A System of Dependent Types, with an Implementation and a Philosophy

M. Randall Holmes

4/14/2020: Some adjustment of the text to new conditions. Let terms have not been discussed, but command descriptions have been updated: this is somewhat useful as a manual. The embedded text was already in a very old style, taking little advantage (or no advantage) of implicit argument inference. 4/12/2020: the embedded code has been adjusted to work in the reimplementation of Lestrade. The text has not been revised along these lines. The rewrite example is disabled. We remind the reader that the embedded Lestrade code in this file is in a very old style, most of it being older than the implicit argument inference mechanism.
8/15/2019: various updates occurring, since this is the closest approach to a manual, as I work on the large Zermelo implementation. Embedded files in this document are old-fashioned in style, but do work with current Lestrade. The source has been removed, as it was wildly outdated; this is separately available in a readable form with comments.

Contents

2

1 Introduction

This paper describes the development of some thoughts on philosophy of mathematics, a system of dependent type theory (for the general context of dependent type theories, [1] is a nice reference), and a computer implementation of these ideas, in parallel. We call the logical framework that we describe "Lestrade" and the software implementing it "the Lestrade Type Inspector", or, when speaking briefly, also "Lestrade" (to go with the author's "Holmes"): we hope that we may be forgiven.

This version of the paper is being adapted to report on the new implementation of Lestrade developed in early 2020.

The ideas on philosophy of mathematics aim at justification of the current practice of classical mathematics in a manner which may seem more likely to motivate a constructive or even finitist view. That this can be done is in our view interesting. The philosophy is certainly adaptable to a frankly constructive view. All infinities invoked in this scheme are in a suitable sense merely potential: the system is Aristotelean in its fashion.

The system of dependent types is of a familiar sort motivated by the Curry-Howard isomorphism (see [4],[13],[7]) and the implementation is recognizably a variant of the system Automath of de Bruijn (for which the omnibus reference is the useful volume [16]; a survey publication by de Bruijn is [5]; a modern implementation is Freek Wiedijk's [8]). There are various modern relations of Automath (source and examples of use of which are more readily available and on a larger scale) which would be more or less distant cousins of Lestrade, such as Coq ([3]). The recent introduction of the ability to introduce rewrite rules by constructing or exhibiting terms of suitable types to justify them, the rewrite rules then being used to justify type equations and rewrite terms, arguably gives us not only a type checking system but a programming language capable of acting on all types of mathematical object and proof: in particular, it appears to implement the style of programming with rewrite rules described by the author in [10]. Much more would need to be done to make Lestrade an actual programming environment!

The source code for Lestrade will always be available at [12].

2 A model of mathematical activity

We present a model of what we might take a mathematician to be doing.

This will be recapitulated with more concrete detail in the description of the implementation of the syntax and semantics of the Lestrade system in section 3. Most comparisons of Lestrade to its precursor Automath will be deferred to section 3, though some do occur here.

Objects of mathematics in the most general sense we will refer to as *entities*. The entities fall into two main species, *objects* and *constructions*[1]. The species are further subdivided into sorts.

Among the sorts of object that the mathematician might consider are the sort `prop` of propositions, and for each proposition p, a sort (`that` p) which we might say is inhabited by proofs of p. We prefer to call inhabitants of (`that` p) evidence for p rather than proofs of p: to say that when we assume p is true for the sake of argument we in fact assume that it has been proved is to presuppose a constructive philosophy of mathematics. In any event, if there is an inhabitant of (`that` p), it is definitely asserted that p is true: when we call an item in `that` p evidence rather than proof, we are not implying that it is partial evidence.

We are of course thinking of the view that mathematical proofs are themselves mathematical objects, which is embodied in the Curry-Howard isomorphism. In the usual presentation of the Curry-Howard isomorphism, the proof of a conjunction $A \wedge B$ is to be thought of as a pair consisting of a proof of A and a proof of B and the proof of an implication is to be thought of as a construction taking proofs of A to proofs of B.[2] A proof of $\neg A$ is to be thought of as a function taking proofs of A to proofs of $\bot$, the absurd. A proof of a universally quantified sentence $(\forall x \in D : A(x))$ is to be thought of as a function taking elements x of the domain D to proofs of $A(x)$ (note that the type of the output will depend on the input). What we will do will differ from this to some extent: but this is the general idea of our approach.

We are agnostic as to whether the objects the mathematician talks about in his or her propositions and proofs are typed or untyped, so we provide support for both approaches (and we suggest that both parts of this ma-

---

[1]Constructions are referred to as functions in the literature on the new implementation, but we will preserve this usage in this document.

[2]It should be noted that we regard a proof as belonging to the species object rather than the species construction (this is indicated by the use of "function" rather than "construction".

chinery might be useful). We provide a sort `obj` of untyped mathematical objects. We provide a sort `type` inhabited by objects which we call "type labels", and for each τ of sort `type`, a corresponding sort (`in` τ): we refer to objects of this sort as objects of type τ.[3].

It should be noted that though we intend the terminology to suggest that sorts (`that` p) are inhabited by proofs/evidence and sorts (`in` τ) are inhabited by typed mathematical objects, most of the logical framework actually treats `prop / that` on the one hand and `type / in` on the other in exactly the same way.[4] In an actual Lestrade book the general axioms for logic on the one hand and manipulation of types on the other are not likely to be exactly parallel.

These are the sorts of *objects* that we postulate. Of course, if we postulate additional objects of type `prop` or `type` we simultaneously postulate new sorts of object.

In addition, we have *constructions* taking given objects (and constructions) to new objects. In earlier versions, we have used the word "function" instead of "construction", but it is better to use the word function for objects implementing or in a sense packaging constructions.

We introduce entities by two processes: *postulation* and *definition*.

The process of postulation is the local version of the introduction of axioms and primitive notions. We describe the case of postulating a construction. This amounts to postulating a method of acting on a concrete finite list of objects and previously given constructions of given sorts to obtain a new object of a given sort. We declare a sequence of variables, giving a sort for each variable, and give an object sort for the output. A subtlety is that our system of sorts admits dependent sorts: later variables in the sequence of arguments of the construction may have sorts that depend on the values of earlier variables in the sequence, and the sort of the output may depend on the values of the input variables. For example, a proof that for all x of type τ, $\phi(x)$ (where ϕ is a previously given construction taking a variable x

---

[3]There is a tension here between the word "sort" and the word "type": we call the sorts of Lestrade itself "sorts", and the special sorts which it uses to represent types of object considered by the ideal mathematician we call "types": an object of sort (`in` τ) is an object of type τ. Of course we may slip and say type when we mean sort, and we generally say "type" for sorts in Automath itself, as the Automath workers did.

[4]The rewrite feature (not yet found in the new implementation) breaks this symmetry. There is code, currently commented out, in the source which restores the symmetry. The implementation of rewriting in the new implementation may turn out to be symmetric.

of type τ to output of type `prop`, the natural typing for a predicate) can be thought of as a function[5] with first argument x of sort (`in` τ), and output xx of sort (`that` $\phi(x)$). Note that the sort of the output depends on the value of the first argument. Even more subtly, the universal quantifier over type τ can be taken to be a construction with two arguments, the first being a construction ϕ from τ to `prop`, the second being a variable x of sort (`in` τ), and the output being a proof xx of sort (`that` $\phi(x)$). Here we see an argument which is a construction rather than an object. Finally, the universal quantifier over general types can be implemented as a construction taking three arguments, a type τ of sort `type`, a predicate ϕ which is a construction from τ to `prop`, and a variable x of sort (`in` τ), and sending this to output a proof xx of sort (`that` $\phi(x)$). In this example we see that the sort of an input to a construction may depend on the values of earlier inputs.

The process of definition allows us to introduce new entities whose existence follows from our axioms and constructions using our primitive notions: this is the local version of proof of theorems and introduction of defined notions. Again, we describe the case of defining constructions. This is done in an entirely familiar way: introduce a sequence of variables of appropriate types as under the previous heading, in which the sorts of later variables may depend on the values of earlier variables. Write out an expression for an object in terms of these variables which is well-sorted, using constructions already given, and compute its sort. We thereby discover a new construction which from any sequence of objects and constructions with the appropriate input sorts generates an object of the appropriate output sort (of course an object sort, and possibly depending on the values of the inputs).

The next ingredient of the philosophical approach we take is a special attitude toward functions (the mathematical notion of function being manifest in our notion of construction). We do not choose to regard functions as infinite tables determined by applying the function to every possible sequence of inputs of the appropriate sorts. We take the approach that a function is an actual rule or construction (necessarily speaking informally). Where x is a natural number variable, we take seriously the idea that the variable expression $x^2 + 1$ determines a function. We really have "variables" in a suitable sense in our metaphysics, as it were. We are avoiding actual infinite totalities in our metaphysics, so we hardly want a function to be an actual

---

[5]We use the word "function" advisedly, to indicate that such a proof is an object (in which a contruction is packaged) rather than a construction itself.

infinite table of the outputs corresponding to given inputs: we prefer to think of it as a (presumably finite) template into which inputs can be plugged to produce outputs. The function $(\lambda x.x^2 + 1)$ is more like an abstraction from the term $x^2 + 1$ than the infinite table $\{(1,1),(2,4),(3,9),\ldots\}$. This may be enough to suggest why we use the word "construction" instead of "function", and we return to that usage. Our attitude that constructions are in a sense finite objects is borne out by the fact that constructions are never introduced except as abstracted from explicit terms with variables in them (which are finite structures): which is not to say that we identify constructions with pieces of text. We are talking about metaphor here. The fact that constructions are abstracted from expressions, which are certainly finite objects, appears to make it reasonable to suppose that a construction is a finite object. The expression tells us how to compute the value of the construction for any input values of appropriate types which are presented to us: we do not need to know all the values in advance: the infinity of values of the construction is potential, not actual. To sumarize, we take the position that a construction is a finite object abstracted from a finite expression, with slots in it (corresponding to variables in the expression) into which objects of the appropriate sorts can be plugged to give an output of the appropriate sort. The same view applies to the axiomatically given constructions: it is just that in this case we cannot look under the hood and see how the output is computed (and we do *not* support any presumption that arbitrary constructions introduced in the future will be definable in terms of ones previously introduced or share any properties provable of constructions definable in terms of previously introduced constructions by some structural induction). We now propose to take seriously the notion of a variable entity. Our suggestion is that a variable entity is an entity in a possible world to which we have limited access: all we know about the entity is its sort (which may contain information about previously postulated variable entities). One metaphorical way to think of a variable entity is that this is an entity of a given sort which may be given to us in the future. We use this temporal metaphor explicitly in the latest version of Lestrade by calling the possible worlds "moves".[6]

An initial version of this scheme which can be used to present the idea

---

[6]This nomenclature is a recent decision. In earlier material, what we call here "moves" were called "worlds", and what we call the "next move" was called the "current world" and what we call the "last move" was called the "parent world".

concretely is that we have a sequence of moves at any given point in the process indexed by concrete finite natural numbers 0 to $i + 1$ (there are always at least two moves). Move i, which we call the last move (or in some texts the current move), and all lower indexed moves, are inhabited by objects and constructions which we currently regard as given constant things of their various sorts. Move $i + 1$ we view as inhabited by variable entities, or in terms of our metaphor, things of determinate sort which have not been fixed yet: we call it the next move.

We can freely declare variables in move $i + 1$ of any object or construction sort we currently have accessible (remember that if we postulate new objects of sort `prop` or `type`, we have thereby postulated new sorts). The declaration of an object in move $i + 1$ may make more sorts available which depend on this object: these may be used as sorts of further objects declared in move $i + 1$. We regard all declarations as having been made in an order, with any variable having been declared later than any variable on which its sort depends. One should remember that "variables" are always entities at the next move.

We can introduce new primitive constructions: when a sequence of variables has been postulated at move $i + 1$ (some of whose sorts may depend on variables declared earlier) we may postulate a construction (a new construction at move i, the last move) which will take any inputs of the given sorts and give an output of a stated object sort (which may depend on the input variables). We may also declare a constant object in move i of a given fixed object sort (this can be viewed as a special case where the argument list is empty; we are here introducing an object constant rather than an object variable). The typical format of a postulation command is "postulate f which takes arguments $x_1, \ldots, x_n$ (variables declared in move $i + 1$ in the order in which they were declared (which ensures that the sort of an x_i can depend on an x_j only if $j < i$) and is such that $f(x_1, \ldots, x_n)$ is of sort τ (an object sort which may depend on the x_i's)." Any variable on which the sort of an x_i depends must appear as an x_j.

We can introduce new defined constructions: any expression using previously given constructions may be taken to determine a new construction at the last move (move i) taking as arguments a list of variables including all variables actually appearing in the expression (and all variables on which the sorts of variables in the argument list depend, in such a way that sorts of later arguments depend only on earlier arguments). We can introduce objects by definition as well. When we introduce an object by definition, we

assign it a name: so we define a new construction or object symbol. The typical form of a definition is "define $f(x_1, \ldots, x_n)$ as t", where $x_1, \ldots, x_n$ is an argument list of variables just as above and t is an object term. This declares the symbol f as an item at move i whose input sorts are of course the sorts of the x_i's, and whose output sort is the sort of t (which obvously may depend on the x_i's as we expect t itself will so depend).

Finally, we observe that the system of moves is dynamic. At any point, we may fix everything we have postulated so far and open a new move $i + 2$ which becomes the next move, whereupon move $i + 1$ becomes the last move and we increment the parameter i. Or we can close move $i + 1$ (if $i \neq 0$) and return to considering objects at move i as variable relative to objects at moves of lower index (thus in effect decrementing the parameter i).

A subtle point is that it may not be clear that we have given ourselves the ability to declare variables of construction sorts. But we have. One procedure for doing this is to open a new move, declare variables of appropriate types desired as input sorts and declare a primitive construction taking this argument list to an output of the desired output sort, then close that move. The primitive construction declared remains as an entity at the next move, a variable construction of the desired type. A recent update allows one-line declarations of construction variables, since we have introduced terms representing construction sorts; it remains interesting that we can declare construction variables while never actually writing a construction sort term.

We have made the perhaps artificial decision that the output of a construction is never a construction[7]: we are in effect reversing the popular operation of "currying" as far as possible. This has a striking effect on the language of the computer implementation, at least in the current version. There is no need for the user ever to write a representation of a construction other than the atomic name under which it was constructed or defined. [This is not to say that the ability to do this may not be desirable; it has in fact been added and is extremely useful in practice: but it is interesting that it is eliminable]. The user's input language originally contained no variable

---

[7]In fact, we do not feel that this is arbitrary. The idea is that the *objects* are the first class citizens of a Lestrade world, and constructions are only declared or defined schematically in terms of their object values. One way of thinking of this is to think of the constructions as "proper class" functions, though in a framework where proper class constructions are distinguished from set constructions with the same extension. The analogy with sets versus proper classes is not perfect, though, since we do not allow constructions to be outputs of constructions, but we do allow them to be inputs of constructions.

binding constructions, in fact. A construction may always be defined as in calculus when we say $f(x) = x^2 + 1$, the name f then being provided for the construction; it never has to be introduced as $(\lambda x.x^2 + 1)$ (anonymously, as it were) though such notations may be generated in sort signatures by Lestrade when the name under which a construction was introduced passes out of scope due to the move at which it was created being closed, and the user may now enter such λ-notations as arguments to Lestrade constructions. A variation on this is that a term $f(t_1, \ldots, t_m)$ where m is less than the arity n of f will be read as $[(x_{m+1} \ldots x_n) \Rightarrow f(t_1, \ldots, t_m, x_{m+1}, \ldots, x_n)]$ if the types of the t_i's are appropriate. This only works if the argument list is explicitly closed with parentheses. Such a term can only appear as an argument. The user may enter λ-terms as arguments. The syntax is of the exact form $[x_1, \ldots, x_n \Rightarrow T]$, the bound variables x_i being variables declared at the next move, separated by commas, and T being an object term. The identifier `=>` is now reserved. The form displayed by Lestrade for these terms will have the bound variables decorated with a namespace index and will be fully adorned with sorts for the variables. The notation for constructions sorts is similar, taking the shape $[x_1, \ldots, x_n \Rightarrow \tau]$, where the x_i's are variables in the next move and τ is an object sort term.

There is a comment to be made here about the relation between Lestrade and Automath: Automath has a sublanguage PAL in which λ-terms are not used ([16], pp. 79-88), but PAL is strictly weaker than Automath in its logical capabilities: Lestrade has a more complex context mechanism which allows the logical power added to Automath by adding λ-terms to be replicated, but without actually requiring that one use λ-terms; we have now added the ability to write anonymous λ-terms as arguments of Lestrade constructions, but this adds convenience rather than logical power. Similar remarks apply to the introduction of terms with variable binding which represent construction sorts.

The system was forced to have an internal representation of anonymous constructions, however, even before user-entered λ-terms were allowed. Whenever it computes the sort of a construction, it produces a dependent type in which the arguments are bound variables: a typical sort is $[(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow \tau]$: this is an expression with bound variables because the types τ_i may depend on x_j for $j < i$ and τ may depend on any x_i. The sort data recorded for a defined construction has the body t of the definition as a further piece of information (the form is $[(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (t, \tau)]$, where τ is the type of t): this "sort" is in effect a λ-term. Now consider what happens when the

last move contains a defined construction and the next move is closed. Primitive constructions declared in the former last move (which is now the next move) become variable constructions. Defined constructions declared in the former last move remain defined, but they are now as it were defined variable expressions. When a construction or definition is declared in a way which uses one of these defined constructions, the name of the defined construction must be eliminated from the information recorded at the last move, because if we close the current next move the name of the defined construction ceases to be available. Where the defined construction appears in applied position, it is eliminated by expanding the definition in the obvious way. Where it appears as an argument, it can be replaced by its sort, which is in effect a lambda term defining it, in which the only terms denoting objects in the same move are bound variables representing its arguments.

One must note that the implementation of this scheme is inevitably recognizably a variant of the Automath type checker. We are quite happy to acknowledge this. Previous versions differed from the Automath type checker in not having lambda abstraction and construction application as operations directly available to the user: the current version does allow the user to enter λ-terms as arguments in Lestrade expressions, and still differs from Automath in that constructions are never defined except by schematically declaring their values, though one-line declarations of construction variables are now possible, since λ-terms representing construction sorts have been introduced.

If this is taken to be vague, it is made concrete by the actual implementation. We support our claim that this scheme implements all mathematical activity by actually presenting declarations implementing standard foundational systems.

A philosophical point to be made about this scheme is that all infinities involved in it can be viewed as potential. One is never directly considering an actual infinity of objects and constructions at any point. One manipulates entire sorts not by considering all the entities of that sort individually (requiring that the whole sort be given at once) but by the device of generality. One shows not how to deal with *every* object of a potentially infinite sort, but how to deal with *any* object of this sort which may be presented to one. We believe that this defuses any objections to impredicative reasoning. From this standpoint it is clear that *definition* is not a logically trivial move: to define a construction (or object) is to explicitly make some potential entity actual, and since we do not suppose that all potential entities are given to

us at once, this is a nontrivial act.

It might be thought that this approach is more suited to constructive reasoning, and indeed it can implement constructive reasoning. But classical reasoning is also amenable to implementation in this style.

We narrate the introduction of the general universal quantifier operation described in our earlier example.

Initially, we have move 0 and move 1, each containing no declarations.

Declare a variable τ of sort `type` in move 1.

Open move 2: the last move is now move 1 and move 2 is the next move.

Declare a variable x_1 of sort (`in` τ) at move 2.

Postulate a construction ϕ such that $\phi(x_1)$ is of type `prop`. ϕ is introduced as an item at move 1.

Close move 2. ϕ is now a variable construction taking a type τ argument and returning a proposition (an arbitrary predicate of type τ objects).

Declare a variable x of sort (`in` τ).

Potulate a construction $\forall$ such that $\forall(\tau, \phi, x)$ is a proposition. $\forall$ is declared at move 0 (an unequivocally constant entity). This is the correct type for the universal quantifier. You can see in the extended example in section 6 how we further postulate constructions implementing the rules for reasoning with the universal quantifier.

3 Formal description of the syntax and sort checking

There is a lot of sample Lestrade input and output to examine in later sections (and some embedded in this section). This Lestrade text is now rather old and for example does not take much if any advantage of the ability to use λ-terms for construction sorts and construction arguments.

3.1 Lestrade Syntax

books made up of lines: A Lestrade book (a Lestrade text is called a book, following Automath usage) is a sequence of lines.

resolution of lines into tokens: Each line breaks up into tokens.

A token is either

1. a non-null string which is the concatenation of a single upper case letter or null string followed by a possibly null string of lowercase letters followed by a possibly null string of digits, followed by a possibly null string of single quotes. One of the components has to be non-null.

2. a non-null string of special characters taken from

   `~!@#$%^&*-+=/<>|?`

   followed by a possibly null string of digits followed by a possibly null string of single quotes.

3. or a single character taken from

   `,:()[]`

4. If a double quote appears in a Lestrade line, the entire remainder of the line is taken to be a token (without the quote). This is used

by some diagnostic commands not listed here: this item is mainly a warning not to use double quotes in Lestrade lines.[8]

Whitepace is ignored, except that it terminates a token. `A 1` is two tokens, where `A1` is one.

command names: The first token in a line will be the special prompt `>>>` and the second will be a command name, one of

```
declare, postulate, define, rewritep, rewrited, open, close, clearcurrent,
Declare, Posit
  [the first line contains all the commands
  that implement the logical framework;
  Declare and Posit are added in the new implementation;
rewritep and rewrited are not yet available in the new system.]

save, Showmoves [this implements old foropen and forclearcurrent]]

[commenting and term-testing commands.]

goal, test

[commands to do with context saving]

Showdec, Showdecs
  [commands that will cause display of information]

Readtex, Readbook, Readonbook, Readontex

[read a first file, output to a second.  The filenames must be
 legal Lestrade identifiers.]
```

---

[8] `parsetest "<text>` will parse the text as a term and display the term and its type. `parsetest2 "<text>` will parse the text as a sort (object or construction). I have more diagnostic constructions which I will probably eventually implement in the Lestrade interface using this device. I preserve this footnote for the moment, but see the new commands `goal` and `test`.

```
Load

[import declarations from other files already run]

Zeroorone, Strongexpand

[technical toggles modifying use of let clauses]

Clearall

[clear the Lestrade environment]

comment

diagnostic, marginup, margindown, versiondate

[miscellaneous items]
```

reserved tokens, identifiers: The reserved tokens are the comma, colon, parentheses, and brackets, the string => used in the innards of λ-terms, and the words `begin`, `end`, `obj`, `prop`, `type`, `that`, and `in` as well as the line prompt >>> and dummy object terms --- and +++. All other tokens are identifiers (there is no reason that the command names cannot be identifiers). An identifier which appears as the next token after the command `declare`, `postulate`, `define`, [rewritep], `Declare`, or `Posit`. should not have been previously used and will be assigned a sort if the line is well-formed and can successfully be executed. An identifier which appears as the next token after the command `rewrited` should already have been declared. Any other identifier which appears in a line should have been declared already and assigned a sort.

A piece of information we need is that the sort of an identifier will tell us whether it represents an object or a construction, and if it represents a construction the sort will tell us the arity of the construction, a positive integer.

extended identifiers: A scheme of renaming is used to avoid name conflicts: an identifier is postpended with single quotes, repeating as neces-

sary until a new identifier is obtained. These are referred to as extended tokens or extended identifiers.

logical commands: The only lines which contain parsed text after the initial command are those which begin with `declare`, `postulate`, and `define`, and now the new commands [`rewritep` and `rewrited`], `Declare` and `Posit`. The commands `goal` and `test` also have parsed text arguments, but have no effect other than term display.

comment commands: Text to be excuted will begin with `begin Lestrade execution` and end with `end Lestrade execution`. Text not in such blocks will be persistent comments. Text in such blocks not beginning with `>>>` will be transient comments (including Lestrade output in response to commands). A line beginning with `>>>comment` will be a permanent comment in an execution block.

The commands `goal` and `test` might be regarded as comment commands. The `goal` command parses and displays an object sort: this is used to assist readability of arguments. The `test` command parses the following text as an argument and displays the argument and its type. This can be used for commenting, and also for live development of command lines.

display commands: The `Showdec` command may be followed by a single identifier whose declaration will be shown. `open`, `clearcurrent`, and `save` may have an identifier argument. Text following other commands will be ignored.

the Readbook, Readtex commands: `Readbook file1 file2` will read the Lestrade commands in `file1.ltxt` and log to `file2.ltxt`. The filenames need to be valid Lestrade tokens. This command should not appear in `.ltxt` files (or in `.tex` files intended to be handled by Lestrade): it will not work correctly (it is not designed to be nested). These commands clear the Lestrade context then read the files.

the Readonbook, Readontex commands: `Readbook file1 file2` will read the Lestrade commands in `file1.ltxt` and log to `file2.ltxt`. The filenames need to be valid Lestrade tokens. This command should not appear in `.ltxt` files (or in `.tex` files intended to be handled by Lestrade): it will not work correctly (it is not designed to be nested).

