Lestrade Type Inspector Source with comments

Randall Holmes

1/2/2020: eliminated some redundant actions in Cleantype and equaltypes functions. 8/16/2019 2 pm This is the literate programming version with labelled Lestrade execution blocks. Currently having qualms about hypothetical rewrites. Fixed margin crash. Prettyprints command lines. Compact type display toggle. Removed restraints on saved move names. Tabulates type and definition computations. Making saved theories complete, not just move 0 declarations. Aggressive backups to avoid file disasters: make sure there is a folder called backups in the folder where Lestrade is run. Supercompact display generating scripts with just command lines is supported. Deferred definition facility implemented in a preliminary form; this might be dangerous.

This document is the working source for the Lestrade Type Inspector in ML. If the initial `%` above is deleted (without deleting the following ML comment opener), the file should work under the appropriate ML interpreter. The ML text immediately below indicates how to adapt the file to run in Moscow ML 2.10 or PolyML; the default is Moscow ML 2.01.

This is now the source of the working file `lestrade.sml`: all edits are made here to the current version of Lestrade.

Please note that it is just the dated comments that are in a tiny typeface: when the code starts below, it is visible!

```
 *)

(* 8/16/2019 literate version*)

(* This is the ML source for the Lestrade Type Inspector. *)

(* moscow ml preamble *)

fun desome x = x;

(* BEGIN for PolyML decomment this;
for Moscow ML 2.10 in addition comment out first line

open PolyML
```

```
fun desome (SOME x) = x |

desome NONE = "";

END *)

fun Inputline x = desome(TextIO.inputLine x);

(* end moscow ml preamble *)

(* smlnj premable

fun desome x = x;

fun makestring s = Int.toString s;

(* BEGIN for PolyML decomment this;
for Moscow ML 2.10 in addition comment out first line

open PolyML *)

fun desome (SOME x) = x |

desome NONE = "";

(* END *)

fun Inputline x = desome(TextIO.inputLine x);

 end smlnj preamble *)



(*


*)

(*
dated notes

1/9/2020  I am going to edit this code carefully to make sure I have consistent expectations about serial numbers, preparatory to trying to get the let update to work.

1/2/2020  I want to solve the definition explosion problem by storing definition bodies as let terms.  In my first pass, I was storing entire types
as let terms, but this will create a constant problem of unpacking the normally used parts of types.  What explodes is the body of a definition, and this
is also seldom used.  This makes it seem that my let terms should be of ML type Entity, not ML type Type as in my first pass.

There is a local versus global problem:  currently, the substitution process automatically explodes definitions in the next move, a bottom up process.  I want this not to happen
in definition bodies, to be replaced with a top down let term construction.  But I evidently still want the local expansion process in types which are not in definition
bodies?

I have done significant repairs to avoid operations on definition bodies which are being discarded, in the Cleartype and equaltype ML functions.  There might be further
corrections to be made along these lines?
```

8/16/2019 Editing comments, very likely no code changes. I did make some small code changes to ImportTheory, and added type checks to equalentities in the case of deferred definitions.

There is a bug in readfile depth in readback and readkoob. This won't cause any bad proofs, but it ought to be fixed sometime.

A user command which extends the world list by adding a new empty world 0 might be very useful. This would be handy when passing to talk of models of the very theory you are workin to WORLDNAMES!

I should improve the deferred definition feature so that it can complete deferred definitions by matching construction arguments, and perhaps in other situations (target the matchings which the implicit argument feature can handle?). Extreme caution is needed because declaration checks in define take advantage of limitations on what a user can enter. In particular, we have to watch for free occurrences of bound variables sneaking into redefinitions.

I observe that the function addtoworld which adds an item to a move and reindexes it is declared, but never used.

I should set things up to make it so that readback and readkoob will not run if the user has paused.

8/16/2019 the deferred definition feature is installed: use with caution, it may be dangerous. The command ddefine with the same conditions on its parameters as postulate introduces a defined notion whose definition is eventually to be inferred by the matching features of the prover. Matching of the yet-to-be-defined notion as an abstraction argument with an abstraction does not yet work.

This involved internalizing a version of the define command into the central logic engine; this is not a trivial maneuver, and may yet be bugged. Some prerequisites for the define command had to be moved (or copied) into the logic engine.

I also installed a user command clearbreakout in the interface (to be used with caution): the deferred definition matching mechanism does not work if BREAKOUT is true (since a version of a user command is being invoked, and we want to know if it raised errors), and the user may know that previous error conditions are harmless.

When the deferred definition is actually made it is made without implicit arguments (the original deferred definition is allowed to have implicit arguments). The implicit argument pattern can of course be restored by making another definition. It also has to be made in the move where the matching occurs, or one may fail to expand notions that need to be expanded in the proposed definition.

Installing the ability to define an identifier directly as a lambda-term would solve the problem of matching yet-to-be-defined abstractions with other abstractions or lambda terms at the same time. This would require automatic generation of a suitable argument list from the lambda-term.

It is also important to notice that the term matching a yet-to-be-defined notion must make sense in the context in which the yet-to-be-defined notion was actually proposed.

Notice that some code had to be moved around, which may mean that a pass with rewriting of comments on local ML functions is needed, since some are now in unexpected places.

I do believe this version is quite safe (or at least, as safe as the previous version) as long as the deferred definition feature is not used.

2 pm made the truncateto function more exact, to defend against circular definitions being accepted by Define0.

8/10/2019 working on version with deferred definitions.

8/8/2019

This is just code optimization. Notably, I removed some dupication of effort in findimplitargs, but it does not seem to affect the bottlenecks in performance.

8/7/2019

technical refinement to searches of long lists. I am not sure whether it actually improves performance. I seem to be adapting well to readbook and readkoob :-)

8/6/2019

Installing aggressive creation of backups to avoid destroying files when readback is run. Also trying out readbook and readkoob instead of readfile2 and readback2.

A directory called backups is now required to exist for Lestrade to execute the readback or readkoob command: an I/O error will be raised if this directory does not exist.

Installed supercompact display mode which produces scripts with only the command lines (by suppressing the action of showdec completely). goal and test commands will still echo type information to scripts.

8/5/2019

Saved theories now contain all theory information, not just move 0 declarations, so load can be used more effectively to avoid running slow logs repeatedly.

More funny business about rewrites in moves of positive index. The functions for making identifiers in a saved move adjoinable with the context at the time the move is re-opened were never set up to accommodate rewrites. The situation with hypothetical rewrites is fraught enough that I am simply not allowing name extensions to correct such moves: if any need for name revision is detected in opening a move with native rewrites, an error will be raised.

8/3/2019 Eliminated the name collision error message sometimes raised by the open command, which is as far as I can tell simply a distraction.

7/24/2019 No essential update. Fixed a bug: the clearallcaches function was not clearing caches!

7/22/2019 Problem with saved move management identified. When a move is cleared which has been saved, moves subsequent to that one need also to be cleared. A similar situation can arise in the save command itself. This version now includes tabulation of type and definition computations, though I do not know whether this actually helps.

7/21/2019 Removed restraints from names of saved moves. This is important for restraining definition expansion. A saved move can have its default name. The only remnant of this is that clearcurrent will not load a saved move with its default name; it will clear the next move as expected. Also ClearAll now behaves correctly when there are saved moves. This is a real demonstration that this feature is needed. There is another version which adds tabulation for the type and definition functions, which I tried out for improved performance when the Zermelo proof was being very slow; this does not seem to have been the issue. But it might be wanted.

The goal and test commands have been added. goal will display a type (usually the type of a proposition to be proved). test will display an argument and display its type. These are very useful tools for incremental development of proofs without rerunning the entire file; both are

important for commenting on proof files (already noted on the 13th).

7/19/2019 removed echo from showdec. made compact display default.

7/18/2019 minor change to margin relaxation.

7/17/2019 This version has automatic prettyprinting of command lines. There are still some issues with it.

Second update adds the command {\tt compactdisplay}, which toggles between showing definition bodies in type declarations (and so in log files) and not showing them.
This is hugely advantageous for reading log files.

7/15/2019 This version makes showdec a persistent logged command.

7/13/2019 This version adds new interface commands goal and test. goal X just causes the type X to be displayed. test X causes the argument X to be displayed
with its type.

7/10/2019 This version fixes the crash which happened when indentations became longer than the margin: the margin is now automatically increased in this case.

6/21/2019 Still musing about hypothetical rewrites. I think the double close option is not the best one: more sublty, simply block the declaration commands
in the presence of hypothetical rewrites. Make it so that any command which records declarations in the previous move is blocked if there are rewrites in
the next move. Changed code as indicated. This should mean that it is now possible to save environments in which one has played with hypothetical rewrites
and return to them without hazard.

6/11/2019 Now contains code for a version of the Close command which clears the next move after closing if it finds any rewrites: the idea being
that exporting anything from that next move to the new current move would bring in dependencies on hypothetical rewrites which are not recorded.
Ideally, pruning of the current move could be restricted to things appearing after the declaration of the rewrite, but that is a good deal of work, and saving
moves before declaring hypothetical rewrites would defend against loss of data (I checked this and I think it will: it should be reflex to save a context
if one wants to introduce hypothetical rewrites, as that context will be erased when you return to it).

Now commented out the code in rewritep and Rewrited so that one can declare hypothetical rewrites -- with the effect that
contexts in which hypothetical rewrites have been declared will be erased as untrustworthy by the Close command. This ought at some
point to be tested to see that it actually works, which will be a bit involved.

6/1/2019 There is a difficulty with rewrite rules. The problem is that dependencies of definitions and declarations
on rewrite rules are not recorded, so it should either be impossible to declare hypothetical rewrite rules or declarations
which might depend on them must be discarded when a world is closed. I dealt with this by allowing rewritep and Rewrited commands
to be issued only at the top level. With a little care, the approach with hypothetical rewrites allowed but things possibly depending on
them discarded can probably be supported. The machinery to maintain rewrite rule lists for each world remains in place.

Philosophically, I think allowing and trying to manage rewrite rule dependencies would be a fundamental complication.

7/19/2018 planning to develop a depostulate command. This takes an identifier already declared by postulate and attempts to define it;
the definition may include special identifiers which are declared or defined in such a way as to make things type correctly: these will be
constructed or defined with special names which signal that they are goals in a deconstruction process under way, and should themselves
be deconstructed.

This requires two new kinds of identifier, matchable undeclared identifiers (ending or beginning with ??, possibly) which when declared or defined
are renamed ending with ? -- identifiers ending in ? being fully privileged, the ? being a signal that these were created in a deconstruction. The deep technical issue is what kind
semi-automation of goal directed reasoning. I am hoping that the matching of ?? identifiers which leads to declaration of ? identifiers
can happen entirely in equaltypes (really in equalentities, it seems). The way in which special identifiers are labelled is modifiable:
having matchable ones starting with ! and ones to be further deconstructed starting with ? recommends itself.

5/14/2018 Lestrade execution blocks now should begin with \begin{verbatim}Lestrade execution:
For the moment, it will still read files with just \begin{verbatim}, but will correct them.

3/29/2018 fixed very minor issue with attachment of quit to ends of new LaTeX log files.

11/23/2017 Purely internal issue, fixed badly written recursion in parsing of user entered abstraction terms.

11/22/2017 added ability to suppress definition bodies in displays, useful when definition expansion gets out of hand.

10/29/17 rewrited didn't work for mysterious reasons. I reimplemented it in a way which
does clearly work, but this does mean that the postulate and rewritep commands at least for
the time being can declare extended identifiers.

10/28/17 fix to rewrites to allow rewriting of complex constant terms.

10/23/2017 I ought to deal with the point that in fact rewrite patterns are never atomic terms. There are no errors due to
this, but there are unnecessary tests. Extraneous code commented out.

A philosophical point with no code yet. Rewriting inside lambda terms might be supportable, but it would be necessary to identify
terms with bound variables in which which might match a pattern with appropriate instantiation of the variable,
in order to block rewrites which might break confluence. There is also the issue that perhaps {\tt fullrewrite} should
work from the top down.

10/22/2017 No code change: describing a feature to be implemented.
  A true analogue to the Automath context feature would be an ability
 to abbreviate lists of arguments.
The idea is that one could use the command {\tt vector} followed by
 an identifier then by a list of arguments to declare a name for
that list of arguments. This name would be
used only at term input: the idea is that the tokenize function would
expand an occurrence of a vector identifier into the appropriate
stream of tokens with alternating commas derived
from its definition. Vectors could then appear at any point in an argument list
(including argument lists in bound variable terms), and
no prover function would ever handle them. The only additional attention required would
be to maintain a list of lists of vectors (analogous for example to the list of lists of rewrites)
 maintained in parallel with the context, and to make sure that one could not declare
an identifier conflicting with a vector or a vector conflicting with an identifier.
 The only question is whether this is really useful: it certainly would be analogous

 to the context mechanism of Automath, and it might be useful in translations of
Automath books. Would it be likely to be used in native Lestrade books?

I further note that the new bound variable features should make translation
of Automath books far easier, and I should work on translations with this in mind.

10/20/2017 tidied up readback commands and testing for end of line in commands.

I believe it may be the case that it is now never necessary to open a move: it may be the case that the bound variable
term features support everything that can be done with moves. I'm not certain.

10/19/2017: The declare command will now take a function sort. Function sorts are represented with bound variables taken from the next move and
body of the abstraction simply an object term. This has the same unprincipled character as use of lambda terms as function arguments,
in that it subverts the move model, but we should understand just as in the previous case that variables in the next move are being in effect
cloned to deeper moves which are not officially being opened.

10/17/2017: moved most of the stray utilities manipulating the master ML types to one place right after the master declaration.

10/17/2017: possible extensions. It would not be hard to enable parsing of function sorts
in much the way that λ-terms were managed, and extend
the {\tt declare} command to allow declaration of function variables.
It would also not be hard to allow declarations of vectors
(atomic abbreviations for repeatedly used argument lists;
a vector could only appear as the tail of an argument list?
Vectors would play a role similar to that of Automath contexts.

10/16/2017: I finished commenting the parser. 8 pm commented the user commands in {\tt readline}.

10/15/2017 I think I fixed multisubstypelist. I believe that I simply had a misconception.

MAJOR UPDATE: I have installed parsing of lambda terms as arguments. I have some concerns about whether some ML functions
may not have cases for lambda terms since they are not expected to be in input. However, simple cases appear to work. The format
of a lambda term is a list of variables separated by commas, declared at the next move, followed by => followed by the body
of the lambda abstraction, all enclosed in brackets.

10/14/2017: I have curbed the excesses of namespace numbering. The new variable Maxfreshindex keeps track of the largest value ever taken by Freshindex; commands that open new en

10/8/2017 headrewrite acts on rewritten atomic constants as well, confluence issue.

10/7/2017 tabulated the fullrewrite and headrewrite commands, so execution behavior should be improved.
  This needs to be tested.   Computing Fibonacci numbers would be a good example.
There is no change whatever in the user interface or observed behavior (barring bugs): this is an internal efficiency issue.

Variable generation in rewrite commands retuned.

9/17/2017 Starting work on streamlining the rewritep and rewrited commands.

Successfully got it to automatically declare the variable name
formerly given as final argument to rewritep. A side effect is that one now
can declare variables which are "extended". Rewrited should also be fixed at this point.

The rewritep and rewrited commands no longer require the predicate argument or the
final variable separated by a colon; these commands generate their own auxiliary variables.
  They still need caching for efficient execution. rewrited needs to be tested:
 there seem to be no tests in my current corpus.

9/14/2017 The rewrite commands need attention. The predicate variable used
should be automatically generated, as the evidence for this predicate holding of
the rewrite target is. Moreover, the names should perhaps be automatically generated
as well (from the name for the object witnessing the rewrite rule, perhaps).
The rewrited rule should perhaps apply only where the rewrite source and target are of sorts
other than prop and type.

9/10/2017 Now readily adaptable for different versions of ML

9/4/2017 projected improvements

the rewrite function should be tabulated for better execution behavior.

the nasty task of adding user entered lambda-terms (with sort inference)

the fairly easy task of adding identifiers representing argument lists -- this is closer to the Automath context device.

when there are both .lti and .tex files with the same name, the readfile commands ought to complain?

9/4/2017 minor edit -- changed a couple of references to ML types which had
been incorrectly changed to "sort" on 9/3. The terminology in the comments and
the Lestrade output should now be in line with the terminology in current
documentation, though the old terminology is still reflected in names of ML
types and functions. For this purpose we provide a translation table

old new

entity   object
abstraction   function
type sort
world   move

current world next move
parent world last move (sometimes "current move", but not here).

The new terminology "current move" for the last move would be confusing given
the old terminology "current world" for the next move. The new terminology "entity"
for "object or function" of course conflicts with the old terminology "entity" for
object.

9/3/2017 editing pass for readability.
I am systematically replacing or supplementing the old
terms entity, abstraction, world, and type with
object, function, move, and sort in comments and Lestrade messages
but not in code.

9/2/2017 slight fix to one line so that the code can be embedded in the manual

8/30/2017 no code change beyond the development of readfile2 which handles LaTeX
documents. Removing most old comments,
other than those which seem to represent cautions or needs for testing.

I am concerned about the scope of the rewrite commands: I am contemplating restricting
the scope of rewrited (not of rewritep) to types (in tau) and (that p).
The rationale is that the formulation of ambiguous TST in foundationsintro.tex
becomes contradictory (I think) if rewrited can be applied to the constructor Ambiguity.

8/4/2017 working on alternative readfile commands which can handle LeTeX documents,
 executing whatever appears in verbatim blocks. It is implemented!
Command readfile2 handles .tex files. The .tex files used need to end with
quit after the end of the document. I made it so that readfile2 will continue
reading after a line with \ at the end. This feature is now preserved in the output.

1/11/17 The Poly/ML version now pretty prints with periods instead of spaces.
  This version is slightly modified so that it executes scripts generated by the Poly/ML version.
  WARNING: only the 2.01 version is up to date: I need to update the
other versions.

12/2/2016 disabled the code restoring prop/type symmetry
re the rewrite command. It is still there in a comment.

11/30/16 Matching of lambda-terms is installed. I'm confident that it is correct but unsure how to test it. Would it make
sense to allow new defined expressions in patterns which will expand to lambda-terms which can be matched? It
also surely has effects in implicit argument matching.

Think about allowing definition expansion in patterns. I have experimentally done this. Testing becomes advisable.

Some rather elaborate testing with either implicit arguments or rewriting will be required to see whether
lambda term matching actually works. Elaborate testing of the rewrite feature is probably a good idea anyway.

11/29/2016 All sorts of name conflict error checks are still in the code but should never actually be invoked.

11/28/2016 The refined definitions of stringdef and stringage are probably no longer
needed (name conflicts can no longer actually occur), but I'll leave them as they are for the moment.

Instead of setting up readfile to nest, I set up the load and import commands so that their error messages
tell you which files need to be read before the given file can be read successfully. I don't really want
readfile commands issued inside scripts.

11/15/2016 restored symmetry between prop and type in the rewrite feature by allowing one place
type constructors to play the same role as predicates (one place proposition constructors). [this was subsequently retracted]

11/2 (no code change) At this point, the implied argument inference function is exactly what it will
ever be, mod debugging. More powerful recursions on nested function types/lambda terms extending
matching and type computations would automatically make it stronger. In a certain sense, I am at a principled
stopping point: with the exception of one place in the implicit argument inference function, I never do structural
recursion on variable binding terms in a way which takes into account the local types of bound variables. Any
further progress on implicit argument inference would involve such recursion in one way or another. It is not
clear to me that practical reasoning in the system requires more.

Another note: it is intellectually sound to let the reindexing feature scrub unwanted definition information
from types, because in fact it is completely ignored in determining whether types are the same: equaltypes is only
used with the "true" option when lambda terms are being compared. I do think I know where the leak is, but I do not
need to fix it in the implicit argument inference function.

Another note: the rewriting function, at my leisure, should be tabulated for sensible recursive behavior.

11/2 (no code change) A practical idea: add an optional further argument to
the declaration commands for comments, which would be displayed by showdec. Alternatively,
add a command specifically for commenting on declarations.

11/1 further note (no code change) Think about elaborate features which need testing.

These include: the last iteration of implicit argument inference.

The rewriting feature: notably rewrited. The rewriting feature would ultimately need
refinements of its execution model (tables of previously computed rewrites to avoid recursion performance
problems).

The next upgrade is the axiomatic dependence and implementation idea. It might be
further improved by having an option to give an implementation but not allow its innards
to be used (abstract data type security). This might be done neatly by changing the dependencies
of the constructed object but not its actual type information (perhaps hide the implementation
in the dependency information).

The performance issues which are encountered with any attempt to change the isapp test

```
in entsubs are interesting.

11/1 (no code change yet) At this point no new direct expansion of findimplicitargs is needed.  Improvements
in substitution and matching will now automatically drive better implicit argument deduction.

(no code change so far)  Introduce a command which allows an implementation of a primitive
as a defined object of the same type to be given.
When such an implementation is given, the implementation feature
is restricted to apply only to primitives whose sort information involves
the primitive implemented, until all such primitives
have been implemented.  This forces a package
of related primitives to be implemented as a bloc.  This would require an implementation command,
a command to view the list of primitives to which implementation is currently restricted
(if there is one) and a mechanism for keeping track of dependencies of primitives,
under which the dependencies of a defined notion
would be the union of the sets of dependencies of the primitives
it mentions and the dependencies of a primitive notion would be the union of the
sets of dependencies of those primitives whose sort information
mentions the primitive (this is opposite what one might expect).

10/20  Parser refined so that the colon is always optional in the postulate command:
the parser knows that an argument list has ended when it encounters a reserved identifier.

10/9  another projected change:  allow other methods of expressing abstraction arguments,
 both application terms with missing arguments (interpreted as if curried) and
non-polymorphic abstractions (require that all types be deducible without explicit types
on input variables, as in the rewrite system.  No code change yet.

10/2  projected possible changes, no modification in code yet.
A dependency system for type 0 postulates and definitions:  the
idea would be to be able to implement constructed objects and
functions by giving definitions with the right types.  Defined objects
depend on the primitive used in their declarations; constructed objects
seem to depend in a sense on other constructed objects which mention
them in their sorts:  at least, it seems that an implement command
would require implementations of all such constructions to be given.

The other possibility I am considering is overloading.

User-entered lambda terms remain a desideratum if I can figure out
to do them neatly.

Testing of rewrited is needed.

9/27 major upgrades:  changed display so that
implicit arguments of functions applied in sort displays
are not shown (unless the command showimplicit is run).
Fixed a bug in the implicit arguments mechanism, so that
yet more arguments can be deduced successfully.

8/20/2016  Attempt to debug possible problems with
interaction of implicit argument and rewriting features.

This needs testing:  examples of rewritep and rewrited
commands with implicit arguments present are needed.
My belief is that rewritep was already set up to work correctly;
rewrited needed an argument list fix inserted,
which is now there but needs testing.

8/12/2016 Installed the version toggles as user commands.  They do make sense.


The reindexing change is perilous:  I need to be sure that all
situations where substitutions are made into abstraction types
 are preceded by bound variable renaming.


--END dated notes *)

(*



*)

(*file and system message utility functions *)

fun fileexists s = OS.FileSys.access ((s^".lti"), []);

(* alternative version for processing LaTeX documents *)
```

```
fun fileexists2 s = OS.FileSys.access ((s^".tex"), []);

val READING = ref false;

(* controls the greeting message when you enter the interface *)

val GREETED=ref false;

(* the file to which all system activity is logged *)

val LOGFILE = ref (TextIO.openOut("default"));

(* close the log file *)

fun closelog() = (TextIO.flushOut(!LOGFILE);
TextIO.closeOut(!LOGFILE);LOGFILE:=TextIO.openOut("default"));

(* say = system messages.   These go to standard output and also
 as temporary comments (ignored and not persisting when logs are
executed) in the log. *)

fun Flush() = (TextIO.flushOut(TextIO.stdOut);TextIO.flushOut(!LOGFILE));

fun say s = (TextIO.output(TextIO.stdOut,"\nInspector Lestrade says:  "^s^"\n\n");
Flush();TextIO.output(!LOGFILE,"\n>> Inspector Lestrade says:  "^s^"\n\n");Flush());

fun saynoreturn s = (TextIO.output(TextIO.stdOut,"\nInspector Lestrade says:  "^s);
Flush();TextIO.output(!LOGFILE,"\n>> Inspector Lestrade says:  "^s^"\n\n");Flush());

(* saypause = system messages which are errors; these will terminate
 scripts being run *)

val BREAKOUT = ref false;

fun saypause s = (saynoreturn (s^"\n>> Hit return to continue");if (!BREAKOUT) = false
then Inputline(TextIO.stdIn) else "";BREAKOUT:=true)

(* the current version.  This is also the greeting the system gives
(and puts at the head of the log) when you enter the interface *)

(* USER COMMAND *)

fun versiondate() = say
("\n>> Welcome to the Lestrade Type Inspector,\n>>  "^
"\n>> User entered lambda terms (arguments)"^
 "\n>> and function sorts (in declare)!"^
```

```
"\n>> literate programming with LaTeX comments in ML source"^
"\n>> version of  8/16/2019\n>>  9:30 am Boise time\n");

(*
```

The first block consists of input/output utilities, in the most general sense (file handling and system messages).

There are commands `fileexists` and `fileexists2` which check for the existence of files with extension `.lti` (Lestrade log or script files) and `.tex` (Lestrade log or script files with LaTeX documentation).

The toggle `READING` controls the behavior of the `readfile2` command (below) which reads log files with LaTeX documentation, telling it when it is reading Lestrade commands rather than echoing LaTeX text.

The toggle `GREETED` tells Lestrade whether the `versiondate` greeting has been given (it is used to avoid unnecessary repetitions of the greeting).

The variable `LOGFILE` represents the log file to which Lestrade activity is being recorded. The `closelog` command closes the log file.

The `Flush` command flushes the standard output and the log file.

The `say` command delivers system messages without pause, but with a return. System messages are logged but as nonpersisting comments.

The `saynoreturn` command delivers system messages without a following return (they do still have following return in the log file: is this an issue?).

The value `BREAKOUT` is used to signal that one should break out of execution of `readfile` or `readfile2` (an error has been raised which causes Lestrade to break out of execution of a file as a Lestrade script).

