The Lestrade Type Inspector – Manual and Discussion

M. Randall Holmes

March 24, 2015

version: 3/24/2015 8 PM added a section of (I hope) helpful remarks; changed text to reflect changes in display of types and \( \lambda \)-terms and the restriction that first arguments of infix operators will not be abstractions.

Contents

1 Introduction 2
2 Access to the software 2
3 Commands which can be given in the Lestrade interface 3
4 The input language 4
5 Worlds 6
6 Declaration commands 6
7 Type checking and definition expansion 8
8 Motivation 9
9 Some Remarks 10
10 Possible further features 12
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 $\lambda$-term or dependent type. I originally thought that this represented a greater difference from Automath than it actually does.

2 Access to the software

Lestrade is contained in a Moscow ML 2.01 program lestrade.sml (an older version implementation.sml has a simpler and therefore more reliable parser). When this has been compiled, loaded and opened, type interface <filename> to reach the Lestrade interface (a log will be recorded to <filename>.mlg). The command readme <file1> <file2> will clear the Lestrade context then execute the commands in <file1>.mlg and log those commands and any commands the user subsequently types to <file2>.mlg. 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 readme 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.
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.

**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.

**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.
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; a side effect is that zero-ary abstractions cannot appear as items in <arglist>s).

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 identifier 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 $a_1 \ldots t_n$
(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. In the infix/mixfix notation, the
arguments after the abstraction may optionally be separated by commas but
cannot be enclosed in parentheses. 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.

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 abstrac-
tion 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 pre-
vent 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.

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.

6 Declaration commands

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

The command construct $x t_1 \ldots t_n : \tau$ 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 $\tau_i$ for each $t_i$. Each primitive identifier declared in the current world on which $\tau$ or any of the $\tau_i$'s depends must appear as one of the $t_i$'s. This is enforced by type checking the entity type $\tau$ 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 $\tau$ 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 $[(t^*_1, \tau^*_1), \ldots, (t^*_n, \tau^*_n) \rightarrow (\ldots, \tau^*)]$, where the star represents the operation of replacing each $t_i$ wherever 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 $\tau^*$ 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 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 $\tau_i$ for each $t_i$. Each primitive identifier declared in the current world on which the entity term $D$ or its type $\tau$ or any of the $\tau_i$'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 $\tau$ 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 $t_i$ wherever 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 \texttt{declare} command) will be assigned the type which has been declared for it (in any world).

An entity term \(a_1, \ldots, t_n\), 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 \([((a_1, \alpha_1), \ldots, (a_n, \alpha_n) \rightarrow (x, \tau))]\) 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 \(\alpha_1\), the type check fails and the term cannot be typed. Otherwise type check \([(t_2, \tau_2), \ldots, (t_n, \tau_n)]\) against \([(a_2, \alpha_2^*), \ldots, (a_n, \alpha_n^*) \rightarrow (x^*, \tau^*)]\), 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 \(\tau^*\) 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 $a_1, \ldots, t_n$ where $a$ is a defined abstraction with type $[(a_1, \alpha_1), \ldots, (a_n, \alpha_n) \rightarrow (D, \tau)]$ 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 $\lambda$-term. When a $\lambda$-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 $\beta$-reduction does happen). Substitutions into $\lambda$-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). $\text{prop}$ is intended to represent the realm of propositions. $\text{type}$ is intended to be inhabited by sorts of typed mathematical object. $\text{that } P$ is intended to represent the type of proofs of (evidence for) the proposition $P$. $\text{in } T$ is intended to be inhabited by objects of the sort denoted by the term $T$. It is useful to note that the type $\text{prop}$ and the type constructor $\text{that}$ have exactly the same relation to each other that $\text{type}$ does to $\text{in}$: the difference is entirely one of intent. The objects of all the types considered in this paragraph are referred to as $\text{entities}$.

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 $\lambda$-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 $a_1 \ldots a_n$ in the current world, which vary as the $a_i$'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 proposed 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. Note that 3/24/2015 I made a change to the parser and display functions not allowing 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.

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.

11 Some sample log files

I provide some samples of Lestrade interactions. These are sample executable log files.

