My Own Private Automath: the Lestrade dependent typed system and the specification and code for its implementation, the Lestrade Type Inspector

M. Randall Holmes

1/5/2022 installation of rewrites in progress
— This is a beta version of the second release, without rewriting. It runs the entire Zermelo implementation. It also ran the Automath translations without any changes. Adapted for PolyML.

Contents

0.1 Version Notes

1/6/2022: Repairing a problem with parenthesizing infix terms to the left of infixes.

1/5/2022: PolyML runs the Zermelo interpretation much faster than Moscow ML, so I am contemplating using it.

This file has a preamble which will probably support modification of the code to run in several versions of ML.

An issue which occurred in Linux, not handled in this copy of the file, is that one needs to replace "LTXTs
" with "LTXTs/": this is probably also best handled in a preamble.

This is still the flagship file, because I ought to make the rewrites work for me. This is more encouraging now that the new version looks more usable.

An issue only handled in this exact copy of the file (which should be implemented in others as well) is that PolyML insists on closing instreams, and I have Readtex and Readontex closing instreams so that I can edit .tex files in the way I am used to.

Eliminating automatic elision of leading zeroes from identifiers, because it does not work correctly; it also eliminates medial zeroes, as written.

10/28/2021: Another change to consider. Can we have parameters which are object sorts? Can we have two species of parameters, one for object sorts, one for sorts in general?

5/25/2021: I have now installed rewrites as an object type, parsable and displayable. Next I have to actually make them do something.

The additional object type construction of rewrite rules present, and all the usual effects should work. What is needed in addition is the application of rewrites – in which functions do we expect these to operate? We expect rewrite rules to be actively applied in the output of definition commands (this is an implementation of computation). In addition, we expect to be able to use rewriting to establish equality of types, to strengthen the type system.

5/21/2020: I seem to have settled on an approach. For objects t, u of the same type, there is an object type `Rewrite`(t, u): an object of this

type is a rewrite rule rewriting t to u. A general rewrite rule is a function $[x_1 \ldots x_n \Rightarrow \texttt{Rewrite}(t(x_1 \ldots x_n), u(x_1 \ldots x_n))]$, representing a rule rewriting terms matching $t(x_1 \ldots x_n)$ to the corresponding terms $u(x_1 \ldots x_n)$.

5/14/2020: Some revisions made already before branching; correct treatment of errors in format doesn't have any effect on performance but implements what is clearly actually intended. Minor edits to text.

This branch is devoted to adding rewriting. My intention is not necessarily to duplicate the rewrite feature of the first implementation, though I'm open to it.

I see three possible approaches:

1. implement something like the approach in the first version, where rewrite rules are bound to functions. I would like to make sure that dependences are managed correctly so that environments with rewrites can be closed while preserving the results proved and objects declared. There are very clever ideas realized in the first implementation about this to be cloned.

2. implement something just like the Watson rewrite language. There is considerable charm to this idea from a personal standpoint if it can be made to work. It can be simulated under approach 1.

3. implement inductive types with definition expansion. The idea is that one can keep declaring constructors in a declared type until one declares something involving functions with that type as domain. Ths is theoretically safe and has the merit that computations will be terminating, but it does not give the same freedom of implementation of computations...or is there an argument against this?

I have the idea that a rewrite is a peculiar sort of object, $t \mapsto u$ where t and u are terms of the same type τ (speaking here of approach 1 or 2; approach 3 has quite simple metaphysics). A rewrite rule is a function with rewrite outputs. Declaration of a rewrite rule (or a simple rewrite) then has side effects: if its left side is a pattern, the rewrite rule is bound to it (when multiple rewrite rules are bound to the same function, the most recently declared one is applied first). Computation of dependencies must consult the rewrite list

There are important issues with patterns: they have to exactly force the types of their inputs.

Matching of a pattern to a term will exclude matching of its non-atomic subterms to terms that match patterns: this cleverly enforces Curry-Howard.

New idea which would implement something like 2: this addresses the problem of dependencies of rewrites. The idea is that a rewrite is an equality theorem appearing in the next move, which has to be supplied as an argument to a definition using it. One then defines the behavior of the rewrite rule application operator (or possibly as in Watson a suite of rewrite rule application operators). Rewrite rules can be introduced as variables or as let terms (based on actual proved theorems). The fact that they must appear as arguments ensures that dependences are not lost. Can recursively chained rewrites be achieved?

New idea: rewrites are arguments in a defintion (they might be let arguments).

4/15/2020: List updates which ought to be made but which are not high priority.

1. Figure out how to display functions in curried format where possible and as far as possible. This might not be hard. This goes with a fugitive thought that pairs might be treated as a native Lestrade sort – but a function sort, presented in curried form.
2. Set up the close command so that it deletes non-default extensions of default moves when they are closed.
3. It might be a good idea to be alerted to changes in order of arguments. These have caused interesting confusions in porting files and recently in development of new material.
4. Is there a way to reduce the suppression of let terms in arguments?
5. The import command is a nice feature. Additionally, it would be good to have the ability to save theory states to files which can be rebuilt directly without checking. Backups as in the first version to prevent accidental destruction of scripts is a good thing.
6. In the last few days I have updated the big code examples to the new version, and they do seem to work. I have inserted new text

in foundationsintroport.tex which illustrates how type checking which relies on rewriting can be implemented without rewriting, at least in principle.

4/14/2020: Improved Showmoves command. Refined save command so that when it overwrites a saved move it will eliminate extensions of that saved move. These things are motivated by rewriting a Lestrade manual for the previous version to accord with this one.

4/13/2020: Some tweaks to unsubs functions which may or may not be needed. I was having a problem in a legacy file which I thought was a problem with the implicit argument mechanism, but it wasn't: it was actually an issue with the order in which implicit arguments were declared. The old version puts implicit arguments in a different order; this version puts them in the literal order in which they are declared, and this can turn out to be an issue! A change in declaration order and the offending file worked correctly.

4/12/2020: I have realized why there are so few let terms even after minimizing obvious expansions: there is heavy expansion going on in arguments, which I might be able to suppress. Actually, it doesn't seem so easy. The idea for preserving more let terms would be to allow object arguments to have let binders.

Fixed error which allowed colons : to be special characters.

4/10/2020: No code update. I should attend to updating all the existing text not using rewrites for testing and demonstration purposes.

4/9/2020: Fixed a bug in the refinement of equalfunctions. Added a toggle `Zeroorone` which revises typesimplify so that it reduces let terms with one (the default) or no occurrences of the bound variable (when strong expansion is off).

4/8/2020: Put in a shortcut for computation of equality between functions with let term arguments. It may in theory cause false negatives, but it doesn't seem likely; if this does turn out to be a danger, this step could be toggled on or off? Basically, if two functions being compared have binders starting with the same let term, drop the let term (changing names of the bound object to bring the names into line in the two terms).

4/7/2020: Updating betared and typered without expansion of let terms.

The fix to make betared and typered not energetically expand let terms makes the function recursive again...

I am beginning to think that, while the let term enhancement is philosophically valuable, the strong expansion approach is the practical one. I should still look at trying to fix equality and matching to be less energetic about expanding let terms.

4/6/2020: Starting a read and comment pass.

I added the `test` command, which here displays a function and so in fact any kind of term. I added application of `typesimplify` to function sorts if strong expansion is not on.

Test beta reduction without expansion of let terms.

Operator precedence?

Figuring out how to remove `def` (and the forced return) from lambda terms would be a good thing.

Closing a move with the default name should eliminate all non-default extensions of that move. The treatment of saved moves should be tested; there remains the one spot of local bad behavior that I had to correct in the Zermelo implementation.

It might be a good idea to eliminate type information about arguments where possible: testing for the presence of free occurrences of bound variables would seem to be an adequate test.

A low priority issue: with a little cleverness we could automatically display functions as curried application terms to whatever extent is possible.

A useful feature to add would be save files which simply record and restore the declarations.

Typesimplify should be run on types independently.

One thing to note is that in the strong expansion version, we can safely discard all type information about arguments, because it is always deducible from the body of the argument. This does not work in versions with visible let terms. This may have some effect in performance, since for example substitutions have to be made in all those types.

There is the continuous extreme slowness when building large terms, which the original also has. One wonders whether there is something one could cache which would solve this problem.

4/5/2020: I seem to have my original let term simplification protocol working. I should probably set up the weak version to work as well. There is quite a lot of tweaking to be done now that let terms can actually be seen in use.

For example, redundant let terms should be eliminated from types of defined notions.

We have unequivocal success of a version with first class let terms, but it is clear that they are not being used optimally.

4/3/2020: I think I have repaired the problem with the approach without strong expansion, which had to do with problems that arose from discarding type information in typesimplify which turns out to be needed for computing function sorts in the let term approach.

Now the monstrous inefficiencies unsuspected of this approach need to be investigated.

4/2/2020: I identified the silly bug which was causing infinite loops without strong expansion. It probably could have happened with strong exppansion, too; my good fortune.

The bug in the Zermelo interpretation without strong expansion is a failure of implicit argument inference; I'm on it.

4/1/2020: This version runs the Zermelo implementation. I believe it is now a full implementation of Lestrade without rewrites, roughly as reliable as the first version, and with much cleaner code, easier to maintain.

For the record, I note that the crucial update for fixing the slowness was caching the dependency list functions.

I want to debug it to the point where the pure approach with minimal type simplification works (so we can really see let terms in use).

Of course rewrites remain to be fixed.

I am going to remove most of the daily notes.

Apart from actual bugs, the issues encountered which required changes in the Zermelo implementation files were parser issues (spaces needed between special character infixes asd following numerals; the final argument of a prefix term cannot be an infix argument) which I want to leave as they are (the new version is better) and some issues with correct handling of saved moved which forced me to change some identifier names at one point in the development. I should check that my environment saving actually works as intended.

The further issue is that this version suppresses unfixed problems with let terms by expanding everything out much as the original implementation did. It ought to be possible to fix this so that the prover also works when the strong expansion toggle is turned off (and it would be interesting to see what happens once this is done).

Let terms *are* used in this version, but invisibly.

An essay on the rather different bound variable management in this version would be a good idea.

3/30/2020: Working on the slowness problem. Made it do head substitution for laughs.

Tweaked the equalobjects algorithm again, making it always prefer to adjust the second item unless ages of definitions dictate otherwise. I still may prefer to adjust the first item by preference if the adjustment of the second is deep.

3/29/2020: Installed the ability to toggle strong expansion (`Strongexpand` command). The version with minimal type simplification (just dropping unused let terms) crashes for reasons I do not understand which should be fixed, though it mostly works.

Commented diagnostic messages out so that arguments are not silently being built.

3/24/2020: Enabled compressed display; an explicit call to Showdec will always give the full display.

3/14/2020: As my last move, I got the new Lestrade to run the entire third file by writing typesimplify in a way which entirely or almost eliminates let terms. This shows that the underlying logic is implemented, and

things can probably be tweaked so that we can take advantage of the let term machinery.

Let terms as part of the logic are of course charming because they make binders of lambda terms exactly like moves.

I'm preserving this old note because it bears on current business.

3/11/2020 status report: There are minor differences between the parsers in the two versions. I am inclined to leave the current parser as it is: the places in which it differs have merit. The final argument of a prefix term will not be an infix argument (parenthesize it if there is an issue). The final argument of a prefix term must be set off from the previous argument by a comma if it is mixfix; the previous version detects that there is no danger in this case, but it is probably better to leave this version as it is.

3/10/2020: Interesting innovation in pretty printing: put breaks before `{def}` `and` `\verb` let . This further suggests that we should not put the def into lambda terms appearing as arguments.

I found a subtle parser difference between the old version and the new which I think I will leave as is (and make corrections when it arises in files). There is already a difference involving the need to put spaces between infixes and numerals.

3/7/2020 status report: This should be a fairly capable implementation of the Lestrade language and declaration system before rewrite rules.

Its language and its command format differ slightly from those of version 1.0, but version 2.0 should accept any version 1.0 declaration command line (with the exception of rewrite commands). Its file format is totally different, and I need to enable version 1.0 to export files for version 2.0 to read.

Some highlights:

1. Let arguments in functions are a profound change in structure. I'm not sure what the performance issues are until I am able to run large files to test. They create an exact parallelism between binder lists in functions and moves as data structures which did not quite exist before.

2. I have plans to put the rewrite system on a firmer foundation by introducing rewrite objects which will similarly appear as arguments.
3. The old version required the user to enter arguments in the order in which they are declared in the next move. This version automatically (and silently) reorders them.
4. I *think* that the implicit argument mechanism might be a little more general. The development of matching was more principled, and this may pay off because the same matching functions may be usable for development of rewriting.
5. The format of console output is tidier (though handling of prompts makes console output a bit uglier than it needs to be when running a file; probably fixable but not important).
6. The type of Lestrade entities is much more compact and the fact that all entities can be type cast to functions is both striking and very useful.
7. The format of files is entirely different, and I think better and easier to handle. I need to write export function so that version 1.0 can produce version 2.0 files. I'm frightened of what will happen, though!

1 Introduction

This document is intended to serve as a specification of the Lestrade dependent type system and to contain code for the implementation (the Lestrade Type Inspector) in parallel with the specification. The document should be easily converted between LaTeX code and Standard ML code. The code is for Moscow ML 2.01; I should add the preamble(s) needed to run it in other versions of ML.

The following code contains list handling procedures and handles input and output file declarations and various flavors of messages from Lestrade. Notice that everything that Lestrade says without exception goes to the output file as well as the screen.

I am thinking of straining the paradigm of literate programming by in effect making this the long version of the paper on Lestrade, too.

In another iteration of literate programming, I now have the ability to embed Lestrade commands in this text and have the Lestrade file reader process those sections and insert the correct dialogue.

This version has met a benchmark: it now runs the first four files in my Zermelo implementation with some minimal changes having to do with slight differences in the parser, and in one place a need to correct for some unexplained difficulty with move saving.

2 Philosophical Introduction

Lestrade is a system of dependent type theory. It is implemented by the Lestrade Type Inspector, an environment for developing and checking systems of Lestrade declarations, which is in effect a theorem prover of the Automath family. Since it is also in effect a type checker for a type system more than adequate for a language like ML, this suggests that the Inspector might reasonably be extended to an interpreter for a strongly typed programming language, with the additional merit that it could handle types for general mathematical objects and proofs.

There is prior art. Lestrade is motivated by prior familiarity with Automath. Automath has other descendants, notably Coq and Lean (to name systems that are often mentioned). Lestrade is distinguished by a minimalist approach: one way of putting this is that Lestrade was designed as an implementation of an approach to the philosophy of mathematics, and with this in mind is very streamlined. Lestrade is also formally weaker than Automath for principled reasons (though the full strength of any Automath theory can be obtained by suitable declarations). We believe that it is also the case that the core logic of any Automath descendant can be implemented in Lestrade, but this requires some verification. It can be noted that Lestrade has a particular advantage over Automath in the area of flexible local use of names, which is only improved in this new version.

We indicate our philosophical motivations, though these may be better explored implicitly by seeing what the system can do.

We are inclined to view functions as determined by expressions with holes in them (finite objects) rather than infinite tables of values. This does not mean that we identify functions with pieces of text[1]. Looking at the status of the "holes" suggests that we should be friendly to the idea of variables as representing "arbitrary entities" [though this is known to be perilous]. In a sense to be made concrete in the description of the system, we view a variable or parameter of a definition as an object in a "possible world" (the "next move": the "worlds" are actually called "moves" in Lestrade parlance). The declaration of a function in Lestrade involves defining an expression for an entity in the "next move" (an expression with the same tenuous metaphysical status as the parameters living in it); one then postulates a function implementing

[1]For example, the identity conditions of functions are different from those of associated text: renaming of bound variables illustrates this.

this, an entity in the "last move", one step less tenuous: the variables free in the original expression for the variable-dependent object declared in the next move are bound in the expression for the function declared in the last move. This may sound mysterious, but it is concretely implemented in the Lestrade declaration commands. Our underlying philosophy is Aristotelean (we want to appeal to potential rather than actual infinities) but not constructivist.

We view proofs as a species of mathematical object. This is implemented using the Curry-Howard isomorphism: for example, a proof of the implication $p \to q$ is associated with a function from proofs of p to proofs of q. This means that each proposition has to have a correlated sort of proofs of (we prefer to say evidence for) that proposition. This in turn commits us to type theory, and in fact dependent type theory (as we will see when we look at examples).

It should be noted that the ingredients we have mentioned so far might imply that we have a constructive viewpoint. In fact, we do not. We are charmed by the apparent fact that classical Cantorian mathematics is perfectly well accommodated by the framework we develop. Lestrade also implements a constructive viewpoint easily: Lestrade has very few primitive notions, and the user has a great deal of freedom in setting up the basics of her logic. It can be noted that the original Automath system, development of which was the context of one of the independent inventions of the Curry-Howard isomorphism, also was used primarily to implement theories with classical logic, though its descendant Coq now encourages a constructive approach.

We also note that our major implementation project is an implementation of Zermelo's 1908 proof of the Well-Ordering Theorem from the Axiom of Choice, which can hardly be viewed as a constructive result!

Further, it appears that our philosophical prejudices strongly suggest a project of computer implementation of mathematics. And this is what we proceed to carry out.

Fog in our remarks above can be dispelled by concrete discussion of what is to be implemented in code below, examination of the code when it exists, and use of the software to implement mathematical proofs. We set about an explicit description of what we are up to, which will probably bring in more philosophical points.

Expressions in the language of Lestrade can represent *sorts* (types in the internal sense of Lestrade) and *entities*, which have sorts. Entities are further partitioned into *objects* and *functions*. It should be noted that objects are metaphysically primary for us; for a set theorist, a nice way of putting

this is that Lestrade functions are best thought of as proper class functions[2] (functions which are sets will be objects generated from proper class functions by suitable constructors: this can be seen in actual theories).

The metaphysical primacy of objects over functions is also seen in our implementation of proofs as objects. The proof of an implication $p \to q$ is not identified with the associated function from proofs of p to proofs of q: proofs are objects, not functions. We provide a constructor sending functions witnessing an implication to proofs of the implication. This can be seen in examples which will be presented.

We describe the primitive system of object sorts[3].

1. `prop` is an object sort, the sort of propositions.

2. `type` is an object sort, the sort of labels for sorts of mathematical object. We think of `type` as inhabited by type *labels* because we resist the idea that a sort is a set: a sort is a (finite) feature of a particular object encountered, not the infinite collection of all objects with that feature, in our Aristotelean philosophical viewpoint.

3. `obj` is a sort provided for mathematical objects thought of as "untyped". The sets in an implementation of ZFC would probably be of this sort.

4. `that` p, where p is a term of sort `prop`, is a sort, inhabited by proofs of (or evidence for) the proposition p.

5. `in` τ, where τ is a term of sort `type`, is a sort, inhabited by objects of the type with label τ. To make it clearer what we are up to, if `Nat` were the type of natural numbers, `Nat` would be of sort `type` and individual natural numbers would be of sort `in Nat`.

Each Lestrade function takes a fixed positive number of arguments. The sort of each of its arguments is fixed by the sort of the function, in a way which may depend on earlier arguments. The output of the function must be of an object sort fixed by the sort of the function and the sorts of arguments presented to it. The fact that we require that function outputs be objects

[2]with the odd point that they can take proper class functions as input.

[3]The implementation of rewriting under construction 5/21/2021 involves another flavor of object sort inhabited by rewrite rules.

is related to our views on the metaphysical primacy of objects, and also encouraged by features of the Lestrade declaration environment. It can be noted that functions supplied with incomplete argument lists are read as functions (by a version of the well-known device of currying) and in this way a Lestrade function may be read as having function output in some contexts.

In the context of this philosophical introduction, this might best be communicated by giving some examples. A more formal description appears below in the comments on the code.

Logical conjunction is a function taking two arguments of type `prop` to a proposition, so its type is $[(p : \texttt{prop}), (q : \texttt{prop}) \Rightarrow \texttt{prop}]$. Decorating arguments with variable names might seem excessive, but the reasons will become clear.

The rule of conjunction takes a proof pp of p and a proof qq of q to a proof of $p \wedge q$. The type of this is

$$[(p : \texttt{ro}), (q : \texttt{prop}), (pp : \texttt{ha}\, p), (qq : \texttt{that}\, q) \Rightarrow \texttt{that}\, p \wedge q].$$

One might think that the rule of conjunction has only two arguments, the proofs of the two components. But it also needs to know what the two propositions are. On the other hand, having to enter the first two arguments and display them seems excessive, since they can be inferred from the last two. The intention is that Lestrade will type the rule of conjunction as shown but allow the suppression of the first two arguments in input and output (the feature of implicit argument inference). This illustrates what is meant by the assertion that the sorts of arguments may depend on earlier arguments.

If pp is a proof of p and `Conj` is the rule of conjunction, then $\texttt{Conj}(p, p, pp, pp)$ is a proof of $p \wedge p$ (which we intend to be able to enter as $\texttt{Conj}(pp, pp)$ when we have full support for implicit argument inference).

We exhibit a capability of the Lestrade Type Inspector.

```
>>> declare p prop

>>> declare q prop

>>> postulate & p q prop

>>> declare pp that p
```

```
>>> declare qq that q

>>> postulate Conj p q pp qq that p & q

>>> define test pp : Conj(p,p,pp,pp)
```

Above are some Lestrade commands. If we enter these commands bracketed by `begin Lestrade execution` and `end Lestrade execution`, the Lestrade Type Inspector can process the commands in this very file and insert the dialogue, as below:

```
begin Lestrade execution

   >>> declare p prop


   p : prop


   {move 1}

   >>> declare q prop


   q : prop


   {move 1}

   >>> postulate & p q prop


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

```
{move 0}

>>> declare pp that p


pp : that p


{move 1}

>>> declare qq that q


qq : that q


{move 1}

>>> postulate Conj p q pp qq that p & q


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


{move 0}

>>> define test pp : Conj (p, p, pp, pp)


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


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

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

The reader may get the idea that we have presented the actual implementation of the discussion of conjunction above.

It seems the moment to talk about the Lestrade declaration environment. This is a list of lists of declarations called "moves". Each declaration introduces an identifier with a hidden numerical parameter indicating the move in which it is declared. Further it gives either a function sort for the entity represented by the identifier (this may be a 0-ary function sort, in which case it is construed as an object) if the entity is a primitive notion, axiom, or primitive rule of inference, or a lambda-term defining the entity precisely (which will be 0-ary if the defined entity is an object). The second case is that of defined entities, theorems, and derived rules of inference. The original commands of Lestrade unpack the function sort or lambda term into a list of parameters and an object sort or specific object output.

There are always at least two moves. Move 0 contains unequivocally declared objects, functions, axioms, theorems and rules of inference of the theory under construction. The highest-indexed move (called the "next move") contains entities which are completely hypothetical in character, used as parameters in definitions and bound variables in declared lambda-terms and function sorts. The entities in the second-to-last move (the "last move" or the "current move") are entities which are for the moment definitely posited, but this is relative in the following sense: the `close` command will discard all declarations in the next move[4], whereupon the declarations in the new next move (the erstwhile last move) will acquire the tenuous character they had when originally introduced; on the other hand, the `open` command will introduce a new empty "next move" and allow us to posit all entities in the erstwhile next move more definitely... for the moment.

Three styles of declaration command occur in the example above. The `declare` command declares an identifier which is intended for use as a parameter. The `postulate` command introduces an identifier (with a list of arguments already declared in the next move) and an object sort, and declares the identifier as a primitive construction (in the last move, not the

[4]This is where we gain the benefit of flexible local use and re-use of names.

next move) taking arguments of the indicated sorts to an object of the given sort. Since declarations in the next move can depend on objects appearing earlier in the next move, this supports declarations of functions with dependent types. The `define` command takes an identifier, a list of arguments taken from the next move, and an object term, and declares the identifier as the function in the last move with the given output (defining a lambda-term in effect). There are natural requirements that all dependencies of the items in one of these declaration commands on things declared in the next move are actually present in the argument list (or, eventually, may be inferred by the implicit argument inference feature).

