The Lestrade Type Inspector – Manual and Discussion

M. Randall Holmes

July 11, 2016

version: 4/3/2015 10:15 AM various updates

Contents

1 Introduction

To begin with, the name is a bit of fun. I am Holmes, I already have a theorem prover called Watson; my son suggested Lestrade. I hope that I may be forgiven.

The system presented here is a variant of Automath. The most obvious difference is that the user never enters a λ-term or dependent type. I originally thought that this represented a greater difference from Automath than it actually does.

This file only documents the version `lestrade_basic.sml` without the new rewriting features. All but one of the source files on the web page will run with the basic version. I'm updating this file in summer 2016 to remove anachronisms, and without trying to document the new rewriting commands.

2 Access to the software

Lestrade is contained in a Moscow ML 2.01 program `lestrade.sml`. When this has been compiled, loaded and opened, type `interface <filename>` at the ML command line to reach the Lestrade interface (a log will be recorded to `<filename>.lti`. The command `readfile <file1> <file2>`, **issued at the ML command line**, will clear the Lestrade context then execute the Lestrade commands in `<file1>.lti` and log those commands and any commands the user subsequently types to `<file2>.lti`. Readfile and interface are ML functions, not Lestrade commands. It is important to end your sessions with the Lestrade interface with `quit` (starting `interface` or typing `quit` in the interface does not clear your declarations; only issuing a `readfile` does that.) If you crash out of the interface, the `Cleanup();` command will close input and output files which otherwise might be damaged. The `versiondate();` command will tell you what version of Lestrade you have. The `setmargin <integer>` command will allow you to set the margin

at which lines break in the (marginally) pretty-printed display. The Lestrade log files with extension `.lti` are readable text files; open them with a text editor. It is very important to end an `.lti` file with the line `quit`! It is also very important to close the interface with `quit` before closing the ML window.

3 Commands which can be given in the Lestrade interface

The explanations of the commands are for reference; the terms used are explained in following sections.

`quit` ends the interface session and closes the log file. The Lestrade context is not affected by this command .

`clearall` resets the Lestrade context to its original state.

`open` Opens a new world with index one higher than the previous current world; this becomes the current world and the previous current world becomes the parent world. With an argument, a saved world named by the argument may be opened. For details, see the new paper.

`close` Deletes the current world (unless it is world 1, which one cannot close). The previously parent world becomes the new current world, and the world with index one lower becomes the new parent world.

`clearcurrent` deletes all declarations from the current world. If an argument is supplied, it may introduce declarations from a saved world at that level: for details see the new paper.

`showdec <ident>` shows the type of the identifier `<ident>` if it is declared. The type will include information about the value of a defined abstraction.

`showall` shows all declarations, sorted by world, most recently declared in each world at the top.

`showrecent` shows the declarations in the current world and the parent world.

`declare <ident> <type>` declares the fresh identifier `<ident>` with entity type `<type>` in the current world (terminology explained below). This declares `<ident>` as primitive.

`construct <ident> <arglist> : <type>` declares the fresh identifier `ident` in the parent world as a primitive abstraction of type determined by the arguments in `<arglist>`, which must each be primitive identifiers declared in the current world (not any lower-indexed world), followed by commas if they represent abstractions [a last abstraction argument doesn't require a comma; any argument can be followed by a comma and sometimes a comma is needed before an abstraction to prevent reading it as an infix], and must appear in the order in which they were declared, and include any primitive identifiers declared in the current world on which `<type>`, an entity type, depends. It is possible for `<arglist>` to be empty: in this case the effect of the command is to declare a variable in the parent world, a new constant from the standpoint of the current world.

`define <ident> <arglist> : <value>` declares the fresh identifier `ident` in the parent world as a defined abstraction of type determined by the arguments in `<arglist>` and the computed type of `<value>`. The arguments must each be primitive identifiers declared in the current world (not any lower-indexed world), followed by commas if they represent abstractions [if not final; same remarks about commas after argument as under `construct`], and must appear in the order in which they were declared, and include any primitive identifiers declared in the current world on which `<value>`, an entity term, or the computed type of `<value>` depends. It is possible for `<arglist>` to be empty (this is how entity definitions are given: a zero arity abstraction identifier is always read as an entity term of the appropriate type, the null argument list being supposed supplied).

4 The input language

An identifier may contain upper case letters, lower case letters, numerals or special characters. Any of these may appear initial to an identifier. An upper case letter appearing in an identifier will be terminal or followed by a lower case letter or numeral. A lower case letter appearing in an identifier will be terminal or followed by a lower case letter or numeral. A numeral appearing in an identifier will be terminal or followed by a numeral. A special character appearing in an identifier will be terminal or followed by another special character.

A token is an identifier or one of the special tokens `,` `:` (a comma or a

colon) or an open or close parenthesis.

A command or expression of the input language consists of tokens and spaces. The only function of the spaces is to serve to separate tokens from one another when necessary. The first token in a command line will of course be the command name.

All declared abstractions have arity. An entity term of the input language will either be a single identifier of entity type or an identifier of abstraction type followed by an argument list with number of items determined by the arity of the abstraction. Notice that an abstraction identifier of zero arity is read as a full entity term by itself. In the latest version, the parser has been upgraded to allow infix notation and also argument lists of a more usual kind. Terms may be enclosed in parentheses. Terms written $at_1 \ldots t_n$ (in Polish notation; this is still accepted, with the freedom to put commas after arguments as desired and the positive need to place them before and after abstraction arguments) can also be written $a(t_1, \ldots, t_n)$ [argument list enclosed in parentheses and (optionally) arguments separated by commas] or $t_1 a t_2 \ldots t_n$ as long as n is at least 2 and t_1 is of entity type. In the infix/mixfix notation, the arguments after the abstraction may optionally be separated by commas but cannot be enclosed in parentheses as a block (unless of course there is only one of them). Any abstraction which occurs finally or followed by a comma, colon or close parenthesis is to be interpreted as an abstraction argument, and takes no arguments of its own (either as a prefix or infix). Use of a comma after an abstraction argument is usually obligatory, except at the end of a term or before colons or close parentheses. Abstractions of arity 1 bind more tightly than infixes; otherwise all precedences are equal and infix terms group to the right as in APL, except as modified by use of parentheses. Abstraction identifiers are not displayed or read as followed by infixes. It is important to note that a parenthesis immediately following an abstraction used as a prefix operator will be interpreted as enclosing the list of arguments, not as enclosing the first argument or an initial subterm of the first argument: **if you want to parenthesize the first argument in an argument list, you must also parenthesize the entire argument list**.