11.1 Russell’s paradox

Russell’s paradox. The point here is that one cannot have the constructions \texttt{comp} and \texttt{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))

13
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,
       R2(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 3/21/2015 3 pm

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

17
\[ \text{construct Andelim1 } p \ q \ rr: \text{that } p \]
\[ \text{construct Andelim1: } [(p_3: \text{prop}), (q_3: \text{prop}), (rr_3: \text{that } (p_3 \& q_3)), (---: \text{that } p_3)] \]

\[ \text{construct Andelim2 } p \ q \ rr: \text{that } q \]
\[ \text{construct Andelim2: } [(p_4: \text{prop}), (q_4: \text{prop}), (rr_4: \text{that } (p_4 \& q_4)), (---: \text{that } q_4)] \]

\text{comment implication/if...then...}
\text{construct } \rightarrow p \ q: \text{prop}

\[ \text{construct } \rightarrow: [(p_5: \text{prop}), (q_5: \text{prop}), (---: \text{prop})] \]

\text{open}
\text{declare } pp2 \text{ that } p

\[ \text{pp2: that } p \]
\[ \text{construct Ded } pp2: \text{that } q \]

\[ \text{Ded: } [(pp2_6: \text{that } p), (---: \text{that } q)] \]

\text{close}
\text{construct Ifproof } p \ q \ \text{Ded:that } p \rightarrow q

\[ \text{Ifproof: } [(p_7: \text{prop}), (q_7: \text{prop}), (\text{Ded}_7: [(pp2_6: \text{that } p_7), (---: \text{that } q_7)]), (---: \text{that } (p_7 \rightarrow q_7))] \]

\text{declare } ss \text{ that } p \rightarrow q

\[ \text{ss: that } (p \rightarrow q) \]
\[ \text{construct } \text{Mp } p \ q \ pp \ ss: \text{that } q \]
>>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)])]

comment derived rules for implication
open
   declare notq that ~q
>>  notq: that ~(q)
construct Ded2 notq:that \neg p

\[ \text{Ded2: } [(\text{notq}\_22: \text{that } \neg(q)), (\neg\neg: \text{that } \neg(p))] \]

declare pp2 that p

\[ \text{pp2: that } p \]

open

declare notq2 that \neg q

\[ \text{notq2: that } \neg(q) \]

define hah notq2:Contradiction p pp2, Ded2 notq2

\[ \text{hah: } [(\text{notq2}\_23: \text{that } \neg(q)), (\text{Contradiction}(p, pp2, \text{Ded2}(\text{notq2}\_23)): \text{that } ??)] \]

close

define hah2 pp2:Negproof \neg q hah

\[ \text{hah2: } [(\text{pp2}\_24: \text{that } p), (\neg(q) \text{ Negproof } [(\text{notq2}\_25: \text{that } \neg(q)), (\text{Contradiction}(p, pp2, \text{Ded2}(\text{notq2}\_25)): \text{that } ??)]: \text{that } \neg(\neg(q)))]] \]

define hah3 pp2:Dneg q hah2 pp2

\[ \text{hah3: } [(\text{pp2}\_28: \text{that } p), (\text{Dneg hah2}(\text{pp2}\_28)): \text{that } q] \]

close

define Indirect p q Ded2:Ifproof p q hah3

\[ \text{Indirect: } [(\text{p}\_31: \text{prop}), (\text{q}\_31: \text{prop}), (\text{Ded2}\_31: [(\text{notq}\_22: \text{that } \neg(q)), (\neg\neg: \text{that } \neg(p))], (\text{Ifproof}(\text{p}\_31, \text{q}\_31), [(\text{pp2}\_41: \text{that } \neg(q)), (\text{Dneg } (\neg(q) \text{ Negproof } [(\text{notq2}\_43: \text{that } \neg(q))]: \text{that } ??))])]} \]
declare vv that \neg 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 \rightarrow q_46)), (vv_46: that \neg (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 \neg (p_46)]

comment derived rules for disjunction
open
declare notq that \neg q

>> notq: that \neg (q)