When function sort and lambda-term expressions *are* entered by the user, bound variables will always be copies of variables declared in the next move. This has the practical effect that it is never necessary to embed types of bound variables in terms as entered by the user, and in fact this is not supported by the parser.

It is a possibly interesting feature of the logic of Lestrade that one can in principle realize any dependent type using just the `open`, `close`, `declare`, `postulate`, and `define` commands, with the user writing no terms other than atomic object and function terms and object application terms (this has the specific effect that one can only `declare` object identifiers). It is not in principle necessary for the user ever to write a term in which variables are bound, whether it is a function sort term or a lambda-term. This was not true in Automath: the version PAL of Automath in which terms with variable binding were not written by the user was of interest but was strictly weaker than Automath. The reason that things are different here is the more complicated declaration environment: the clever use of moves allows one to avoid variable binding (though one does not want to do this in practice). The first version of Lestrade actually enforced this restriction. We overstated the extent to which this original version of Lestrade "did not have bound variables" : the appearance of function sorts and lambda terms in output was unavoidable. It also turned out to be extremely valuable in practice to be able to `declare` identifiers of function sorts and to supply lambda-terms as arguments to functions. The language of Lestrade continues (in an analogy to the language of Russell and Whitehead in *Principia Mathematica*) not to support lambda-terms in applied position: substitution of a lambda-term for an applied function variable always induces immediate beta-reduction.

The `close` command discards the declarations made in the next move. It is however possible to save the next move beforehand (usually with an

explicit name), which allows a set of local declarations of interest to be reopened for later use. This has the side-effect that masking of declarations is possible. It is never possible to declare an identifier in the next move or the last move which conflicts with an identifier already introduced in either of those moves or any earlier move, but it is possible to save a move, close it, introduce identifiers in earlier moves which conflict with those in the saved move, and reopen the saved move. In this case, the identifiers of the saved move will mask the ones introduced later but at earlier moves, but declarations depending on the masked identifiers will work correctly (the current implementation handles this more neatly than the original implementation).

It is a further technicality (mostly important in move 1, where the `close` command will not work) that one may clear the next move without exiting it, using the `clearcurrent` command, and one may use the `clearcurrent` command as well as the `open` command with a name as argument to open a saved set of declarations as the next move.

As we have intimated above, our idea is that the moves are to be thought of as possible worlds. A modality which might be useful as a metaphor here is temporal: think of an entity declared in the next move as an entity of the appropriate type which is unknown to the user, and so can play the role of any entity of that type she may encounter in the future.

In the current implementation, the role of the entities declared in the next move is primarily as bound variables. The original `declare`, `postulate`, and `define` commands are implemented in terms of new `Declare` and `Posit` commands. Each of these commands takes a function sort (for a primitive notion) or a lambda-term (for a defined notion) as its argument: `Declare` makes declarations to the next move and `Posit` to the last move. The other commands are implemented by using the argument lists as binders along with the final argument as body to construct a function sort or lambda-term. But the legacy commands remain available to the user, and the fact that these parameter based commands can implement anything normally implemented with variable binding remans interesting.

We introduce an extended example in the old style which does not support user-entered function sorts and lambda-terms.

```
begin Lestrade execution

   >>> Clearall
```

```
{move 1}

>>> open


   {move 2}

   >>> declare x obj


   x : obj


   {move 2}

   >>> postulate P x : prop


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


   {move 1}

   >>> close


{move 1}

>>> comment The effect of the little block \
    above is to declare P as a function (actually \
    predicate) parameter .Being postulated \
    when move 2 is the next move then closing \
    leaves P as a predicate of untyped objects \
    (obj) at move 1, a predicate parameter \
    .
```

```
{move 1}

>>> postulate set P : obj


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


{move 0}

>>> comment Above we declare set builder \
    notation : set P can be read {x : P (x)} .


{move 1}

>>> declare x obj


x : obj


{move 1}

>>> declare y obj


y : obj


{move 1}

>>> postulate E x y : prop


E : [(x_1 : obj), (y_1 : obj) =>
```

```
    (--- : prop)]


{move 0}

>>> comment We declare the membership \
    relation .


{move 1}

>>> declare x1 that P x


x1 : that P (x)


{move 1}

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


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


{move 0}

>>> declare x2 that E x set P


x2 : that x E set (P)


{move 1}
```

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


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


{move 0}

>>> declare p prop


p : prop


{move 1}

>>> comment We declare the unrestricted \
    axiom of set comprehension (sending evidence \
    for P (x) to evidence for {x : P (x)} and \
    vice versa) .Of course this assumption \
    will prove disastrous .


{move 1}

>>> declare q prop


q : prop


{move 1}

>>> postulate Implies p q : prop
```

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


{move 0}

>>> postulate False : prop


False : prop


{move 0}

>>> declare pp that p


pp : that p


{move 1}

>>> declare rr that Implies p q


rr : that p Implies q


{move 1}

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


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

```
{move 0}

>>> declare absurd that False


absurd : that False


{move 1}

>>> postulate Panic p absurd : that p


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


{move 0}

>>> define Not p : Implies p False


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


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


{move 0}

>>> open


   {move 2}
```

```
   >>> declare pp2 that p


   pp2 : that p


   {move 2}

   >>> postulate Ded pp2 : that q


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


   {move 1}

   >>> close


{move 1}

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


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


{move 0}

>>> comment Above find the needed primitives \
    of propositional logic .
```

```
{move 1}

>>> define Russell x : Not E x x


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


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


{move 0}

>>> define R : set Russell


R : [({def} set (Russell) : obj)]


R : obj


{move 0}

>>> open


   {move 2}

   >>> comment We define the Russell predicate \
       of non - self - membership and the \
       associated set .


   {move 2}

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

```
    Russell


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


{move 2}

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


R2 : [(R1_1 : that set (Russell) E set
    (Russell)) => ({def} comp2
    (Russell, set (Russell), R1_1) : that
    Russell (set (Russell)))]


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


{move 1}

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


R3 : [(R1_1 : that set (Russell) E set
    (Russell)) => ({def} Mp (set
    (Russell) E set (Russell), False, R1_1, R2
    (R1_1)) : that False)]


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

```
      {move 1}

      >>> close


   {move 1}
end Lestrade execution
```

Here we insert a comment in ordinary text (by interrupting the Lestrade execution block): R4 takes definition arguments, which we originally had to supply in this version, but which are now supplied implicitly.

```
begin Lestrade execution

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


   R4 : [({let} .R2_1 : [(R1_2 : that
          set (Russell) E set (Russell)) =>
          ({def} comp2 (Russell, set
          (Russell), R1_2) : that Russell
          (set (Russell)))]), ({let} .R3_1
       : [(R1_2 : that set (Russell) E set
          (Russell)) => ({def} Mp (set
          (Russell) E set (Russell), False, R1_2, .R2_1
          (R1_2)) : that False)]) =>
       ({def} Impliesproof (set (Russell) E set
       (Russell), False, .R3_1) : that
       (set (Russell) E set (Russell)) Implies
       False)]


   R4 : [({let} .R2_1 : [(R1_2 : that
          set (Russell) E set (Russell)) =>
```

```
       ({def} comp2 (Russell, set
       (Russell), R1_2) : that Russell
       (set (Russell)))]), ({let} .R3_1
    : [(R1_2 : that set (Russell) E set
       (Russell)) => ({def} Mp (set
       (Russell) E set (Russell), False, R1_2, .R2_1
       (R1_2)) : that False)]) =>
    (--- : that (set (Russell) E set
    (Russell)) Implies False)]


{move 0}

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


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


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


{move 0}

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


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


R6 : that False
```

```
   {move 0}

   >>> comment and the familiar proof of \
       absurdity ensues .


   {move 1}
end Lestrade execution
```

Now we begin the actual code of the Type Inspector with some utilities.

```
*)

(* 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 *)

(* utility code *)

open TextIO;

open OS;

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

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

val READFILE = ref(openIn("default2"));

fun say s = (output(stdOut,"\n"^s);
output (stdOut,"\n\n(paused, type something to continue) >");
 flushOut(stdOut);
output(!LOGFILE,"\n Inspector Lestrade says:  "^s);
output (!LOGFILE,"\n\n(paused, type something to continue) >");
 flushOut(!LOGFILE);
input(stdIn);());

fun say2 s = (output(stdOut,"\n"^s);
output (stdOut, "\n"^ ">>"); flushOut(stdOut);
output(!LOGFILE,"\n"^s);
(* output (!LOGFILE,"\n >>"); *) flushOut(!LOGFILE);
(* input(stdIn); *)());

fun say3 s = (output(stdOut,"\n"^s);
output (stdOut,"\n"); flushOut(stdOut);
output(!LOGFILE,"\n"^s);
(* output (!LOGFILE,"\n"); *) flushOut(!LOGFILE);
(* input(stdIn); *)());
```

```
(* fun say s = (output(stdOut,"\n"^s);flushOut(stdOut);
output (stdOut,"\n\n(paused, type something to continue) >");
 flushOut(stdOut);
output(!LOGFILE,"\n Inspector Lestrade says:  "^s);
output (!LOGFILE,"\n\n(paused, type something to continue) >");
 flushOut(!LOGFILE);
input(stdIn);()); 

fun say2 s = (output(stdOut,"\n"^s);flushOut(stdOut);
output (stdOut,"\n >>"); flushOut(stdOut);
output(!LOGFILE,"\n"^s);flushOut(!LOGFILE);
(* output (!LOGFILE,"\n >>"); *) flushOut(!LOGFILE);
(* input(stdIn); *)()); 

fun say3 s = (output(stdOut,"\n"^s);flushOut(stdOut);
output (stdOut,"\n"); flushOut(stdOut);
output(!LOGFILE,"\n"^s);flushOut(!LOGFILE);
output (!LOGFILE,"\n"); flushOut(!LOGFILE);
(* input(stdIn); *)()); *)

val DIAGNOSTIC = ref false;

fun diagnostic() = DIAGNOSTIC := not(!DIAGNOSTIC);

fun sayD s = if (!DIAGNOSTIC) then say3 s else (); (* diagnostic messages *)

fun say2D s = if (!DIAGNOSTIC) then say2 s else (); (* diagnostic messages *)

fun versiondate() = say3
("Welcome to the Lestrade Type Inspector\n"^
"version 2.0, release of 1/5/2022\n4  pm Boise time, for PolyML")

fun explain b s = if b then b else (say s;b);

fun explain2 b s = if not b then b else (say s;b);

fun max(m,n) = if m>n then m else n;
```

```
fun p1(x,y) = x;

fun p2(x,y) = y;

(* kill empty errors *)

fun Hd default nil = (say "underflow error in Hd";default) |

Hd default (x::L) = x;

fun Tl nil = (say "underflow error in Tl";nil) |

Tl x = tl x;


(* is an item in a list? *)

fun find0  s nil = false |

find0 s (t::L) = if s=t then true else find0 s L;

fun drop0 s nil = nil |

drop0 s (t::L) = if s=t then drop0 s L
else (t::(drop0 s L))

fun add0 s  L = s::(drop0 s L);

fun merge0 L nil = L |

merge0 nil M = M |

merge0 ((s)::L) ((t)::M) =

add0 s  (add0 t  (merge0 L M));
```

```
(* look up an item in a labelled list *)

fun find default s nil = default |

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



fun drop s nil = nil |

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

fun findfancy default s LREF =

let val FOUND = find default s (!LREF) in

if FOUND = default then default

else (LREF:= (s,FOUND)::(drop s (!LREF)); FOUND) end;

fun add s t L = (s,t)::(drop s L);

fun extendslist nil L = true |

extendslist L nil = false |

extendslist L M = hd(rev L) = hd(rev M)
andalso extendslist (rev(tl(rev L))) (rev(tl(rev M)));


(*
```

3 Tokens and Identifiers

We begin the discussion of Lestrade notation. It should be noted that Lestrade's output notation is a proper superset of its input notation. In particular, bound variables in output are in a format which Lestrade does not know how to parse. Our view is that this is harmless (and that it both reveals and enforces something about our underlying approach), but it does require comment.

Tokens are open and close parentheses, open and close brackets, commas, colons, and identifiers.

If a double quote " occurs in a line, the entire rest of the line is a token (dropping the quote). This allows tokens of quite uncontrolled form to be supplied (for example, file names). This could be changed to support quoted strings as tokens, probably a good thing.

Identifiers come in three flavors:

1. strings of digits (not zero-initial unless of length one; this is true of all strings of digits in this definition) optionally followed by one or more single quotes;

2. strings of letters with the first optionally capitalized followed optionally by a string of digits followed optionally by one or more single quotation marks ';

3. strings of special characters (any typewriter character which is not a letter, a token, a period, a single quotation mark, or an underscore _) followed optionally by a string of digits followed optionally by one or more single quotes.

Whitespace (made up of spaces and backslashes \ , and units composed of a backslash followed by zero or more spaces followed by a carriage return) is only significant to the tokenizer where it separates identifiers which might otherwise run together. A carriage return ends text to be tokenized unless it is preceded by a backslash followed optionally by additional whitespace.

Identifiers ending in single quotes are usually generated automatically by Lestrade, though users can declare them[5]. When Lestrade needs to generate

[5]In the previous implementation, identifiers with special characters were extended with dollar signs. Also note that at the moment one may use identifiers which are strings of single quotes, which turned out to be convenient while debugging the code. It would be easy to forbid such identifiers but I haven't bothered.

an identifier from an existing name for various special purposes, it will add enough single quotes to get an identifier which is not declared yet.[6]

The output notation contains extended identifiers which consist of an identifier as described above, optionally preceded with a period and optionally followed by an underscore followed by a sequence of digits. These are generated by Lestrade, and the parser does not understand them.

It is worth noting that the identifier reading function omits initial zeroes from sequences of digits of length greater than one.

The function `Getline()` tests the tokenizer. The user enters input (more than one line may be used if lines end with backslashes): the system then displays the tokenized version of the input.

The conversion from lists of tokens to displayed text takes special properties of parentheses, braces, brackets, commas, and underscores into account when deciding whether to insert spaces.

The token display mechanism has odd effects on the display of punctuation in comments in Lestrade output files, but this seems harmless.

[6]For the moment, the only use for automatically generated identifiers is for matching variables in the implicit argument inference mechanism. These can occasionally end up embedded in prover output as bound variables.

```
*)

(* Here we should put tokenizer and line-reading code *)

(* character classes *)

fun islower c = c <= #"z" andalso c >= #"a";

fun isupper c =  c <= #"Z" andalso c >= #"A";

fun isnumeral c = c <= #"9" andalso c >= #"0"

fun isspecial c = not(islower c) andalso not(isupper c)
 andalso not(isnumeral c) andalso not(find0 c
 [#"(",#")",#"[",#"]",#"{",#"}",#",",
 #".",#"_",#"\"",#"'",#" ",#"\n",#"\\",#":"]);

 (* this function reads an identifier initial in a stream of characters *)

 (* I like the two character lookahead in this version.
I changed the extend mechanism
 to use single quotes for all identifiers. *)

 fun getidentifier nil = nil |

 getidentifier [x] = if islower x orelse isupper x
orelse isnumeral x orelse isspecial x orelse x= #"'" then [x]
    else nil |

getidentifier (x::y::L) = if not
(islower x orelse isupper x
orelse isnumeral x orelse isspecial x orelse x= #"'")
then nil
else if not (islower y orelse isupper y
orelse isnumeral y orelse isspecial y
orelse y = #"'") then [x]
else if isupper x andalso
```

```
(islower y orelse isnumeral y orelse y = #"'")
then (x::(getidentifier (y::L)))
else if islower x andalso
(islower y orelse isnumeral y orelse y = #"'")
then (x::(getidentifier (y::L)))
(* else if x= #"0" andalso isnumeral y
then getidentifier (y::L) *)
else if isnumeral x andalso
(isnumeral y orelse y = #"'")
then x::(getidentifier(y::L))
else if isspecial x andalso
(isspecial y orelse isnumeral y
orelse y = #"'") then x::(getidentifier (y::L))
else if x = #"'" andalso y = #"'"
then x::(getidentifier(y::L))
else [x]


 and restidentifier  nil = nil |

 restidentifier [x] = if islower x orelse isupper x
orelse isnumeral x orelse isspecial x orelse x = #"'" then nil
    else [x] |

restidentifier (x::y::L) = if not
(islower x orelse isupper x
orelse isnumeral x orelse isspecial x orelse x = #"'")
then (x::y::L)
else if not
(islower y orelse isupper y
orelse isnumeral y orelse isspecial y
orelse y = #"'") then (y::L)
else if isupper x andalso
(islower y orelse isnumeral y orelse y = #"'")
then ((restidentifier (y::L)))
else if islower x andalso
(islower y orelse isnumeral y orelse y = #"'")
then ((restidentifier (y::L)))
```

```
(* else if x= #"0" andalso isnumeral y
then restidentifier (y::L) *)
else if isnumeral x andalso isnumeral y
orelse y = #"'" then (restidentifier(y::L))
else if isspecial x andalso
(isspecial y orelse isnumeral y
orelse y = #"'") then (restidentifier (y::L))
else if x = #"'" andalso y = #"'"
then restidentifier(y::L)
else (y::L);

 fun Getidentifier s = implode(getidentifier(explode s));
 fun Restidentifier s = restidentifier(explode s);

 (* this is the tokenizer *)

 fun tokenize nil = nil |

 tokenize (#"\""::L) = [implode L] |

 tokenize (#" "::L) = tokenize L |

 tokenize (#"\t"::L) = tokenize L |

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

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

 tokenize (#"\n"::L) = nil |

 tokenize (c::L) = if islower c
 orelse isupper c orelse isnumeral c
 orelse isspecial c then
 (implode(getidentifier (c::L)))
 ::(tokenize (restidentifier(c::L)))

 else (implode [c])::(tokenize L)
```

```
 and resttokens nil = nil |

 resttokens (#"\""::L) = nil |

 resttokens (#" "::L) = resttokens L |

 resttokens (#"\t"::L) = resttokens L |

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

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

 resttokens (#"\n"::L) = nil |

 resttokens (c::L) = resttokens L;

fun Tokenize s = tokenize(explode s);

fun displaytokens nil = "\n" |

displaytokens [c] = c |

displaytokens (c::d::L) =
c^(if find0 d [",",")","}","]","_"]
orelse find0 c ["_","(","{","[","."]
then "" else " ")^(displaytokens(d::L));

fun despace [x] = [x] |
despace (#" "::L) = despace L|
  despace (#"\t"::L) = despace L|
  despace L = L;

val LINESOFAR = ref "";

fun getline() = (LINESOFAR:=input(stdIn);
    if (!LINESOFAR) <> "\n"
andalso Hd #"\n" (despace(Tl(rev(explode (!LINESOFAR)))))= #"\\"
then (output(stdOut,">> ");flushOut(stdOut);(!LINESOFAR)^(getline()))
```

```
else ((!LINESOFAR)));

fun Getline() = let val TEXT = getline() in

output(stdOut,displaytokens (Tokenize TEXT)) end;

fun getline2() = (LINESOFAR:=Inputline(!READFILE);
    if (!LINESOFAR) <> "\n"
andalso Hd #"\n" (despace(Tl(rev(explode (!LINESOFAR)))))= #"\\"
then (output(stdOut,">> ");flushOut(stdOut);(!LINESOFAR)^(getline2()))
else ((!LINESOFAR)));

(*
```

4 The basic data types of Lestrade entities: objects, object sorts, and functions

This is a much more compact basic declaration than in the original implementation!

Lestrade entities fall into four categories: objects, object sorts, functions, and function sorts. The Type Inspector has a dedicated data type for objects and a dedicated data type for object sorts. Functions and function sorts are both handled by the same data type (am n-ary lambda term, with its three components being the list of bound variables, the body of the lambda term, an object value, and the object output sort of the lambda term). The implementation puts functions and function sorts in the same type: a function sort can be identified by the presence of the dummy body `Unknown`. Moreover, objects and object sorts can be type-cast into the type of functions, by being supplied with null argument lists; an object is identified with the corresponding 0-ary function, and an object sort is similarly represented by a 0-ary term with body `Unknown`. It should be noted that objects can have either function or object input, but they always have object output. Object terms represented by formal functions may have a dummy output type for two distinct reasons: the type of an object can be computed without it needing to be recorded in the function type object representing it, and some object terms do not represent objects (an applicative term with a short argument list represents a curried function); when these are type cast to function terms, they cannot of course be assigned an object type output[7]. The output type of a function does need to be recorded, because the object term body of the function will contain bound variables, and the type algorithm of Lestrade does not work directly with bound variables (this is due to the same attitude reflected in our not supporting parsing of bound variables).

Identifiers contain a boolean tag indicating whether they are implicit arguments (hidden arguments of atomic functions which are deduced from the explicitly given arguments). In the original implementation, implicit status was handled by awkward manipulation of identifier names. Identifiers are tagged internally (no effect on the display) with the environment in which they are declared; this will allow much more elegant handling of masked identifiers. It can be noted that the same identifier with or without the

[7]Such fake object terms now appear only as input: the `typefix` function eliminates them in favor of eta-equivalent function terms.

implicit argument tag represents the same entity, and substitution, equality, and matching functions of the prover take this into account. Update on this: it seems that an approach allowing the user to see let terms requires that type information about arguments be preserved.

Food for thought: there is an alternative model in my mind under which more type information will be stored in terms, though usually not displayed. This would need the additional provision for a type field for the `App` construction. We could then have types on bound variables, for example. It is an interesting question how far our philosophy of forgetting type information about expressions buried in function sorts or lambda terms can be carried.

Further food for thought: perhaps the error atoms should have an obligatory argument, an error message which can be given an appropriate value when one is created.

The `Deferred` field is intended to support a feature in the original implementation which we have not added here, which allows declaration of a defined notion whose actual definition body is deferred, and which may be set automatically by matching.

5/17/2021: introducing the `Rewrite` function type. `Rewrite`(t, u) is an instruction to rewrite object terms matching t to the corresponding object terms matching u. The only way in which a rewrite term is a subterm of another object term is as an argument driving rewrites. Functions generating rewrites can be postulated. It is wirth noting that all veriables taken from the next move in a rewrite object are actually bound.

```
*)


(* Here we should put the basic term and sort declarations *)

datatype Object =

     Oatom of (string*int*int*bool)
    (* atomic object term:  the second integer is the
    binding context (0 if free);  the first integer is
    the move in which the atom is declared *) |

     App of (string*int*int*bool*(Function list))
```

```
    (* application term:  the Fatom with the
    first three arguments is applied to the
    list of argument Functions (using type
   casting from objects to functions for
   object arguments *) |

    Unknown (* the dummy output value of a function sort *) |

    Deferred (* the unknown output value
                   associated with a deferred definition *) |

    Oerror (* object term error *)

and Otype =

    PROP |

    TYPE |

   OBJ |

    IN of Object |

   THAT of Object |

  Rewrite of (Object*Object)

  (* Rewrite(t,u) is the type of a rule allowing
rewriting of t to u.  Rewrite rules are functions
with values of Rewrite types:
(Lx_1...x_n:Rewrite(t(x_1...x_n),u(x_1,...,x_n)))
will rewrite terms matching t(x_1...x_n) to
terms matching u(x_1,...,x_n) *)

|

   Tdummy |
```

```
   (* dummy type for use in objects in
   applicative argument lists *)

   Terror (* object type error *)

and Function =

   Fatom of (string*int*int*bool)

  (* the second integer is binding context and the first is
  context of declaration *) |



   Lambda of ((((string*(int*int*int*bool*Function))) list)*Object*Otype)

  (* the initial integer in each quadruple in the argument list
  is its index, the string is the name of the argument, the
 first integer is binding context and the second context
 of declaration of the original variable from which it is copied

Adding boolean component for implicit arguments.
*)|

   Fdummy

   (* dummy function type for use in parsing abstractions --
   type to be filled in *)

   |

   Ferror

  (* the Function type contains both lambda terms
  (with a real object component)
  and function sorts (with Unknown as the object component) as well as
  as yet undetermined definitions with Deferred as the object component *)
```

```
  (* further, it contains objects and object sorts
   as 0-ary functions and function sorts *)

;

(*
```

5 Format and arity

This is the machinery for representation of declaration information needed by the parser. The basic format information for an identifier is a list of arguments qualified as implicit, explicit, and let terms. From this two different arities can be computed, the number of explicit arguments and the number of non-let arguments. The number of explicit arguments is of course what is of interest to the parser. This format information was initially recorded as a separate table used by the parser, but format is now deduced on the fly from identifier declarations. The tables may be reintroduced later to enhance performance.

Revised 5/14/2021 to get the error treatment of final implicit arguments to work as must have been intended.