The liberalized syntax does not apply to what appears to the left of the colon in a `define` or `construct` command: this will be a declared abstraction name followed by a list of identifiers optionally separated with commas (abstraction arguments in non-final position must be followed by commas and it is wise to precede them with commas as well), and the argument list

will not be enclosed in parentheses.

An argument list is a list of items which are either abstraction identifiers of positive arity followed by a comma [unless final] or full entity terms (which may or may not be followed by a comma). The comma is a device to prevent one from reading the abstraction identifier appearing as an argument from taking following terms up as arguments of its own; in the new version a comma before an abstraction argument may also be obligatory to avoid reading it as an infix. It is probably a good idea to separate arguments with commas if in any doubt.

An entity type is one of `obj`, `prop`, `type` by itself, or else `that` or `in` followed by an entity term. An entity type is well-typed iff it is `obj`, `prop` or `type`, or it is of the form `that` P where P type checks with type `prop`, or it is of the form `in` T, where T type checks as of type `type`.

This section gives mostly syntactical information about well formed entity terms and types; further issues of type enter into whether such terms are accepted.

5 Worlds

The Lestrade context consists of a list of worlds indexed by natural numbers starting at 0. Each of these is a list of declarations. Initially, there is world 0 and world 1, both empty, and the `clearall` command restores this state. The initial state is also restored when the `readfile` command is called from the ML interface.

There are always at least two worlds. The highest indexed world, which we will call world $i + 1$, is called the current world, and world i is called the parent world.

The `open` command creates world $i + 2$ with an empty declaration list. After this command, world $i + 2$ is the current world and world $i + 1$ is the parent world.

The `close` command deletes world $i + 1$, unless $i = 0$, in which case it reports an error. World 1 cannot be closed. All declarations in a closed world are lost. After this command, world i is the current world and world $i - 1$ is the parent world.

The `clearcurrent` command empties world $i+1$ of declarations but does not close it.

All items declared in any world are declared in the context and can be used in terms and types. An item cannot be declared in more than one world.

If one thinks of these as possible worlds with variable objects in them, one should note that when abstractions are declared they actually depend only on subworlds (the primitives declared in their argument lists). The entire context of world $i+1$ in a nontrivial proof is very likely to be an inconsistent collection of entities and abstractions; it is formally easier for the prover to view it as a whole, but in any particular construction one is likely to be postulating only a small part of it.

One should think of objects declared in the current world as variable or arbitrary objects, of the sort which are postulated for the sake of argument. One cannot assume that one knows anything about an object declared in the current world, except insofar as the supposed existence of an object postulated later may tell you something about it. Abstractions that we declare using the `construct` or `define` command are supposed to exist for all objects of the appropriate types (and note that these abstractions are declared in the parent world; they are not hypothetical, except relative to worlds of even lower index). Objects in the parent world or worlds of lower index are objects to whose existence we are more firmly committed – relatively. When we close the current world, the parent world becomes current world to the even lower indexed new parent world. Declaration of abstractions of arity zero has the effect of declaring genuine (relative) constants of entity type in the parent world.

A new feature outlined in the new paper allows one to assign names to worlds and save them. When a new world is opened, it can optionally be assigned a name (an argument to the open command). If no argument is supplied, a world being opened as world $i+1$ gets the numeral $i+1$ as its name. The save command takes an argument (if no argument is supplied, the world is saved with its current name); a world cannot be saved with its default numeral name [or not directly; this can happen as a side-effect, though]. The effect of the save command is to change the name of the current world to the argument of the save command, then save the current world with its attached name (its actual address is the full list of names for all the worlds up to $i+1$, giving a tree structure) and save each world with index $\leq i+1$ with its current name [of course the internal index being the full list of names as indicated]. The clearcurrent command can also take the name of a saved version of the world at the current level as an argument, and load that version

instead of clearing the world. If either open or clearcurrent is issued with an argument which is not the name of an available saved world, an empty world is created with that name. Commands `foropen` and `forclearcurrent` are provided to show lists of names of saved worlds accessible to the open or clearcurrent command in the current context.

6 Declaration commands

The command `declare` x τ declares x in the current world with type τ, if x is a fresh identifier (not already declared in any world) and τ is an entity type which type checks correctly. An identifier declared by this command is a primitive identifier.

The command `construct` x $t_1 \ldots t_n$: τ will first check that x is a fresh identifier, that each term t_i is a primitive entity identifier or a primitive abstraction identifier of positive arity followed by a comma [unless final], that all of these are declared in the current world, and that the t_i's appear in the order in which they were declared [it is possible for there to be no t_i's at all]. It computes types τ_i for each t_i. Each primitive identifier declared in the current world on which τ or any of the τ_i's depends must appear as one of the t_i's. This is enforced by type checking the entity type τ in an environment in which only the t_i's are declared in the current world (including type checking each of the t_i's themselves) after expanding all defined abstractions appearing in τ and declared in the current world (details of this expansion discussed below). If all these checks are passed, x is declared in the parent world (not the current world!) with abstraction type $[(t_1^*, \tau_1^*), \ldots, (t_n^*, \tau_n^*) \rightarrow (\texttt{---}, \tau^*)]$, where the star represents the operation of replacing each t_i whereever it appears with an identifier t_i^* taken from a fresh namespace unique to this type notation (achieved by affixing a numerical index). Note that τ^* is also changed by the indicated expansion of defined notions declared in the current world. An identifier declared by this command is a primitive identifier.

The construct command with an empty argument list simply declares a variable of the stated sort in the parent world (which must type check in the parent world).

The command `define` x $t_1 \ldots t_n$: D will first check that x is a fresh identifier, that each term t_i is a primitive entity identifier or a primitive abstraction identifier of positive arity followed by a comma [unless final],

that all of these are declared in the current world, and that the t_i's appear in the order in which they were declared. It computes types τ_i for each t_i. Each primitive identifier declared in the current world on which the entity term D or its type τ or any of the τ_i's depends must appear as one of the t_i's. This is enforced by type checking D in an environment in which only the t_i's are declared in the current world (including type checking each of the t_i's themselves) after expanding all defined abstractions appearing in D and in τ and declared in the current world (details of this expansion discussed below). If all these checks are passed, x is declared in the parent world (not the current world!) with abstraction type $[(t_1^*, \tau_1^*), \ldots, (t_n^*, \tau_n^*) \rightarrow (D^*, \tau^*)]$, where the star represents the operation of replacing each t_i whereever it appears with an identifier t_i^* taken from a fresh namespace unique to this type notation (achieved by affixing a numerical index). Note that D^* is also affected by the expansion of all defined abstractions declared in the current world. An identifier declared by this command is a defined and not a primitive identifier.

