there may be some kind of possible worlds semantics at work.

## 3.1 The basic functionality of Lestrade

We describe the basic functionality of Lestrade commands (some have further extensions). The command `open` creates a new (empty) next move and increments the parameter $i$. The command `close` discards the next move and decrements the parameter $i$: all declarations in the former move $i+1$ are discarded. 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 $i$ (so one can clear declarations from move 1). These are the basic commands managing the move system.

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

The command `construct` $x$ $\tau$, where $x$ is a fresh identifier and $\tau$ is an entity sort term, introduces a new declaration of the identifier $x$ as a primitive constant of sort $\tau$, which is introduced at the last position in the order on move $i$. 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` $f$ $x_1, \ldots, x_n$ (:) $\tau$ (colon optional), where $f$ is a fresh identifier and the $x_j$'s are previously declared (not defined) in world $i+1$ in the order in which they are given in the argument list[5], each $x_j$ declared with type $\tau_j$, and each variable appearing in each $\tau_j$ or in $\tau$ appearing in the list, and $\tau$ is an entity sort term, declares $f$ as a primitive with type

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

recorded at the end of move $i$. The basic Lestrade functionality does not provide a way to declare a variable of Function type, but note that a Function declared using the `construct` command in move $i$ becomes a Function variable if the `close` command is issued from that move (an identifier which is declared at move $i+1$ and was not introduced by the `define` command is a variable) .

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 $T$. 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 $x_j$ declared with type $\tau_j$, and each variable appearing in each $\tau_j$ or in $T$ or in the type of $T$ appearing in the list, and $T$ is an entity term, defines $f$ as the Function

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

of type $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau]$, recorded at the end of move $i$. Expressions $f(t_1, \ldots, t_n)$ involving Functions $f$ declared at move $i+1$ may be referred to as

[5]The rule that the arguments in the list must be given in order of declaration averts the need for quite complex declaration checks.

"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. Such expansions are not displayed, except as required to eliminate notions whose definitions may pass out of scope, so noted below.

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 $f(t_1, \ldots, t_n)$ where the $t_i$'s are enterable terms, *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): we add this capacity later, for convenience rather than because it adds logical power.

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.

It will be seen in examples that the term notation of the Lestrade Type Inspector supports infix notation, and supports a somewhat relaxed attitude toward the inclusion of parentheses and commas in Function application terms.

## 3.2 A brief example

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

The Lestrade Type Inspector supports literate programming, and in fact runs the current document as a script file, executing the bits of Lestrade text that it finds in verbatim blocks, so what the reader sees below is actual dialogue with the Inspector. Indentation based on current length of the move list is a useful feature of the Lestrade display.

```
construct Number type
>> Number: type {move 0}
```

Note the difference between the declaration of a primitive constant 1 of type `Number` and the declaration of variables $a$ and $b$ of this type.

```
construct 1 in Number
>> 1: in Number {move 0}
declare a in Number
>> a: in Number {move 1}
declare b in Number
>> b: in Number {move 1}
```

There follow the definitions of two operations and an entity constant 2. Notice that while the operations are presented in prefix form in their declarations, Lestrade does understand them as infixes in other user entered text and displays them as such.

```
construct + a b in Number
>> +: [(a_1:in Number),(b_1:in Number) => (---:
>>        in Number)]
>>   {move 0}
define 2:  1+1
>> 2: [((1 + 1):in Number)]
>>   {move 0}
construct * a b in Number
>> *: [(a_1:in Number),(b_1:in Number) => (---:
>>        in Number)]
>>   {move 0}
open
     declare x in Number
>>      x: in Number {move 2}
```

Notice in the following declaration of a function $f$ (in move 1, when move 1 is move $i$) the letters $a$ and $b$ are treated as constants (in a way familiar from algebra) although we know with the other side of our mind that $a$ and $b$ are also variables, while the variable $x$ (from move 2) is treated properly as the defining parameter of the function.

```
     define f x : (a * x) + (b * x) +1
>>      f: [(x_1:in Number) => (((a * x_1) +
>>              ((b * x_1) + 1)):in Number)]
>>        {move 1}
     close
```