```
*)

(* expanded arity formation *)

datatype FormatItem =

Explicit | Implicit | LetTerm | FormatError;

(* val FORMATS = ref (nil:((string*int)*(FormatItem list)) list); *)

fun formattoarity1 nil = 0 |

formattoarity1 [Implicit]= ~1 |

formattoarity1 (Explicit::L) = let val A = formattoarity1 L
in if A = ~1 then ~1 else A+1 end |

formattoarity1 (x::L) = formattoarity1 L;

fun formattoarity2 nil = 0 |

formattoarity2 [Implicit]= ~1 |
```

```
formattoarity2 (LetTerm::L) = formattoarity2 L |

formattoarity2 (x::L) = let val A = formattoarity2 L in
if A = ~1 then ~1 else A+ 1 end;

(*
```

I believe that format and arity are under control. Food for thought, to be attended to later in the file: error messaging needs output commands which ignore format considerations (displaying all arguments).

6 The Lestrade Environment

A Lestrade environment is a list of "moves": a move is a list of Lestrade declarations tagged with a name.

It is worth noting at the outset that a Lestrade move (disregarding the tag) is actually precisely the same data structure as the inputs list of a Lestrade function sort or lambda term, though displayed differently. This was not quite true in the previous implementation: the addition of let notation made this fully the case.

The ML functions implementing the user commands to open a new environment, to close an environment, and to clear the current environment are provided here.

It can be noted that the index which is the first component of items in binder lists or moves is actually not used at all now. It was used in the previous implemention as a quick and easy way to enforce dependency conditions in argument lists.

I need to give thought to the possibility that meanings of identifiers in earlier moves will be masked by identifiers in later moves and how this should be handled. Internal representations will draw correct distinctions between different versions of the identifiers, but output will not. It is significant that the Lestrade user never has occasion to parse output notation (since Lestrade cannot in general do this!) Masking is possible for a subtle reason. The user cannot declare identifiers in a later move which conflict with identifiers in an earlier move. What the user *can* do is save a later move, go back to an earlier move, declare new identifiers which conflict with those in the saved later move, and then reopen the saved later move. This will cause masking. Attempts to reference the masked identifiers directly should be unsuccessful; but definitions which depend on the masked identifiers should work correctly.

The internal representations of the move opening, closing, and clearing commands are provided here.

Move saving is now also supported. The move is internally represented by the sequence of names of moves down to move 0. The name of move 0 is immutable since it is never the next move. The default name for a move is of course the string form of its index. The save command saves all moves on the current path. The open command with an argument opens the next move with name its argument or creates a new one. The clearcurrent command clears the next move and replaces it with one with name its argument if there is one (otherwise creating a blank move with that name). If the save

command has an argument, it saves the current moves with the next move having a new name; if it does not it saves it with the current name. The open command does, as in the previous implementation, now load a saved move with the default name if there is one. It is important to note (and required correction) that the environment after the open or save command only adds or removes declarations at the next move (this being a new move in the case of the open command). It may be that only the state of the next move should be saved under each key (this is now the case).

The maintenance of the format list in parallel with the declaration context was abandoned: currently formats are computed as needed from identifier declarations.

Notice the appearance of caches for the `objectsort` and `functionsort` commands below. These appear at this point because they need to be partially purged whenever a move is closed by the close or clearcurrent commands.

I made a significant savings by saving only the next move rather than the entire context for each list of move names serving as a key.

A subtle point about the clearcurrent command: when it does not have an argument, it creates a blank move with the default name, and erases any saved move of that name which may already exist and all of its extensions. When one opens a move with the default name, one erases any existing saved move with that name, but not extensions.

```
*)


(* basic declaration of the context *)

val CONTEXT = ref [(nil:(((string*(int*int*int*bool*Function))) list)),
      (nil:(((string*(int*int*int*bool*Function))) list))];

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

val CONTEXTS = ref [(["1","0"],hd(!CONTEXT))];

val _ = CONTEXTS:=nil;

val  OBJECTSORTCACHE = ref [((!CONTEXTNAMES,Oatom("?!?",0,0,true)),Ferror)];

val _ = OBJECTSORTCACHE:=nil;

val FUNCTIONSORTCACHE = ref [((!CONTEXTNAMES,Fatom("?!?",0,0,true)),Ferror)];

val _ = FUNCTIONSORTCACHE:=nil;

val SORTOFCACHE = ref(nil:(((string*int)*Function) list));

val DEPLISTCACHE = ref(nil:(((string*int)*((string*int*int) list))list));

val SAVED = ref [("",((!CONTEXT),!CONTEXTNAMES,!CONTEXTS))];

val _ =SAVED := nil;

fun Load s =

let val (x,y,z) = find (nil,nil,nil) s (!SAVED) in

if x=nil then say "No such theory to load"

else (
```

```
OBJECTSORTCACHE := nil;
FUNCTIONSORTCACHE := nil;
SORTOFCACHE:=nil;
DEPLISTCACHE:=nil;
CONTEXT := x;
CONTEXTNAMES := y;
CONTEXTS := z

)

end;

(* purge caches before close or clearcurrent *)

(* there are two different approaches *)

fun purgecaches() =  (OBJECTSORTCACHE:=nil;
FUNCTIONSORTCACHE:=nil; SORTOFCACHE:=nil;
DEPLISTCACHE:=
drop (("",~1))
(map (fn ((s,m),x)=> if m=length(!CONTEXT)-1
then (("",~1),nil)
else ((s,m),x)) (!DEPLISTCACHE))
);

(* (OBJECTSORTCACHE:=
drop (nil,Oatom("?!?",0,0,true))
(map (fn ((x,y),z)=> if x = (!CONTEXTNAMES)
then ((nil,Oatom("?!?",0,0,true)),Ferror)
else ((x,y),z)) (!OBJECTSORTCACHE));
FUNCTIONSORTCACHE:=drop (nil,Fatom("?!?",0,0,true))
(map (fn ((x,y),z)=> if x = (!CONTEXTNAMES)
then ((nil,Fatom("?!?",0,0,true)),Ferror)
else ((x,y),z)) (!FUNCTIONSORTCACHE))); *)

(* clear everything and start over *)

fun basicstart() = (CONTEXTS:=nil;
```

```
 CONTEXT:= [nil,nil];
OBJECTSORTCACHE:=nil;
FUNCTIONSORTCACHE:=nil;
SORTOFCACHE:=nil;
DEPLISTCACHE:=nil;
CONTEXTNAMES:=["1","0"] );

(* open, close and clearcurrent environment
commands with no attention to environment
saving, using default move names *)

fun basicclose() = (if length(!CONTEXT) > 2
then (purgecaches();CONTEXT := Tl(!CONTEXT);
CONTEXTNAMES:= Tl(!CONTEXTNAMES))
else say "Cannot close world 1");

fun basicopen() = (CONTEXTNAMES:=
   (makestring(length(!CONTEXT)))::(!CONTEXTNAMES);
CONTEXT:=nil::(!CONTEXT));

(* clearcurrent in default mode must eliminated
saved moves extending its position *)

fun basicclearcurrent() =
(purgecaches();CONTEXT:= nil::(Tl(!CONTEXT));
CONTEXTNAMES:= (makestring(length(!CONTEXT)-1)::(Tl(!CONTEXTNAMES)));
CONTEXTS := drop nil (map (fn(w,x)=>
(if extendslist (!CONTEXTNAMES) w
then nil else w,x)) (!CONTEXTS)));

(* save command *)

val BACKUPCONTEXT = ref(!CONTEXT);

val BACKUPCONTEXTNAMES = ref(!CONTEXTNAMES);

(* the context save command.  Everything is saved
down to the root. *)
```

```
fun savecontext0() =

( CONTEXTS:= add (!CONTEXTNAMES)(hd(!CONTEXT))(!CONTEXTS);

if length(!CONTEXT)>2 then

(CONTEXT:=Tl(!CONTEXT);
CONTEXTNAMES:=
   Tl(!CONTEXTNAMES);savecontext0())

else ());

fun savecontext() =

(BACKUPCONTEXT:=(!CONTEXT);
BACKUPCONTEXTNAMES:=(!CONTEXTNAMES);
savecontext0();
CONTEXT:=(!BACKUPCONTEXT);
CONTEXTNAMES:=(!BACKUPCONTEXTNAMES));

fun save s = (
if s <> hd(!CONTEXTNAMES) then
CONTEXTS := drop nil (map (fn(w,x)=>
(if extendslist (s::(Tl((!CONTEXTNAMES)))) w
then nil else w,x)) (!CONTEXTS))
else();
CONTEXTNAMES:=s::(Tl(!CONTEXTNAMES));
savecontext());

fun Open s =

(CONTEXTNAMES:=s::(!CONTEXTNAMES);
CONTEXT:= ((* hd*) (find (nil)
 (!CONTEXTNAMES)(!CONTEXTS)))::((!CONTEXT));
 if s = makestring(length(!CONTEXT)-1) then CONTEXTS:=
 drop (!CONTEXTNAMES) (!CONTEXTS) else ());
(* FORMATS:= find (!FORMATS) (!CONTEXTNAMES)(!FORMATLISTS)*)
```

```
fun clearcurrent s =

(purgecaches();CONTEXTNAMES:=(s::(Tl(!CONTEXTNAMES)));
CONTEXT:= ((* hd *)(find (nil)
 (!CONTEXTNAMES)(!CONTEXTS)))::(Tl(!CONTEXT))
);

(* find (relative) declaration information about a name in the context *)

fun contextfind0 default s nil = default |

contextfind0 default s (L::LL) =

let val FOUND = find default s L in

if FOUND <> default then FOUND else contextfind0 default s LL end;

fun contextfind s = contextfind0 (~1,~1,0,true,Ferror) s (!CONTEXT);

fun age s = let val (i,m,n,b,T) = contextfind s in

i end;

fun env s =  let val (i,m,n,b,T) = contextfind s in

m end;

fun sortof s =  let val (i,m,n,b,T) = contextfind s in

T end;

fun world0 default 0 L = if L=nil then default else Hd default L |

world0 default n L = if n<0 orelse L=nil
then default else world0 default (n-1) (Tl L);

fun world n = world0 nil n (rev(!CONTEXT));
```

```
fun worldname n = world0 "" n (rev(!CONTEXTNAMES));

fun absolutecontextfind s m = find (~1,~1,0,true,Ferror) s (world m);

fun absoluteage s m = let val (i,mm,n,b,T) = absolutecontextfind s m in

i end;

fun absolutesortof s m =

let val A = findfancy Ferror (s,m) SORTOFCACHE in

if A <> Ferror then A else

let val (i,mm,n,b,T) = absolutecontextfind s m in

(SORTOFCACHE:=((s,m),T)::(!SORTOFCACHE);T) end  end;

(* fun absolutesortof s m =  let val (i,mm,n,b,T) = absolutecontextfind s m in

T end; *)

(* generate a new undeclared identifier *)

fun extend s = if env s <> ~1 then extend(s^"'") else s;

(*
```

6.1 Pretty printing functions

The aim here is to present functions which will reformat tokenized output with helpful line breaks and indentation. The degree of indentation is determined by the move we are in and the degree of nesting within brackets [] at which we are processing. Note the theme of identification of declaration environments (moves) with variable binding environments (lambda terms and function sorts).

There are separate pretty printers for output and for command lines, since backslashes need to be inserted in command lines. The special prompt ¿ is inserted by the pretty printer, and the log file reader uses the prompt to identify commands to be executed.[8]

The pretty printer is devious about the appearance of underscores in bound variables in output, and it takes special account of the behavior of parentheses, braces, brackets and commas when deciding whether to insert spaces.

There are user commands to widen or narrow the text (increment or decrement the margin parameter).

```
*)

val MARGIN = ref 40;

val TEMPMARGIN = ref 40;

val INDENT = "   ";

fun INDENTS n = if n <=0 then "\n" else (INDENTS (n-1)^INDENT);

fun INDENTS2 n = if n<0 then "\\\n" else (INDENTS2(n-1)^INDENT);

fun slen s = length(explode s);

fun prettyprint depth pos nil = [""] |
```

[8]This has the effect that the file format for this version is quite different from that of the previous version; but the previous version has a facility to export files in a format this version can read.

```
prettyprint depth pos [c] = [c] |

prettyprint depth pos ("["::L) =
(if (depth+1)*(slen INDENT) > (!TEMPMARGIN+5)
 then TEMPMARGIN := (!TEMPMARGIN)+5 else ();"["
::(prettyprint (depth+1) (pos+2) L))|

prettyprint depth pos ("]"::L) =
(if (depth+1)*(slen INDENT) <(!TEMPMARGIN-5)
andalso (!TEMPMARGIN)-5 >= (!MARGIN)
 then TEMPMARGIN:=(!TEMPMARGIN)-5 else ();
"]"::(prettyprint (depth-1)(pos+2) L)) |

prettyprint depth pos (x::"_"::y::L) =
prettyprint depth pos ((x^"_"^y)::L) |

prettyprint depth pos (".":x::L)
= prettyprint depth pos ((".^x)::L) |

prettyprint depth pos ("("::"{"::L) =

(INDENTS depth)::"("::"{"::
(prettyprint depth (depth*(length(explode INDENT))+2) L) |

prettyprint depth pos (c::"("::"{"::L) =

c::(INDENTS depth)::"("::"{"::
(prettyprint depth (depth*(length(explode INDENT))+2) L) |

prettyprint depth pos (c::L) =

if slen c = 1
 then c::(prettyprint depth (pos+2) (L))

      else if pos+slen c+1 > (!TEMPMARGIN)
  andalso L <> nil
  andalso not (find0 (Hd "\n" L) [",",")","]","}"])
```

```
  andalso not (find0 c ["(","{","["])

 then c::(INDENTS depth)::
(prettyprint depth (depth*(length(explode INDENT))) (L))

else c::(prettyprint depth (pos+slen c+1) (L));


fun prettyprint2 depth pos nil = [""] |

prettyprint2 depth pos [c] = [c] |

prettyprint2 depth pos ("["::L) =
(if (depth+1)*(slen INDENT) > (!TEMPMARGIN+5)
 then TEMPMARGIN := (!TEMPMARGIN)+5 else ();"["
::(prettyprint2 (depth+1) (pos+2) L))|

prettyprint2 depth pos ("]"::L) =
(if (depth+1)*(slen INDENT) <(!TEMPMARGIN-5)
andalso (!TEMPMARGIN)-5 >= (!MARGIN)
 then TEMPMARGIN:=(!TEMPMARGIN)-5 else ();
"]"::(prettyprint2 (depth-1)(pos+2) L)) |

prettyprint2 depth pos (x::"_"::y::L) =
prettyprint2 depth pos ((x^"_"^y)::L) |

prettyprint2 depth pos (".":x::L)
= prettyprint2 depth pos ((."^x)::L) |

prettyprint2 depth pos ("("::"{"::L) =

(INDENTS depth)::"("::"{"::
(prettyprint2 depth (depth*(length(explode INDENT))+2) L) |

prettyprint2 depth pos (c::"("::"{"::L) =

c::(INDENTS depth)::"("::"{"::
(prettyprint2 depth (depth*(length(explode INDENT))+2) L) |
```

```
prettyprint2 depth pos (c::L) =

if slen c = 1
 then c::(prettyprint2 depth (pos+2) (L))

      else if pos+slen c+1 > (!TEMPMARGIN)
  andalso L<>nil
  andalso not (find0 (Hd "\n" L) [",",")","]","}"])
  andalso not (find0 c ["(","{","["])

 then c::(INDENTS2 depth)::
(prettyprint2 depth (depth*(length(explode INDENT))) (L))

else c::(prettyprint2 depth (pos+slen c+1) (L));




fun reprocess n s = (TEMPMARGIN:=(!MARGIN);
(INDENTS n)^(displaytokens
(prettyprint n (n*(slen INDENT)) (Tokenize s))))

fun reprocess2 n s = ((TEMPMARGIN):=(!MARGIN);
(* ">>> \\"^*) (INDENTS n)^(displaytokens(
prettyprint2 n (n*(slen INDENT)) (Tokenize s))))

(*
```

7 Features of identifiers

The identifiers `prop`, `type`, `obj`, `in`, `that`, `=>`, `---`, `+++`, `>>>`, `begin`, `end`, `-->` are reserved. We refer to non-reserved identifiers as *names*. It is worth noting that it does not seem necessary to reserve the names of Lestrade commands, though this may eventually seem advisable.

Each name has some limited information about its arguments which is supplied independently of full type declarations. The base information is a format list for its arguments, identifying them as explicit, implicit and let terms. From this, two different flavors of arity can be computed. For the parser, only the number of explicit arguments counts (the arity in the following discussion). The display function uses the format of a declare operator to determine which arguments of the operator are implicit and so should be concealed.

It turned out to be crucially important for tolerable speed of execution that the functions computing dependencies on identifiers of terms be cached!

It is possible that we may install operator precedence in this version, since the type system is largely decoupled from the parser and display functions.

Basic information about dependences of terms on identifiers and detection of error terms is detected by functions given below. Functions are provided to determine identifiers in the next move on which a term depends, to catch errors in the innards of a term, and to compute the full argument list of a declared function from its list of explicit arguments.

The implicit argument list construction reads through the next move and produces a list, in the order of appearance in the next move (the order of the explicit arguments may automatically be changed) including the values which appear as dependencies of explicit arguments, the explicit arguments themselves, and defined values which appear as dependencies of the output. Variable implicit arguments must be dependencies of an explicit argument. Let terms can be dependencies of the output.

5/18/2021 adding new reserved symbol `-->` used in the representation of rewrites.

`*)`

```
fun isreserved s =
find0 s
["prop", "type","obj", "in", "that","=>",
"---", "+++",">>>","begin","end","-->"];


     (* development of dependencies on free variables *)



fun objectdeplist (Oatom(s,m,n,b)) =

if n<>0 then nil else

let  val A = findfancy [("",~1,~1)] (s,m) DEPLISTCACHE in

if A <> [("",~1,~1)] then A else

 let val T = (if n=0 then  add0 (s,m,n) (functiondeplist
(absolutesortof s m))  else nil) in

(if n=0 then DEPLISTCACHE:=
((s,m),T)::(!DEPLISTCACHE) else ();T) end end

  |

objectdeplist (App(s,m,n,b,nil)) =
let  val A = findfancy [("",~1,~1)] (s,m) DEPLISTCACHE in

if n<>0 then nil else

if A <> [("",~1,~1)] then A else

 let val T = (if n=0 then  add0 (s,m,n) (functiondeplist
(absolutesortof s m))  else nil) in

(if n=0 then DEPLISTCACHE:=((s,m),T)::
(!DEPLISTCACHE) else ();T) end end
```

```
   |

objectdeplist (App(s,m,n,b,(x::L))) =

merge0 (functiondeplist x)
(objectdeplist(App(s,m,n,b,L))) |

objectdeplist x = nil

and typedeplist (THAT p) = objectdeplist p |

typedeplist (IN tau) = objectdeplist tau |

typedeplist(Rewrite(t,u)) =

merge0(objectdeplist t)
(objectdeplist u)


|

typedeplist x = nil

and functiondeplist (Lambda(nil,t,T)) =

merge0 (objectdeplist t)(typedeplist T) |

functiondeplist (Lambda(((s,(i,m,n,b,TT))::B),t,T)) =

merge0 (functiondeplist TT)
(((drop0 (s,m,n)))
(functiondeplist(Lambda(B,t,T)))) |

functiondeplist (Fatom(s,m,n,b)) =
if n<>0 then nil else

let  val A = findfancy [("",~1,~1)] (s,m) DEPLISTCACHE in
```

```
if A <> [("",~1,~1)] then A else

 let val T = (if n=0 then  add0 (s,m,n) (functiondeplist
(absolutesortof s m))  else nil) in

(if n=0 then DEPLISTCACHE:=((s,m),T)::
(!DEPLISTCACHE) else ();T) end end
 |

functiondeplist L = nil;

fun objecterrorsfound Oerror = true |

objecterrorsfound (App(s,m,n,b,L)) =

map (functionerrorsfound) L <> map (fn x => false) L |

objecterrorsfound x = false

and typeerrorsfound Terror = true |

typeerrorsfound (THAT p) = objecterrorsfound p |

typeerrorsfound (IN tau) = objecterrorsfound tau |

typeerrorsfound (Rewrite(t,u)) =
objecterrorsfound t orelse objecterrorsfound u |

typeerrorsfound x = false

and functionerrorsfound Ferror = true |

functionerrorsfound (Lambda (nil,t,T)) =
objecterrorsfound t
orelse typeerrorsfound T |

functionerrorsfound (Lambda(((s,(i,m,n,b,TT))::M),t,T))
```

```
= functionerrorsfound TT orelse functionerrorsfound (Lambda(M,t,T)) |

functionerrorsfound x = false;

fun objectoccurrences s m n (Oatom(S,M,N,B)) =

(if s=S andalso m=M andalso N=n then 1 else 0)+(if n=0 then
(functionoccurrences s m n (absolutesortof s m)) else 0)|

objectoccurrences s m n (App(S,M,N,B,nil)) =

(if s=S andalso m=M andalso N=n then 1 else 0)+(if n=0 then
(functionoccurrences s m n (absolutesortof s m)) else 0) |

objectoccurrences s m n (App(S,M,N,B,(x::L))) =
objectoccurrences s m n (App(S,M,N,B,L)) + functionoccurrences s m n x |

objectoccurrences s m n x = 0

and functionoccurrences s m n (Fatom(S,M,N,B)) =
(if s=S andalso m=M andalso N=n then 1 else 0)+(if n=0 then
(functionoccurrences s m n (absolutesortof s m)) else 0) |

functionoccurrences s m n (Lambda(nil,t,T)) =

objectoccurrences s m n t + typeoccurrences s m n T |

functionoccurrences s m n (Lambda((S,(i,M,N,B,TT))::BB,t,T)) =

 if s=S andalso m =M andalso N=n then
functionoccurrences s m n TT

else   max(functionoccurrences s m n TT,
functionoccurrences s m n (Lambda(BB,t,T))) |

functionoccurrences s m n x = 0

and typeoccurrences s m n (THAT p) = objectoccurrences s m n p |
```

```
typeoccurrences s m n (IN p) = objectoccurrences s m n p |

typeoccurrences s m n (Rewrite(t,u)) =
(objectoccurrences s m n t) +
(objectoccurrences s m n u) |

typeoccurrences s m n x = 0;

fun  maxdepthobject (Oatom(s,m,n,b)) = n |

maxdepthobject (App(s,m,n,b,nil)) = n |

maxdepthobject (App(s,m,n,b,x::L)) =
max(maxdepthobject (App(s,m,n,b,L)),maxdepthfunction x) |

maxdepthobject x = 0

and  maxdepthtype (THAT p) = maxdepthobject p |

maxdepthtype (IN p) = maxdepthobject p |

maxdepthtype (Rewrite(t,u)) =
max(maxdepthobject t,maxdepthobject u) |

maxdepthtype x = 0

and  maxdepthfunction (Lambda(nil,t,T)) =

max(maxdepthobject t,maxdepthtype T) |

maxdepthfunction (Lambda((s,(i,m,n,b,TT))::B,t,T)) =

max(n,max(maxdepthfunction TT,maxdepthfunction (Lambda(B,t,T)))) |

maxdepthfunction (Fatom(s,m,n,b)) = n |

maxdepthfunction x = 0;
```

```
(* this says that a function term is not a type. *)

fun islambda2 (Lambda(B,t,T)) = t<>Unknown |

islambda2 x = false;