The `saypause` command, intended for error messages in particular, delivers system messages with a pause (the user must hit return). This command terminates execution of any file being run as a Lestrade script.

The `versiondate` command gives version information. It is used as the system greeting.

```
*)

(* Lestrade version toggles.  These are now internal Lestrade commands. *)

val REWRITEVER = ref true;

val IMPLICITVER = ref true;

fun basic() = (REWRITEVER := false;  IMPLICITVER:=false);

fun explicit() = (REWRITEVER:=true;IMPLICITVER:=false);

fun fullversion() = (REWRITEVER := true;IMPLICITVER:=true);
```

```
(*
```

These commands are version toggles, of which I make no particular use at the moment. `REWRITEVER` is a toggle which is supposed to turn on the rewriting features. `IMPLICITVER` is a toggle which is suppose to turn on the implicit argument feature. One can see by examination which features are turned on in the basic, explicit, and full versions. In practice, I run the full version.

```
*)

(* projections *)

fun pi13(x,y,z) = x;

fun pi23(x,y,z) = y;

fun pi33(x,y,z) = z;

(* find the sort associated with an identifier in a move or argument list *)

(* an item (n,t,u) in one of these lists has n the numerical
age of the item, t a term (usually a suitably packaged identifier, but there is
an exception) and u a sort. *)

fun find s nil = nil |

find s ((n,t,u)::L) = if s=t then [u] else find s L;

fun findandreplace s u nil = nil |

findandreplace s u ((n,t,v)::L)=

if s=t then ((n,t,u)::L)

else ((n,t,v)::(findandreplace s u L));

fun findandpurge s nil = nil |

findandpurge s ((n,t,u)::L) = if s=t then L
else ((n,t,u)::(findandpurge s L));
```

```
(* more general find function for, e.g., matching lists *)

fun abstractfind s nil = nil |

abstractfind s ((t,u)::L) = if s=t then [u] else abstractfind s L;

(* drop function for general lists *)

fun abstractdrop s nil = nil |

abstractdrop s ((t,u)::L) =

   if s = t then abstractdrop s L
   else (t,u)::(abstractdrop s L);

fun abstractdrop1 s nil = nil |

abstractdrop1 s ((t,u)::L) =

   if s = t then L
   else (t,u)::(abstractdrop1 s L);

fun Abstractfind s LP =

let val U = abstractfind s (!LP) in

if U = nil then nil

else (LP := (s,hd U)::(abstractdrop1 s (!LP));U)

end

fun listextends nil x = true |

listextends x nil = false |

listextends (a::L) (b::M) = (a=b) andalso listextends L M;

fun abstractdrop2 s nil = nil |

abstractdrop2 s ((t,u)::L) =

if listextends (rev s) (rev t) then abstractdrop2 s L

else (t,u)::(abstractdrop2 s L);
```

```
(* version which overwrites earlier matches *)

fun abstractmerge nil L = L |

   abstractmerge ((s,t)::M) L =

      (s,t)::(abstractmerge M (abstractdrop s L));

(* intersection of arguments in two sort dec lists -- no compatibility check *)

fun intersection nil L = nil |

intersection ((n,t,u)::L) M = if find t M <> nil then ((n,t,u)::(intersection L M))
else intersection L M;

fun union nil L = L |

union ((n,t,u)::L) M = if find t M = nil then ((n,t,u)::(union L M))
else union L M;

fun unionoflist nil = nil |

unionoflist (L::M) = union L (unionoflist M);

(* find the age of the declaration of a term in a move.
This is used to check the requirement that (explicit) parameters
appear in a function definition or declaration in the order in which they are declared.
The age of a defined object or function declaration is always zero. *)

fun age s nil = nil |

age s ((n,t,u)::L) = if s=t then [n] else age s L;

(* check a condition for all elements of a list *)

fun testall test nil = true |

testall test (s::L) = test s andalso testall test L;

fun inlist x nil = false |

inlist x (s::L) = if x=s then true else inlist x L;

fun allinlist nil L = true |

allinlist (s::M) L = inlist s L andalso allinlist M L;
```

```
(* drop items from a declaration list *)

fun drop s nil = nil |

drop s ((i,a,t)::L) = if s=a then drop s L else ((i,a,t)::(drop s L));

fun droplist L nil = nil |

droplist nil L = L |

droplist (s::M) L = drop s(droplist M L);

(*
```

The block above contains operations on tuples, lists, and sets.

Special support is given to Lestrade environments (used in the representation of "moves" and also in the representation of function sorts and anonymous function notations) which are lists of triples (n, t, u) where n is the age of the tuple, t is a term (a variable whose declaration is being recorded, except possibly in the last triple in the list), which we will call the key, and u is the sort associated with the item t. We will refer to these structures as *declaration lists*.

The functions `pi13`, `pi23`, and `pi33` are projection functions for triples.

The `find` function finds the sort associated with a key in a declaration list. It actually returns a one-element list containing the sort if it finds one and nil otherwise.

The functions `findandreplace` and `findandpurge` are used to find and either replace or purge items in a declaration list: they are so far used only by the deferred definition feature.

The `abstractfind` function is a more general find function for lists of pairs in which the first item in each pair is the key (such as matching lists). We will refer to such lists as *abstract lists*. The `Abstractfind` function does the same thing, and moves searched items to the front of the list (so it acts on a reference to the list rather than the list itself).

The `abstractdrop` function drops all items with a given key from an abstract list.

The function `abstractdrop2` is similar: it is used for lists in which the keys are lists and keys extending a given key must be removed when the item at that key is removed or changed. The lists of saved moves and rewrites associated with moves have this characteristic.

The `abstractmerge` function merges two abstract lists. When it adds another item with the same key, it overwrites rather than recognizing a conflict.

The `intersection` command returns all items in a first declaration list for which there is an item with the same key in a second declaration list, without a sort compatibility check.

The `union` command returns a list containing all items in a first list which have keys not found in a second list, followed by all items in the second list, without any sort compatibility checks.

The `unionlist` command takes unions in the same way as `union` but of a list of lists.

The `age` function acts on declaration lists as `find` above, but returning the age associated with a key rather than the sort.

The `testall` function reports whether a condition represented by a function `test` (taking list items to booleans) holds of all items in a list.

```
*)



(* the internal representation of basic sorts (types) and objects *)

(* this type declaration seems to be marvellous,
as it seems to capture all sorts of things that we talk about, all the way from
mathematical objects to moves *)

datatype EntType (* basic sorts of objects *) =

obj (* mathematical objects *) |

prop (* propositions *) |

TYPE (* types of mathematical object *) |

that of Entity (* sort of proofs of a given proposition *) |

IN of Entity | (* the sort associated with a particular object of sort TYPE *)

error

and AbstType =

World of (int*Argument*Type) list (* this is the metasort of worlds (moves), and
                      also of
                      sorts assigned to functions *)

(* the integer is ``age", for ensuring correct order in
argument lists. Age is 0 for defined notions *)

and Type (* general sorts of objects and functions *) =
```

```
(* it is worth noting here that the current partition of entities
into objects and functions corresponds to older terminology "entities"
and "abstractions", which is reflected in class and constructor names
in the source.  I am correcting comments but not the code. *)

EType of EntType (* sort of an object (entity) *)|

AType of AbstType (* sort of a function (abstraction) *)

and Entity (* first order objects (entities):
typed and untyped objects, propositions, types, and proofs *) =

Unknown (* postulated, unknown--this is the ---
appearing in the sort in a function declaration *) |

Deferred (* this is the marker for a deferred definition *) |

Error |

Ent of string*int (* the numeral indicates namespace when nonzero,
used for renaming bound variables in dependent sorts and lambda terms *) |

App of string*int*(Argument list)  (* the numeral again indicates namespace *)

and Argument (* first and second order entities,
objects and functions, as they appear in argument lists *) =

EntArg of Entity (* objects *) |

AbstArg of string*int (* functions *) | (* numeral is again namespace *)

Lambda of AbstType;  (* lambda terms appearing in sorts *)

fun truncateto2 s nil = nil |

truncateto2 s ((n,t,u)::L) = if s=t then ((n,t,u)::L) else truncateto2 s L;

fun truncateto1 s L = rev(truncateto2 s (rev L));

fun truncateto s nil = nil |

truncateto s ((World L)::M) = if find s L = nil then truncateto s M else (World (truncateto1

(* the user never has to enter a lambda-term as a function argument [though she can now], or
but the system generates lambda-terms when an identifier passes out of scope,
and also in the implicit arguments mechanism *)
```

```
(*
```

The block above is of central importance. It contains the master declaration of structures handled by the Type Inspector, including representations of objects, functions, sorts, argument lists, and moves.

The nomenclature reflects the old terminology "entities" for what are now called "objects" and "abstractions" for what we later called "functions" and now call "constructions".

The ML type `EntType` contains representations of the basic object sorts.

1. `obj` represents the sort of untyped mathematical objects.
2. `prop` represents the sort of propositions.
3. `TYPE` represents the sort of type labels (Lestrade sort `type`).
4. `that(x)` when `x` is of type `Entity` represents the sort of proofs of the proposition represented by `x` (Lestrade sort `that x`; the type checker of Lestrade expects `x` to be of Lestrade sort `prop`, but ML does not enforce this).
5. `IN(x)` when `x` is of type `Entity` represents the sort of typed mathematical objects of Lestrade sort `in x` (the type checker of Lestrade expects `x` to be of Lestrade sort `type`, but ML does not enforce this).
6. `error` represents an unsuccessful attempt to represent a sort (an error).

The ML type `AbstType` is related to the type of *declaration lists* mentioned above: more detail is given here. An object of this type is the result of applying the constructor `World` to a list of triples in which each triple has the first projection of type `int`, the second projection of type `Argument`, and the third projection of type `Type`. "World" is old terminology for what we now call "moves", and this is the type of moves.

The ML type `Type` represents general Lestrade sorts. An element of type `Type` is obtained by applying the constructor `EType` to an element of type `EntType` (obtaining a Lestrade object sort) or applying the constructor `AType` to an element of type `AbstType` (obtaining a Lestrade function/construction sort).

The ML type `Entity` contains representations of Lestrade objects.

1. `Ent(s,n)` where `s` is a string and `n` is an integer represents an atomic object constant. The integer `n` signals namespace: if it is 0, this should be a declared constant or variable object, while if it is nonzero this should represent a namespace in which it is bound (a function sort term or an anonymous function term has all its bound variables tagged with a unique identifier).

2. `App(s,n,L)` where `s` is a string and `n` is an integer and `L` is a list of arguments (of ML type `Argument`) represents application of a function with the name `s` to the argument list `L`. The numeral `n` is namespace as above: if it is 0 then `s` should be a currently declared function, while if it is nonzero this indicates that `s` is bound in a function sort or anonymous function notation, `n` identifying the correct namespace.

3. `Unknown` is a pseudo-object. `EntArg Unknown` appears as the key in the final item in the list `L` appearing in a function sort `AType(World L)` recorded as the sort of the output of the function, if the function is introduced by the `postulate` command. This is one of the two exceptions to the usual condition that key in an item in the list `L` in an element `World L` is a suitably packaged identifier `EntArg(Ent(s,n))` or `AbstArg(s,n)`. There are other internal uses of it as a sort of dummy object in ML function definitions. If a function is introduced by the `define` command, the key of the last element of the list `L` appearing in its recorded sort `AType(World L)` is the body of the definition (the list is the same as would appear in the representation of the function as a λ-term described below); its actual sort is obtained by replacing the key with `EntArg Unknown`, an operation carried out by functions we introduce shortly, but the extra declaration information about the function is useful and this is a compact way to report it.

   `Deferred` is another pseudo-object, standing in for a deferred definition body.

4. `Error` is an error object.

The ML type `Argument` is used to represent individual items in argument lists.

1. `EntArg(e)`, where `e` is of type `Entity`, yields `e` as an entity argument.

2. `AbstArg(s,n)`, where `s` is a string and `n` is an integer, yields the function named by `s` as an argument, with `n` serving to indicate namespace (if `n` is 0, `s` is currently declared, otherwise `s` is bound in a function sort term or an anonymous function term).

3. `Lambda(World L)` represents an anonymous function term as an argument. In this case the last term of the declaration list `L` will have the body of the represented λ term as its key, this being the other exception to keys in declaration lists being suitably packaged identifiers.

The function `truncateto` is a utility for the deferred definition mechanism which may have further uses: it recreates the exact context after an identifier was declared.

```
*)
```

```
(* utilities for manipulating the master ML types *)

fun isapp (App(m,s,t)) = true |
isapp x = false;

(* get function sort (or move) from a general sort *)

fun getabstype (AType x) = x|

getabstype x = (saypause "getabstype error";World nil);

(* identify entity arguments *)

fun isentarg (EntArg x) = true |

isentarg x = false;

(* utility identifies an object sort as opposed to a function sort *)

fun isenttype (EType x) = true |
isenttype (AType x) = false;

(* utility for adding an item to a move at the beginning *)

fun addworld2 x ((World M)) = (World (x::M));

(* utilities for taking apart an application term *)

fun appof (App(t,n,L)) = t |

appof x = "";

fun argsof (App(t,n,L)) = L |

argsof x = nil;

(* body of a lambda term -- I do wonder if this is a duplicate *)

fun lambdabody [(n,EntArg a,t)] = a |

lambdabody (x::L) = lambdabody L |

lambdabody x = Error;

fun lambdainputs nil = nil |
```

```
lambdainputs L = rev(tl(rev L));

(*
```

`lambdabody` pulls out the body of a lambda term `Lambda(World L)` from the list `L`; it is the second component of the last triple in `L`.

`lambdainputs` applied to `L` gives the types of the inputs to the anonymous function represented by `Lambda (World L)`, by stripping off the last item which handles the body of the term and the output type.

```
*)

(* get object sort from a general sort *)

fun getenttype (EType x) = x|

getenttype x = (saypause "getenttype error";error)

(* extract name from an atomic term *)

fun nameof (AbstArg(s,n)) = s |

nameof (EntArg(Ent(s,n))) = s |

nameof x = "";

(* convert a function sort to a declaration list *)

fun deworld2 (AType(World L)) = L |

deworld2 x = nil;

(* identifies an object argument which is not an atomic term *)

fun notvararg (EntArg(Ent(s,0))) = false |

notvararg (EntArg x) = true |

notvararg x = false;

(* utility adds an item to a move at the end *)

fun addtoworld0 (World L) x = World(L @ [x]);
```

```
fun deabst(AbstArg(s,0)) = s |

deabst x = "?!?!";

fun deworld (World L) = L;

(* a utility -- coerce an argument to an object. *)

fun deent (EntArg x) = x|
deent x = Error;

(*
```

Above find utilities for manipulating the master types in various ways. These were scattered through the source originally: I gathered them in one place.

```
*)

(* clean definition information out of sorts *)

fun cleantypelist L = rev((pi13 (hd(rev L)),EntArg Unknown,
      pi33(hd(rev L)))::(tl(rev L)));

fun Cleantype0(World L) =
      World((* cleantypelist *) (map (fn (x1,x2,x3) => (x1,x2,Cleantype1 x3)) (cleantypelist

and Cleantype1 (AType(World L)) = AType(Cleantype0(World L)) |
Cleantype1 x = x;

fun Cleantype2(World L) = World(map (fn(x1,x2,x3) => (x1,x2,Cleantype1 x3)) L);

fun Cleantype3(AType(World L)) = AType(Cleantype2(World L));

fun Cleantype4 (Lambda(World L)) = Lambda(Cleantype2(World L));

(*
```

The `Cleantype` functions systematically replace keys of final elements of lists `L` in function sorts `AType(World L)` with `EntArg Unknown`, clearing information about the bodies of definitions from the types given for defined functions.

`cleantypelist` carries out the actual replacement of the key on the last triple in a list with `EntArg Unknown`.

`Cleantype0` performs the action of `cleantypelist` on the list in an element `World L`; it also cleans all types of items in the list with `Cleantype1`, which executes the same action on the move component of a function sort `AType(World L)` but leaves object sorts alone.

`Cleantype2` applied to `World L` applies `Cleantype1` to all types appearing in `L` but does not strip definition body information from the last item in `L`. `Cleantype3` performs the action of `Cleantype2` on the move component of an element `AType(World L)`. This is a cleanup function which can be applied to the saved sort of a defined function without destroying the definition information. `Cleantype4` acts similarly on `Lambda(World L)`.

```
*)

(* change all namespace indices to their additive inverses;
used for rewriting patterns *)

fun Negindex1 obj = obj |(* mathematical objects *)

Negindex1 prop = prop | (* propositions *)

Negindex1 TYPE = TYPE (* types *) |

Negindex1 (that E) = that (Negindex4 E)
     (* sort of proofs of a proposition *) |

Negindex1 (IN E) = IN (Negindex4 E) |

Negindex1 error = error

and Negindex2 (World L) =
     Cleantype2(World(map (fn (i,A,T) => (i,Negindex5 A, Negindex3 T)) L))

and Negindex3 (EType E) = EType (Negindex1 E) (* sort of an object *)|

Negindex3 (AType A) = Cleantype3(AType (Negindex2 A)) (* sort of a function *)

and Negindex4 Unknown = Unknown (* postulated, unknown--this is the ---
appearing in the sort in a function declaration *) |

Negindex4 Deferred = Deferred |

Negindex4 Error = Error |

Negindex4 (Ent(s,n)) = (Ent(s,~n)) (* the numeral indicates namespace,
used for renaming bound variables in dependent sorts and lambda terms *) |
```

```
Negindex4 (App(s,n,L)) = (App(s,~n, map Negindex5 L))
      (* the numeral again indicates namespace *)

and Negindex5 (EntArg E) = EntArg(Negindex4 E) |

Negindex5 (AbstArg(s,n)) = AbstArg(s,~n)| (* numeral is again namespace *)

Negindex5 (Lambda A) = Cleantype4(Lambda (Negindex2 A));
    (* lambda terms appearing in sorts *)

(*
```

Systematically replace namespace indices with their additive inverses. This is a technical trick for rewriting patterns. (NOTE: comment later on how it is used). I'm wondering if the applications of `Cleantype` are accidental (copied from another function).

Adding facilities to help manage the global namespace index: `Maxfreshindex` keeps track of the largest value that `Freshindex` has ever had.

```
*)

(* namespace reindexing utilities *)

(* find a number in a list of non negative integers:  used in namespace
reindexing for display in showdec *)

(* These functions are now being applied to
sorts entered in declarations, not only to their displays. *)

fun Find n nil = ~1 |

Find n ((p,q)::L) = if n=p then q else Find n L;

val Freshindex = ref 0;

val Maxfreshindex = ref 0;

val Indexlist = ref [(~1,~1)];

val _ = Indexlist := nil;

fun Renumber n = if n=0 then 0 else let val N = Find n (!Indexlist) in
```

```
if N= ~1

then (Indexlist:=(n,!Freshindex)::(!Indexlist);Freshindex:=1+(!Freshindex);
if (!Freshindex)>(!Maxfreshindex) then Maxfreshindex := (!Freshindex) else ();
     (!Freshindex)-1)

else N end

fun Reset() = (Freshindex := 1;Indexlist := nil);

(*
```

The block above has a very specific purpose: it renames bound variables in Lestrade sort notations and anonymous function notations. Bound variables have numerical tags which are unique to their binding context: this function finds these tags and associates them with new successive numerical values. The numerical tags occur more than once, so the renumbering feature has to keep a table of correspondences between old numerical tags and new ones. The `Reset` function reinitializes this feature, setting the counter for the next new numerical tag to 0 and the list of renamings to nil. Only `Renumber` and `Reset` are called outside this block.

```
*)

(* namespace reindexing function:  this is a utility which avoids runaway indices
on new bound variables in declaration displays *)

(* This function is now applied
to the actual recorded sorts; in earlier versions it was only applied to displays *)

(* this required care that renaming of bound variables (the
function renamespace) be applied before *any* substitution into
a function sort or lambda term *)

(* these functions now also clean inappropriate definition information out of sorts;
apparently such information sneaks into arguments
somewhere in the implicit argument inference
mechanism *)

fun Reindex1 obj = obj |(* mathematical objects *)

Reindex1 prop = prop | (* propositions *)

Reindex1 TYPE = TYPE (* types *) |
```

```
Reindex1 (that E) = that (Reindex4 E)  (* sort of proofs of a proposition *) |

Reindex1 (IN E) = IN (Reindex4 E) |

Reindex1 error = error

and Reindex2 (World L) =
     Cleantype2(World(map (fn (i,A,T) => (i,Reindex5 A, Reindex3 T)) L))

and Reindex3 (EType E) = EType (Reindex1 E) (* sort of an object *)|

Reindex3 (AType A) = Cleantype3(AType (Reindex2 A)) (* sort of a function *)

and Reindex4 Unknown = Unknown (* postulated, unknown--this is the ---
 appearing in the sort in a function declaration *) |

Reindex4 Deferred = Deferred |

Reindex4 Error = Error |

Reindex4 (Ent(s,n)) = (Ent(s,Renumber n))
(* the numeral indicates namespace,
used for renaming bound variables in dependent sorts and lambda terms *) |

Reindex4 (App(s,n,L)) = (App(s,Renumber n, map Reindex5 L))
(* the numeral again indicates namespace *)

and Reindex5 (EntArg E) = EntArg(Reindex4 E) |

Reindex5 (AbstArg(s,n)) = AbstArg(s,Renumber n)| (* numeral is again namespace *)

Reindex5 (Lambda A) = Cleantype4(Lambda (Reindex2 A));
     (* lambda terms appearing in sorts *)

fun addtoworld (World L) x
= (Reset();World(L @ [(fn (i,A,T)=>(i,Reindex5 A,Reindex3 T)) x]));

(*
```

This is the function which actually does reindexing of sorts when they are saved after declarations, using the functions in the previous block. It also purges inappropriate information about definition bodies from types in some places; it appeared that such information started leaking into saved types during the development of the implicit arguments mechanism. The cleanup process here is

limited: declarations of defined functions still contain definition body information, but it is purged from the types of variables.

The utility `addtoworld` for adding an item to a move at the end appears here because it calls the reindexing functions. I note that it is actually never used (should I replace subsequent occurrences of `addtoworld0` with this function?)

```
*)

(* CONTEXT is the list of moves in use *)

val CONTEXT = ref  [World nil, World nil];

val SAVECONTEXT0 = ref((!CONTEXT));

(* the list of rewrite rules declared or justified *)

val REWRITES = ref [[("bogus",(Unknown,Unknown))],[("bogus",(Unknown,Unknown))]];

val _ = REWRITES := [nil,nil];

(* the list of names attached to moves in the current sequence *)

val WORLDNAMES = ref ["1","0"];

(* the list of moves which have been saved *)

val SAVEDWORLDS = ref [(!WORLDNAMES,hd(!CONTEXT))];

val _ = SAVEDWORLDS:=nil;

(* saved rewrite rules, really a component of the previous
     though implemented separately *)

val SAVEDREWRITES = ref [(!WORLDNAMES,hd(!REWRITES))];

val _ = SAVEDREWRITES:=nil;

(* serial number used for recording age of declarations *)

val SERIAL = ref 0

(* the index for generating fresh names in new namespaces,
 for bound variable renaming in dependent sorts and lambda-terms *)

val NAMESERIAL = ref 0;
```

```
(* full theories, saved when files are loaded using readfile or readfile2 *)

val SAVEDTHEORIES = ref [("bogus",(!CONTEXT,!REWRITES,!SERIAL,!NAMESERIAL,!Maxfreshindex,!W

(* fun ClearAll() = (clearallcaches();GREETED:=false;CONTEXT:=[World nil,World nil];
REWRITES:=[nil,nil];SERIAL:=0;NAMESERIAL:=0;Maxfreshindex:=0;WORLDNAMES:=["1","0"];SAVEDWORL

val _ = SAVEDTHEORIES := nil;

(*
```

Here we find references to structures in which declarations are saved.

`CONTEXT` points to the actual list of moves. The head of the list is the "next move"; the second item in the list is the "last move"; the list is always of length at least two.

`REWRITES` points to the list of lists of rewrite rules which is maintained in parallel.

`WORLDNAMES` is the list of names of moves in the current sequence.

`SAVEDWORLDS` is the list of saved moves (more a tree structure than a list). A saved move is specified not just by its name, but by the sequence of names of moves leading to it. Different moves in the tree are allowed to have the same string as a name; but different moves following the same previous move are not. No move with a name other than its default numeral name will have a preceding move other than move 0 which has its default numeral name; a move with its default numeral name will preceded by a move with its default numeral name (unless it is move 0).

`SAVEDREWRITES` is the list of saved rewrite rules saved in parallel with the saved moves.

`SAVEDTHEORIES` is a list of saved theories: the move 0 declarations and some indices are saved.

`SAVECONTEXT0` is a reference used by the deferred definition feature to back up the entire context.

```
*)

(* determine the number of the next move and attach
its name if it has a non-default name (not simply its numeral index)  *)

fun worldname0 n nil = "bogus" |

worldname0 0 L = if hd L = makestring(length L-1) then "" else ":"^(hd L) |

worldname0 n L = worldname0 (n-1) (tl L);
```

```
fun worldname n = worldname0 n (!WORLDNAMES);

(* does a move (other than move 0) have a trivial name
     (i.e., simply its numerical index) *)

fun defaultworld L = (length L >=2 andalso hd L = makestring((length L) -1));

(* recover lists of moves which can be opened using
open or clearcurrent commands *)

fun savedfor M nil = "" |

   savedfor M ((s,t)::L) =

   if tl s = M then (hd s)^"\n"^(savedfor M L)
   else (savedfor M L);

(* USER COMMAND *)

fun savedforopen() = savedfor (!WORLDNAMES) (!SAVEDWORLDS);

(* USER COMMAND *)

fun savedforclearcurrent() = savedfor (tl(!WORLDNAMES)) (!SAVEDWORLDS);

(*
```

Utilities for management of saved worlds.

The function `worldname` is used to extract the name of a move (which is not displayed if it has its default numerical value).

`defaultworld` identifies lists of moves in which the next move (at the head of the list) has the default name.