Notice that the function `Test` which we define just below (defined when $i = 0$ and move 1 is move $i + 1$ instead of move $i$) has the erstwhile (relative) constants $a$ and $b$ as its variable parameters, and notice also that $f(2)$ is a variable expression, since $f$ is defined at move $i + 1$, and is duly expanded out in the displayed form of the body of the definition of `Test`).

```
define Test a b: f(2)
>> Test: [(a_1:in Number),(b_1:in Number) =>
>>        (((a_1 * 2) + ((b_1 * 2) + 1)):in Number)]
>>   {move 0}
```

Extensive examples of theory development in Lestrade, using all the features of Lestrade discussed here and below, are found in [5].

### 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 representing a predicate of entities of the common sort of `source` and `target` 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 (termination is the responsibility of the user). 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 in a partial simulation of the type checking algorithm of homotopy type theory ([6]).

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 $x_1, \ldots, x_n, P, y$ to output of type `that` $P$(`target`) as above. In this way, theorems proved within a Lestrade theory can be used to extend the type checking algorithm with new rewrite rules, and to justify expansions of defined terms.

The use of rewrite rules to expand the displayed forms of defined terms arguably turns Lestrade into a programming language: we have written a Lestrade book ([7]) 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). But it is the ability to type-check using rewriting that makes the rewrite feature an essential extension of the logic of Lestrade.