These commands do not clear the Lestrade context before reading the files.

remark on pretty printing: Lestrade will pretty print both type and definition information which it outputs and command lines, with indentation which is attentive to information about moves and depth of abstraction. In command lines, the backslash signals continuation of the line on the next line, and the prettyprinter works on command lines by inserting backslashes and suitable indentation.

the load and import commands: The `Load` command will be followed by a token (the name of a Lestrade log file which has been read by the system, it is hoped).[9] The `import` command of the old version has not been implemented, but probably would be nice to have.

syntactical forms of the logical commands: The form of a `declare` line is the keyword `declare`, followed by the undeclared identifier to be declared (which cannot be a reserved or extended token), followed by an object sort term. These classes of strings will be explained. Note that there is **not** a colon before the object sort term as in the following command.

The form of a `postulate` line is the keyword `postulate`, followed by the undeclared identifier to be declared (which cannot be a reserved or extended token) followed by an argument list, followed (optionally) by a colon[10], followed by an object sort term.

The form of a `define` line is the keyword `define`, followed by the undeclared identifier to be declared (which cannot be a reserved or extended token), followed by an argument list, followed by a colon (required), followed by an object term.

The form of a `rewritep` or `rewrited` command is the respective keyword, followed by the identifier to be declared (`rewritep`) [in this case

_____

[9]The double quote token construction can handle file names which are not well formed Lestrade tokens otherwise.

[10]The colon in this and following commands is **not** there to set off a following sort term but to terminate an argument list whose length cannot be predicted: in the postulate command its use is optional; in the define command, where what follows is **not** a sort, its use is mandatory.

not reserved or extended] or used as witnessing proof [in this case already declared] (`rewrited`), followed by an argument list. These commands are not implemented in the new implementation, and may be somewhat different when they are implemented.

The form of a `Declare` or `Posit` command is the keyword followed by a fresh identifier followed by a term of any of the types supported (object, object sort, construction or construction sort). These commands are new in the new implementation, and `declare, postulate, define` are all implemented in terms of these commands.

object sort terms: An object sort term is either `obj`, or `prop`, or `type`, or the keyword `that` followed by an object term, or the keyword `in` followed by an object term.

construction sort terms: A construction sort term consists of an opening bracket followed by a comma-separated list of variables declared in the next move followed by `=>` followed by an object sort term followed by a closing bracket. They can only be used as final arguments of the `declare` command.

object terms: An object term is either an identifier declared to be of object type, or an identifier declared to be a construction of arity n, followed by an argument list of length n, or an object term followed by an identifier declared as a construction of arity $n \geq 2$ followed by an argument list of length $n - 1$ which cannot be enclosed in parentheses unless it is of length 1 (this is an infix or mixfix term). An object term may optionally be enclosed in parentheses. The precedence order assumed is that of the old computer language APL (every infix or mixfix is of the same precedence, except unary prefix operators, which bind more tightly, and everything groups to the right).

The display functions will always use infix form when a construction is of arity 2 and has first argument an object. Mixfix forms other than infix are never displayed. The display functions use lots of parentheses and commas.

The final argument of a prefix term whose argument list is not enclosed in parentheses cannot be a mixfix term: this is a new condition in the new implementation (it was true in the old implementation for prefix terms of arity 1 only). The new implementation requires that

a mixfix argument be preceded by a comma even if it is the last argument; the previous implementation detected this condition and did not require the comma. It should also be noted that in the new implementation a space is needed between a special character identifier and a following numeral, since the new implementation allows numerical subscripts to such terms. Special character identifiers are extended with single quotes as letteral identifiers are (the previous implementation used dollar signs).

The new implementation treats binder lists and the argument lists in commands differently from the old one: commas are never required, though they are allowed.

argument lists: An argument list may be of length 0, in which case it is the null string (null argument lists occur only as the third item in a `postulate` or `define` line: the parser never finds them). An argument list of positive length n may optionally be enclosed in parentheses unless it is of length greater than one and follows a mixfix operator. The n items in it are either object terms, lone identifiers of construction type, curried construction terms or λ-terms; individual items may optionally be separated by commas, which may be necessary to avoid a construction item from being read as a prefix, infix, or mixfix operator, depending on its arity.

A term $f(t_1, \ldots, t_m)$ where m is less than the arity n of f will be read as $[(x_{m+1} \ldots x_n) \Rightarrow f(t_1, \ldots, t_m, x_{m+1}, \ldots, x_n)]$ if the types of the t_i's are appropriate. This only works if the argument list is explicitly closed in parentheses. Such a term can only appear as an argument.

The user may enter λ-terms as arguments. The syntax is of the exact form $[x_1, \ldots, x_n \Rightarrow T]$, the bound variables x_i being variables declared at the next move, separated by commas, and T being an object term. The identifier `=>` is now reserved. The form displayed by Lestrade for these terms will have the bound variables decorated with a namespace index and will be fully adorned with sorts for the variables.

serious warning about the parser: It is important to note that if the first item in an argument list following a prefix operator is enclosed in (or begins with) parentheses, one must also enclose the entire argument list in parentheses, to avoid reading the first item as the entire argument list. This is still true in the new implementation.

It is also important to note that all infix or mixfix operators have the same precedence and group to the right, as in the old language APL. Unary operators bind more tightly than infix or mixfix operators (this is generalized in the new implementation by the requirement that the last argument of a prefix term which does not have its argument list enclosed in parentheses must not be a mixfix term).

3.2 Lestrade Sort Declaration and Checking

All that we have revealed so far is the syntactical requirements for a line. There are semantic requirements as well, of course, handled by the sort checking functions of the prover.

the scheme of moves: At any given point the user has a finite list of sort declarations of identifiers which appear in the order in which they were added to moves 0 to $i + 1$, where i is a parameter maintained by the program. Any identifier declared at any move is accessible to the parser and sort-checker. We call move $i + 1$ the next move and we call move i the last move. Some moves may be assigned names other than the default numerical name derived from their distance from move 0: moves which do not have such names are assigned their default numerical names.

opening a new move: The `open` command causes i to be incremented and a new move $i + 1$ to be opened with no definitions in it. [The command `open` with an argument will open an existing version of move $i + 1$ named by that argument or create a new one.] In the new implementation, name conflicts in a saved move which is reopened are handled differently: the identifier is read as the saved move wants one to read it, but occurrences in earlier moves are still handled correctly; the old implementation renamed such identifiers in the saved and reopened move by extending them. When a move with the default numerical name is opened, the saved version is discarded after opening.

saving the next move: The `save` command will save the current state of moves 1 to i with their current attached names and the next move with the name supplied to it as an argument, or its current name if no argument is supplied. The option of saving the next move with a new name allows Lestrade to emulate aspects of the Automath context

system (though more verbosely). The next move will be renamed to the argument of the save command if the save is successful.

closing the next move: The `close` command does nothing if i is 0. If i is positive, the command discards move $i+1$ and all declarations contained in it and decrements the counter i. The close command does not save any declaration information: saving of a move must be done explicitly.

I believe that when a move with the default numerical name is closed, all non-default-named moves extending it should be cleared. This has not been implemented yet.

the clearcurrent command: The `clearcurrent` command has the effect of `close` followed by `open`: it empties the next move of declarations while not changing the index of the next move. `clearcurrent` is needed as a separate command, however, because move 1 cannot be closed, but can be cleared of declarations. `clearcurrent` with an argument will empty the current version of move $i+1$ and replace it with a new one or previously saved one named by the argument, except that if the argument is the default numerical name of the next move it is always cleared, and in this case any moves extending it are forgotten. The first named move that is opened in a given session will be opened with `clearcurrent` (or have its name changed to a non-default name by `save`).

discovering saved moves: The `Showmoves` command will display saved moves available for `open` and `clearcurrent`.

purging of saved moves: When `clearcurrent` overwrites a nonempty saved move with default numerical name with a blank move or when `save` saves the next move with a name already in use in that context, the overwritten saved move and all of its extensions with further moves are deleted. It should be noted that what identifies a saved move to Lestrade is not its name by itself but the whole string of names of previous moves at the time it was saved (and whose state as saved moves is updated when it is saved); different moves may have the same name if their contexts are different without confusion.

the display commands: The `Showdec` command displays declaration information about an identifier. The `Showdecs` command shows all declarations. I do not have convenient methods of breaking out of the latter command installed yet.

the Load command: This command with argument `<filename1>` (which must be a token) will clear everything and load the prover state as it was when the command `readline(2) <filename1> <filename2>` was last run. Don't put the file name in quotes in the Lestrade interface!

the import command: This command with argument `<filename1>` will add the information in move 0 saved by the previous reading of `<filename1>.lti` or `<filename1>.tex` as a saved move 1 with the name `<filename1>`. This is nondestructive: information in the current working environment is otherwise unaffected (unless a saved move with the same name is overwritten). This allows proofs written in other files to be imported, with care (one needs to note the automated changes of identifiers which correct name conflicts, noted just below). This command is not installed in the new implementation.

warning: name conflicts Instances of the `open`, `clearcurrent`, and `import` commands may cause name conflicts. The new implementation should handle these seamlessly; it is true though that while the prover can tell which meaning of an identifier it needs to pay attention to, it will not show you when an identifier from an earlier move has a different meaning.

variables: We refer to all identifiers declared at the next move without definitions as *variables*. Construction variables may have been introduced with the `postulate` command at some stage when the current next move was the last move, but can also be introduced directly.

argument list semantic conditions: We provide further that all items in the argument lists which follow the keyword and the identifier being declared in `declare`, `postulate`, and `define` commands must be variables. Moreover, they should appear in the order in which they were declared. This is an easy way to enforce the constraint that the sort of a variable in such an argument list may depend on a second variable appearing in the argument list only if this second variable appears earlier in the list. Conversely, if the sort of a variable in the list depends

on any second variable, this second variable must appear earlier in the argument list. The new implementation handles this condition without warning by reordering the arguments to agree with the order in which they are declared in the next move, in the internals of the `Posit` command, which is used to implement both `postulate` and `define`.

semantics of the declare command: The command

```
declare <identifier> <object sort>
```

declares the identifier as having the given object or construction sort (as long as the identifier is undeclared and the sort term sort-checks).

The object sort terms `obj`, `prop`, `type` of course will sort check. The term `that <object term>` sort-checks iff the object term has sort `prop`. The term `in <object term>` sort-checks iff the object term has sort `type`.

The last argument can also be a construction sort term, allowing one-line declarations of construction variables. A construction sort term has the shape $[x_1, \ldots, x_n \Rightarrow \tau]$ and will type check if each x_i is a variable in the next move and τ type checks as an object sort.

This declaration is added to the next move if it succeeds.

constructing an object: The command

```
postulate <identifier> :  <object sort>
```

(with null argument list) works as the previous line does, except that the identifier is declared (and type checked) at the last move.

general semantics of the postulate command: The command

```
postulate <identifier> <argument list> :  <object sort>
```

declares the identifier (which must not have been previously declared) as a construction of arity equal to the (positive) length of the argument

list.[11]. If the argument list is $(x_1, \ldots, x_n)$, with the type of x_i being τ_i, and the object sort is τ, the type recorded is

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

where the variables x_i are to be understood as bound. This may be a quite complex dependent type: observe that each τ_i may contain occurrences of x_j for $j < i$, and τ may contain occurrences of any of the x_i's (and any variable occurring free in τ or any τ_i must be one of the x_i's). Lestrade in fact renames all the bound variables, attaching a fresh numerical tag (the same tag for all variables in this argument list) to each of the x_i's, and this procedure is repeated whenever a substitution is made into a construction sort, to avoid bound variable collision problems. Note that some or all of the τ_i's may themselves be construction sorts, but the output type τ must be an object sort. The symbol - is a marker indicating that this is a primitive construction rather than a defined construction. This declaration is added to the last move.

defining an object: The command

```
define <identifier> :  <object term>,
```

where the object term (which we write as D) type checks, declares the identifier (which must not have been declared previously) with the sort $[(D, \tau)]$ [as if it were a nullary construction], where τ is the type of D. The identifier can be expanded to D by prover constructions as required.

general semantics of the define command: The command

```
define <identifier> <argument list> :  <object term> ,
```

where the object term (which we write as D) checks with type τ, will cause the identifier (which must not have been declared previously) to be assigned the sort $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (D, \tau)]$: this will succeed

---

[11]The command was originally called `construct` and this may need to be fixed in old scripts.

exactly if τ is the sort of D and $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (-, \tau)]$ is a well-formed construction sort (satisfying expected restrictions on variable dependencies, etc.) This serves as the recorded type of the identifier declared (and the declaration is created at the last move). The identifier can be expanded to its sort (understood as a λ-term) by prover constructions, and any term $D(t_1, \ldots, t_n)$ which is well-typed can be expanded by substitution in the natural (and rather complex) way.

semantics of the `rewritep` and `rewrited` commands: These commands are not yet avialable in the new version, and may change materially if and when they are introduced there.

These commands allow construction of a primitive construction witnessing validity of a rewrite rule (`rewritep`[12]), or introduction of a rewrite rule by exhibiting a construction already constructed or defined of the appropriate type (`rewrited`). The argument list component consists of an initial segment of the actual argument list of variable arguments for the initial identifier, followed by two complex terms, the pattern `pattern` and the target `target`. The pattern and the target must be of the same type τ; Lestrade generates an additional argument P typed as a construction from τ to `prop`. The intended sort of the initial identifier is then the construction sort sending the argument list before the pattern with P and a variable of sort **that** $P(\texttt{pattern})$ appended, to the output type **that** $P(\texttt{target})$. All the variables in the explicitly given argument list must actually appear in `pattern`; all variables that appear in `target` must appear in `pattern`. The existence of a construction of the indicated type will witness that any proof of $P(\texttt{pattern})$ can be mapped to a proof of $P(\texttt{target})$, which certainly looks like evidence that `pattern` must be equal to `target` for all values of the included variables. The initial identifier in the case of `rewritep` must be undeclared, and is declared with the indicated type if all the conditions hold (and a rewrite rule `pattern := target` is recorded, with variables moved to a fresh namespace). The initial identifier in the case of `rewrited` must already have been defined or constructed with the indicated type – if this checks out, a rewrite rule is recorded in the same way. When a rewrite rule is added to the last

---

[12]the shape of this command was originally `rewritec`, and this may need to be revised in old scripts.

move with `rewrited`, any rewrite rule already associated with the same first identifier in the last move is deleted (but such rules are not deleted from lower indexed moves): this supports changes in the order in which rewrite rules are applied, as roughly speaking the most recently introduced rewrite rule is attempted first. It should be noted again that a `rewritep` or `rewrited` command declares a number of additional variables before declaring (or confirming the validity of the declaration of) the initial identifier.[13] [14]

Use of rewriting in a definition does not record dependencies on the rewrite theorem (whether postulated or defined) in the resulting term; thus Lestrade does not allow export of things defined in a given move using rewrites declared in that move to lower indexed moves. The way this is enforced is that the postulate and construct commands will not work if there are rewrites declared in the next move. Further, an attempt to open a saved move with rewrites in it will fail if there are conflicts with names introduced since that move was saved (in the absence of rewrites, a renaming scheme fixes this).

semantics of the `Declare` and `Posit` commands: These commands are new in the new implementation and are mainly used to implement the original logical commands, but they are available to users. Each of them takes a fresh identifier and equates it with what is underneath a lambda term (terms of all four types are accepted and suitably type cast). The `Declare` command puts the declaration in the next move: this allows a very free declaration of primitive or defined terms not supported directly in the old version. The `Posit` command puts the declaration in the last move, and reorders arguments, supplying implicit

---

[13]The symmetry between `prop` and `type` could be restored by providing versions of these commands which generate the new variable P as a variable construction from the common type of the source and target to `type` rather than `prop`, but we see no particular reason to do this.

[14]It is important to notice that introduction of an object by `rewritep` definitely amounts to introducing a new primitive to a theory, but further the introduction of an object by `rewrited` may fundamentally change a theory (`rewrited` is not just a definition facility, or viewing it as such amounts to adopting a fundamentally stronger logical framework). We have an example of a theory which becomes inconsistent on carrying out a `rewrited` command; we also have an example of a theory in which new types become possible on execution of a `rewrited` command, because rewriting may be used in checking whether types match.

arguments where needed, and expands definitions. The old logical commands are implemented using these commands. The old commands are philosophically different from the new ones in preferring parameters to bound variables.

expansion of local definitions: Whenever a declaration is created in the last move, all defined identifiers in it whose declarations are found in the next move must be expanded (this is one way that the internal λ-terms appear; they can also be introduced by the implicit argument inference mechanism): this must be done because a declaration at the last move needs to continue to make sense if the next move is closed. Of course, if such expansions reveal dependencies on variables not found in the argument list, an error is reported.

computation of sorts of terms: We now discuss assignment of sorts to terms, and computation of identity of sort terms [and, as it turns out, computation of definitional expansions].

A bare identifier (whether an object or a construction) is assigned type by lookup, with the proviso that an identifier typed (D, τ) (denoting a defined object) is simply assigned type τ. A term $f(t_1, \ldots, t_n)$ (or $t_1 f t_2, \ldots, t_n$) is assigned a sort using a procedure of matching of the declared type $(x_1, \sigma_1), \ldots, (x_n, \sigma_n) \Rightarrow (D, \sigma)$ of the identifier f (where D can be – if f is constructed or a term if f is defined) with the list of types τ_i of the t_i's. If the type τ_1 is identical to the type σ_1, we continue by replacing x_1 with t_1 in each σ_i and σ to obtain σ_i^* and σ^*, then continuing the matching of $(t_2, \ldots, t_n)$ with $(x_2, \sigma_2^*), \ldots, (x_n, \sigma_n^*) \Rightarrow (D^*, \sigma^*)$. The lengths of the two argument lists must be the same for the matching to succeed, and the final type assigned is the final form of σ^*. It is worth noting that if f is defined, the term D^* obtained at the end of this procedure is the definitional expansion of $f(t_1, \ldots, t_n)$.

considerations of bound variable naming and definitions may allow typographically distinct terms and sorts to be identified: construction sorts are regarded as identical when one can be converted to the other by renaming of bound variables: the computational procedure for checking this is similar to the term typing procedure. Terms are regarded as identical when an expansion of defined terms or an application of rewrite rules can convert one to the other: when a failure to

match is encountered, an attempt is made to correct it by expanding definitions [or by rewriting]. Due to the simple form of our terms, this is straightforward to compute.

expansion of definitions: Expansion of $F(t_1, \ldots, t_n)$ where the type of F is $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (D, \tau)$ is a straightforward matter of substitution [already described incidentally in the discussion of type matching], and sort safe as long as all terms involved have already been sort-checked. Have t_1 replace x_1 in all terms and types, then t_2 replace x_2, and so forth. As noted above, a defined construction appearing by itself as an argument expands to its sort, which can be understood as a λ-term.

action of rewrite rules: This is a new feature. When a term is rewritten, the most recently introduced rewrite rule whose pattern matches it is applied to it [the precise order is, most recently introduced rewrite rule in the highest indexed move which contains such a rewrite rule; the use of the most recently introduced rule first makes it easier for the user to reprogram the rewrite system]. Equations between objects and types will succeed if they can be made to work by rewriting, just as they will succeed if they can be made to work by definition expansion. Only the rewrites introduced at the last move or moves of lower index are applied, though rewrites at the next move are stored. The list of rewrites recorded at each move is managed in the same way as the moves are managed by the open, close, clearcurrent, and save commands.

To maintain confluence, there is a subtlety about matching: when a pattern is matched with a term, proper subterms of the target term are head-rewritten before being matched with the source, and the match succeeds only if it still works after this rewriting.

The object output of the define command is rewritten. Types are not rewritten in any command (though rewriting may be used to verify equivalences between types: the fact that the system does is an indication that the rewrite system adds something to the logical framework, and we have actual examples which demonstrate this).

another type inference feature (implicit arguments): Lestrade has an extensive ability to accept construction and definition declarations of constructions with missing arguments. Lestrade infers and supplies the

required missing arguments at declaration time by finding variables in the sorts of the explicitly given arguments, and it is able to determine where to put them to get a well formed argument list. When typing an instance of a construction with implicit arguments, it deduces the values of the implicit arguments by sort matching. This includes the ability to find construction arguments which have not been given explicit names. For a construction implicitly given to be recovered when it appears in a sort in applied position, it must have a fully abstract occurrence, that is, one in which its arguments coincide with an initial segment of the bound variables in its context in the correct order, or it must be applied to concrete fully sortable arguments [the circumstances under which implicit arguments can be inferred are quite complicated to describe; there are extensive examples in sample files on the Lestrade page, mostly simple ones, but a better essay on this is needed].

Implicit arguments can be recognized in the output because their names have a prepended dot. The implicit argument mechanism does not affect the core logic of Lestrade at all: all terms in fact have all usually expected arguments. It is purely a function of input and output. Sort displays show all input arguments, implicit and explicit of a construction whose sort is being displayed: the default behavior is that only explicit arguments of constructions appearing in applied position in sort displays are shown: this behavior can be changed as desired with commands `showimplicit` and `hideimplicit`. The sort matching which supplies implicit arguments may fail if some definitional expansion of the sort being matched actually eliminates the information about the implicit argument in the sort: there is no guarantee that the way the implicit argument is presented in the sort in the original declaration is deducible semantically from the value of the sort: it is entirely a matter of syntax. Rewriting cannot at present be used to assist implicit argument extractions, though we have encountered situations which suggest to us that this should be implemented.

The precise condition under which an implicit construction argument appearing in applied position in the sort of an explicit argument can be matched is that it appears either applied to arguments which do not depend on variables bound in that sort term (in which case it will match an explicit construction term in applied position applied to matching terms) or applied to arguments which are an initial segment

of the variables bound in the context of that sort term, appearing in the precise order in which they appear bound in the context (in which case it will match a lambda term constructed by Lestrade, if the term matching it does not depend on any other bound variables in the context). [This is obscure and needs supporting examples.]

Introduction of this feature made it much less urgent to install user entered lamba terms (though we have now done this): laborious construction of constructions by the user (notably predicates quantified over) is greatly reduced, as the values of these arguments can often be inferred by higher order matching as anonymous internal lambda terms. It does also mean that one should look carefully at the actual type of a defined or constructed construction, including its implicit arguments: all computations with a construction with implicit arguments will use its full type.

A term $f(t_1, \ldots, t_m)$ where m is less than the arity n of f will be read as $[(x_{m+1} \ldots x_n) \Rightarrow f(t_1, \ldots, t_m, x_{m+1}, \ldots, x_n)]$ if the types of the t_i's are appropriate. This only works if the argument list is explicitly closed with parentheses. Such a term can only appear as an argument.

The user may enter λ-terms as arguments. The syntax is of the exact form $[x_1, \ldots, x_n \Rightarrow T]$, the bound variables x_i being variables declared at the next move, separated by commas, and T being an object term. The identifier => is now reserved. The form displayed by Lestrade for these terms will have the bound variables decorated with a namespace index and will be fully adorned with sorts for the variables.

3.2.1 Analogies between Lestrade lines and Automath lines; remarks on differences between the Lestrade and Automath type systems

An Automath line consists of four components: an indicator of context, an identifier being declared, a definition of that identifier, and a type. The analogue of the identifier component in a Lestrade line is always evident: it is the identifier following the command keyword. We are here considering only `declare`, `postulate`, and `define` lines in a Lestrade text: other lines do not have analogues in Automath.

In Automath, the context mechanism is fairly simple, consisting of either 0 or a reference to a previous identifier. One is then defining the new identifier as indicated by the definition component with a stated type in a context in which the previous identifier and all identifiers one finds by backtracking to its context indicator and iterating are present.

In Lestrade, the context is more complex: the intersection of the explicit context given by Automath with variables declared in the next move is present as the argument list to the identifier introduced by a `postulate` or `define` command (all items being listed, with no chained expansion of the context as in Automath), but additional related roles are played by the system of moves. A variable introduced in Lestrade using the `declare` command may be thought of as having all previous variables in its (unindicated) context, and certainly at least those variables which explicitly appear in its sort: note that when a variable appears in an argument list, it must be preceded by all the variables mentioned in its sort.[15] Argument lists in Lestrade are more flexible than contexts in Automath: Automath contexts are restricted to forming a tree under the initial segment relation.

The construction of the special definition body PN used in Automath for a primitive notion is handled in Lestrade by using the `postulate` command rather than the `define` command. In the `postulate` command a type is explicitly given; in the `define` command, Lestrade computes and displays the sort: the user does not need to supply it. The construction of the definition body - used for variables in Automath is handled in Lestrade by using the `declare` command, for objects, and by using the `postulate` command then closing the next move, to obtain variable constructions.