Execution of any of these commands will be followed by the display of the resulting type declaration, if it succeeds. The output language of Lestrade is less parsimonious than the input language: argument lists are delimited with parentheses and separated with commas [after the parser upgrade, the user is permitted but not required to write argument lists thus], and dependent types are displayed as shown here. In the latest version (with the parser upgrade) all arity 2 operators are displayed as infix operators, with all parentheses shown. We believe that bits of output language will now parse as input if they contain no anonymized variables and no lambda-terms or dependent type notations. Further (version of 3/21/2015) the display of commands and declarations is now indented in a way determined by the number of worlds there are (the more levels of supposition we have, the more we are indented). This should help the reader to understand the scope of identifiers.

7 Type checking and definition expansion

We have already indicated above how entity types are type checked (subject to information about how entity terms are type checked which will be given here).

An undeclared identifier will trigger type check failure.

An entity identifier (declared using the `declare` command) will be assigned the type which has been declared for it (in any world).

An entity term $at_1, \ldots, t_n$, where a is an abstraction identifier of arity n and each t_i is either an entity term or an abstraction identifier of positive arity followed by a comma [unless final], will be typed by the following algorithm: let $[(a_1, \alpha_1), \ldots, (a_n, \alpha_n) \to (x, \tau)]$ be the declared type of a (whether the value x is an entity term or `---` is immaterial). Let $[(t_1, \tau_1), \ldots, (t_n, \tau_n)]$ be the list of t_i's with their computed types. If t_1 does not match α_1, the type check fails and the term cannot be typed. Otherwise type check $[(t_2, \tau_2), \ldots, (t_n, \tau_n)]$ against $[(a_2, \alpha_2^*), \ldots, (a_n, \alpha_n^*) \to (x^*, \tau^*)]$, where the starring denotes the operation of replacing a_1 with t_1 throughout. When the process terminates with matching the empty list against $[x^*, \tau^*]$, return τ^* as the type (notice that this will happen as the first step if a is of arity 0). Type matching does include expanding defined abstractions if necessary. A term t_i which is of the form (abstraction identifier followed by a comma [unless final]) is of course assigned its own abstraction type as declared (and type check fails if it is not defined). Abstraction types match if they can be identified up to changes of name of bound variables.

An entity term $at_1, \ldots, t_n$ where a is a defined abstraction with type $[(a_1, \alpha_1), \ldots, (a_n, \alpha_n) \to (D, \tau)]$ will expand out to the result of replacing each a_i with t_i in D. If an abstraction identifier of positive arity appears followed by a comma [or final] in an argument list, it expands (if this is forced) to its declared type, which is in effect a λ-term. When a λ-term replaces a variable in applied position in a definition expansion, the obvious substitution is carried out. Lestrade can be forced to do this (explicit β-reduction does happen). Substitutions into λ-terms force a change of namespace of its bound variables as in the `construct` command if the substitution makes any change in the term. These changes in namespace prevent bound variable capture.

8 Motivation

`obj` is intended to represent the realm of untyped mathematical objects (if one were implementing ZFC, all one's official objects would live here). `prop` is intended to represent the realm of propositions. `type` is intended to be inhabited by sorts of typed mathematical object. `that` P is intended to represent the type of proofs of (evidence for) the proposition P. `in` T is

intended to be inhabited by objects of the sort denoted by the term T. It is useful to note that the type `prop` and the type constructor `that` have exactly the same relation to each other that `type` does to `in`: the difference is entirely one of intent. The objects of all the types considered in this paragraph are referred to as *entities*.

The other objects handled by Lestrade are the functions or abstractions declared with `construct` and `define`. Abstractions are treated with some caution by Lestrade. The user is not allowed to type λ-terms (or dependent types) herself; these are always generated by the system. No entity actually *is* an abstraction: the user must always declare a construction (making some sort of axiomatic commitment) to involve an abstraction in the definition of an entity (a proof for example). But Lestrade allows one to formulate and adopt such commitments quite freely: one can for example implement quantification over abstraction types quite readily. This is a higher order logical framework.

The relationship between worlds is best described in this way. The parent world and the lower indexed worlds represent objects to which one is currently committed as fixed objects. The objects in the current world are variable or arbitrary objects, which are allowed to vary freely in their declared types. A `construct` or `define` command allows one to write new terms $at_1 \ldots t_n$ in the current world, which vary as the t_i's vary; the denotation of the identifier a in the parent world is a new fixed (not varying) object obtained by abstraction from this complex variable term.

Propositional and first-order logic (and higher-order logic as well) can be implemented using the Curry-Howard isomorphism. This is best discussed along with sample declarations. A constructive logic of course can be implemented in this way, but so can classical logic, just as it was in Automath. It is worth noting that this system generates and maintains proof objects.

There is a philosophical program behind this software. I suggest implicitly that one can take the notion of a variable or arbitrary object seriously: it works in this formalism. This allows a function to be understood in an old-fashioned way as abstracted from a dependent variable expression or a rule. A function is not an infinite table of values which can only be understood after all of its values are understood. Similarly, a universally quantified statement is not an infinitary conjunction which can only be understood after all of its conjuncts are understood (impredicativity is not a real difficulty). A further point is that it is strongly suggested that mathematics (even classical

impredicative two-valued mathematics) can be done in a way which does not involve postulation of completed infinities, with the support of the device of variable objects.

9 Some Remarks

Some things seem be be done with mirrors in Lestrade.

The type declaration dependence of constructed and defined notions is checked only once, when they are declared. This is done by first expanding all defined notions declared in the current world in the type and/or definition of the proposed abstraction [because the type and definition to be recorded for the proposed abstraction will be placed in the parent world, and cannot depend on information which will disappear if the current world is closed], then temporarily cutting down the current world to the argument list of the proposed abstraction and type checking the curtailed current world itself, and the type and/or definition of the proposed object. The type checker will then detect any dependencies that were not taken into account in the argument list by finding undeclared identifiers. The necessary dependence relations between items in the parameter list are enforced magically by requiring that parameters to the proposied abstraction appear in the order in which they were originally declared.

There is no need to parse or type check λ- terms or dependent types in Lestrade, because the user never enters such a term and such terms are known to be soundly typed (mod bugs in the type checker) because they were constructed by Lestrade itself. This is why the definition expansion function does not need to have type checking as it goes.

It may seem odd that λ-terms are treated in effect as subtypes of dependent types, but this is very convenient structurally and it appears that I am making no logical assumptions of any particular moment by doing so.