### 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}
```

When `Conj` 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 terms in practice. 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). Implicit argument deduction using higher order matching is very useful for bearable implementation of quantifier and equality rules, for example.

## 3.5 A further enhancement: user-entered terms with bound variables

Users of Lestrade never strictly need to write Function or Function sort terms and so can avoid ever writing terms with bound variables, but it is quite useful to add this functionality. Commands `declare` $f$ $\tau$ may contain Function sorts $\tau$ and so declare Function variables. The form of a Function sort term entered by a user is `[x1,...,xn => tau]`, where the `xi`'s are variables declared at the next move and `tau` is an entity sort term. Note that the need to write out types

for the inputs is avoided. Note that these bound occurrences of `x1,...,xn` do *not* have the same reference as the variables of the same shape declared in move $i+1$: what they share is their sorts (with the subtlety that occurrences of $x_j$ for $j < k$ in the type of a bound $x_k$ refer to the bound $x_j$ preceding it, not to the ambient $x_j$ in move $i+1$). If direct declaration of a Function variable were avoided, the variables appearing bound in the sort term would actually be declared in a further move or moves: these bound variables are in effect clones of variables in the current "next move" standing in for variables notionally to be declared in a further move or moves never opened 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. We note that in the Type Inspector's term language, 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 $\tau$ 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.

# 4 Automath: semantics

We present the semantics of AUT-QE, the more elaborate of the two implemented dialects of Automath. Lestrade is more closely related to the weaker dialect AUT-68, but this can conveniently be described as a subset of AUT-QE. The terminology we use is often our own. The dialects are described in relation to each other in [13]

Things that we talk about in Automath are conveniently divided into three species, entities, sorts, and metasorts. Each entity has a type which is a sort; each sort has a type which is a metasort.

There are four kinds of terms in the Automath language: there are atomic terms, abstraction terms, application terms, and construction terms.

An atomic term can be of any of the three species. `prop` and `type` are reserved metasort terms.

An abstraction term is of the shape $[x : A]T$, where $x$ is a variable bound in $T$ (but not in $A$) and $A$ is the type of $x$. In classical Automath, we would require $x$ to be an entity variable and thus require $A$ to be a sort. For reasons to be seen, we also allow $x$ to be a sort variable and $A$ thus its type, a metasort. $[x : A]T$ is of the same species (entity, sort, or metasort) as $T$. If $T$ is of type $U$, $[x : A]T$ is of type $[x : A]U$.

An application term is of the shape $\langle a\rangle\, f$ (traditional order reversed). If the type of $a$ is $A$, the domain of the type of $f$ must be $A$. If the type $T$ of $f$ is of the shape $[x : A]U$ or reduces to this shape, it has domain $A$. If $T$ is not of the shape $[x : A]U$ and does not reduce to this shape, then $T$ is a sort, and the type of $T$, a metasort, will be of the shape $[x : A]V$ or will reduce to this shape, for some $A$ which is the domain of $T$ as well as being the domain of the type of $T$. If the type of $f$ is $U$ and the type of $a$ is $A$, the type of $\langle a\rangle\, f$ is $\langle a\rangle\, U$ (usually further reducible). In classical Automath $a$ would be an entity, but we also allow $a$ to be a sort.

A construction term is of the shape $f(t_1, \ldots, t_n)$ where $f$ is an atomic term declared as a construction and $t_1, \ldots, t_n$ are entity and/or sort terms of appropriate types determined by $f$. We regard this term as abbreviating $\langle t_n\rangle \ldots \langle t_1\rangle\, f!$, where $f!$ is an abstraction term determined by the declaration of $f$ as a construction in a way that we explain below. Thus, the explanation of the typing of a construction term is subsumed under the explanations of the other types of terms, in principle. We are able to do this precisely because we allow abstractions over sort variables and applications to sort terms. There are intimations of this approach in the Automath literature. A construction $f$ of arity $n$ may appear with an argument list of length $< n$ or even as an atomic term: missing arguments are supplied in a way to be explained below. Note that a construction $f$ appearing as an atomic term does not represent the associated abstraction $f!$ but a generic value of this abstraction.

To save space, we do not give the formal definition of substitution: we merely note that the usual attention must be paid to the fact that $x$ is bound in $T$ but not in $A$ in the notation $[x : A]T$, and we provide the notation $T[A/x]$ for substitution of the term $A$ for the variable $x$ in the term $T$.

We describe the notion of equivalence of terms. Equivalence is reflexive, symmetric, and transitive. If $A$ is equivalent to $A'$, $T[A/x]$ is equivalent to $T[A'/x]$. $[x : A]U$ is equivalent to $[y : A]U[y/x]$ if $y$ does not occur in $U$. $[x : A][\langle x\rangle\, T$ is equivalent to $T$ if $x$ does not occur in $T$. $\langle a\rangle\, [x : A]T$ is equivalent to $T[a/x]$. We assume of course that all terms discussed are well-typed. No terms are equivalent which are not shown to be equivalent by applications of these rules.

Further, there is a notion of subtyping. A sort may have more than one type (an entity always has a unique type up to equivalence): *the* type of a sort refers to its minimal type in the subtype order (a sort always has a minimal type). Subtyping is reflexive and transitive. $[x : A]$`prop` is a subtype of `prop` when $A$ is a sort (not when $A$ is a metasort). $[x : A]$`type` is a subtype of `type` when $A$ is a