The extra trick which allows user-entered lambda terms to be avoided (so

---

[15] We ignore the curious effects of the implicit argument inference feature of Lestrade for the moment.

far) is that whenever an expression is defined using a given context, what is defined is not a variable expression specific to that context, in Automath usable in a different context by supplying terms of appropriate types to replace the elements of the original context not shared with the current context (considered as an argument list), but a construction present at the last move, in effect a name for the construction implementing the variable expression, which would have to be expressed in Automath as a lambda term. Further, constructing a primitive construction of a given sort at the last move, then closing the next move, gives one a variable construction of that type in the formerly last, now next move. By the use of these two devices, and the fact that constructions may be supplied as arguments to other constructions using their atomic names, one entirely avoids the need to write nonatomic terms of construction sorts. The further feature that constructions do not have construction output ensures that the user never needs to type a construction sort. The ability to enter λ-terms and construction sorts has been installed in Lestrade, though.

It does appear that for fluent use of the system it would be advisable to introduce an ability to declare construction variables directly using a notation for construction sorts and to enter explicit lambda terms as arguments to Lestrade terms (and we have now done this). But it is useful to see that no logical power will be added to our framework when we do this.

In Lestrade, propositions are not themselves sorts, but objects p of a sort `prop`, correlated with sorts `that` p of proofs of p.[16] Mathematical types are for us similarly objects τ of a sort `type`, correlated with sorts `in` τ inhabited by specific objects of those types. Operations on propositions or types can then be postulated naturally by declaring constructions with arguments of type `prop` or `type` (not actual sorts as arguments).

Constructions are not objects and cannot be outputs of constructions.[17] But they can be inputs to constructions, and one can create objects correlated with constructions of a particular sort. For example, we do not identify proofs of $p \to q$ with constructions from `that` p to `that` q (which are not

---

[16]Later dialects of Automath have propositions as types; references I have looked at suggest that earlier dialects have a system analogous to the Lestrade system, with p an object of type `prop` and a separate type `Proofs`(p) inhabited by proofs of p; it may always have been the case that the same notation was used in Automath code for the object of type `prop` and the associated type.

[17]This might superficially seem to be modified by the 10/10 update allowing construction arguments to be constructed by currying.

objects), but instead provide a construction `Ifproof` which sends such a construction to an object of the sort `that` $p \rightarrow q$. One can see in the extensive Lestrade book given in section 6 that this does not obstruct the usual sorts of reasoning in a system of this kind. The move system of Lestrade quite naturally implements reasoning under hypotheses ending with the proof of an implication or reasoning about arbitrarily postulated objects ending in the proof of a universally quantified statement.

It is worth noting that Lestrade does not have any specific notion of construction application, any more than it has user-written lambda terms. Lestrade constructions are always applied to their complete argument lists[18] in the format in which they were defined (or appear by themselves as arguments[19]). The application of Lestrade constructions to their argument lists corresponds to the application of atomic defined terms in Automath to argument lists replacing items in the context in which they were originally defined (substitution rather than application), although the atomic Automath terms are as it were variable expressions and the Lestrade construction terms in effect denote functions.

A real difference in strength between Automath and Lestrade can be seen in connection with the odd subtyping supported in later versions of Automath. Automath regards a type inhabited by functions from type τ to type `prop` as a subtype of type `prop`. Further, it regards an element p of `prop` as being itself a type (in our terms, `that` p is identified with p). This gives quantification over type τ for free. Any predicate of type τ is itself a proposition. If a predicate P is inhabited by an object pp, then this object is itself a function from elements n of type τ to proofs of $P(n)$: so such an inhabitant is a proof of $(\forall n \in \tau : P(n))$. Further, τ itself may be a quite complex function type: we get quantification of all orders for free from the type system. In Lestrade, it is actually possible to postulate quantification over all object types and construction sorts uniformly, indirectly, in a way suggested at the end of our large example of Lestrade text, but it takes considerable work. To provide quantification over any type (or indeed to get any inhabitant of `prop` at all) requires some declarations in Lestrade.

Let's explore why an inhabitant of P is a function from $n \in \tau$ to type $P(n)$ in the Automath framework. The underlying idea is that an expression

---

[18]or to partial argument lists in which the missing implicit arguments can be extracted from the sorts of the arguments given explicitly.

[19]The 10/10 update allows constructions with shortened argument lists to represent "curried" construction arguments.

$f(x)$ of type $\sigma(x)$ is an expression $\lambda x.f(x)$ of type $(\lambda x.\sigma(x))$. So if f is of type $(\lambda x.\sigma(x))$, $f(x)$ is of type $\sigma(x)$. So if P is of type $\tau \to \texttt{prop}$, which is supposed to be an element of type $\texttt{prop}$ as well, then an inhabitant of P is a function f such that $f(x)$ is an inhabitant of $P(x)$. But $P(x)$ is also of type $\texttt{prop}$, so $f(x)$ is a proof of $P(x)$ for each x. This is very cute. It also has all the advantages of theft over honest toil, as Russell said in some similar context.

For us, for any type τ we must declare a constructor sending any predicate P of type τ to an element $\forall P$ of $\texttt{prop}$, then further postulate a construction sending any construction which takes $x : \tau$ to an element of $\texttt{that } P(x)$ to a proof of $\forall P$. We can declare these uniformly over the object types fairly easily, and with a little trickery we can indirectly declare them over all construction sorts as well. But we also have the option of declaring only the quantifiers we want, and never enabling higher order quantification. An Automath theory seems to be of necessity a higher order theory.

In his paper [9] describing his implementation [8] of Automath, Wiedijk discusses the difference between the λ-types of dependently typed functions found in Automath and the Π-types found in the currently popular type systems. I am frankly not certain of the place of the Lestrade type system on this dimension. The difficulties that arise in Automath related to this issue seem to relate to the possibility of the same term appearing as both an object of the system and a sort, and this is ruled out in Lestrade. The rule that if F types as G, then Fa types as Ga, which seems to characterize the λ-approach, holds for Lestrade, but note that a must be a complete argument list, Fa must be an object (and so cannot be a construction) and Ga must be an object sort (and the object sorts have no intersection with the objects of the theory, though objects can be packaged in them via the $\texttt{that}$ and $\texttt{in}$ constructors). In fact, where $F : G$ and $F(a) : G(a)$ in Lestrade, the four objects mentioned are all of different metasorts: F is a construction, G is a construction sort, $F(a)$ is a object and $G(a)$ is an object sort.

3.2.2 A sample Lestrade book with rewriting

We introduce a brief Lestrade book to illustrate capabilities of the rewrite system.

Watson, referred to in the comments, is an earlier theorem proving project of mine (see [11]), an equational prover which incorporated a fairly elaborate scheme of programming using interlocking rewrite rules, described in [10]. We believe that most features of the Watson programming system are implementable in Lestrade rewrites. The rule `Assocs` in the second example, which implements regrouping of arbitrarily complex nested sums so that all grouping is to the right, parallels a basic Watson example.

Notice in both examples that the final examples of `define` commands exhibit "execution behavior". It would be useful to give an example in which the basic properties underlying the rewrite rules are proved then the rewrites introduced using the `rewrited` command, rather than having the constructions providing evidence for the validity of the rewrite rules introduced by fiat using the `rewritep` command: though one should also note that `rewritep` is a perfectly respectable way to introduce equational axioms.

Execution of this Lestrade block is suppressed in the version of this file addressing the new implementation.

```
% begin Lestrade execution

   >>> declare x obj


   x : obj


   {move 1}

   >>> declare y obj


   y : obj
```

35

```
   {move 1}

   >>> postulate pair x y obj


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


   {move 0}

   >>> postulate p1 x obj


   p1 : [(x_1 : obj) => (--- : obj)]


   {move 0}

   >>> postulate p2 x obj


   p2 : [(x_1 : obj) => (--- : obj)]


   {move 0}

   >>> rewritep First x y, p1 pair x y, x

{function error}

 Inspector Lestrade says:  general failure of functionsort line 3030

(paused, type something to continue) >

   {move 1}

   >>> rewritep Second x y, p2 pair x y, y
```

```
{function error}

 Inspector Lestrade says:  general failure of functionsort line 3030

(paused, type something to continue) >

   {move 1}

  >>> open


     {move 2}

     >>> declare x1 obj


     x1 : obj


     {move 2}

     >>> declare y1 obj


     y1 : obj


     {move 2}

     >>> define reverse x1 : pair (p2 x1, p1 \
         x1)


     reverse : [(x1_1 : obj) => (---
         : obj)]
```

```
    {move 1}

    >>> define reversetest x1 y1 : reverse \
        (pair x1 y1)


    reversetest : [(x1_1 : obj), (y1_1
        : obj) => (--- : obj)]


    {move 1}

    >>> close


{move 1}

>>> define testing x y : reversetest x y


testing : [(x_1 : obj), (y_1 : obj) =>
    ({def} p2 (x_1 pair y_1) pair p1
    (x_1 pair y_1) : obj)]


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


{move 0}

>>> clearcurrent

{move 1}

>>> postulate Nat type
```

```
Nat : type


{move 0}

>>> declare m in Nat


m : in Nat


{move 1}

>>> declare n in Nat


n : in Nat


{move 1}

>>> declare p in Nat


p : in Nat


{move 1}

>>> postulate + m n in Nat


+ : [(m_1 : in Nat), (n_1 : in Nat) =>
     (--- : in Nat)]


{move 0}
```

```
>>> postulate assoc m in Nat

assoc : [(m_1 : in Nat) => (--- : in
    Nat)]


{move 0}

>>> postulate assocs m in Nat


assocs : [(m_1 : in Nat) => (--- : in
    Nat)]


{move 0}

>>> rewritep Assocfails m, assoc m, m
```
{function error}

 Inspector Lestrade says: general failure of functionsort line 3030

(paused, type something to continue) >
```
   {move 1}

   >>> rewritep Assocsfails m, assocs m, m
```
{function error}

 Inspector Lestrade says: general failure of functionsort line 3030

(paused, type something to continue) >
```
   {move 1}
```

```
   >>> rewritep Assocrule m n p, (m + n) + p, m + (n + p)

{function error}

 Inspector Lestrade says:  general failure of functionsort line 3030

(paused, type something to continue) >

   {move 1}

   >>> rewritep Assocsrule m n p, (m + n) + p, assocs \
       (assoc (m + (assocs (n + p))))

{function error}

 Inspector Lestrade says:  general failure of functionsort line 3030

(paused, type something to continue) >

   {move 1}

   >>> declare q in Nat


   q : in Nat


   {move 1}

   >>> define test m n p q : (m + n) + (p + q)


   test : [(m_1 : in Nat), (n_1 : in
       Nat), (p_1 : in Nat), (q_1 : in
       Nat) =>
       ({def} (m_1 + n_1) + p_1 + q_1 : in
       Nat)]
```

41

```
test : [(m_1 : in Nat), (n_1 : in
    Nat), (p_1 : in Nat), (q_1 : in
    Nat) => (--- : in Nat)]


{move 0}

>>> declare r in Nat


r : in Nat


{move 1}

>>> declare s in Nat


s : in Nat


{move 1}

>>> define test2 m n p q r s : ((m + n) + p) + ((q + r) + s)


test2 : [(m_1 : in Nat), (n_1 : in
    Nat), (p_1 : in Nat), (q_1 : in
    Nat), (r_1 : in Nat), (s_1 : in
    Nat) =>
    ({def} ((m_1 + n_1) + p_1) + (q_1
    + r_1) + s_1 : in Nat)]


test2 : [(m_1 : in Nat), (n_1 : in
    Nat), (p_1 : in Nat), (q_1 : in
    Nat), (r_1 : in Nat), (s_1 : in
```

```
          Nat) => (--- : in Nat)]


    {move 0}
% end Lestrade execution
```

3.3 A sample Lestrade book with implicit arguments

```
begin Lestrade execution

   >>> Clearall


   {move 1}

   >>> declare p prop


   p : prop


   {move 1}

   >>> declare q prop


   q : prop


   {move 1}

   >>> postulate & p q prop


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


   {move 0}

   >>> declare pp that p
```

```
pp : that p


{move 1}

>>> declare qq that q


qq : that q


{move 1}

>>> postulate Andintro pp qq : that p & q


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


{move 0}

>>> declare rr2 that p & q


rr2 : that p & q


{move 1}

>>> postulate Andelim1 rr2 : that p


Andelim1 : [(.p_1 : prop), (.q_1
    : prop), (rr2_1 : that .p_1 & .q_1) =>
    (--- : that .p_1)]
```

```
{move 0}

>>> postulate Andelim2 rr2 : that q


Andelim2 : [(.p_1 : prop), (.q_1
    : prop), (rr2_1 : that .p_1 & .q_1) =>
    (--- : that .q_1)]


{move 0}

>>> define Ptest pp : Andintro pp pp


Ptest : [(.p_1 : prop), (pp_1 : that
    .p_1) =>
    ({def} pp_1 Andintro pp_1 : that .p_1
    & .p_1)]


Ptest : [(.p_1 : prop), (pp_1 : that
    .p_1) => (--- : that .p_1 & .p_1)]


{move 0}

>>> postulate -> p q prop


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


{move 0}
```

```
>>> open

   {move 2}

   >>> declare pp2 that p


   pp2 : that p


   {move 2}

   >>> postulate Ded pp2 that q


   Ded : [(pp2_1 : that p) => (---
       : that q)]


   {move 1}

   >>> close


{move 1}

>>> postulate Ifintro Ded : that p -> \
    q


Ifintro : [(.p_1 : prop), (.q_1 : prop), (Ded_1
    : [(pp2_2 : that .p_1) => (---
        : that .q_1)]) => (--- : that
    .p_1 -> .q_1)]


{move 0}
```

```
>>> define Ifintro0 p q Ded : Ifintro \
    Ded


Ifintro0 : [(p_1 : prop), (q_1 : prop), (Ded_1
    : [(pp2_2 : that p_1) => (--- : that
        q_1)]) =>
    ({def} Ifintro (Ded_1) : that p_1
    -> q_1)]


Ifintro0 : [(p_1 : prop), (q_1 : prop), (Ded_1
    : [(pp2_2 : that p_1) => (--- : that
        q_1)]) => (--- : that p_1 ->
    q_1)]


{move 0}

>>> open


   {move 2}

   >>> declare q2 that q


   q2 : that q


   {move 2}

   >>> define qid q2 : q2


   qid : [(q2_1 : that q) => (---
        : that q)]
```

```
    {move 1}

    >>> close


{move 1}

>>> define Selfimp q : Ifintro qid


Selfimp : [(q_1 : prop) =>
    ({def} Ifintro ([(q2_2 : that q_1) =>
        ({def} q2_2 : that q_1)]) : that
    q_1 -> q_1)]


Selfimp : [(q_1 : prop) => (--- : that
    q_1 -> q_1)]


{move 0}

>>> declare rr that p -> q


rr : that p -> q


{move 1}

>>> postulate Mp pp rr : that q


Mp : [(.p_1 : prop), (.q_1 : prop), (pp_1
    : that .p_1), (rr_1 : that .p_1
    -> .q_1) => (--- : that .q_1)]
```

```
{move 0}

>>> open


    {move 2}

    >>> declare x obj


    x : obj


    {move 2}

    >>> postulate P x prop


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


    {move 1}

    >>> close


{move 1}

>>> postulate Forall P : prop


Forall : [(P_1 : [(x_2 : obj) =>
       (--- : prop)]) => (--- : prop)]


{move 0}
```

```
>>> declare U that Forall P


U : that Forall (P)


{move 1}

>>> declare y obj


y : obj


{move 1}

>>> postulate Ug U y that P y


Ug : [(.P_1 : [(x_2 : obj) => (---
      : prop)]), (U_1 : that Forall
   (.P_1)), (y_1 : obj) => (---
    : that .P_1 (y_1))]


{move 0}

>>> open


   {move 2}

   >>> declare z obj


   z : obj
```

```
{move 2}

>>> postulate ui z that P z


ui : [(z_1 : obj) => (--- : that
    P (z_1))]


{move 1}

>>> close


{move 1}

>>> postulate Ui P, ui : that Forall \
    P


Ui : [(P_1 : [(x_2 : obj) => (---
    : prop)]), (ui_1 : [(z_2
    : obj) => (--- : that P_1 (z_2))]) =>
  (--- : that Forall (P_1))]


{move 0}

>>> open


{move 2}

>>> declare w obj


w : obj
```

```
{move 2}

>>> open


   {move 3}

   >>> declare zz that P w


   zz : that P (w)


   {move 3}

   >>> define zzid zz : zz


   zzid : [(zz_1 : that P (w)) =>
       (--- : that P (w))]


   {move 2}

   >>> close


{move 2}

>>> define Q w : P w -> P w


Q : [(w_1 : obj) => (--- : prop)]


{move 1}
```

```
>>> define zzz w : Ifintro zzid


zzz : [(w_1 : obj) => (--- : that
    P (w_1) -> P (w_1))]


{move 1}

>>> close


{move 1}

>>> define test P : Ui Q, zzz


test : [(P_1 : [(x_2 : obj) => (---
    : prop)]) =>
  ({def} Ui ([(w_2 : obj) =>
    ({def} P_1 (w_2) -> P_1 (w_2) : prop)], [(w_2
    : obj) =>
    ({def} Ifintro ([(zz_3 : that
      P_1 (w_2)) =>
      ({def} zz_3 : that P_1 (w_2))]) : that
    P_1 (w_2) -> P_1 (w_2))]) : that
  Forall ([(w_2 : obj) =>
    ({def} P_1 (w_2) -> P_1 (w_2) : prop)]))]


test : [(P_1 : [(x_2 : obj) => (---
    : prop)]) => (--- : that Forall
  ([(w_2 : obj) =>
    ({def} P_1 (w_2) -> P_1 (w_2) : prop)]))]


{move 0}
```

54

```
>>> declare r prop


r : prop


{move 1}

>>> open


   {move 2}

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


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


   {move 2}

   >>> define firstlink outerhyp : Andelim1 \
       outerhyp


   firstlink : [(outerhyp_1 : that (p ->
       q) & q -> r) => (--- : that p ->
       q)]


   {move 1}

   >>> define secondlink outerhyp : Andelim2 \
       outerhyp
```

```
secondlink : [(outerhyp_1 : that
    (p -> q) & q -> r) => (--- : that
    q -> r)]


{move 1}

>>> open


    {move 3}

    >>> declare innerhyp that p


    innerhyp : that p


    {move 3}

    >>> define step1 innerhyp : Mp innerhyp \
        firstlink outerhyp


    step1 : [(innerhyp_1 : that p) =>
        (--- : that q)]


    {move 2}

    >>> define step2 innerhyp : Mp (step1 \
        innerhyp, secondlink outerhyp)


    step2 : [(innerhyp_1 : that p) =>
        (--- : that r)]
```

```
        {move 2}

        >>> close


    {move 2}

    >>> define step3 outerhyp : Ifintro \
        step2


    step3 : [(outerhyp_1 : that (p ->
        q) & q -> r) => (--- : that p ->
        r)]


    {move 1}

    >>> close


{move 1}

>>> define Transimp0 p q r : Ifintro0 \
    ((p -> q) & (q -> r), p -> r, step3)


Transimp0 : [(p_1 : prop), (q_1 : prop), (r_1
    : prop) =>
    ({def} Ifintro0 ((p_1 -> q_1) & q_1
    -> r_1, p_1 -> r_1, [(outerhyp_2
        : that (p_1 -> q_1) & q_1 -> r_1) =>
      ({def} Ifintro ([(innerhyp_3
          : that p_1) =>
        ({def} innerhyp_3 Mp Andelim1
        (outerhyp_2) Mp Andelim2 (outerhyp_2) : that
        r_1)]) : that p_1 -> r_1)]) : that
    ((p_1 -> q_1) & q_1 -> r_1) ->
```

```
      p_1 -> r_1)]


Transimp0 : [(p_1 : prop), (q_1 : prop), (r_1
    : prop) => (--- : that ((p_1 ->
    q_1) & q_1 -> r_1) -> p_1 -> r_1)]


{move 0}

>>> define Transimp p q r : Ifintro step3


Transimp : [(p_1 : prop), (q_1 : prop), (r_1
    : prop) =>
    ({def} Ifintro ([(outerhyp_2 : that
        (p_1 -> q_1) & q_1 -> r_1) =>
      ({def} Ifintro ([(innerhyp_3
          : that p_1) =>
        ({def} innerhyp_3 Mp Andelim1
        (outerhyp_2) Mp Andelim2 (outerhyp_2) : that
        r_1)]) : that p_1 -> r_1)]) : that
    ((p_1 -> q_1) & q_1 -> r_1) ->
    p_1 -> r_1)]


Transimp : [(p_1 : prop), (q_1 : prop), (r_1
    : prop) => (--- : that ((p_1 ->
    q_1) & q_1 -> r_1) -> p_1 -> r_1)]


{move 0}

>>> open


   {move 2}
```

58

```
    >>> declare x obj


    x : obj


    {move 2}

    >>> postulate ev x that P x


    ev : [(x_1 : obj) => (--- : that
        P (x_1))]


    {move 1}

    >>> close


{move 1}

>>> postulate Ui2 ev : that Forall P


Ui2 : [(.P_1 : [(x_2 : obj) => (---
        : prop)]), (ev_1 : [(x_2
        : obj) => (--- : that .P_1 (x_2))]) =>
    (--- : that Forall (.P_1))]


{move 0}

>>> open


    {move 2}
```

```
>>> declare x17 obj


x17 : obj


{move 2}

>>> open


   {move 3}

   >>> declare ev2 that P x17


   ev2 : that P (x17)


   {move 3}

   >>> define evid2 ev2 : ev2


   evid2 : [(ev2_1 : that P (x17)) =>
       (--- : that P (x17))]


   {move 2}

   >>> close


{move 2}

>>> define theimp x17 : Ifintro evid2
```

```
    theimp : [(x17_1 : obj) => (---
        : that P (x17_1) -> P (x17_1))]



    {move 1}

    >>> close


{move 1}

>>> define testing P : Ui2 theimp


testing : [(P_1 : [(x_2 : obj) =>
        (--- : prop)]) =>
    ({def} Ui2 ([(x17_2 : obj) =>
        ({def} Ifintro ([(ev2_3 : that
            P_1 (x17_2)) =>
            ({def} ev2_3 : that P_1 (x17_2))]) : that
        P_1 (x17_2) -> P_1 (x17_2))]) : that
    Forall ([(x'_2 : obj) =>
        ({def} P_1 (x'_2) -> P_1 (x'_2) : prop)]))]


testing : [(P_1 : [(x_2 : obj) =>
        (--- : prop)]) => (--- : that
    Forall ([(x'_2 : obj) =>
        ({def} P_1 (x'_2) -> P_1 (x'_2) : prop)]))]


{move 0}
end Lestrade execution
```

3.4 Formalization of the sort system of Lestrade

We use the notation $T[a/x]$ for substitution of a notation a for a variable x in a notation T.

Metasorts **esort** (object sort), **asort** (construction sort),[20] and **arglist** n (argument list sorts for argument lists of length n) for each positive n are postulated. These are not internal sorts of Lestrade at all. The union of the metasorts **esort** and **asort** is the metasort **sort**.

`prop` and `type` are of metasort **esort**.

If p is of sort `prop`, (`that` p) is of metasort **esort**.

If τ is of sort `type`, (`in` τ) is of metasort **esort**.

All terms of metasort **esort** are built in these ways.

If x is a term of sort τ, then τ is of metasort **sort**.

A countable supply of variables of each sort is given.

We view a list $[t_1, \ldots, t_n]$ as a construction in the sense of the metatheory with domain $\{1,\ldots,n\}$ sending each i to t_i: thus $[t]$ is different from t and any notation $[t_1, \ldots, t_n]$ is handled.

If x_i is a variable of sort τ_i for each $i \leq n$, all x_i's are distinct and no x_i occurs in τ_j for $j \leq i$, then $[(x_1, \tau_1), \ldots, (x_n, \tau_n)]$ is a term of metasort **arglist** n. All terms of this metasort are built in this way. Notice that τ_i's may be of metasort **esort** or **asort**.

If t is a term of sort τ, then $[t]$ is an argument list of list sort $[(x, \tau)]$ (the list sort being of metasort **arglist** 1) (for any variable x of sort τ, technically speaking not occurring in τ, something which will not naturally happen). For $n > 1$, $[t_1, \ldots t_n]$ is an argument list of list sort

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

where no x_i occurs in any t_j, iff the list sort is of metasort **arglist** n and t_1 is of sort τ_1 and $[t_2, \ldots, t_n]$ is of list sort $[(x_2, \tau_2[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1])]$. For other list sorts it may have, see the definition of equivalence of objects of metasorts **arglist** n given below [basically a rule for renaming bound variables x_i].

If L is of metasort `arglist` n and τ is of metasort **esort**, then $(L \Rightarrow \tau)$ is of metasort **asort**. All terms of this metasort are built in this way. Notice that the input sorts found in L may be either construction sorts or object

---

[20]These names reflect the older terminology "entity" and "abstraction" for "object" and "construction".

sorts, but the output sort is always an object sort. Note that **asort** and **esort** are disjoint.

If t is a term of sort τ_1 and f is a term of sort $[(x_1, \tau_1)] \Rightarrow \tau$ (where x_1 does not occur in t), then $f[t]$ is a term of sort $\tau[t/x_1]$.