There are some cautions about the parser. Arguments to abstractions can always be separated by commas (and perhaps this should be universally required, but so far I have not done it). However, abstractions of positive arity may have to be followed by commas to avoid capturing terms after them as arguments, and preceded by commas if they have arity greater than one to avoid being read as infixes and capturing preceding terms. Terms and argument lists may be enclosed in parentheses; note a particular hazard,

which is that if one encloses a first argument in an argument list of length greater than one in parentheses one must enclose the entire argument list in parentheses as well; the parser reads a parenthesis immediately following an abstraction as opening an argument list, not a term. This is not the case for parentheses following an abstraction being used as an infix or mixfix operator. Notice that the display functions will always present an abstraction of arity 2 as an infix operator [unless the first argument supplied to it is an abstraction], although the user is free to enter it as a prefix operator. Note that the left side of a `construct` or `define` command is always written in prefix order (and never with parentheses around the parameters). Parameters like arguments to any abstraction can be comma separated, and commas will be necessary after abstractions with positive arity and before abstractions with arity greater than one. **The parser and display functions do not allowi an abstraction whose first argument is itself an abstraction to be read or displayed as infix or mixfix.**

The display of both commands and Lestrade output (other than error messages) on the console and in log files is indented in a way which indicates the depth of the current world. This should be useful in determining the structure of arguments and the scope of identifiers.

The nomenclature "current world" and "parent world" for world $i+1$ and world i may need to be revisited. The objects and abstractions to which one is currently committed are in the parent world; objects in the current world are hypothetical objects allowed to vary freely.

10 Possible further features

I would like to have the ability to save a closed world with the possibility of opening it again. This does seem to require modifying its namespace to avoid collision with anything that is declared in the interim, or of course searching saved worlds when checking identifier freshness as well as the accessible worlds. Note that this is now implemented.

I would like to support program execution. The most natural way to do this would seem to be to allow declaration of rewrite rules to be applied when definitions are being expanded or type matches are being checked. The kinds of proofs which allow rewrites are easily recognized (actually, this was a bit tricky). This is now implemented, but see the new paper.

11 Some sample log files

I provide some samples of Lestrade interactions. These are sample executable log files. They were run under old versions of Lestrade; I do not plan to redo them for this document.

11.1 Russell's paradox

Russell's paradox. The point here is that one cannot have the constructions `comp` and `comp2` which implement unrestricted comprehension. This file contains its own definitions of basic propositional logic notions in an older style than the following section of logic declarations.

```
>>Inspector Lestrade says:  Welcome to the Lestrade Type Inspector, version of

open
     declare x obj

>>     x:  obj

     construct P x:prop

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

     close
construct set P:obj

>>set:  [(P_2:[(x_1:obj),(---:prop)]),(---:obj)]

declare x obj

>>x:  obj

declare y obj
```

```
>>y:  obj

construct E x y:prop

>>E:  [(x_3:obj),(y_3:obj),(---:prop)]

declare x1 that P x

>>x1:  that P(x)

construct comp P, x x1:that E x set P

>>comp:  [(P_4:[(x_1:obj),(---:prop)]),(x_4:obj),(x1_4:that
>>   P_4(x_4)),(---:that (x_4 E set(P_4)))]

declare x2 that E x set P

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

construct comp2 P, x x2:  that P x

>>comp2:  [(P_5:[(x_1:obj),(---:prop)]),(x_5:obj),(x2_5:that
>>   (x_5 E set(P_5))),(---:that P_5(x_5))]

declare p prop

>>p:  prop

declare q prop

>>q:  prop

construct If p q:prop

>>If:  [(p_6:prop),(q_6:prop),(---:prop)]

construct False:prop
```

```
>>False:  [(---:prop)]

declare pp that p

>>pp:  that p

declare rr that If p q

>>rr:  that (p If q)

construct Mp p q pp rr:that q

>>Mp:  [(p_8:prop),(q_8:prop),(pp_8:that p_8),(rr_8:that
>>   (p_8 If q_8)),(---:that q_8)]

declare absurd that False

>>absurd:  that False

construct Panic p absurd: that p

>>Panic:  [(p_9:prop),(absurd_9:that False),(---:that
>>   p_9)]

define Not p:If p False

>>Not:  [(p_10:prop),((p_10 If False):prop)]

define Russell x:Not E x x

>>Russell:  [(x_11:obj),(Not((x_11 E x_11)):prop)]

open
     declare pp2 that p

>>     pp2:  that p
```

```
     construct Ded pp2:that q

>>     Ded:  [(pp2_12:that p),(---:that q)]

     close
construct Ifproof p q Ded:that If p q

>>Ifproof:  [(p_13:prop),(q_13:prop),(Ded_13:[(pp2_12:that
>>   p_13),(---:that q_13)]),(---:that (p_13 If q_13))]

open
     define R: set Russell

>>     R:  [(set(Russell):obj)]

     declare R1 that E set Russell, set Russell

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

     define R2 R1:comp2 Russell, set Russell, R1

>>     R2:  [(R1_15:that (set(Russell) E set(Russell))),
>>        (comp2(Russell,set(Russell),R1_15):that Russell(set(Russell)))]

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

>>     R3:  [(R1_16:that (set(Russell) E set(Russell))),
>>        (Mp((set(Russell) E set(Russell)),False,R1_16,
>>        R2(R1_16)):that False)]

     close
define R4:Ifproof E set Russell, set Russell, False R3

>>R4:  [(Ifproof((set(Russell) E set(Russell)),False,
>>   [(R1_16:that (set(Russell) E set(Russell))),(Mp((set(Russell)
>>   E set(Russell)),False,R1_16,comp2(Russell,set(Russell),
```

```
>>   R1_16)):that False)]):that ((set(Russell) E set(Russell))
>>   If False))]

define R5:comp Russell, set Russell, R4

>>R5:  [(comp(Russell,set(Russell),R4):that (set(Russell)
>>   E set(Russell)))]

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

>>R6:  [(Mp((set(Russell) E set(Russell)),False,R5,R4):that
>>   False)]

quit
```

11.2 Logic declarations and proofs

This file contains all the declarations of primitive concepts and rules and definitions of derived rules from my manual of logical style for students.

```
>>Inspector Lestrade says:  Welcome to the Lestrade Type Inspector, version of

comment  The treatment of logic in my style manual
declare p prop

>>p:  prop

declare q prop

>>q:  prop

comment   conjunction/and
construct & p q:prop
```