(* generate the implicit argument list from the explicit argument list *)

fun implicitarglist1 (Lambda(B,t,T)) nil = nil |

implicitarglist1 (Lambda(B,t,T)) ((s,(i,m,n,b,TT))::L) =

if find0 (s,m,n) (functiondeplist(Lambda(B,Unknown,Tdummy)))
orelse (find0 (s,m,n) (functiondeplist(Lambda(nil,t,T)))
andalso islambda2 (absolutesortof s m))
orelse find0 (s,m,n) (map (fn (s,(i,m,n,b,TT)) => (s,m,n)) B)

then (s,(i,m,n,find0 (s,m,n) (map (fn (s,(i,m,n,b,TT)) => (s,m,n)) B) ,TT))
::(implicitarglist1 (Lambda(B,t,T)) L)

else implicitarglist1 (Lambda(B,t,T)) L |

implicitarglist1 x y = nil;

fun implicitarglist (Lambda(B,t,T)) =

Lambda(rev(implicitarglist1 (Lambda(B,t,T)) (Hd nil (!CONTEXT))),t,T) |

implicitarglist x = (say "implicitarglist failure line 1905";Ferror);

(*
```

8 Object, function, and sort terms of the input language

An open argument list is a sequence of names and commas which starts with a name and ends with a name, and in which each non-final name of positive arity is followed by a comma and each non-initial name of arity greater than one is preceded by a comma, and in which no comma is followed by another comma. Additional commas between names are allowed.

A closed argument list is an open parenthesis `(` followed by an open argument list followed by a close parenthesis `)`.

An open term list is a sequence of function terms and commas in which each non-final name of positive arity serving as a function term in the sequence is followed by a comma and each name which is of arity greater than one and initial in one of the non-initial function terms in the sequence is preceded by a comma, and in which no comma is followed by another comma. Additional commas between function terms are allowed.

The final term of an open term list is not a mixfix term unless it is followed by a non-identifier or end of text.

A closed term list is an open parenthesis `(` followed by an open term list followed by a close parenthesis `)`.

The length of an open or closed term list is the number of function terms in it (commas are not counted).

An object term is

name: a name of arity 0

applicative: or a name of positive arity followed by a closed term list of length equal to its arity, or by an open term list of length equal to its arity whose first token is not a open parenthesis `(`.

The meaning of an applicative term depends only on the list of function terms appearing in its term list, not on whether it is open or closed.

parenthesized: or an open parenthesis `(` followed by an object term followed by a close parenthesis `)`,

The meaning of a parenthesized term is the same as the meaning of the term enclosed in the parentheses.

mixfix: or a non-mixfix object term followed by a name of arity greater than 1 followed by an open term list of length one less than the arity of the name used as a mixfix.

(Implicit in this description is that all names when used as mixfixes are of the same precedence and group to the rignt.)

The meaning of a mixfix term is the same as the meaning of the applicative term beginning with the name used as the mixfix, followed by `, followed by the initial object term, followed by a comma, fo` the intial open term were parenthesized, or if the final open term list began with a name of arity greater than one).

A function term is

1. an object term,
2. or a name of positive arity,
3. or a name of arity greater than 1 followed by a closed term list of length less than the arity of the initial name,
4. or an open bracket `[` followed by an open argument list (in which commas are completely optional) followed by the reserved identifier `=>` followed by an object term followed by a close bracket `]` .

An object sort term is of one of the following shapes:

1. a single token, which must be `prop` `type` or `obj`;
2. one of the tokens `that` or `in` followed by an object term.
3. a opening brace followed by an object term followed by `-->` followed by an object term followed by a closing brace

A function sort term consists of an open bracket `[` followed by an open argument list (in which commas are completely optional) followed by the reserved identifier `=>` followed by an object sort term followed by a close bracket `]`.

The merit of this description is that it separates out the quite minimal information about type declarations needed for parsing, and our intention in this implementation is to have the declaration mechanism record information needed by the parser separately. This does mean that we establish the intention to fill in implicit arguments at a post-processing step, not at parse time as in the original implementation.

We will have uses (as in the existing code) for terms which do not type correctly. In this version, we expect the parser to output terms which are not necessarily correctly typed and pass them to a utility which adds implicit arguments. The mechanism for deducing implicit arguments from explicitly given argument lists will involve (as it already does) nonce construction of ill-typed object sort terms.

The display functions and some discussion of additional features of the output notation appear before the parser code.

We note that there are some differences between the terms accepted by the parser of the original version and those accepted here. These led to a few revisions of the Zermelo implementation files. This version supports numerical suffixes to special character identifiers, so whitespace has to appear between a special character identifier and a following numeral. This version does not allow the last argument of a prefix term with its arguments not enclosed in parentheses to be a mixfix term: the previous version enforced this only for unary prefix terms.

The most distracting feature of the parser in our experience is that the first argument of a prefix term with more than one argument cannot be enclosed in parentheses unless the entire argument list is also enclosed in parentheses: this is true in both versions.

```
*)



(* this function computes I/O format
from function terms *)

fun lambdaformat (Lambda(nil,t,T)) = nil |

lambdaformat (Lambda((s,(i,m,n,b,TT))::B,t,T)) =

    (if islambda2 TT then LetTerm
else if b then Explicit
else Implicit)::
(lambdaformat (Lambda(B,t,T))) |

lambdaformat x = [Implicit];

(* these functions completely replace appeals
to arity and format lists which are no longer
maintained *)

fun arity0 s m = formattoarity1 (lambdaformat(absolutesortof s m));

fun Arity2 s m = formattoarity2 (lambdaformat(absolutesortof s m));

fun arity1 s L = if L = nil then ~1 else
let val A = arity0 s (length L - 1) in
if A = ~1 then arity1 s (Tl L) else A end;

fun arity s = arity1 s (!CONTEXT);

(* does an object term represent an object? *)

(* Unknown does not, and curried function terms do not *)

(* this is an *internal* term representing an object *)
```

```
fun isobject0 (App(s,m,n,b,L)) = n<>0 orelse length L = Arity2 s m |

isobject0 Unknown = false |

isobject0 Oerror = false |

isobject0 x = true;

(* does a function term represent an
object? *)

(* this is a very quick and dirty test:  there are other
terms (0-ary functions with let term arguments) which
represent objects. *)

fun isobject (Lambda(nil,t,T)) = isobject0 t |

isobject x = false;

(* this says that a formally function
term actually stands for an object
or object sort *)

fun isobjecttype (Lambda(B,t,T)) =

map(fn (s,(i,m,n,b,TT)) => islambda2 TT) B =
map (fn x=>true) B  andalso (t = Unknown orelse isobject0 t)|

isobjecttype x = false;

(* repaired versions of islambda0
and islambda.  I am not sure that they
won't cause trouble. *)

(* this identifies lambda terms representing
objects *)
```

```
fun islambda0 (Lambda(B,t,T)) =
t<>Unknown andalso isobjecttype (Lambda(B,t,T)) |
islambda0 x = false;

(* this identifies lambda terms representing
functions *)

fun islambda (Lambda(B,t,T)) =
t <> Unknown andalso T <> Tdummy andalso not(islambda0 (Lambda(B,t,T))) |

islambda x = false;

fun isoatom (Oatom(s,m,n,b)) = true |

isoatom x = false;

fun isapp (App(s,m,n,b,L)) = true |

isapp x = false;

fun applist (App(s,m,n,b,L)) = L |

applist x = (say "Bad applist 2580";[Ferror]);

fun isfatom (Fatom(s,m,n,b)) = true |

isfatom x = false;

fun islambdaterm (Lambda(B,t,T)) = true|

islambdaterm x = false;

fun bindersof (Lambda(B,t,T)) = B |

bindersof x = nil;

fun bindersof2 (Lambda((s,(i,m,n,b,TT))::B,t,T)) = TT |
```

```
bindersof2 x = (say "Bad bindersof2 2596";Ferror);

fun lambdatail (Lambda((s,(i,m,n,b,TT))::B,t,T)) = Lambda(B,t,T) |

lambdatail x = x;

(* The development of the display function starts here *)

val ERRORDISPLAY = ref false;

fun formatmask [Implicit] L =
(say "Format masking failure line 2093";[Ferror]) |

formatmask x nil = nil |  (* handles curried terms *)

formatmask (LetTerm::L) M = formatmask L M |

formatmask nil (x::L) =
(say "format masking failure line 2101";[Ferror]) |

formatmask (Explicit::L) (x::M) = x::(formatmask L M) |

formatmask (Implicit::L) (x::M) = formatmask L M |

formatmask x y = (say "format masking failure line 2107";[Ferror]);

fun displayatom(s,m,n,b) =(if b then "" else ".")^
 s^(if n=0 then ""
 else ((* "_"^(makestring m)^ *) "_"^(makestring n)));

(* this is designed to be updatable when we work out something to handle
masking:  there will be a context dependence eventually *)

fun suppresstype (Lambda(nil,t,T)) = Lambda(nil,t,Tdummy) |

suppresstype x = x;

(* suppress types in object arguments *)
```

```
fun parenthesize (Lambda(nil,(App(s,m,n,b,L)),Tdummy))

= if  arity s = 2 andalso length L > 1

(* andalso length L = 2
andalso L<>nil andalso isobject(Hd Ferror L) *)
then "("^(displayobject (App(s,m,n,b,L)))^")"

else displayfunction (Lambda(nil,(App(s,m,n,b,L)),Tdummy)) |

parenthesize x = displayfunction x

(* parenthesize infix terms where needed *)

and displayobject (Oatom(s,m,n,b)) = displayatom(s,m,n,b) |

displayobject (App(s,m,n,b,LL)) =

(* the use of the let clause here
is to remove implicit arguments *)

let val L = if n<>0 orelse (!ERRORDISPLAY)then LL

else formatmask (lambdaformat(absolutesortof s m)) LL

in

if  arity s = 2 andalso length L > 1 andalso (isobject(Hd Ferror L))

(* andalso length L = 2
andalso L<>nil andalso (isobject(Hd Ferror L)) *)

then (parenthesize(suppresstype(Hd Ferror L)))^
" "^s^" "^
(displayargumentlist (Tl(L))) else

(displayatom(s,m,n,b))^"("^(displayargumentlist L)^")" end |
```

```
displayobject Unknown = "---" |

displayobject Deferred = "+++" |

displayobject Oerror = "{error object}"

and displaytype PROP = "prop" |

displaytype TYPE = "type" |

displaytype OBJ = "obj" |

displaytype (THAT p) = "that "^(displayobject p) |

displaytype (IN tau) = "in "^(displayobject tau) |

displaytype (Rewrite(t,u)) =
"{"^(displayobject t)^"-->"
^(displayobject u)^"}" |

displaytype Tdummy = "{dummy type}" |

displaytype Terror = "{error type}"

and displayargumentlist nil = "" |

displayargumentlist [x] = displayfunction (suppresstype x) |

displayargumentlist (x::L) =
(displayfunction (suppresstype x))^
", "^(displayargumentlist L)

and displayfunction (Fatom(s,m,n,b)) = displayatom(s,m,n,b) |

displayfunction (Lambda (nil,t,Tdummy)) = displayobject t |

     (* do not display types for objects in argument lists *)
```

```
displayfunction (Lambda(nil,Unknown,T)) = displaytype T |

displayfunction (Lambda (L,t,T)) = "["^
(displaybinderlist L)^
(if T = Tdummy then displayobject t
else "("^(if t <> Unknown andalso t <> Deferred then "{def} " else "")^
(displayobject t)^" : "^(displaytype T)^")")^"]" |

displayfunction Fdummy = "{dummy function}" |

displayfunction Ferror = "{function error}"

and displaybinderlist nil = "" |

displaybinderlist [((s,(i,m,n,b,T)))] =

if T = Fdummy then displayatom(s,m,n,b)^" => "

else "("^(
if islambda2 T then "{let} " else ""
)^(displayatom(s,m,n,b))^
" : "^(displayfunction T)^") => "

 |

displaybinderlist (((s,(i,m,n,b,T)))::L) =

if T = Fdummy then

(displayatom(s,m,n,b))^", "^(displaybinderlist L)

else "("^(
if islambda2 T then "{let} " else ""
)^(displayatom(s,m,n,b))
^" : "^
(displayfunction T)^"), "
^(displaybinderlist L)
```

```
;

fun displayerrorfunction x = if (!DIAGNOSTIC) then (ERRORDISPLAY:= true;
let val RESULT = displayfunction x in (ERRORDISPLAY:=false;RESULT) end) else ""

fun displayerrorobject x = if (!DIAGNOSTIC) then (ERRORDISPLAY:= true;
let val RESULT = displayobject x in (ERRORDISPLAY:=false;RESULT) end) else "";

fun displayerrortype x = if (!DIAGNOSTIC) then (ERRORDISPLAY:= true;
let val RESULT = displaytype x in (ERRORDISPLAY:=false;RESULT) end) else "";

(*
```

9 Additional features of the output language

This subsection is placed here just below the display function code which implements it.

The output language will be pretty printed, with indentations which will hopefully help the reader to divine the structure of displayed terms, which will usually be function sort terms.

Displayed object terms will always contain as many parentheses (term lists will be closed) and commas as allowed. Object terms which can be input as infix (mixfix of arity 2) will always be displayed in this way (parenthesized as necessary for precedence); object terms which can be input as mixfix of higher arity will always be displayed in applicative form.

Bound variables in the bracketed object or function sort terms will always be renamed, by appending an underscore followed by a numeral (roughly[9]) representing the depth in brackets of their binding context, and further will be followed by a colon followed by their sort where they appear in the initial argument list. Implicit arguments will in addition be adorned with an initial period. A function sort term $[\mathtt{x1}, \mathtt{x2}, \mathtt{x3} => \mathtt{t}]$ will be displayed in the form $[(\mathtt{x1} : \mathtt{t1}), (\mathtt{x2} : \mathtt{t2}), (\mathtt{x3} : \mathtt{t3}) => (--- : \mathtt{t})]))$. A function term $[\mathtt{x1}, \mathtt{x2}, \mathtt{x3} => \mathtt{d}]$ will be displayed in the form

$$[(\mathtt{x1} : \mathtt{t1}), (\mathtt{x2} : \mathtt{t2}), (\mathtt{x3} : \mathtt{t3}) => (\{\mathtt{def}\}\mathtt{d} : \mathtt{t})].$$

The type information for a defined function will be displayed just as the function would be displayed (its pure type being obtained by scrubbing out the `d` and replacing it with the internal dummy object term represented by `---`) and moreover type information for defined object terms will be represented in the form `[({def} d:t)]`, as functions of arity 0, though the input language does not allow such 0-ary lambda terms.

The display considerations in the previous paragraph reflect the fact that function sorts are being treated as a subtype of general dependently typed lambda terms, the sort of a lambda term being obtained by scrubbing its body. Further, an object term is identified with the 0-ary lambda term with that value, so objects (and object sorts) can also be viewed as a subtype of general lambda terms. This subtyping is purely a matter of convenience: the underlying philosophy of Lestrade enforces a strong distinction between objects and functions and between their respective sorts.

[9]The exact way that indices on bound variables are computed is determined by syntactical depth but there are complications and possible buggy quirks.

There is a further complication in display having to do with let terms. A term displayed as $[(\texttt{x1} : \texttt{t1}), (\{\texttt{let}\}\texttt{x2} : \texttt{t2}), (\texttt{x3} : \texttt{t3}) => (\texttt{d} : \texttt{t})]$ actually represents a function of two arguments: the bound variable `x2` is to be replaced everywhere by `t2`, which we expect not to be a type but a full lambda term (when a variable in applied position is replaced by a lambda term, beta reduction takes place).

In the examples above we did not append underscores followed by numerals to bound variables. What should be noted here is that an expression $[(\texttt{x1} : \texttt{t1}), (\texttt{x2} : \texttt{t2}), \ldots, (\texttt{xn} : \texttt{tn}) => (\texttt{d} : \texttt{t})]$ is dependently typed: each x_i may appear in t_j for $j > i$ and of course in d, t. Lestrade attaches the same underscored numeral index to each of the x_i's to signal the context in which they are bound, the index being determined (roughly) by syntactical depth[10].

[10]The handling of these indices was quite different in the previous version, and led to very large indices on occasion.

```
*)

(* Here we should put the basic parser *)

(* the parser functions take a list of tokens and output
the appropriate term read from the front *)

(* parse atom innards using the arity list *)

fun parseatom1 s L = if L = nil then ("",0,0,true)
else let val A = arity0 s (length L -1) in
if A = ~1 then parseatom1 s (Tl L) else (s,length L -1,0,true) end

fun parseatom s = parseatom1 s (!CONTEXT);

fun getobject1 L = p1(getobject1E L)

and restobject1 L = p2(getobject1E L)

and getobject1E nil = (Oerror,nil) |

getobject1E (s::L) =

if s = "("

then let val R = getobjectE L in

if p2 R <> nil andalso Hd "\n" (p2 R) = ")"

then (p1 R,Tl(p2 R))

else (Oerror,s::L) end


   else
       if arity s = 0
```

```
   then (Oatom(parseatom s),L)

       else if arity s > 0

      then (let val SS = parseatom s in
           if SS = ("",0,0,true)

       then (Oerror,(s::L))
   else (let val (S,M,N,b) = SS
                                  and LL = getarglistE true true (arity s) L
              in if p1 LL = [Ferror] then (Oerror,(s::L)) else

             (App(S,M,N,b,p1 LL),p2 LL) end) end)

        else (Oerror,(s::L))



and getobject L = p1(getobjectE L)

and restobject L = p2(getobjectE L)

and getobjectE L =

let val (TT,LL) = getobject1E L in

if TT = Oerror then (Oerror,L)

else if LL = nil then (TT,LL)

else if arity (Hd "\n" LL) < 2 then (TT,LL)

else let val (S,M,N,b) = parseatom(Hd "\n" LL) in

let val LLL = getarglistE false false (arity(Hd "\n" LL) - 1) (Tl LL) in

if p1 LLL = [Ferror] then (Oerror,L)
```

```
else (App(S,M,N,b,
Lambda(nil,TT,Tdummy)::(p1 LLL)),p2 LLL) end

end

end



and gettype L = p1(gettypeE L)

and resttype L = p2(gettypeE L)

and gettypeE nil = (Terror,nil) |

gettypeE (s::L) = if s = "prop" then (PROP,L)

else if s = "type" then (TYPE,L)

else if s = "obj" then (OBJ,L)

else if s = "that" then
let val (TT,LL) = getobjectE L in
if TT = Oerror then (Terror,s::L)
else (THAT TT,LL) end

else if s = "in" then
let val (TT,LL) = getobjectE L in
if TT = Oerror then (Terror,s::L)
else (IN TT,LL) end

else if s = "{" then
let val (TT1,LL1) = getobjectE L in (* 2 *)

if TT1 = Oerror orelse  LL1 = nil orelse hd LL1 <> "-->"

then (Terror,s::L)
```

```
else let val (TT2,LL2) = getobjectE(tl LL1) in  (* 1 *)

if TT2 = Oerror orelse LL2 = nil orelse hd LL2 <> "}"

then (Terror,s::L)

else (Rewrite(TT1,TT2),tl LL2)

end (* 1 *)

end (* 2 *)

else (Terror,s::L)

and getfunction L = p1(getfunctionE L)

and restfunction L = p2(getfunctionE L)

and getfunctionE nil = (Ferror,nil) |

getfunctionE (s::L) =

(* object type *)

if s="prop" orelse s = "type" orelse s = "obj"
orelse s = "that" orelse s = "in"  orelse s = "{"

then let val T = gettypeE (s::L) in

if p1 T = Terror then (Ferror,s::L)

else (Lambda(nil,Unknown,p1 T),p2 T)

end

else

(* function name *)
```

```
if arity s > 0 andalso (L = nil
orelse (Hd "\n" L <> "(" andalso Hd "\n"  L <> "["
andalso arity (Hd "\n"  L) = ~1))
then (Fatom (parseatom s),L)

(* object term packaged as function;
may be ill-typed if it is a curried
function, fixed on a later pass *)

else if s = "(" orelse arity s <> ~1 then

let val T = getobjectE(s::L) in

if p1 T = Oerror then (Ferror,s::L)

else  (Lambda(nil,p1 T,Tdummy),p2 T)

end

(* lambda term in user input
format *)

else if s = "[" then let val (BB,LL1) = getbindersE L
in

if BB= nil orelse BB = [("",(0,0,0,true,Ferror))]
orelse LL1 = nil orelse Hd "\n" LL1 <> "=>" then (Ferror,s::L)

else let val (TT,LL2) = getobjectE(Tl LL1)

in

if TT = Oerror orelse LL2 = nil orelse Hd "\n" LL2 <> "]"

then let val (TT2,LL3) = gettypeE (Tl LL1)

in if TT2 = Terror orelse LL3 = nil orelse Hd "\n" LL3 <> "]"
```

```
then (Ferror,s::L)

else (Lambda(BB,Unknown,TT2),Tl LL3)

end

else (Lambda(BB,TT,Tdummy),Tl LL2)

end

end

else (Ferror,s::L)


(* doesnt take mixfix object terms *)

and getfunction1 L = p1(getfunction1E L)

and restfunction1 L = p2(getfunction1E L)

and getfunction1E nil = (Ferror,nil) |

getfunction1E (s::L) =

(* object type *)

if s="prop" orelse s = "type" orelse s = "obj"
orelse s = "that" orelse s = "in" orelse s = "{"

then let val T = gettypeE (s::L) in

if p1 T = Terror then (Ferror,s::L)

else (Lambda(nil,Unknown,p1 T),p2 T)
```

```
end

else

(* function name *)

if arity s > 0 andalso (L = nil
orelse (Hd "\n" L <> "(" andalso Hd "\n" L <> "["
andalso arity (Hd "\n" L) = ~1))
then (Fatom (parseatom s),L)

(* object term packaged as function;
may be ill-typed if it is a curried
function, fixed on a later pass *)

else if s = "(" orelse arity s <> ~1 then

let val T = getobject1E (s::L) in

if p1 T = Oerror then (Ferror,s::L)

else  (Lambda(nil,p1 T,Tdummy),p2 T)

end

(* lambda term in user input
format *)

else if s = "[" then let val (BB,LL1) = getbindersE L
in

if BB= nil orelse BB = [("",(0,0,0,true,Ferror))]
orelse LL1 = nil orelse Hd "\n" LL1 <> "=>" then (Ferror,s::L)

else let val (TT,LL2) = getobjectE(Tl LL1)

in
```

```
if TT = Oerror orelse LL2 = nil orelse Hd "\n" LL2 <> "]"

then let val (TT2,LL3) = gettypeE (Tl LL1)

in if TT2 = Terror orelse LL3 = nil orelse Hd "\n" LL3 <> "]"

then (Ferror,s::L)

else (Lambda(BB,Unknown,TT2),Tl LL3)

end

else (Lambda(BB,TT,Tdummy),Tl LL2)

end

end

else (Ferror,s::L)

and getarglist closable last1 n L = p1(getarglistE closable last1 n L)

and restarglist closable last1 n L = p2(getarglistE closable last1 n L)

and getarglistE closable last1 0 L = (nil,L) |

getarglistE closable last1 n nil =
 if n>0 then ([Ferror],nil) else (nil,nil) |

getarglistE closable last1 n (s::L) =

if closable andalso s = "(" andalso n > 0

then let val (LL,RR) = getarglistE false false ~1 L
in

if RR = nil orelse Hd "\n" RR <> ")"
    orelse length LL > n then ([Ferror],s::L)
```