The `savedforopen` and `saveforclearcurrent` commands are user commands which display the names of the saved moves you can open with `open` or `clearcurrent`.

```
*)

(* pretty printing *)

val EXTRAINDENTS = ref 0;
    (* keep track of extra indentation needed in display of function sorts *)
```

```
(* break after the first comma, colon, or space after n characters *)

(* also, indent displays more deeply as one moves to deeper moves *)

(* also, indent more deeply at breaks inside function sorts or lambda terms
and always break after a closing bracket
 (possibly picking up some following punctuation) *)

fun tolinebreak n nil = nil |

tolinebreak n (#"]":: #")":: #",":: L) =
       (EXTRAINDENTS:=(!EXTRAINDENTS)-1;[#"]", #")", #","]) |

tolinebreak n (#"]":: #")":: #")":: L) =
      (EXTRAINDENTS:=(!EXTRAINDENTS)-1;[#"]", #")", #")"]) |

tolinebreak n (#"]":: #")":: L) =
 (EXTRAINDENTS:=(!EXTRAINDENTS)-1;[#"]", #")"]) |

tolinebreak n (#"\n":: #"\n":: L) = [#"\n", #"\n"]@(tolinebreak 0 L) |

tolinebreak 0 (c::L) = (
if c = #"[" then EXTRAINDENTS := 1+(!EXTRAINDENTS) else ();
if c = #"]" then EXTRAINDENTS := (!EXTRAINDENTS)-1 else ();
if c = #"," orelse c = #":" orelse c= #"]" orelse (c = #" "
andalso (L = nil orelse hd L <> #" ")) then [c] else c::(tolinebreak 0 L)) |

tolinebreak n (#" " :: #"\\" :: #"\n" :: L) = tolinebreak (n-3) L |

tolinebreak n (#"\\" :: #"\n" :: L) = tolinebreak (n-2) L |

tolinebreak n (#"\\"::L) = tolinebreak (n-1) L |

tolinebreak n (#" ":: #" ":: L) = #" ":: #" ":: tolinebreak (n-2) L |



tolinebreak n (x:: #" ":: #" ":: L) =

(
if x = #"[" then EXTRAINDENTS := 1+(!EXTRAINDENTS) else ();
if x = #"]" then EXTRAINDENTS := (!EXTRAINDENTS)-1 else ();
tolinebreak n (x:: #" " :: L)) |

(* tolinebreak n (x :: #" "::nil) = [x] | *)
```

```
tolinebreak n (#"\n" :: #" " :: L) = tolinebreak n (#"\n" :: L) |

tolinebreak n (#"\n" :: L) = tolinebreak n (#" "::L)  |

(* tolinebreak n (#"\n"::nil) = #"\n"::nil | *)

tolinebreak n (c::L) = (
if c = #"[" then EXTRAINDENTS := 1+(!EXTRAINDENTS) else ();
if c = #"]" then EXTRAINDENTS := (!EXTRAINDENTS)-1 else ();
if c= #"]" then [#"]"] else
c::(tolinebreak (n-1) L));

fun restlinebreak n nil = nil |

restlinebreak n (#"]":: #")":: #",":: L) = L |

restlinebreak n (#"]":: #")":: #")":: L) = L |

restlinebreak n (#"]":: #")":: L) = L |

restlinebreak n (#"\n":: #"\n":: L) = restlinebreak 0 L |

restlinebreak 0 (c::L) = if c = #"," orelse c= #":" orelse c = #"]"
orelse (c = #" " andalso (L = nil orelse hd L <> #" "))  then L
else (restlinebreak 0 L) |

restlinebreak n (#" " :: #"\\" :: #"\n" :: L) = restlinebreak (n-3) L |

restlinebreak n (#"\\" :: #"\n" :: L) = restlinebreak (n-2) L |


restlinebreak n (#"\\"::L) = restlinebreak (n-1) L |

restlinebreak n (#" ":: #" ":: L) = restlinebreak (n-2) L |

restlinebreak n (x:: #" ":: #" ":: L) = restlinebreak n (x:: #" " :: L) |

restlinebreak n (#"\n" :: #" " :: L) = restlinebreak n (#"\n" :: L) |

(* restlinebreak n (x:: #"\n"::nil) = nil | *)

(* restlinebreak n (#"\n"::nil) = nil | *)

restlinebreak n (#"\n" :: L) = restlinebreak n (#" " :: L) |
```

```
restlinebreak n (c::L) = if c = #"]" then L else (restlinebreak (n-1) L);

(*
```

These functions determine the initial segment of a string being read to the next line break (the numerical parameter is an estimate of how far it should be to the next line break) and the rest of the string after the next line break and keep track of how much indentation is expected after the line break (indentation is determined by the length of `CONTEXT` (the move depth) and the depth of variable binding (the number of function sort terms and anonymous function terms (λ-terms) in which the position of the line break lies).

```
*)

val MARGIN = ref 40;

val MARGINTEMP = ref (!MARGIN);

(* USER COMMAND -- modified in the interface *)

fun setmargin n = MARGIN:= n;

fun setmargintemp n = MARGINTEMP:=n;

val spaceblock = "   ";

fun indents n = if n<= 0 then "" else (indents (n-1))^spaceblock;

val INDENTWIDTH = ref 3

fun INDENTS() = let val I = indents (((length(!CONTEXT)-2))+(!EXTRAINDENTS)) in
if length(explode I)+(!INDENTWIDTH) > (!MARGINTEMP) then (setmargintemp(length(explode I)+2*


fun despace0 (#" "::L) = despace0 (L) |

despace0 (#"\\"::L) = despace0 L |

(* despace0 (#"\n" :: #"\n" ::L) = despace0 (#"\n" :: L) |

despace0 (#"\n"::L) = despace0 L | *)

despace0 L = L
```

```
fun despace1 (""::L) = despace1 L |

despace1 L = L

and despace s = implode(rev(despace0(rev(despace0(explode s)))));

fun initial s = (INDENTS())^(despace(implode(tolinebreak(!MARGINTEMP)(explode (s)))));

fun final s = implode(restlinebreak(!MARGINTEMP)(explode (s)));


fun prettyprint s = let val I = INDENTS() in
(initial ((I)^(despace s)))^(if final (I^(despace s)) = "" then ""
 else "\n   "^(I)^(prettyprint(final ((I)^(despace s))))) end;

(* this produces the same output as prettyprint with each output
formatted for the log file as a temporary comment *)

fun prettyprint2 s = let val I = INDENTS() in (initial
((I^(despace s))))^(if final ((I)^(despace s))  = "" then "" else
"\n>>   "^(prettyprint2(final ((I^(despace s)))))) end;

fun prettyprint3 s = if hd (explode s) = #"\n" then
prettyprint2 s else
"\n>> "^(prettyprint2 s);

fun prettyprint2a s = let val I = INDENTS() in (initial
((I^(despace s))))^(if final ((I)^(despace s))  = "" then "" else
if despace (final ((I^(despace s)))) = "" then "" else
" \\\n   "^(prettyprint2a(final ((I^(despace s)))))) end;

fun prettyprint2b s = let val I = INDENTS() in I^(despace (prettyprint2a s)) end;

fun prettyprint3a s = if hd (explode s) = #"\n" then
prettyprint2b s else
"\n"^(prettyprint2b s);
(*
```

This block of code completes the pretty printing functions.

`MARGIN` is the margin, which the user can set using the `setmargin` command. The margin sometimes has to be temporarily adjusted, which is the use of `MARGINTEMP`.

The `INDENTS()` function returns the indent needed after a line break based on the current move depth and variable binding term depth. `indents` is an obvious internal part of that command.

The despace functions strip initial spaces (and null strings) from lists of characters.

initial produces the intial segment of a line to the first line break; final produces the rest of it.

prettyprint produces pretty printed text; prettyprint2 and prettyprint3 cooperate to produce pretty printed text packaged as temporary comments in log files. Lettered versions are designed to make it possible to pretty-print command lines.

```
*)

(* post a pretty printed piece of Lestrade notation to standard output *)

fun say0 s=
(EXTRAINDENTS:=0;
TextIO.output(TextIO.stdOut,
(prettyprint s)^"\n");Flush());

(* post a pretty printed piece of Lestrade notation to standard output and the log file *)

fun say1 s= (say0 s;EXTRAINDENTS:=0;
TextIO.output(!LOGFILE,
(prettyprint3 s)^"\n");Flush());

fun say2 s= (say0 s;EXTRAINDENTS:=0;
TextIO.output(!LOGFILE,
(prettyprint3a s)^"\n");Flush());

(*
```

say0 posts a pretty printed piece of Lestrade notation to standard output, using prettyprint; say1 posts pretty-printed notation to standard output and also to the log file, using prettyprint3. say2 is intended to support line breaks and indentation in echoed command lines.

```
*)

(* find the Argument reference of the string s in
the list of moves L (this returns the sort of the
object or function named by s if there is one, which does
contain enough information to tell whether the object
is declared and whether it is an object or a function *)
```

```
fun Find s (World L) = find s L;

(* similarly this returns the age of any declaration in any move of s --
the main use of this is to identify defined functions, which have age 0 *)

fun Age s (World L) = age s L;

(* stringdef returns the singleton list of
the sort of an identifier using a given move list argument,
paired with the numerical index of the move it is found in,
 or nil for error *)

fun pi1(x,y)=x;


fun pi2(x,y)=y;

fun stringdef s nil = nil |

stringdef s L =

let val A = Find (EntArg(Ent (s,0))) (hd L) in

if A <> nil then

if stringdef s (tl L) <> nil then (saypause ("Name collision error: "^s);nil) else

[(hd A,0)] else

let val B = Find (AbstArg (s,0)) (hd L) in

if B <> nil then

if stringdef s (tl L) <> nil then (saypause ("Name collision error: "^s);nil) else

[(hd B,0)] else

let val C = stringdef s (tl L) in

if C = nil then nil else [(pi1(hd C),pi2(hd C)+1)] end

end

end;

(* this returns the singleton list of the sort of an identifier in the Lestrade context,
```

```
paired with the numerical index of the move it is found in,
or nil for error *)

fun stringtype s =  stringdef s (!CONTEXT);

(*
```

These are functions which determine the semantics of strings in Lestrade contexts. `Find` (a second use of this function name, which cannot conflict with the use in the internals of the renumbering mechanism) acts as `find` on the output of a `World` constructor. Similarly, `Age` acts as `age` on the output of a `World` constructor; these allow finding the sorts and ages associated with keys in moves.

`pi1` and `pi2` are just projections of pairs.

The function `stringdef` applied to a context list returns the sort associated with the string in that context paired with the numerical index of the move in which it occurs relative to the next move (not the absolute index relative to move 0; the relative index of the next move is 0, of the last move is 1 and so forth). Errors are reported if the same string appears as a key in more than one move in the context list. The function `stringtype` returns the same information about a string in the `CONTEXT` context list.

```
*)

(* build function to extract explicit argument list *)

val DISPLAYIMPLICIT = ref false

(* USER COMMAND *)

fun showimplicit() = DISPLAYIMPLICIT := true

(* USER COMMAND *)

fun hideimplicit() = DISPLAYIMPLICIT := false

fun purgeimplicit (World L) nil = nil |

purgeimplicit (World ((n,EntArg(Ent(s,m)),t)::L)) (u::M) =

if s<> "" andalso hd(explode s) = #"." then purgeimplicit (World L) M
else (u::(purgeimplicit (World L) M)) |

purgeimplicit (World ((n,AbstArg(s,m),t)::L)) (u::M) =
```

```
if s<> "" andalso hd(explode s) = #"." then purgeimplicit (World L) M
else (u::(purgeimplicit (World L) M)) |

purgeimplicit (World(x::L)) (u::M) = u::(purgeimplicit (World L) M);

fun explicitlist s n L = if (!DISPLAYIMPLICIT)
orelse n<>0 orelse stringtype s = nil then L
else purgeimplicit (getabstype((pi1(hd(stringtype s))))) L;

(*
```

These are functions which must be rather mysterious at this point related to the display or non-display of implicit arguments. Arguments with an initial dot in sorts of declared functions are implicit and should not be displayed; the `purgeimplicit` function is used to eliminate arguments which should not be displayed in the display functions, by correlating the argument list of an instance of the function with the argument list in the declaration.

The user command `showimplicit` forces display of implicit arguments; `hideimplicit` restores the normal behavior.

`explicitlist` actually generates explicit argument lists.

```
*)

(* display functions of Lestrade *)

val TYPESONLY = ref false;

fun typesonly() = TYPESONLY:=true;

fun showdefs() = TYPESONLY:=false;
(*
```

The `typesonly` command turns off display of definition bodies (useful for reducing size of displays when definition expansion gets out of hand); `showdefs` reverses this effect.

```
*)

(* display1 displays object sorts *)

fun display1 obj = "obj" |

display1 prop = "prop" |
```

```
display1 TYPE = "type" |

display1 (that P) = "that "^(display2 P) |

display1 (IN P) = "in "^(display2 P) |

display1 error = "error"

(* display2 displays object terms.  Note that ---
is reserved for the pseudo-object Unknown used
 as output for primitive functions, and ??? for Error. *)

and display2 (Ent(s,n)) = if n=0 then s else s^"_"^(makestring n)|

display2(App(s,n,nil)) = display2(Ent(s,n)) |

display2 (App(s,n,LL)) =

let val L = explicitlist s n LL in

if length L = 2 andalso isentarg (hd L) then
"("^(display4 (hd L))^" "^s^(if n=0 then ""
 else "_"^(makestring n))^" "^(display4(hd(tl L)))^")"

else s^(if n=0 then "" else "_"^(makestring n))^"("^(display3 L) end |

display2 Unknown = "---" |

display2 Deferred = "+++" |

display2 Error = "???"

(* display3 displays argument lists *)

and display3 [a] = (display4 a)^")" |

display3 (a::L) = (display4 a)^","^(display3 L) |

display3 nil = "*?*?)"

and display4 (EntArg x) = display2 x |

display4 (AbstArg(s,n)) = display2(Ent(s,n)) |

display4 (Lambda x) = "["^(display5a x)
```

```
(* display5 displays dependent sorts and anonymous function terms (lambda terms).
  This is the part of the output language about which the parser knows nothing.  display5a,
differs in that the definition body cannot be suppressed in the display. *)

and display5 (World [(n,a,t)]) = "("^(if (!TYPESONLY) andalso a <> EntArg Unknown then " ...

display5 (World((n,a,t)::L)) = "("^(display4 a)^":"^(display6 t)^
(if length L = 1 then ") => " else "),")^(display5 (World L)) |

display5 (World nil) = "(?*?*?*?)"

and display5a (World [(n,a,t)]) = "("^((display4 a))^":"^(display6 t)^")]" |

display5a (World((n,a,t)::L)) = "("^(display4 a)^":"^(display6 t)^
(if length L = 1 then ") => " else "),")^(display5a (World L)) |

display5a (World nil) = "(?*?*?*?)"

(* display6 displays general sorts *)

and display6 (EType x) = display1 x |

display6 (AType(World x)) =  "["^(display5 (World x));

(* display a move.  This is the same type displayed
by display5, with different intent *)

fun displayworld (World nil) = "" |

displayworld (World((n,a,t)::L)) = (displayworld (World L))
^(INDENTS())^(display4 a)^":"^(display6 t)^"\n\n";

(*
```

These are the display functions that go with the master type declaration. The functions other than `displayworld` are engaged in display of terms of various kinds; `displayworld` displays a move (as a list of declarations). It is interesting to note that `display5` displays exactly the same ML type but in the different role of the internal part of a function sort or a λ-term.

```
*)

(* display the indexed list of moves.
displacement corrects move indices in the showrecent command below *)
```

```
fun displayworlds displacement nil = "\n\n" |
displayworlds displacement L = "\n\nMove "^(makestring(displacement+(length L)-1))
^(worldname (length (!WORLDNAMES)-(displacement+(length L))) )
^":\n\n"^(displayworld (hd L))^(displayworlds displacement (tl L));

(* display a rewrite list *)

fun displayarewritelist nil = "\n\n" |

displayarewritelist ((s,(t,u))::L) = (s^":  "
^(display2 t)^" := "^(display2 u)^"\n")^(displayarewritelist L);

fun displayrewrites0 nil = "\n\n" |

displayrewrites0 (L::M) = (displayarewritelist L)^(displayrewrites0 M);

(* USER COMMAND *)

fun displayrewrites() = say (displayrewrites0 (tl(!REWRITES)));

(* display all moves -- this shows all declarations in detail *)

(* USER COMMAND *)

fun showall () = say0 (displayworlds 0 (!CONTEXT));

(* display the next move
 and the last move (confusingly also called current move)
-- these are the moves in which you can actually make declarations,
 and the showall display is likely to be huge.
So is the showrecent display if move 0 is displayed! *)

(* USER COMMAND *)

fun showrecent() = say0
(displayworlds (length(!CONTEXT)-2) [hd(!CONTEXT),hd(tl(!CONTEXT))]);

(*
```

These are commands devoted to the display of declarations.

`displayworlds` is the general function for display of a context list (a list of moves). Some hacking is needed (and consultation of `worldname`) to post the correct names of moves. The user command `showall` shows all declarations; the user command `showrecent` shows the last move and the next move. The

displayrewrites command shows the currently active rewrite rules.

```
*)

(* display the declaration information of an identifier on standard output.  Send
this to the log as well. Bound variable indices are rationalized.  *)

(* USER COMMAND *)

val COMPACTDISPLAY = ref true

fun compactdisplay() = COMPACTDISPLAY := not(!COMPACTDISPLAY);

val SUPERCOMPACTDISPLAY = ref false

fun supercompactdisplay() = SUPERCOMPACTDISPLAY := not(!SUPERCOMPACTDISPLAY);

fun showdec s = let val S = stringtype s in

if S = nil then saypause (s^" is not declared")

else if (!SUPERCOMPACTDISPLAY) then ()

else if (!COMPACTDISPLAY) andalso length(!CONTEXT)-1-(pi2 (hd S)) > 0

then (Reset(); say1(s^":  "^(display6(Cleantype1(Reindex3(pi1 (hd S)))))^
" {move "^(makestring(length(!CONTEXT)-1-(pi2 (hd S))))
^(worldname(pi2(hd S)))^"}\n\n"))


else (Reset(); say1(s^":  "^(display6(Reindex3(pi1 (hd S))))^
" {move "^(makestring(length(!CONTEXT)-1-(pi2 (hd S))))
^(worldname(pi2(hd S)))^"}\n\n"))end;

(* utility for converting a move to a list of sort declarations *)

fun deworld (World L) = L;

(* showdecs will display declarations one by one.  Useful if move 0
is being displayed.  It goes through the next move in order, then the
last move in reverse order *)

fun showdecs0 nil = () |

showdecs0 ((i,EntArg(Ent(s,0)),t)::L) =
```

```
(showdec s;saynoreturn "Hit return to continue or q to break out:";
Flush(); TextIO.output(TextIO.stdOut,"\n");
if TextIO.input(TextIO.stdIn)="q\n" then ()
 else showdecs0 L)|

showdecs0 ((i,AbstArg(s,0),t)::L) =
(showdec s;saynoreturn "Hit return to continue or q to break out:";
 Flush();TextIO.output(TextIO.stdOut,"\n");
if TextIO.input(TextIO.stdIn)="q\n" then ()
 else showdecs0 L)|

showdecs0 L = ();

(* USER COMMAND *)

fun showdecs() = (say
 "Hit return after each declaration or q to quit";
say "Next move declarations"; showdecs0(deworld(hd(!CONTEXT)));
say "Present move declarations:"; showdecs0(rev(deworld(hd(tl(!CONTEXT))))));

(*
```

Here we have commands which display declarations of individual identifiers. It is worth noting here that the fact that `stringtype` returns relative index rather than absolute index of a move has to be taken into account, and also the fact that the argument of `worldname` is actually the relative index of the move.

`showdecs` is a command which displays all declarations in the next move and in the last move, one by one (the user hits return for the next one or `q` to break out). The order in which it displays them is opposite (this appears practical): in the next move it displays the most recent declarations first and in the last move the oldest first (or the reverse, I'll have to check).

It isn't clear to me that renumbering is needed any more in these commands, as I believe declarations are renumbered before being saved.

```
*)

(* age of an identifier in a given context packaged in a singleton list
 -- useful mainly for identifying defined functions *)

fun stringage s nil = nil |

stringage s L =

let val A = Age (EntArg(Ent (s,0))) (hd L) in
```

```
if A <> nil then

if stringage s (tl L) <> nil then

(saypause ("Name collision error :"^s); nil)

else

A else

let val B = Age (AbstArg (s,0)) (hd L) in

if B <> nil then

if stringage s (tl L) <> nil then

(saypause ("Name collision error :"^s); nil) else

B

else stringage s (tl L)

end

end;

(* as the previous command, in the Lestrade context *)

fun stringAge s = stringage s (!CONTEXT);

(*
```

These functions are analogous to `stringdef` and `stringtype`, except that they return the age of an identifier. A principal application of this is that the age of a defined identifier is 0. It is also required that arguments in an argument list in a declaration command appear in order of age.

```
*)

(* declaration checking for all types (in the ML sense)
 against a context given as an argument *)

fun (* check object sorts *) deccheck1 L obj = true
```

```
| deccheck1 L prop = true

| deccheck1 L TYPE = true

| deccheck1 L (that P) = deccheck2 L P

| deccheck1 L (IN P) = deccheck2 L P

| deccheck1 L error = false

and deccheck2 L Unknown = true |

deccheck2 L Deferred = true |

deccheck2 L Error = false |

(* check objects *) deccheck2 L
(Ent (s,n)) = n<>0 orelse let val S = stringdef s L in
(if S = nil
then saypause ("Did not find object "^s^" (deccheck2)")else ();
Flush();

S <> nil andalso isenttype(pi1(hd S))) end |

deccheck2 L (App(s,n,M)) =
      (n<>0 orelse
      let val S = stringdef s  L in

(if S = nil
then saypause ("Did not find function "^s^" (deccheck 2)") else();
Flush();

      S <> nil andalso not (isenttype(pi1(hd S)))) end)
      andalso testall (deccheck3 L) M



and (* check arguments *)
deccheck3 L (EntArg s) = deccheck2 L s

|  deccheck3 L (AbstArg (s,n))
= n<>0 orelse let val S = stringdef s  L
in  (if S = nil
then saypause ("Did not find function "^s^" (deccheck3)")else();
Flush();
```

```
      S <> nil andalso not (isenttype(pi1(hd S)))) end

| deccheck3 L (Lambda T) = deccheck4 L T

and (* check moves *) deccheck4 L (World M) =

testall (fn (n,a,t) => deccheck3 L a andalso deccheck5 L t) M

and (* check general sorts *) deccheck5 L (EType x) = deccheck1 L x |

deccheck5 L (AType x) = deccheck4 L x;

(*
```

These functions handle basic declaration checking for all the master ML types. The `error` object type and the `Error` object signal failure. The `Unknown` pseudo-object is accepted. Failure to find a declaration for a string used as a name (unless its namespace indicates that it is bound) signals failure. The function checks whether the sort associated with an identifier is of the right species (object or function) indicated by its context.

```
*)



(* get a new namespace serial number *)

fun newnameserial() = (NAMESERIAL:=1+(!NAMESERIAL);(!NAMESERIAL))

(*
```

This generates fresh numerical tags for bound variables. In any term, all variables bound with scope a particular function sort term or anonymous function term have the same numerical tag, and no two terms have the same associated tag. The renumbering commands will suppress the very large indices which will eventually occur. Would it make sense to try to dial back this counter now and then?

```
*)

(* s is a declared identifier new in the next move *)
```

```
fun isnew s = Find s (hd(!CONTEXT)) <> nil;

(* s is a variable, new and not defined *)

fun isvariable s = isnew s andalso hd(Age s (hd(!CONTEXT))) <> 0;

(*
```

`isnew` identifies strings declared at the next move ("new" things).
`isvariable` identifies strings declared at the next move which are not defined (variables).

```
*)

(* conditions for all sorts to be deducible in a term,
used to test appropriate rewrite patterns *)

fun isdefvar x = isnew x andalso not (isvariable x);
(* not currently used *)

(* here I exclude new defined variables
from type rigid terms, which is appropriate for
patterns, but one might want
to expand defined terms
 if one is using this for user-entered
lambda terms *)

(* new defined identifiers might be wanted because matching with lambda terms is now
supported:  they would be expanded -- this has been done *)

fun typerigid0 weak (EntArg(Ent(s,0))) =
(weak (* andalso not (isdefvar (EntArg(Ent(s,0)))) *) )
 orelse not(isnew(EntArg(Ent(s,0)))) |

typerigid0 weak (EntArg(App(s,0,L))) =
(weak (* andalso not (isdefvar(AbstArg(s,0)) *)
andalso testall (typerigid0 false) L) orelse
(not(isnew(AbstArg(s,0))) andalso testall (typerigid0 true) L) |

typerigid0 weak (AbstArg(s,0)) = (weak
(* andalso not (isdefvar (AbstArg(s,0))) *) )
orelse not(isnew(AbstArg(s,0))) |
```

```
typerigid0 weak x = false;

fun typerigid x = typerigid0 false (EntArg x);

(*
```

This function allows determination of terms whose type can be deduced exactly from context, a characteristic needed for rewrite patterns. It is theoretically interesting for other applications, perhaps, though type inference is not often employed in Lestrade.

The theory of how this works might of interest some time.

10/11/2017 eliminated last use of `isdefvar`; I think from the comments that this was already intended.

```
*)


(* (* rewrite tabulation lists *)

val FULLREWRITES= ref [(Unknown,Unknown)];

val _ = FULLREWRITES:=nil;

val HEADREWRITES= ref [(Unknown,Unknown)];

val _ = HEADREWRITES:=nil; *)

(* tabulation lists *)

val FULLREWRITES= ref [(Unknown,Unknown)];

val _ = FULLREWRITES:=nil;

val HEADREWRITES= ref [(Unknown,Unknown)];

val _ = HEADREWRITES:=nil;

val TYPEMATCHES = ref [((World nil,[EntArg Error]),error)];

val _ = TYPEMATCHES := nil;

val DEFMATCHES = ref [((World nil,[EntArg Error]),Error)];

val _ = DEFMATCHES := nil;
```

```
fun clearallcaches() = (FULLREWRITES:=nil;HEADREWRITES:=nil;TYPEMATCHES:=nil;DEFMATCHES:=nil
```

```
(*
```

These lists should support more efficient execution behavior for rewrite rules. Caching for type and definition information is now also implemented.