If $[t_1, \ldots, t_n]$ is an argument list of list sort $[(x_1, \tau_1), \ldots, (x_n, \tau_n)]$ where no t_j contains any x_i, and f is a term of sort $[(x_1, \tau_1), \ldots, (x_n, \tau_n)] \Rightarrow \tau$ then $f[t_1, \ldots, t_n]$ is a term of the same sort as $g[t_2, \ldots, t_n]$, where g is a variable of sort

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

An equivalence relation on objects of metasort **arglist** n for each n is defined: $[(x_1, \tau_1)]$ is equivalent to any $[(x_1^*, \tau_1)]$. $[(x_1^*, \tau_1^*), \ldots, (x_n^*, \tau_n^*)]$ is equivalent to $[(x_1, \tau_1), \ldots, (x_n, \tau_n)]$ iff $\tau_1 = \tau_1^*$ and $[(x_2^*, \tau_2^*), \ldots, (x_n^*, \tau_n^*)]$ is equivalent to

$$[(x_2, \tau_2[x_1^*/x_1]), \ldots, (x_n, \tau_n[x_1^*/x_1])],$$

where none of the starred variables occur in the unstarred term; the relation is then extended to be transitive. Equivalent argument list sorts may freely replace one another in all contexts. This amounts to the observation that the x_i's are bound in this construction and can freely be renamed. Similarly, $[(x_1, \tau_1), \ldots, (x_n, \tau_n)] \Rightarrow \tau$ is equivalent to $[(x_1^*, \tau_1^*), \ldots, (x_n^*, \tau_n^*)] \Rightarrow \tau^*$ under the same conditions under which any term $[(x_1, \tau_1), \ldots, (x_n, \tau_n), (x_{n+1}, \tau)]$ with x_{n+1} a variable of correct sort is equivalent to

$$[(x_1^*, \tau_1^*), \ldots, (x_n^*, \tau_n^*), (x_{n+1}, \tau^*)].$$

In particular, a term or list of any sort has all equivalent sorts as well.[21]

Note that application terms are always of sorts which are of metasort **esort**. Non-atomic terms of sorts of metasort **asort** are of the shape

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

where D is a term of sort τ. $([(x_1, \tau_1), \ldots, (x_n, \tau_n)] \Rightarrow (D, \tau))[t_1, \ldots, t_n]$ reduces to $D[t_1/x_1][t_2/x_2] \ldots [t_n/x_n]$, if the argument list is of the correct argument list sort. The Lestrade user does not in the current version write any such terms (they are introduced implicitly by definitions of constructions) or set up any such reductions, but they do appear in displayed sorts and such reductions happen in sort computations. Equivalence induced by renaming

---

[21] An error in this description is most likely to have occurred here: bound variable renaming is tricky!

of bound variables is defined for these terms in essentially the same way as for construction sorts and argument list metasorts.

When the Lestrade engine computes a sort for a term or otherwise makes a substitution into a bound variable construction, it renames all bound variables in a way guaranteed to give fresh names. When attempting to match sorts, the engine attempts definitional expansion of the sorts (and also rewriting of the sorts) if it does not initially see them as equivalent up to renaming of bound variables.

10/10/16 change allowing more construction arguments: The code of 10/10 while not implementing λ terms per se permits easier formation of construction arguments. A term $f(t_1, \ldots, t_m)$ where m is less than the arity n of f will be read as $[(x_{m+1} \ldots x_n) \Rightarrow f(t_1, \ldots, t_m, x_{m+1}, \ldots, x_n)]$ if the sorts of the t_i's are appropriate. This only works if the argument list is explicitly closed with parentheses. If f is a defined construction, it will be definitionally expanded. Such a term can only appear as an argument

10/16/17 change allowing λ-terms: The user may enter λ-terms as arguments. The syntax is of the exact form $[x_1, \ldots, x_n \Rightarrow T]$, the bound variables x_i being variables declared at the next move, separated by commas, and T being an object term. The identifier `=>` is now reserved. The form displayed by Lestrade for these terms will have the bound variables decorated with a namespace index and will be fully adorned with sorts for the variables.

3.5 A sketch of semantics for a large class of Lestrade theories

The referent of a sort is a set. The referent of a term of a particular sort will be an element of the referent of the sort. We will refer to the sets which are referents of sorts (**in** τ) as "types". [we may further suppose, if we are willing to be boringly classical, that the referent of `prop` has exactly two elements and that types (**that**,τ) are also sets, each such type having one element or none: these sets may also be viewed as types].

The referent of a construction sort $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (---, \tau)$ is a space of functions whose range is the set which is the referent of τ and whose domain is a complex set of n-tuples: the domain of the referent of $((x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (---, \tau)$ is the set of all n-element lists whose whose head t_1 belongs to the referent of τ_1 and whose tail belongs to the domain of the referent of $((x_2, \tau_2[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1]) \Rightarrow (---, \tau[t_1/x_1])$.

The scope of this type system is better understood by adding dependent product constructions and dependent function space constructions: where τ is a type and F is a construction from the type τ to types, the dependent product $\tau \times F$ of τ and F is the set of ordered pairs (t, u) where $t \in \tau$ and $u \in F(t)$. The dependent arrow type $\tau \rightarrow F$ is the set of constructions f with domain τ such that $f(t) \in F(t)$. Iteration of dependent product will give referents of the domain types of all of our construction sorts: the construction sorts themselves are dependent function types.

If the cardinality of the domain of types is taken to be inaccessible and each type is taken to be of smaller cardinality than the domain of types, it is demonstrable that each dependent product type and dependent function type whose first component is a type and values of whose second component are types is of lower cardinality than the domain of types. Of course we can produce paradoxes by using `type` as a component of a sort or by asserting existence of constructions which violate cardinality restrictions on types.

Lestrade declarations thus translate into assertion of existence of elements of certain collections of functoions in an ambient set theory.

At the end of the sample book in section 6 we exhibit declarations of dependent product and dependent function types: although the Lestrade user does not directly type construction sorts, the user can develop theories in which these sorts are represented internally by types accessible to the user. This could be done more faithfully to the Lestrade sort system: what is done in the book is a proof of concept. It is very important to notice that the

declarations of dependent product and dependent function types are new axioms asserted in the logical framework, not consequences of the built-in features of the logical framework: the logical framework cannot describe its own inner workings unaided.

One can formulate Lestrade theories which do not fit this semantics: a constructive theory would have quite different semantics, for example. Some interesting manipulations of `type` as a type which are used in current type systems would of course not fit with the naively set theoretical scheme of semantics described here.

4 Using Lestrade

Lestrade is implemented currently in Moscow ML 2.01. The files can be found on my web page ([12]). I am planning to produce an implementation in Python 3 (there is a previous implementation in Python, which is deprecated: it contains errors which I do not intend to correct – instead I plan to port the ML version back into Python).

I have now installed a function basic() which if called from the ML command line will disable rewriting and implicit arguments, and a command explicit() which will disable implicit arguments but leave rewriting constructions active. These are not internal commands of Lestrade. fullversion() will restore the usual behavior.

Features which we plan to introduce are user-entered λ-constructions (entered as arguments, and only when types of the bound variables are fully deducible from the body of the construction). A blue-sky idea I am thinking about is the implementation of imperative programming constructions: it does not seem impossible that this can be done in a type-safe manner, and it would be very interesting if it could be done.

Lestrade books are text files with the extension `.lti`. Edit them with a text editor (fill them with lines in the format described in section 4) and the Lestrade checker should be able to do something with them.

The basic file handling command of Lestrade is

`readfile "<source file>" "<target file>";`

(to be typed on the ML command line!)

which executes the Lestrade lines in the file `<source file>.lti` and echoes the commands and any responses from the system to `target file>.lti`. After running the file, the user can type Lestrade lines in the interface and receive immediate feedback both at the console and echoed into the target file, though our usual approach is to edit the source file and issue the readfile command again. To exit the interface, type `quit`. To avoid poisonous problems with execution of readfile, always end a file of Lestrade commands run with readfile with the line `quit`. Execution of files can be chained if subsequent files start with a `load` command: in this case it is important that file names be Lestrade tokens!

The variant command `readbook` works on `.tex` files, instead of `.lti` files, executing Lestrade text which it finds in verbatim blocks in the LaTeX file and otherwise echoing LaTeX code. This allows scripts to be maintained with nice typeset comments.

One can also type

`interface "<target file>";` to enter Lestrade lines at the console and echo the results to the given target file. If the null string is used, no target file is used (or at least none is intended).

If the source file contains valid Lestrade commands, the target file will in fact be executable as a source file with the same effects, and will in addition be better formatted and commented with such things as the sorts of all the terms declared.

If an error is encountered while reading a file, Lestrade will stop and issue an error message. Hitting return will scroll through any further error messages, and Lestrade will eventually stop the reading of the file and leave the user in the interface: it will not continue to execute commands in a file after an error is encountered.

Upgrades now allow the `readfile` command to be issued in the interface (it should not appear in a script file, where it will not work correctly). Filenames which are Lestrade tokens (not enclosed in parentheses) may be used naturally as arguments to `readfile` in the interface. If an argument is not a Lestrade token, it can still be given as the final argument if preceded by a double quote. If `readfile` appears with only one argument (which will be the case if an argument begins with a double quote) the second argument is implicitly `scratch` (`scratchtex` for `readbook`). The target file (second argument) becomes impossible to open for editing when `readfile` is issued in a script; it can be released by running it with output to `scratch` (or any other file). Similar considerations apply to `readbook`.

Notice that a file which appears as the argument to `load` or `import` in a script file must already have been read before the containing script is read. The error messages will remind the user of this if a problem arises.

A file should always end with the line `quit`.

5 Distinctive features of our approach, and vague philosophical speculations

The general fact that mathematical reasoning and construction of mathematical objects can be managed using a type checking system of this kind is already well known, from work with other systems (Automath, Coq and their relations), and examples of constructions of both kinds under Lestrade appear in section 6.

There are some distinctive features of our approach.

One feature which the reader and the operator of Lestrade will notice is that the user never has to write any construction sort and does not have to write λ-terms (though he or she now can do this). All constructions which are introduced may be assigned names, as if we introduced all constructions in the style $f(x) = x^2 + 1$ instead of the style $f = (\lambda x.x^2 + 1)$ or $f = (x \mapsto x^2 + 1)$, though it is now possible to write construction arguments of Lestrade terms as anonymous λ-terms. I thought originally that no λ-terms were needed at all, but this is not the case: it is easy to get into a situation where the name of a construction passes out of scope when a move is closed, but the construction appears as an argument in a sort, and so is expanded to a λ-term. An construction can also appear in applied position and pass out of scope, in which case β-reduction will occur automatically: a λ-term can only appear as an argument, never in applied position, and this continues to be true now that the user can enter their own λ-terms.

I have found it interesting working on a style in which constructions which might appear anonymously as scopes of variable binding constructions have to be explicitly set up and assigned names in advance, though as I observed above user-entered λ-terms are now supported. The need to use a lot of names is limited by the fact that identifiers are regularly freed up when moves are closed. Fluent handling of construction arguments may be improved by the recent introduction of constructions applied to shortened argument lists, interpreted as complex construction terms via currying, as well as the latest innovation of user-entered λ-terms. It remains the case that an atomic construction term cannot be declared or defined except schematically via its values: this reflects a view that the real first-class citizens of a Lestrade world are its objects.

There is a limitation of my type system. The output sorts of all constructions are object sorts (this is superficially modified by the new device

of curried construction arguments). This I have no intention of changing: I take the view that constructions are not *prima facie* first-class objects, and attempts to convert them into objects (first-class objects) must involve postulation of suitable constructions by the user. For example, the following theory introduces a very powerful ability to code constructions using objects, which leads to Russell's paradox. The approach I take to implementing type theory of sets in the previous example has some similarities.

```
begin Lestrade execution

   >>> Clearall


   {move 1}

   >>> open


      {move 2}

      >>> declare x obj


      x : obj


      {move 2}

      >>> postulate P x : prop


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


      {move 1}

      >>> close


   {move 1}
end Lestrade execution
```

The constructor set is postulated to cast predicates of type obj to objects
of type obj. The fact that postulating an object to correspond to each
predicate is a nontrivial logical move is concealed in the intuitive argument

71

for Russell's "paradox" and is in my view the reason why it is a mistake, not a paradox. The mere existence of set is not enough to derive the paradox: more dubious assumptions must be made, and duly will be!

```
begin Lestrade execution

   >>> postulate set P : obj


   set : [(P_1 : [(x_2 : obj) => (---
           : prop)]) => (--- : obj)]


   {move 0}

   >>> declare x obj


   x : obj


   {move 1}

   >>> declare y obj


   y : obj


   {move 1}
end Lestrade execution
```

The membership relation is postulated.

```
begin Lestrade execution
```

```
>>> postulate E x y : prop


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



{move 0}
end Lestrade execution
```

Here is the crux of the mistake. We postulate the comprehension axioms, the flat-footed assertion of a one-to-one correspondence, for each predicate P, between evidence for $P(x)$ and evidence for $x \in \{x : P(x)\}$.

```
begin Lestrade execution

   >>> declare x1 that P x


   x1 : that P (x)


   {move 1}

   >>> postulate comp P, x x1 : that E x set \
       P


   comp : [(P_1 : [(x_2 : obj) => (---
           : prop)]), (x_1 : obj), (x1_1
         : that P_1 (x_1)) => (--- : that
         x_1 E set (P_1))]


   {move 0}

   >>> declare x2 that E x set P
```

```
   x2 : that x E set (P)


   {move 1}

   >>> postulate comp2 P, x x2 : that P x


   comp2 : [(P_1 : [(x_2 : obj) => (---
          : prop)]), (x_1 : obj), (x2_1
       : that x_1 E set (P_1)) => (---
       : that P_1 (x_1))]


   {move 0}

   >>> declare p prop


   p : prop


   {move 1}

   >>> declare q prop


   q : prop


   {move 1}
end Lestrade execution
```

To complete the execution of our folly, we need to declare the logical operations of implication and negation in a familiar way. One should note that these declarations are fine from a constructive standpoint.

begin Lestrade execution

>>> postulate Implies p q : prop

Implies : [(p_1 : prop), (q_1 : prop) =>
    (--- : prop)]

{move 0}

>>> postulate False : prop

False : prop

{move 0}

>>> declare pp that p

pp : that p

{move 1}

>>> declare rr that Implies p q

rr : that p Implies q

{move 1}

>>> postulate Mp p q pp rr : that q

```
Mp : [(p_1 : prop), (q_1 : prop), (pp_1
    : that p_1), (rr_1 : that p_1 Implies
    q_1) => (--- : that q_1)]
```

{move 0}

>>> declare absurd that False

absurd : that False

{move 1}

>>> postulate Panic p absurd : that p

```
Panic : [(p_1 : prop), (absurd_1
    : that False) => (--- : that p_1)]
```

{move 0}

>>> define Not p : Implies p False

```
Not : [(p_1 : prop) =>
    ({def} p_1 Implies False : prop)]
```

```
Not : [(p_1 : prop) => (--- : prop)]
```

{move 0}

>>> open

```
    {move 2}

    >>> declare pp2 that p


    pp2 : that p


    {move 2}

    >>> postulate Ded pp2 : that q


    Ded : [(pp2_1 : that p) => (---
        : that q)]


    {move 1}

    >>> close


{move 1}

>>> postulate Impliesproof p q Ded : that \
    Implies p q


Impliesproof : [(p_1 : prop), (q_1
    : prop), (Ded_1 : [(pp2_2 : that
      p_1) => (--- : that q_1)]) =>
    (--- : that p_1 Implies q_1)]


{move 0}
end Lestrade execution
```

Russell is the Russell predicate $(\lambda x : x \notin x\}$ while R is the Russell (paradoxical) set. The argument which follows is familiar. R6 is a proof of the False in move 0, a disaster.

```
begin Lestrade execution

   >>> define Russell x : Not E x x


   Russell : [(x_1 : obj) =>
       ({def} Not (x_1 E x_1) : prop)]


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


   {move 0}

   >>> open


      {move 2}

      >>> define R : set Russell


      R : obj


      {move 1}

      >>> declare R1 that E set Russell, set \
          Russell


      R1 : that set (Russell) E set (Russell)
```

```
{move 2}

>>> define R2 R1 : comp2 Russell, set \
    Russell, R1


R2 : [(R1_1 : that set (Russell) E set
    (Russell)) => (--- : that Russell
    (set (Russell)))]


{move 1}

>>> define R3 R1 : Mp E set Russell, set \
    Russell, False R1 R2 R1


R3 : [(R1_1 : that set (Russell) E set
    (Russell)) => (--- : that False)]


{move 1}

>>> close


{move 1}

>>> define R4 : Impliesproof E set Russell, set \
    Russell, False R3


R4 : [
    ({def} Impliesproof (set (Russell) E set
    (Russell), False, [(R1_2 : that
        set (Russell) E set (Russell)) =>
        ({def} Mp (set (Russell) E set
```
79

```
      (Russell), False, R1_2, comp2
      (Russell, set (Russell), R1_2)) : that
      False)]) : that (set (Russell) E set
  (Russell)) Implies False)]
```

```
R4 : that (set (Russell) E set (Russell)) Implies
 False
```

{move 0}

>>> define R5 : comp Russell, set Russell, R4

```
R5 : [
    ({def} comp (Russell, set (Russell), R4) : that
    set (Russell) E set (Russell))]
```

```
R5 : that set (Russell) E set (Russell)
```

{move 0}

>>> define R6 : Mp E set Russell, set \
    Russell, False R5 R4

```
R6 : [
    ({def} Mp (set (Russell) E set
    (Russell), False, R5, R4) : that
    False)]
```

```
R6 : that False
```

```
{move 0}
end Lestrade execution
```

An important thing to notice about the crucial last few lines of the argument is that Lestrade is automatically expanding definitions as it type checks these lines. I should add comments on each line pointing out the definitional expansions.

I hope you enjoyed that!

Usually the proofs of implications are identified with actual functions under the Curry-Howard isomorphism. My requirement that user defined constructions have object output requires that objects `Ifproof p q Ded` are obtained by a user-declared construction from constructions `Ded`, not identified with constructions. This is useful. There is no need for proofs of implications $p \rightarrow q$ to have the same identity conditions as constructions from proofs of p to proofs of q, and one can create situations where there are "too many" distinct proofs if the proofs are actually constructions. [I seem to recall that universal quantifiers over `prop` lead to some weirdness.]

This goes along with the idea that treating constructions as objects requires one to declare constructions that effect this reduction. This casts a kind of light on reasons behind the "paradoxes of set theory".

Another limitation of the type system is that there are no disjoint union types or existential types "built in" (though they certainly can be declared). In this Lestrade is similar to Automath (of which it is arguably a flavor). Essentially the only built in type constructor of Lestrade is the dependent type construction of constructions, of which the usual Curry-Howard implementations of implication and universal quantification are examples. This gives the implementations of disjunction and the existential quantifier as logical operations an indirect character [much as in Automath].

We have no constructive prejudices, though we observe that Lestrade supports constructive logic perfectly well with some modifications of the basic declarations of logical operations. We are interested in a more constructive approach for other reasons: we would like to make an obvious further extension and make Lestrade a programming environment in which programs can be written which will operate effectively on the wide range of mathematical objects which its rich [and easily user-extendible!] system of sorts makes accessible. The rewriting features added recently go some way toward implementing this.

Another class of philosophical objections to classical mathematics is addressed by our approach here. We have no sympathy with predicativist scruples and we are very fond of second-order logic (a logic which supports quantification over universals). One should note in the Lestrade book that we showed how to quantify over untyped objects, then how to quantify uniformly over any sort (**in** τ), then how to quantify universally over binary relations (constructions!) from natural numbers to an arbitrary type. This last is an example of second order quantification. We note that it appears that predicativist scruples correspond in the Lestrade context to an unwillingness to generalize over construction variables.

On the other hand, Lestrade does not automatically commit to the ability to quantify over any type, as Automath does, at least in later versions, due to the fact that constructions with output in `prop` or `type` are also typed as instances of `prop` or `type` respectively. It appears to be possible to restrict oneself to the resources of first-order reasoning by suitable choice of primitives.

The induction axiom on the natural number type defined in the book works on predicates of natural numbers defined in any way that any extension of the book might provide for. We argue that the intended referent of this type is the true type of natural numbers with a second-order axiomatization. We intend to extend the book to present an axiomatization of Peano arithmetic for contrast, for which a Lestrade specification of first-order logic formulas over the Peano naturals would have to be given, and induction provided only for properties of the Peano naturals expressible by such formulas. One could then postulate other predicates of the Peano naturals which did not respect this induction axiom, thus supporting reasoning about "nonstandard" natural numbers which would not be possible for the type of natural numbers currently described.

Similarly, I would like to present first-order versus second-order Zermelo set theory and ZF.

Predicativist scruples seem to us to arise from a disagreement about the nature of generality. For us, a construction is not an infinite table of values. I do not need to be acquainted with every natural number n and compute $n^2 + 1$ for each of them to be acquainted with the construction $f(n) = n^2 + 1$: I have to know the method of computation. I do not need to know every element of the domain D to be able to prove $(\forall x \in D : \phi(x))$: I need a construction which given an $x \in D$ will generate a proof of $\phi(x)$ (note the dependent typing), and this construction may be a finite object in the same

sense that the expression $n^2 + 1$ is.

I do not regard the notion "set of natural numbers" as vague because I do not have a way of effectively listing all sets of natural numbers. A set of natural numbers is a gadget which given a natural number input gives a propositional output: I can recognize such an object without having a clue as to how many such objects there are. I do not need any familiarity with the full extent of the domain of sets of natural numbers to be able to prove a universal statement about it. In particular, if I am given zero and a successor operation in a domain which may be supposed to properly include (an implementation of) the natural numbers, I can present a uniform proof that 3 (defined in the obvous way) belongs to every inductive set as a "finite gadget". And I can abstract from this to the notion that 3 is a natural number, defined in the usual impredicative way. I do not regard universal quantifications as infinitary conjunctions which require for their understanding that $\phi(x)$ be previously understood for each $x \in D$: such an understanding does make impredicative definitions circular. But note that it is obvious that we cannot possibly be thinking that way about such sentences in practice: our understanding of universal quantification, which makes reasoning about it clearly possible as a finite act, also dispels the apparently problematic character of impredicative reasoning (without removing the signs that impredicative definition is a very powerful move).

Another assumption which seems to underly the predicativist view is that all constructions should be definable. We make no such assumption: as yet unknown constructions are not presumed to be constructed in any definable way. Inductive arguments on the structure of our current resources for defining constructions are not expected to capture features of all possible constructions.

I'm well aware that philosophical speculations can be apparently vague and unsatisfactory. One of our aims in designing Lestrade is to create an environment in which one has as it were hands-on access to the full range of mathematical constructions, in the incontrovertible sense that one can actually design the objects and execute proofs of theorems about them. An environment in which very general mathematical objects can be manipulated precisely should be an aid to philosophical contemplation of them. The claim that this environment does so implement the mathematical objects and proofs is itself a definite philosophical claim, and interaction with the software should make it easier to understand and evaluate this claim.

6 A moderately extensive Lestrade book

To see how this works, we present a moderately extensive development of some basic logical and mathematical concepts in Lestrade. The considerations in section 2 should give a general idea of what is going on: details of syntax and command format in Lestrade are given in section 4, and one may look forward to that point to see details of how the book is to be parsed and executed.

This can be compared with text produced under Automath as in [6] and [14]. The development of arithmetic by Jutting in [14] (following Landau's classic [15]) is available from Freek Wiedijk at the same place as his Automath implementation, [8].

This book makes no use of the new implicit argument feature. I ought to introduce some declarations with implicit arguments and examples of their use (there is a short section of this kind above).

6.1 Propositional logic of conjunction and implication

```
begin Lestrade execution

    >>> Clearall


    {move 1}

    >>> declare p prop


    p : prop


    {move 1}

    >>> declare q prop


    q : prop
```

```
{move 1}

>>> declare pp that p


pp : that p


{move 1}

>>> declare qq that q


qq : that q


{move 1}
end Lestrade execution
```

We declare the conjunction operation on propositions.

```
begin Lestrade execution

>>> comment Declare the conjunction operator


{move 1}

>>> postulate & p q : prop


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

```
    {move 0}
end Lestrade execution
```

We present the rule of conjunction introduction as a mathematical object, taking the two propositions and proofs of each of them as arguments and returning a proof of the conjunction.

Notice that while we always declare a construction with two arguments using prefix notation, it will be displayed in infix notation (as long as its first argument is not a construction), as conjunction is here. The parser can handle mixfix notation for constructions with three or more arguments but Lestrade does not choose to display things in this way.

```
begin Lestrade execution

    >>> comment The rule of conjunction


    {move 1}

    >>> postulate Andproof p q pp qq : that \
        p & q


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


    {move 0}

    >>> declare rr that p & q


    rr : that p & q


    {move 1}
end Lestrade execution
```

Similarly, we present the rules of conjunction elimination (simplication) as mathematical objects.