```
else (LL,Tl RR)

end

else let val (TT,RR) = if n=1  andalso last1
then getfunction1E (s::L) else getfunctionE(s::L)

in

let val RRR = if  n<>1 andalso  RR <> nil
andalso Hd "\n" RR = ","
andalso RR <> [","] andalso Hd "\n" (Tl RR) <> ")"
then Tl RR
else RR
in
if TT = Ferror then if n <0
then (nil,s::L) else ([Ferror],s::L)
else let val LL = getarglistE false last1 (n-1) RRR in

if p1 LL = [Ferror] then ([Ferror],s::L)

else (TT::p1 LL,p2 LL)

end

end

end



and getbinders L = p1(getbindersE L)

and restbinders L = p2(getbindersE L)

and getbindersE nil = (nil,nil) |
```

```
getbindersE (","::(s::L)) = getbindersE (s::L) |

getbindersE (s::L) = if arity0 s
   (length (!CONTEXT) -1) <> ~1 then
let val (S,M,N,b) = parseatom s

and LL = getbindersE L in

(* I could impose the usual use of parameters
from the next move *)

if  M <> length(!CONTEXT)-1 orelse

find0 (S,
(1,M,N,true,if N=0
then absolutesortof S M else Fdummy)) (p1 LL )
orelse p1 LL = [("",(0,0,0,true,Ferror))] then

([("",(0,0,0,true,Ferror))],s::L)

else ((S,(1,M,N,true,if N=0
then absolutesortof S M else Fdummy))::(p1 LL),p2 LL)

end

else (nil,s::L);



fun Parseobject() = let val TEXT = getline() in

(output(stdOut,displaytokens (Tokenize TEXT));
output(stdOut,displayobject(getobject(Tokenize TEXT)));
output(stdOut,"\n\n");
getobject(Tokenize TEXT)) end;

fun Parsetype() = let val TEXT = getline() in
```

```
(output(stdOut,displaytokens (Tokenize TEXT));
output(stdOut,displaytype(gettype(Tokenize TEXT)));
output(stdOut,"\n\n");
gettype(Tokenize TEXT)) end;

fun Parsefunction() = let val TEXT = getline() in

(output(stdOut,displaytokens (Tokenize TEXT));
output(stdOut,displayfunction(getfunction(Tokenize TEXT)));
output(stdOut,"\n\n");
getfunction(Tokenize TEXT)) end;

(*
```

Above find the parser code which implements the description given above of the term language.

The functions `Parseobject()`, `Parsetype()`, `Parsefunction()` have the same minimal interface as `Getline()` and in addition both display and return parsed versions of an initial segment of their input.

10 The Lestrade type system

The world of Lestrade is inhabited by objects and functions. Objects and functions have sorts. Objects can for some purposes be identifed with 0-ary functions ($x() = x$).

The sorts of objects are restricted to the following:

1. `prop`, the type of propositions;
2. `type`, the sort of types of mathematical objects (we think of objects of sort τ as type labels);
3. `obj`, a convenient sort to be inhabited by "untyped" mathematical objects (as for example in set theory, in which we propose to implement all types of objects without actually assigning types);
4. `that` p, where p is any term of sort `prop`, the sort of proofs of or evidence for p;
5. `in` τ, where τ is any term of sort `type`, the sort inhabited by objects of the type labelled by τ.

The functions of Lestrade are dependently typed, with object and function inputs and object outputs.

A Lestrade function sort is represented by an abstract notation of the form $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau]$, in which τ is an object sort and each x_i is a variable of object or function sort τ_i which is bound in the given sort term and satisfies the constraint that x_i can only occur in τ_j with $j > i$ or in τ.

If f is of sort $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau]$, then a term $f(t_1, \ldots, t_n)$ is type checked as follows: if $n = 1$, this term type checks and has type τ iff the type of t_1 is τ_1; otherwise let f^* be of sort $[(x_2, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau][t_1/x_1]$, the result of substituting t_1 for x_1 in $[(x_2, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau]$, and the type of $f(t_1, \ldots, t_n)$ will be the type of $f^*(t_2, \ldots, t_n)$, if this is well-typed, and otherwise ill-typed.

A defined term of function type (a lambda term) is represented by an abstract notation of the form $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow D]$. This notation has sort $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau]$ if assignment of type τ_i to each x_i in turn succeeds (in the sense that each τ_i types correctly after the typing of x_j's with $j < i$) and further D is assigned sort τ after these assignments of sorts to the x_i's; otherwise it is ill-typed.

The result of substituting $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow D]$ for f in a term $f(t_1, \ldots, t_n)$ is (if $n = 1$) $D[t_1^*/x_1]$ and otherwise is the result of replacing f^* with $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow D][t_1/x_1]$ in $f^*(t_2^*, \ldots, t_n^*)$, where each t_i^* is the result of making the same substitution in t_i. It is important to notice that in an abstract applicative term $f(t_1, \ldots, t_n)$, f is always an atomic term: beta reduction is carried out as part of the substitution process.

The fact that the x_i's are bound in terms $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau]$ or $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow D]$ can be handled by the convention that each x_i should be replaced with a fresh variable x_i^* before any substitution is made. Other clauses of the definition of substitution are entirely natural and are left to the reader.

The internal representation of function sorts and lambda terms in Lestrade differs a little from this account in ways suggested by the remarks on output notation above. The internal notation for $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow \tau]$ is of the form $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (\circ : \tau)]$ where $\circ$ is a dummy object term, and the internal notation for $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow D]$ is $[(x_1, \tau_1), \ldots, (x_n, \tau_n) \Rightarrow (D : \tau)]$ with the type incorporated. Further, there are 0-ary terms $[(\circ, \tau)]$ synonymous with object type τ and $[(D : \tau)]$ synonymous with the object term D of type τ.

A further refinement in Lestrade's internal notation (and its output notation, as noted above) is the presence of let notation: if a τ_i is a lambda term rather than a type, the meaning of a function sort or lambda term T with an argument (x_i, τ_i) is the same as the meaning of $T^*[\tau_i/x_i]$, where T^* is the lambda term or function sort obtained by deleting the formal argument (x_i, τ_i) from T. Note that object terms/sorts can have let components, but only if they are presented as 0-ary lambda terms/function sorts. All formal arguments of this kind should be expanded before computing types or carrying out beta reductions. This is new in this implementation, and I do not know all the pitfalls yet!

If f has type $[(x_1, \tau_1), \ldots, (x_i, \tau_i), (x_{i+1}, \tau_{i+1}), \ldots, (x_n, \tau_n) \Rightarrow \tau]$, then $f(t_1, \ldots, t_i)$ is an alternative notation for $[(x_{i+1}, \tau_{i+1}), \ldots, (x_n, \tau_n) \Rightarrow f(t_1, \ldots, t_i)]$, if this is well-typed: this is a version of currying. It is important to note that an object term is here being used to represent a non-object.

The abstract notation we are implicitly describing here contains exactly the constructions given and atomic free variables and bound variables of all sorts.

10.1 Substitution and beta-reduction

The code which follows first implements substitution then the type system.

It is worth noting that substitution for an atom in applied position causes beta reduction: lambda terms occur in our notation, but not in applied position. Further, it is not possible for an atom to replace an atom (object or function) whose sort depends on that atom[11]. This plays an important role in enforcing proper dependency structure in lambda terms.

The scheme of variable binding needs some discussion to assure soundness. Our intention is that all free variables have depth index 0. All substitutions are of a term T for a variable v: the term T either has all free variables at depth 0 or is a variable of arbitrary depth replacing bound variables at that depth in the course of bound variable renaming.

The idea is that when one substitutes t for variable v in a term with bound variables at depth d, one first replaces all the bound variables in the target term with variables at depth $d + 1$, then makes substitutions for the bound variables as well as the main substitution in the components at depth of higher index that that of any variable in t, in the components of the bound variable term. Substitutions are always to be made at depth lower than the depth of the variable v. In this way, bound variable capture is always avoided.

There is a general feature of the Type Inspector which should be noted: it avoids manipulation of the innards of terms with bound variables. There is no declaration information associated with bound variables except indirectly via types in binder lists, and appeals to this information are avoided. When a term is entered, all of its bound variables are actually free and have declaration information associated with them in the current context, which is used in type checking the term before it is put in the usual format with bound variables indexed by depth. When information is wanted about innards of a lambda term, the usual strategy is to evaluate it at an input. The theory about terms with standard variable binding is that they have all been generated and type checked earlier and can be relied upon to be structured correctly. For this reason, beta reduction actually does no type checking. The analogous operation for function sorts below does do type checking, of course.

[11]This remains true, but it is no longer throws an error; it simply doesn't happen.

```
*)

val STRONGEXPAND = ref true;

fun Strongexpand() = STRONGEXPAND := not(!STRONGEXPAND);

fun isbadapp (Lambda(nil,App(s,m,n,b,L),Tdummy)) = true |

isbadapp x = false;

fun badappbody (Lambda(nil,App(s,m,n,b,L),Tdummy)) = App(s,m,n,b,L) |

badappbody x = Oerror;

fun isanobject s m = islambda0(absolutesortof s m);

fun isafunction s m = islambda(absolutesortof s m);

val AFUNCTION = ref [Ferror];

val ALIST = ref [[Ferror]];

(* eta reduction, needed for equality below *)

fun etared depth s m  b =

if not(islambdaterm(absolutesortof s m) )

then ("Bad etared 3271";Ferror)

else let val Lambda(B,t,T)  = absolutesortof s m in

Lambda(B,App(s,m,0,b,map(fn (S,(I,M,N,bb,TT)) =>
referentof(S,M,N,bb,TT))
(drop "" (map (fn(S,(I,M,N,B,TT)) => (if islambda2 TT then ""
 else  S,(I,M,depth+1,B,TT))) B))),T)
```

```
end

and Etared depth (Fatom(s,m,0,b)) =

if islambdaterm (absolutesortof s m) then etared depth s m b

else Fatom(s,m,0,b) |

Etared depth x = x

(* substitution:  we substitute a function value for an atom,
with appropriate expansions
as required.  Of course, the function may be 0-ary,
representing an object.  We should still
require no more declaration information than the parser does. *)

(* handle bound variable renaming
by providing syntactic depth as a parameter *)

(* this computes the object represented by
a function term, if there is such an object *)

and objectof depth (Lambda(nil,t,T)) = t |

objectof depth  (Lambda((s,(i,m,n,b,T))::B,tt,TT)) =

if islambda2 T then

objectof depth (Subsfunction0  depth s m n b T
   (Lambda(B,tt,TT))) else (say ("objectof a function 3308: "
^(displayfunction
(Lambda((s,(i,m,n,b,T))::B,tt,TT))));Oerror) |

objectof depth x = (say ("objectof a non object line 2749: "
^(displayfunction x));Oerror)

and theobjectsort(Lambda(nil,t,T)) =
```

```
if t = Unknown orelse T<>Tdummy then T
else (say ("theobjectsort inapplicable line 3066:  "
^(displayfunction(Lambda(nil,t,T))));Terror) |

theobjectsort (Lambda((s,(i,m,n,b,TT))::B,t,T)) =

if islambda2 TT then

theobjectsort (subsfunction0 (n+1)  s m n b TT
(Lambda(B,t,T)))

else (say "theobjectsort inapplicable line 3075";Terror) |

theobjectsort x = (say "theobjectsort inapplicable line 3077";Terror)

and  referentof(s,m,n,b,T) = if isobjecttype T
then Lambda(nil,Oatom(s,m,n,b),theobjectsort T)
else Fatom(s,m,n,b)

and

(* count non-let term arguments *)

binderlength nil = 0 |

binderlength ((s,(i,m,n,b,t))::L) =
if not(islambda2 t) then (binderlength L) + 1
else binderlength L

and betared depth x L =

if L = nil then x

else if isbadapp x then

let val App(s0,m0,n0,b,M) = badappbody x in

(Lambda(nil,App(s0,m0,n0,b,M@L),Tdummy))
```

```
end

else if isfatom x then

let val Fatom(s,m,n,b) = x in
(Lambda(nil,App(s,m,n,b,L),Tdummy))

end

else if (islambdaterm x) then

if  length L > binderlength (bindersof x) then
(say "betared error line 2786:  too many arguments for lambda term";Ferror)

else

(AFUNCTION := x::(!AFUNCTION);
ALIST := L::(!ALIST);

while(hd(!AFUNCTION) <>Ferror andalso
bindersof(hd(!AFUNCTION)) <> nil
andalso hd(!ALIST) <> nil)

do

(
let val (s,(i,m,n,b,TT)) = hd(bindersof(hd(!AFUNCTION))) in
let val (Lambda(B2,t2,T2)) = hd(!AFUNCTION) in
if islambda2 TT then

if (!STRONGEXPAND) then

AFUNCTION := (Subsfunction0 (max(depth,n+1)) s m n b TT
(Lambda(tl B2,t2,T2))::(tl(!AFUNCTION)) else

let val (Lambda(B3,t3,T3)) =
betared depth (Lambda(tl B2,t2,T2)) (hd(!ALIST)) in
```

```
(AFUNCTION :=
(Lambda((s,(i,m,n,b,TT))::B3,t3,T3))
::(tl(!AFUNCTION));
ALIST:=nil::(tl(!ALIST)))

end

else if hd(!ALIST) = nil then ()

else

(AFUNCTION:= (Subsfunction0 (max(depth,n+1)) s m n b
(hd(hd(!ALIST))) (Lambda(tl B2,t2,T2)))::(tl(!AFUNCTION));
ALIST:=(tl(hd(!ALIST)))::(tl(!ALIST)))


end end

);

let val RESULT =

if not (islambdaterm (hd(!AFUNCTION))) then Ferror

else (hd(!AFUNCTION))


in

(AFUNCTION:=tl(!AFUNCTION);ALIST:=tl(!ALIST);RESULT)

end
)


else (say ("General failure of betared line 3350");Ferror)

(* top level definition expansion for objects *)
```

```
and defexpand (App(s,m,n,b,TT)) =
if n=0 andalso isafunction s m then
(* subsobject0 0 s m n b (absolutesortof s m) (App(s,m,n,b,TT)) *)
objectof 0(betared 0 (absolutesortof s m) TT)

else (App(s,m,n,b,TT)) |

defexpand (Oatom(s,m,n,b)) =
if n=0 andalso isanobject s m then
(* subsobject0 0 s m n b (absolutesortof s m) (Oatom(s,m,n,b)) *)

objectof 0 (absolutesortof s m)
else Oatom(s,m,n,b) |

defexpand x = x

and defexpand2 (Fatom(s,m,n,b)) =

let val SS = absolutesortof s m in

if n=0 andalso islambda SS then SS

else (Fatom(s,m,n,b))  end |

defexpand2 x = x

and Subsobject0 depth s m n b T  (Oatom(S,M,N,B)) =

(* if N > depth then (say "depth error line 2806";Oerror) else *)

 if s = S andalso m = M andalso n = N (* andalso b = B *)
then ((* sayD ("3408:  "^(displayerrorfunction T)^"; "^s^": "^S); *) objectof d

(* if n=0 andalso N=0 andalso
m = length(!CONTEXT)-1 andalso
find0 (s,m,n)
 (objectdeplist((Oatom(S,M,N,B))))
```

```
then (say2D s;  say2D "=============";
say2D (displayfunction T); say2D "=============";
say2D S; say2 "=============";
map (fn (s,m,n) => say2D s) (objectdeplist((Oatom(S,M,N,B))));
say "Dependency error line 2814";Oerror) else *)

if n=0 andalso N=0 (* andalso
m = length(!CONTEXT)-1 *) andalso
find0 (s,m,n)
 (objectdeplist((Oatom(S,M,N,B)))) andalso islambda2 (absolutesortof S M)

then Subsobject0 depth s m n b T (objectof depth (absolutesortof S M)) else

Oatom(S,M,N,B) |

Subsobject0 depth s m n b T (App(S,M,N,B,L))
  =  (* if N > depth   then (say "Depth error line 2819";Oerror) else *)

 (*  let val LL = map (Subsfunction0 depth s m n b T) L in *)

if s=S andalso M=m andalso n=0 andalso N=0
andalso islambda2(absolutesortof s m) (* andalso b = B *) then
((* sayD"3429"; *)

Subsobject0 depth s m n b T(
objectof depth ((betared depth  T L) )))

else if s=S andalso M=m andalso n=N
then objectof depth(betared depth T
(map (Subsfunction0 depth s m n b T) L))

else (* if n=0 andalso N=0
andalso m =length(!CONTEXT)-1
andalso find0 (s,m,n) (functiondeplist((Fatom(S,M,N,B))))
then (say2D s;  say2D "=============";
say2D (displayfunction T);  say2D "=============";
say2D S;  say2D "=============";
```

```
map (fn (s,m,n) => say2D s) (functiondeplist((Fatom(S,M,N,B))));
say "Dependency error line 2826";0error)
else *)

if n=0 andalso N=0
(* andalso m =length(!CONTEXT)-1 *)
andalso find0 (s,m,n) (functiondeplist((Fatom(S,M,N,B))))
andalso islambda2 (absolutesortof S M)

then Subsobject0 depth s m n b T
(objectof depth (betared depth (absolutesortof S M) L)) else

App(S,M,N,B,map (Subsfunction0 depth s m n b T) L)  |

Subsobject0 depth s m n b T x = x

and subsobject0 depth s m n b T T2 =
((* sayD(s^" "^(makestring m)^" "^(makestring n)^" "^(if b
 then "" else ". "^(displayerrorfunction T)^" "^(displayerrorobject T2))); *)
Subsobject0 (max(depth,n)) s m n b ((* levelfunction depth *) (T)) T2)

and Substype0 depth s m n b T (THAT p) =
 THAT (Subsobject0 depth s m n b T p) |

Substype0 depth s m n b T (IN p) =
 IN (Subsobject0 depth s m n b T p) |

Substype0 depth s m n b T (Rewrite(TT,UU)) =

Rewrite(Subsobject0 depth s m n b T TT,Subsobject0 depth s m n b T UU) |

Substype0 depth s m n b T x = x

and substype0 depth s m n b T T2 =

Substype0 (max(depth,n)) s m n b
((* levelfunction depth *) (T)) T2
```

```
and Subsfunction0 depth s m n b T (Fatom(S,M,N,B)) =

(* if N > depth then (say "Depth error line 2843";Ferror) else *)

if s=S andalso m=M andalso n=N (* andalso b=B *)  then T
else (* if n=0 andalso N=0
andalso m = length(!CONTEXT)-1
 andalso find0 (s,m,n) (functiondeplist((Fatom(S,M,N,B))))
then (say2D s;  say2D "=============";
say2D (displayfunction T); say2D "=============";
say2D S; say2D "=============";
map (fn (s,m,n) => say2D s) (functiondeplist((Fatom(S,M,N,B))));
say "Dependency error line 2848";Ferror)
else *)

if n=0 andalso N=0
(* andalso m = length(!CONTEXT)-1 *)
 andalso find0 (s,m,n) (functiondeplist((Fatom(S,M,N,B))))
 andalso islambda2 (absolutesortof S M)

 then Subsfunction0 depth s m n b T (absolutesortof S M) else

Fatom(S,M,N,B) |

Subsfunction0 depth s m n b T (Lambda(nil,t,TT)) =

let val UU = maxdepthfunction T in

((*sayD("subsfunction0:  "^s^" "
^(makestring m)^" "^(makestring n)^
" "^(if b then "" else ". ")^(displayerrorfunction T)
^" "^(displayerrorfunction (Lambda(nil,t,TT)))
(* ^" "^(displayfunction T)^(displayfunction T2)*) ); *)

Lambda(nil,Subsobject0 (max(n,
(max(depth+1,(UU)+1)))) s m n b T t,
    Substype0 (max(n,(max(depth+1,
(UU)+1)))) s m n b T TT))  end|
```

```
Subsfunction0 depth s m n bb T
 (Lambda((S,(i,M,N,b,TT))::B,tt,TTT)) =

 ((* sayD("Subsfunction0:  "^s^" "
^(makestring m)^" "^(makestring n)^
" "^(if b then "" else ". ")^(displayerrorfunction T)
^" "^(displayerrorfunction (Lambda((S,(i,M,N,b,TT))::B,tt,TTT)))
(* ^" "^(displayfunction T)^(displayfunction T2)*) ); *)

let val X =
Subsfunction0 depth s m n bb T

let val DDD = maxdepthfunction TT in
(
(Lambda(
map (fn (s2,(i2,m2,n2,b2,TT2)) =>
(s2,(i2,m2,n2,b2,
(* Subsfunction0 (max(depth+2,(DDD) + 1)) s m n bb T *)
(Subsfunction0
(max(depth+2,(DDD) + 1)) S M N b
(referentof(S,M,depth+1,b,TT)) TT2))))

B,
(* Subsobject0 (max(depth+2,(DDD) +1)) s m n bb T *)
(Subsobject0 (max(depth+2,(DDD) +1)) S M N b
(referentof(S,M,depth+1,b,TT)) tt),

(* Substype0 (max(depth+2,(DDD)+1)) s m n bb T *)
(Substype0 (max(depth+2,(DDD)+1)) S M N b
(referentof(S,M,depth+1,b,TT)) TTT)))) end

in

if islambdaterm X then let val (Lambda(B0,tt0,TTT0)) = X in

 (Lambda((S,(i,M,depth+1, b,
Subsfunction0 (max(n,(max(depth+1,
```

```
(maxdepthfunction T)+1)))) s m n bb T TT))::B0,
tt0,TTT0)) end

else (say "Substitution failure line 2873";Ferror)

 end) |

Subsfunction0 depth s m n b T x = x

and subsfunction0 depth s m n b T T2 =
((* sayD("subsfunction0:  "^s^" "^
(makestring m)^" "^(makestring n)^
" "^(if b then "" else ". ")^(displayerrorfunction T)
^" "^(displayerrorfunction T2)
(* ^" "^(displayfunction T)^(displayfunction T2)*) ); *)
Subsfunction0 (max(depth,n)) s m n b )
((* levelfunction depth *) (T)) T2

and levelobject n x =
 (* Subsobject0 n "?!?" 0 0 true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ))) *) x

and levelfunction n x =
 (* Subsfunction0 n "?!?" 0 0 true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ))) *) x

and leveltype n x =
 (* Substype0 n "?!?" 0 0 true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ))) *) x  ;

fun subsobject s t T = let val (S,M,N,B) = parseatom s in

subsobject0 0 s M N B t T

end
```

```
fun subsobjecttest s t T =

displayobject(subsobject

(Getidentifier (s))

(getfunction(Tokenize(t)))

(getobject(Tokenize(T))));

fun substype s t T = let val (S,M,N,B) = parseatom s in

substype0 0 s M N B t T

end;

fun substypetest s t T =

displaytype(substype

(Getidentifier (s))

(getfunction(Tokenize(t)))

(gettype(Tokenize(T))));

fun subsfunction s t T = let val (S,M,N,B) = parseatom s in

subsfunction0 0 s M N B t T

end;

fun subsfunctiontest s t T =

displayfunction(subsfunction
```

```
(Getidentifier (s))

(getfunction(Tokenize(t)))

(getfunction(Tokenize(T))));

(*
```

10.2 Sorting terms and sort reduction: the type system

Below find the algorithm for computing sorts, including a reduction process for types analogous to beta reduction.