Below is the beginning of the massive complex of recursively defined functions which handles substitution, matching, type computations, rewriting, equality, and doubtless other things. This is the central engine of Lestrade, not to be touched unless absolutely necessary. The end of this block is specifically marked below: there will be various comments internal to it.

```
*)

(* This is the central dependent sort checking engine, not to be touched except
as absolutely necessary.  It appears to be quite stable *)

(* substitution and sort checking all in one package *)

(* this is a huge block of recursively declared functions *)

(* all substitutions are of an argument for another argument,
in whatever type (in the ML sense) *)

(* substitution into general sorts *)


fun typesubs a A (EType t) = EType(etypesubs a A t) |

typesubs a A (AType t) = AType(atypesubs a A (renamespace t))

(* substitutions into moves = function sorts *)

and atypesubs a A ((World nil)) = (World nil) |

atypesubs a A ((World((n,b,t)::M))) =

addworld2 (n,argsubs a A b,typesubs a A t)
(* argsubs on first component is used only for the defined value *)

(atypesubs a A ((World M)))
```

```
(* substitutions into object sorts *)

and etypesubs a A obj = obj |

etypesubs a A prop = prop |

etypesubs a A (that P) = that (entsubs a A P)|

etypesubs a A TYPE = TYPE |

etypesubs a A (IN P) = IN (entsubs a A P) |

etypesubs a A error = error

(* substitutions into object terms *)

(* notice that defined functions declared
in the next move are expanded using defmatchcomp.
Trivial substitutions are often made to enforce this expansion.

The reason for this is that these substitutions
 are made for sorts to be recorded in the parent context,
so functions (and objects) defined in the next move must pass out of scope.
*)

(* the precise way that this works when the first argument
is not a variable could be tweaked.  This option is only used
in the deduction of implicit arguments.  A further refinement
which would further enhance the power of the implicit argument
feature is the ability to substitute for a lambda term *)

and entsubs (EntArg x) (EntArg A) (Ent(s,n)) =

    if x = Ent(s,n) then A else Ent(s,n) |

entsubs (EntArg x) (EntArg A) (App(s,n,M)) =

(* the isapp x test here prevents everything from slowing to
a crawl.  It might more accurately be "x is not a variable" *)

(* isapp is the precise test that is usable without serious
performance deficits -- presumably caused by lots of wheel spinning
in equalentities. *)

    if isapp x andalso equalentities x (App(s,n,M)) then A else
```

```
    if n=0 andalso Age (AbstArg(s,n)) (hd(!CONTEXT)) = [0]

    then entsubs (EntArg x) (EntArg A)

(defmatchcomp true (getabstype(hd(Find (AbstArg(s,n)) (hd(!
        CONTEXT)))) )
    (map(argsubs (EntArg x) (EntArg A)) M))

    else App(s,n,map(argsubs (EntArg x) (EntArg A)) M) |

entsubs (AbstArg(s,n)) (AbstArg(S,N)) (App(t,m,M)) =

   if m=0 andalso Age (AbstArg(t,m)) (hd(!CONTEXT)) = [0]
   then (* entsubs (AbstArg(s,n)) (AbstArg(S,N)) *)
   (defmatchcomp true (getabstype(hd(Find (AbstArg(t,m)) (hd(!
        CONTEXT)))) )
    (map(argsubs (AbstArg(s,n)) (AbstArg(S,N))) M))

else if s=t andalso n=m

then (* entsubs (AbstArg(s,n)) (AbstArg(S,N)) *)
    (App(S,N,map(argsubs (AbstArg(s,n)) (AbstArg(S,N))) M))

else App(t,m,map(argsubs (AbstArg(s,n)) (AbstArg(S,N))) M) |

(* frank beta substitution -- I do not know if this can
actually happen *)  (* yes, it can, and it was scrambled *)

entsubs (AbstArg(s,n))(Lambda w) (App(t,m,M)) =

if m=0 andalso Age (AbstArg(t,m)) (hd(!CONTEXT)) = [0]
   then entsubs (AbstArg(s,n)) (Lambda w)
   (defmatchcomp true (getabstype(hd(Find (AbstArg(t,m)) (hd(!
        CONTEXT)))) )
    (map(argsubs (AbstArg(s,n)) (Lambda w)) M))

else if s=t andalso n=m

then (* entsubs (AbstArg(s,n)) (Lambda w) *)
    (defmatchcomp true w
    (map(argsubs (AbstArg(s,n)) (Lambda w)) M))

else App(t,m,map(argsubs (AbstArg(s,n)) (Lambda w)) M)|

(* this is a new case which will be used by the multisubs
function in very fancy implicit argument inference -- it
```

```
will take exactly this form, a known function being
replaced by application of a bound variable *)

entsubs (Lambda w) (AbstArg(s,n)) (App(t,m,M)) =

let val T = App(t,m,M) in

let val BODY = lambdabody (deworld w) in

let val INPUTS = lambdainputs (deworld w) in

let val MATCH = ematch false BODY T in

if MATCH <> nil then listsubsmod entsubs (hd MATCH)
(App(s,n,map pi23 INPUTS))

else App(t,m,map (argsubs (Lambda w) (AbstArg(s,n))) M)

end end end end |

entsubs x y T = T

(* substitutions into arguments --
notice that function arguments declared in the next move
are replaced with lambda-terms.
Again, this is because the terms we produce with these substitution functions
need to make sense in the last move,
where identifiers declared in the next move are out of scope. *)

and argsubs a A (EntArg x) = EntArg (entsubs a A x) |

argsubs x y (AbstArg(t,m)) =

   if m=0 andalso Age (AbstArg(t,m)) (hd(!CONTEXT)) = [0]

   then argsubs x y
(Lambda(getabstype(hd(Find(AbstArg(t,m))(hd(!CONTEXT))))))

else if x=AbstArg(t,m) andalso x<>y

then argsubs x y y  (*this weird seeming maneuver forced in case
y is expandable *)

else AbstArg(t,m) |
```

```
argsubs x y (Lambda T) =

if x=y then

Lambda (atypesubs x y T)

else  Lambda (atypesubs x y (renamespace T))

(*
```

The first block in the central engine is made up of substitution functions. All of these functions substitute an `Argument` for another `Argument`, in whatever type.

`typesubs` implements substitutions into `Type`, calling `atypesubs` and `etypesubs`.

`atypesubs` substitutes into moves.

`etypesubs` substitutes into object sorts.

`entsubs` makes substitutions into object terms. There is considerable power in the way `entsubs` is carried out. If the target to be replaced is a complex term, Lestrade will check whether the term it is looking at matches it (using `equalentities`) and so may carry out a non-obvious replacement. Definitional expansion of defined notions defined in the next move will be carried out before substitutions (as these need to be eliminated before being saved to the last move). Trivial substitutions are often forced (as of `Unknown` for `Unknown`) to trigger definitional expansions of this kind. Function argument with atomic names may be replaced with lambda terms, which will cause beta reduction. There is a fancy matching mode of `entsubs` in which matching may be used to replace (expanded!) application of a target lambda term with application of a bound function variable.

For most purposes, `entsubs` is replacing atomic items (objects and functions with string names) with target items in more or less straightfoward ways. The other cases are used for implicit argument inference.

`argsubs` carries out substitutions into arguments. Note that `argsubs` carries out the replacement of named functions defined at the next move with anonymous function terms (λ terms) because their names would otherwise pass out of scope if recorded at the last move.

```
*)

(* this is actually matching of sorts not strict equality:
the first sort may be vaguer in having Unknown in the definition field
at the end where the second is actually defined. *)

(* we need equalentities here, allowing definition expansion (and rewrites) to
establish equality *)
```

```
(* the exact argument if true causes us to check
for equal lambda-terms rather than equal dependent sorts,
so we pay attention to the first projection of the last item *)

and equaltypes exact (EType x)  (EType y) = x<> error
andalso y<> error andalso (x=y orelse equalenttypes x y) |

equaltypes exact (AType(World [(n,a,t)]))  (AType(World [(m,b,u)]))
 = equaltypes false t u   andalso (not exact orelse
equalentities (deent a) (deent b))   |

equaltypes exact (AType(World ((n,a,t)::L)))(AType(World((m,b,u)::M)))=
if not(equaltypes false t u) then false
else
if exact then
equaltypes true (typesubs a b (AType(World L)))
(AType(World M))

else equaltypes true (AType(Cleantype0(World ((n,a,t)::L))))(AType(Cleantype0(World((m,b,u)

 |

equaltypes exact x y = false

(* equality of object sorts *)

and equalenttypes (that P) (that Q) = equalentities P Q |

equalenttypes (IN P) (IN Q) = equalentities P Q |

equalenttypes error x = false |

equalenttypes x error = false |

equalenttypes x y = x=y

(* equality of lambda terms *)

and equivlambdas (Lambda x) (Lambda y) =

if x=y then true else

(* there needs to be a way to handle matching with abstractions
with deferred definitions, but it is tricky *)
```

```
equaltypes true (AType x) (AType y) |

equivlambdas x y = x=y

(* equality of object terms.  This function allows expansion of defined functions
declared anywhere in the context to justify equation of object terms *)

and equalentities  (App(s,n,M)) (App(t,u,N))

= if App(s,n,M)=App(t,u,N) then true else

if s=t andalso n=u andalso equalentitieslist M N then true else
(* testing different order 10/12/2017 *)

let val T = expand3(App(s,n,M)) and U = expand3 (App(t,u,N))
and V = rewriteonce  (App(s,n,M)) and W=rewriteonce  (App(t,u,N)) in

if T=Deferred andalso U = Deferred then false else

if
    if V <> App(s,n,M) then equalentities V (App(t,u,N))
(* changed order of arguments here 10/12/17 *)

    else if W <> App(t,u,N) then equalentities W (App(s,n,M)) else false

       then true

       else

            if  T <> Deferred andalso T <> App(s,n,M)

                then equalentities T (App(t,u,N))

                else if U <> Deferred andalso U <> App(t,u,N)

                       then equalentities U (App(s,n,M))

                       else

   if T = Deferred
andalso  u = 0 andalso n=0 andalso
equalenttypes (entitytype(App(s,n,M)))
(entitytype(App(t,u,M)))
then (Define0 s M (App(t,u,N));not(!BREAKOUT))

   else if U = Deferred
```

```
andalso n=0 andalso u = 0 andalso
equalenttypes (entitytype(App(s,n,M)))
(entitytype(App(t,u,M)))
then (Define0 t N (App(s,n,M));not(!BREAKOUT))


   else false

(* else s=t andalso n=u andalso equalentitieslist M N *)

end

|

equalentities (App(s,n,M)) x =

if App(s,n,M) = x then true else

let val T = expand3(App(s,n,M)) and V = rewriteonce  (App(s,n,M)) in

if V <> App(s,n,M) andalso equalentities V x

   then true

   else if T <> Deferred andalso T <> App(s,n,M) andalso equalentities T x

   then true else

   if T = Deferred andalso n=0
 andalso equalenttypes(entitytype(App(s,n,M)))(entitytype x)
then (Define0 s M x;not(!BREAKOUT)) else
   false end

|  equalentities x (App(s,n,M)) =

equalentities (App(s,n,M)) x  (* modified order of execution 10/12/2017 *)

(* if x=App(s,n,M) then true else

let val T = expand(App(s,n,M)) and V = rewriteonce  (App(s,n,M)) in

if V <> App(s,n,M)

then equalentities x V

else if T <> App(s,n,M)
```

```
then equalentities x T

else false end *)
 |

equalentities x y = x=y

(* equality of argument lists [name is deceptive] *)

and equalentitieslist nil nil = true |

equalentitieslist ((EntArg a)::L) ((EntArg b)::M) = equalentities a b andalso
  equalentitieslist L M |

equalentitieslist (a::L) (b::M) = (a=b orelse (equivlambdas (expand2 a) (expand2 b)))

andalso equalentitieslist L M |

equalentitieslist x y = false

(*
```

Here we have equality functions which try to determine whether objects of various ML types are the same.

`equaltypes` determines whether function sorts are equal. In some cases it is actually determining whether anonymous function terms (λ-terms) are equal, in which case it pays attention to the second component of the last triple (if the `exact` parameter is true). When it is checking equality of function sorts (`exact` is false) it ignores the second component of the last triple. Note that `equaltypes` handles α-conversion (renaming of bound variables) tidily.

`equalenttypes` handles equality of object sorts.

`equivlambdas` implements equality of lamba terms by invoking equality of function sorts in its exact form.

`equalentities` handles equality of object terms. It will attempt rewriting, then definitional expansion, to get equality if it does not see literal equality. Notice that it rewrites in single steps of rewriting, then single steps of definitional expansion, when attempting to establish equality. It will now backtrack from a rewrite step to attempt a definitional expansion step.

corrected a case in `equalentities` where rewriting was left out, 10/11/2017. Major modifications of execution order in `equalentities` 10/12/2017. It now backtracks to expand definitionally if a rewrite leads it astray. Two changes: it checks for matching of arguments if top operators match before attempting rewriting or definitional expansion (this appeared to make it much faster) and it switches the order of arguments when it does do definitional expansion or

rewriting, so that it will attempt to work on both sides of an equation. The latter move might not be helpful, but is readily fixed.

I am considering whether it is dangerous to allow it to attempt definitional expansion if rewriting takes it astray. Would this lead to too much backtracking? I have installed this 10/12: if the approach using rewriting fails, it will return and attempt definitional expansion.

`equalentities` is currently the one context in which deferred definitions can be automatically updated to full definitions. It should eventually be possible for `equivlambdas` to trigger redefinition.

`equalentitieslist` is equivalence of argument lists (which incorporates the entire problem of equality of arguments). It is worth noting that it `will` successfully match function constants to lambda terms to which they are definitionally equivalent, using `expand2`.

```
*)

(* work on sort computation *)

(* compute object sorts *)

and entitytype (Ent(s,0)) = let val S = stringdef s (!CONTEXT)
in if S=nil orelse not(isenttype(pi1(hd S))) then

(saypause ("Did not find object "^s^" (entitytype)");Flush();
error)
else getenttype(pi1(hd S))  end |

entitytype (App (s,0,M)) = let val S = stringdef s (!CONTEXT)
in if S=nil orelse isenttype(pi1(hd S)) then
(saypause ("Did not find function "^s^" (entitytype)");Flush();error)
else typematchcomp (getabstype(pi1(hd S))) M end |

entitytype x = error

(* compute argument sorts.  Notice the easy case for lambda terms *)

and argtype (EntArg x) = EType(entitytype x) |

argtype (AbstArg (x,n)) = let val S = stringtype x in
   if S=nil orelse isenttype(pi1(hd S)) then
(saypause ("Did not find function "^x^" (argtype)");
EType error)
   else pi1(hd S) end |

argtype (Lambda x) = Cleantype1(AType x)
```

```
(*
```

We begin the section on computing sorts of terms.

The function `entitytype` computes types of object terms; it is an easy lookup for atomic terms and calls the main sort computation algorithm `typematchcomp` for application terms.

The function `argtype` computes types of arguments. It probably should invoke a `Cleantype` function in addition to doing the very simple thing it does for λ-terms.

```
*)

(* the sort matching algorithm for dependent sorts; this
returns the correct sort given the full sort of the applied
function (inputs and output) and the list of sorts of the input,
or error if matches fail. *)

(*

and typematchcomp (World L) M = typematchcomp0 (deworld(renamespace(World L)))
    (map (fn m=>(0,m,argtype m)) M)

and typematchcomp0 [(n,a,EType t)] nil
    = etypesubs (EntArg Unknown) (EntArg Unknown) t |

typematchcomp0 ((n,a,t)::L) ((m,A,T)::M) =
     if not(equaltypes false t T) then (saypause ("Sort "^(display6 t)^" of "^
(display4 a)^" does not match sort "^(display6 T)^" of "^(display4 A));Flush();error)
     else typematchcomp0
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |

typematchcomp0 x y = error

and typematchcomp1 [(n,a,t)] nil = typesubs (EntArg Unknown) (EntArg Unknown) t |

(* typematchcomp1 is used to make substitutions into initial
segments of types in FixListType below,
to handle curried notations for function arguments 10/10 mods *)

typematchcomp1 ((n,a,t)::L) ((m,A,T)::M) =
     if not(equaltypes false t T) then (saypause ("Sort "^(display6 t)^" of "
     ^(display4 a)^" does not match sort "^(display6 T)^" of "^(display4 A));
```

```
    Flush();EType error)
     else typematchcomp1
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |

typematchcomp1 x y = EType error


(* sort matching with no error message reports -- used by
the implicit argument discovery feature *)

and silenttypematchcomp0 [(n,a,EType t)] nil
     = etypesubs (EntArg Unknown) (EntArg Unknown) t |

silenttypematchcomp0 ((n,a,t)::L) ((m,A,T)::M) =
     if not(equaltypes false t T) then
((* saypause ("Type "^(display6 t)^" of "^(display4 a)^
" does not match type "^(display6 T)^" of "^(display4 A));Flush(); *) error)
     else silenttypematchcomp0
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |

silenttypematchcomp0 x y = error


(* this function computes values of defined functions.
It has a parameter which if set to false would do sort checking,
but in fact it is only used in safe context so far
(on things already known to have been sort checked). *)

and defmatchcomp safe (World L) M = defmatchcomp0 safe (deworld(renamespace(World L)))
    (map (if safe then (fn m=>(0,m,EType error)) else (fn m=>(0,m,argtype m))) M)

and defmatchcomp0 safe [(n,EntArg a,EType t)] nil = a |

defmatchcomp0 safe ((n,a,t)::L) ((m,A,T)::M) =
     if (not safe) andalso (not(equaltypes false t T)) then Error
     else defmatchcomp0 safe
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |
defmatchcomp0 safe x y = Error

*)

and typematchcomp (World L) M =
```

```
let val FF = Abstractfind (World L,M) (TYPEMATCHES) in

if FF <> nil then hd FF

else let val M2 = (map (fn m=>(0,m,argtype m)) M) in

let val RESULT = typematchcomp0 (deworld(renamespace(World L)))
    M2 in

(TYPEMATCHES:=((World L,M),RESULT)::(!TYPEMATCHES);RESULT)

end end end

and typematchcomp0 [(n,a,EType t)] nil
    = etypesubs (EntArg Unknown) (EntArg Unknown) t |

typematchcomp0 ((n,a,t)::L) ((m,A,T)::M) =
     if not(equaltypes false t T) then (saypause ("Sort "^(display6 t)^" of "^
(display4 a)^" does not match sort "^(display6 T)^" of "^(display4 A));Flush();error)
     else typematchcomp0
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |

typematchcomp0 x y = error

and typematchcomp1 [(n,a,t)] nil = typesubs (EntArg Unknown) (EntArg Unknown) t |

(* typematchcomp1 is used to make substitutions into initial
segments of types in FixListType below,
to handle curried notations for function arguments 10/10 mods *)

typematchcomp1 ((n,a,t)::L) ((m,A,T)::M) =
     if not(equaltypes false t T) then (saypause ("Sort "^(display6 t)^" of "
     ^(display4 a)^" does not match sort "^(display6 T)^" of "^(display4 A));
    Flush();EType error)
     else typematchcomp1
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |

typematchcomp1 x y = EType error


(* sort matching with no error message reports -- used by
the implicit argument discovery feature *)

(* and silenttypematchcomp0 (World L) M =
```

```
let val FF = abstractfind (World L,map (fn (a,b,c) => b) M) (!TYPEMATCHES) in

if FF <> nil then hd FF

else (* let val M2 = (map (fn m=>(0,m,argtype m)) M) in *)

let val RESULT = silenttypematchcomp1 (deworld(renamespace(World L)))
    M in

(TYPEMATCHES:=((World L,map (fn (a,b,c) => b) M),RESULT)::(!TYPEMATCHES);RESULT)

end end (* end *)


and silenttypematchcomp1 [(n,a,EType t)] nil
     = etypesubs (EntArg Unknown) (EntArg Unknown) t |

silenttypematchcomp1 ((n,a,t)::L) ((m,A,T)::M) =
     if not(equaltypes false t T) then
((* saypause ("Type "^(display6 t)^" of "^(display4 a)^
" does not match type "^(display6 T)^" of "^(display4 A));Flush(); *) error)
     else silenttypematchcomp1
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M | *)

and silenttypematchcomp0 x y =

let val Y = map (fn (a,b,c) => b) y in

let val FF = Abstractfind (World x,Y) (TYPEMATCHES) in

if FF <> nil then hd FF

else (* let val M2 = (map (fn m=>(0,m,argtype m)) M) in *)

let val RESULT = silenttypematchcomp1 (deworld(renamespace(World x)))
    y in

(TYPEMATCHES:=((World x,Y),RESULT)::(!TYPEMATCHES);RESULT)

end end end (* end *)

and silenttypematchcomp1 [(n,a,EType t)] nil
     = etypesubs (EntArg Unknown) (EntArg Unknown) t |
```

```
silenttypematchcomp1 ((n,a,t)::L) ((m,A,T)::M) =
     if not(equaltypes false t T) then
((* saypause ("Type "^(display6 t)^" of "^(display4 a)^
" does not match type "^(display6 T)^" of "^(display4 A));Flush(); *) error)
     else silenttypematchcomp1
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |

silenttypematchcomp1 x y = error


(* this function computes values of defined functions.
It has a parameter which if set to false would do sort checking,
but in fact it is only used in safe context so far
(on things already known to have been sort checked). *)

and defmatchcomp safe (World L) M =

let val FF = Abstractfind (World L,M) (DEFMATCHES) in

if FF <> nil then hd FF else
let val RESULT=
defmatchcomp0 safe (deworld(renamespace(World L)))
    (map (if safe then (fn m=>(0,m,EType error)) else (fn m=>(0,m,argtype m))) M)
in
(DEFMATCHES:=((World L,M),notdeferred RESULT)::(!DEFMATCHES);RESULT) end end

and defmatchcomp1 safe (World L) M =

let val FF = Abstractfind (World L,M) (DEFMATCHES) in

if FF <> nil then hd FF else
let val RESULT=
defmatchcomp0 safe (deworld(renamespace(World L)))
    (map (if safe then (fn m=>(0,m,EType error)) else (fn m=>(0,m,argtype m))) M)
in
(DEFMATCHES:=((World L,M),RESULT)::(!DEFMATCHES);RESULT) end end

and defmatchcomp0 safe [(n,EntArg a,EType t)] nil = a |

defmatchcomp0 safe ((n,a,t)::L) ((m,A,T)::M) =
     if (not safe) andalso (not(equaltypes false t T)) then Error
     else defmatchcomp0 safe
     (map (fn(u,v,w) => (u,argsubs a A v,typesubs a A w)) L)
     M |
defmatchcomp0 safe x y = Error
```

```
(*
```

Here are various versions of the central iterated substitution process which computes both sorts of application terms and definitional expansions of application terms. Things to note are the essential role of clever equality functions in matching and the use of trivial rewrites to force expansion of defined notions declared at the next move. These functions have been updated with caching.

`typematchcomp` computes types of application terms. It has a silent version which doesn't raise error messages when errors are encountered, which is used by the implicit argument inference feature.

`defmatchcomp` computes values of defined functions by basically the same algorithm. This function does no internal sort checking, though it is set up with a parameter which would force this, because it is never applied except in situations where the components used have already been sort checked.

All of these functions take first argument `World L` which is the sort of the function being applied (input and output) and the list `M` of sorts of the inputs (which will be one shorter, except in the case of the function `typematchcomp1` which computes types of "curried function arguments" which may have arguments lists which are shorter.

```
*)

(* one step of expansion of a defined function in applied position *)

and notdeferred Deferred = (saypause "Cannot expand deferred definition";Error) |

notdeferred x = x

and expand (App(s,n,M)) =

    if n=0 andalso stringAge s = [0] andalso stringtype s <> nil
    then (entsubs (EntArg Unknown) (EntArg Unknown)
    (defmatchcomp true(getabstype(pi1(hd(stringtype s)))) M))
    else App(s,n,M) |

expand x = x

(* expansion of an argument to a lambda-term if it is defined *)

and expand2 (AbstArg(s,n)) =

if n=0 andalso stringAge(s) = [0] andalso stringtype s <> nil
```

```
then argsubs (EntArg Unknown) (EntArg Unknown)
(Lambda(getabstype(pi1(hd(stringtype s)))))
else AbstArg(s,n) |

expand2 x=x

and expand3 (App(s,n,M)) =

    if n=0 andalso stringAge s = [0] andalso stringtype s <> nil
    then entsubs (EntArg Unknown) (EntArg Unknown)
    (defmatchcomp1 true(getabstype(pi1(hd(stringtype s)))) M)
    else App(s,n,M) |

expand3 x = x

(*
```

Here are some expansion functions.

`expand` does one step of definitional expansion of an application term with a defined top level operation. It leaves other terms alone.

`expand2` will expand a defined function appearing as an argument by itself to a λ-term.