construct ruleout notq: that p

23
\[
\text{ruleout: } [(\text{not}_q_51: \text{that } \neg(q)), (-\text{-}: \text{that } p)]
\]

declare neither that \(\neg(p \lor q)\)

\[
\text{neither: } \text{that } \neg((p \lor q))
\]

open

\[
\text{declare notq2 that } \neg(q)
\]

\[
\text{notq2: that } \neg(q)
\]

declare notq2 that \(\neg(q)\)

\[
\text{define problem notq2: Addition1 } p \ q \ \text{ruleout notq2}
\]

\[
\text{problem: } [(\text{not}_q_52: \text{that } \neg(q)), (\text{Addition1}(p, q, \text{ruleout(not}_q_52)): \text{that } (p \lor q))]
\]

declare notq2 that \(\neg(q)\)

\[
\text{define problem2 notq2: Contradiction } p \lor q, \text{ problem notq2, neither}
\]

\[
\text{problem2: } [(\text{not}_q_53: \text{that } \neg(q)), (\text{Contradiction}(p \lor q, \text{problem(not}_q_53), \text{neither}): \text{that } ??)]
\]

close

\[
\text{define problem3 neither: Negproof } \neg(q), \text{ problem2}
\]

\[
\text{problem3: } [(\text{neither}_54: \text{that } \neg((p \lor q))), (\neg(q) \text{ Negproof } [(\text{not}_q_55: \text{that } \neg(q)), (\text{Contradiction}(p \lor q, \text{Addition1}(p, q, \text{ruleout(not}_q_55)), \text{neither}_54): \text{that } ??]): \text{that } \neg(\neg(q)))]
\]

define problem4 neither: Dneg \(q\) problem3 neither

\[
\text{problem4: } [(\text{neither}_58: \text{that } \neg((p \lor q))), ((q \text{ Dneg problem3(neither}_58)): \text{that } q)]
\]

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 ~((p_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

\[\text{qfollow2: } [(qq2_{77}:\text{that } q), (qq2_{77}:\text{that } q)]\]

close

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

\[\text{Ds1: } [(p_{78}:\text{prop}), (q_{78}:\text{prop}), (uu_{78}:\text{that } (p_{78} \lor q_{78})), (notp_{78}:\text{that } \neg(p_{78})), \text{Cases}(p_{78}, q_{78}, uu_{78}, q_{78}), [(pp2_{82}:\text{that } p_{78}), ((q_{78} \text{ Panic Contradiction}(p_{78}, pp2_{82}, notp_{78})):\text{that } q_{78})], [(qq2_{81}:\text{that } q_{78}), (qq2_{81}:\text{that } q_{78})]:\text{that } q_{78}]\]

declare notq that \neg q

\[\text{notq: } \neg(q)\]

open

declare pp2 that p

\[\text{pp2: } \text{that } p\]

define notqcase1 pp2:pp2

\[\text{notqcase1: } [(pp2_{83}:\text{that } p), (pp2_{83}:\text{that } p)]\]

declare qq2 that q

\[\text{qq2: } \text{that } q\]

define notqcase2 qq2: Panic p, Contradiction q qq2 notq

\[\text{notqcase2: } [(qq2_{84}:\text{that } q), ((p \text{ Panic Contradiction}(q, qq2_{84}, notq)):\text{that } p)]\]

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 ??)]
define problema3 neither:Dneg p, Negproof ~p problem2

define problema4 neither:Addition1 p q, problema3 neither

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

define 0rproof2 p q ruleout2:Dneg p v q, Negproof ~(p v q), problema5

declare notp2 that ~p

define samenotp notp2:notp2

28
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

declare x in Nat

declare y in Nat

open
  declare z in Nat
\[
\begin{align*}
\text{construct } & P \ z : \text{prop} \\
\text{construct } & x \ y : \text{prop} \\
\text{construct } & \text{subs } x \ y \ P, \ \text{eq} \ \text{px} : \text{that } P \ y \\
\text{construct } & u : \text{in Nat} \\
\text{construct } & P_1 \ u : \text{prop} \\
\text{construct } & P_1 : \ [(u_{145} : \text{in Nat}),(---:\text{prop})] \\
\end{align*}
\]
\[
\text{construct Eq P1, px1:that P1 y}
\]