```
*)

(* the type system.  We begin with declarations,
and here we need to think about implicit arguments. *)



val XX = ref [Ferror];

val YY = ref [Ferror];

val XXO = ref [Oerror];

val XXOA = ref [false];

val XXOB = ref[false];

val YYO = ref [Oerror];


val ZEROORONE = ref 1;

fun Zeroorone() = ZEROORONE:=1-(!ZEROORONE);

fun topdiff (App(s,m,n,b,L)) (App(S,M,N,B,LL)) =

(s<> S orelse m <>M orelse n <> N)
andalso (n<>0 orelse N <> 0 orelse
(not(islambda2 (absolutesortof s m))
andalso not(islambda2 (absolutesortof S M)))) |
```

```
topdiff (Oatom(s,m,n,b)) (Oatom(S,M,N,B)) =
(s<> S orelse m <>M orelse n <> N)
andalso (n<>0 orelse N <> 0 orelse
(not(islambda2 (absolutesortof s m))
andalso not(islambda2 (absolutesortof S M)))) |

topdiff (App(s,m,n,b,L)) (Oatom(S,M,N,B)) =
(s<> S orelse m <>M orelse n <> N)
andalso (n<>0 orelse N <> 0 orelse
(not(islambda2 (absolutesortof s m))
andalso not(islambda2 (absolutesortof S M)))) |

topdiff (Oatom(s,m,n,b))(App(S,M,N,B,LL)) =
(s<> S orelse m <>M orelse n <> N)
andalso (n<>0 orelse N <> 0 orelse
(not(islambda2 (absolutesortof s m))
 andalso not(islambda2 (absolutesortof S M)))) |

topdiff x y = x <> y;

fun topsame (App(s,m,n,b,L)) (App(S,M,N,B,LL)) =
s=S andalso m = M andalso n =N |

topsame x y = false;

fun objectsort x = (* preobjectsort x *)

 let val A = find Ferror (!CONTEXTNAMES,x)
 (!OBJECTSORTCACHE) in

if A <> Ferror then (OBJECTSORTCACHE:=
((!CONTEXTNAMES,x),A)::
(drop(!CONTEXTNAMES,x)(!OBJECTSORTCACHE));A)

else let val B = (preobjectsort x) in

((OBJECTSORTCACHE:= ((!CONTEXTNAMES,x),B)::
(!OBJECTSORTCACHE));B)
```

```
end end

and functionsort x = (* prefunctionsort x *)

 let val A = find Ferror (!CONTEXTNAMES,x)
 (!FUNCTIONSORTCACHE) in

if A <> Ferror then (FUNCTIONSORTCACHE:=
((!CONTEXTNAMES,x),A)::
(drop(!CONTEXTNAMES,x)(!FUNCTIONSORTCACHE));A)

else let val B = (if (!STRONGEXPAND)  then (fn x => x)
else typesimplify) (prefunctionsort x) in

((FUNCTIONSORTCACHE:= ((!CONTEXTNAMES,x),B)
::(!FUNCTIONSORTCACHE));B)

end end

and preobjectsort (Oatom(s,m,0,b)) =

(* it is worth noting that for type casting reasons
this function actually returns a function sort *)

(* fixindices will change all environment m index 0 atoms
to index n *)

let val T = (absolutesortof s m)

in

(* making sure we return a type *)

(* let val T2 = silentfunctionsort T

in
```

```
if T2 = Ferror then T else T2

end *)  if islambda2 T then functionsort T else T

end  |

preobjectsort (App(s,m,0,b,L)) =

typered 0 (functionsort(Fatom(s,m,0,b))) L |

preobjectsort x =
(say "general failure of objectsort line 2989"
;Ferror)

(* I cannot have error reporting in function
sort for reasons I cannot figure out *)

(* and functionsort x = silentfunctionsort x *)

and prefunctionsort (Fatom(s,m,0,b)) =

(* here I am making sure that functionsort
actually returns a type *)

let val T = (absolutesortof s m)

in (* let val T2 = silentfunctionsort T

in

if T2 = Ferror then T else T2

end *)  if islambda2 T then functionsort T else T

end

 |
```

```
 prefunctionsort (Lambda(nil,t,T)) =

 if T = Tdummy then objectsort t
 else if t = Unknown then
 ( say "cannot take functionsort of a sort line 3015";
 Ferror)
 else (Lambda(nil,Unknown,T))

 |

prefunctionsort (Lambda(B,t,T)) =

if t = Unknown then
( say2 ("cannot take function sort of a sort line 3024 :  "
^(displayfunction(Lambda(B,t,T))))
;  Ferror)

else (Lambda((* map updatebinderitem *) B,Unknown,T)) |

prefunctionsort x = ( say
"general failure of functionsort line 3030";  Ferror)

and silentfunctionsort x = functionsort x

and typered depth x y =

if not(islambdaterm x) then
(say "general failure in typered line 3794";Ferror)

else if y = nil then x

else let val (Lambda(B,t,T)) = x in

(
AFUNCTION := (Lambda(B,t,T))::(!AFUNCTION);

ALIST:=y::(!ALIST);
```

```
while(hd(!AFUNCTION)<>Ferror andalso
bindersof(hd(!AFUNCTION)) <> nil
andalso hd(!ALIST)<>nil)

do(

let val (s,(i,m,n,b,TT)) =
hd(bindersof(hd(!AFUNCTION))) in

let val (Lambda(B,t,T)) = hd(!AFUNCTION) in

if islambda2 TT then

if (!STRONGEXPAND) then

AFUNCTION := (subsfunction0 depth s m n b TT
(Lambda(tl B,t,T)))::(tl(!AFUNCTION))  else

let val (Lambda(B3,t3,T3))
= typered depth (Lambda(tl B,t,T)) (hd(!ALIST)) in

(AFUNCTION :=
(Lambda((s,(i,m,n,b,TT))::B3,t3,T3))
::(tl(!AFUNCTION));
ALIST:=nil::(!ALIST))

end

else if hd(!ALIST) = nil then ()

else if equalfunctions TT (functionsort(hd(hd(!ALIST)))) then

(* let val (Lambda(B,t,T)) = hd(!AFUNCTION) in *)

(AFUNCTION := (subsfunction0 depth s m n b
     (hd(hd(!ALIST)))
(Lambda(tl(B),t,T)))::(tl(!AFUNCTION));
ALIST := (tl(hd(!ALIST)))::(tl(!ALIST))) (* end *)
```

```
else (say("Comparison failed of "
^(displayfunction TT)^" and "
^(displayfunction(functionsort(hd(hd(!ALIST))))));
AFUNCTION:=Ferror::(tl(!AFUNCTION)))

end end ) ;

let val RESULT = if hd(!ALIST) = nil
then hd(!AFUNCTION) else ("Ran out of arguments 3857";Ferror) in
(AFUNCTION:=tl(!AFUNCTION);
ALIST:=tl(!ALIST);RESULT) end) end




(* for the moment I am implementing
equality up to alpha reduction and definitional expansion *)

(*
```

10.3 Equality of objects and functions

There is no break in the code here: equality of objects and functions is defined along with the type system. But there is a conceptual break (could this actually be pulled out and appear earlier?)

Equality as defined here handles renaming of bound variables and expansion of definitions as needed.

The original recursive code has been removed for surveyability.

```
*)


and equalobjectsorts (THAT p) (THAT P) =
equalobjects p P |

equalobjectsorts (IN p) (IN P) =
equalobjects p P  |

equalobjectsorts (Rewrite(t1,u1)) (Rewrite(t2,u2))

= equalobjects t1 t2 andalso equalobjects u1 u2 |

equalobjectsorts x y = x=y andalso x <> Terror



and defadjust (App(s,m,n,b,L)) =
if n = 0 andalso islambda2(absolutesortof s m)
then defexpand(App(s,m,n,b,L))

else if L = nil then (App(s,m,n,b,L))

else if equalfunctions(defadjustfun (hd L))( hd L) then

(fn (App(s2,m2,n2,b2,M)) => (App(s2,m2,n2,b2,(hd L)::M)))
(defadjust(App(s,m,n,b,tl L)))
```

```
else App(s,m,n,b,(defadjustfun(hd L))::L) |

defadjust x = defexpand x

and defadjustfun (Fatom(s,m,n,b)) =
defexpand2 (Fatom(s,m,n,b)) |

defadjustfun (Lambda(nil,t,T)) = Lambda(nil,defadjust t,T) |

defadjustfun (Lambda((s3,(i3,m3,n3,b3,TT3))::B,t,T)) =

if islambda0 (Lambda((s3,(i3,m3,n3,b3,TT3))::B,t,T))
 then defadjustfun (subsfunction0 0 s3 m3 n3 b3
TT3 (Lambda(B,t,T)))

else (Lambda((s3,(i3,m3,n3,b3,TT3))::B,t,T)) |

defadjustfun x = x

and defadjuststage (App(s,m,n,b,L)) =
if n = 0 andalso islambda2(absolutesortof s m)
then m

else if L = nil then ~1

else if equalfunctions(defadjustfun (hd L))( hd L) then

defadjuststage(App(s,m,n,b,tl L))

else defadjustfunstage(hd L) |

defadjuststage (Oatom(s,m,n,b)) = if n = 0
andalso islambda2(absolutesortof s m)
then m else ~1 |

defadjuststage x = ~1

and defadjustfunstage (Fatom(s,m,n,b)) =
```

```
 if n = 0 andalso islambda2(absolutesortof s m)
then m else ~1|

defadjustfunstage (Lambda(nil,t,T)) = defadjuststage t |

defadjustfunstage (Lambda((s3,(i3,m3,n3,b3,TT3))::B,t,T)) =

if islambda0 (Lambda((s3,(i3,m3,n3,b3,TT3))::B,t,T))
 then defadjustfunstage (subsfunction0 0 s3 m3 n3 b3
TT3 (Lambda(B,t,T)))

else ~1 |

defadjustfunstage x = ~1

 and congruentobjects (Oatom(s,m,n,b))
(Oatom(S,M,N,B)) =

s=S andalso m=M andalso n=N |

congruentobjects (App(s,m,n,b,nil)) (App(S,M,N,B,nil)) =
s=S andalso m=M andalso n=N |

congruentobjects (App(s,m,n,b,(x::L)))
 (App(S,M,N,B,(y::MM))) =
(s=S andalso m=M andalso n=N
andalso (x::L)=(y::MM)) orelse
(equalfunctions x y andalso
congruentobjects (App(s,m,n,b,(L))) (App(S,M,N,B,(MM)))) |

congruentobjects x y = x <> Oerror andalso x=y

and equalobjects x y =

(XXO := x::(!XXO);

YYO := y::(!YYO);
```

```
XXOA:=false::(!XXOA);

XXOB:=false::(!XXOB);

while (not(congruentobjects (hd(!XXO)) (hd(!YYO)))
andalso hd(!XXO) <> Oerror)

do (

if topsame (hd(!XXO)) (hd(!YYO))
andalso equalfunctionlists
(applist(hd(!XXO)))(applist(hd(!YYO)))

then YYO:=(hd(!XXO))::(tl(!YYO))

else if topdiff (hd(!XXO)) (hd(!YYO))
     then XXO := Oerror::(tl(!XXO)) else (* (); *)
 (* XXOA:=(hd(!XXO))::(tl(!XXOA)); *)
let val XXX = defadjust (hd(!XXO)) in
let val YYY = defadjust (hd(!YYO)) in
let val BBB = congruentobjects YYY (hd(!YYO)) in
(XXOA :=(congruentobjects (hd(!XXO)) XXX)
   ::(tl(!XXOA));

if (* hd(!XXO) <> Oerror andalso
not(congruentobjects(hd(!XXO))(hd(!YYO))) andalso *)
not(congruentobjects (hd(!XXO)) XXX)
andalso (((* congruentobjects
 (defexpand (hd(!YYO))) (hd(!YYO) *) BBB ))

 (* reduce concepts from higher indexed worlds
 by preference *)

 orelse defadjuststage(hd(!XXO)) > defadjuststage(hd(!YYO))
(* orelse (topsame(XXX)(hd(!YYO)) andalso not(hd(!XXOA))) *)
then (XXOB:=true::(tl(!XXOB));XXO := XXX
::(tl(!XXO))) else(*  (); *)
```

```
(* let val YYY = defadjust (hd(!YYO)) in *)

if (* hd(!XXO) <> Oerror andalso
not(congruentobjects (hd(!XXO)) (hd(!YYO)) )
andalso (not(hd(!XXOB)))
andalso *)  not((* congruentobjects YYY (hd(!YYO))*) BBB)

then YYO := YYY::(tl(!YYO)) else

if (hd(!XXOA)) andalso
not(congruentobjects (hd(!XXO)) (hd(!YYO))) then

XXO:=Oerror::(tl(!XXO)) else ()



)
end end end
);

let val RESULT =
hd(!XXO) <> Oerror andalso
congruentobjects (hd(!XXO)) (hd(!YYO))


in (XXO := tl(!XXO);XXOA:=tl(!XXOA);
XXOB:=tl(!XXOB);YYO := tl(!YYO);RESULT) end)




and equalfunctionlists (x::L) (y::M) =

equalfunctions x y
andalso equalfunctionlists L M |

equalfunctionlists x y = x=y
```

```
(* this is a rather excessive degree of success at making
equalfunctions iterative *)

and equalfunctions x y =

(XX:=x::(!XX);YY:=y::(!YY);equalfunctions0())

and equalfunctions0() =

let val x = hd(!XX) and y = hd(!YY) in

if hd(!XX) = hd(!YY) then (XX:=tl(!XX);YY:=tl(!YY);true) else

(if islambdaterm x andalso islambdaterm y then

let val (Lambda(B,t,T)) = x and (Lambda(B2,t2,T2)) = y

in

(* let val XX = ref x and YY = ref y in *)

((* XX := x::(!XX);  YY := y::(!YY);*)

let val BREAKOUT = ref false in
while ((bindersof (hd(!XX)) <> nil
orelse bindersof (hd(!YY)) <> nil)
andalso not(!BREAKOUT))

do

(

if bindersof(hd(!XX))<>nil andalso bindersof(hd(!YY))<>nil andalso
islambda2(bindersof2(hd(!XX))) andalso islambda2(bindersof2(hd(!YY))) andalso
(XX:=(bindersof2(hd(!XX))::((!XX));YY:=(bindersof2(hd(!YY))::((!YY));equalfun
let val (s,(i,m,n,b,TT)) = hd(bindersof(hd(!XX)))
and (s2,(i2,m2,n2,b2,TT2)) = hd(bindersof(hd(!YY))) in
```

```
(XX:=(subsfunction0 (n+n2+1) s m n b (referentof(s,m,n,b,TT))(lambdatail(hd(!XX
YY:=  ( subsfunction0 (n+n2+1) s2 m2 n2 b2 (referentof(s,m,n,b,TT))   (lambdata

) end

else

if bindersof (hd(!XX)) <> nil
andalso islambda2(bindersof2 (hd(!XX)))

then let val (s,(i,m,n,b,TT)) = hd(bindersof (hd(!XX))) in

XX := (subsfunction0 (n+1) s m n b TT
(lambdatail(hd(!XX))))::(tl(!XX))

end

else if bindersof (hd(!YY)) <> nil
andalso islambda2(bindersof2 (hd(!YY)))

then (let val (s,(i,m,n,b,TT)) = hd(bindersof (hd(!YY))) in

YY := (subsfunction0 (n+1) s m n b TT
(lambdatail(hd(!YY))))::(tl(!YY))

end)

else if (bindersof (hd(!XX)) <> nil
andalso bindersof (hd(!YY)) <> nil
andalso (*equalfunctions (bindersof2
(hd(!XX))) (bindersof2 (hd(!YY)))*)

(XX:=(bindersof2 (hd(!XX)))::(!XX);
YY:=(bindersof2 (hd(!YY)))::(!YY);equalfunctions0()))

then

let val (s,(i,m,n,b,TT)) = hd(bindersof (hd(!XX)))
```

```
and (s2,(i2,m2,n2,b2,TT2)) = hd(bindersof (hd(!YY))) in

(
XX:= (subsfunction0 (n+n2+1) s m n b (referentof(s,m,n,b,TT))
(lambdatail(hd(!XX)))):(tl(!XX));
YY:= (subsfunction0 (n+n2+1) s2 m2 n2 b2 (referentof(s,m,n,b,TT))
 (lambdatail(hd(!YY)))):(tl(!YY))

)

end

else

(
BREAKOUT:=true
)

) (* end *);

let val RESULT = (bindersof (hd(!XX)) = nil andalso bindersof (hd(!YY)) = nil

andalso

(let val (Lambda(nil,t,T)) = (hd(!XX))
and (Lambda(nil,t2,T2)) = (hd(!YY))

in
(t <> Unknown andalso equalobjects t t2) orelse

(t = Unknown andalso t2 = Unknown
 andalso equalobjectsorts T T2 ) end)) in (XX:=tl(!XX);YY:=tl(!YY);RESULT)

end
```

```
end)

end

else

if defexpand2 x <> x then let val RESULT = equalfunctions (defexpand2 x) y
    in (XX:=tl(!XX);YY:=tl(!YY);RESULT) end

else if defexpand2 y <>y then let val RESULT = equalfunctions  x (defexpand2 y)
    in (XX:=tl(!XX);YY:=tl(!YY);RESULT) end

else if Etared 0 x <> x then let val RESULT = equalfunctions (Etared 0 x) y
    in (XX:=tl(!XX);YY:=tl(!YY);RESULT) end

else if Etared 0 y <> y then let val RESULT = equalfunctions x (Etared 0 y)
    in (XX:=tl(!XX);YY:=tl(!YY);RESULT) end

else false) end



(* converted to iterative style... *)

and typesimplify x =

(if not (islambdaterm x) then x

else

let val (Lambda(B,t,T)) = x in (*1 *)

let val BB = ref B and XX = ref x in (* 2 *)

(

BB := rev B;
```

```
XX := Lambda(nil,t,T);

(while ((!BB) <> nil)

do

(

let val (s,(i,m,n,b,TT)) = hd (!BB) in

(if islambda2 TT

andalso ((!STRONGEXPAND) orelse
functionoccurrences s m n
(Lambda(rev(tl(!BB)),t,T)) <= (!ZEROORONE))

 then

XX := subsfunction0 0 s m n b TT (!XX)

else XX := (fn (Lambda(B2,t2,T2)) => Lambda ((hd(!BB))::B2,t2,T2)) (!XX);

BB := tl(!BB))

end)); !XX)



end (* 2 *)


end (* 1 *)

)
```

```
fun objectsorttest s = objectsort(getobject(Tokenize s));

fun functionsorttest s = functionsort(getfunction(Tokenize s));


(*
```

10.4 Matching and multiple substitution

The matching and multiple substitution features here (with some modest higher order matching) support the implicit argument inference feature, which seems to be very important for fluent use of Lestrade. This is quite complex.

```
*)

(* matching and multiple substitution code *)

(* undoing substitution *)

(* undo a single substitution *)

fun unsubstype S x (THAT p) =
THAT (unsubsobject S x p) |

unsubstype S x (IN tau) =

IN (unsubsobject S x tau) |

unsubstype S x (Rewrite(t,u)) =

Rewrite(unsubsobject S x t,unsubsobject S x u) |

unsubstype S x y = y

and  unsubsobject  S x (Oatom(s,m,n,b)) =

if equalobjects (Oatom(s,m,n,b)) (objectof 0 x)
then ((* sayD "4294"; *) objectof 0 ((referentof(S,length(!CONTEXT)-1,0,true,
functionsort x))))

(* else let val DD =
defexpand  (Oatom(s,m,n,b)) in

if DD<>(Oatom(s,m,n,b)) then
```

```
unsubstype S x DD  *)

(* else if defexpand(Oatom(s,m,n,b)) <> Oatom(s,m,n,b)

then let val X = unsubsobject  S x (defexpand(Oatom(s,m,n,b))) in

if X <> defexpand(Oatom(s,m,n,b)) then X else Oatom(s,m,n,b)

end *)

else Oatom(s,m,n,b)

(* end *) |

unsubsobject S x (App(s,m,n,b,L)) =

if  equalobjects
(levelobject 0 (App(s,m,n,b,L)))
(levelobject 0 (objectof 0 x) )
then ((* say3 "4313"; *) objectof 0 (referentof(S,length(!CONTEXT)-1,0,true,
functionsort x)))

else (* if defexpand(App(s,m,n,b,L)) <> App(s,m,n,b,L)

then let val X = unsubsobject S x (defexpand(App(s,m,n,b,L))) in

if X <> defexpand(App(s,m,n,b,L)) then X else

let val LL = map (unsubsfunction S x) L in

if x = Fatom(s,n,m,b) then

App(S,length(!CONTEXT)-1,0,true,LL)

(* here we could have another case
using matching if x is a lambda term *)

else App(s,m,n,b,LL)
```

```
end end

else *)

let val LL = map (unsubsfunction S x) L in

if x = Fatom(s,n,m,b) then

App(S,length(!CONTEXT)-1,0,true,LL)

(* here we could have another case
using matching if x is a lambda term *)

else App(s,m,n,b,LL)

end
 |

unsubsobject S x y = y

and unsubsfunction  S x (Fatom(s,m,n,b)) =

if equalfunctions x (Fatom(s,m,n,b))
then  (referentof(S,length(!CONTEXT)-1,0,true,
functionsort x))

else (* if defexpand2(Fatom(s,m,n,b)) <> Fatom(s,m,n,b)
then let val X = unsubsfunction S x (defexpand2(Fatom(s,m,n,b)))

in

if X <> defexpand2(Fatom(s,m,n,b)) then X

else Fatom(s,m,n,b)

end
```

```
else *)  (Fatom(s,m,n,b)) |

(* here we assume that variables which are bound are actually
marked as bound -- this can be fixed if necessary *)

unsubsfunction S x (Lambda(B,t,T)) =

if equalfunctions
(levelfunction 0 x)
(levelfunction 0 (Lambda(B,t,T)))
then  (referentof(S,length(!CONTEXT)-1,0,true,
functionsort x))

else Lambda (map (fn (s,(i,m,n,b,TT)) =>
(s,(i,m,n,b,unsubsfunction S x TT))) B,
unsubsobject S x t,
unsubstype S x T) |

unsubsfunction S x y = y;

fun varsof s =  let val SS = absolutesortof s (length(!CONTEXT)-1) in

if SS = Ferror then [""] else

let val (Lambda(B,t,T)) = SS in

map (fn (s,(i,m,n,b,TT)) => s) B end end;

fun unapply [""] L T =
(say "declaration failure in unapply line 3429"
;Ferror) |

unapply nil nil T = Lambda(nil,T,theobjectsort(objectsort T)) |

unapply (s::L1) (x::L2) T =

let val A = unapply L1 L2 T in if A = Ferror then
(say "unapply didn't yield lambda term line 3448";Ferror) else
```

```
 let val S= extend s and (Lambda(B,t,T)) = A in

 subsfunction0 1 "?!?" 0 0 true
(referentof("?!?",0,0,true,Lambda(nil,Unknown,OBJ)))
(Lambda((S,(1,length(!CONTEXT)-1,0,true,functionsort x))::
(map (fn (s2,(i2,m2,n2,b2,TT2)) =>
(s2,(i2,m2,n2,b2,unsubsfunction S x TT2))) B),
unsubsobject S x t,unsubstype S x T) )end end|

unapply x y z =
(say "general failure of unapply line 3458";Ferror);

(* mechanics of matching lists *)

fun isvariable (Lambda(nil,Oatom(s,m,0,b),Tdummy)) = m=length(!CONTEXT)-1 |

isvariable (Fatom(s,m,0,b)) = m=length(!CONTEXT)-1 |

isvariable x = false;

fun addmatch key value [(""),Ferror)] = [(""),Ferror)] |

addmatch key Ferror L = [("",Ferror)] |

addmatch key value nil = [(key,value)] |

addmatch key value ((key2,value2)::L) =

if key = key2 then if equalfunctions
(levelfunction 0 value)
(levelfunction 0 value2)

(* orelse (isvariable value
orelse isvariable value2) *)

then ( (* sayD ("Matched "^key^" with "
^(displayerrorfunction value)^" and "^(displayerrorfunction value2)); *)
```

```
((key2, (*if isvariable value2 then value  else *) value2)::(drop key2 L)))

else
(  say ("Failed to match "^(key2)
^" with "^(displayfunction value2)
^" and "^(displayfunction value)^" line 3485");
[(("),Ferror)])

else let val LL = addmatch key value L in

if LL =  [(("),Ferror)] then  [(("),Ferror)]
else (key,value)::((key2,value2)::(drop key (drop key2 LL))) end;

fun mergematch  [(("),Ferror)] L =  [(("),Ferror)] |

mergematch L  [(("),Ferror)] =  [(("),Ferror)] |

mergematch x nil = x |

mergematch nil x = x |

mergematch ((key,value)::L) ((key2,value2)::L2) =

addmatch key value (addmatch key2 value2 (mergematch L L2));

val LOCALMATCHES = ref (nil:((string * Function) list));

(* this supports a strategy of actually building local match lists
but while also consulting a global list of matches *)

fun showmatches0 nil = " is localmatches" |

showmatches0 ((s,t)::L) = s^(":  ")
^(displayfunction t)^"; "^(showmatches0 L);

fun showmatches() = showmatches0 (!LOCALMATCHES);
```

```
fun addtomatches key value match =
(LOCALMATCHES:= addmatch key value
(!LOCALMATCHES);(* sayD(showmatches()) ; *)
     if (!LOCALMATCHES) = [("",Ferror)] then [("",Ferror)]
    else addmatch key value match);

fun matchesfail() = (LOCALMATCHES:=[("",Ferror)];[("",Ferror)]);