`expand3` is the only version which will not trigger error if the result is `Deferred`. It is used in matching of object terms to notice when redefinition should be forced.

```
*)

(* moving all bound variables in a dependent sort or lambda term
to a new namespace, before a substitution into one of these is made *)

and renamespace (World L) = World (renamespace0 (newnameserial())L)

and renamespace0 N [(n,a,t)] = [(n,a,t)] |

renamespace0 N ((m,(AbstArg(s,n)),t)::L)=
(m,(AbstArg(s,N)),t)::
(deworld(atypesubs (AbstArg(s,n)) (AbstArg(s,N)) (World (renamespace0 N L)))) |

renamespace0 N ((m,(EntArg(Ent(s,n))),t)::L)=
(m,(EntArg(Ent(s,N))),t)::
(deworld(atypesubs (EntArg(Ent(s,n))) (EntArg(Ent(s,N))) (World(renamespace0 N L)))) |

renamespace0 N x = (saypause "Bad case in renamespace";nil)
```

```
(*
```

This function applies a fresh numerical tag to all variables bound at the top level in a function sort term or anonymous function term. This function is applied before any substitution is made into such a term to avert bound variable collision problems.

```
*)

(* matching function to be added at this point.   Two object terms are matched.
A list of matches for variables is produced *)

(* it is demonstrable that if two sort safe expressions, the first of which
is not a variable, match successfully, then they will in fact be of the same sort *)

(* solving confluence issues by requiring that executable subterms in the body
be head-rewritten before matching:  an executable can only match anything in a context
in which it has no execution behavior (except at the top of course) *)

and ematch b (Ent(s,n)) t =

   if n<>0

   then [[(EntArg(Ent(s,n)),EntArg t)]]

   else if (Ent(s,n)) = t (* andalso (not b orelse headrewrite (Ent(s,n)) = Ent(s,n)) *)

(* rewrite patterns are never atomic *)

       then [nil]
       else nil |

ematch b (App(s,n,L)) (App(t,n2,M)) = (HEADREWRITES:=nil;

   let val T = if b
               then (*if  *) headrewrite (App(t,n2,M)) (*=App(t,n2,M)
                       then App(t,n2,M) else Error *)
               else App(t,n2,M) in

   if s <> appof T then nil

   else argmatch  L (argsof T) end) |
```

```
ematch b x y = nil

and argmatch  nil nil = [nil] |

argmatch  x nil = nil |

argmatch  nil x = nil |

argmatch  ((EntArg x)::L) ((EntArg y)::M) =

   mergematch (ematch true x y) (argmatch  L M) |

argmatch  (AbstArg(s,n)::L) (t::M) =

  if n<>0 then

  mergematch [[(AbstArg(s,n),t)]] (argmatch  L M)

  else if AbstArg(s,n) = t then argmatch  L M else nil |

argmatch ((Lambda (World[(n,s,t)]))::LL) ((Lambda (World[(N,S,T)]))::MM)
    = argmatch (s::LL) (S::MM) |

argmatch ((Lambda (World((n,s,t)::L)))::LL)
     ((Lambda (World((N,S,T)::M)))::MM) =

argmatch ((Lambda (World L))::LL)
      ((Lambda (atypesubs S s (World M)))::MM) |

argmatch  x y = nil

and mergematch nil x = nil |

mergematch x nil = nil |

mergematch [nil] x = x |

mergematch x [nil] = x |

mergematch x y = let val M = mergematch0 (hd x) (hd y) in
   if M = nil then nil else [M] end

and mergematch0 nil L = nil |

mergematch0 L nil = nil |
```

```
mergematch0 ((s,t)::L) M =

   let val N = abstractfind s (L @ M) in

   if N = nil then if L <> nil then (s,t)::(mergematch0 L M)
      else (s,t)::M

   else if equalarguments t (hd N) then

     if abstractdrop s L = nil andalso abstractdrop s M = nil then [(s,t)]

     else if abstractdrop s L <> nil
     then (s,t)::(abstractdrop s (mergematch0 L M))

     else (s,t)::(abstractdrop s M)

   else nil end

and equalarguments (EntArg s) (EntArg t) = equalentities s t |

equalarguments (Lambda x) (Lambda y) = equivlambdas (Lambda x) (Lambda y) |

equalarguments x y = x = y

(* tools for implementing substitutions by pattern matching over
any of the various sorts for which we have substitution functions *)

and listsubsmod subsfun nil T = T |

listsubsmod subsfun ((s,t)::L) T = subsfun s t (listsubsmod subsfun L T)

and matchsubs subsfun pattern target body =
let val M = ematch false pattern body in if M = nil then body else
listsubsmod subsfun (hd M) target end

(*
```

Matching and matching-driven substitution, used by the rewriting feature and also by implicit argument inference.

Matching functions output a list of substitutions of arguments with the desired result.

`ematch` handles matching of object terms. To enforce confluence, it is required that no subterm of a pattern can match something which can itself be head-rewritten: subterms of the target are head-rewritten before matching to enforce this. An entire pattern can of course match something which can be

rewritten; the parameter `b` of `ematch` is false in this case. I believe that checks of atomic terms for being rewriteable are unnecessary, as in fact atomic terms cannot be rewrite patterns (or this is not possible at the moment).

The function `argmatch` handles matching of arguments. Notable is an attempt to implement matching of λ-terms which needs to be tested.

`mergematch` handles merging of match lists in a standard way: note the use of `equalarguments` to extend our ability to reconcile matches.

The function `equalarguments` looks like a stray from the equality section.

`listsubsmod` is a polymorphic tool for executing substitutions based on match lists.

`matchsubs` implements substitutions driven by rewrite rules.

```
*)

(* apply first applicable rewrite rule, just once *)

and rewriteoncewithalist  nil t = t |

rewriteoncewithalist  ((s,(t,u))::L) T =

   let val M = ematch false t T in

       if M = nil then rewriteoncewithalist  L T

       else matchsubs  entsubs t u T end

and rewriteoncewithalistlist  nil T = T |

rewriteoncewithalistlist  (L::M) T =

let val U =rewriteoncewithalist  L T in

if U <> T then U

else rewriteoncewithalistlist M T end

and rewriteonce T = if (!REWRITEVER)
   then rewriteoncewithalistlist (tl(!REWRITES)) T else T

(* complete rewriting *)

and fullrewrite (App(s,0,L)) =

let val FF = Abstractfind (App(s,0,L)) (FULLREWRITES) in
```

```
if FF <> nil then hd FF

else

let val L1 = map fullrewrite2 L in

let val T1 = rewriteonce (App(s,0,L1))

in if T1 = App(s,0,L1) then (FULLREWRITES:=(App(s,0,L),T1)::(!FULLREWRITES);T1)

else let val T2 = fullrewrite T1 in
(FULLREWRITES:=(App(s,0,L),T2)::(!FULLREWRITES);T2)

end

end end end |

(* fullrewrite (Ent(s,0)) =

let val FF = Abstractfind (Ent(s,0)) (FULLREWRITES) in

if FF <> nil then hd FF

else

let val T1 = rewriteonce (Ent(s,0)) in

if T1 = Ent(s,0) then (FULLREWRITES:=((Ent(s,0)),T1)::(!FULLREWRITES);T1)

else let val T2 = fullrewrite T1 in
(FULLREWRITES:=(Ent(s,0),T2)::(!FULLREWRITES);T2)

end


end end | *)

(* rewrite patterns are never atomic *)

fullrewrite x = x

and fullrewrite2 (EntArg x) = EntArg (fullrewrite x) |

fullrewrite2 x = x

(* head rewriting (just from the top) *)
```

```
and headrewrite (App(s,0,L)) =

let val FF = Abstractfind (App(s,0,L)) (HEADREWRITES) in

if FF <> nil then hd FF else

let val T1 = rewriteonce  (App(s,0,L))

in if T1 = App(s,0,L) then (HEADREWRITES:= (App(s,0,L),T1)::(!HEADREWRITES);T1)

else let val T2 = headrewrite T1 in

(HEADREWRITES:= (App(s,0,L),T2)::(!HEADREWRITES);T2)


end

end end |

(* headrewrite (Ent(s,0)) =

let val FF = Abstractfind (Ent(s,0)) (HEADREWRITES) in

if FF <> nil then hd FF

else

let val T1 = rewriteonce (Ent(s,0)) in

if T1 = Ent(s,0) then (HEADREWRITES:=((Ent(s,0)),T1)::(!HEADREWRITES);T1)

else let val T2 = headrewrite T1 in
(HEADREWRITES:=(Ent(s,0),T2)::(!HEADREWRITES);T2)

end


end end | *)

(* rewrite patterns are never atomic *)

headrewrite x = x

and isordered nil = true |
```

```
isordered [a] = true |

isordered (a::(b::L)) =

hd(Age a (hd(!CONTEXT))) <> 0 andalso

(hd(Age a (hd(!CONTEXT))) < hd(Age(b)(hd(!CONTEXT)))
   andalso isordered (b::L))

(*
```

Checks the property of the argument list of identifiers in a declaration command that they must appear in order of nonzero age (and they must appear in the next move). (zero age signals that an identifier is defined, and so cannot be a parameter).

This now appears here because a version of the `define` user command appears in the internals of the logic engine for deferred definitions.

```
*)

and worlditem0 s = (hd(Age s (hd(!CONTEXT))),s,
typesubs (EntArg Unknown) (EntArg Unknown)
(hd(Find s (hd(!CONTEXT)))))

and worldof0 L = World((* guardedexpandlist *) (map worlditem0 L))

and Define0 s L T = (FULLREWRITES:=nil;

if hd(!REWRITES) <> nil then saypause "Define command blocked by hypothetical rewrites" else

if not(testall isvariable L) then saypause "Some argument is not variable"

(* same remark on the argument order test as above *)

else if not (isordered L) then saypause "Arguments are in the wrong order"

else

let val T0 = T in

(* if reserved s orelse extended s orelse stringtype s <> nil
then saypause ("Identifier "^s^" is not fresh")

else *) if not let val T2 = ( (entsubs (EntArg Unknown) (EntArg Unknown) T))
```

```
and THETYPE = (* dotfix (deworld(worldof L)) *)
(etypesubs (EntArg Unknown) (EntArg Unknown) (entitytype T0)) in (
SAVECONTEXT0:=(!CONTEXT);

CONTEXT:= (worldof0 (* worldof2 *) L)::(map(fn (World LL) =>World(findandpurge(AbstArg(s,0)

let val CHECK = deccheck4 (!CONTEXT) (hd(!CONTEXT)) andalso
deccheck1 (!CONTEXT) THETYPE andalso deccheck2 (!CONTEXT) T2

in (CONTEXT:=(!SAVECONTEXT0);CHECK)

end
) end then saypause "Sort check or dependency failure"
else let val TT = fullrewrite(entsubs (EntArg Unknown) (EntArg Unknown) (T))
 and  THETYPE = (* dotfix(deworld(worldof L)) *)
(etypesubs (EntArg Unknown) (EntArg Unknown) (entitytype (T0)) ) in (
let val TTTT = (Reset();Reindex3(
AType(renamespace(addtoworld0 (worldof0 L) (0,EntArg TT,EType (THETYPE)))))) in

(CONTEXT:= map (fn World LL => World(findandreplace (AbstArg(s,0)) TTTT LL)) (!CONTEXT);show

end)



end end);

(*
```

These are the rewriting functions.

`rewriteoncewithalist` implements rewriting, just once, using the first applicable rewrite rule appearing in a list given as an argument.

`rewriteoncewithalistlist` does that same thing with a list of rewrite lists.

`rewriteonce` uses the actual current list of active rewrites (which is a list of rewrite lists from different moves) as the list of lists parameter to `rewriteoncewithalistlist`.

`fullrewrite` implements aggressive rewriting wherever possible. The object term output of the `Define` command is aggressively rewritten; note that sorts are never rewritten, though rewrites can be used to justify viewing sorts as equal. It is useful to note that `fullrewrite` does not rewrite the innards of λ-terms: only entity arguments are rewritten by the auxiliary function `fullrewrite2`. This might be something to explore.

`headrewrite` implements rewriting from the top only; it is currently used only in the innards of the matching function.

This is the end of the huge block of mutually recursive functions which makes up the central engine of Lestrade.

```
*)

(* utilities for name collision checks *)

(* utility for extending names *)

fun isnumeral c = (#"0" <= c andalso c <= #"9") orelse c = #"'";

fun isspecial  c = c= #"~"

orelse c = #"@" orelse c = #"#" orelse c = #"$"
orelse c = #"%" orelse c = #"^" orelse c = #"&"
 orelse c = #"*" orelse c = #"-" orelse c = #"+"
 orelse c = #"=" orelse c = #"|" orelse c = #";" orelse c = #"." orelse c = #"<"
orelse c = #">" orelse c = #"?" orelse c = #"/"
 orelse c = #"!" orelse c = #".";

(* an identifier starting with a special character can be extended with $;
any other identifier can be extended with ' (single quote) *)

fun extend s = if isspecial(hd(explode s)) then s^"$" else s^"'";

fun extended s = length(explode s)>1 andalso (
hd(rev(explode s)) = #"$"
orelse hd(rev(explode s)) = #"'");

fun extendenough s context =
   if stringdef s context = nil then s else extendenough(extend s)context;

fun extendenough2 (AbstArg(s,0)) context = AbstArg(extendenough s context,0) |

extendenough2 (EntArg(Ent(s,0))) context = (EntArg(Ent(extendenough s context,0))) |

extendenough2 x context = x;

fun makeadjoinable (World nil) rewrites context = World nil |

makeadjoinable (World ((n,s,t)::L)) rewrites context =

    let val LL = makeadjoinable (World L) rewrites context in

    if stringdef (nameof s) [LL] <> nil
    then (saypause ("Essential name conflict with "^(nameof s));World nil) else

    if stringdef (nameof s) (context) = nil
```

```
    then addworld2 (n,s,t) LL

    else if rewrites <> nil then (saypause "Lestrade will not extend names in saved worlds w

    else let val ss = extendenough2 s (context) in

    atypesubs s ss (addworld2 (n,s,t) LL)

    end end

(*
```

This section provides functions for generating new names to replace ones that cannot be used due to name collisions.

The classes of characters `isnumeral` and `isspecial` play a role in the formulation of the possible shapes of Lestrade identifiers, discussed below.

The function `extend` applied to an identifier string extends it with `$` if it begins with a special character, and with `'` otherwise; the rules for identifiers ensure that what results will still be an identifier.

The function `extended` checks whether an identifier is extended.

The functions `extendenough` and `extendenough2` are used to extend identifiers enough times that they become undeclared.

The function `makeadjoinable` acts on a move by globally replacing each name declared in the move which is declared in the current context with a sufficiently extended version to avoid name conflicts. This is used when opening saved moves which might contain names which conflict with names declared in the context since they were saved.

```
*)


(* user command:  open a new move (or a previously saved move) *)

(* USER COMMAND *)

fun Open s = (  (*if defaultworld (!WORLDNAMES)
andalso not(defaultworld(s::(!WORLDNAMES)))
 then saypause "Cannot follow default move with named move" else *)

(NAMESERIAL:=(!Maxfreshindex); let val W =

(*let val WW1 =  *)abstractfind (s::(!WORLDNAMES)) (!SAVEDWORLDS) (* in
```

```
if WW1 <> nil then WW1 else abstractfind (s::

(makestring(length(tl(!WORLDNAMES))))::(tl(!WORLDNAMES)))
(!SAVEDWORLDS) end *)

and R = (* let val RR1 = *)

abstractfind (s::(!WORLDNAMES)) (!SAVEDREWRITES)(* in
if RR1 <> nil then RR1 else abstractfind (s
::(makestring(length(tl(!WORLDNAMES))))::(tl(!WORLDNAMES)))
(!SAVEDREWRITES)  end *)
in

if W = nil then

(say0 "blank move created";CONTEXT := (World nil)::(!CONTEXT);
REWRITES:= (nil::(!REWRITES));WORLDNAMES := s::(!WORLDNAMES))


else let val WW = makeadjoinable (hd W) (if R = nil then nil else hd R) (!CONTEXT) in

(* if (!BREAKOUT) then saypause "Name collision issues cause open command to fail"

else *) (say0 "saved move loaded";CONTEXT := (WW)::(!CONTEXT);
REWRITES := (hd R)::(!REWRITES);
WORLDNAMES := s::(!WORLDNAMES))  end end));

(*
```

This is the user command which opens a new next move. The argument is a name to be assigned to the move to be opened. One cannot open a move with a non-default name (the default name being the numeral index of the move) unless all positive indexed moves preceding it have non-default names.

If there is no saved move with the indicated name extending the current context, an empty move is adjoined to the context and given that name.

If there is a saved move with the indicated name, `makeadjoinable` is applied to it and the result becomes the next move. I do not believe the alternative which raises an error message can actually occur (it could in earlier versions).

The serial counter for namespaces, `NAMESERIAL`, is set back to `Maxfreshindex`, which is maintained as an upper bound on namespace indices in stored declarations.

```
*)
```

```
(* extract the pattern and target from the list component
of the sort of a function justifying rewrites *)

fun getpattern nil = nil |

getpattern [x] = nil |

getpattern ((i,a,EType(that (App(y,n,[t])))::x) = [t] |

getpattern (x::L) = getpattern L;

fun gettarget nil = nil |

gettarget ((i,a,EType(that (App(y,n,[t])))::nil) = [t] |

gettarget [x] = nil |

gettarget (x::L) = gettarget L;

(*
```

These functions extract the pattern and target of a rewrite rule from the sort of a function presented to justify it. I'm not sure why this utility appears at this point in the code.

```
*)

(* make a single substitution in an argument/sort list *)

fun singlesubslist s t nil = nil |

singlesubslist s T ((i,a,t)::L)
   = (1,argsubs s T a,typesubs s T t)::(singlesubslist s T L);


(*
```

The function `singlesubslist` may fall into the stray utility category.

```
*)

(* multiple substitutions for a specific case of higher order matching *)
```

```
fun negvar (EntArg(Ent(s,n))) = EntArg(Ent(s,0-n-1)) |

negvar (AbstArg(s,n)) = AbstArg(s,0-n-1) |

negvar x = x;

fun multisubs nil U T = T |

multisubs (x::L) ((n,a,t)::U) T = entsubs x (negvar a) (multisubs L U T);

fun multisubstypelist nil U = nil |

multisubstypelist (x::L) ((n,a,t)::U)
    = (0, negvar a, t)::(singlesubslist a (negvar a) (multisubstypelist L U));

(*
```

This is a utility used by the implicit type inference mechanism to deduce the value of a function variable applied to a list of constant values. The function `multisubs` delivers the body of the function and the function `multisubstypelist` delivers the type assignment for its variables. I believe that `multitypesubs` had a bug in it, which I fixed 10/15/2017 (actually `multisubstypelist` is fairly trivial now, but I'd rather leave it as it is so as not to have to rewrite the code in the implicit argument inference function).

I'm contemplating making reverse substitution (substitution of expressions for complex expressions) use a much harder criterion of equivalence than `equalentities`, which is what it currently uses. This might give finer control over implicit argument inference.

```
*)


(* this computes the list of variable dependencies of terms of various kinds. *)

fun deps (Ent(s,0)) = if isnew(EntArg(Ent(s,0)))
then (if isvariable(EntArg(Ent(s,0)))
then [(EntArg(Ent(s,0)))] else nil)@
(typedeps(argtype(EntArg(Ent(s,0))))) else nil |

deps (App(s,0,L)) = if isnew(AbstArg(s,0))
 then (if isvariable(AbstArg(s,0))
then [(AbstArg(s,0))]else nil)@
(typedeps(argtype(AbstArg(s,0)))@(depsarg L)) else depsarg L |
```

```
deps x = nil

and depsarg nil = nil |

depsarg((EntArg x)::L) = (deps x)@(depsarg L) |

depsarg ((AbstArg(s,0))::L) = if isnew(AbstArg(s,0))
then (if isvariable(AbstArg(s,0)) then [AbstArg(s,0)] else nil)
@(typedeps(argtype(AbstArg(s,0))))@(depsarg L)
   else depsarg L |

depsarg ((Lambda x)::L) = typedeps(AType x) |

depsarg (x::L) = depsarg L

and typedeps (EType(that x)) = deps x |

typedeps (EType(IN x)) = deps x |

typedeps (EType x) = nil |

typedeps (AType(World [(i,EntArg x,T)])) = (deps x)@(typedeps T) |

typedeps (AType(World((i,A,T)::L))) =
(typedeps T)@(typedeps (AType(World L))) |

typedeps (AType x) = nil;

(*
```

These functions determine the list of variables on which items of an ML type depend.

`deps` computes dependencies of an object term.

`depsarg` computes dependencies of an argument.

`typedeps` determines dependencies of a sort.

```
*)

(* functions for argument redundancy *)

(* a device for type casting an argument to a sort, used
in the internals of the main argument reduction functions
because they are doing very general term matching disguised
```

```
as sort matching, so casting is needed *)

fun arg2type (EntArg x) = EType(that x) |

arg2type (Lambda x) = AType x |

arg2type x = if expand2 x <> x then arg2type(expand2 x) else EType error;

(*
```

The `arg2type` function is a perverse gadget for type casting an argument to a sort. The implicit argument inference mechanism makes use of very general term matching disguised as sort matching, so needs such a mechanism.

```
*)

(* moretypes discovers candidate implicit arguments in the sorts
of explicitly given arguments at declaration time *)

(* it is initially analyzing a sort, but it looks into component application
terms, so it is constantly type casting arguments to sorts, weirdly *)

fun moretypes (EType(that (Ent(s,0)))) =
if isvariable (EntArg(Ent(s,0))) then [(EntArg(Ent(s,0)),EType prop)] else nil |

moretypes (EType(IN (Ent(s,0)))) = if isvariable (EntArg(Ent(s,0)))
 then [(EntArg(Ent(s,0)),EType TYPE)] else nil |

moretypes (EType(that (App(s,n,(x::L))))) =
(if isvariable(AbstArg(s,n)) then [(AbstArg(s,n),argtype(AbstArg(s,n)))]
@(moretypes(argtype(AbstArg((s,n))))) else nil)

@(if isvariable x then [(x,argtype x)]@(moretypes(EType(that(App(s,n,L)))))
else (moretypes((arg2type x)))@(moretypes(EType(that(App(s,n,L)))))) |

moretypes (EType(IN (App(s,n,(x::L))))) = moretypes (EType(that (App(s,n,(x::L))))) |

moretypes (AType(World(nil))) = nil |

moretypes (AType(World([(i,a,t)]))) = (moretypes (EType(that (deent a))))@(moretypes t) |

moretypes (AType(World((i,a,t)::L))) = (moretypes t)@(moretypes(AType(World L)))|

moretypes x = nil;
```

```
(*
```

This function discovers all variables on which a sort depends and returns a list of these variables paired with their types. It ends up doing structural induction on general terms, which means that it is doing some rather strange casting of terms of other ML types to sort terms. The output of `moretypes` is used to identify candidates for implicit arguments to be identified to an explicitly given argument list at the time a function is declared.

```
*)

(* now outline the strategy:  take the first element in the argument list.
compute the expanded alternate list of its tail.  Compute the deps of the item
and drop all dotted and undotted versions of the deps from the previous list.
Then add the item.  Then add dotted versions of its moretypes list, and replace
undotted versions with dotted versions throughout. *)

(* add or remove the initial period (.) which distinguishes
the name of an implicit argument from the name of an explicit argument *)

fun dot s = if s = "" then "" else if hd(explode s) = #"." then s else "."^s;

fun undot s = if s="" then ""
else if hd(explode s) = #"." then implode(tl(explode s)) else s;

fun argdot (EntArg(Ent(s,n))) = (EntArg(Ent(dot s,n))) |

argdot (AbstArg(s,n)) = AbstArg(dot s,n) |

argdot x = x;

fun argundot (EntArg(Ent(s,n))) = (EntArg(Ent(undot s,n))) |

argundot (AbstArg(s,n)) = AbstArg(undot s,n) |

argundot x = x;

(*
```

Implicit arguments in reported sorts of declared functions are dotted (an initial period is affixed). These functions manage the attachment of dots and removal of dots.

```
*)

fun incrementlist nil = nil |

incrementlist ((i,a,t)::L) = (i+1,a,t)::(incrementlist L);

(*
```

Some trouble is taken to keep the ages of items in argument lists in the correct order. It isn't entirely clear to me as I work on the first pass of the literate programming version whether this is actually needed, but there might indeed be circumstances where expanded argument lists are converted to moves, and in such circumstances this would be important. `incrementlist` is a utility used for this purpose.

```
*)


(* this function adds dotted items to an argument list. It is complicated
by the need to ensure that when dotted items
are added in a bloc (output of moretypes)
to an argument list that their order is corrected if necessary
to keep dependencies sound *)

fun addotlist nil L = L |

addotlist ((s,t)::M) nil = [(1,argdot s,t)] |

addotlist ((s,t)::M) ((i,a,T)::L) = addotlist (moretypes t)((i,argdot s,t)
::(singlesubslist s (argdot s) ((droplist(typedeps t)
(droplist (map argdot (typedeps t))
(drop s(drop (argdot s)(addotlist M (incrementlist((i,a,T)::L)))))))));

(*
```

This function adds dotted versions of implicit arguments that need to be added (using `moretypes`). It removes additional subsequent dotted or undotted copies of the arguments added as well as copies of anything that the new type added depends on.

```
*)
```

```
(* this replaces a list of items with their dotted versions
throughout an argument/sort list *)

fun dotsubslist ((s,t)::L)  nil = nil |

dotsubslist nil L = L |

dotsubslist ((s,t)::L) ((i,a,T)::M) = dotsubslist L
((i, argsubs s (argdot s) a, typesubs s (argdot s) T)::(dotsubslist ((s,t)::L) M));


(* expand the list of arguments presented for a function
at declaration time
with inferred implicit arguments  *)

(*
```

Replace a list of items with their dotted versions throughout an argument list. Probably a stray utility to be moved near the earlier list of such utilities.

```
*)


fun expandlist nil = nil |

expandlist ((i,EntArg(Ent(s,0)),t)::L) =  addotlist(moretypes t)
(dotsubslist((moretypes t))(droplist(map pi1 (moretypes t))((i,EntArg(Ent(s,0)),t)::
(singlesubslist (EntArg(Ent(dot s,0))) (EntArg(Ent(s,0)))
(drop (EntArg(Ent(s,0))) (drop (EntArg(Ent(dot s,0)))
(droplist(typedeps t) (droplist(map argdot (typedeps t))(expandlist L)))))))))

|

expandlist ((i,AbstArg((s,0)),t)::L) =  addotlist(moretypes t)
(dotsubslist((moretypes t))
(droplist(map pi1 (moretypes t))((i,AbstArg((s,0)),t)::
(singlesubslist (AbstArg((dot s,0))) (AbstArg((s,0)))
(drop (AbstArg((s,0))) (drop (AbstArg((dot s,0)))
(droplist(typedeps t)
(droplist(map argdot (typedeps t))(expandlist L)))))))))

|
```

```
expandlist ((i,a,t)::L) =  addotlist(moretypes t)
(dotsubslist((moretypes t))(droplist(map pi1 (moretypes t))((i,a,t)::
((* drop (EntArg(Ent(s,0))) *) (droplist(typedeps t)
(droplist(map argdot (typedeps t))(expandlist L)))))))
;

fun guardedexpandlist L = if (!IMPLICITVER) then (expandlist L) else L;

(*
```

Here are the functions which generate full argument lists of declared functions from the arguments explicitly given. The overall strategy is best described by quoting a comment above:

"take the first element in the argument list ('the item'); compute the expanded alternate list of the tail of the argument list ('the previous list'); Compute the deps of the item and drop all dotted and undotted versions of the deps from the previous list. Then add the item. Then add dotted versions of the moretypes list of the item, and replace undotted versions of these variables with dotted versions throughout."

```
*)

(* functions to repair an argument list,
adding values for implicit arguments in the sort *)

fun firstundotted nil = EType error |

firstundotted ((i,EntArg(Ent(s,n)),t)::L) =

if s<> "" andalso hd(explode s) = #"." then firstundotted L

else t |

firstundotted ((i,AbstArg((s,n)),t)::L) =

if s<> "" andalso hd(explode s) = #"." then firstundotted L

else t |

firstundotted ((i,a,t)::L) = t;

(*
```

`firstundotted` should go with other dot manipulations. The latter function actually returns the sort of the first undotted argument.

```
*)

fun initialsegment nil L = true |

initialsegment (x::L) (y::M) = if x<>y then false else initialsegment L M |

initialsegment L M = false;


fun matchsegment nil L = nil |

matchsegment (x::L) (y::M) =

   (y::(matchsegment L M)) |

matchsegment L M = M;

(*
```

`initialsegment` tests whether its first argument is an initial segment of its second argument.