\[
\text{Eq: } [(P1_{146}:[(u_{145}:\text{in Nat}),(-:-\text{prop})]),(px1_{146}:\text{that P1}_{146}(x)),(-:-\text{that P1}_{146}(y))]}
\]

\[
\text{construct Eqproof x y Eq:that = x y}
\]

\[
\text{Eqproof: } [(x_{147}:\text{in Nat}),(y_{147}:\text{in Nat}),(Eq_{147}:[(P1_{146}:[(u_{145}:\text{in Nat}),(-:-\text{prop})]),(px1_{146}:\text{that P1}_{146}(x_{147})),(-:-\text{that P1}_{146}(y_{147}))]),(-:-\text{that } (x_{147} = y_{147}))]
\]

\[
\text{open}
\]

\[
\text{open}
\]

\[
\text{declare u in Nat}
\]

\[
\text{u: in Nat}
\]

\[
\text{construct P2 u:prop}
\]

\[
\text{P2: } [(u_{148}:\text{in Nat}),(-:-\text{prop})]
\]

\[
\text{close}
\]

\[
\text{declare refl1 that P2 x}
\]

\[
\text{refl1: that P2(x)}
\]

\[
\text{define refl2 P2, refl1:refl1}
\]

\[
\text{refl2: } [(P2_{149}:[(u_{148}:\text{in Nat}),(-:-\text{prop})]),(refl1_{149}:\text{that P2}_{149}(x)),(refl1_{149}:\text{that P2}_{149}(x))]}
\]

\[
\text{close}
\]

\[
\text{define Refl x:Eqproof x x refl2}
\]
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
\textbf{symm}: \((x_{157}: \text{in} \ \text{Nat}), (y_{157}: \text{in} \ \text{Nat}), (\text{symm3}_{157}: \text{that} (x_{157} = y_{157})), (\text{subs}(x_{157}, y_{157}, [(u_{158}: \text{in} \ \text{Nat}), ((u_{158} = x_{157}): \text{prop})], \text{symm3}_{157}, \text{Refl}(x_{157})):\text{that} (y_{157} = x_{157}))\) \\

declare z \text{ in} \ \text{Nat} \\
\textbf{z}: \text{ in} \ \text{Nat} \\
open \\
declare \text{trans1} \text{ that} = x \ y \\
\textbf{trans1}: \text{ that} (x = y) \\
declare \text{trans2} \text{ that} = y \ z \\
\textbf{trans2}: \text{ that} (y = z) \\
open \\
declare u \text{ in} \ \text{Nat} \\
\textbf{u}: \text{ in} \ \text{Nat} \\
define \text{P5} u::= x \ u \\
\textbf{P5}: \ [(u_{159}: \text{in} \ \text{Nat}), ((x = u_{159}): \text{prop})] \\
close \\
define \text{trans3} \text{ trans1} \text{ trans2}: \text{subs} y \ z \ \text{P5}, \ \text{trans2} \ \text{trans1} \\
\textbf{trans3}: \ [(\text{trans1}_{160}: \text{that} (x = y)), (\text{trans2}_{160}: \text{that} (y = z)), (\text{subs}(y, z, [(u_{159}: \text{in} \ \text{Nat}), ((x = u_{159}): \text{prop})], \text{trans2}_{160}, \text{trans1}_{160}): \text{that} (x = z))] \\
close \\
declare \text{t1} \text{ that} = x \ y
\[
\text{\texttt{t1}}: \quad \text{that } (x = y) \\
\text{declare t2 that } =y z \\
\text{\texttt{t2}}: \quad \text{that } (y = z) \\
\text{define \texttt{trans }x y z t1 t2:trans3 t1 t2} \\
\text{\texttt{trans}}: \quad \left[(x_{161}:\text{in } \text{Nat}), (y_{161}:\text{in } \text{Nat}), (z_{161}:\text{in } \text{Nat}), \right. \\
\left. (t1_{161}:\text{that } (x_{161} = y_{161})), (t2_{161}:\text{that } (y_{161} = z_{161})), (\text{subs}(y_{161}, z_{161}, [(u_{162}:\text{in } \text{Nat}), ((x_{161} = u_{162}) : \text{prop})], t2_{161}, t1_{161}):\text{that } (x_{161} = z_{161})]\right) \\
\text{construct 1: in } \text{Nat} \\
\text{\texttt{1}}: \quad \left[(-:-:\text{in } \text{Nat})\right] \\
\text{construct \texttt{succ }x:in } \text{Nat} \\
\text{\texttt{succ}}: \quad \left[(x_{164}:\text{in } \text{Nat}), (-:-:\text{in } \text{Nat})\right] \\
\text{construct \texttt{notone }x: that } \sim = \text{succ } x 1 \\
\text{\texttt{notone}}: \quad \left[(x_{165}:\text{in } \text{Nat}), (-:-:\text{that } \sim((\text{succ}(x_{165}) = 1))\right)\right] \\
\text{declare \texttt{samesucc1 that } = \text{succ } x \text{ succ } y \\
\text{\texttt{samesucc1}}: \quad \text{that } (\text{succ}(x) = \text{succ}(y)) \\
\text{construct \texttt{samesucc }x y \text{ samesucc1} that } = x y \\
\text{\texttt{samesucc}}: \quad \left[(x_{166}:\text{in } \text{Nat}), (y_{166}:\text{in } \text{Nat}), (\text{samesucc1}_{166}:\text{that } (\text{succ}(x_{166}) = \text{succ}(y_{166}))), (-:-:\text{that } (x_{166} = y_{166}))\right]\right) \\
\text{open} \\
\quad \text{\texttt{declare u in } \text{Nat}} \\
\]
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)
\(\forall x_170, y_170 \in \text{Nat}, \neg ((x_170 = y_170)) \)\n
open