sort (not when $A$ is a metasort). If $M$ is a subtype of $N$, $[x : A]M$ is a subtype of $[x : A]N$, for any $A$. All subtype relations are computed using these rules alone. Further, all metasort terms other than `prop` and `type` are constructed by applying a sequence of abstraction prefixes to `type` or `prop`, or are application terms which reduce to a term of this form.

Well-typedness of Automath terms is computable on the basis of the information given above.

The subtyping in Automath has caused (largely negative) comment. A diabolical variant of the system described so far would extend the subtyping by allowing $[x : A]$`prop` to be a subtype of `prop` and $[x : A]$`type` to be a subtype of `type` when $A$ is a metasort. We do not need this, and in fact deprecate it in a logical framework: we believe that logical frameworks should have a minimum of logical strength and a maximum of expressive power, and would be inclined to use a variant of AUT-QE extended (as we have presented it) with abstraction over sort variables but with all subtyping removed. However, we conjecture that the diabolical variant may be both consistent and interesting as a grand logic, though it looks perfectly mad.

# 5 Automath: pragmatics

The Automath checker reads a sequence of lines from a book. We describe a style of writing lines which is adapted from an older Automath style, but is easier to describe.

Each line has four parts, a *context* part, an *identifier* part (indicating the identifier declared by the line), a *value* part, and a *type* part. We implemented a dialect (not witnessed in the sources, which use various delimiters, but witnessed in our sample code below) in which these parts are simply written on one line separated by whitespace.

Any Automath line is read in a context. The context will be used as the argument list of constructions to be defined in the future. The context is a list of variables each assigned an entity or sort type. Items appearing later in the context may have types depending on items appearing earlier. The context is either empty (in which case we use the name 00 for it) or has the same name as its last entry.

The context part of a line is an identifier (either 00 or a variable name). The most recently declared context with that identifier as its name is used as the context for reading the rest of the line.

If the value part of the line is `EB` (which the yellow Automath book says abbreviates "empty block opener") the effect of the line is to declare a new context obtained by appending the identifier part of the line to the current context, with the type indicated by the type part of the line.

If the value part of the line is `PN` (abbreviating "primitive notion") the effect of the line is to declare a new primitive construction, named by the identifier part of the line, with argument list determined by the current context and output type determined by the type part of the line. Our exact formulation of this is as

follows: if the context is $((x_1, \tau_1), \ldots, (x_n, \tau_n))$ (the $x_i$'s being the variables and the $\tau_i$'s their types) and $T$ is the type part of the line, and $f$ is the identifier part of the line, the effect is to define $f!$ as having type $[x_1 : \tau_1] \ldots [x_n : \tau_n]T$ (recalling that $f(t_1, \ldots, t_n)$ is to be interpreted as $\langle t_n \rangle \ldots \langle t_1 \rangle f!$).

If the value part of the line is a term $T$ and the type part of the line is a term $U$, and $U$ is a type of $T$ (Automath software checks this and rejects the line if this is not the case; *a* type because of the subtyping), and the context and identifier parts are as in the previous paragraph, then the effect of the line is to define $f!$ as $[x_1 : \tau_1] \ldots [x_n : \tau_n]T$, which of course has as a type $[x_1 : \tau_1] \ldots [x_n : \tau_n]U$.

The Automath workers were aware of the possibility of explaining constructions in terms of abstraction and application, as can be seen in [2], [3].

When a constructor name $f$ occurs with no arguments or not enough arguments, the initial segment of the current context of the appropriate length replaces the missing arguments, subject to type checking. In classical Automath, this must coincide with the initial segment of the same length of the context in which $f$ was declared, but we do not require this.

There are some significant variations in notation for Automath lines, especially in Jutting's translation of Landau. We have entirely avoided discussing the Automath paragraph system.

# 6 Comparison of Automath and Lestrade

Any Lestrade theory can be presented in AUT-QE (and in fact in the older dialect AUT-68). Declare a sort `Prop` of propositions. Let this represent the sort `prop` in Lestrade. Declare a primitive construction `That` such that `That`$(p)$ is a sort if $p$ is of type `Prop`. Similarly declare a type `Type` and a type constructor `In` with precisely analogous behavior. All Functions that we might want to declare in our Lestrade theory then have precise analogues with abstractions of sensible Automath types. It should be noted that the method of treatment of `Prop` and the use of a constructor acting as `That` does (in the Automath literature, tbis construction is called `Proofs`) is exactly how propositions and their proofs were treated in the first implemented dialect AUT-68 of Automath, in which `type` was the only metasort (and there was no use of metasort abstractions, and so no subtyping). We do not know whether analogues of `Type` and `In` appear in the Automath sources.

The subtyping of AUT-QE was designed to make certain logical operations definable in the Automath logic, which they were not in AUT-68 or in Lestrade. Our preference is that these operations be primitives, due to a general attitude toward what a logical framework (as opposed to a full-fledged logic) is supposed to do.

The following snippet of AUT-QE code defines implication outright in the logic of Automath, without introducing any primitives. Subtyping plays a crucial role in making this possible: the minimal type of $[pp : p]q$ is $[pp : p]$`prop`, but this is a subtype of `prop`, so the claimed output type of `imp` checks.[6]

[6]The interaction between quantification and the AUT-QE subtyping is remarkable. If $\tau$