```
begin Lestrade execution

   >>> comment The rules of simplification


   {move 1}

   >>> postulate And1 p q rr : that p


   And1 : [(p_1 : prop), (q_1 : prop), (rr_1
       : that p_1 & q_1) => (--- : that
       p_1)]


   {move 0}

   >>> postulate And2 p q rr : that q


   And2 : [(p_1 : prop), (q_1 : prop), (rr_1
       : that p_1 & q_1) => (--- : that
       q_1)]


   {move 0}
end Lestrade execution
```

We declare implication just as we declared conjunction.

```
begin Lestrade execution

   >>> comment The implication operator
```

```
{move 1}

>>> postulate -> p q : prop


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


{move 0}
end Lestrade execution
```

We develop the rule of conditional proof (the deduction theorem) as a mathematical object. This is more exciting, because one of its inputs is a construction.

```
begin Lestrade execution

>>> comment Development of conditional \
    proof


{move 1}

>>> open


  {move 2}

  >>> declare pp2 that p


  pp2 : that p
```

```
    {move 2}

    >>> comment Ded below does not need \
        p or q in its argument list


    {move 2}

    >>> comment because they are not locally \
        variables .


    {move 2}

    >>> postulate Ded pp2 : that q


    Ded : [(pp2_1 : that p) => (---
        : that q)]


    {move 1}

    >>> close


{move 1}

>>> comment Note that once the move at \
    which Ded was constructed closes,


{move 1}

>>> comment it is a variable of desirable \
    construction type
```

89

```
    {move 1}

    >>> postulate Ifproof p q Ded : that p -> \
        q


    Ifproof : [(p_1 : prop), (q_1 : prop), (Ded_1
        : [(pp2_2 : that p_1) => (--- : that
            q_1)]) => (--- : that p_1 ->
        q_1)]


    {move 0}
end Lestrade execution
```

We demonstrate our powers by actually proving a theorem $(P \to P)$.

```
begin Lestrade execution

    >>> comment Now, for fun, we will postulate \
        an actual proof


    {move 1}

    >>> open


        {move 2}

        >>> declare pp2 that p


        pp2 : that p


        {move 2}
```

```
>>> define Ppid pp2 : pp2


Ppid : [(pp2_1 : that p) => (---
      : that p)]


{move 1}

>>> close


{move 1}

>>> define Selfimp p : Ifproof p p Ppid


Selfimp : [(p_1 : prop) =>
    ({def} Ifproof (p_1, p_1, [(pp2_2
        : that p_1) =>
        ({def} pp2_2 : that p_1)]) : that
    p_1 -> p_1)]


Selfimp : [(p_1 : prop) => (--- : that
    p_1 -> p_1)]


{move 0}

>>> comment Notice in the sort of Selfimp \
    that Ppid has


{move 1}

>>> comment been expanded as a lambda \
```

```
        - term .


    {move 1}

    >>> comment Develop the rule of modus \
        ponens


    {move 1}

    >>> declare ss that p -> q


    ss : that p -> q


    {move 1}
end Lestrade execution
```

We complete the basics of implication by defining the rule of modus ponens as a mathematical object.

```
begin Lestrade execution

    >>> postulate Mp p q pp ss : that q


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


    {move 0}
end Lestrade execution
```

6.2 The universal quantifier

In this section we do the basic development of the universal quantifier. After this we will return to propositional logic and after that to a little more quantification.

The argument of the universal quantifier is a construction, which requires setup here similar to that for the conditional proof rule above.

```
begin Lestrade execution

    >>> comment Opening an environment to \
        set up definition of a predicate variable \
        P


    {move 1}

    >>> open


        {move 2}

        >>> declare xx obj


        xx : obj


        {move 2}

        >>> postulate P xx : prop


        P : [(xx_1 : obj) => (--- : prop)]


        {move 1}
```

```
      >>> close


   {move 1}

   >>> comment Declaring the universal quantifier


   {move 1}

   >>> postulate Forall P : prop


   Forall : [(P_1 : [(xx_2 : obj) =>
         (--- : prop)]) => (--- : prop)]


   {move 0}
end Lestrade execution
```

We declare the rule of universal instantiation. It is worth noting that the parser sometimes requires us to guard constructions appearing as arguments with commas so that they will not be applied to what follows them (or what is before them if they are capable of being read as infix or mixifx operators).

```
begin Lestrade execution

   >>> comment Declaring the rule U I of \
       universal instantiation


   {move 1}

   >>> declare P2 that Forall P
```

```
P2 : that Forall (P)


{move 1}

>>> declare x obj


x : obj


{move 1}

>>> postulate Ui P, P2 x : that P x


Ui : [(P_1 : [(xx_2 : obj) => (---
        : prop)]), (P2_1 : that Forall
     (P_1)), (x_1 : obj) => (---
     : that P_1 (x_1))]


{move 0}

>>> comment Note in the previous line \
    that we follow P


{move 1}

>>> comment with a comma : a construction \
    argument may need to be


{move 1}

>>> comment guarded with commas so it \
    will not be read as applied .
```

```
{move 1}

>>> comment Opening an environment to \
    declare a construction


{move 1}

>>> comment that witnesses provability \
    of a universal statement



{move 1}
end Lestrade execution
```

We declare the rule of universal quantifier introduction. Note that the second argument is a construction which takes an object x to a proof of $P(x)$, a dependently typed construction witnessing the truth of the universal statement.

```
begin Lestrade execution

>>> open


   {move 2}

>>> declare u obj


   u : obj


   {move 2}
```

```
>>> postulate Q2 u : that P u


Q2 : [(u_1 : obj) => (--- : that
    P (u_1))]


{move 1}

>>> close


{move 1}

>>> comment The rule of universal generalization


{move 1}

>>> postulate Ug P, Q2 : that Forall \
    P


Ug : [(P_1 : [(xx_2 : obj) => (---
      : prop)]), (Q2_1 : [(u_2
      : obj) => (--- : that P_1 (u_2))]) =>
    (--- : that Forall (P_1))]


{move 0}
end Lestrade execution
```

6.3   Negation (made classical) interacts with implication: proof of the contrapositive theorem

In this section we introduce $\perp$ (??), a false statement, as a primitive, then introduce logical negation and the biconditional as defined notions. We then

prove the contrapositive theorem and develop derived rules relating impli-
cation and negation. We make our logic classical by declaring the rule of
double negation, but a constructive approach is certainly possible.

```
begin Lestrade execution

    >>> comment Develop rules for negation \
        (which will be classical !)


    {move 1}

    >>> comment and prove the contrapositive \
        theorem .


    {move 1}

    >>> comment The absurd proposition .


    {move 1}

    >>> postulate ?? : prop


    ?? : prop


    {move 0}

    >>> comment The negation operation .


    {move 1}

    >>> define ~ p : p -> ??
```

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


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


{move 0}

>>> define Existsdef P : ~ Forall [x => \
        ~ (P x)]


Existsdef : [(P_1 : [(xx_2 : obj) =>
        (--- : prop)]) =>
    ({def} ~ (Forall ([(x_3 : obj) =>
        ({def} ~ (P_1 (x_3)) : prop)])) : prop)]


Existsdef : [(P_1 : [(xx_2 : obj) =>
        (--- : prop)]) => (--- : prop)]


{move 0}
end Lestrade execution
```

(The declaration of Existsdef is a test of the new ability to parse terms with bound variables).

Here we introduce the primitive that makes our logic classical.


```
begin Lestrade execution

>>> comment We make our logic classical \
    : the rule of double negation
```

```
{move 1}

>>> declare maybe that ~ ~ p


maybe : that ~ (~ (p))


{move 1}

>>> postulate Dneg p maybe : that p


Dneg : [(p_1 : prop), (maybe_1 : that
     ~ (~ (p_1))) => (--- : that p_1)]


{move 0}
end Lestrade execution
```

Here we show that contradictions in the usual sense of the term imply
the absurd primitive. It is worth noting that Lestrade does recognize that a
negative statement has the form of an implication and applies modus ponens
without any need for special action to unpack the definition of negation.


```
begin Lestrade execution

>>> comment Contradictions are absurd \
     .


{move 1}

>>> declare nn that ~ p
```

```
    nn : that ~ (p)


    {move 1}

    >>> define Contra p pp nn : Mp p ?? pp \
        nn


    Contra : [(p_1 : prop), (pp_1 : that
        p_1), (nn_1 : that ~ (p_1)) =>
        ({def} Mp (p_1, ??, pp_1, nn_1) : that
        ??)]


    Contra : [(p_1 : prop), (pp_1 : that
        p_1), (nn_1 : that ~ (p_1)) =>
        (--- : that ??)]


    {move 0}

    >>> comment Notice that Lestrade does \
        expand the definition


    {move 1}

    >>> comment of the negation operation \
        as we expect .



    {move 1}
end Lestrade execution
```

We start the development of the rule of negation introduction. We have
to do a little extra work, because the most direct approach gives us a rule in
which the output is typed in expanded form as an implication. But this can

be fixed.

begin Lestrade execution

    >>> open

        {move 2}

        >>> declare pp2 that p

        pp2 : that p

        {move 2}

        >>> postulate Negded pp2 : that ??

        Negded : [(pp2_1 : that p) => (---
            : that ??)]

        {move 1}

        >>> close

    {move 1}

    >>> define Negintro1 p Negded : Ifproof \
        p ?? Negded

    Negintro1 : [(p_1 : prop), (Negded_1
        : [(pp2_2 : that p_1) => (--- : that

```
        ??)]) =>
    ({def} Ifproof (p_1, ??, Negded_1) : that
    p_1 -> ??)]


Negintro1 : [(p_1 : prop), (Negded_1
    : [(pp2_2 : that p_1) => (--- : that
        ??)]) => (--- : that p_1 ->
    ??)]


{move 0}

>>> comment Negation introduction .But \
    it is defective in actually


{move 1}

>>> comment reporting an implication type \
    .Let' s see if we can fix this .


{move 1}

>>> open


    {move 2}

    >>> declare proof1 that p -> ??


    proof1 : that p -> ??


    {move 2}
```

103

```
>>> define Negproofid proof1 : proof1


Negproofid : [(proof1_1 : that p ->
    ??) => (--- : that p -> ??)]



{move 1}

>>> close


{move 1}

>>> define Negfix p : Ifproof ((p -> \
    ??), ~ p, Negproofid)


Negfix : [(p_1 : prop) =>
    ({def} Ifproof (p_1 -> ??, ~ (p_1), [(proof1_2
        : that p_1 -> ??) =>
        ({def} proof1_2 : that p_1 -> ??)]) : that
    (p_1 -> ??) -> ~ (p_1))]



Negfix : [(p_1 : prop) => (--- : that
    (p_1 -> ??) -> ~ (p_1))]


{move 0}

>>> define Negintro p Negded : Mp ((p -> \
    ??), ~ p, Negintro1 p Negded, Negfix \
    p)


Negintro : [(p_1 : prop), (Negded_1
    : [(pp2_2 : that p_1) => (--- : that
```

104

```
          ??)]) =>
       ({def} Mp (p_1 -> ??, ~ (p_1), p_1
       Negintro1 Negded_1, Negfix (p_1)) : that
       ~ (p_1))]


   Negintro : [(p_1 : prop), (Negded_1
       : [(pp2_2 : that p_1) => (--- : that
          ??)]) => (--- : that ~ (p_1))]



   {move 0}

   >>> comment I succeed in defining a proper \
       negation introduction rule


   {move 1}

   >>> comment using the defined symbol .This \
       is important because of limitations


   {move 1}

   >>> comment of circumstances under which \
       Lestrade expands definitions .



   {move 1}
end Lestrade execution
```

We define the biconditional and introduce its rules. Of course since it is a defined operation we do not need to declare any new primitives in this connection.


```
begin Lestrade execution
```

```
>>> comment We define the biconditional \
    .


{move 1}

>>> define <-> p q : (p -> q) & (q -> \
    p)


<-> : [(p_1 : prop), (q_1 : prop) =>
    ({def} (p_1 -> q_1) & q_1 -> p_1
    : prop)]


<-> : [(p_1 : prop), (q_1 : prop) =>
    (--- : prop)]


{move 0}
end Lestrade execution
```

The biconditional elimination rules are variations of modus ponens.

```
begin Lestrade execution

    >>> comment Biconditional elimination \
        rules


    {move 1}

    >>> declare tt that p <-> q


    tt : that p <-> q
```

```
{move 1}

>>> define Mpb1 p q pp tt : Mp p q pp, And1 \
    ((p -> q), (q -> p), tt)


Mpb1 : [(p_1 : prop), (q_1 : prop), (pp_1
    : that p_1), (tt_1 : that p_1 <->
    q_1) =>
    ({def} Mp (p_1, q_1, pp_1, And1
    (p_1 -> q_1, q_1 -> p_1, tt_1)) : that
    q_1)]


Mpb1 : [(p_1 : prop), (q_1 : prop), (pp_1
    : that p_1), (tt_1 : that p_1 <->
    q_1) => (--- : that q_1)]


{move 0}

>>> define Mpb2 p q qq tt : Mp q p qq, And2 \
    ((p -> q), (q -> p), tt)


Mpb2 : [(p_1 : prop), (q_1 : prop), (qq_1
    : that q_1), (tt_1 : that p_1 <->
    q_1) =>
    ({def} Mp (q_1, p_1, qq_1, And2
    (p_1 -> q_1, q_1 -> p_1, tt_1)) : that
    p_1)]


Mpb2 : [(p_1 : prop), (q_1 : prop), (qq_1
    : that q_1), (tt_1 : that p_1 <->
    q_1) => (--- : that p_1)]
```

107

```
{move 0}

>>> comment In both of the last two commands, there \
    are subtle parser issues .


{move 1}

>>> comment Before And1, And2, the comma \
    is needed to prevent Andi


{move 1}

>>> comment from being read as an infix \
    .


{move 1}

>>> comment Because we enclose the argument \
    (p -> q) in parentheses


{move 1}

>>> comment we need to enclose the entire \
    argument list in parentheses


{move 1}

>>> comment because a parenthesis after \
    a prefixed construction is
```

```
    {move 1}

    >>> comment always interpreted as enclosing \
        an argument list,


    {move 1}

    >>> comment not a term .


    {move 1}

    >>> comment the classic Reductio ad Absurdum \
        (which is not the same as neg intro !)



    {move 1}
end Lestrade execution
```

We develop the rule of *reductio ad absurdum* (which is not the same as negation introduction, though both are carelessly called proof by contradiction) and the rule that anything can be deduced from a falsehood.


```
begin Lestrade execution

    >>> open


        {move 2}

    >>> declare aa that ~ p


        aa : that ~ (p)
```

```
    {move 2}

    >>> postulate reductioarg aa : that \
        ??


    reductioarg : [(aa_1 : that ~ (p)) =>
        (--- : that ??)]


    {move 1}

    >>> close


{move 1}

>>> define Reductio p reductioarg : Dneg \
    p (Negintro ~ p reductioarg)


Reductio : [(p_1 : prop), (reductioarg_1
    : [(aa_2 : that ~ (p_1)) => (---
        : that ??)]) =>
    ({def} p_1 Dneg ~ (p_1) Negintro
    reductioarg_1 : that p_1)]


Reductio : [(p_1 : prop), (reductioarg_1
    : [(aa_2 : that ~ (p_1)) => (---
        : that ??)]) => (--- : that
    p_1)]


{move 0}

>>> comment Everything follows from the \
    False !
```

```
{move 1}

>>> declare huh that ??

huh : that ??

{move 1}

>>> open

   {move 2}

   >>> declare negp that ~ p

   negp : that ~ (p)

   {move 2}

   >>> define panick negp : huh

   panick : [(negp_1 : that ~ (p)) =>
        (--- : that ??)]

   {move 1}

   >>> close

{move 1}
```

```
>>> define Panic p huh : Reductio p panick


Panic : [(p_1 : prop), (huh_1 : that
    ??) =>
    ({def} p_1 Reductio [(negp_2 : that
        ~ (p_1)) =>
      ({def} huh_1 : that ??)] : that
    p_1)]


Panic : [(p_1 : prop), (huh_1 : that
    ??) => (--- : that p_1)]



{move 0}
end Lestrade execution
```

We develop the biconditional introduction rule. This is similar to the rule of conditional proof. We once again have to do a little extra work to get an output which is actually typed as a biconditional rather than as a conjunction of implications.

```
begin Lestrade execution

    >>> comment We develop the biconditional \
        introduction rule .


    {move 1}

    >>> comment In this environment we postulate \
        reasoning


    {move 1}
```

```
>>> comment leading from p to q and q to \
    p


{move 1}

>>> open


    {move 2}

    >>> declare pp2 that p


    pp2 : that p


    {move 2}

    >>> postulate Ded1 pp2 : that q


    Ded1 : [(pp2_1 : that p) => (---
        : that q)]


    {move 1}

    >>> declare qq2 that q


    qq2 : that q


    {move 2}

    >>> postulate Ded2 qq2 : that p
```

```
    Ded2 : [(qq2_1 : that q) => (---
        : that p)]


    {move 1}

    >>> close


{move 1}

>>> comment Here we prove an initial version,


{move 1}

>>> comment defective in having expanded \
    output


{move 1}

>>> define Biintro1 p q, Ded1, Ded2 \
    : Andproof ((p -> q), (q -> p), Ifproof \
    p q Ded1, Ifproof q p Ded2)


Biintro1 : [(p_1 : prop), (q_1 : prop), (Ded1_1
    : [(pp2_2 : that p_1) => (--- : that
        q_1)]), (Ded2_1 : [(qq2_2
        : that q_1) => (--- : that p_1)]) =>
    ({def} Andproof (p_1 -> q_1, q_1
    -> p_1, Ifproof (p_1, q_1, Ded1_1), Ifproof
    (q_1, p_1, Ded2_1)) : that (p_1
    -> q_1) & q_1 -> p_1)]
```

```
Biintro1 : [(p_1 : prop), (q_1 : prop), (Ded1_1
    : [(pp2_2 : that p_1) => (--- : that
        q_1)]), (Ded2_1 : [(qq2_2
        : that q_1) => (--- : that p_1)]) =>
    (--- : that (p_1 -> q_1) & q_1 ->
    p_1)]
```

{move 0}

>>> open

    {move 2}

    >>> declare bb that p <-> q

    bb : that p <-> q

    {move 2}

    >>> define biid bb : bb

    biid : [(bb_1 : that p <-> q) =>
        (--- : that p <-> q)]

    {move 1}

    >>> close

{move 1}

```
>>> comment We fix the defective version \
    much as we fixed Negintro above


{move 1}

>>> define Bifix p q : Ifproof (((p -> \
    q) & (q -> p)), p <-> q, biid)


Bifix : [(p_1 : prop), (q_1 : prop) =>
    ({def} Ifproof ((p_1 -> q_1) & q_1
    -> p_1, p_1 <-> q_1, [(bb_2 : that
      p_1 <-> q_1) =>
      ({def} bb_2 : that p_1 <-> q_1)]) : that
    ((p_1 -> q_1) & q_1 -> p_1) ->
    p_1 <-> q_1)]


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


{move 0}

>>> define Biintro p q, Ded1, Ded2 : Mp \
    ((((p -> q) & (q -> p)), p <-> \
    q, Biintro1 (p, q, Ded1, Ded2), Bifix \
    p q)


Biintro : [(p_1 : prop), (q_1 : prop), (Ded1_1
    : [(pp2_2 : that p_1) => (--- : that
      q_1)]), (Ded2_1 : [(qq2_2
      : that q_1) => (--- : that p_1)]) =>
    ({def} Mp ((p_1 -> q_1) & q_1 ->
    p_1, p_1 <-> q_1, Biintro1 (p_1, q_1, Ded1_1, Ded2_1), p_1
```

116

```
    Bifix q_1) : that p_1 <-> q_1)]


Biintro : [(p_1 : prop), (q_1 : prop), (Ded1_1
    : [(pp2_2 : that p_1) => (--- : that
      q_1)]), (Ded2_1 : [(qq2_2
       : that q_1) => (--- : that p_1)]) =>
    (--- : that p_1 <-> q_1)]



    {move 0}
end Lestrade execution
```

We prove the contrapositive theorem. The proof follows the structure of the proof using my favorite natural deduction strategy for propositional logic exactly.

```
begin Lestrade execution

    >>> comment We prove the contrapositive \
        theorem,


    {move 1}

    >>> comment (p -> q) <-> (~ q <-> ~ p)


    {move 1}

    >>> open


        {move 2}

        >>> declare aa that p -> q
```

```
aa : that p -> q


{move 2}

>>> comment Our goal is to postulate \
    a proof of ~ q -> ~ p


{move 2}

>>> comment To do this, we need a construction \
    from


{move 2}

>>> comment proofs of ~ q to proofs \
    of ~ p


{move 2}

>>> open

   {move 3}

   >>> declare bb that ~ q


   bb : that ~ (q)


   {move 3}

   >>> comment Now our goal is to prove \
```

```
   ~ p .


{move 3}

>>> comment For this we need a construction \
    from


{move 3}

>>> comment proofs of p to proofs \
    of ??


{move 3}

>>> open


   {move 4}

   >>> declare cc that p


   cc : that p


   {move 4}

   >>> comment prove q by m .p .


   {move 4}

   >>> define dd cc : Mp p q cc \
       aa
```

```
    dd : [(cc_1 : that p) => (---
        : that q)]


    {move 3}

    >>> comment and we have a contradiction


    {move 4}

    >>> define ee cc : Contra q (dd \
        cc) bb


    ee : [(cc_1 : that p) => (---
        : that ??)]


    {move 3}

    >>> close


{move 3}

>>> define ff bb : Negintro p ee


ff : [(bb_1 : that ~ (q)) =>
    (--- : that ~ (p))]


{move 2}

>>> close
```

```
{move 2}

>>> define gg aa : Ifproof ((~ q), (~ p), ff)


gg : [(aa_1 : that p -> q) => (---
    : that ~ (q) -> ~ (p))]


{move 1}

>>> comment Now we need the construction \
    acting in


{move 2}

>>> comment the other direction


{move 2}

>>> declare hh that ~ q -> ~ p


hh : that ~ (q) -> ~ (p)


{move 2}

>>> comment Our goal is p -> q so we \
    want to assume p


{move 2}

>>> open
```

```
{move 3}

>>> declare ii that p


ii : that p


{move 3}

>>> comment Now our goal is q, but \
    we will


{move 3}

>>> comment actually aim for ~~ \
    q and so


{move 3}

>>> comment assume ~ q


{move 3}

>>> open


   {move 4}

   >>> declare jj that ~ q


   jj : that ~ (q)
```

```
{move 4}

>>> comment Now use modus ponens \
    to prove ~ p


{move 4}

>>> define kk jj : Mp (~ q, ~ p, jj, hh)


kk : [(jj_1 : that ~ (q)) =>
    (--- : that ~ (p))]


{move 3}

>>> comment Now we have a contradiction


{move 4}

>>> define ll jj : Contra p ii \
    kk jj


ll : [(jj_1 : that ~ (q)) =>
    (--- : that ??)]


{move 3}

>>> close


{move 3}
```

```
>>> define mm ii : Negintro (~ q, ll)


mm : [(ii_1 : that p) => (---
    : that ~ (~ (q)))]


{move 2}

>>> define nn2 ii : Dneg q mm ii


nn2 : [(ii_1 : that p) => (---
    : that q)]


{move 2}

>>> close


{move 2}

>>> define oo hh : Ifproof p q nn2


oo : [(hh_1 : that ~ (q) -> ~ (p)) =>
    (--- : that p -> q)]


{move 1}

>>> close


{move 1}
```

```
>>> define Contrapositive p q : Biintro \
    ((p -> q), (~ q -> ~ p), gg, oo)


Contrapositive : [(p_1 : prop), (q_1
    : prop) =>
    ({def} Biintro (p_1 -> q_1, ~ (q_1) ->
    ~ (p_1), [(aa_2 : that p_1 ->
      q_1) =>
      ({def} Ifproof (~ (q_1), ~ (p_1), [(bb_3
         : that ~ (q_1)) =>
         ({def} p_1 Negintro [(cc_4
            : that p_1) =>
            ({def} Contra (q_1, Mp
            (p_1, q_1, cc_4, aa_2), bb_3) : that
            ??)] : that ~ (p_1))]) : that
      ~ (q_1) -> ~ (p_1))], [(hh_2
      : that ~ (q_1) -> ~ (p_1)) =>
      ({def} Ifproof (p_1, q_1, [(ii_3
         : that p_1) =>
         ({def} q_1 Dneg ~ (q_1) Negintro
         [(jj_5 : that ~ (q_1)) =>
            ({def} Contra (p_1, ii_3, Mp
            (~ (q_1), ~ (p_1), jj_5, hh_2)) : that
            ??)] : that q_1)]) : that
      p_1 -> q_1)]) : that (p_1 ->
    q_1) <-> ~ (q_1) -> ~ (p_1))]


Contrapositive : [(p_1 : prop), (q_1
    : prop) => (--- : that (p_1 -> q_1) <->
    ~ (q_1) -> ~ (p_1))]


{move 0}

>>> comment Now is a good point to notice \
    that
```

```
    {move 1}

    >>> comment Lestrade definitely saves \
        proof objects in detail .


    {move 1}
end Lestrade execution
```