`matchsegment` returns as long an initial segment as possible of a second list which is the same length as an initial segment of the first list.

```
*)

(* This function finds the value of a given implicit argument
by sort matching.  Since it is actually looking deep into the structure
of ordinary terms embedded in sorts, including ordinary terms with
variable binding, it is doing a lot of weird type casting, sometimes
quite incompatible with Lestrade's own type system :-)

In principle, this search can fail, if an argument is implicitly sorted
using a subterm of a sort which can actually be eliminated by a definitional
expansion.  I have not seen this happen.  It would not represent a failure
of the logic:  the implicit argument feature actually touches the logic
not at all.

It can fail quite easily for an implicit function argument.

*)
```

```
(*
```

arguments of the monster function `findimplicitarg`:

`Types1`: This is the list of bound identifiers and sorts in the term containing the implicit argument.

`Types`: This is the list of sorts of locally bound identifiers in the term from which we are trying to recover a value for the implicit argument by matching.

`a`: This is the implicit argument one is trying to evaluate.

`atype`: This is the sort of the implicit argument one is trying to evaluate.

```
*)

fun findimplicitarg Types1 Types a atype
(EType(that(App(s,n,(x::L))))) (EType(that(App(t,m,(y::M))))) =

let val VV18 = if a = AbstArg(s,n)
andalso initialsegment (x::L) (map (fn (x1,y1,z1) => y1) Types1) then
silenttypematchcomp0 (deworld2 atype)
(matchsegment (x::L) Types) else error in

if a = AbstArg(s,n)
andalso initialsegment (x::L) (map (fn (x1,y1,z1) => y1) Types1)
andalso VV18 <> error then

if (y::M) = (map (fn (x1,y1,z1) => y1) (matchsegment (x::L) Types))
     then AbstArg(t,m) else

Lambda((World((matchsegment (x::L) Types)@[(0,EntArg(App(t,m,(y::M))),
EType (VV18 ))])))

else if a = AbstArg(s,n) (* andalso
entitytype (App(t,m,y::M)) <> error *) andalso
equalenttypes (silenttypematchcomp0 (deworld2 atype)
(map (fn xx => (1,xx,argtype xx)) (y::M)))
(entitytype (App(t,m,y::M)))  then AbstArg(t,m)

(*
```

This (minus the last paragraph) is a clever gadget for evaluating function implicit arguments as anonymous function terms. Where the argument list of a term $f(x_1, \ldots, x_n)$ is an initial segment of the locally bound variables, f is the implicit argument we are trying to evaluate, and T is the term we are matching, we can identify f as $(\lambda x_1, \ldots, x_n.T)$, or if T is of the form $g(x_1, \ldots, x_n)$ we may identify f as g.

It would be nice to be able to do this when the argument list was an arbitrary sublist of the locally bound variables (subject to issues about dependencies). But the restriction to initial segments is powerfully simplifying.

The last paragraph implements the condition that one can also identify f with g if the term T is of the form $g(u_1, \ldots, u_n)$, containing no bound variables, and has the correct types.

```
*)

else let val TTT = (App(t,m,(y::M)))  and UU = if a = AbstArg(s,n) then silenttypematchcomp(
   (multisubstypelist (x::L) (deworld2 atype)) else error in

if a = AbstArg(s,n)

    andalso UU <> error

   then (Lambda(renamespace(World((multisubstypelist (x::L) (deworld2 atype))@
[(0,EntArg(multisubs (x::L) (deworld2 atype) TTT),
EType(UU))])))

(*
```

This is a partial implementation of the idea that we can match $f(u_1, \ldots, u_n)$ with a term T, where f is the variable to be identified and the u_i's contain no bound variables, with a deduced (really guessed) function of the u_i's constructed by a reverse substitution process (replace the u_i's with bound variables and build a λ-term). This is where the functions `multisubs` and `multisubstypelist` find their only use.

The problem with `multisubstypelist` destroying dependent types has been fixed. As it happens, `multisubstypelist` is a rather trivial function and much of what goes on here is now wheel-spinning. I'm going to leave it alone for now to avoid typos.

```
*)


else

if s<>t orelse m<>n then

let val T = expand (App(s,n,(x::L))) and U = expand (App(t,m,(y::M))) in

if T <> (App(s,n,(x::L)))
```

```
   then

   let val V = findimplicitarg Types1 Types a atype
   (EType(that(T))) (EType(that(App(t,m,(y::M)))))

   in if V <> EntArg Error

     then V

     else if U = (App(t,m,(y::M)))  then EntArg Error

         else findimplicitarg Types1 Types a atype
         (EType(that(App(s,n,(x::L))))) (EType(that(U))) end
else if U = (App(t,m,(y::M)))  then EntArg Error

         else findimplicitarg Types1 Types a atype
         (EType(that(App(s,n,(x::L))))) (EType(that( U))) end

(*
```

Here Lestrade attempts to deduce the implicit argument by definitional expansion. Rewriting is not used.

```
*)

else if a = x then y

else let val T = findimplicitarg Types1 Types a atype (arg2type x) (arg2type y) in

if T <> EntArg Error then T

else findimplicitarg Types1 Types a atype
 (EType(that(App(s,n,L)))) (EType(that(App(t,m,M)))) end end end

(*
```

If the first argument of the term being matched is the implicit argument to be evaluated, we can set its value to the first argument of the term it is being matched to.

Otherwise we can attempt implicit argument matching of the types of the respective first arguments.

Otherwise we can match the terms with the first arguments dropped.

```
*)

|

findimplicitarg Types1 Types a atype (EType(that(App(s,n,(L))))) (EType(that(T))) =

let val VV19=if a = AbstArg(s,n)

   andalso initialsegment L (map (fn (x1,y1,z1) => y1) Types1) then silenttypematchcomp0 (de
           (matchsegment L Types) else error in

if a = AbstArg(s,n)

   andalso initialsegment L (map (fn (x1,y1,z1) => y1) Types1)

       andalso VV19 <> error

       then (Lambda(World((matchsegment L Types)@
       [(0,EntArg T, EType (VV19))])))

else let val VV17 = if a = AbstArg(s,n) then silenttypematchcomp0 (deworld2 atype)
   ((multisubstypelist L ((deworld2 atype)))) else error in


if a = AbstArg(s,n)

    andalso VV17 <> error

   then (

Lambda(renamespace(World((multisubstypelist L (deworld2 atype))@
[(0,EntArg(multisubs L (deworld2 atype) T),
EType(VV17))])))

   else if expand (App(s,n,L)) <> App(s,n,L) then

findimplicitarg Types1 Types a atype
(EType(that(expand(App(s,n,(L)))))) (EType(that(T)))

 else if expand T <> T then

findimplicitarg Types1 Types a atype
(EType(that(App(s,n,L)))) (EType(that(expand(T))))

else EntArg Error end end
```

```
(*
```

The various approaches taken here are closely analogous to the approaches taken in the previous clause, except that T has no application structure to appeal to. It should be noted that T is presumably an atomic constant or variable.

```
*)


|

findimplicitarg Types1 Types a atype
(EType(IN(App(s,n,(L))))) (EType(IN(T))) =
findimplicitarg Types1 Types a atype
(EType(that(App(s,n,(L))))) (EType(that(T)))

(*
```

Sorts built with `in` instead of `that` are handled in the same way.

```
*)


 |

findimplicitarg Types1 Types a atype (EType(that b)) (EType(that c)) =
if a=EntArg b then EntArg c else EntArg Error

(*
```

Where we are matching an atomic term to a term, the only way we can succeed is if the atomic term is the implicit argument we are trying to find a value for.

```
*)

|

findimplicitarg Types1 Types a atype (EType(IN b)) (EType(IN c)) =
if a=EntArg b then EntArg c else EntArg Error

(*
```

Where we are matching an atomic term to a term, the only way we can succeed is if the atomic term is the implicit argument we are trying to find a value for.

```
*)


|

findimplicitarg Types1 Types a atype
     (AType(World([(i,b,t)]))) (AType(World([(j,c,u)]))) =

let val T = findimplicitarg Types1 Types a atype t u in

if T = EntArg Error then findimplicitarg Types1 Types a atype
   (EType (that (deent b))) (EType (that (deent c)))
else T end

(*
```

Here we are matching the tail of a λ-term or function sort term: we attempt to find the implicit argument by matching sorts, and then by matching definition bodies.

```
*)

|

findimplicitarg Types1 Types a atype
     (AType(World((i,b,t)::L))) (AType(World((j,c,u)::M))) =

let val T = findimplicitarg Types1 Types
 a atype t u in if T = EntArg Error
then findimplicitarg  (Types1@[(i,b,t)]) (Types@[(j,c,u)])
a atype (AType (World L)) (AType (World M)) else T end

(*
```

Here we are matching a term in which a variable is bound. We first attempt to recover the implicit argument by matching the types of the variables; we then attempt to recover the implicit argument by matching the rest of the term, adding the two bound variables matched and their sorts to `Types1` and `Types`.

```
*)

|

findimplicitarg Types1 Types a atype x y = EntArg Error;

(*
```

This monstrously involved function uses matching in various ways to try to divine correct values for implicit arguments in concretely given instances of functions declared with implicit arguments. The most reassuring thing is that the entire implicit argument feature touches the logic not at all; if implicit argument inference fails, one can always fix things by supplying the argument which the system failed to deduce. However, implicit argument inference is enormously useful, especially when the value of a function argument can be deduced.

This function needs to be revisited and decorated with local comments on its cases. (I have done this, and when I did it I found the function much more intelligible).

```
*)

(* this function repairs the argument list supplied to a function with
implicit arguments at parse time.  It is important to notice that implicit arguments
play no role in the logic at all! *)

fun fixarglist nil x = nil |

fixarglist x nil = nil |

fixarglist ((i,EntArg(Ent(s,n)),t)::L)  ((T)::M) =

if s = "" orelse hd(explode s) <> #"."
     then (T)::(fixarglist (singlesubslist (EntArg(Ent(s,n))) T L) M)

else

let val MM = (findimplicitarg nil nil (EntArg(Ent(s,n)))
(typesubs (EntArg Unknown) (EntArg Unknown) t) (firstundotted L) (argtype T)) in

(MM::(fixarglist (singlesubslist (EntArg(Ent(s,n))) MM L)((T)::M))) end |
```

```
fixarglist ((i,AbstArg((s,n)),t)::L)  ((T)::M) =

if s="" orelse hd(explode s) <> #"."
     then (T)::(fixarglist (singlesubslist (AbstArg(s,n)) T L) M)

else

let val MM = findimplicitarg nil nil ((AbstArg(s,n)))
(typesubs (EntArg Unknown) (EntArg Unknown) t)  (firstundotted L) (argtype T) in

(MM::(fixarglist (singlesubslist (AbstArg(s,n)) MM L) ((T)::M))) end |

fixarglist (x::L) (y::M) = y::(fixarglist L M);

fun guardedfixarglist L M = if (!IMPLICITVER) then fixarglist L M else M;

(*
```

This function uses `findimplicitarg` to expand explicitly given argument lists to full argument lists expected by declared functions. The customer of this function is *the parser*, not any core function of the logic. It is useful to note that this will work correctly on an initial segment of an explictly argument list (in the case of "curried function arguments".)

```
*)

(* The following block of functions is used in the quite elaborate
check of the structure of inputs to the rewritep and rewrited commands *)

fun lasttwo (x::y::z::L) = lasttwo (y::z::L) |

lasttwo L = L;

fun allbutlasttwo nil = nil |

allbutlasttwo [x] = nil |

allbutlasttwo [x,y] = nil |

allbutlasttwo (x::L) = x::(allbutlasttwo L);

(* check correctness of argument lists for rewriting commands *)

(* a good rewrite list has at least two elements in it *)
```

```
fun goodrewritelist nil = false |

goodrewritelist [x] = false |

goodrewritelist L =

   let val [Q,R] = lasttwo(L) in let val T = argtype Q in

   (* Q is an object argument and not polymorphic *)

   notvararg Q andalso typerigid(deent Q) andalso

   (* the last two items have the same sort *)

   equaltypes false T (argtype R)

     (*  andalso

   (* the first of the last three items
    is a predicate (or type constructor) variable over this sort *)

   isvariable P andalso

   (equaltypes true (argtype P)
   (AType(World([(~1,EntArg(Ent("???",1)),T),
   (~1,EntArg(Unknown),EType prop)])))

(* restoring symmetry between prop and type -- commented out, could be restored *)

   (* orelse equaltypes true (argtype P)
   (AType(World([(~1,EntArg(Ent("???",1)),T),
   (~1,EntArg(Unknown),EType TYPE)]))) *) )


   andalso

   (* P does not appear in the deps of Q *)

   not (inlist P (depsarg [Q])) *)

   (* everything in allbutlasttwo L appears in the deps of Q *)

   andalso allinlist (allbutlasttwo L) (depsarg [Q])

   (* everything in the deps of R appears in Q *)
```

```
   andalso allinlist (depsarg [R]) (depsarg [Q]) end end;

(*
```

This function checks the structure of argument lists of the `rewritep` and `rewrited` commands. The embedded comments are somewhat outdated. Originally, the last three arguments were a predicate of objects of a certain type, then the pattern and target of the rewrite rule, two terms of that object type. The predicate is no longer present (it is automatically generated by the declaration commands). The pattern needs to be a type rigid object term. Every preceding variable must appear in the dependencies of the pattern. The dependencies of the target must be a subset of the dependencies of the pattern.

```
*)

(* user command:  close the last move opened;
 this sends an error message if it attempts to close world 1 *)

(* Close does not automatically save the next move:  that has to be done with Save() *)

(*  drafting a way to deal with the problem of hypothetical rewrites:  since
dependencies on hypothetical rewrites are not recorded, if we close a world
in which there are such rewrites, we must discard whatever might depend
on them, which is everything (or everything after them?) *)

(* USER COMMAND *)

fun Close() = (if length(!CONTEXT) > 2 then
(clearallcaches();NAMESERIAL:=(!Maxfreshindex);
CONTEXT:= (tl (!CONTEXT));REWRITES:=tl(!REWRITES);
(*  if hd(!REWRITES) <> nil then (say "Eliminating dependencies on hypothetical rewrites";C
WORLDNAMES:= tl(!WORLDNAMES))
else saypause ("Cannot undo move 1:"^(hd (!WORLDNAMES))));

(*
```

This is the user command `close` which simply closes the next move. One cannot close world 1, and in this case an error message will be issued.

The serial counter for namespaces, `NAMESERIAL`, is set back to `Maxfreshindex`, which is maintained as an upper bound on namespace indices in stored declarations.

```
*)

fun savelist worldnames context =

if length context <= 1 orelse length worldnames <= 1 then nil
   else (worldnames,hd context)::(savelist (tl worldnames) (tl context));

(* save all moves on path to next move *)
(* it is not possible to save a move which has its default numerical name *)

(* USER COMMAND *)

fun Save s = (* if s = makestring(length(!CONTEXT)-1)
then saypause "Cannot save a move with the default numeral name"

else *)

(* if defaultworld (tl(!WORLDNAMES))
then saypause "Cannot save a default move" else *)

(if s <> hd(!WORLDNAMES) then
(SAVEDWORLDS:= abstractdrop2 (s::(tl(!WORLDNAMES))) (!SAVEDWORLDS);
SAVEDREWRITES:= abstractdrop2 (s::(tl(!WORLDNAMES))) (!SAVEDREWRITES)) else ();
SAVEDWORLDS := abstractmerge
(savelist (s::(tl(!WORLDNAMES))) (!CONTEXT))(!SAVEDWORLDS);
SAVEDREWRITES := abstractmerge
(savelist (s::(tl(!WORLDNAMES))) (!REWRITES))(!SAVEDREWRITES);
WORLDNAMES:=s::(tl(!WORLDNAMES)));

(*
```

This is the user command `save` which saves a move with a string argument as name. A move cannot be saved with its default numeral name, nor can it be saved if the last move has its default numeral name, unless the last move is move 0.

When a move is "saved over", that move is replaced and all moves extending it are eliminated.

```
*)


(* USER COMMAND *)

fun ClearCurrent s = (* if defaultworld (tl(!WORLDNAMES))
```

```
 andalso not(defaultworld(s::(tl(!WORLDNAMES))))

then saypause "Named move cannot follow a default move" else *)

(clearallcaches();NAMESERIAL:=(!Maxfreshindex);
let val W = abstractfind (s::(tl(!WORLDNAMES))) (!SAVEDWORLDS)
and R = abstractfind (s::(tl(!WORLDNAMES))) (!SAVEDREWRITES)
in

if W = nil orelse s = makestring(length(!WORLDNAMES)-1) then

(say0 "clearing current next move";CONTEXT:=(World nil)::(tl (!CONTEXT)); REWRITES:=nil
::(tl(!REWRITES));WORLDNAMES:=s::(tl(!WORLDNAMES));
SAVEDWORLDS:= abstractdrop2 (!WORLDNAMES) (!SAVEDWORLDS);
SAVEDREWRITES:= abstractdrop2 (!WORLDNAMES) (!SAVEDREWRITES))

 else let val WW = makeadjoinable(hd W) (if R = nil then nil else hd R) (tl(!CONTEXT)) in

if (!BREAKOUT) then (saypause "Clearcurrent command fails due to name conflicts")

else (say0 "replacing current move with saved current move";CONTEXT:=(WW)::(tl (!CONTEXT));
REWRITES:=(hd R)::(tl (!REWRITES));
WORLDNAMES:=s::(tl(!WORLDNAMES)))  end  end);

(*
```

This is the user command `clearcurrent`. Without an argument, it will discard all contents of the next move, leaving the next move empty (not closing it). This is useful because otherwise declarations in world 1 could not be cleared. If `clearcurrent` is called with an argument, it will load the saved move with that name if there is an appropriate one and the argument is not the default numerical name of the move; it will in any case use the argument as the name of the next move. When the argument is the default name and there is a saved move with the default name, that move and all moves extending it are eliminated.

The case where the command fails due to name conflicts is I believe now not possible, due to the use of `makeadjoinable`.

The serial counter for namespaces, `NAMESERIAL`, is set back to `Maxfreshindex`, which is maintained as an upper bound on namespace indices in stored declarations.

```
*)
```

```
(*
```

This is the serial number for a fresh declaration line.

```
*)

(* completely clear the Lestrade context, user command, also
issued by readfile *)

(* USER COMMAND *)

fun ClearAll() = (clearallcaches();GREETED:=false;CONTEXT:=[World nil,World nil];
REWRITES:=[nil,nil];SERIAL:=0;NAMESERIAL:=0;Maxfreshindex:=0;WORLDNAMES:=["1","0"];SAVEDWORI

(*
```

This user command clears all declarations and sets all indices to initial values.

```
*)

(* load a named theory.
This completely clears the context, supplying the move 0
declarations of the saved theory --
without rewrite decs, perhaps I should fix this.  *)

(* USER COMMAND *)

fun LoadTheory s = let val S = abstractfind s (!SAVEDTHEORIES) in

if s="" orelse S = nil then

saypause
("No such theory to load:\n"^s
^".lti must be read before this file can be read")

else let val (S1,S2,S3,S4,S5,S6,S7,S8) = hd S in

(clearallcaches();GREETED:=false;CONTEXT:=S1;
REWRITES:=S2;SERIAL:=S3;NAMESERIAL:=S4;Maxfreshindex:=S5;WORLDNAMES:=S6;SAVEDWORLDS:=S7;SAVE

end end;
```

```
(*
```

This user command clears the context entirely then completely restores the saved theory named by the argument with all details (in older versions, it loaded move 0 of the named theory; this is no longer the case; it loads the exact context with which the theory was saved).

```
*)

fun max x y = if x>= y then x else y;

fun ImportTheory s =

let val S0 = (abstractfind s (!SAVEDTHEORIES)) in

if s="" orelse S0 = nil then
saypause ("No such theory to import:\n"^s
^".lti must be read before this file can be read")

else

let val (S1,S2,S3,S4,S5,S6,S7,S8) = hd S0 in

let val WW = makeadjoinable(hd(rev S1)) (hd (rev S2)) [hd(rev(!CONTEXT))] in

(* if !BREAKOUT then saypause "Import fails due to name conflicts" else *)

(* if S2 <> nil then saypause "Theories with rewrites cannot be imported." else *)

(SERIAL:=max (!SERIAL) (S3);
NAMESERIAL:=max (!NAMESERIAL) (S4);
Maxfreshindex:=max(!Maxfreshindex)(S5);
SAVEDWORLDS := ([s,"0"],WW)::(abstractdrop2 [s,"0"] (!SAVEDWORLDS));
SAVEDREWRITES := ([s,"0"],nil)::(abstractdrop2 [s,"0"] (!SAVEDREWRITES))) end end end;



(*
```

This user command imports move 0 of a named theory as a new move 1 with the theory name as its name (applying `makeadjoinable` to avert name conflicts). Rewrites are restored; this has hazards.

```
*)

(* sort check an object sort.
This checks that P in that P is a prop and T in in T is a type *)

fun typecheck obj = true |

typecheck prop = true |

typecheck TYPE = true |

typecheck (that P) =

let val ANSWER = (entitytype P = prop) in
(if not ANSWER then
saypause ((display2 P)^" is not of sort prop (typecheck)")else();
Flush();ANSWER) end|

typecheck (IN P) =

let val ANSWER = (entitytype P = TYPE) in
(if not ANSWER then saypause
((display2 P)^" is not of sort 'type' (typecheck)")else();Flush();ANSWER) end|

typecheck error = false;

(*
```

This command sort checks an object sort. Is this really the right place for it to appear? Maybe it is: it has dynamic behavior really only appropriate in the context of user commands about to be presented.

```
*)


(* the object declaration command will take as arguments a string
s and an EntType  T.  It needs to check that s is not already declared,
then check that T declaration checks,
then add (EntArg(Ent(s)),T) as an entry to the first move in the context *)

(* actually it needs to do a full type check that T is of type prop *)

(* increment the declaration age counter *)
```

```
fun newserial() = (SERIAL:=1+(!SERIAL);(!SERIAL));

(* reserved identifiers. *)

fun reserved s = s="obj" orelse s="prop" orelse s="that"
orelse s="type" orelse s="in" orelse s = "---" orelse s = "+++" orelse s = "???" orelse s =

(*
```

`newserial` increments the number reserved for the next declaration line.
`reserved` tells us if an identifier is reserved by the system and so cannot be declared.

```
*)

(* command for postulating an object in the next move *)

(* USER COMMAND *)

fun Declare s (EType T) =

   if reserved s (* orelse extended s *) orelse stringdef s (!CONTEXT) <> nil
       then saypause ("Identifier "^s^" is not fresh")
   else if not (typecheck T) then saypause "Sort check fails"
   else (CONTEXT := (addtoworld0 (hd(!CONTEXT))(newserial(),
      EntArg(Ent (s,0)),EType T))::(tl(!CONTEXT));showdec s) |

Declare s (AType T) =     if reserved s (* orelse extended s *) orelse stringdef s (!CONTEXT
       then saypause ("Identifier "^s^" is not fresh")
   else (CONTEXT := (addtoworld0 (hd(!CONTEXT))(newserial(),
      AbstArg(s,0),AType T))::(tl(!CONTEXT));showdec s);

(*
```

This command allows the user to declare a variable of an object sort. 10/19/2017 The user may also declare a variable of a function sort. Function sort terms consist of a bracket followed by a list of variables whose types are taken from the next move followed by an object sort term. This short-circuits the move model a bit, in the same way that lambda term arguments do.

```
*)

(* the list of names of identifiers is in order of age;
```

```
this ensures sensible dependencies without evilly recursive
checks *)

(* modified to support the notion that definitions have
age 0 *)

(* fun isordered nil = true |

isordered [a] = true |

isordered (a::(b::L)) =

hd(Age a (hd(!CONTEXT))) <> 0 andalso

(hd(Age a (hd(!CONTEXT))) < hd(Age(b)(hd(!CONTEXT)))
   andalso isordered (b::L)); *)

(* there are some functions here for interaction with
      the implicit arguments feature *)

fun dotfix nil t = t |

dotfix ((i,a,t)::L) t2 = if (!IMPLICITVER)
andalso argundot a <> a
    then etypesubs (argundot a) a (dotfix L t2) else dotfix L t2;

fun dotfix2 nil t = t |

dotfix2 ((i,a,t)::L) t2 = if (!IMPLICITVER)
andalso argundot a <> a
    then entsubs (argundot a) a (dotfix2 L t2) else dotfix2 L t2;

(* remove dotted items from a list; used by the parser *)

fun dotpurge nil = nil |

dotpurge ((i,EntArg(Ent(s,n)),t)::L) = if (!IMPLICITVER) then

   if s <> "" andalso hd(explode s) = #"." then dotpurge L
   else ((i,EntArg(Ent(s,n)),t)::(dotpurge L))
   else ((i,EntArg(Ent(s,n)),t)::L) |

dotpurge((i,AbstArg(s,n),t)::L) = if (!IMPLICITVER) then

   if s <> "" andalso hd(explode s) = #"." then dotpurge L
   else ((i,AbstArg(s,n),t)::(dotpurge L)) else
```

```
  ((i,AbstArg(s,n),t)::L) |

dotpurge (x::L) = x::(dotpurge L);

(*
```