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

declare pp that p

>>pp:  that p

declare qq that q

>>qq:  that q

construct Andproof p q pp qq:that p & q

>>Andproof:  [(p_2:prop),(q_2:prop),(pp_2:that p_2),(qq_2:that
>>   q_2),(---:that (p_2 & q_2))]

declare rr that p & q

>>rr:  that (p & q)

construct Andelim1 p q rr:that p

>>Andelim1:  [(p_3:prop),(q_3:prop),(rr_3:that (p_3 &
>>   q_3)),(---:that p_3)]

construct Andelim2 p q rr: that q

>>Andelim2:  [(p_4:prop),(q_4:prop),(rr_4:that (p_4 &
>>   q_4)),(---:that q_4)]

comment implication/if...then...
construct -> p q:prop

>>->:  [(p_5:prop),(q_5:prop),(---:prop)]

open
     declare pp2 that p
```

```
>>     pp2:  that p

     construct Ded pp2:that q

>>     Ded:  [(pp2_6:that p),(---:that q)]

     close
construct Ifproof p q Ded:that p -> q

>>Ifproof:  [(p_7:prop),(q_7:prop),(Ded_7:[(pp2_6:that
>>   p_7),(---:that q_7)]),(---:that (p_7 -> q_7))]

declare ss that p -> q

>>ss:  that (p -> q)

construct Mp p q pp ss:that q

>>Mp:  [(p_8:prop),(q_8:prop),(pp_8:that p_8),(ss_8:that
>>   (p_8 -> q_8)),(---:that q_8)]

comment negation (not defining it this time)
construct ~ p:prop

>>~:  [(p_9:prop),(---:prop)]

construct ??:prop

>>??:  [(---:prop)]

open
     declare pp2 that p

>>     pp2:  that p

     construct contra pp2:that ??
```

```
>>     contra:  [(pp2_11:that p),(---:that ??)]

     close
construct Negproof p contra:that ~p

>>Negproof:  [(p_12:prop),(contra_12:[(pp2_11:that p_12),
>>   (---:that ??)]),(---:that ~(p_12))]

declare tt that ~p

>>tt:  that ~(p)

construct Contradiction p pp tt:that ??

>>Contradiction:  [(p_13:prop),(pp_13:that p_13),(tt_13:that
>>   ~(p_13)),(---:that ??)]

declare absurd that ??

>>absurd:  that ??

construct Panic p absurd:that p

>>Panic:  [(p_14:prop),(absurd_14:that ??),(---:that
>>   p_14)]

declare maybe that ~ ~p

>>maybe:  that ~(~(p))

construct Dneg p maybe:that p

>>Dneg:  [(p_15:prop),(maybe_15:that ~(~(p_15))),(---:that
>>   p_15)]

comment basic rules for disjunction
construct v p q:prop
```

```
>>v:  [(p_16:prop),(q_16:prop),(---:prop)]

construct Addition1 p q pp:that p v q

>>Addition1:  [(p_17:prop),(q_17:prop),(pp_17:that p_17),
>>   (---:that (p_17 v q_17))]

construct Addition2 p q qq:that p v q

>>Addition2:  [(p_18:prop),(q_18:prop),(qq_18:that q_18),
>>   (---:that (p_18 v q_18))]

comment prepare for proof by cases
declare uu that p v q

>>uu:  that (p v q)

declare r prop

>>r:  prop

open
     declare pp2 that p

>>     pp2:  that p

     construct case1 pp2:that r

>>     case1:  [(pp2_19:that p),(---:that r)]

     close
open
     declare qq2 that q

>>     qq2:  that q
```

```
     construct case2 qq2:that r

>>     case2:  [(qq2_20:that q),(---:that r)]

     close
construct Cases p,q,uu,r,case1,case2:that r

>>Cases:  [(p_21:prop),(q_21:prop),(uu_21:that (p_21
>>   v q_21)),(r_21:prop),(case1_21:[(pp2_19:that p_21),
>>   (---:that r_21)]),(case2_21:[(qq2_20:that q_21),(---:that
>>   r_21)]),(---:that r_21)]

comment derived rules for implication
open
     declare notq that ~q

>>     notq:  that ~(q)

     construct Ded2 notq:that ~p

>>     Ded2:  [(notq_22:that ~(q)),(---:that ~(p))]

     declare pp2 that p

>>     pp2:  that p

     open
          declare notq2 that ~q

>>          notq2:  that ~(q)

          define hah notq2:Contradiction p pp2, Ded2 notq2

>>          hah:  [(notq2_23:that ~(q)),(Contradiction(p,
>>             pp2,Ded2(notq2_23)):that ??)]

          close
```

```
     define hah2 pp2:Negproof ~q hah

>>     hah2:  [(pp2_24:that p),((~(q) Negproof [(notq2_25:that
>>        ~(q)),(Contradiction(p,pp2_24,Ded2(notq2_25)):that
>>        ??)]):that ~(~(q)))]

     define hah3 pp2:Dneg q hah2 pp2

>>     hah3:  [(pp2_28:that p),((q Dneg hah2(pp2_28)):that
>>        q)]

     close
define Indirect p q Ded2:Ifproof p q hah3

>>Indirect:  [(p_31:prop),(q_31:prop),(Ded2_31:[(notq_22:that
>>   ~(q_31)),(---:that ~(p_31))]),(Ifproof(p_31,q_31,[(pp2_41:that
>>   p_31),((q_31 Dneg (~(q_31) Negproof [(notq2_43:that
>>   ~(q_31)),(Contradiction(p_31,pp2_41,Ded2_31(notq2_43)):that
>>   ??)])):that q_31)]):that (p_31 -> q_31))]

declare vv that ~q

>>vv:  that ~(q)

open
     declare pp2 that p

>>     pp2:  that p

     define hmmm pp2:Mp p q pp2 ss

>>     hmmm:  [(pp2_44:that p),(Mp(p,q,pp2_44,ss):that
>>        q)]

     define hmmm2 pp2:Contradiction q hmmm pp2 vv

>>     hmmm2:  [(pp2_45:that p),(Contradiction(q,hmmm(pp2_45),
```

```
>>        vv):that ??)]

     close
define Mt p q ss vv:Negproof p hmmm2

>>Mt:  [(p_46:prop),(q_46:prop),(ss_46:that (p_46 ->
>>   q_46)),(vv_46:that ~(q_46)),((p_46 Negproof [(pp2_50:that
>>   p_46),(Contradiction(q_46,Mp(p_46,q_46,pp2_50,ss_46),
>>   vv_46):that ??)]):that ~(p_46))]

comment derived rules for disjunction
open
     declare notq that ~q

>>     notq:  that ~(q)

     construct ruleout notq:that p

>>     ruleout:  [(notq_51:that ~(q)),(---:that p)]

     declare neither that ~(p v q)