We develop the derived logical rules which mix implication and negation, modus tollens and proof by contrapositive.

```
begin Lestrade execution

    >>> comment Develop indirect proof strategies \
        for implication .


    {move 1}

    >>> comment Modus Tollens


    {move 1}

    >>> declare negc that ~ q


    negc : that ~ (q)


    {move 1}

    >>> define Mt p q ss negc : Mp (~ q, ~ p, negc, Mpb1 \
        ((p -> q), (~ q -> ~ p), ss, Contrapositive \
```

```
   p q))


Mt : [(p_1 : prop), (q_1 : prop), (ss_1
    : that p_1 -> q_1), (negc_1 : that
    ~ (q_1)) =>
    ({def} Mp (~ (q_1), ~ (p_1), negc_1, Mpb1
    (p_1 -> q_1, ~ (q_1) -> ~ (p_1), ss_1, p_1
    Contrapositive q_1)) : that ~ (p_1))]


Mt : [(p_1 : prop), (q_1 : prop), (ss_1
    : that p_1 -> q_1), (negc_1 : that
    ~ (q_1)) => (--- : that ~ (p_1))]


{move 0}

>>> comment Rule of contrapositive or \
    indirect proof


{move 1}

>>> open


   {move 2}

   >>> declare negq that ~ q


   negq : that ~ (q)


   {move 2}

   >>> postulate indarg negq : that ~ p
```

```
        indarg : [(negq_1 : that ~ (q)) =>
            (--- : that ~ (p))]


        {move 1}

        >>> close


    {move 1}

    >>> define Indirect p q indarg : Mpb2 \
        ((p -> q), (~ q -> ~ p), Ifproof \
        (~ q, ~ p, indarg), Contrapositive \
        p q)


    Indirect : [(p_1 : prop), (q_1 : prop), (indarg_1
        : [(negq_2 : that ~ (q_1)) =>
            (--- : that ~ (p_1))]) =>
        ({def} Mpb2 (p_1 -> q_1, ~ (q_1) ->
        ~ (p_1), Ifproof (~ (q_1), ~ (p_1), indarg_1), p_1
        Contrapositive q_1) : that p_1 ->
        q_1)]


    Indirect : [(p_1 : prop), (q_1 : prop), (indarg_1
        : [(negq_2 : that ~ (q_1)) =>
            (--- : that ~ (p_1))]) =>
        (--- : that p_1 -> q_1)]


    {move 0}
end Lestrade execution
```

128

6.4 The development of disjunction

We declare the disjunction operation and introduce its constructively valid
rules (addition and proof by cases) then derive the more powerful rules mixing
disjunction and negation.

```
begin Lestrade execution

   >>> comment Now start the development \
       of disjunction .


   {move 1}

   >>> comment disjunction declared


   {move 1}

   >>> postulate v p q : prop


   v : [(p_1 : prop), (q_1 : prop) =>
       (--- : prop)]


   {move 0}

   >>> comment basic disjunction introduction \
       rules (addition)


   {move 1}

   >>> postulate Addition1 p q pp : that \
       p v q
```

```
Addition1 : [(p_1 : prop), (q_1 : prop), (pp_1
    : that p_1) => (--- : that p_1 v q_1)]


{move 0}

>>> postulate Addition2 p q qq : that \
    p v q


Addition2 : [(p_1 : prop), (q_1 : prop), (qq_1
    : that q_1) => (--- : that p_1 v q_1)]


{move 0}

>>> comment the basic disjunction elimination \
    rule (proof by cases)


{move 1}

>>> declare r prop


r : prop


{move 1}

>>> declare disj that p v q


disj : that p v q


{move 1}
```

```
>>> open

   {move 2}

   >>> declare pp2 that p


   pp2 : that p


   {move 2}

   >>> postulate case1 pp2 : that r


   case1 : [(pp2_1 : that p) => (---
       : that r)]


   {move 1}

   >>> declare qq2 that q


   qq2 : that q


   {move 2}

   >>> postulate case2 qq2 : that r


   case2 : [(qq2_1 : that q) => (---
       : that r)]


   {move 1}
```

```
      >>> close


   {move 1}

   >>> postulate Cases p q r disj, case1, case2 \
       : that r


   Cases : [(p_1 : prop), (q_1 : prop), (r_1
       : prop), (disj_1 : that p_1 v q_1), (case1_1
       : [(pp2_2 : that p_1) => (--- : that
          r_1)]), (case2_1 : [(qq2_2
           : that q_1) => (--- : that r_1)]) =>
       (--- : that r_1)]


   {move 0}

   >>> comment The rule of proof by cases \
       really is quite complicated !



   {move 1}
end Lestrade execution
```

We prove the equivalences whcih support the rules mixing disjunction and negation: these are $P \vee Q \leftrightarrow \neg P \rightarrow Q$ and $P \vee Q \leftrightarrow \neg Q \rightarrow P$.


```
begin Lestrade execution

   >>> comment Prove the basic equivalence \
       theorem


   {move 1}
```

```
>>> comment which supports mixed rules \
    for disjunction


{move 1}

>>> comment The theorem is (p v q) <-> \
    (~ p -> q)


{move 1}

>>> open


   {move 2}

   >>> declare aa that p v q


   aa : that p v q


   {move 2}

   >>> comment our goal is to prove ~ p -> \
       q


   {move 2}

   >>> open


      {move 3}

      >>> declare bb that ~ p
```

```
bb : that ~ (p)


{move 3}

>>> comment prove this by cases


{move 3}

>>> open


    {move 4}

    >>> declare hyp1 that p


    hyp1 : that p


    {move 4}

    >>> declare hyp2 that q


    hyp2 : that q


    {move 4}

    >>> define casea2 hyp2 : hyp2


    casea2 : [(hyp2_1 : that q) =>
        (--- : that q)]
```

```
{move 3}

>>> open

    {move 5}

    >>> declare cc that ~ q


    cc : that ~ (q)


    {move 5}

    >>> define panic cc : Contra \
        p hyp1 bb


    panic : [(cc_1 : that ~ (q)) =>
        (--- : that ??)]


    {move 4}

    >>> close


{move 4}

>>> define casea1 hyp1 : Dneg \
    q (Negintro ~ q panic)


casea1 : [(hyp1_1 : that p) =>
    (--- : that q)]
```

135

```
        {move 3}

        >>> close


    {move 3}

    >>> define gotq bb : Cases p q q aa, casea1, casea2


    gotq : [(bb_1 : that ~ (p)) =>
        (--- : that q)]


    {move 2}

    >>> close


{move 2}

>>> define notpimpq aa : Ifproof ~ p q gotq


notpimpq : [(aa_1 : that p v q) =>
    (--- : that ~ (p) -> q)]


{move 1}

>>> declare bb that ~ p -> q


bb : that ~ (p) -> q
```

```
{move 2}

>>> open


    {move 3}

    >>> declare cc that ~ (p v q)


    cc : that ~ (p v q)


    {move 3}

    >>> comment this is a hypothesis \
        for reduction ad absurdum


    {move 3}

    >>> comment our aim is prove ~ p so \
        we can use the hypothesis bb


    {move 3}

    >>> open


        {move 4}

        >>> declare pp2 that p


        pp2 : that p
```

```
    {move 4}

    >>> define dd pp2 : Addition1 \
        p q pp2


    dd : [(pp2_1 : that p) =>
        (--- : that p v q)]


    {move 3}

    >>> define ee pp2 : Contra (p v q, dd \
        pp2, cc)


    ee : [(pp2_1 : that p) =>
        (--- : that ??)]


    {move 3}

    >>> close


{move 3}

>>> define ff cc : Negintro p ee


ff : [(cc_1 : that ~ (p v q)) =>
    (--- : that ~ (p))]


{move 2}

>>> define gg2 cc : Mp (~ p, q, ff \
    cc, bb)
```

```
      gg2 : [(cc_1 : that ~ (p v q)) =>
            (--- : that q)]


      {move 2}

      >>> define hh cc : Addition2 p q gg2 \
            cc


      hh : [(cc_1 : that ~ (p v q)) =>
            (--- : that p v q)]


      {move 2}

      >>> define ii cc : Contra (p v q, hh \
            cc, cc)


      ii : [(cc_1 : that ~ (p v q)) =>
            (--- : that ??)]


      {move 2}

      >>> close


{move 2}

>>> define jj bb : Reductio (p v q, ii)


jj : [(bb_1 : that ~ (p) -> q) =>
      (--- : that p v q)]
```

```
    {move 1}

    >>> close


{move 1}

>>> define Orthm p q : Biintro (p v q, ~ p -> \
    q, notpimpq, jj)


Orthm : [(p_1 : prop), (q_1 : prop) =>
    ({def} Biintro (p_1 v q_1, ~ (p_1) ->
    q_1, [(aa_2 : that p_1 v q_1) =>
       ({def} Ifproof (~ (p_1), q_1, [(bb_3
          : that ~ (p_1)) =>
          ({def} Cases (p_1, q_1, q_1, aa_2, [(hyp1_4
             : that p_1) =>
             ({def} q_1 Dneg ~ (q_1) Negintro
             [(cc_6 : that ~ (q_1)) =>
                ({def} Contra (p_1, hyp1_4, bb_3) : that
                ??)] : that q_1)], [(hyp2_4
             : that q_1) =>
             ({def} hyp2_4 : that q_1)]) : that
          q_1)]) : that ~ (p_1) ->
       q_1)], [(bb_2 : that ~ (p_1) ->
       q_1) =>
       ({def} (p_1 v q_1) Reductio [(cc_3
          : that ~ (p_1 v q_1)) =>
          ({def} Contra (p_1 v q_1, Addition2
          (p_1, q_1, Mp (~ (p_1), q_1, p_1
          Negintro [(pp2_7 : that p_1) =>
             ({def} Contra (p_1 v q_1, Addition1
             (p_1, q_1, pp2_7), cc_3) : that
             ??)], bb_2)), cc_3) : that
          ??)] : that p_1 v q_1)]) : that
```

140

```
    (p_1 v q_1) <-> ~ (p_1) -> q_1)]


Orthm : [(p_1 : prop), (q_1 : prop) =>
    (--- : that (p_1 v q_1) <-> ~ (p_1) ->
    q_1)]


{move 0}

>>> comment Prove the symmetric version \
    p v q <-> ~ q -> p


{move 1}

>>> open


    {move 2}

    >>> declare aa that p v q


    aa : that p v q


    {move 2}

    >>> define bb aa : Mpb1 (p v q, ~ p -> \
        q, aa, Orthm p q)


    bb : [(aa_1 : that p v q) => (---
        : that ~ (p) -> q)]


    {move 1}
```

```
>>> define cc aa : Mpb1 (~ p -> q, ~ q -> \
    ~ ~ p, bb aa, Contrapositive ~ p q)


cc : [(aa_1 : that p v q) => (---
    : that ~ (q) -> ~ (~ (p)))]


{move 1}

>>> open


   {move 3}

   >>> declare negq that ~ q


   negq : that ~ (q)


   {move 3}

   >>> define dd negq : Mp ~ q ~ ~ p negq \
       cc aa


   dd : [(negq_1 : that ~ (q)) =>
       (--- : that ~ (~ (p)))]


   {move 2}

   >>> define yesp negq : Dneg p dd \
       negq
```

```
    yesp : [(negq_1 : that ~ (q)) =>
        (--- : that p)]


    {move 2}

    >>> close


{move 2}

>>> define ee aa : Ifproof ~ q p yesp


ee : [(aa_1 : that p v q) => (---
    : that ~ (q) -> p)]


{move 1}

>>> declare ff that ~ q -> p


ff : that ~ (q) -> p


{move 2}

>>> comment Prove that ~ p implies \
    q then use Orthm


{move 2}

>>> open


    {move 3}
```

```
>>> declare negp that ~ p


negp : that ~ (p)


{move 3}

>>> comment prove q by reductio


{move 3}

>>> open


   {move 4}

   >>> declare negq that ~ q


   negq : that ~ (q)


   {move 4}

   >>> define pfollows negq : Mp \
       ~ q p negq ff


   pfollows : [(negq_1 : that
       ~ (q)) => (--- : that
       p)]


   {move 3}
```

```
>>> define disaster negq : Contra \
      p, pfollows negq negp


disaster : [(negq_1 : that
    ~ (q)) => (--- : that
    ??)]


{move 3}

>>> close


{move 3}

>>> define kk negp : Reductio q disaster


kk : [(negp_1 : that ~ (p)) =>
    (--- : that q)]


{move 2}

>>> close


{move 2}

>>> define ll ff : Ifproof ~ p q kk


ll : [(ff_1 : that ~ (q) -> p) =>
    (--- : that ~ (p) -> q)]


{move 1}
```

```
>>> define mm ff : Mpb2 (p v q, ~ p -> \
    q, ll ff, Orthm p q)


mm : [(ff_1 : that ~ (q) -> p) =>
    (--- : that p v q)]


{move 1}

>>> close


{move 1}

>>> define Orthm2 p q : Biintro (p v q, ~ q -> \
    p, ee, mm)


Orthm2 : [(p_1 : prop), (q_1 : prop) =>
    ({def} Biintro (p_1 v q_1, ~ (q_1) ->
    p_1, [(aa_2 : that p_1 v q_1) =>
        ({def} Ifproof (~ (q_1), p_1, [(negq_3
            : that ~ (q_1)) =>
            ({def} p_1 Dneg Mp (~ (q_1), ~ (~ (p_1)), negq_3, Mpb1
            (~ (p_1) -> q_1, ~ (q_1) ->
            ~ (~ (p_1)), Mpb1 (p_1
            v q_1, ~ (p_1) -> q_1, aa_2, p_1
            Orthm q_1), ~ (p_1) Contrapositive
            q_1)) : that p_1)]) : that
        ~ (q_1) -> p_1)], [(ff_2
        : that ~ (q_1) -> p_1) =>
        ({def} Mpb2 (p_1 v q_1, ~ (p_1) ->
        q_1, Ifproof (~ (p_1), q_1, [(negp_4
            : that ~ (p_1)) =>
            ({def} q_1 Reductio [(negq_5
                : that ~ (q_1)) =>
```

146

```
          ({def} Contra (p_1, Mp
          (~ (q_1), p_1, negq_5, ff_2), negp_4) : that
          ??)] : that q_1)]), p_1
      Orthm q_1) : that p_1 v q_1)]) : that
    (p_1 v q_1) <-> ~ (q_1) -> p_1)]


  Orthm2 : [(p_1 : prop), (q_1 : prop) =>
      (--- : that (p_1 v q_1) <-> ~ (q_1) ->
      p_1)]


  {move 0}
end Lestrade execution
```

We derive stronger disjunction introduction rules and the rules of disjunctive syllogism.

```
begin Lestrade execution

  >>> comment Develop the full dress disjunction \
      introduction rule


  {move 1}

  >>> comment reversal of numbering is due \
      to proving the less preferred


  {move 1}

  >>> open


    {move 2}
```

```
   >>> declare negq that ~ q


   negq : that ~ (q)


   {move 2}

   >>> postulate thusp negq : that p


   thusp : [(negq_1 : that ~ (q)) =>
       (--- : that p)]


   {move 1}

   >>> close


{move 1}

>>> define Disjintro p q thusp : Mpb2 \
    (p v q, ~ q -> p, Ifproof ~ q p thusp, Orthm2 \
    p q)


Disjintro : [(p_1 : prop), (q_1 : prop), (thusp_1
    : [(negq_2 : that ~ (q_1)) =>
       (--- : that p_1)]) =>
    ({def} Mpb2 (p_1 v q_1, ~ (q_1) ->
    p_1, Ifproof (~ (q_1), p_1, thusp_1), p_1
    Orthm2 q_1) : that p_1 v q_1)]


Disjintro : [(p_1 : prop), (q_1 : prop), (thusp_1
    : [(negq_2 : that ~ (q_1)) =>
       (--- : that p_1)]) => (---
```

```
       : that p_1 v q_1)]


{move 0}

>>> open


    {move 2}

    >>> declare negp that ~ p


    negp : that ~ (p)


    {move 2}

    >>> postulate thusq negp : that q


    thusq : [(negp_1 : that ~ (p)) =>
        (--- : that q)]


    {move 1}

    >>> close


{move 1}

>>> define Disjintro2 p q thusq : Mpb2 \
    (p v q, ~ p -> q, Ifproof ~ p q thusq, Orthm \
    p q)


Disjintro2 : [(p_1 : prop), (q_1
```

```
    : prop), (thusq_1 : [(negp_2 : that
       ~ (p_1)) => (--- : that q_1)]) =>
    ({def} Mpb2 (p_1 v q_1, ~ (p_1) ->
    q_1, Ifproof (~ (p_1), q_1, thusq_1), p_1
    Orthm q_1) : that p_1 v q_1)]


Disjintro2 : [(p_1 : prop), (q_1
    : prop), (thusq_1 : [(negp_2 : that
       ~ (p_1)) => (--- : that q_1)]) =>
    (--- : that p_1 v q_1)]


{move 0}

>>> comment Rules of disjunctive syllogism


{move 1}

>>> declare line1 that p v q


line1 : that p v q


{move 1}

>>> declare line2 that ~ q


line2 : that ~ (q)


{move 1}

>>> define Ds1 p q line1 line2 : Mp (~ q, p, line2, Mpb1 \
    (p v q, ~ q -> p, line1, Orthm2 p q))
```

```
Ds1 : [(p_1 : prop), (q_1 : prop), (line1_1
    : that p_1 v q_1), (line2_1 : that
    ~ (q_1)) =>
    ({def} Mp (~ (q_1), p_1, line2_1, Mpb1
    (p_1 v q_1, ~ (q_1) -> p_1, line1_1, p_1
    Orthm2 q_1)) : that p_1)]


Ds1 : [(p_1 : prop), (q_1 : prop), (line1_1
    : that p_1 v q_1), (line2_1 : that
    ~ (q_1)) => (--- : that p_1)]


{move 0}

>>> declare line3 that p v q


line3 : that p v q


{move 1}

>>> declare line4 that ~ p


line4 : that ~ (p)


{move 1}

>>> define Ds2 p q line3 line4 : Mp (~ p, q, line4, Mpb1 \
    (p v q, ~ p -> q, line3, Orthm p q))


Ds2 : [(p_1 : prop), (q_1 : prop), (line3_1
```

```
      : that p_1 v q_1), (line4_1 : that
      ˜ (p_1)) =>
      ({def} Mp (˜ (p_1), q_1, line4_1, Mpb1
      (p_1 v q_1, ˜ (p_1) -> q_1, line3_1, p_1
      Orthm q_1)) : that q_1)]


   Ds2 : [(p_1 : prop), (q_1 : prop), (line3_1
      : that p_1 v q_1), (line4_1 : that
      ˜ (p_1)) => (--- : that q_1)]



   {move 0}
end Lestrade execution
```

6.5   The existential quantifier and a quantifier proof

In this section we introduce the existential quantifier and its primitive rules, then prove the quantifier theorem $(\forall x : P(x) \rightarrow Q(x)) \wedge (\forall x : Q(x) \rightarrow R(x)) \rightarrow (\forall x : P(x) \rightarrow R(x))$.


```
begin Lestrade execution

   >>> comment The existential quantifier


   {move 1}

   >>> postulate Exists P : prop


   Exists : [(P_1 : [(xx_2 : obj) =>
         (--- : prop)]) => (--- : prop)]


   {move 0}
end Lestrade execution
```

The existential quantifier introduction rule (EG).

```
begin Lestrade execution

    >>> comment the rule E G (existential \
        introduction)


    {move 1}

    >>> declare ev that P x


    ev : that P (x)


    {move 1}

    >>> postulate Eg P, x ev : that Exists \
        P


    Eg : [(P_1 : [(xx_2 : obj) => (---
            : prop)]), (x_1 : obj), (ev_1
        : that P_1 (x_1)) => (--- : that
        Exists (P_1))]


    {move 0}
end Lestrade execution
```

The existential quantifier elimination rule (EI). This is rather complicated!

```
begin Lestrade execution
```

```
>>> comment the rule E I (existential \
    elimination)


{move 1}

>>> declare g prop


g : prop


{move 1}

>>> declare ex that Exists P


ex : that Exists (P)


{move 1}

>>> open


   {move 2}

   >>> declare w obj


   w : obj


   {move 2}

   >>> declare ev2 that P w
```

```
    ev2 : that P (w)


    {move 2}

    >>> postulate wi w ev2 : that g


    wi : [(w_1 : obj), (ev2_1 : that
        P (w_1)) => (--- : that g)]


    {move 1}

    >>> close


  {move 1}

  >>> postulate Ei P, g, ex, wi : that \
      g


  Ei : [(P_1 : [(xx_2 : obj) => (---
          : prop)]), (g_1 : prop), (ex_1
      : that Exists (P_1)), (wi_1 : [(w_2
          : obj), (ev2_2 : that P_1 (w_2)) =>
          (--- : that g_1)]) => (---
      : that g_1)]


  {move 0}
end Lestrade execution
```

The proof of $(\forall x : P(x) \to Q(x)) \land (\forall x : Q(x) \to R(x)) \to (\forall x : P(x) \to R(x))$. Notice that while we never write quantified statements with variable binding constructions explicit, they do actually appear in the display of the final result, because the identifiers used to specify the component construc-

tions pass out of scope.

```
begin Lestrade execution

    >>> comment A quantifier proof


    {move 1}

    >>> open


        {move 2}

        >>> declare xx obj


        xx : obj


        {move 2}

        >>> postulate Pp xx : prop


        Pp : [(xx_1 : obj) => (--- : prop)]


        {move 1}

        >>> postulate Qq xx : prop


        Qq : [(xx_1 : obj) => (--- : prop)]


        {move 1}
```

```
>>> postulate Rr xx : prop


Rr : [(xx_1 : obj) => (--- : prop)]


{move 1}

>>> define Ss xx : (Pp xx) -> (Qq \
    xx)


Ss : [(xx_1 : obj) => (--- : prop)]


{move 1}

>>> define Tt xx : (Qq xx) -> (Rr \
    xx)


Tt : [(xx_1 : obj) => (--- : prop)]


{move 1}

>>> define Uu xx : (Pp xx) -> (Rr \
    xx)


Uu : [(xx_1 : obj) => (--- : prop)]


{move 1}

>>> declare ss2 that Forall Ss
```

ss2 : that Forall (Ss)

{move 2}

>>> declare tt2 that Forall Tt

tt2 : that Forall (Tt)

{move 2}

>>> comment Our goal is to prove Forall \
    Uu

{move 2}

>>> open

   {move 3}

   >>> declare yy obj

   yy : obj

   {move 3}

   >>> comment Our goal is to show \
       (Pp yy) -> (Rr yy)

   {move 3}

```
>>> open

   {move 4}

   >>> declare ppyy that Pp yy


   ppyy : that Pp (yy)


   {move 4}

   >>> define imp1 : Ui Ss, ss2 \
       yy


   imp1 : that Ss (yy)


   {move 3}

   >>> define line5 ppyy : Mp (Pp \
       yy, Qq yy, ppyy, imp1)


   line5 : [(ppyy_1 : that Pp
       (yy)) => (--- : that Qq
       (yy))]


   {move 3}

   >>> define imp2 : Ui Tt, tt2 \
       yy
```

```
    imp2 : that Tt (yy)


    {move 3}

    >>> define line6 ppyy : Mp (Qq \
        yy, Rr yy, line5 ppyy, imp2)


    line6 : [(ppyy_1 : that Pp
        (yy)) => (--- : that Rr
        (yy))]


    {move 3}

    >>> close


{move 3}

>>> define line7 yy : Ifproof (Pp \
    yy, Rr yy, line6)


line7 : [(yy_1 : obj) => (---
    : that Pp (yy_1) -> Rr (yy_1))]


{move 2}

>>> close


{move 2}

>>> define Univimp1 ss2 tt2 : Ug Uu, line7
```

```
    Univimp1 : [(ss2_1 : that Forall
        (Ss)), (tt2_1 : that Forall
        (Tt)) => (--- : that Forall
        (Uu))]


    {move 1}

    >>> declare conj1 that (Forall Ss) & (Forall \
        Tt)


    conj1 : that Forall (Ss) & Forall
     (Tt)


    {move 2}

    >>> define Univimp2 conj1 : Univimp1 \
        (And1 (Forall Ss, Forall Tt, conj1), And2 \
        (Forall Ss, Forall Tt, conj1))


    Univimp2 : [(conj1_1 : that Forall
        (Ss) & Forall (Tt)) => (---
        : that Forall (Uu))]


    {move 1}

    >>> close


{move 1}

>>> define Univimp Pp, Qq, Rr : Ifproof \
    ((Forall Ss) & (Forall Tt), Forall \
```

Uu, Univimp2)