```
00  p EB prop
p  q  EB  prop
q imp  [pp:p]q prop
```

We further present modus ponens and the deduction theorem. In AUT-QE both modus ponens and the deduction theorem are simply definable.

```
q pp  EB  p
pp rr  EB imp(p,q)
rr mp  <pp>rr  q
q arg  EB  [pp:p]q
arg Ded  arg imp(p,q)
```

Application of modus ponens is function application of the function witnessing $p \to q$ to the evidence given for $p$. The Deduction Theorem is simply the identity operation, since the content of the deduction theorem argument for $p \to q$ is precisely the sort of thing we define as an object of sort $p \to q$!

In AUT-68, we take the liberty of showing the reader our dialogue with our own Automath checker, up to the development of a proof of $p \to p$[7]:

```
00 Prop PN TYPE
%1. Prop = PN : type
00 p EB Prop
% (p:Prop)
p  Proofs  PN TYPE
%2. Proofs(p:Prop) = PN : type
p q EB Prop
% (p:Prop,q:Prop)
q imp PN Prop
%3. imp(p:Prop,q:Prop) = PN : Prop
q pp EB Proofs(p)
% (p:Prop,q:Prop,pp:Proofs(p))
pp rr EB Proofs(imp(p,q))
% (p:Prop,q:Prop,pp:Proofs(p),rr:Proofs(imp(p,q)))
rr mp PN  Proofs(q)
%4. mp(p:Prop,q:Prop,pp:Proofs(p),rr:Proofs(imp(p,q))) = PN : Proofs(q)
q  arg EB  [pp:Proofs(p)]Proofs(q)
```

is a type and $P$ is of type $[x : \tau]$`prop`, there are two quite different interpretations of $P$. It is obviously a predicate of type $\tau$ objects. Because of the subtyping, it is also of type `prop`: what proposition would it be? A proof of $(\forall x : \tau)P(x)$ will be a function taking each $x$ of type $\tau$ to a proof of $P(x)$, that is, a function of type $[x : \tau]P(x)$, that is, a function of type $P$: the same object which is for AUT-QE the predicate $P$ is also for AUT-QE the proposition $(\forall x : \tau)P(x)$, whence it follows that the function implementing the universal quantifier over type $\tau$ is actually the identity function on $[x : \tau]$`prop`! We don't advocate this (though it does work): in AUT-68 or Lestrade, the universal quantifier must be introduced as a primitive, with somewhat different typing because propositions are a sort not a metasort.

[7]We turned off the omission of arguments feature: this makes the display of Automath terms shorter but often makes them hard to read!

```
% (p:Prop,q:Prop,arg:[pp:Proofs(p)]Proofs(q))
arg Ded PN  Proofs(imp(p,q))
%5. Ded(p:Prop,q:Prop,arg:[pp:Proofs(p)]Proofs(q)) = PN : Proofs(imp(p,q))
p  arg1  [pp:Proofs(p)]pp  [pp:Proofs(p)]Proofs(p)
%6. arg1(p:Prop) = [pp:Proofs(p)]pp : [pp:Proofs(p)]Proofs(p)
p Selfimp Ded(p,p,arg1)  Proofs(imp(p,p))
%7. Selfimp(p:Prop) = Ded(p,p,arg1(p)) : Proofs(imp(p,p))
```

We went a little further and proved $p \to p$, as the reader might decipher. In Lestrade, the following code has the same effect. We have a little fun, writing this in bound variable free style: it could be done a little more succinctly (and more similarly to the AUT-68 code) using user-side variable binding (there are lots of bound variables, as always, in the Lestrade system replies).

```
clearcurrent
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 rr that p->q
>> rr: that (p -> q) {move 1}
construct Mp pp rr that q
>> Mp: [(.p_1:prop),(pp_1:that .p_1),(.q_1:prop),
>>        (rr_1:that (.p_1 -> .q_1)) => (---:that
>>        .q_1)]
>>   {move 0}
open
     declare pp2 that p
>>      pp2: that p {move 2}
     construct ded pp2 that q
>>      ded: [(pp2_1:that p) => (---:that q)]
>>        {move 1}
     define selfimp pp2: pp2
>>      selfimp: [(pp2_1:that p) => (pp2_1:that
>>             p)]
>>        {move 1}
     close
construct Ded ded :  that p->q
>> Ded: [(.p_1:prop),(.q_1:prop),(ded_1:[(pp2_2:
>>             that .p_1) => (---:that .q_1)])
>>        => (---:that (.p_1 -> .q_1))]
```