(* substitutions using a match *)

fun matchsubstype depth match (THAT p) =
THAT (matchsubsobject depth match p) |

matchsubstype depth match (IN tau) =
IN (matchsubsobject depth match tau) |

matchsubstype depth match (Rewrite(t,u)) =

Rewrite(matchsubsobject depth match t, matchsubsobject depth match u) |

matchsubstype depth x y = y

and matchsubsobject depth match (Oatom(s,m,n,b)) =

if n <>0 orelse m<> length(!CONTEXT)-1
then Oatom(s,m,n,b)

else subsobject0 depth s m n b
(find Ferror  s match) (Oatom(s,m,n,b)) |

matchsubsobject depth match (App(s,m,n,b,L)) =

let val LL = map (matchsubsfunction depth match) L  in

if map (fn x => x<> Ferror) LL = map (fn x => true) LL then

if n<>0 orelse m <> length(!CONTEXT)-1
```

```
then (App(s,m,n,b,LL))

else subsobject0 depth s m n b
(find Ferror s match)  (App(s,m,n,b,LL))

else (say "matchsubsobject failure line 3527";Oerror) end |

matchsubsobject depth match x = x

and matchsubsfunction depth match (Fatom(s,m,n,b)) =

if n<>0 orelse m <> length(!CONTEXT) - 1
then Fatom(s,m,n,b)

else subsfunction0 depth s m n b
(find Ferror s match)  (Fatom(s,m,n,b)) |

matchsubsfunction depth match (Lambda(B0,t0,T0)) =

let val XX =  subsfunction0 depth "?!?" 0 0
true  (referentof("?!?",0,0,true,(Lambda(nil,Unknown,OBJ))))
(Lambda(B0,t0,T0))  in if not(islambdaterm XX) then
(say "matchsubsfunction fails to generate lambda term line 3565";
Ferror)
else let val (Lambda(B,t,T)) = XX in

let val BB = map (fn (s,(i,m,n,b,TT)) =>
 (s,(i,m,n,b,matchsubsfunction (depth+1) match TT))) B

and tt = matchsubsobject (depth+1) match t

and TT = matchsubstype (depth+1) match T in

if map (fn(s,(i,m,n,b,TT)) => TT <> Ferror) BB
= map (fn x=>true) BB
andalso tt <> Oerror andalso TT <> Terror

then Lambda(BB,tt,TT)
```

```
else (say "matchsubsfunction failure line 3582";Ferror)

end end end |

matchsubsfunction depth match x = x;

fun matchallsubs t = matchsubsfunction 0 (!LOCALMATCHES) t;

(*

(* actual matching function *)

fun matchtypes (THAT p) (THAT P) = matchobjects p P |

matchtypes (IN p) (IN P) = matchobjects p P |

matchtypes Terror Terror = matchesfail() |

matchtypes x y = if x=y then nil else  matchesfail()


and matchobjects (Oatom(s,m,0,b)) T =

if m <> length(!CONTEXT) - 1 then

if equalobjects (Oatom(s,m,0,b)) T then nil

else  matchesfail()

else addtomatches s (Lambda(nil,T,Tdummy)) (matchfunctions
(absolutesortof s m) (objectsort T))

 |

matchobjects (App(s,m,n,b,L)) (App(s2,m2,n2,b2,L2)) =
```

```
( sayD ("Matching "^(displayerrorobject (App(s,m,n,b,L)))^
" with "^(displayerrorobject (App(s2,m2,n2,b2,L2))));

if n<>0 orelse n2<>0 orelse m <> length(!CONTEXT) -1

then if s=s2 andalso m=m2 andalso n=n2
then matchfunctionlists L L2

 else let val DD2 = defexpand  (App(s2,m2,n2,b2,L2))  in

if DD2 <>  (App(s2,m2,n2,b2,L2))

then matchobjects (App(s,m,n,b,L)) DD2

(* else let val DD = defexpand (App(s,m,n,b,L)) in

if DD <> (App(s,m,n,b,L)) then

matchobjects DD (App(s2,m2,n2,b2,L2))  *)

else matchesfail()

 end   (* end *)

else let val RESULT =
unapply (varsof s)
(map matchallsubs L) (App(s2,m2,0,b2,L2)) in

if RESULT = Ferror then matchesfail()

else addtomatches s RESULT
(matchfunctions (absolutesortof s m)
(functionsort RESULT))  end) |

matchobjects (App(s,m,0,b,L)) T =

((* sayD ("Matching "^(displayerrorobject (App(s,m,0,b,L)))^
" with "^(displayerrorobject T)); *)
```

```
if m<> length(!CONTEXT)-1

then  (* let val DD = defexpand (App(s,m,0,b,L)) in

if DD <>(App(s,m,0,b,L)) then matchobjects DD T

else *) let val DD2 = defexpand T in

if DD2 <> T then matchobjects (App(s,m,0,b,L)) DD2

else matchesfail() end (* end *)

else  let val RESULT = unapply (varsof s) (map  matchallsubs L) T in
( (* say3(displayfunction RESULT); *)
if RESULT = Ferror then matchesfail()

else addtomatches s RESULT
(matchfunctions (absolutesortof s m)
(functionsort RESULT)))  end) |

matchobjects x y =

if x=y andalso x<>Oerror then nil

else matchesfail()

and matchfunctions (Fatom(s,m,n,b)) T =

if n<>0 orelse m <> length(!CONTEXT)-1

then if  equalfunctions (Fatom(s,m,n,b)) T then nil

else matchesfail()

else

addtomatches s T
```

```
(matchfunctions (absolutesortof s m)
(functionsort T)) |

matchfunctions (Lambda(nil,t,T)) (Lambda(nil,t2,T2)) =

let val MMM = matchobjects t t2 in

let val M=mergematch (MMM) (matchtypes T T2)

in
(LOCALMATCHES:= mergematch M (!LOCALMATCHES);
if (!LOCALMATCHES)=[("",Ferror)] then [("",Ferror)]
else M)

end end

 |

 matchfunctions (Lambda((s,(i,m,n,b,TT))::B,t,T))
   (Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2)) =

((* sayD("Matching "^(displayerrorfunction
(Lambda((s,(i,m,n,b,TT))::B,t,T)))^
 " with "^(displayerrorfunction
(Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2)))); *)

if islambda2 TT then

matchfunctions (subsfunction0 (n+1) s m n b TT
(Lambda(B,t,T)))
(Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2))

else if islambda2 TT2 then

matchfunctions (Lambda((s,(i,m,n,b,TT))::B,t,T))
(subsfunction0 (n2+1) s2 m2 n2 b2 TT2 (Lambda(B2,t2,T2)))

else (let val S = extend s and S2 = extend s2 in
```

```
let val TTT = matchfunctions TT TT2 in

(CONTEXT:=((S,(1,length(!CONTEXT)-1,0,b,TT))::
(S2,(1,length(!CONTEXT)-1,0,b2,TT2))
::(Hd nil (!CONTEXT)))::(Tl(!CONTEXT));
addtomatches S
(referentof(S2,length(!CONTEXT)-1,0,b2,TT2))(!LOCALMATCHES);
let val MMM =(mergematch(TTT)
(matchfunctions (subsfunction0 (n+1) s m n b
(referentof(S,(length(!CONTEXT)-1),0,b,TT)) (Lambda(B,t,T)))
(subsfunction0 (n2+1) s2 m2 n2 b2
(referentof(S2,length(!CONTEXT)-1,0,b2,TT2))
 (Lambda(B2,t2,T2)))))
in (LOCALMATCHES:=mergematch MMM (!LOCALMATCHES);
if (!LOCALMATCHES)=[("",Ferror)] then [("",Ferror)]
else MMM) end

) end

end)) |

matchfunctions (Lambda(nil,t,T))
 (Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2)) =

 ((* sayD("Matching "^(displayerrorfunction (Lambda(nil,t,T)))^
 " with "^(displayerrorfunction (Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2)))); *)

if islambda2 TT2 then

matchfunctions (Lambda(nil,t,T))
(subsfunction0 (n2+1) s2 m2 n2 b2 TT2 (Lambda(B2,t2,T2)))

else matchesfail()) |

matchfunctions (Lambda((s,(i,m,n,b,TT))::B,t,T))
   (Lambda(nil,t2,T2)) =
```

```
((* sayD("Matching "^(displayerrorfunction (Lambda((s,(i,m,n,b,TT))::B,t,T)))^
 " with "^(displayerrorfunction (Lambda(nil,t2,T2)))); *)

if islambda2 TT

then matchfunctions
(subsfunction0 (n+1) s m n b TT (Lambda(B,t,T)))
(Lambda(nil,t2,T2))

else matchesfail()) |

matchfunctions x y = if  x=y andalso x <> Ferror then nil

else

let val A = defexpand2 y in

if A <> y then matchfunctions x A

else let val B = Etared 0 y in

if B <>y then matchfunctions x B

(* else let val C = Etared 0 x in

if C <> x then matchfunctions C y *)

else matchesfail() end end (* end *)

and matchfunctionlists nil nil = nil |

matchfunctionlists (s::L) (t::M) =

let val MM = matchfunctions s t in

(LOCALMATCHES := mergematch (MM) (!LOCALMATCHES);
if (!LOCALMATCHES) = [("",Ferror)] then [("",Ferror)]
else mergematch(MM) (matchfunctionlists L M)) end |
```

```
matchfunctionlists x y =

matchesfail();

*)

(* matching functions with a target *)


fun matchtypest v (THAT p) (THAT P) =

if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else

matchobjectst v p P |

matchtypest v (IN p) (IN P) =

if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else

matchobjectst v p P |

matchtypest v Terror Terror =
((* sayD "matchesfail 4388"; *)matchesfail()) |

matchtypest v (Rewrite(t1,u1)) (Rewrite(t2,u2)) =

if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


let val MM = matchobjectst v t1 t2 in

(LOCALMATCHES := mergematch (MM) (!LOCALMATCHES);
if (!LOCALMATCHES) = [("",Ferror)] then [("",Ferror)]
else mergematch(MM) (matchobjectst v u1 u2)) end |
```

```
matchtypest v x y = if

find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else

if x=y orelse y=Tdummy then nil
else  ((* sayD ("matchesfail 4396: "^(displaytype x)^"; "
^(displaytype y)); *) matchesfail())


and matchobjectst v (Oatom(s,m,0,b)) T =

( (* sayD ("Matching "^(displayerrorobject (Oatom(s,m,0,b)))^
" with "^(displayerrorobject (T))); *)

if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else

if not (find0 (v,length(!CONTEXT)-1,0)
(objectdeplist (Oatom(s,m,0,b)))) then nil else

if m <> length(!CONTEXT) - 1 then

if equalobjects (Oatom(s,m,0,b)) T then nil

else  ( (* sayD "matchesfail 4413"; *) matchesfail())

else addtomatches s (Lambda(nil,T,Tdummy)) (matchfunctionst v
(objectsort(Oatom(s,m,0,b))) (objectsort T)))

 |

matchobjectst v (App(s,m,n,b,L)) (App(s2,m2,n2,b2,L2)) =

if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else
```

```
if not(find0 (v,length(!CONTEXT)-1,0)
(objectdeplist (App(s,m,n,b,L)))) then nil else

( (* sayD ("Matching "^(displayerrorobject (App(s,m,n,b,L)))^
" with "^(displayerrorobject (App(s2,m2,n2,b2,L2)))); *)

if n<>0 orelse n2<>0 orelse m <> length(!CONTEXT) -1

then if s=s2 andalso m=m2 andalso n=n2
then matchfunctionliststst v L L2

else let val DD2 = defexpand  (App(s2,m2,n2,b2,L2))  in

if DD2 <>  (App(s2,m2,n2,b2,L2))

then matchobjectst v (App(s,m,n,b,L)) DD2

else let val DD = defexpand (App(s,m,n,b,L)) in

if DD <> (App(s,m,n,b,L)) then

matchobjectst v DD (App(s2,m2,n2,b2,L2))

else ( (* sayD "matchesfail 4448"; *) matchesfail())

end  end

else let val RESULT =
unapply (varsof s)
(map matchallsubs L) (App(s2,m2,0,b2,L2)) in

if RESULT = Ferror then
( (* sayD "matchesfail 4456"; *) matchesfail())

else addtomatches s RESULT
(matchfunctionst v (absolutesortof s m)
(functionsort RESULT))  end) |
```

```
matchobjectst v (App(s,m,0,b,L)) T =

if not(find0 (v,length(!CONTEXT)-1,0)
(objectdeplist (App(s,m,0,b,L)))) then nil else

( (* sayD ("Matching "^(displayerrorobject (App(s,m,0,b,L)))^
" with "^(displayerrorobject T)); *)

if m<> length(!CONTEXT)-1

then let val DD2 = defexpand T in

if DD2 <> T then matchobjectst v (App(s,m,0,b,L)) DD2

else let val DD = defexpand (App(s,m,0,b,L)) in

if DD <>(App(s,m,0,b,L)) then matchobjectst v DD T

else ( (* sayD "matchesfail 4479"; *) matchesfail()) end end

else  let val RESULT = unapply (varsof s) (map  matchallsubs L) T in
( (* say3(displayfunction RESULT); *)
if RESULT = Ferror then (
(* sayD "matchesfail 4483"; *) matchesfail())

else addtomatches s RESULT
(matchfunctionst v (functionsort(Fatom(s,m,0,b)))
(functionsort RESULT)))  end) |

matchobjectst v x y =
( (* sayD ("Matching "^(displayerrorobject x)^
" with "^(displayerrorobject (y))); *)
if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


if x=y andalso x<>Oerror then nil
```

```
else (say "matchesfail 4498";matchesfail()))

and matchfunctionst v (Fatom(s,m,n,b)) T =

if not (find0 (v,length(!CONTEXT)-1,0)
(functiondeplist (Fatom(s,m,n,b)))) then nil else

if n<>0 orelse m <> length(!CONTEXT)-1

then if  equalfunctions (Fatom(s,m,n,b)) T then nil

else ( (* sayD "matchesfail 4508"; *) matchesfail())

else

addtomatches s T
(matchfunctionst v (functionsort(Fatom(s,m,n,b)))
(functionsort T)) |

matchfunctionst v (Lambda(nil,t,T0)) (Lambda(nil,t2,T20)) =

let val T = if t=Unknown then T0 else theobjectsort(objectsort t) and
T2 = if t2 =Unknown then T20 else theobjectsort(objectsort t2) in

if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


if not(find0 (v,length(!CONTEXT)-1,0)
(functiondeplist (Lambda(nil,t,T)) )) then nil else

let val MMM = matchtypest v T T2 in

let val M=mergematch (MMM) (matchobjectst v t t2)

in
(LOCALMATCHES:= mergematch M (!LOCALMATCHES);
```

```
if (!LOCALMATCHES)=[("",Ferror)] then [("",Ferror)]
else M)

end end end

 |



matchfunctionst v (Lambda((s,(i,m,n,b,TT))::B,t,T))
   (Lambda((s2,(i2,m2,n2,b2,TT2))::B2,t2,T2)) =

   if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


if not (find0 (v,length(!CONTEXT)-1,0)
(functiondeplist (Lambda((s,(i,m,n,b,TT))::B,t,T))))
then nil else

( (* sayD("Matching "^
(displayerrorfunction (Lambda((s,(i,m,n,b,TT))::B,t,T)))^
 " with "^(displayerrorfunction
(Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2)))); *)

if islambda2 TT then

matchfunctionst v (subsfunction0 (n+1) s m n b TT
(Lambda(B,t,T)))
(Lambda((s2,(i2,m2,n2,b2,TT2))::B2,t2,T2))

else if islambda2 TT2 then

matchfunctionst v (Lambda((s,(i,m,n,b,TT))::B,t,T))
(subsfunction0 (n2+1) s2 m2 n2 b2 TT2 (Lambda(B2,t2,T2)))

else (let val S = extend (s) and S2 = extend (s2) in
```

```
let val TTT = matchfunctionst v TT TT2 in

(CONTEXT:=((S,(1,length(!CONTEXT)-1,0,b,TT))::
(S2,(1,length(!CONTEXT)-1,0,b2,TT2))
::(Hd nil (!CONTEXT)))::(Tl(!CONTEXT));
addtomatches S
(referentof(S2,length(!CONTEXT)-1,0,b2,TT2))(!LOCALMATCHES);
let val MMM =(mergematch(TTT)
(matchfunctionst v (subsfunction0 (n+1) s m n b
(referentof(S,(length(!CONTEXT)-1),0,b,TT)) (Lambda(B,t,T)))
(subsfunction0 (n2+1) s2 m2 n2 b2
(referentof(S2,length(!CONTEXT)-1,0,b2,TT2))
 (Lambda(B2,t2,T2)))))
in (LOCALMATCHES:=mergematch MMM (!LOCALMATCHES);
if (!LOCALMATCHES)=[("",Ferror)] then [("",Ferror)]
else MMM) end

) end

end)) |

matchfunctionst v (Lambda(nil,t,T))
 (Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2)) =

 if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


 if not (find0 (v,length(!CONTEXT)-1,0)
(functiondeplist (Lambda(nil,t,T)))) then nil else

 ( (* sayD("Matching "
^(displayerrorfunction (Lambda(nil,t,T)))^
 " with "^(displayerrorfunction
(Lambda((s2,(i,m2,n2,b2,TT2))::B2,t2,T2)))); *)

if islambda2 TT2 then
```

```
matchfunctionst v (Lambda(nil,t,T))
(subsfunction0 (n2+1) s2 m2 n2 b2 TT2 (Lambda(B2,t2,T2)))

else ( (* sayD "matchfail 4602"; *)matchesfail())) |

matchfunctionst v (Lambda((s,(i,m,n,b,TT))::B,t,T))
   (Lambda(nil,t2,T2)) =

   if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


if not (find0 (v,length(!CONTEXT)-1,0) (functiondeplist
(Lambda((s,(i,m,n,b,TT))::B,t,T)))) then nil else

( sayD("Matching "
^(displayerrorfunction (Lambda((s,(i,m,n,b,TT))::B,t,T)))^
 " with "^(displayerrorfunction (Lambda(nil,t2,T2))));

if islambda2 TT

then matchfunctionst v
(subsfunction0 (n+1) s m n b TT (Lambda(B,t,T)))
(Lambda(nil,t2,T2))

else ( (* sayD "matchesfail 4623"; *) matchesfail())) |

matchfunctionst v x y =
if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


if  x=y andalso x <> Ferror then nil

else

let val A = defexpand2 y in
```

```
if A <> y then matchfunctionst v x A

else let val B = Etared 0 y in

if B <>y then matchfunctionst v x B

else let val C = Etared 0 x in

if C <> x then matchfunctionst v C y

else ( (* sayD "matchesfail 4646"; *) matchesfail()) end end end

and matchfunctionlistst v nil nil = nil |

matchfunctionlistst v (s::L) (t::M) =

if find Ferror v (!LOCALMATCHES) <> Ferror
then (!LOCALMATCHES) else


let val MM = matchfunctionst v s t in

(LOCALMATCHES := mergematch (MM) (!LOCALMATCHES);
if (!LOCALMATCHES) = [("",Ferror)] then [("",Ferror)]
else mergematch(MM) (matchfunctionlistst v L M)) end |

matchfunctionlistst v x y =

( (* sayD "matchesfail 4664"; *) matchesfail());



(* start working on fixing explicit argument lists *)

(* generate a list of fresh free variables
from the binders in a function sort or lambda term *)

fun openlambda nil = nil |
```

```
openlambda ((s,(i,m,n,b,TT))::B) =

let val S = extend (s) in

((S,(i,length(!CONTEXT),0,b,TT))::((map (fn (s2,(i2,m2,n2,b2,TT2)) =>
(s2,(i2,m2,n2,b2,subsfunction0 (n+1) s m n b
(referentof(S,length(!CONTEXT),0,b,TT)) TT2))) (openlambda B))))

end;

(* evaluate an implicit argument by matching types
in the first place in the given list of arguments where
this argument appears in the type *)

(* fun findfirstmatchD s B L =

(say2D s; say2D (displaybinderlist B); say2D (displayargumentlist L);
map(say2D)(map (displayerrorfunction)(map functionsort L));
let val M = findfirstmatch s B L in if M = Ferror
then say ("Failed to find implicit argument "^s^
" using binder list "^(displaybinderlist B)
^" and argument list "^(displayargumentlist L)
^"; "^(showmatches())) else ();M end) *)

fun findfirstmatch s x nil =
(say "findfirstmatch runs out of format information line 3796"
;Ferror) (* runs out of format information *) |

findfirstmatch s nil x =
(say "format is too long in findfirstmatch? line 3800";Ferror) |

findfirstmatch s ((s2,(i,m2,n2,b2,TT2))::L1) (x::L2) =

if islambda2 TT2 then

(* expand out let arguments *)
```

```
findfirstmatch s (map
(fn (s3,(i3,m3,n3,b3,TT3)) =>
(s3,(i3,m3,n3,b3,
subsfunction0 (n2+1) s2 m2 n2 b2 TT2 TT3))) L1)
(x::L2)

else if b2 andalso
find0 (s,length(!CONTEXT)-1,0) (functiondeplist TT2) then
( (*LOCALMATCHES:=nil; *) (* no commitments to global matches *)
(matchfunctionst s (TT2) (functionsort x));
find Ferror  s (!LOCALMATCHES))

else if (not b2) andalso
  find0 (s,length(!CONTEXT)-1,0) (functiondeplist TT2) then

let val X = findfirstmatch s2 L1 (x::L2) in

((* LOCALMATCHES:=nil; *)
(matchfunctionst s (TT2) (functionsort X));
find Ferror s (!LOCALMATCHES))

end

else if not b2
then findfirstmatch s L1 (x::L2)
 else findfirstmatch s L1 L2;

(* the first argument of this function is the format list;
the second argument
is the list of explicitly given arguments *)

(* when all explicitly given arguments are processed, you are done;
there might be subsequent let terms in the format but they will
not be used *)

fun findallarguments x nil = nil |

findallarguments nil L =
```

```
(say "ran out of type information in findallarguments line 3840";
[Ferror]) |

findallarguments ((s,(i,m,n,b,TT))::L1) (x::L2) =

(* this is the case where the first argument is explicit *)

(CONTEXT:= (((s,(i,m,n,b,TT))::L1))::(!CONTEXT);
CONTEXTNAMES:= (makestring(length(!CONTEXT)-1))::(!CONTEXTNAMES);

if b andalso not(islambda2 TT) then

( (* sayD ("Setting explicit argument "^s^" to "^(displayerrorfunction x)); *)
purgecaches();CONTEXT:=tl(!CONTEXT);
CONTEXTNAMES:=tl(!CONTEXTNAMES);

x::(findallarguments

(let val XX = Subsfunction0 ~1 s m n b x
(Lambda(L1,Unknown,Tdummy)) in
if not(islambdaterm XX) then (say2 "substitution crash 1";nil)
else let val (Lambda(B2,t2,T2)) = XX in
B2 end end)
(L2)))


(* eliminate let term arguments from the format *)

else if islambda2 TT then

(purgecaches();CONTEXT:=tl(!CONTEXT);
CONTEXTNAMES:=tl(!CONTEXTNAMES);

findallarguments

(let val XX = Subsfunction0 ~1 s m n b TT
(Lambda(L1,Unknown,Tdummy)) in
if not(islambdaterm XX) then (say2 "substitution crash 2";nil)
```

```
else let val (Lambda(B2,t2,T2)) = XX in
B2 end end)

(x::L2))

(* find implicit arguments by
doing the earliest possible type match *)

else let val M = findfirstmatch s L1 (x::L2) in

( (* sayD ("Setting implicit argument "^s^" to "^(displayfunction M)); *)
purgecaches();CONTEXT:=tl(!CONTEXT);
CONTEXTNAMES:=tl(!CONTEXTNAMES);

M::(findallarguments

(let val XX = Subsfunction0 ~1 s m n b M
(Lambda(L1,Unknown,Tdummy)) in
if not(islambdaterm XX) then (say2 "substitution crash 3";nil)
else let val (Lambda(B2,t2,T2)) = XX in
B2 end end)

(x::L2)))

end);

fun findimplicitargs (App(s,m,n,b,L)) =

( (* sayD ("Finding implicit arguments for "
^(displayerrorobject (App(s,m,n,b,L)))); *)

if n<> 0 then App(s,m,n,b,L) else

let val F = lambdaformat(absolutesortof s m) in

(* all arguments are explicit or let terms *)

if not(find0 Implicit F) then
```

```
if objectsort(App(s,m,n,b,L))<>Ferror then

( (* sayD ("Reporting unchanged implit argument for"
^(displayerrorobject (App(s,m,n,b,L)))); *)
App(s,m,n,b,L))



else (say ("5300:  Object type error in "^(displayobject(App(s,m,n,b,L))));
Oerror)

else if F = [Implicit]
then (say "bad format error line 3855 (findimplicitargs)";Oerror)

else if absolutesortof s m = Ferror
then (say "declaration error line 3857 (findimplicitargs)";Oerror) else

let val (Lambda(B,t,T)) = absolutesortof s m in

let val LL = ( LOCALMATCHES:=nil; findallarguments (openlambda B) L )in

if objectsort (App(s,m,n,b,LL)) <> Ferror then

( (* sayD ("Reporting corrected arguments "
^(displayerrorobject(App(s,m,n,b,LL)))); *)
App(s,m,n,b,LL) )

else (say ("5317:  Object type error in "^(displayobject(App(s,m,n,b,LL))));
Oerror)

end

end end) |

findimplicitargs T =
(say "weird findimplicitargs failure line 3871";Oerror);
```

```
(*
```

10.5 Other type utilities: checking object sorts, definition expansion, inserting output types into function terms

The subsection title describes what is here.

The `typefix` function is the postprocessor for terms which adds type information and implicit arguments. In the previous implementation, all of this is jammed into the parser. The `typefix` function can only be run on a term in which bound variables have not yet been reindexed (in which of course they have to have the intended types).