Univimp : [(Pp_1 : [(xx_2 : obj) =>
      (--- : prop)]), (Qq_1 : [(xx_2
      : obj) => (--- : prop)]), (Rr_1
    : [(xx_2 : obj) => (--- : prop)]) =>
    ({def} Ifproof (Forall ([(xx_4
      : obj) =>
      ({def} Pp_1 (xx_4) -> Qq_1 (xx_4) : prop)]) & Forall
    ([(xx_4 : obj) =>
      ({def} Qq_1 (xx_4) -> Rr_1 (xx_4) : prop)]), Forall
    ([(xx_3 : obj) =>
      ({def} Pp_1 (xx_3) -> Rr_1 (xx_3) : prop)]), [(conj1_2
      : that Forall ([(xx_5 : obj) =>
        ({def} Pp_1 (xx_5) -> Qq_1
        (xx_5) : prop)]) & Forall
      ([(xx_5 : obj) =>
        ({def} Qq_1 (xx_5) -> Rr_1
        (xx_5) : prop)])) =>
    ({def} Ug ([(xx_3 : obj) =>
        ({def} Pp_1 (xx_3) -> Rr_1
        (xx_3) : prop)], [(yy_3
        : obj) =>
        ({def} Ifproof (Pp_1 (yy_3), Rr_1
        (yy_3), [(ppyy_4 : that
          Pp_1 (yy_3)) =>
          ({def} Mp (Qq_1 (yy_3), Rr_1
          (yy_3), Mp (Pp_1 (yy_3), Qq_1
          (yy_3), ppyy_4, Ui ([(xx_7
            : obj) =>
            ({def} Pp_1 (xx_7) ->
            Qq_1 (xx_7) : prop)], And1
          (Forall ([(xx_9 : obj) =>
            ({def} Pp_1 (xx_9) ->
            Qq_1 (xx_9) : prop)]), Forall
          ([(xx_9 : obj) =>
            ({def} Qq_1 (xx_9) ->

162

```
                    Rr_1 (xx_9) : prop)]), conj1_2), yy_3)), Ui
              ([(xx_6 : obj) =>
                 ({def} Qq_1 (xx_6) ->
                 Rr_1 (xx_6) : prop)], And2
              (Forall ([(xx_8 : obj) =>
                 ({def} Pp_1 (xx_8) ->
                 Qq_1 (xx_8) : prop)]), Forall
              ([(xx_8 : obj) =>
                 ({def} Qq_1 (xx_8) ->
                 Rr_1 (xx_8) : prop)]), conj1_2), yy_3)) : that
              Rr_1 (yy_3))]) : that
            Pp_1 (yy_3) -> Rr_1 (yy_3))]) : that
        Forall ([(xx_3 : obj) =>
           ({def} Pp_1 (xx_3) -> Rr_1
           (xx_3) : prop)])]) : that
     (Forall ([(xx_4 : obj) =>
        ({def} Pp_1 (xx_4) -> Qq_1 (xx_4) : prop)]) & Forall
   ([(xx_4 : obj) =>
        ({def} Qq_1 (xx_4) -> Rr_1 (xx_4) : prop)])) ->
     Forall ([(xx_3 : obj) =>
        ({def} Pp_1 (xx_3) -> Rr_1 (xx_3) : prop)])]


   Univimp : [(Pp_1 : [(xx_2 : obj) =>
        (--- : prop)]), (Qq_1 : [(xx_2
        : obj) => (--- : prop)]), (Rr_1
     : [(xx_2 : obj) => (--- : prop)]) =>
     (--- : that (Forall ([(xx_4 : obj) =>
        ({def} Pp_1 (xx_4) -> Qq_1 (xx_4) : prop)]) & Forall
   ([(xx_4 : obj) =>
        ({def} Qq_1 (xx_4) -> Rr_1 (xx_4) : prop)])) ->
     Forall ([(xx_3 : obj) =>
        ({def} Pp_1 (xx_3) -> Rr_1 (xx_3) : prop)])]


   {move 0}
end Lestrade execution
```

163

6.6 Sample declarations of theories of typed objects: natural numbers and the simple theory of types

Lestrade is not exclusively devoted to constructing proof objects. We give some very compact definitions for typed objects – natural numbers and the sets of a model of the simple typed theory of sets.

```
begin Lestrade execution

    >>> comment Declarations of typed objects


    {move 1}

    >>> comment The type of (true) natural \
        numbers .The theory of these


    {move 1}

    >>> comment objects will be second order \
        arithmetic .Peano arithmetic


    {move 1}

    >>> comment will be defined : it will \
        be instructive how hard it is to do this \
        .


    {move 1}
end Lestrade execution
```

The natural numbers as a type, the successor operation, and 1 are declared.

```
begin Lestrade execution

    >>> postulate Nat : type


    Nat : type


    {move 0}

    >>> postulate 1 : in Nat


    1 : in Nat


    {move 0}

    >>> declare n in Nat


    n : in Nat


    {move 1}

    >>> postulate Succ n : in Nat


    Succ : [(n_1 : in Nat) => (--- : in
        Nat)]


    {move 0}
end Lestrade execution
```

The declaration of mathematical induction. A serious development would
include quantifiers over the natural numbers.

```
begin Lestrade execution

    >>> open


        {move 2}

        >>> declare n2 in Nat


        n2 : in Nat


        {move 2}

        >>> postulate Pn n2 : prop


        Pn : [(n2_1 : in Nat) => (--- : prop)]


        {move 1}

        >>> close


    {move 1}

    >>> declare basis that Pn 1


    basis : that Pn (1)


    {move 1}

    >>> open
```

```
    {move 2}

    >>> declare k in Nat


    k : in Nat


    {move 2}

    >>> declare indhyp that Pn k


    indhyp : that Pn (k)


    {move 2}

    >>> postulate indstep k indhyp : that \
        Pn Succ k


    indstep : [(k_1 : in Nat), (indhyp_1
        : that Pn (k_1)) => (--- : that
        Pn (Succ (k_1)))]


    {move 1}

    >>> close


{move 1}

>>> postulate Induction n Pn, basis, indstep \
     : that Pn n
```

```
   Induction : [(n_1 : in Nat), (Pn_1
       : [(n2_2 : in Nat) => (--- : prop)]), (basis_1
       : that Pn_1 (1)), (indstep_1 : [(k_2
          : in Nat), (indhyp_2 : that Pn_1
          (k_2)) => (--- : that Pn_1 (Succ
          (k_2)))]) => (--- : that
       Pn_1 (n_1))]



   {move 0}
end Lestrade execution
```

Equality and its rules are declared.

```
begin Lestrade execution

   >>> comment We introduce the declarations \
       for the properties


   {move 1}

   >>> comment of equality of natural numbers \
       .


   {move 1}

   >>> declare m in Nat


   m : in Nat


   {move 1}
```

```
>>> declare m2 in Nat


m2 : in Nat


{move 1}

>>> postulate Eqn n m : prop


Eqn : [(n_1 : in Nat), (m_1 : in
    Nat) => (--- : prop)]


{move 0}
end Lestrade execution
```

Equality elimination (the rule of substitution)

```
begin Lestrade execution

>>> comment We develop the substitution \
    rule (equality elimination)


{move 1}

>>> declare eqev that Eqn m m2


eqev : that m Eqn m2


{move 1}
```

```
>>> declare pnpf that Pn m


pnpf : that Pn (m)


{move 1}

>>> postulate Subs Pn, m m2 eqev pnpf \
    : that Pn m2


Subs : [(Pn_1 : [(n2_2 : in Nat) =>
        (--- : prop)]), (m_1 : in
    Nat), (m2_1 : in Nat), (eqev_1
    : that m_1 Eqn m2_1), (pnpf_1 : that
    Pn_1 (m_1)) => (--- : that Pn_1
    (m2_1))]


{move 0}
end Lestrade execution
```

Equality introduction (indiscerniblity).


```
begin Lestrade execution

>>> comment We develop the equality introduction \
    rule (Leibniz)


{move 1}

>>> open


    {move 2}
```

170

```
>>> open

    {move 3}

    >>> declare n3 in Nat


    n3 : in Nat


    {move 3}

    >>> postulate Pn2 n3 : prop


    Pn2 : [(n3_1 : in Nat) => (---
        : prop)]


    {move 2}

    >>> close


{move 2}

>>> declare pnn that Pn2 n


pnn : that Pn2 (n)


{move 2}

>>> postulate eqpf Pn2, pnn : that \
    Pn2 m
```

```
        eqpf : [(Pn2_1 : [(n3_2 : in Nat) =>
               (--- : prop)]), (pnn_1
             : that Pn2_1 (n)) => (--- : that
            Pn2_1 (m))]



        {move 1}

        >>> close



    {move 1}

    >>> postulate Eqnproof n m, eqpf : that \
        n Eqn m



    Eqnproof : [(n_1 : in Nat), (m_1
        : in Nat), (eqpf_1 : [(Pn2_2 : [(n3_3
              : in Nat) => (--- : prop)]), (pnn_2
           : that Pn2_2 (n_1)) => (---
           : that Pn2_2 (m_1))]) => (---
         : that n_1 Eqn m_1)]



    {move 0}
end Lestrade execution
```

We prove the trivial theorem of reflexivity of equality.

```
begin Lestrade execution

    >>> comment We test the equality introduction \
        rule
```

```
{move 1}

>>> comment by proving reflexivity of \
    equality .


{move 1}

>>> open


   {move 2}

   >>> open


      {move 3}

      >>> declare n3 in Nat


      n3 : in Nat


      {move 3}

      >>> postulate Pn2 n3 : prop


      Pn2 : [(n3_1 : in Nat) => (---
            : prop)]


      {move 2}

      >>> close
```

173

```
{move 2}

>>> declare pnn that Pn2 n


pnn : that Pn2 (n)


{move 2}

>>> define eqpftest Pn2, pnn : pnn


eqpftest : [(Pn2_1 : [(n3_2 : in
       Nat) => (--- : prop)]), (pnn_1
     : that Pn2_1 (n)) => (--- : that
     Pn2_1 (n))]


{move 1}

>>> close


{move 1}

>>> define Refln n : Eqnproof n n, eqpftest


Refln : [(n_1 : in Nat) =>
    ({def} Eqnproof (n_1, n_1, [(Pn2_2
       : [(n3_3 : in Nat) => (--- : prop)]), (pnn_2
       : that Pn2_2 (n_1)) =>
       ({def} pnn_2 : that Pn2_2 (n_1))]) : that
    n_1 Eqn n_1)]
```

```
    Refln : [(n_1 : in Nat) => (--- : that
        n_1 Eqn n_1)]


    {move 0}
end Lestrade execution
```

We had to declare equality in order to declare the other two Peano axioms:
here they are.

```
begin Lestrade execution

    >>> postulate Pa3 n : that ~ (Succ n Eqn \
        1)


    Pa3 : [(n_1 : in Nat) => (--- : that
        ~ (Succ (n_1) Eqn 1))]


    {move 0}

    >>> postulate Pa4 n m : that (Succ n Eqn \
        Succ m) -> n Eqn m


    Pa4 : [(n_1 : in Nat), (m_1 : in
        Nat) => (--- : that (Succ (n_1) Eqn
        Succ (m_1)) -> n_1 Eqn m_1)]


    {move 0}

    >>> comment These definitions are by no \
        means exhaustive .One wants
```

```
   {move 1}

   >>> comment to declare quantifiers over \
       natural numbers for example .


   {move 1}
end Lestrade execution
```

Here are a set of very economical declarations for the simple type theory of
sets. Once again, I have not declared the quantifiers that are surely wanted.

```
begin Lestrade execution

   >>> comment Declarations for second order \
       type theory .


   {move 1}
end Lestrade execution
```

I could declare the types without using natural numbers at all, but since
I have them I will use them.

```
begin Lestrade execution

   >>> postulate level n : type


   level : [(n_1 : in Nat) => (--- : type)]


   {move 0}

   >>> comment level n is what we usually \
       call type n .
```

```
   {move 1}

   >>> comment The bottom type will be type \
       1 .


   {move 1}

   >>> declare n3 in Nat


   n3 : in Nat


   {move 1}

   >>> declare x10 in level n3


   x10 : in level (n3)


   {move 1}

   >>> declare y10 in level Succ n3


   y10 : in level (Succ (n3))


   {move 1}
end Lestrade execution
```

Here is the membership relation. It is ternary: we must unavoidably put
a natural number first to indicate the type of the element.

```
begin Lestrade execution

   >>> comment Declare the membership relation \
       (with a type argument)


   {move 1}

   >>> postulate E n3 x10 y10 : prop


   E : [(n3_1 : in Nat), (x10_1 : in
       level (n3_1)), (y10_1 : in level
       (Succ (n3_1))) => (--- : prop)]


   {move 0}
end Lestrade execution
```

Here is the set abstract primitive, taking predicates of type n objects to sets of type $n + 1$.

```
begin Lestrade execution

   >>> comment Declare the set abstract constructor


   {move 1}

   >>> open


      {move 2}

      >>> declare x11 in level n3
```

```
        x11 : in level (n3)


        {move 2}

        >>> postulate Pt x11 : prop


        Pt : [(x11_1 : in level (n3)) =>
              (--- : prop)]


        {move 1}

        >>> close


    {move 1}
end Lestrade execution
```

Here are the comprehension axioms, declared in a most economical way.

```
begin Lestrade execution

    >>> comment Declare the comprehension \
        axioms


    {move 1}

    >>> postulate setof n3 Pt : in level Succ \
        n3


    setof : [(n3_1 : in Nat), (Pt_1 : [(x11_2
          : in level (n3_1)) => (--- : prop)]) =>
        (--- : in level (Succ (n3_1)))]
```

```
{move 0}

>>> declare compev1 that E (n3, x10, setof \
    n3 Pt)


compev1 : that E (n3, x10, n3 setof
 Pt)


{move 1}

>>> postulate Comp1 n3 x10, Pt : that \
    Pt x10


Comp1 : [(n3_1 : in Nat), (x10_1
    : in level (n3_1)), (Pt_1 : [(x11_2
        : in level (n3_1)) => (--- : prop)]) =>
    (--- : that Pt_1 (x10_1))]


{move 0}

>>> declare compev2 that Pt x10


compev2 : that Pt (x10)


{move 1}

>>> postulate Comp2 n3 x10, Pt : that \
    E (n3, x10, setof n3 Pt)
```

```
    Comp2 : [(n3_1 : in Nat), (x10_1
        : in level (n3_1)), (Pt_1 : [(x11_2
          : in level (n3_1)) => (--- : prop)]) =>
        (--- : that E (n3_1, x10_1, n3_1
        setof Pt_1))]


    {move 0}
end Lestrade execution
```

Here is the extensionality axiom, whose force is that things having the same elements (and belonging to a successor type) themselves belong to the same sets. In the presence of comprehension, this is enough: objects with the same elements will thus be indiscernible. Of course a definition of equality (and definitions of quantifiers) would appear in a full treatment.

```
begin Lestrade execution

    >>> comment Declare the extensionality \
        axiom


    {move 1}

    >>> declare xx10 in level Succ n3


    xx10 : in level (Succ (n3))


    {move 1}

    >>> declare yy10 in level Succ n3


    yy10 : in level (Succ (n3))
```

```
{move 1}

>>> declare ww10 in level Succ (Succ \
    n3)


ww10 : in level (Succ (Succ (n3)))


{move 1}

>>> declare xinw that (Succ n3) E xx10 \
    ww10


xinw : that E (Succ (n3), xx10, ww10)


{move 1}

>>> open


   {move 2}

   >>> declare z11 in level n3


   z11 : in level (n3)


   {move 2}

   >>> declare zinx that n3 E z11 xx10


   zinx : that E (n3, z11, xx10)
```

```
{move 2}

>>> declare ziny that n3 E z11 yy10


ziny : that E (n3, z11, yy10)


{move 2}

>>> postulate xincy z11 zinx : that \
    n3 E z11 yy10


xincy : [(z11_1 : in level (n3)), (zinx_1
    : that E (n3, z11_1, xx10)) =>
    (--- : that E (n3, z11_1, yy10))]


{move 1}

>>> postulate yincx z11 ziny : that \
    n3 E z11 xx10


yincx : [(z11_1 : in level (n3)), (ziny_1
    : that E (n3, z11_1, yy10)) =>
    (--- : that E (n3, z11_1, xx10))]


{move 1}

>>> close


{move 1}
```

```
>>> postulate Extensionality n3 xx10 yy10 \
    ww10, xinw, xincy, yincx : that (Succ \
    n3) E yy10 ww10


Extensionality : [(n3_1 : in Nat), (xx10_1
    : in level (Succ (n3_1))), (yy10_1
    : in level (Succ (n3_1))), (ww10_1
    : in level (Succ (Succ (n3_1)))), (xinw_1
    : that E (Succ (n3_1), xx10_1, ww10_1)), (xincy_1
    : [(z11_2 : in level (n3_1)), (zinx_2
      : that E (n3_1, z11_2, xx10_1)) =>
      (--- : that E (n3_1, z11_2, yy10_1))]), (yincx_1
    : [(z11_2 : in level (n3_1)), (ziny_2
      : that E (n3_1, z11_2, yy10_1)) =>
      (--- : that E (n3_1, z11_2, xx10_1))]) =>
    (--- : that E (Succ (n3_1), yy10_1, ww10_1))]


   {move 0}
end Lestrade execution
```

6.7 Quantifiers over general types

In this section we present the primitives supporting quantification over all types of sort `type`. Quantifiers over the sort `type` itself could be introduced independently but would of course be more dangerous (it would be easier to utter paradoxes). These tools could be used to define quantifiers over the natural numbers and over each of the levels of type theory without any need for new primitives.

This is an illustration of the fact that we have a general ability to postulate and define operations on a domain of types.

The text is cloned from the text for the quantifiers over the sort `obj` above, and the comments have mostly not been revised to reflect the changes in identifiers.

```
begin Lestrade execution

   >>> comment Declaring the universal quantifier \
       for general types .


   {move 1}

   >>> clearcurrent

{move 1}

   >>> declare tau type


   tau : type


   {move 1}

   >>> open


      {move 2}

      >>> declare uu in tau


      uu : in tau


      {move 2}

      >>> postulate Ptt uu : prop


      Ptt : [(uu_1 : in tau) => (---
             : prop)]
```

```
   {move 1}

   >>> close


{move 1}

>>> postulate Forallt tau Ptt : prop


Forallt : [(tau_1 : type), (Ptt_1
     : [(uu_2 : in tau_1) => (--- : prop)]) =>
     (--- : prop)]


{move 0}

>>> comment Declaring the rule U I of \
     universal instantiation (for general \
     types)


{move 1}

>>> declare Ptt2 that Forallt tau Ptt


Ptt2 : that tau Forallt Ptt


{move 1}

>>> declare xt in tau


xt : in tau
```

186

```
{move 1}

>>> postulate Uit tau Ptt, Ptt2 xt : that \
    Ptt xt


Uit : [(tau_1 : type), (Ptt_1 : [(uu_2
        : in tau_1) => (--- : prop)]), (Ptt2_1
    : that tau_1 Forallt Ptt_1), (xt_1
    : in tau_1) => (--- : that Ptt_1
    (xt_1))]


{move 0}

>>> comment Note in the previous line \
    that we follow P


{move 1}

>>> comment with a comma : a construction \
    argument may need to be


{move 1}

>>> comment guarded with commas so it \
    will not be read as applied .


{move 1}

>>> comment Opening an environment to \
    declare a construction
```

```
{move 1}

>>> comment that witnesses provability \
    of a universal statement


{move 1}

>>> open


   {move 2}

   >>> declare ut in tau


   ut : in tau


   {move 2}

   >>> postulate Qt2 ut : that Ptt ut


   Qt2 : [(ut_1 : in tau) => (---
       : that Ptt (ut_1))]


   {move 1}

   >>> close


{move 1}

>>> comment The rule of universal generalization \
    (for general types)
```

```
{move 1}

>>> postulate Ugt tau Ptt, Qt2 : that \
    Forallt tau Ptt


Ugt : [(tau_1 : type), (Ptt_1 : [(uu_2
      : in tau_1) => (--- : prop)]), (Qt2_1
    : [(ut_2 : in tau_1) => (--- : that
       Ptt_1 (ut_2))]) => (--- : that
    tau_1 Forallt Ptt_1)]


{move 0}

>>> comment The existential quantifier \
    (for general types)


{move 1}

>>> postulate Existst tau Ptt : prop


Existst : [(tau_1 : type), (Ptt_1
    : [(uu_2 : in tau_1) => (--- : prop)]) =>
    (--- : prop)]


{move 0}

>>> comment the rule E G (existential \
    introduction) (for general types)


{move 1}
```

```
>>> declare evt that Ptt xt


evt : that Ptt (xt)


{move 1}

>>> postulate Egt tau Ptt, xt evt : that \
    Existst tau Ptt


Egt : [(tau_1 : type), (Ptt_1 : [(uu_2
        : in tau_1) => (--- : prop)]), (xt_1
    : in tau_1), (evt_1 : that Ptt_1
    (xt_1)) => (--- : that tau_1 Existst
    Ptt_1)]


{move 0}

>>> comment the rule E I (existential \
    elimination) (for general types)


{move 1}

>>> declare gt prop


gt : prop


{move 1}

>>> declare ext that Existst tau Ptt
```

```
ext : that tau Existst Ptt


{move 1}

>>> open


    {move 2}

    >>> declare wt in tau


    wt : in tau


    {move 2}

    >>> declare evt2 that Ptt wt


    evt2 : that Ptt (wt)


    {move 2}

    >>> postulate wit wt evt2 : that gt


    wit : [(wt_1 : in tau), (evt2_1
        : that Ptt (wt_1)) => (--- : that
        gt)]


    {move 1}

    >>> close
```

```
{move 1}

>>> postulate Eit tau Ptt, gt, ext, wit \
    : that gt


Eit : [(tau_1 : type), (Ptt_1 : [(uu_2
      : in tau_1) => (--- : prop)]), (gt_1
    : prop), (ext_1 : that tau_1 Existst
    Ptt_1), (wit_1 : [(wt_2 : in tau_1), (evt2_2
       : that Ptt_1 (wt_2)) => (---
       : that gt_1)]) => (--- : that
    gt_1)]


{move 0}
end Lestrade execution
```

6.8 Equality and the definite description operator for untyped and typed objects

```
begin Lestrade execution

>>> comment Equality uniqueness and definite \
    description


{move 1}

>>> open


   {move 2}
```

192

```
>>> declare x17 obj

x17 : obj

{move 2}

>>> postulate P x17 prop

P : [(x17_1 : obj) => (--- : prop)]

{move 1}

>>> close


{move 1}

>>> declare x obj


x : obj


{move 1}

>>> declare y obj


y : obj


{move 1}

>>> comment Equality of untyped objects
```

```
{move 1}

>>> postulate = x y : prop


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


{move 0}

>>> comment Develop equality introduction \
    rule (indiscernibility)


{move 1}

>>> open


   {move 2}

   >>> open


      {move 3}

      >>> declare x2 obj


      x2 : obj


      {move 3}

      >>> postulate Peq2 x2 : prop
```

```
    Peq2 : [(x2_1 : obj) => (---
        : prop)]


    {move 2}

    >>> close


{move 2}

>>> declare pxev that Peq2 x


pxev : that Peq2 (x)


{move 2}

>>> postulate pyev Peq2, pxev : that \
    Peq2 y


pyev : [(Peq2_1 : [(x2_2 : obj) =>
        (--- : prop)]), (pxev_1
    : that Peq2_1 (x)) => (--- : that
    Peq2_1 (y))]


    {move 1}

    >>> close


{move 1}
```

```
>>> postulate Eqintro x y pyev : that \
    x = y


Eqintro : [(x_1 : obj), (y_1 : obj), (pyev_1
    : [(Peq2_2 : [(x2_3 : obj) =>
          (--- : prop)]), (pxev_2
        : that Peq2_2 (x_1)) => (---
        : that Peq2_2 (y_1))]) => (---
    : that x_1 = y_1)]


{move 0}

>>> comment Construct equality elimination \
    rule (substitution)


{move 1}

>>> declare xyeqev that x = y


xyeqev : that x = y


{move 1}

>>> declare pxev that P x


pxev : that P (x)


{move 1}

>>> postulate Eqelim P, x y xyeqev pxev \
    : that P y
```

```
Eqelim : [(P_1 : [(x17_2 : obj) =>
        (--- : prop)]), (x_1 : obj), (y_1
    : obj), (xyeqev_1 : that x_1 = y_1), (pxev_1
    : that P_1 (x_1)) => (--- : that
    P_1 (y_1))]
```

{move 0}

>>> comment The same rules for equality, adapted \
    to general types

{move 1}

>>> declare yt in tau

yt : in tau

{move 1}

>>> postulate eqt tau xt yt : prop

```
eqt : [(tau_1 : type), (xt_1 : in
    tau_1), (yt_1 : in tau_1) => (---
    : prop)]
```