declare hyp that \(x =/= y\)

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

open

declare counterhyp that succ \(x =\) succ \(y\)

counterhyp: that \((\text{succ}(x) = \text{succ}(y))\)

define oops counterhyp: Contradiction \(x = y\), \((\text{samesucc} x y \text{ counterhyp})\),

oops: \([(\text{counterhyp}_{171}: \text{that} (\text{succ}(x) = \text{succ}(y))),

(\text{Contradiction}((x = y), \text{samesucc}(x, y, \text{counterhyp}_{171}),

\text{hyp}) : \text{that} ??)]\)

close

define conc hyp: Negproof \(\text{succ} x = \text{succ} y\), oops

conc: \([(\text{hyp}_{172}: \text{that} (x =/= y)), ((\text{succ}(x) =\text{ succ}(y)))\]

Negproof \([(\text{counterhyp}_{173}: \text{that} (\text{succ}(x) =\text{ succ}(y)))\]

\((\text{Contradiction}((x = y), \text{samesucc}(x, y, \text{counterhyp}_{173}),

\text{hyp}) : \text{that} ??)) : \text{that} \neg ((\text{succ}(x) =\text{ succ}(y)))\]

close

define satz1 \(x y\): Ifproof \(x =/= y\), \((\text{succ} x =/= \text{succ} y)\), conc

satz1: \([(x_{174} : \text{in Nat}), (y_{174} : \text{in Nat}), (\text{Ifproof}((x_{174} =\text{ succ}(y_{174})))\]

Negproof \([(\text{counterhyp}_{182}: \text{that} (\text{succ}(x_{174}) =\text{ succ}(y_{174})))\]

\((\text{Contradiction}((x_{174} = y_{174}), \text{samesucc}(x_{174}, y_{174},

\text{counterhyp}_{182}), \text{hyp}_{180}) : \text{that} ??)) : \text{that} \neg ((\text{succ}(x_{174})

\text{ = succ}(y_{174})))]) : \text{that} ((x_{174} =\text{ succ}(y_{174}) \rightarrow (\text{succ}(x_{174})

36
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: satz_1 succ u u

> yy: [(u_200: in Nat), ((succ(u_200) satz_1 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 satz_2 x: Induction x, theprop, not one 1, zz2

37
\begin{verbatim}
>>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
\end{verbatim}
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

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)]),
(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