>>     neither:  that ~((p v q))

     open
          declare notq2 that ~q

>>          notq2:  that ~(q)

          define problem notq2:Addition1 p q ruleout notq2

>>          problem:  [(notq2_52:that ~(q)),(Addition1(p,
>>             q,ruleout(notq2_52)):that (p v q))]

          define problem2 notq2: Contradiction p v q, problem notq2, neither

>>          problem2:  [(notq2_53:that ~(q)),(Contradiction((p
```

```
>>             v q),problem(notq2_53),neither):that ??)]

          close
     define problem3 neither:Negproof ~q, problem2

>>     problem3:  [(neither_54:that ~((p v q))),((~(q)
>>        Negproof [(notq2_55:that ~(q)),(Contradiction((p
>>        v q),Addition1(p,q,ruleout(notq2_55)),neither_54):that
>>        ??)]):that ~(~(q)))]

     define problem4 neither:Dneg q problem3 neither

>>     problem4:  [(neither_58:that ~((p v q))),((q Dneg
>>        problem3(neither_58)):that q)]

     define problem5 neither: Addition2 p q problem4 neither

>>     problem5:  [(neither_59:that ~((p v q))),(Addition2(p,
>>        q,problem4(neither_59)):that (p v q))]

     define disaster neither:Contradiction p v q, problem5 neither, neither

>>     disaster:  [(neither_60:that ~((p v q))),(Contradiction((p
>>        v q),problem5(neither_60),neither_60):that ??)]

     close
define Orproof p q ruleout:Dneg p v q, Negproof ~(p v q), disaster

>>Orproof:  [(p_63:prop),(q_63:prop),(ruleout_63:[(notq_51:that
>>   ~(q_63)),(---:that p_63)]),(((p_63 v q_63) Dneg (~((p_63
>>   v q_63)) Negproof [(neither_73:that ~((p_63 v q_63))),
>>   (Contradiction((p_63 v q_63),Addition2(p_63,q_63,(q_63
>>   Dneg (~(q_63) Negproof [(notq2_75:that ~(q_63)),(Contradiction((p_63
>>   v q_63),Addition1(p_63,q_63,ruleout_63(notq2_75)),neither_73):that
>>   ??)])))),neither_73):that ??)])):that (p_63 v q_63))]

comment rules of disjunctive syllogism
```

```
declare notp that ~p

>>notp:  that ~(p)

open
     declare pp2 that p

>>     pp2:  that p

     define qfollows pp2:Panic q, Contradiction p pp2 notp

>>     qfollows:  [(pp2_76:that p),((q Panic Contradiction(p,
>>        pp2_76,notp)):that q)]

     declare qq2 that q

>>     qq2:  that q

     define qfollow2 qq2:qq2

>>     qfollow2:  [(qq2_77:that q),(qq2_77:that q)]

     close
define Ds1 p q uu notp:Cases p q uu q, qfollows, qfollow2

>>Ds1:  [(p_78:prop),(q_78:prop),(uu_78:that (p_78 v
>>   q_78)),(notp_78:that ~(p_78)),(Cases(p_78,q_78,uu_78,
>>   q_78,[(pp2_82:that p_78),((q_78 Panic Contradiction(p_78,
>>   pp2_82,notp_78)):that q_78)],[(qq2_81:that q_78),(qq2_81:that
>>   q_78)]):that q_78)]

declare notq that ~q

>>notq:  that ~(q)

open
     declare pp2 that p
```

```
>>     pp2:  that p

     define notqcase1 pp2:pp2

>>     notqcase1:  [(pp2_83:that p),(pp2_83:that p)]

     declare qq2 that q

>>     qq2:  that q

     define notqcase2 qq2: Panic p, Contradiction q qq2 notq

>>     notqcase2:  [(qq2_84:that q),((p Panic Contradiction(q,
>>        qq2_84,notq)):that p)]

     close
define Ds2 p q uu notq:Cases p q uu p, notqcase1, notqcase2

>>Ds2:  [(p_85:prop),(q_85:prop),(uu_85:that (p_85 v
>>   q_85)),(notq_85:that ~(q_85)),(Cases(p_85,q_85,uu_85,
>>   p_85,[(pp2_88:that p_85),(pp2_88:that p_85)],[(qq2_89:that
>>   q_85),((p_85 Panic Contradiction(q_85,qq2_89,notq_85)):that
>>   p_85)]):that p_85)]

comment delayed Orproof2 needed
open
     declare notp2 that ~p

>>     notp2:  that ~(p)

     construct ruleout2 notp2:that q

>>     ruleout2:  [(notp2_90:that ~(p)),(---:that q)]

     declare neither that ~(p v q)
```

```
>>     neither:  that ~((p v q))

     open
          declare notp3 that ~p

>>          notp3:  that ~(p)

          define problem notp3:Addition2 p q, ruleout2 notp3

>>          problem:  [(notp3_91:that ~(p)),(Addition2(p,
>>             q,ruleout2(notp3_91)):that (p v q))]

          define problem2 notp3:Contradiction p v q,problem notp3, neither

>>          problem2:  [(notp3_92:that ~(p)),(Contradiction((p
>>             v q),problem(notp3_92),neither):that ??)]

          close
     define problema3 neither:Dneg p, Negproof ~p problem2

>>     problema3:  [(neither_93:that ~((p v q))),((p
>>        Dneg (~(p) Negproof [(notp3_94:that ~(p)),(Contradiction((p
>>        v q),Addition2(p,q,ruleout2(notp3_94)),neither_93):that
>>        ??)])):that p)]


     define problema4 neither:Addition1 p q, problema3 neither

>>     problema4:  [(neither_97:that ~((p v q))),(Addition1(p,
>>        q,problema3(neither_97)):that (p v q))]

     define problema5 neither:Contradiction p v q, problema4 neither, neither

>>     problema5:  [(neither_98:that ~((p v q))),(Contradiction((p
>>        v q),problema4(neither_98),neither_98):that ??)]