{move 0}

>>> comment Develop equality introduction \
    rule (indiscernibility)

```
{move 1}

>>> open

    {move 2}

    >>> open

        {move 3}

        >>> declare x2 in tau


        x2 : in tau


        {move 3}

        >>> postulate Peqt2 x2 : prop


        Peqt2 : [(x2_1 : in tau) => (---
            : prop)]


        {move 2}

        >>> close


    {move 2}

    >>> declare pxevt that Peqt2 xt
```

```
      pxevt : that Peqt2 (xt)


   {move 2}

   >>> postulate pyevt Peqt2, pxevt : that \
       Peqt2 yt


   pyevt : [(Peqt2_1 : [(x2_2 : in
          tau) => (--- : prop)]), (pxevt_1
       : that Peqt2_1 (xt)) => (---
       : that Peqt2_1 (yt))]


   {move 1}

   >>> close


{move 1}

>>> postulate Eqintrot tau xt yt pyevt \
    : that tau eqt xt yt


Eqintrot : [(tau_1 : type), (xt_1
    : in tau_1), (yt_1 : in tau_1), (pyevt_1
    : [(Peqt2_2 : [(x2_3 : in tau_1) =>
          (--- : prop)]), (pxevt_2
       : that Peqt2_2 (xt_1)) => (---
       : that Peqt2_2 (yt_1))]) =>
     (--- : that eqt (tau_1, xt_1, yt_1))]


{move 0}

>>> comment Construct equality elimination \
```

```
        rule (substitution)


{move 1}

>>> declare xyeqevt that tau eqt xt yt


xyeqevt : that eqt (tau, xt, yt)


{move 1}

>>> declare pxevt that Ptt xt


pxevt : that Ptt (xt)


{move 1}

>>> postulate Eqelimt tau Ptt, xt yt \
    xyeqevt pxevt : that Ptt yt


Eqelimt : [(tau_1 : type), (Ptt_1
    : [(uu_2 : in tau_1) => (--- : prop)]), (xt_1
    : in tau_1), (yt_1 : in tau_1), (xyeqevt_1
    : that eqt (tau_1, xt_1, yt_1)), (pxevt_1
    : that Ptt_1 (xt_1)) => (--- : that
    Ptt_1 (yt_1))]


{move 0}

>>> comment The definite description operator
```

```
{move 1}

>>> declare atleast1 that Exists P


atleast1 : that Exists (P)


{move 1}

>>> open


   {move 2}

   >>> declare x1 obj


   x1 : obj


   {move 2}

   >>> declare x2 obj


   x2 : obj


   {move 2}

   >>> declare thatpx1 that P x1


   thatpx1 : that P (x1)


   {move 2}
```

```
>>> declare thatpx2 that P x2


thatpx2 : that P (x2)


{move 2}

>>> postulate atmost1 x1 x2 thatpx1 \
    thatpx2 : that x1 = x2


atmost1 : [(x1_1 : obj), (x2_1
    : obj), (thatpx1_1 : that P (x1_1)), (thatpx2_1
    : that P (x2_1)) => (--- : that
    x1_1 = x2_1)]


{move 1}

>>> close


{move 1}

>>> postulate The P, atleast1 atmost1 \
    : obj


The : [(P_1 : [(x17_2 : obj) => (---
      : prop)]), (atleast1_1 : that
   Exists (P_1)), (atmost1_1 : [(x1_2
      : obj), (x2_2 : obj), (thatpx1_2
      : that P_1 (x1_2)), (thatpx2_2
      : that P_1 (x2_2)) => (--- : that
      x1_2 = x2_2)]) => (--- : obj)]
```

```
{move 0}

>>> postulate Theprop P, atleast1 atmost1 \
    : that P (The (P, atleast1, atmost1))


Theprop : [(P_1 : [(x17_2 : obj) =>
       (--- : prop)]), (atleast1_1
    : that Exists (P_1)), (atmost1_1
    : [(x1_2 : obj), (x2_2 : obj), (thatpx1_2
       : that P_1 (x1_2)), (thatpx2_2
       : that P_1 (x2_2)) => (--- : that
       x1_2 = x2_2)]) => (--- : that
    P_1 (The (P_1, atleast1_1, atmost1_1)))]


{move 0}

>>> comment The definite description operator \
    (for general types)


{move 1}

>>> declare atleastt1 that Existst tau \
    Ptt


atleastt1 : that tau Existst Ptt


{move 1}

>>> open


   {move 2}
```

```
>>> declare x1 in tau

x1 : in tau

{move 2}

>>> declare x2 in tau

x2 : in tau

{move 2}

>>> declare thatpx1 that Ptt x1

thatpx1 : that Ptt (x1)

{move 2}

>>> declare thatpx2 that Ptt x2

thatpx2 : that Ptt (x2)

{move 2}

>>> postulate atmostt1 x1 x2 thatpx1 \
    thatpx2 : that tau eqt x1 x2

atmostt1 : [(x1_1 : in tau), (x2_1
```

```
        : in tau), (thatpx1_1 : that
        Ptt (x1_1)), (thatpx2_1 : that
        Ptt (x2_1)) => (--- : that eqt
        (tau, x1_1, x2_1))]


    {move 1}

    >>> close


{move 1}

>>> postulate Thet tau Ptt, atleastt1 \
    atmostt1 : in tau


Thet : [(tau_1 : type), (Ptt_1 : [(uu_2
        : in tau_1) => (--- : prop)]), (atleastt1_1
     : that tau_1 Existst Ptt_1), (atmostt1_1
     : [(x1_2 : in tau_1), (x2_2 : in
        tau_1), (thatpx1_2 : that Ptt_1
        (x1_2)), (thatpx2_2 : that
        Ptt_1 (x2_2)) => (--- : that
        eqt (tau_1, x1_2, x2_2))]) =>
     (--- : in tau_1)]


{move 0}

>>> postulate Thepropt tau Ptt, atleastt1 \
    atmostt1 : that Ptt (Thet (tau, Ptt, atleastt1, atmostt1))


Thepropt : [(tau_1 : type), (Ptt_1
     : [(uu_2 : in tau_1) => (--- : prop)]), (atleastt1_1
     : that tau_1 Existst Ptt_1), (atmostt1_1
     : [(x1_2 : in tau_1), (x2_2 : in
```

```
        tau_1), (thatpx1_2 : that Ptt_1
        (x1_2)), (thatpx2_2 : that
        Ptt_1 (x2_2)) => (--- : that
        eqt (tau_1, x1_2, x2_2))]) =>
      (--- : that Ptt_1 (Thet (tau_1, Ptt_1, atleastt1_1, atmostt1_1)))]


    {move 0}
end Lestrade execution
```

6.9 Declarations for complex type theories, up to the level of bootstrapping Lestrade's own construction sorts

The declarations here will support reasoning in Church's simple type theory of [2] (or modern variations); they are further augmented with dependent product and construction types of a sort which would be required to emulate Lestrade's own system of construction sorts.

```
begin Lestrade execution

   >>> postulate One type


   One : type


   {move 0}

   >>> postulate Unique : in One


   Unique : in One


   {move 0}
```

```
>>> declare xx1 in One


xx1 : in One


{move 1}

>>> postulate Oneproof xx1 : that One \
    eqt xx1 Unique


Oneproof : [(xx1_1 : in One) => (---
    : that eqt (One, xx1_1, Unique))]


{move 0}

>>> declare sigma type


sigma : type


{move 1}

>>> postulate X tau sigma : type


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


{move 0}

>>> declare xt2 in tau
```

```
xt2 : in tau


{move 1}

>>> declare ys in sigma


ys : in sigma


{move 1}

>>> postulate pair tau sigma xt2 ys : in \
    tau X sigma


pair : [(tau_1 : type), (sigma_1
    : type), (xt2_1 : in tau_1), (ys_1
    : in sigma_1) => (--- : in tau_1
    X sigma_1)]


{move 0}

>>> declare zp in tau X sigma


zp : in tau X sigma


{move 1}

>>> postulate pi1 tau sigma zp : in tau


pi1 : [(tau_1 : type), (sigma_1 : type), (zp_1
    : in tau_1 X sigma_1) => (--- : in
```

```
    tau_1)]


{move 0}

>>> postulate pi2 tau sigma zp : in sigma


pi2 : [(tau_1 : type), (sigma_1 : type), (zp_1
    : in tau_1 X sigma_1) => (--- : in
    sigma_1)]


{move 0}

>>> postulate Xexact tau sigma zp : that \
    (tau X sigma) eqt zp, pair tau sigma \
    (pi1 tau sigma zp) (pi2 tau sigma zp)


Xexact : [(tau_1 : type), (sigma_1
    : type), (zp_1 : in tau_1 X sigma_1) =>
    (--- : that eqt (tau_1 X sigma_1, zp_1, pair
    (tau_1, sigma_1, pi1 (tau_1, sigma_1, zp_1), pi2
    (tau_1, sigma_1, zp_1))))]


{move 0}

>>> postulate Pow tau type


Pow : [(tau_1 : type) => (--- : type)]


{move 0}

>>> open
```

```
    {move 2}

    >>> declare xt3 in tau


    xt3 : in tau


    {move 2}

    >>> postulate tausub xt3 : prop


    tausub : [(xt3_1 : in tau) => (---
        : prop)]


    {move 1}

    >>> close


{move 1}

>>> postulate Setc tau tausub : in Pow \
    tau


Setc : [(tau_1 : type), (tausub_1
    : [(xt3_2 : in tau_1) => (--- : prop)]) =>
    (--- : in Pow (tau_1))]


{move 0}

>>> declare Ac in Pow tau
```

210

```
Ac : in Pow (tau)


{move 1}

>>> postulate Ec tau xt Ac : prop


Ec : [(tau_1 : type), (xt_1 : in
      tau_1), (Ac_1 : in Pow (tau_1)) =>
      (--- : prop)]


{move 0}

>>> declare ev1 that tausub xt


ev1 : that tausub (xt)


{move 1}

>>> declare ev2 that tau Ec xt tau Setc \
      tausub


ev2 : that Ec (tau, xt, tau Setc tausub)


{move 1}

>>> postulate Compc1 tau xt, tausub, ev1 \
      : that tau Ec xt tau Setc tausub
```

```
Compc1 : [(tau_1 : type), (xt_1 : in
    tau_1), (tausub_1 : [(xt3_2 : in
        tau_1) => (--- : prop)]), (ev1_1
    : that tausub_1 (xt_1)) => (---
    : that Ec (tau_1, xt_1, tau_1 Setc
    tausub_1))]
```

{move 0}

>>> postulate Compc2 tau xt, tausub, ev2 \
    : that tausub xt

```
Compc2 : [(tau_1 : type), (xt_1 : in
    tau_1), (tausub_1 : [(xt3_2 : in
        tau_1) => (--- : prop)]), (ev2_1
    : that Ec (tau_1, xt_1, tau_1 Setc
    tausub_1)) => (--- : that tausub_1
    (xt_1))]
```

{move 0}

>>> declare Bc in Pow tau

Bc : in Pow (tau)

{move 1}

>>> open

   {move 2}

   >>> declare xt1 in tau

```
xt1 : in tau


{move 2}

>>> declare xtina1 that tau Ec xt1 \
    Ac


xtina1 : that Ec (tau, xt1, Ac)


{move 2}

>>> postulate aincb xt1 xtina1 : that \
    tau Ec xt1 Bc


aincb : [(xt1_1 : in tau), (xtina1_1
    : that Ec (tau, xt1_1, Ac)) =>
    (--- : that Ec (tau, xt1_1, Bc))]


{move 1}

>>> declare xtinb1 that tau Ec xt1 \
    Bc


xtinb1 : that Ec (tau, xt1, Bc)


{move 2}

>>> postulate binca xt1 xtinb1 : that \
    tau Ec xt1 Ac
```

```
      binca : [(xt1_1 : in tau), (xtinb1_1
           : that Ec (tau, xt1_1, Bc)) =>
           (--- : that Ec (tau, xt1_1, Ac))]


   {move 1}

   >>> close


{move 1}

>>> postulate Extc tau Ac Bc, aincb, binca \
     : that (Pow tau) eqt Ac Bc


Extc : [(tau_1 : type), (Ac_1 : in
     Pow (tau_1)), (Bc_1 : in Pow (tau_1)), (aincb_1
     : [(xt1_2 : in tau_1), (xtina1_2
        : that Ec (tau_1, xt1_2, Ac_1)) =>
        (--- : that Ec (tau_1, xt1_2, Bc_1))]), (binca_1
     : [(xt1_2 : in tau_1), (xtinb1_2
        : that Ec (tau_1, xt1_2, Bc_1)) =>
        (--- : that Ec (tau_1, xt1_2, Ac_1))]) =>
     (--- : that eqt (Pow (tau_1), Ac_1, Bc_1))]


{move 0}

>>> postulate ==> tau sigma : type


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

```
{move 0}

>>> open


   {move 2}

   >>> declare var in tau


   var : in tau


   {move 2}

   >>> postulate lambdabody var : in sigma


   lambdabody : [(var_1 : in tau) =>
       (--- : in sigma)]


   {move 1}

   >>> close


{move 1}

>>> postulate Lambda tau sigma lambdabody \
    : in tau ==> sigma


Lambda : [(tau_1 : type), (sigma_1
    : type), (lambdabody_1 : [(var_2
       : in tau_1) => (--- : in sigma_1)]) =>
    (--- : in tau_1 ==> sigma_1)]
```

```
{move 0}

>>> declare Fc in tau ==> sigma


Fc : in tau ==> sigma


{move 1}

>>> declare Gc in tau ==> sigma


Gc : in tau ==> sigma


{move 1}

>>> declare xt4 in tau


xt4 : in tau


{move 1}

>>> postulate Applyc tau sigma Fc, xt4 \
    : in sigma


Applyc : [(tau_1 : type), (sigma_1
    : type), (Fc_1 : in tau_1 ==> sigma_1), (xt4_1
    : in tau_1) => (--- : in sigma_1)]


{move 0}
```

216

```
>>> postulate Beta tau sigma lambdabody, xt4 \
    : that sigma eqt Applyc tau sigma (Lambda \
    tau sigma lambdabody) xt4 lambdabody \
    xt4


Beta : [(tau_1 : type), (sigma_1
    : type), (lambdabody_1 : [(var_2
      : in tau_1) => (--- : in sigma_1)]), (xt4_1
    : in tau_1) => (--- : that eqt (sigma_1, Applyc
    (tau_1, sigma_1, Lambda (tau_1, sigma_1, lambdabody_1), xt4_1), lambdabo
    (xt4_1)))]


{move 0}

>>> open

   {move 2}

   >>> declare xt5 in tau


   xt5 : in tau


   {move 2}

   >>> postulate sameval xt5 : that sigma \
       eqt (Applyc tau sigma Fc xt5) (Applyc \
       tau sigma Gc xt5)


   sameval : [(xt5_1 : in tau) => (---
       : that eqt (sigma, Applyc (tau, sigma, Fc, xt5_1), Applyc
       (tau, sigma, Gc, xt5_1)))]
```

217

```
   {move 1}

   >>> close


{move 1}

>>> postulate Extfnc tau sigma Fc Gc sameval \
     : that (tau ==> sigma) eqt Fc Gc


Extfnc : [(tau_1 : type), (sigma_1
     : type), (Fc_1 : in tau_1 ==> sigma_1), (Gc_1
     : in tau_1 ==> sigma_1), (sameval_1
     : [(xt5_2 : in tau_1) => (--- : that
       eqt (sigma_1, Applyc (tau_1, sigma_1, Fc_1, xt5_2), Applyc
       (tau_1, sigma_1, Gc_1, xt5_2)))]) =>
     (--- : that eqt (tau_1 ==> sigma_1, Fc_1, Gc_1))]


{move 0}

>>> open


   {move 2}

   >>> declare ys5 in tau


   ys5 : in tau


   {move 2}

   >>> postulate Rhofun ys5 : type
```

```
        Rhofun : [(ys5_1 : in tau) => (---
            : type)]


    {move 1}

    >>> close


{move 1}

>>> postulate Xx tau Rhofun : type


Xx : [(tau_1 : type), (Rhofun_1 : [(ys5_2
        : in tau_1) => (--- : type)]) =>
    (--- : type)]


{move 0}

>>> declare xt5 in tau


xt5 : in tau


{move 1}

>>> declare ys5 in Rhofun xt5


ys5 : in Rhofun (xt5)


{move 1}
```

```
>>> postulate paird tau Rhofun, xt5 ys5 \
    : in tau Xx Rhofun


paird : [(tau_1 : type), (Rhofun_1
    : [(ys5_2 : in tau_1) => (--- : type)]), (xt5_1
    : in tau_1), (ys5_1 : in Rhofun_1
    (xt5_1)) => (--- : in tau_1 Xx
    Rhofun_1)]


{move 0}

>>> declare zp5 in tau Xx Rhofun


zp5 : in tau Xx Rhofun


{move 1}

>>> postulate Pi1 tau Rhofun, zp5 : in \
    tau


Pi1 : [(tau_1 : type), (Rhofun_1
    : [(ys5_2 : in tau_1) => (--- : type)]), (zp5_1
    : in tau_1 Xx Rhofun_1) => (--- : in
    tau_1)]


{move 0}

>>> postulate Pi2 tau Rhofun, zp5 : in \
    Rhofun (Pi1 tau Rhofun, zp5)


Pi2 : [(tau_1 : type), (Rhofun_1
```

220

```
   : [(ys5_2 : in tau_1) => (--- : type)]), (zp5_1
   : in tau_1 Xx Rhofun_1) => (--- : in
   Rhofun_1 (Pi1 (tau_1, Rhofun_1, zp5_1)))]


{move 0}

>>> postulate Xxexact tau Rhofun, zp5 \
   : that (tau Xx Rhofun) eqt zp5, paird \
   tau Rhofun, (Pi1 tau Rhofun, zp5) (Pi2 \
   tau Rhofun, zp5)


Xxexact : [(tau_1 : type), (Rhofun_1
   : [(ys5_2 : in tau_1) => (--- : type)]), (zp5_1
   : in tau_1 Xx Rhofun_1) => (--- : that
   eqt (tau_1 Xx Rhofun_1, zp5_1, paird
   (tau_1, Rhofun_1, Pi1 (tau_1, Rhofun_1, zp5_1), Pi2
   (tau_1, Rhofun_1, zp5_1))))]


{move 0}

>>> postulate =>> tau Rhofun : type


=>> : [(tau_1 : type), (Rhofun_1
   : [(ys5_2 : in tau_1) => (--- : type)]) =>
   (--- : type)]


{move 0}

>>> open


   {move 2}
```

221

```
    >>> declare var in tau


    var : in tau


    {move 2}

    >>> postulate lambdabodyd var : in \
        Rhofun var


    lambdabodyd : [(var_1 : in tau) =>
        (--- : in Rhofun (var_1))]


    {move 1}

    >>> close


{move 1}

>>> postulate Lambdad tau Rhofun, lambdabodyd \
    : in tau =>> Rhofun


Lambdad : [(tau_1 : type), (Rhofun_1
    : [(ys5_2 : in tau_1) => (--- : type)]), (lambdabodyd_1
    : [(var_2 : in tau_1) => (--- : in
      Rhofun_1 (var_2))]) => (---
    : in tau_1 =>> Rhofun_1)]


{move 0}

>>> declare Fd in tau =>> Rhofun
```

```
Fd : in tau =>> Rhofun


{move 1}

>>> declare Gd in tau =>> Rhofun


Gd : in tau =>> Rhofun


{move 1}

>>> declare xt6 in tau


xt6 : in tau


{move 1}

>>> postulate Applyd tau Rhofun, Fd, xt6 \
    : in Rhofun xt6


Applyd : [(tau_1 : type), (Rhofun_1
    : [(ys5_2 : in tau_1) => (--- : type)]), (Fd_1
    : in tau_1 =>> Rhofun_1), (xt6_1
    : in tau_1) => (--- : in Rhofun_1
    (xt6_1))]


{move 0}

>>> postulate Betad tau Rhofun, lambdabodyd, xt6 \
    : that (Rhofun xt6) eqt Applyd tau Rhofun, (Lambdad \
    tau Rhofun, lambdabodyd) xt6 lambdabodyd \
```

```
   xt6


Betad : [(tau_1 : type), (Rhofun_1
    : [(ys5_2 : in tau_1) => (--- : type)]), (lambdabodyd_1
    : [(var_2 : in tau_1) => (--- : in
      Rhofun_1 (var_2))]), (xt6_1
    : in tau_1) => (--- : that eqt (Rhofun_1
    (xt6_1), Applyd (tau_1, Rhofun_1, Lambdad
    (tau_1, Rhofun_1, lambdabodyd_1), xt6_1), lambdabodyd_1
    (xt6_1)))]


{move 0}

>>> open


   {move 2}

   >>> declare xt7 in tau


   xt7 : in tau


   {move 2}

   >>> postulate samevald xt7 : that (Rhofun \
      xt7) eqt (Applyd tau Rhofun, Fd \
      xt7) (Applyd tau Rhofun, Gd xt7)


   samevald : [(xt7_1 : in tau) =>
      (--- : that eqt (Rhofun (xt7_1), Applyd
      (tau, Rhofun, Fd, xt7_1), Applyd
      (tau, Rhofun, Gd, xt7_1)))]
```

224

```
   {move 1}

   >>> close


{move 1}

>>> postulate Extfnd tau Rhofun, Fd Gd \
    samevald : that (tau =>> Rhofun) eqt \
    Fd Gd


Extfnd : [(tau_1 : type), (Rhofun_1
    : [(ys5_2 : in tau_1) => (--- : type)]), (Fd_1
    : in tau_1 =>> Rhofun_1), (Gd_1
    : in tau_1 =>> Rhofun_1), (samevald_1
    : [(xt7_2 : in tau_1) => (--- : that
      eqt (Rhofun_1 (xt7_2), Applyd
      (tau_1, Rhofun_1, Fd_1, xt7_2), Applyd
      (tau_1, Rhofun_1, Gd_1, xt7_2)))]) =>
    (--- : that eqt (tau_1 =>> Rhofun_1, Fd_1, Gd_1))]


{move 0}

>>> postulate Empty : type


Empty : type


{move 0}

>>> declare xnot in Empty


xnot : in Empty
```

225

```
   {move 1}

   >>> postulate notthere xnot : that ??


   notthere : [(xnot_1 : in Empty) =>
       (--- : that ??)]


   {move 0}
end Lestrade execution
```

7 Bibliography

References

[1] Barendregt, Henk (1992). "Lambda calculi with types". In S. Abramsky, D. Gabbay and T. Maibaum. *Handbook of Logic in Computer Science.* Oxford Science Publications.

[2] Church, A. , "A Formulation of the Simple Theory of Types", *Journal of Symbolic Logic*, 5 (1940): 5668

[3] The Coq Proof Assistant is found at `https://coq.inria.fr/`. The software itself and documentation are found there.

[4] Curry, H. B. and Feys, R. *Combinatory Logic*, 1958, chapter 9, section E.

[5] de Bruijn, N. G., "The Mathematical Language Automath, its Usage, and some of its Extensions", in [16], pp. 73-100.

[6] de Bruijn, N. G., "Example of a Text written in Automath", in [16], pp. 687-700.

[7] de Bruijn, N. G., "On the role of types in mathematics", in Ph. de Groote, ed. *The Curry Howard Isomorphism*, Academia, Louvain-la-Neuve, 1995.

[8] Wiedijk, F., Implementation of Automath, found at `http://www.cs.ru.nl/F.Wiedijk/aut/index.html`. One can also get the source files for the Jutting implementation of Landau there.

[9] Wiedijk, F., "A new implementation of Automath", *Journal of Automated Reasoning*, September 2002, Volume 29, Issue 3, pp 365-387

[10] Holmes, M. Randall, "Disguising Recursively Chained Rewrite Rules as Equational Theorems, as Implemented in the Prover EFTTP Mark 2" in *Rewriting Techniques and Applications, proceedings of RTA '95, Lecture Notes in Computer Science 914*, Springer, 1995, pp. 432-437.

[11] Holmes, M. Randall, and Alves-Foss, J., "The Watson theorem prover", *Journal of Automated Reasoning*, vol. 26 (2001), no. 4, pp. 357-408.

[12] Holmes, M. Randall, source of the Lestrade system: http://math.boisestate.edu/∼holmes/automath/lestrade.sml or the simpler `lestrade_basic.sml` without type inference features such as rewriting.

http://math.boisestate.edu/∼holmes/automath/lestradespecification.sml is the source of the 2020 reimplementation, which as yet does not support rewriting.

[13] Howard, W. A., "The Formulae as Types Notion of Construction", in Seldin J. P. and Hindley, J. R., eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, Academic Press, 1980, pp. 479-490.

[14] Jutting, L. S. van Benthem, "Checking Landau's Grundlagen in the Automath System", selections found in [16], pp. 701-732. The complete Automath book can be found at the URL of [8].

[15] Landau, E,. *Grundlagen der Analysis.* Chelsea Pub. Co., New York, NY, USA, 1965.

[16] Nederpelt, R.P., Geuvers, J.H., and De Vrijer, R.C, eds. *Selected Papers on Automath*, North Holland, 1994.