```
>>   {move 0}
define Selfimp p:  Ded selfimp
>> Selfimp: [(p_1:prop) => (Ded([(pp2_2:that
>>             p_1) => (pp2_2:that p_1)])
>>        :that (p_1 -> p_1))]
>>   {move 0}
```

Notice appearances of implicit arguments in the code above. The objects `Ded`($F$), where $F$ is a Function from proofs of $p$ to proofs of $q$, are examples of (lower case) functions as entities, as would be clearer in an analogous (via Curry-Howard) construction of arrow types and application of functions of such types, which cannot be inserted here due to limitations of space (see [5] for a complete development).

We would have liked to present similar illustrations of quantification, but space forbids. Very large examples of Automath text on more substantial topics, and moderately large examples of such texts in Lestrade, exist. Examples for Lestrade are found in [5] and elsewhere on the same web page; the principal example for Automath is the complete implementation of Landau's little elementary analysis book [11] by Jutting described in [**?**], with Automath text [10] (rescued from oblivion by Wiedijk) available.

We think that the principal merit of Lestrade relative to AUT-68, which has basically the same logic, is that it has a more flexible context mechanism than Automath, and has the ability to declare functions more directly. It does seem natural that $f(x, y) = x^2 + y^2$ should define the function $f$, not the expression $x^2 + y^2$. The Automath context is a uniform long list of dependently typed variables, and one can observe it getting very long indeed in places in the Jutting translation of Landau's little analysis book. The Automath device of allowing elimination of arguments of constructions which can be supplied from an initial segment of the context is quite clever, and does not have a direct analogue in Lestrade. But one can use the move mechanism to reduce the number of arguments that functions require to a manageable (and natural) level.

Automath has a subset called PAL, which does not contain variable binding. Some reasoning can be done in PAL, but it is strictly weaker than Automath. Some point is made about the relationship between the Lestrade approach and the Automath approach by the fact that user-side notation with bound variables is in principle optional, though of course highly convenient, in Lestrade.

A subtle difference between the Automath and Lestrade approaches is that the primitive notion of function in Automath is that of "function with one argument". Functions with many arguments reduce to functions of one argument by currying, and our reduction of constructions in Automath follows this path. In Lestrade, the primitive notion of function is that of "function with $n$ arguments" for each concrete value of $n$. In effect, the application terms and construction terms of Lestrade simply coincide. Neither theory has a native notion of pair or tuple: in both theories, dependent function typing is almost all the typing provided.

The implicit argument mechanism is a solution for Lestrade of a problem

which also exists for Automath (see [**?**], where a never-implemented approach to this problem is considered).. It may be noted that our Automath checker in fact implements not only AUT-68 and AUT-QE, the previously implemented dialects, but AUT-SL, the single line variant, or something functionally equivalent, as well, as a side effect of our permitting the binding of sort variables in abstractions (see [2],[3]): our proposal of admitting abstraction over sorts allows the preparation of terms representing entire books essentially as described in these sources, but we do not think our proposal is exactly the same). Rewriting is another example of a feature which could be added to Automath in basically the same way we added it to Lestrade, and we are planning to extend our Automath implementation in this way.

# References


[1] de Bruijn A2, discovered Curry-Howard (give a page reference)

[2] de Bruijn, B2, single line Automath

[3] de Bruijn on using abstraction and application to handle definitions: B7 p. 336 Also p. 275 in autsl

[4] Holmes, Lestrade source on the web page

[5] Holmes, Lestrade manual on the web page

[6] Holmes, partial development of HOTT typing

[7] Holmes, programming of the Fibonacci numbers

[8] Jutting's discussion of AUT-SYNT

[9] Jutting discussion of his implementation.

[10] Jutting, curated by Wiedijk, the original Automath book

[11] Landau's book

[12] Nederpelt, the Automath book

[13] van Daalen, description of AUT-68 and AUT-QE

[14] Wiedijk, the Automath manual

[15] Wiedijk, the Automath paper