     close
```

```
define Orproof2 p q ruleout2:Dneg p v q, Negproof ~(p v q), problema5

>>Orproof2:  [(p_101:prop),(q_101:prop),(ruleout2_101:[(notp2_90:that
>>   ~(p_101)),(---:that q_101)]),(((p_101 v q_101) Dneg
>>   (~((p_101 v q_101)) Negproof [(neither_111:that ~((p_101
>>   v q_101))),(Contradiction((p_101 v q_101),Addition1(p_101,
>>   q_101,(p_101 Dneg (~(p_101) Negproof [(notp3_113:that
>>   ~(p_101)),(Contradiction((p_101 v q_101),Addition2(p_101,
>>   q_101,ruleout2_101(notp3_113)),neither_111):that ??)])))),
>>   neither_111):that ??)])):that (p_101 v q_101))]

open
     declare notp2 that ~p

>>     notp2:  that ~(p)

     define samenotp notp2:notp2

>>     samenotp:  [(notp2_114:that ~(p)),(notp2_114:that
>>        ~(p))]

     close
define Excmid p:Orproof2 p, ~p, samenotp

>>Excmid:  [(p_139:prop),(Orproof2(p_139,~(p_139),[(notp2_140:that
>>   ~(p_139)),(notp2_140:that ~(p_139))]):that (p_139 v
>>   ~(p_139)))]

quit
```

11.3 Working on Landau...

Landau up to proposition 2. This file is preceded by a copy of the logical declarations given above (previous versions contained embedded duplicates of earlier forms of those logical declarations).

```
comment we assume the totality of natural numbers
construct Nat:type

>>Nat:  [(---:type)]

declare x in Nat

>>x:  in Nat

declare y in Nat

>>y:  in Nat

open
     declare z in Nat

>>     z:  in Nat

     construct P z:prop

>>     P:  [(z_142:in Nat),(---:prop)]

     close
construct = x y:prop

>>=:  [(x_143:in Nat),(y_143:in Nat),(---:prop)]

declare eq that = x y

>>eq:  that (x = y)

declare px that P x

>>px:  that P(x)

construct subs x y P, eq px:that P y
```

```
>>subs:  [(x_144:in Nat),(y_144:in Nat),(P_144:[(z_142:in
>>   Nat),(---:prop)]),(eq_144:that (x_144 = y_144)),(px_144:that
>>   P_144(x_144)),(---:that P_144(y_144))]

open
     open
          declare u in Nat

>>          u:  in Nat

          construct P1 u:prop

>>          P1:  [(u_145:in Nat),(---:prop)]

          close
     declare px1 that P1 x

>>     px1:  that P1(x)

     construct Eq P1, px1:that P1 y

>>     Eq:  [(P1_146:[(u_145:in Nat),(---:prop)]),(px1_146:that
>>        P1_146(x)),(---:that P1_146(y))]

     close
construct Eqproof x y Eq:that = x y

>>Eqproof:  [(x_147:in Nat),(y_147:in Nat),(Eq_147:[(P1_146:[(u_145:in
>>   Nat),(---:prop)]),(px1_146:that P1_146(x_147)),(---:that
>>   P1_146(y_147))]),(---:that (x_147 = y_147))]

open
     open
          declare u in Nat

>>          u:  in Nat
```

```
          construct P2 u:prop

>>          P2:  [(u_148:in Nat),(---:prop)]

          close
     declare refl1 that P2 x

>>     refl1:  that P2(x)

     define refl2 P2, refl1:refl1

>>     refl2:  [(P2_149:[(u_148:in Nat),(---:prop)]),
>>        (refl1_149:that P2_149(x)),(refl1_149:that P2_149(x))]

     close
define Refl x:Eqproof x x refl2

>>Refl:  [(x_150:in Nat),(Eqproof(x_150,x_150,[(P2_151:[(u_148:in
>>   Nat),(---:prop)]),(refl1_151:that P2_151(x_150)),(refl1_151:that
>>   P2_151(x_150))]):that (x_150 = x_150))]

open
     declare symm1 that = x y

>>     symm1:  that (x = y)

     open
          declare u in Nat

>>          u:  in Nat

          define P3 u:=y u

>>          P3:  [(u_152:in Nat),((y = u_152):prop)]

          define P4 u:=u x
```

```
>>          P4:  [(u_153:in Nat),((u_153 = x):prop)]

          close
     define symm2 symm1:subs x y P4, symm1 Refl x x

>>     symm2:  [(symm1_156:that (x = y)),(subs(x,y,[(u_153:in
>>        Nat),((u_153 = x):prop)],symm1_156,Refl(x)):that
>>        (y = x))]

     close
declare symm3 that = x y

>>symm3:  that (x = y)

define symm x y symm3:symm2 symm3

>>symm:  [(x_157:in Nat),(y_157:in Nat),(symm3_157:that
>>   (x_157 = y_157)),(subs(x_157,y_157,[(u_158:in Nat),
>>   ((u_158 = x_157):prop)],symm3_157,Refl(x_157)):that
>>   (y_157 = x_157))]

declare z in Nat

>>z:  in Nat

open
     declare trans1 that = x y

>>     trans1:  that (x = y)

     declare trans2 that = y z

>>     trans2:  that (y = z)

     open
          declare u in Nat
```

```
>>          u:  in Nat

          define P5 u:= x u

>>          P5:  [(u_159:in Nat),((x = u_159):prop)]

          close
     define trans3 trans1 trans2:subs y z P5, trans2 trans1

>>     trans3:  [(trans1_160:that (x = y)),(trans2_160:that
>>        (y = z)),(subs(y,z,[(u_159:in Nat),((x = u_159):prop)],
>>        trans2_160,trans1_160):that (x = z))]

     close
declare t1 that =x y

>>t1:  that (x = y)

declare t2 that =y z

>>t2:  that (y = z)

define trans x y z t1 t2:trans3 t1 t2

>>trans:  [(x_161:in Nat),(y_161:in Nat),(z_161:in Nat),
>>   (t1_161:that (x_161 = y_161)),(t2_161:that (y_161 =
>>   z_161)),(subs(y_161,z_161,[(u_162:in Nat),((x_161 =
>>   u_162):prop)],t2_161,t1_161):that (x_161 = z_161))]

construct 1: in Nat

>>1:  [(---:in Nat)]

construct succ x:in Nat

>>succ:  [(x_164:in Nat),(---:in Nat)]
```

```
construct notone x: that ~ = succ x 1

>>notone:  [(x_165:in Nat),(---:that ~((succ(x_165) =
>>   1)))]

declare samesucc1 that = succ x succ y

>>samesucc1:  that (succ(x) = succ(y))

construct samesucc x y samesucc1:that = x y

>>samesucc:  [(x_166:in Nat),(y_166:in Nat),(samesucc1_166:that
>>   (succ(x_166) = succ(y_166))),(---:that (x_166 = y_166))]

open
     declare u in Nat

>>     u:  in Nat

     construct Indp u:prop

>>     Indp:  [(u_167:in Nat),(---:prop)]

     close
declare basis that Indp 1

>>basis:  that Indp(1)

open
     declare u in Nat

>>     u:  in Nat

     declare indhyp that Indp u

>>     indhyp:  that Indp(u)
```

```
     construct indstep u indhyp:that Indp succ u

>>     indstep:  [(u_168:in Nat),(indhyp_168:that Indp(u_168)),
>>        (---:that Indp(succ(u_168)))]

     close
construct Induction x Indp, basis indstep:that Indp x

>>Induction:  [(x_169:in Nat),(Indp_169:[(u_167:in Nat),
>>   (---:prop)]),(basis_169:that Indp_169(1)),(indstep_169:[(u_168:in
>>   Nat),(indhyp_168:that Indp_169(u_168)),(---:that Indp_169(succ(u_168)))]),
>>   (---:that Indp_169(x_169))]

comment try to prove Satz 1
define =/= x y: ~(x=y)