I'm not sure why these dot manipulation tools are in this location.

```
*)

(* toolkit for turning argument lists into moves -- the
sort of a function is a little move *)

fun worlditem s = (hd(Age s (hd(!CONTEXT))),s,
typesubs (EntArg Unknown) (EntArg Unknown)
(hd(Find s (hd(!CONTEXT)))));

fun worldof L = World(guardedexpandlist(map worlditem L));

(*
```

Turn an argument list into a move (adding implicit arguments as needed).

```
*)

(* the next move is temporarily replaced
during the declaration process for functions -- this is a place to keep it *)

val SAVECONTEXT = ref (hd(!CONTEXT));

val SAVECONTEXT2 = ref (hd(!CONTEXT));

(*
```

It is convenient during a declaration to cut down the next move into just the arguments being used in that declaration. These are places to keep different versions of the next move.

```
*)

(* in postulate, the user supplies as an argument list
```

```
all notions in the next move on which the construction
depends, in order of construction.
We do a dynamic maneuver:  replace the next move with the
part of the next move indicated by the argument list;
declaration check this move using the resulting context
(checking that it includes all its own needed dependencies)
and sort check the object sort argument in this context; then restore the context. *)

(* command for postulating a construction in the last move *)
(* s is a name.  L could also be a list of names.  T is an object sort. *)
(* USER COMMAND *)

fun postulate s L T =

if hd(!REWRITES) <> nil then saypause "postulate command blocked by hypothetical rewrites" e

if not(testall isvariable L) then saypause "Some argument is not variable"

(* testing for order is a cute way to enforce sensible dependencies;
the implicit arguments feature now does the laborious checks for this, but
if it is turned off this condition handles deps just fine *)

else if not (isordered L) then saypause "Arguments are in the wrong order"

else let val T = dotfix (deworld(worldof L)) T in

(

if reserved s (* orelse extended s *) orelse stringtype s <> nil
then saypause ("Identifier "^s^" is not fresh")

else if L = nil then
    let val TT = etypesubs (EntArg Unknown) (EntArg Unknown) T in

    (SAVECONTEXT:=hd(!CONTEXT);
    CONTEXT:=(World nil)::tl(!CONTEXT);if not (typecheck TT)

    then (saypause "Sort check fails in declaration of constant";
    CONTEXT:=(!SAVECONTEXT)::(!CONTEXT))

else (CONTEXT := (!SAVECONTEXT)::
(addtoworld0 (hd(tl(!CONTEXT)))(newserial(),EntArg(Ent (s,0)),EType TT))
::(tl(tl(!CONTEXT)));showdec s)) end

else if not
```

```
let val TT =  (etypesubs (EntArg Unknown) (EntArg Unknown) T) in

(
SAVECONTEXT:=(hd(!CONTEXT));

CONTEXT:= (worldof (*worldof2*) L)::(tl(!CONTEXT));

let val CHECK = deccheck4 (!CONTEXT) (hd(!CONTEXT))
andalso typecheck (TT) in (CONTEXT:=(!SAVECONTEXT)::(tl(!CONTEXT));CHECK)

end
) end then saypause "Dependency or sort check failure"

else (

let val newparentcontext = addtoworld0 (hd(tl(!CONTEXT)))
(newserial(),AbstArg(s,0),(Reset();
Reindex3(AType(renamespace(addtoworld0 (worldof L)
(0,EntArg Unknown,typesubs (EntArg Unknown)  (EntArg Unknown) (EType T)))))) in

(CONTEXT:= (hd(!CONTEXT))::newparentcontext::(tl(tl(!CONTEXT)));showdec s) end)

) end;

fun ddefine s L T =

if hd(!REWRITES) <> nil then saypause "deferred definition command blocked by hypothetical r

if not(testall isvariable L) then saypause "Some argument is not variable"

(* testing for order is a cute way to enforce sensible dependencies;
the implicit arguments feature now does the laborious checks for this, but
if it is turned off this condition handles deps just fine *)

else if not (isordered L) then saypause "Arguments are in the wrong order"

else let val T = dotfix (deworld(worldof L)) T in

(

if reserved s (* orelse extended s *) orelse stringtype s <> nil
then saypause ("Identifier "^s^" is not fresh")

(* else if L = nil then
    let val TT = etypesubs (EntArg Unknown) (EntArg Unknown) T in
```

```
    (SAVECONTEXT:=hd(!CONTEXT);
    CONTEXT:=(World nil)::tl(!CONTEXT);if not (typecheck TT)

    then (saypause "Sort check fails in declaration of constant";
    CONTEXT:=(!SAVECONTEXT)::(!CONTEXT))

else (CONTEXT := (!SAVECONTEXT)::
(addtoworld0 (hd(tl(!CONTEXT)))(newserial(),EntArg(Ent (s,0)),EType TT))
::(tl(tl(!CONTEXT)));showdec s)) end *)

else if not

let val TT =  (etypesubs (EntArg Unknown) (EntArg Unknown) T) in

(
SAVECONTEXT:=(hd(!CONTEXT));

CONTEXT:= (worldof (*worldof2*) L)::(tl(!CONTEXT));

let val CHECK = deccheck4 (!CONTEXT) (hd(!CONTEXT))
andalso typecheck (TT) in (CONTEXT:=(!SAVECONTEXT)::(tl(!CONTEXT));CHECK)

end
) end then saypause "Dependency or sort check failure"

else (

let val newparentcontext = addtoworld0 (hd(tl(!CONTEXT)))
(0,AbstArg(s,0),(Reset();
Reindex3(AType(renamespace(addtoworld0 (worldof L)
(0,EntArg Deferred,typesubs (EntArg Unknown)  (EntArg Unknown) (EType T)))))) in

(CONTEXT:= (hd(!CONTEXT))::newparentcontext::(tl(tl(!CONTEXT)));showdec s) end)

) end;

(*
```

The user command `postulate` which declares primitive notions and axioms.

"in postulate, the user supplies as an argument list all notions in the next move on which the construction depends, in order of construction. We do a dynamic maneuver: replace the next move with the part of the next move indicated by the argument list; declaration check this move using the resulting context (checking that it includes all its own needed dependencies) and sort check the object sort argument in this context; then restore the context."

If there is no argument list, we are in effect declaring an object constant at the last move instead of the next move.

```
*)

(* USER COMMAND *)

fun Define s L T = (FULLREWRITES:=nil;

if hd(!REWRITES) <> nil then saypause "Define command blocked by hypothetical rewrites" else

if not(testall isvariable L) then saypause "Some argument is not variable"

(* same remark on the argument order test as above *)

else if not (isordered L) then saypause "Arguments are in the wrong order"

else

let val T0 = T and T = dotfix2 (deworld(worldof L)) T in

if reserved s orelse extended s orelse stringtype s <> nil
then saypause ("Identifier "^s^" is not fresh")

else if not let val T2 = ( (entsubs (EntArg Unknown) (EntArg Unknown) T))
and THETYPE = dotfix (deworld(worldof L))
(etypesubs (EntArg Unknown) (EntArg Unknown) (entitytype T0)) in (
SAVECONTEXT:=(hd(!CONTEXT));

CONTEXT:= (worldof (* worldof2 *) L)::(tl(!CONTEXT));

let val CHECK = deccheck4 (!CONTEXT) (hd(!CONTEXT)) andalso
deccheck1 (!CONTEXT) THETYPE andalso deccheck2 (!CONTEXT) T2

in (CONTEXT:=(!SAVECONTEXT)::(tl(!CONTEXT));CHECK)

end
) end then saypause "Sort check or dependency failure"
else let val TT = fullrewrite(entsubs (EntArg Unknown) (EntArg Unknown) (T))
 and  THETYPE = dotfix(deworld(worldof L))
(etypesubs (EntArg Unknown) (EntArg Unknown) (entitytype (T0)) ) in (
let val newparentcontext = addtoworld0 (hd(tl(!CONTEXT)))
(0,AbstArg(s,0),(Reset();Reindex3(
AType(renamespace(addtoworld0 (worldof L) (0,EntArg TT,EType (THETYPE))))))) in
```

```
(CONTEXT:= (hd(!CONTEXT))::newparentcontext::(tl(tl(!CONTEXT)));showdec s)

end)



end end);

(*
```

The user command `define` which declares defined notions. The body of the definition is rewritten aggressively using `fullrewrite` (the only place this function is used, so far); notice that the table of remembered rewrites is cleared when the command starts. The overall structure of the execution of this command is very similar to that of `postulate`: the arguments are different, in that the last one is an object term. There is no object sort argument (as there is in Automath), since the sort of the term can after all be computed from the term.

```
*)

(* postulate a function witnessing validity of a rewrite rule *)

(* USER COMMAND *)

fun rewritep s L =
(clearallcaches();
if not(!REWRITEVER)
then saypause "Rewriting is turned off" else

(* if length(!CONTEXT) <> 2 then saypause "Hypothetical rewrite declarations not supported"
 else *)

if hd(!REWRITES) <> nil then saypause "Rewrite construction blocked by hypothetical rewrites

if reserved s (* orelse extended s *) orelse stringtype s <> nil
then saypause ("Identifier "^s^" is not fresh")

else

let val V = extendenough (extend s) (!CONTEXT) in (*A*)

 if not (goodrewritelist L)
then saypause "Proposed rewrite list does not sort check"
```

```
else let val [Q,R] = lasttwo L and L1 = allbutlasttwo L in (*B*)

let val P = AbstArg(extendenough (extend V) (!CONTEXT),0) in (*C*)

(* set type of P to (AType(World([(~1,EntArg(Ent("???",1)),T),
   (~1,EntArg(Unknown),EType prop)]))  where T is argtype Q *)

( (*E*)

let val T = argtype Q in (*D*)

let val newvariablecontext = addtoworld0 (hd((!CONTEXT)))
(newserial(),P,(Reset();
Reindex3((AType(World([(~1,EntArg(Ent(
extendenough (extend (extendenough (extend V) (!CONTEXT))) (!CONTEXT)
,1)),T),
   (~1,EntArg(Unknown),EType prop)])))))) in (*Q*)

(CONTEXT:= newvariablecontext::(tl((!CONTEXT)));showdec (deabst P)) end (*Q*)end (*D*)
;

Declare (V) (EType(that (App(deabst P,0,[Q]))));
postulate s (L1 @ [P,EntArg(Ent(V,0))]) (that (App(deabst P,0,[R])));

let val T = stringtype s in (*F*) if T = nil
then saypause ("construction of "^s^" failed for some reason")

else let val (L1::L2::L3) = (!REWRITES) in(*G*)

REWRITES:= (L1::((s,(Negindex4((deent(hd(getpattern (deworld2(pi1(hd T)))))),
(Negindex4(deent(hd(gettarget (deworld2(pi1(hd T))))))))) :: L2)::L3)

end (*G*)

end (*F*)) (*E*)
end (*C*) end (*B*) end (*A*));

(* show that the function named by s
 witnesses the validity of proposed rewrite rule *)

(* USER COMMAND *)

val OLDCONTEXT = ref(!CONTEXT);
val OLDREWRITES = ref(!REWRITES);

 fun Rewrited s L  =
```

```
(clearallcaches();

if not(!REWRITEVER) then say "Rewriting is turned off" else

(* if length(!CONTEXT) <> 2 then saypause "Hypothetical rewrite declarations not supported"
 else *)

if hd(!REWRITES)<>nil then saypause "Rewrite definition blocked by hypothetical rewrites" el

if stringtype s = nil
then saypause ("Evidence function "^s^" is not declared")

else

let val S = extendenough (extend s) (!CONTEXT) in

(OLDCONTEXT:= (!CONTEXT);
rewritep S L;
if stringtype S <> nil andalso
equaltypes false (pi1(hd(stringtype s)))(pi1(hd(stringtype S)))
then say "Rewrite demonstration succeeded"
else (CONTEXT:=(!OLDCONTEXT);REWRITES:=(!OLDREWRITES);saypause "Rewrite demonstration failed
)


end);




(*
```

The `rewritep` and `rewrited` commands justify and create rewrite rules. The `rewritep` declares a new function of a correct sort to justify a rewrite rule and installs the rewrite rule; the `rewrited` command has as its first argument an already declared identifier of a correct sort to justify a rewrite rule, and introduces the desired rewrite rule. If a `rewrited` command is repeated, this moves the relevant rewrite rule into the position in the list where it will be applied first. In general, the most recently declared rewrite rule in the most recent move is applied. Rewrite rules associated with the next move are kept, but are not active.

The `rewrited` command needs to be tested.

In general terms, the rewrite feature arguably makes Lestrade a programming language.

```
*)

(* parser *)

(* this was originally Polish notation with a following comma to signal that
a function appears as an argument;
it was then upgraded to suppport use of
 functions of arity greater than 1
as infix or mixfix operators, and commas
are allowed between any arguments, and mandatory before and after
function identifier arguments
to avoid confusion with functions in applied or infix/mixfix position.

10/10 functions applied to shortened argument lists represent functions
(currying);
argument lists must be explicitly enclosed in parentheses for this to be
understood.

10/15 user entered lambda terms as arguments are supported.

 *)

fun islower c = #"a" <= c andalso c <= #"z";

fun isupper c = #"A" <= c andalso c <= #"Z";

fun isnumeral c = (#"0" <= c andalso c <= #"9") orelse c= #"'";

fun isspecial  c = c= #"~"

orelse c = #"@" orelse c = #"#" orelse c = #"$"
orelse c = #"%" orelse c = #"^" orelse c = #"&"
 orelse c = #"*" orelse c = #"-" orelse c = #"+"
 orelse c = #"=" orelse c = #"|" orelse c = #";" orelse c = #"." orelse c = #"<"
orelse c = #">" orelse c = #"?" orelse c = #"/"
 orelse c = #"!" orelse c = #".";

(*
```

Classes of character of interest in Lestrade. The single quote is counted as a numeral for technical reasons having to do with generating new alphanumeric identifiers.

```
*)

(* get first identifier from a list of characters *)

fun getident nil = nil |

getident (#"\"" :: L) = L |

getident [c] = if islower c orelse isupper c orelse isnumeral c
orelse isspecial c orelse c = #"," orelse c= #":"
orelse c = #"(" orelse c = #")" orelse c= #"[" orelse c= #"]" then [c] else nil |

(* I could fiddle with allowed shapes of identifiers here *)

getident (a::(b::L)) =

if a = #"," orelse a = #":" orelse a = #"(" orelse a = #")" orelse a = #"[" orelse a = #"]"

if a = #" " orelse a= #"\n" orelse a= #"\\" then getident (b::L)
     else if isupper a

         then if islower b orelse isnumeral b

         then a::(getident(b::L))

         else [a]

     else if islower a

         then if islower b orelse isnumeral b

         then a::(getident(b::L))

         else [a]

     else if isnumeral a

         then if isnumeral b

         then a::(getident(b::L))

         else [a]

     else if isspecial a

         then if isspecial b
```

```
          then a::(getident(b::L))

          else [a]

     else nil;

(* the rest of the stream of characters after the first identifier is read *)

fun restident nil = nil |

restident (#"\"" :: L) = nil |

restident [c] = nil |

restident (a::(b::L)) =

if a = #"," orelse a = #":" orelse a = #")" orelse a = #"(" orelse a= #"[" orelse a= #"]"  t

if a = #" " orelse a= #"\n" orelse a= #"\\" then restident (b::L)
     else if isupper a

          then if islower b orelse isnumeral b

          then restident(b::L)

          else (b::L)

     else if islower a

          then if islower b orelse isnumeral b

          then restident (b::L)

          else (b::L)

     else if isnumeral a

          then if isnumeral b

          then restident(b::L)

          else b::L

     else if isspecial a
```

```
          then if isspecial b

          then restident(b::L)

          else b::L

     else nil;

(* utility for tokenization *)

fun testidentlist nil = nil |

testidentlist (#">":: #">"::L) = ">> "::[implode L] |

testidentlist L = (implode(getident(L))::(testidentlist (restident L)));

(* get a list of tokens (identifiers and punctuation) from a string *)

fun tokenize s = testidentlist(explode s);

(*
```

The block of functions above comprises the tokenization feature of Lestrade. `getident` picks out the next identifier or punctuation mark from a stream of characters; `restident` returns the rest of the stream of characters after the first identifier or punctuation mark is read. Commas, colons, parentheses, and brackets are punctuation. Spaces, returns, and backslashes are discarded. An identifier is either a string of special characters or a string of nonempty length consisting of zero or one upper case character followed by zero or more lower case letters followed by zero or more numerals (including single quote as a numeral).

```
*)

(* repair an application term with missing arguments into a
lambda-term 10/10 mods *)

(* convert sort of a primitive construction into a lambda term *)

fun lambdaform s L = (* if pi23 (hd(rev L)) = EntArg Unknown
then *) rev((pi13(hd(rev L)),
EntArg(App(s,0,map pi23 (rev(tl(rev L)))),pi33(hd(rev L)))::(tl(rev L))) (* else L *);

(*
```

This function will convert the declaration list in the sort of a primitive construction into the declaration list in an internal representation of a λ-term for that construction. No such conversion is needed for defined notions.

```
*)

(* construct correct argument list for a curried function argument *)

fun FixListType final L T1 =

if T1 = nil then nil else

if length T1 = length L + 1 then [(pi13(hd (rev T1)),
if final then EntArg(defmatchcomp0 true T1 L)
else pi23(hd(rev T1)), typematchcomp1 T1 L)]

else let val LL = FixListType false L (rev(tl(rev T1))) in

LL @ (FixListType final (L@LL) T1) end;

(* transform an argument which has explicitly closed argument
list into a curried function if appropriate 10/10 mods *)

fun FixApp(EntArg(App(a,0,L))) =

let val T1 = stringtype a in

if length L >= length(deworld(getabstype(pi1(hd(T1)))))-1

then EntArg(App(a,0,L))

(* else if stringAge a <> [0] then
(saypause "Cannot curry a primitive construction";EntArg Error) *)

else Lambda (renamespace(World(FixListType true
(map (fn m =>(0,m,argtype m)) L)
(lambdaform a(deworld(getabstype(pi1(hd(T1)))))))) end |

FixApp t = t;

(*
```

These two functions compute the representation of the function represented by an application term with too few arguments (a curried function argument).

```
*)

(* get a non-infix term from a list of tokens *)

fun getterm nil = EntArg Error |

getterm (a::L) =

if reserved a then EntArg Error else

if a = "[" then let val BODY = getlambdalist L in

if BODY = nil then EntArg Error

else (Lambda (renamespace (World BODY))) end else

if a = "(" then let val TERM = getterms L and REST = restterms L in

if REST<>nil andalso hd REST = ")" then TERM else EntArg Error

end

else if a = "," then getterm L else

let val T1 = stringtype a in

if T1 = nil then EntArg Error

else if isenttype (pi1(hd T1)) then EntArg(Ent(a,0))

else if length(dotpurge(deworld(getabstype(pi1(hd(T1)))))) = 1

then EntArg(App(a,0,nil))

else if L = nil orelse hd L = "," orelse hd L = ":"
orelse hd L = ")" orelse reserved (hd L) then AbstArg(a,0)

else if length(dotpurge(deworld(getabstype(pi1(hd T1))))) = 2

then EntArg(App(a,0,
guardedfixarglist (deworld(getabstype(pi1(hd T1))))[getterm L]))

(* adding possibility of application terms with missing
arguments representing functions 10/10 mods -- argument
list enclosed by parentheses can be of variable length *)
```

```
else if hd L = "(" then let val TERM =
    guardedfixarglist (deworld(getabstype(pi1(hd T1))))
    (getopenarglist (tl L))
    and REST =
    restopenarglist (tl L)
    in  if REST <> nil andalso hd REST = ")"
    then FixApp(EntArg(App(a,0,TERM))) else EntArg Error end


else EntArg(App(a,0,guardedfixarglist (deworld(getabstype(pi1(hd T1))))
(getarglist (length(dotpurge(deworld(getabstype(pi1(hd(T1))))))-1)  (L))))

end

(* the rest of the list of tokens after the first non-infix term is read *)

and restterm nil = nil |

restterm (a::L) =

if reserved a then (a::L) else

if a = "[" then let val BODY = getlambdalist L in

if BODY = nil then (a::L)

else restlambdalist L end else

if a = "(" then let val REST = restterms L in

if REST <> nil andalso hd REST = ")" then tl REST else nil

end

else if a = "," then restterm L else

let val T1 = stringtype a in

if T1 = nil then a::L

else if isenttype (pi1(hd T1)) then L

else if length(dotpurge(deworld(getabstype(pi1(hd(T1)))))) = 1

then L
```

```
else if L = nil then nil

else if hd L = ","  then L

else if reserved(hd L) then L

else if length(dotpurge(deworld(getabstype(pi1(hd T1))))) = 2

then restterm L


else if hd L = "(" then let val REST =
    restarglist (length(dotpurge(
    deworld(getabstype(pi1(hd(T1))))))-1)  (tl L)
 in  if REST <> nil andalso hd REST = ")" then tl REST else nil end

else restarglist (length(dotpurge(deworld(getabstype(pi1(hd(T1))))))-1)  (L)

end

(*
```

`getterm` gets the first non-infix term from a stream of tokens; `restterm` returns the rest of the stream of tokens.

The alternatives are a λ-term (starting with a bracket), a parenthesized term, an object or function atomic constant, or an application term (an atomic constant applied to an argument list) or a curried function term (an atomic constant applied to "too few arguments").

Notice that the parser calls `guardedfixarglist` to attempt to fill in implicit arguments.

```
*)

(* get an infix term from a stream of tokens *)

and getterms L =

if L = nil orelse reserved(hd L) then EntArg Error else

let val TERM = getterm L and REST = restterm L in

if REST = nil orelse hd REST = "," orelse hd REST = ")"
orelse hd REST = "(" orelse hd REST = ":" orelse reserved (hd REST) then TERM
```

```
else let val T1 = stringtype (hd REST)

in if T1=nil orelse isenttype (pi1(hd T1)) then TERM

else if length(dotpurge(deworld(getabstype(pi1(hd(T1)))))) <=2 then TERM

else if tl REST = nil orelse hd(tl REST) = ","
orelse hd(tl REST) = ":" orelse hd(tl REST) = ")"
orelse reserved(hd(tl(REST))) then TERM

else EntArg(App(hd REST,0,guardedfixarglist
(deworld(getabstype(pi1(hd T1))))((TERM)::
(getarglist
(length(dotpurge(deworld(getabstype(pi1(hd(T1)))))-2)  (tl REST)))))

end end

(* the rest of the stream of tokens after the first infix term is read *)

and restterms L =

if L = nil orelse reserved(hd L) then L else

let val REST = restterm L in

if REST = nil orelse hd REST = "," orelse hd REST = ")"
orelse hd REST = "(" orelse hd REST = ":" orelse reserved (hd REST) then REST

else let val T1 = stringtype (hd REST)

in if T1=nil orelse isenttype (pi1(hd T1)) then REST

else if length(dotpurge(deworld(getabstype((pi1(hd(T1))))))) <=2 then REST

else if tl REST = nil orelse hd(tl REST) = ","
orelse hd(tl REST) = ":" orelse hd(tl REST) = ")"
orelse reserved(hd(tl REST)) then REST


else restarglist (length(dotpurge(deworld(getabstype(pi1(hd(T1)))))-2)  (tl REST)

end end

(*
```

The function `getterms` reads the longest possible infix term from a stream of tokens, and `restterms` returns what is left of the stream. There are similar observations to be made about the responsibility of the parser to fill in implicit arguments.

```
*)

(* a list of arguments of known length,
 without enclosing parentheses (these are handled by getterm *)

and getarglist 0 L = nil |

getarglist n nil = [EntArg Error] |

getarglist n L = (getterms L)::(getarglist (n-1) (restterms L))

(* what is left after reading a list of arguments
 of known length without enclosing parentheses *)

and restarglist 0 L = L |

restarglist n nil = nil |

restarglist n L = restarglist (n-1) (restterms L)

(* get a list of arguments of unknown length
(as in an function declaration or (10/10) a prefix term with
explicit argument list) *)

and getopenarglist nil = nil |

getopenarglist ((":")::L) = nil |

getopenarglist ((")")::L) = nil |


getopenarglist L =

if restterms L = L then nil else

(getterms L)::(getopenarglist(restterms L))

(* what is left after reading a list of arguments of unknown length *)

and restopenarglist nil = nil |
```

```
restopenarglist ((":")::L)  = L |

restopenarglist ((")")::L)  = (")"::L) |


restopenarglist L =

if restterm L = L then L else
restopenarglist(restterms L)

(*
```

These functions read argument lists from a stream of tokens (and companion functions return the rest of the stream of tokens). `getarglist` reads argument lists of known length (where the arity of a function is known); `getopenarglist` reads argument lists of unknown length, as in function declarations or curried function argument expressions.

```
*)

and guardedgetterms L = if despace1(restterms L) = nil
(* orelse hd(restterms L) = "]"
andalso despace1(tl(restterms L)) = nil *) then getterms L
else (saypause "Term not completely read";EntArg Error)

(*
```

This guarded version of `getterms` will return an error term if it does not exhaust the stream of tokens. This is used to detect dangling extra arguments in command lines.

```
*)


and getlambdalist L =

if L = nil then nil

else let val T1 = stringdef (hd L)[(hd(!CONTEXT))] in

if T1 = nil orelse stringage(hd L)[hd(!CONTEXT)] = [0] then nil
```

```
else if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = ","
then (1,EntArg(Ent(hd L,0)),pi1(hd T1))::(getlambdalist (tl(tl L)))

else if (tl L) <> nil andalso hd(tl(L)) = ","
then (1,AbstArg(hd L,0),pi1(hd T1))::(getlambdalist (tl(tl L)))

else let val TERM = getterms ((tl(tl L)))
   and REST = restterms ((tl(tl L))) in

if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = "=>" andalso (tl(tl L)) <> nil
andalso TERM <> EntArg Error andalso
REST <> nil andalso
hd(REST) = "]"
then  [(1,EntArg(Ent(hd L,0)),pi1(hd T1)),
(1,TERM,argtype (TERM))]

else if  (tl L) <> nil andalso hd(tl L) = "=>"
andalso (tl(tl L)) <> nil
andalso TERM <> EntArg Error andalso
REST <> nil andalso
hd(REST) = "]"
then [(1,AbstArg(hd L,0),pi1(hd T1)),
(1,TERM,argtype (TERM))]

else nil

end end

and restlambdalist L =

if L = nil then L

else let val T1 = stringdef (hd L)[(hd(!CONTEXT))] in

if T1 = nil orelse stringage(hd L)[hd(!CONTEXT)] = [0] then L

else if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = "," then restlambdalist (tl(tl L))

else if (tl L) <> nil andalso hd(tl(L)) = "," then restlambdalist (tl(tl L))

else let val TERM = getterms ((tl(tl L)))
   and REST = restterms ((tl(tl L))) in
```

```
if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = "=>" andalso (tl(tl L)) <> nil
andalso TERM <> EntArg Error andalso
REST <> nil andalso
hd(REST) = "]" then  tl(REST)

else if  (tl L) <> nil andalso hd(tl L) = "=>"
andalso (tl(tl L)) <> nil andalso getterms ((tl(tl L))) <> EntArg Error andalso
REST <> nil andalso
hd(REST) = "]" then tl(REST)

else L

end end

(*
```

These functions parse the innards of λ-terms.

```
*)

;

fun readenttype nil = error |

readenttype (a::L) = if a = "obj" then obj

else if a="prop" then prop

else if a="that" then let val P = deent(getterms L) in

if P = Error then error else

that P end

else if a="type" then TYPE

else if a="in" then let val P = deent(getterms L )in

if P = Error then error else

IN P end else error;

fun restenttype nil = nil |
```

```
restenttype (a::L) = if a = "obj" then L

else if a="prop" then L

else if a="that" then let val P = deent(getterms L) in

if P = Error then (a::L) else

restterms L end

else if a="type" then L

else if a="in" then let val P = deent(getterms L) in

if P = Error then (a::L) else

restterms L end else (a::L);

fun getlambdalist2 L =

if L = nil then nil

else let val T1 = stringdef (hd L)[(hd(!CONTEXT))] in

if T1 = nil orelse stringage(hd L)[hd(!CONTEXT)] = [0] then nil

else if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = ","
then (1,EntArg(Ent(hd L,0)),pi1(hd T1))::(getlambdalist2 (tl(tl L)))

else if (tl L) <> nil andalso hd(tl(L)) = ","
then (1,AbstArg(hd L,0),pi1(hd T1))::(getlambdalist2 (tl(tl L)))

else if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = "=>" andalso (tl(tl L)) <> nil
andalso readenttype ((tl(tl L))) <> error andalso
typecheck(readenttype (tl(tl L))) andalso
restenttype (tl(tl L)) <> nil andalso
hd(restenttype ((tl(tl L)))) = "]"
then  [(1,EntArg(Ent(hd L,0)),pi1(hd T1)),
(1,EntArg Unknown,EType(readenttype(tl(tl L))))]

else if  (tl L) <> nil andalso hd(tl L) = "=>"
andalso (tl(tl L)) <> nil
andalso readenttype ((tl(tl L))) <> error andalso
```

```
typecheck(readenttype (tl(tl L))) andalso
restenttype (tl(tl L)) <> nil andalso
hd(restenttype ((tl(tl L)))) = "]"
then [(1,AbstArg(hd L,0),pi1(hd T1)),
(1,EntArg Unknown,EType(readenttype (tl(tl L))))]

else nil

end

and restlambdalist2 L =

if L = nil then L

else let val T1 = stringdef (hd L)[(hd(!CONTEXT))] in

if T1 = nil orelse stringage(hd L)[hd(!CONTEXT)] = [0] then L

else if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = "," then restlambdalist2 (tl(tl L))

else if (tl L) <> nil andalso hd(tl(L)) = "," then restlambdalist2 (tl(tl L))

else if isenttype(pi1(hd T1)) andalso (tl L) <> nil
andalso hd(tl L) = "=>" andalso (tl(tl L)) <> nil
andalso readenttype ((tl(tl L))) <> error andalso
restenttype((tl(tl L))) <> nil andalso
typecheck(readenttype (tl(tl L))) andalso
hd(restenttype ((tl(tl L)))) = "]" then  tl(restenttype ((tl(tl L))))

else if  (tl L) <> nil andalso hd(tl L) = "=>"
andalso (tl(tl L)) <> nil andalso readenttype ((tl(tl L))) <> error andalso
typecheck(readenttype (tl(tl L))) andalso
restenttype((tl(tl L))) <> nil andalso
hd(restenttype ((tl(tl L)))) = "]" then tl(restenttype(tl(tl L)))

else L

end;

fun readabstype L =

if L<>nil andalso hd L = "[" then
let val LL = getlambdalist2 (tl L) in
if LL=nil then EType error else (AType (renamespace(World (LL)))) end
else EType error;
```

```
fun restabstype L = if L<>nil andalso hd L = "[" then restlambdalist2 (tl L) else L;

fun readtype L = if despace1(resttype L) <> nil then (saypause "Term not completely read"; E

else let val ET = readenttype L in if ET = error

then readabstype L

else EType (ET) end

and resttype L = if readenttype L = error

then restabstype L

else restenttype L;


(*
```

`readenttype` parses object sorts (with its partner `restenttype` keeping track of tails of lists.

`getlambdalist2` reads the innards of function sort terms (with the aid of a partner function keeping track of tails).

`readabstype` parses function sorts, and `readtype` parses general types, each with a partner function.

Function sort terms consist of a bracket followed by a comma separated list of variables from the next move followed by `=>` followed by an object sort term followed by a close bracket.

The new device for parsing λ-terms has now been adapted to parse function sorts; I have extended the `declare` command to allow declaration of function variables. This does obscure some philosophical points (as λ-term arguments do as well).

We have now arrived at the end of the parser source. It is useful to note that the display language is different from the parsed language: anonymous function terms and function sort terms have their bound variables without type labels (with types read from the next move) and the body of a function sort term is simply an object sort term. We do not intend to support typing of bound variables by explicit labelling (nor do we intend to support parsing of subscripted variables), and we do not intend to implement any sort of type inference.

```
*)
```

```
(* the command line just read *)

val THELINE = ref "";
val THELINE2 = ref "";

(* the file from which commands are being read, used by readfile
in indented and unindented versions *)

val READFILE = ref (TextIO.openIn("default"));

val LOGNAME = ref "";

val LOGNAME2 = ref "";

fun Hd nil = "" |

Hd x = hd x;

fun Tl nil = nil |

Tl x = tl x;

(*
```

Here are some items useful for the command line reading commands below.
`THELINE` is the command line just read. `THELINE2` no doubt has a related use.
`READFILE` is a file from which command lines are being read.
`LOGNAME(2)` is the name of some log file.
`Hd` and `Tl` are guarded head and tail command for use with lists of strings.

```
*)

(* test functions -- two of them are user commands *)

fun sarg s = getterms(tokenize s);

fun sent s = deent(sarg s);

fun stype s = readtype(tokenize s);

fun sType s = (stype s);

fun slist s = getopenarglist (tokenize s);
```

```
fun slist2 n s = getarglist n(tokenize s);

(* USER COMMAND *)

fun Sent s = (say (display2(sent s)); say (display1(entitytype(sent s))));

(* USER COMMAND *)

fun Stype s = (say (display6(stype s)));

fun Moretypes s = say (display5(World(map (fn (x,y) => (1,x,y))
(moretypes (argtype(getterm (tokenize s)))))));

fun Expandlist s = say(display5(World(expandlist (map (fn x => (1,x,argtype x))
(getopenarglist(tokenize s))))));

fun Fixarglist s t = say(display5(World(map (fn x => (1,x,argtype x))(fixarglist

(deworld(getabstype(pi1(hd(stringtype s)))))

((getopenarglist (tokenize t)))

))));

(*
```

Functions used for diagnostics during debugging which I will not comment on.

```
*)

val READFILEDEPTH = ref 0;

(* read a command line from a stream of tokens *)

val THEORYNAME = ref "bogus";

val BACKUPINDEX = ref 0;

fun readline nil = () |

readline (a::L) =
```

```
if a = "setmarginup" then MARGIN := (!INDENTWIDTH)+(!MARGIN)

else if a = "setmargindown" then if (!MARGIN)>(!INDENTWIDTH) then MARGIN:=(!MARGIN)-(!INDENT

(*

    increase or decrease the margin by five.


*)

else if a = "readfile" andalso length(L)>=2
then (Flush();closelog();READFILEDEPTH:=1+(!READFILEDEPTH);readfile (hd L) (hd(tl L)))

else if a = "readfile" andalso length(L) = 1
then (Flush();closelog();READFILEDEPTH:=1+(!READFILEDEPTH);readfile (hd L) "scratch")

else if a = "readback" andalso length(L) = 1
then (Flush();closelog();READFILEDEPTH:=1+(!READFILEDEPTH);readfile (hd L) ("backups\\"^(hd

else if a = "readbook" andalso length(L)>=2
then (Flush();closelog();READFILEDEPTH:=1+(!READFILEDEPTH);readfile2 (hd L) (hd(tl L)))

else if a = "readbook" andalso length(L) = 1
then (Flush();closelog();READFILEDEPTH:=1+(!READFILEDEPTH);readfile2 (hd L) "scratchtex")

else if a = "readkoob" andalso length(L) = 1
then (Flush();closelog();READFILEDEPTH:=1+(!READFILEDEPTH);readfile2 (hd L) ("backups\\"^(hd


(*

    The commands readfile and readbook can be invoked in the interface, but
should not be invoked in log files.
    readback and readkoob are new 10/16/17, read the default file back into
the named file.
    readbook and readkoob replace readfile2 and readback2 in older versions.


*)

else if a = "parsetest" andalso L<>nil then Sent (hd L)

else if a = "parsetest2" andalso L<> nil then Stype(hd L)
```

```
(*
```

These commands will parse an argument (starting with a double quote) as an object term or as an object type, respectively.

```
*)

else if a = "declare" andalso L<> nil andalso tl L<>nil
then let val s = (hd L) and t = readtype (tl L) in

(if resttype (tl L)<> nil andalso resttype (tl L)<> [""]
then saypause ("Declaration line not completely read:  "^(hd L)) else ();
say2(!THELINE);
say0("Your command: "^(!THELINE2)); Declare s t) end

else if a = "goal" andalso L<> nil
then let val t = readtype L in

(if resttype (L)<> nil andalso resttype (L)<> [""]
then saypause ("Declaration line not completely read") else ();
say2(!THELINE);
say0("Your command: "^(!THELINE2)); say1("Goal: "^(display6 t)^"\n")) end

else if a = "test" andalso L<> nil
then let val t = (guardedgetterms L) in

(if restterms (L)<> nil andalso restterms (L)<> [""]
then saypause ("Declaration line not completely read") else ();
say2(!THELINE);
say0("Your command: "^(!THELINE2)); say1("Test: "^(display4 t)^"\n");say1(display6(argtype t
```

```
(*
```

The `declare` user command: declare an identifier (first argument) with a given object sort (second argument) at the next move.

The `goal` command parses an entity type and displays it. Its intended index use is actually to make notes of goals in structured proofs.

The `test` command parses an argument term and displays it with its type. Since the `goal` command gives me this ability for entity types, I thought I would like to have it for terms as well. Note that `test` will handle anything which occurs as an argument, which includes certain terms representing constructions.

Like `goal`, its main use in log files is to generate comments. It is very handy in the interface for building the next command!

```
*)

else if a = "postulate" andalso L<>nil andalso tl L <> nil
then let val s = hd L and L1 = getopenarglist (tl L)
and T = readenttype (restopenarglist(tl L))

in (if restenttype (restopenarglist(tl L))<> nil
andalso restenttype (restopenarglist(tl L))<> [""]
then saypause ("construction line not completely read:  "
^(hd (restenttype (restopenarglist(tl L))))) else ();

say2(!THELINE);
say0("Your command: "^(!THELINE2));postulate s L1 T) end

else if a = "ddefine" andalso L<>nil andalso tl L <> nil
then let val s = hd L and L1 = getopenarglist (tl L)
and T = readenttype (restopenarglist(tl L))

in (if restenttype (restopenarglist(tl L))<> nil
andalso restenttype (restopenarglist(tl L))<> [""]
then saypause ("construction line not completely read:  "
^(hd (restenttype (restopenarglist(tl L))))) else ();

say2(!THELINE);
say0("Your command: "^(!THELINE2));ddefine s L1 T) end

(*
```

The `postulate` user command: declare an identifier (first argument) applied to a list of arguments (subsequent arguments) with output a given object sort: the resulting function (or object if the argument list is null) is declared at the last move.

```
*)

else if a = "define" andalso L<>nil andalso tl L<>nil
then let val s = hd L and L1 = getopenarglist (tl L)
and T = deent(guardedgetterms(restopenarglist(tl L)))

in if T = Unknown then
saypause "Sorry, cannot define something as a function" else
```

```
(if restterms(restopenarglist(tl L)) <> nil
andalso restterms(restopenarglist(tl L)) <> [""]
then saypause ("Definition line not completely read:  "^(hd(restopenarglist(tl L))))else ();
say2(!THELINE);say0("Your command: "^(!THELINE));Define s L1 T) end

(*
```

The `define` user command: declare an identifier (first argument) applied to a list of arguments (subsequent arguments) with output a given object term (the definition body): the resulting function (or object if the argument list is null) is declared at the last move.

```
*)

else if a = "rewritep" andalso L <> nil andalso tl L <> nil
then let val s = hd L and L1 = getopenarglist (tl L)
(* and V = Hd(restopenarglist(tl L)) *) in

(if (Tl(restopenarglist(tl L))) <> nil
andalso (restopenarglist(tl L)) <> [""]
then saypause ("Rewrite construction line not completely read:  "
^(hd(restterms(restopenarglist(tl L)))))else ();say2(!THELINE);
say0("Your command: "^(!THELINE));rewritep s L1 (* V *) ) end

else if a = "rewrited" andalso L <> nil andalso tl L <> nil
then let val s = hd L and L1 = getopenarglist (tl L)
(* and V = Hd(restopenarglist(tl L)) *) in

(if (Tl(restopenarglist(tl L))) <> nil
andalso (restopenarglist(tl L)) <> [""]
then saypause ("Rewrite demonstration line not completely read:  "
 ^(hd(restopenarglist(tl L))))else ();say2(!THELINE);
say0("Your command: "^(!THELINE));Rewrited s L1 (* V *)) end

(*
```

The `rewritep` and `rewrited` user commands: postulate or exhibit as already defined a function which justifies a rewrite rule, and record the rewrite rule.

```
*)

else if a = "open" then (say2(!THELINE);
```

```
say0 ("Your command:  "^(!THELINE)^"\n");
Open(if L=nil orelse hd L = "" then makestring(length(!CONTEXT)) else hd L))

(*
```

`open` opens a new move: it will have the default numeral name if it has no argument.

```
*)

else if a = "close" then (say2(!THELINE)
;say0("Your command: "^(!THELINE)^"\n");Close())

(*
```

`close` closes the next move, unless it is move 1.

```
*)

else if a = "save" then (say2(!THELINE)
;say0 ("Your command:  "^(!THELINE)^"\n"); Save(if L=nil orelse hd L = "" then
hd(!WORLDNAMES) else hd L))

(*
```

`save` saves the next move with the name given as argument or the default numeral name if no argument is given.

```
*)

else if a = "load" then (say2(!THELINE)
;say0 ("Your command:  "^(!THELINE)^"\n"); LoadTheory(if L=nil orelse hd L = "" then
"" else hd L))

else if a = "import" then (say2(!THELINE)
;say0 ("Your command:  "^(!THELINE)^"\n"); ImportTheory(if L=nil orelse hd L = "" then
"" else hd L))

(*
```

These commands restore saved theories (move 0 declarations plus some indices. `load` clears the environment and makes the saved theory the entire context; `import` makes move 0 of the saved theory a new move 1.

```
*)

else if a = "versiondate" then versiondate()

(*
```

Report the current version.

```
*)

else if a = "showall" then showall()

(*
```

Show all declarations. The output will be huge.

```
*)

else if a = "showimplicit" then (showimplicit();say2(!THELINE))

else if a = "hideimplicit" then (hideimplicit();say2(!THELINE))

else if a = "typesonly" then (typesonly();say2(!THELINE))

else if a = "showdefs" then (showdefs();say2(!THELINE))

(*
```

Show or hide implicit arguments. Show or hide definition bodies.

```
*)
```

```
else if a = "displayrewrites" then displayrewrites()

(*

    Show all active rewrite rules.

*)

else if a = "showrecent" then showrecent()

(*

    Display the declarations in the next move and the last move.

*)

else if a = "showdec" andalso L<>nil then (showdec (hd L))

(*

    Display the declaration of a single identifier, the argument.

*)

else if a = "showdecs" then showdecs()

else if a ="compactdisplay" then compactdisplay()

else if a = "supercompactdisplay" then supercompactdisplay()

(*
```

Show all declarations in the next move and last move, one by one, waiting until the user hits enter (or `q` to break out).

`compactdisplay` turns on or off display of definition bodies at positive moves.

`supercompactdisplay` turns on or off display of any type information feedback from a command line.

```
*)

else if a = "foropen"  then say ("\n\n"^(savedforopen()))

else if a = "forclearcurrent"  then say ("\n\n"^(savedforclearcurrent()))

(*
```

Display the names of saved environments which could be opened with the indicated command.

```
*)

else if a="comment" orelse a="%"
then (TextIO.output(!LOGFILE,!THELINE^"\n");say0((!THELINE^"\n")))

else if a="comment1" orelse a="%%"
then (TextIO.output(!LOGFILE,!THELINE);say0((!THELINE)))

else if a =">> " then ()

(*
```

Comments. `comment` or `%` is the last line of a comment (followed by a return).
`comment1` or `%%` is a non-last line of a comment.
The two flavors of comment line above persist when the log file is run: the `>>` comment is transitory.

```
*)

else if a = "clearcurrent" then
(ClearCurrent(if L = nil orelse hd L = "" then makestring(length(!CONTEXT)-1)
     else hd L);
TextIO.output(!LOGFILE,!THELINE^"\n"))

(*
```

Clear the next move and name it with the string argument (loading a saved environment of that name if it is present and the argument is not the default nuemral name of the next move). This command is the only way to clear move 1 declarations.

```
*)

else if a = "clearall" then (ClearAll();TextIO.output(!LOGFILE,!THELINE);showall())

(*
```

Clear the Lestrade environment completely.

```
*)

else if a = "clearbreakout" then BREAKOUT:=false

(*
```

Clear an error condition. Use this judiciously, it is not a safe move.

```
*)

else if a = "basic" then (basic();say2(!THELINE))

else if a = "explicit" then (explicit();say2(!THELINE))

else if a = "fullversion" then (fullversion();say2(!THELINE))

else if a="clearallcaches" then clearallcaches()

(*
```

Version toggles, not used.

```
*)

else if a = "pause" then
(say ("Pausing in "^(!LOGNAME)^":\n>>  type lines or type quit to resume");
TextIO.output(!LOGFILE,!THELINE);interface " ")

(*
```

Pause and wait for a user entered return to resume. Useful for looking at what is happening at particular points in scripts.

```
*)

else if a = "" then () else saypause "Line is not a Lestrade command"

(* purge indentation from command lines *)

(* and  despace0 (#" "::L) = despace0 L |

despace0 L = L

and despace s = implode(despace0(explode s)) *)

and unindent0 (#" "::L) = unindent0 L |

unindent0 (#"." :: #"." :: #"." :: #"." :: #" " ::L) = unindent0 L |

unindent0 L = L

and unindent s = implode(unindent0(explode s))

(*
```

The `readline` command executes a string of tokens as a Lestrade command. More comments on commands and their format may be wanted.

```
*)

(* read a command line from a string *)

and Readline s = (THELINE2:=(unindent s)^"\n";
THELINE:=s^"\n";readline(tokenize (unindent s)))

(*
```

`Readline` tokenizes a string and calls `readline`. It also records the line for logging.

```
*)


(* read command lines from standard output and receive feedback;
 output is logged to a file, end with quit *)
```

```
and interface filename =
(if filename = "" orelse filename = " "
then () else LOGFILE:=TextIO.openOut((filename^".lti"));
 (if not(!GREETED) then (versiondate();GREETED:=true) else ();
let val LINE = Inputline(TextIO.stdIn) in

if LINE = "quit\n" then (if filename <> " "
then TextIO.output(!LOGFILE,"quit") else ();
TextIO.flushOut(!LOGFILE);
if filename <> " "
then (TextIO.flushOut(!LOGFILE);closelog())
 else TextIO.flushOut(!LOGFILE);say "Bye!")

else (if implode(rev(tl(rev(explode LINE)))) = "" then
TextIO.output(TextIO.stdOut,"The Inspector awaits your instructions:  ")
     else ();Flush();
Readline (implode(rev(tl(rev(explode LINE)))));interface "") end))

(* read commands from a first log file after clearing the Lestrade context,
 logging to a second log file, and ending in the interface
where you can continue to enter commands logged to the second file.  End with quit *)

and readfile filename1 filename2 =

if filename1 <> "" andalso not (fileexists filename1)
then saypause ("The book "^filename1^" does not exist.")

else if filename1 = "scratchtex" orelse filename2 = "scratchtex"
     then saypause "Probably wrong readfile command!"

else(

(if filename1 <> "" then BREAKOUT:=false else ()
;if filename1 <> "" then THEORYNAME := filename1 else ();
if filename1 = "" then ()
else (ClearAll();READFILE:=TextIO.openIn((filename1^".lti")));
if filename2 = "" then ()
else (ClearAll();LOGNAME:=filename1;LOGNAME2:=filename2;
LOGFILE:=TextIO.openOut((filename2^".lti")));
(if not(!GREETED) then (versiondate();GREETED:=true) else ();
let val LINE = getline(!READFILE) in

if LINE = "quit\n" orelse (!BREAKOUT) then (BREAKOUT:=false;
TextIO.closeIn(!READFILE);
say("Done reading "^(!LOGNAME)^" to "^(!LOGNAME2)^":\n>>"
```

```
^"  type lines or type quit to exit interface\n\nquit\n");
NAMESERIAL:=(!Maxfreshindex);
SAVEDTHEORIES:=
(!THEORYNAME,(!CONTEXT,!REWRITES,!SERIAL,!NAMESERIAL,!Maxfreshindex,!WORLDNAMES,!SAVEDWORLDS
::(abstractdrop filename2 (!SAVEDTHEORIES)); if (!READFILEDEPTH)=0 then
interface "" else READFILEDEPTH :=(!READFILEDEPTH)-1)

(* fun ClearAll() = (clearallcaches();GREETED:=false;CONTEXT:=[World nil,World nil];
REWRITES:=[nil,nil];SERIAL:=0;NAMESERIAL:=0;Maxfreshindex:=0;WORLDNAMES:=["1","0"];SAVEDWORI

else (Readline (implode(rev(tl(rev(explode LINE)))));readfile "" "") end)))

and getline(targetfile) =  let val PRELINE = Inputline(targetfile)

in

if length(explode PRELINE) <2 orelse not(hd(tl(rev(explode PRELINE))) = #"\\")

then PRELINE

else PRELINE^(getline(targetfile))

end

and readfile2 filename1 filename2 =

if filename1 <> "" andalso not (fileexists2 filename1)
then saypause ("The book "^filename1^" does not exist.")

else if filename1 = "scratch" orelse filename2 = "scratch"
     then say "Probably wrong readfile command!"

else(

(if filename1 <> "" then BREAKOUT:=false else ()
;if filename1 <> "" then THEORYNAME := filename1 else ();
if filename1 = "" then ()
else (ClearAll();READING:=false;READFILE:=TextIO.openIn((filename1^".tex")));
if filename2 = "" then ()
else (ClearAll();LOGNAME:=filename1;LOGNAME2:=filename2;
LOGFILE:=TextIO.openOut(filename2^".tex");
GREETED:=true);
let val LINE =

getline(!READFILE)
```

```
in

if (not(!READING) andalso LINE = "quit\n") orelse (!BREAKOUT) then (BREAKOUT:=false;
TextIO.closeIn(!READFILE);
say0("Done reading "^(!LOGNAME)^" to "^(!LOGNAME2)^":\n>>"
^"  type lines or type quit to exit interface\n\nquit\n");
TextIO.output(!LOGFILE, "quit\n"); TextIO.flushOut(!LOGFILE);
NAMESERIAL:=(!Maxfreshindex);
SAVEDTHEORIES:=
(!THEORYNAME,(!CONTEXT,!REWRITES,!SERIAL,!NAMESERIAL,!Maxfreshindex,!WORLDNAMES,!SAVEDWORLDS
::(abstractdrop filename2 (!SAVEDTHEORIES)); if (!READFILEDEPTH)=0 then
interface "" else READFILEDEPTH :=(!READFILEDEPTH)-1)

 else (if LINE=("\\"^"end{verbatim}\n") then
    (READING:=false; TextIO.output(!LOGFILE,LINE);say0(LINE))
     else if (!READING)
         then Readline (implode(rev(tl(rev(explode LINE)))))
         else if (* LINE="\\begin{verbatim}\n" orelse  LINE="\\begin{verbatim} %Lestrade\n"

            then (READING:=true;TextIO.output(!LOGFILE,"\\begin{verbatim}Lestrade execution
            else (TextIO.output(!LOGFILE,LINE);say0(LINE));
     readfile2 "" "") end));

(* else (Readline (implode(rev(tl(rev(explode LINE)))));readfile "" "") end)); *)

fun fixargtest s t n = (deworld(getabstype (pi1(hd(stringtype s)))),
guardedfixarglist (deworld(getabstype (pi1(hd(stringtype s)))))
(getarglist n (tokenize t)));

(*
```

`interface` reads commands from the command line and gives feedback to standard output and to a log file.

`readfile` reads commands from a Lestrade log file and gives feedback to standard output and another Lestrade log file (by default `scratch.lti`). The extension of Lestrade log files is `.lti`.

`readfile2` is as `readfile` but works with `.tex` files as log files. The default output is `scratchtex.tex`. Note that either log file clears the environment (a script is not run after another script). `readfile(2)` can be run sensibly in `interface` but not in log files.

Running either kind of log file sets the theory name to the name of the source file.

```
*)
```

```
(* disaster cleanup -- close the files if you crash out of the interface *)

fun Cleanup() = (TextIO.closeOut(!LOGFILE); TextIO.closeIn(!READFILE));

fun senttype s = entitytype(sent s);

fun typetest1 s = display6(Cleantype1(pi1(hd(stringtype s))));
fun typetest2 s = display6(pi1(hd(stringtype s)));

(*
```