>>=/=:  [(x_170:in Nat),(y_170:in Nat),(~((x_170 = y_170)):prop)]

open
     declare hyp that x =/= y

>>     hyp:  that (x =/= y)

     open
          declare counterhyp that succ x = succ y

>>          counterhyp:  that (succ(x) = succ(y))

          define oops counterhyp:Contradiction x=y, (samesucc x y counterhyp),

>>          oops:  [(counterhyp_171:that (succ(x) = succ(y))),
>>             (Contradiction((x = y),samesucc(x,y,counterhyp_171),
>>             hyp):that ??)]

          close
     define conc hyp: Negproof succ x = succ y, oops
```

```
>>     conc:  [(hyp_172:that (x =/= y)),(((succ(x) =
>>        succ(y)) Negproof [(counterhyp_173:that (succ(x)
>>        = succ(y))),(Contradiction((x = y),samesucc(x,
>>        y,counterhyp_173),hyp_172):that ??)]):that ~((succ(x)
>>        = succ(y))))]

     close
define satz1 x y:Ifproof x=/=y, succ x =/= succ y,conc

>>satz1:  [(x_174:in Nat),(y_174:in Nat),(Ifproof((x_174
>>   =/= y_174),(succ(x_174) =/= succ(y_174)),[(hyp_180:that
>>   (x_174 =/= y_174)),(((succ(x_174) = succ(y_174)) Negproof
>>   [(counterhyp_182:that (succ(x_174) = succ(y_174))),
>>   (Contradiction((x_174 = y_174),samesucc(x_174,y_174,
>>   counterhyp_182),hyp_180):that ??)]):that ~((succ(x_174)
>>   = succ(y_174))))]):that ((x_174 =/= y_174) -> (succ(x_174)
>>   =/= succ(y_174))))]

define theprop x:succ x =/= x

>>theprop:  [(x_183:in Nat),((succ(x_183) =/= x_183):prop)]

open
     declare u in Nat

>>     u:  in Nat

     declare xx that theprop u

>>     xx:  that theprop(u)

     define yy u:satz1 succ u u

>>     yy:  [(u_200:in Nat),((succ(u_200) satz1 u_200):that
>>        ((succ(u_200) =/= u_200) -> (succ(succ(u_200))
>>        =/= succ(u_200))))]
```

```
     define zz u xx:Mp succ u =/= u, succ succ u =/= succ u, xx, yy u

>>     zz:  [(u_201:in Nat),(xx_201:that theprop(u_201)),
>>        (Mp((succ(u_201) =/= u_201),(succ(succ(u_201))
>>        =/= succ(u_201)),xx_201,yy(u_201)):that (succ(succ(u_201))
>>        =/= succ(u_201)))]

     define zz2 u xx:Mp theprop u, theprop succ u, xx, yy u

>>     zz2:  [(u_202:in Nat),(xx_202:that theprop(u_202)),
>>        (Mp(theprop(u_202),theprop(succ(u_202)),xx_202,
>>        yy(u_202)):that theprop(succ(u_202)))]

     close
define satz2 x:Induction x, theprop, notone 1, zz2

>>satz2:  [(x_203:in Nat),(Induction(x_203,theprop,notone(1),
>>   [(u_202:in Nat),(xx_202:that theprop(u_202)),(Mp(theprop(u_202),
>>   theprop(succ(u_202)),xx_202,(succ(u_202) satz1 u_202)):that
>>   theprop(succ(u_202)))]):that theprop(x_203))]

define badsatz2 x:Induction x,theprop,notone 1,zz

>>badsatz2:  [(x_204:in Nat),(Induction(x_204,theprop,
>>   notone(1),[(u_201:in Nat),(xx_201:that theprop(u_201)),
>>   (Mp((succ(u_201) =/= u_201),(succ(succ(u_201)) =/=
>>   succ(u_201)),xx_201,(succ(u_201) satz1 u_201)):that
>>   (succ(succ(u_201)) =/= succ(u_201)))]):that theprop(x_204))]

comment not bad any more after bugs fixed...
comment existence and uniqueness stuff -- definition of "the"; quantifiers
open
     declare n in Nat

>>     n:  in Nat
```

```
     construct Uprop n:prop

>>     Uprop:  [(n_205:in Nat),(---:prop)]

     close
construct the Uprop:in Nat

>>the:  [(Uprop_206:[(n_205:in Nat),(---:prop)]),(---:in
>>   Nat)]

open
     declare m in Nat

>>     m:  in Nat

     declare n in Nat

>>     n:  in Nat

     declare mm that Uprop m

>>     mm:  that Uprop(m)

     declare nn that Uprop n

>>     nn:  that Uprop(n)

     construct allthesame m n mm nn:that m=n

>>     allthesame:  [(m_207:in Nat),(n_207:in Nat),(mm_207:that
>>        Uprop(m_207)),(nn_207:that Uprop(n_207)),(---:that
>>        (m_207 = n_207))]

     close
declare w in Nat

>>w:  in Nat
```

```
declare ww that Uprop w

>>ww:  that Uprop(w)

construct Theproof Uprop, allthesame, w ww:that Uprop the Uprop

>>Theproof:  [(Uprop_208:[(n_205:in Nat),(---:prop)]),
>>   (allthesame_208:[(m_207:in Nat),(n_207:in Nat),(mm_207:that
>>   Uprop_208(m_207)),(nn_207:that Uprop_208(n_207)),(---:that
>>   (m_207 = n_207))]),(w_208:in Nat),(ww_208:that Uprop_208(w_208)),
>>   (---:that Uprop_208(the(Uprop_208)))]

quit
```