```
*)

fun objectsortcheck (THAT t) = objectsort t = Lambda(nil,Unknown,PROP) |

objectsortcheck (IN t) = objectsort t = Lambda(nil,Unknown,TYPE) |

objectsortcheck (Rewrite(t,u)) = equalfunctions(objectsort t)(objectsort u) |

objectsortcheck Terror = false |

objectsortcheck t = true;



(* function for adding type information
and implicit arguments
to parsed lambda terms *)

fun typefixobject (App(s,m,n,b,L)) =

let val LL = (if (!STRONGEXPAND) then
map suppresstype else (fn x => x))  (map typefix L) in
```

```
(findimplicitargs(App(s,m,n,b,LL))) end |

typefixobject x = x

and typefixtype (THAT p) = let val TT = THAT (typefixobject p)
in if objectsortcheck TT then TT
else (say "bad proof/evidence type, body not prop line 3913";Terror) end |

typefixtype (IN tau) = let val TT = IN (typefixobject tau)
in if objectsortcheck TT then TT
else (say "bad typed object sort: body not type line 3916";Terror) end |

typefixtype (Rewrite(t,u)) =
let val TT = Rewrite(typefixobject t, typefixobject u) in
if objectsortcheck TT then TT
else (say "bad typed rewrite sort: line 5226";Terror) end |

typefixtype x = x

and typefix (Lambda(nil,App(s,m,n,b,L),Tdummy)) =

let val XX = typefixobject(App(s,m,n,b,L))

in

if XX = Oerror then
(say3 "Typefix failure with application term";Ferror)

else if n<> 0 orelse (islambda0(absolutesortof s m)) then

( (* sayD ("Encountered bad case "^(displayerrorobject XX)
^"; "^s^"; "^(makestring m)^"; "
^(makestring n)); *) Lambda(nil, XX,Tdummy))

else let val (App(S,M,N,B,LL)) = XX  in

betared 0 (Etared 0 (Fatom(s,m,0,b))) LL
```

```
end end |


typefix (Lambda(B,t,Tdummy)) =

let val t1 = typefixobject t

in let val T1 = objectsort t1 in

if theobjectsort T1 = Terror then
(say "failure to compute object sort in fixtypes";
Ferror)

else (Lambda(B,t1,theobjectsort T1)) end end |

(* typefix (Lambda(B,App(s,m,n,b,L),T))

= if equalfunctions (Fatom(s,m,n,b)) (Lambda(B,App(s,m,n,b,L),T))

then Fatom(s,m,n,b) else

Lambda(B,typefixobject (App(s,m,n,b,L)), typefixtype T) | *)

typefix (Lambda(B,t,T)) = Lambda(B,typefixobject t,typefixtype T) |

typefix x = x;

fun Getfunction s = typefix(getfunction(Tokenize s));

fun Getsort s = functionsort(Getfunction s);

(* fun matchtest f g = (LOCALMATCHES:=nil;map
(fn (s,F) => (say3 (s^":");say3(displayfunction F)))
(matchfunctions(typefix(getfunction(Tokenize f)))
(typefix(getfunction(Tokenize g))))); *)
```

(*

11 Lestrade declaration commands and the command interface; creation and execution of log files.

The code implementing command lines begins below.

In general, a declarative command line consists of an identifier naming the command, followed by a non-reserved identifier, followed sometimes by an open term list (usually but not always an argument list), followed sometimes by a colon, followed by an object or sort term (depending on the character of the command).

In the previous version, the condition that arguments taken from the next move must appear in the order in which they are declared is enforced. In this version, the implicit argument inference mechanism puts the arguments you supply into the order in which they are declared, without comment!

We refer to a non-reserved identifier which has not been declared as a name as a potential name.

It can be noted that we find no reason to treat Lestrade command names as reserved identifiers.

What follows is a description of the command lines in the original implementation, which we intend to implement later.

A `declare` command line consists of `declare` followed by a potential name followed by an object or function sort term.

A `postulate` command consists of `postulate` followed by a potential name followed optionally by an open argument list followed optionally by a colon followed by an object sort term.

A `define` command consists of `define` followed by a potential name followed optionally by an open argument list, followed by a colon (not optional!). followed by an object term.

A rewrite command consists of `rewritep` or `rewrited` followed by an identifier (a name in the case of `rewrited` and a potential name in the case of `rewritec`) followed by an open term list (with very specific requirements of form dictated by the commands). As of March 2020, rewriting is not supported in this version.

There are other command lines which do not (directly) introduce or modify declarations.

Important examples are `open` followed optionally by an identifier, `close` followed optionally by an identifier, `clearcurrent` followed optionally by an

identifier, and save followed optionally by an identifier.

```
*)

(*  beginning test declaration commands *)

val SHOWFULL = ref false;

fun showmove n = say2(
reprocess(length(!CONTEXT)-1)
("{move "^(makestring n)^
(if worldname n = makestring n
then "}" else ": "^(worldname n)^"}")));

fun Showdec s = (if length(!CONTEXT) = 2 orelse (!SHOWFULL)
then say3(reprocess (length(!CONTEXT)-1)
(s^": "^(displayfunction(sortof s)))) else ()
;if islambda2 (sortof s)
then say3(reprocess(length(!CONTEXT)-1)(s^": "^
(displayfunction(subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ)))(functionsort (sortof s))))))
else (
if length(!CONTEXT)<>2 andalso not (!SHOWFULL)
then say3(reprocess (length(!CONTEXT)-1)
(s^": "^(displayfunction(sortof s)))) else ()
);showmove(env s))

fun Showdecfull s = ((* if length(!CONTEXT) = 2 then*)
say3(reprocess (length(!CONTEXT)-1)
(s^": "^(displayfunction(sortof s)))) (* else () *)
;if islambda2 (sortof s)
then say3(reprocess(length(!CONTEXT)-1)(s^": "^
(displayfunction(subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ)))(functionsort (sortof s))))))
```

```
else ( (*
if length(!CONTEXT)<>2 then say3(reprocess (length(!CONTEXT)-1)
(s^": "^(displayfunction(sortof s)))) else () *)
);showmove(env s))

fun Showdec2 s m = (say3(reprocess (length(!CONTEXT)-1)
(s^": "^(displayfunction(absolutesortof s m))))
;if islambda2(absolutesortof s m)
then say3(reprocess(length(!CONTEXT)-1)(s^": "^
(displayfunction(subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ)))(functionsort (absolutesortof s m))))))
else ();showmove(m))

fun Declare s t =

if s="" orelse Restidentifier s <> nil
orelse isreserved s orelse arity s <> ~1

then say (s
^" is badly formed or already reserved or declared")

else if Tokenize t = nil
orelse restfunction(Tokenize t) <> nil

then say (t^" is not well-formed")

else let val XX =  subsfunction0 0 "?!?" 0 0 true
(referentof("?!?",0,0,true,Lambda(nil,Unknown,OBJ)))
(typefix(getfunction(Tokenize t))) in
((* say3(displayfunction XX); *)
if islambdaterm XX then let val (Lambda(B,T1,T2)) = XX in

if (* map (fn (s,x) => arity s <> ~1) B = map (fn (s,x) => true) B

andalso  (T1 = Unknown
orelse theobjectsort(objectsort T1) <> Terror)
```

```
andalso (T2 = Tdummy orelse objectsortcheck T2) *) 0=0

then

let val X = (* subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ))) *) ((Lambda(B,T1,T2)))

in

if functionerrorsfound X
then say("Errors found in declaring "
^s^" as "^(displayfunction X))

else let val fourx = typesimplify X in

if islambda2 (fourx) andalso
functionsort (fourx) = Ferror

then say ("Invalid sort in Declare command")

else

(CONTEXT:=

(add s (length(Hd nil (!CONTEXT)),
length(!CONTEXT)-1,0,true,
subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ))) (fourx)) (Hd nil (!CONTEXT)))
::(Tl(!CONTEXT));


(* FORMATS :=
add (s,length(!CONTEXT)-1)
(lambdaformat X) (!FORMATS); *)

Showdec s) end end
```

```
else say ("Some sort of declaration or type failure in declaring "
^s^" as "^(displayfunction(subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,Lambda(nil,Unknown,OBJ)))
 (typefix(Lambda(B,T1,T2))))))  end else
 say ("Parse or typefix error in"
 ^(displayfunction(getfunction(Tokenize t))))) end;

fun Posit s t =

if s="" orelse Restidentifier s <> nil
orelse isreserved s orelse arity s <> ~1

then say (s
^" is badly formed or already reserved or declared")

else if Tokenize t = nil
orelse restfunction(Tokenize t) <> nil

then say (t^" is not well-formed")

else let val XX = ((* sayD "got to 5276";*)subsfunction0 0 "?!?" 0 0 true
(referentof("?!?",0,0,true,Lambda(nil,Unknown,OBJ)))
(implicitarglist(typefix(getfunction(Tokenize t))))) in

( (* say3(displayfunction XX); *) if islambdaterm XX
then let val (Lambda(B,T1,T2)) = XX in

if (* map (fn (s,x) => arity s <> ~1) B =
map (fn (s,x) => true) B

andalso  (T1 = Unknown
orelse theobjectsort(objectsort T1) <> Terror)

andalso (T2 = Tdummy orelse objectsortcheck T2) *) 0=0

then
```

```
let val X = (* subsfunction0 0 "(?!?"  0 0  true
(referentof("?!?",0,0,true,Lambda(nil,Unknown,OBJ)))
 (typefix *)(Lambda(B,T1,T2))

in

if functionerrorsfound X
then say("Errors found in declaring "
^s^" as "^(displayfunction X))

else if let val DEPS = ((* sayD "got to 5302" ; *)drop0 ""
(map (fn (s,m,n) =>
if m = length(!CONTEXT)-1 andalso n=0 then s else "")
(functiondeplist X)))

in

((* sayD "got to 5309" ; *)map (fn x => find0 x (map p1 B)) DEPS <>
map (fn x => true) DEPS)

end

then (map say2D (map p1 B);

say2D "========";

map say2D
(drop0 ""
(map (fn (s,m,n) =>
if m = length(!CONTEXT)-1 andalso n=0 then s else "")
(functiondeplist X)))
;
say("Cannot export "^(displayfunction X)^" to last move"))

else let val fourx = ( (* sayD "got to 5326"; *)typesimplify X) in


if (sayD "got to 5329";islambda2 (fourx) andalso
```

```
functionsort (fourx) = Ferror)

then say ("Invalid sort in Posit command")

else

((* sayD"got to 5336"; *)CONTEXT:=

(Hd nil (!CONTEXT))::
(add s (length(Hd nil (Tl(!CONTEXT))),
length(!CONTEXT)-2,0,true,
((* sayD"got to 5340"; *) subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,
Lambda(nil,Unknown,OBJ))) (fourx))) (Hd nil (Tl(!CONTEXT))))
::(Tl(Tl(!CONTEXT)));


(* FORMATS :=
add (s,length(!CONTEXT)-2)
(lambdaformat XX) (!FORMATS); *)

Showdec s) end end


else say ("Some sort of declaration or type failure in declaring "
^s^" as "^(displayfunction(subsfunction0 0 "?!?"  0 0  true
(referentof("?!?",0,0,true,Lambda(nil,Unknown,OBJ)))
 (typefix(Lambda(B,T1,T2))))))  end



 else
 say ("Parse or typefix error in"^
 (displayfunction(getfunction(Tokenize t)))) )end;


(*
```

The code above implements new `Declare` and `Posit` user commands which will be used to implement the declaration commands of the original version.

Each of these commands takes a single argument, a function term. The `Declare` command introduces this function at the "next move", as a parameter for definitions. The `Posit` command introduces this function at the "last move", in effect as a primitive or defined notion (at least locally). When the function term argument is a type, one is declaring a function (or object, if the function term is 0-ary) of that type, and when the function term is actually a function or object, one is defining a function or object.

The binder lists of the function argument of these commands corresponds to the argument list in the format of the original commands. The `Readline` command below will include the original commands as syntactical variants of `Declare` and `Posit`.

The use of `Declare` and `Posit` to implement the original declaration commands has philosophical interest. The most basic version of Lestrade does not require (and originally did not permit) the user to enter any term with variable binding (function sort or lambda term) though she did have to read such terms. We regard it as interesting that this could be done. We did eventually introduce function sorts as arguments to `declare` and lambda terms as arguments to application terms, for convenience, but their use can always be eliminated in principle. Here, though, we act in reverse: we have reimplemented the legacy declaration commands in terms of new commands which take variable binding terms as arguments.

The practical use of variable binding cannot be disputed. But the philosophical interest of avoiding bound variables in favor of parameters remains. It can also be noted that the use of bound variables has some philosophical dishonesty in it, or at least hides details. All bound variables in a user entered term need to be taken from the next move. But analysis suggests that variables bound in multiple nested variable binding terms actually translate to declarations in further moves never actually opened. We think that such analysis may shed some light on what is going on in variable binding generally.

We now have the ability to save our activity to executable log files with the extension `.ltxt`. These are currently kept in a directory `LTXTs` to avoid clutter.

```
*)

(* a list manipulation needed for conversion
of legacy declaration commands without
parsing or display *)

fun replacetail L1 L2 nil = nil |

replacetail L1 L2 (x::M) = if M = L1 then x::L2
else x::(replacetail L1 L2 M);

fun decolon (":"::L) = L |

decolon L = L;

fun Showdecs() =
 map (fn L => map (fn (s,(i,m,n,L,T)) =>
(Showdec2 s m;say "")) L) (!CONTEXT);

fun Showmoves() =
(say3 "for open:";
map (fn (s,x) => if Tl s = (!CONTEXTNAMES)
then say3 (Hd "\n"  s) else ()) (!CONTEXTS);
say3 "for clearcurrent:";
map (fn (s,x) => if Tl s = Tl (!CONTEXTNAMES)
then say3 (Hd "\n"  s) else ()) (!CONTEXTS));

val ECHOING = ref true

fun Interface0() =

let val NEXTLINE = getline()

in

if  NEXTLINE = "quit\n"
then (output(stdOut,"\nquit\n");output(!LOGFILE,"\nquit\n"))
```

```
else ((* say3(reprocess2(length(!CONTEXT)-1)
(displaytokens(Tokenize NEXTLINE))); *)
say3(reprocess2(length(!CONTEXT)-1)
(displaytokens(Tokenize NEXTLINE)));
Readline (Tokenize NEXTLINE);Interface0())

end

and Interface() = (versiondate();say2 "";Interface0())

and shorten s = if s ="" then "" else implode(rev(Tl(rev(explode s))))

and Readfile0() =

let val NEXTLINE = getline2();

in

if  NEXTLINE = "quit\n" orelse NEXTLINE = "(* quit *)\n"
 then (output(!LOGFILE, "\n(* quit *)\n");
flushOut(!LOGFILE);closeOut(!LOGFILE);
LOGFILE:=openOut("default"))

else if not (!ECHOING) then

if NEXTLINE = "end Lestrade execution\n"
then (ECHOING:=true;output(!LOGFILE,"\n"
^(shorten NEXTLINE));Readfile0())

else let val NL = Tokenize NEXTLINE in

if NL = nil orelse NL = [">>>"]

then Readfile0() else if Hd "\n" NL = ">>>"

then ((* say3(reprocess2(length(!CONTEXT)-1)
(displaytokens(tl NL))); *)
```

```
say3(reprocess2(length(!CONTEXT)-1)(displaytokens NL));
Readline (Tl NL);Readfile0()) else Readfile0() end

else
(output(!LOGFILE, "\n"^(shorten NEXTLINE));
if NEXTLINE = "begin Lestrade execution\n"
then ECHOING:=false else();
Readfile0())

end

and Readfile() = (ECHOING:=true;Readfile0())


and Readline nil = () |

Readline ("Declare"::(s::L)) = Declare s (displaytokens L) |

(* declare is Declare restricted to
declaration of object of object or function type *)
(* Declare also handles some definitions *)

Readline ("declare"::(s::L)) =
let val T1 = gettype L
in if T1 <> Terror
then Declare s (displaytokens L)

else let val T2 = getfunction L
in if T2 <> Ferror andalso not(islambda2 T2)

then Declare s (displaytokens L)
else say "{declare command error}" end end |

Readline ("Posit"::(s::L)) = Posit s (displaytokens L) |

Readline ("postulate"::(s::L)) =
let val (B,LL) = getbindersE L in
```

```
Readline ("Posit"::s::(if B = nil then (decolon LL) else
("["::(replacetail LL (["=>"]@
(decolon LL)@["]"])  (L))))) end

 |

Readline ("define"::(s::L)) = let val (B,LL) = getbindersE L in

Readline ("Posit"::s::(if B = nil then (decolon LL) else
("["::(replacetail LL (["=>"]@
(decolon LL)@["]"])  (L))))) end

 |

Readline ["Showdec",s] = Showdecfull s |

Readline ["Showdecs"] = (Showdecs();
showmove(length(!CONTEXT)-1)) |

Readline ["Showmoves"] = (Showmoves();
showmove(length(!CONTEXT)-1)) |

Readline ["open"] =
(Open (makestring(length(!CONTEXT)));
showmove(length(!CONTEXT)-1)) |

Readline("open"::(s::L)) =  (Open s;
showmove(length(!CONTEXT)-1)) |

Readline ["close"] = (basicclose();
showmove(length(!CONTEXT)-1)) |

Readline ["save"] =
(save (makestring(length(!CONTEXT)-1));
showmove(length(!CONTEXT)-1)) |

Readline ("save"::(s::L)) =  (save s;
showmove(length(!CONTEXT)-1)) |
```

```
Readline ("Load"::(s::L)) =  (Load s;
showmove(length(!CONTEXT)-1)) |

Readline ["clearcurrent"] = (basicclearcurrent();
say2 ("{move "^(makestring(length(!CONTEXT)-1))^"}")) |

Readline("clearcurrent"::(s::L)) =  (clearcurrent s;
showmove(length(!CONTEXT)-1)) |

Readline ["Clearall"]  = (basicstart();
showmove(length(!CONTEXT)-1)) |

Readline ("goal":: L) =

let val T1 = typefixtype(gettype L) in
(say3(reprocess (length(!CONTEXT)-1) (displaytype T1));

showmove(length(!CONTEXT)-1)) end |

Readline ("test":: L) =

let val T1 = typefix(getfunction  L) in
(say3(reprocess (length(!CONTEXT)-1) (displayfunction T1));

showmove(length(!CONTEXT)-1)) end |

Readline("comment"::L)=
showmove(length(!CONTEXT)-1) |

Readline(["marginup"]) = (MARGIN:=(!MARGIN)+5;
showmove(length(!CONTEXT)-1)) |

Readline(["margindown"]) = (MARGIN:=max(20,(!MARGIN)-5);
showmove(length(!CONTEXT)-1)) |

Readline ["versiondate"] =
(versiondate();
```

```
showmove(length(!CONTEXT)-1)) |

Readline ["Zeroorone"] =
(Zeroorone();
say3("Let terms with "
^(makestring(!ZEROORONE))
^" occurrences expanded");
showmove(length(!CONTEXT)-1)) |

Readline ["Strongexpand"] =
(Strongexpand();
say ("Strong expansion toggled "
^(if (!STRONGEXPAND) then "on" else "off"));
showmove(length(!CONTEXT)-1)) |

Readline ["diagnostic"] =
(diagnostic();
showmove(length(!CONTEXT)-1)) |

(* for the moment, these can only really be
called in the interface, not in an executed
log file *)

Readline ["Readbook",s,t] = Readbook s t|

Readline ["Readtex",s,t] = Readtex s t|

Readline ["Readonbook",s,t] = Readonbook s t|

Readline ["Readontex",s,t] = Readontex s t|

(* toggle display of definition bodies in
local environments *)

Readline["Showfull"] = (SHOWFULL := not(!SHOWFULL);
showmove(length(!CONTEXT)-1)) |
```

```
Readline x = let val T1 = typefix(getfunction x) in
(say3(displayfunction T1);
let val T2 = silentfunctionsort T1 in
if T2 = Ferror then () else
say3(displayfunction T2) end;
showmove(length(!CONTEXT)-1)) end

and Readonbook x y =
if not (fileexists (x^".ltxt"))
then say "There is no such book" else
(READFILE:=
 openIn("LTXTs\\"^x^".ltxt");flushOut(!LOGFILE);
closeOut(!LOGFILE);LOGFILE:=openOut("LTXTs\\"^y^".ltxt");
Readfile();flushOut(!LOGFILE);
closeOut(!LOGFILE);LOGFILE:=openOut("default");
SAVED:=(x,(!CONTEXT,!CONTEXTNAMES,!CONTEXTS))
::(drop x (!SAVED)))

and Readontex x y =
if not (fileexists ("LTXTs\\"^x^".tex"))
 then say "There is no such book" else
(READFILE:=
openIn("LTXTs\\"^x^".tex");
flushOut(!LOGFILE);
closeOut(!LOGFILE);
LOGFILE:=openOut("LTXTs\\"^y^".tex");
Readfile();flushOut(!LOGFILE);
closeOut(!LOGFILE);closeIn(!READFILE); LOGFILE:=openOut("default");
SAVED:=(x,(!CONTEXT,!CONTEXTNAMES,!CONTEXTS))
::(drop x (!SAVED)))

and Readbook x y =

if not (fileexists ("LTXTs\\"^x^".ltxt"))
then say "There is no such book" else
(basicstart();READFILE:=
 openIn("LTXTs\\"^x^".ltxt");flushOut(!LOGFILE);
closeOut(!LOGFILE);LOGFILE:=openOut("LTXTs\\"^y^".ltxt");
```

```
Readfile();flushOut(!LOGFILE);
closeOut(!LOGFILE);LOGFILE:=openOut("default");
SAVED:=(x,(!CONTEXT,!CONTEXTNAMES,!CONTEXTS))
::(drop x (!SAVED)))

and Readtex x y =

if not (fileexists ("LTXTs\\"^x^".tex"))
then say "There is no such book" else
(basicstart();READFILE:=
openIn("LTXTs\\"^x^".tex");
flushOut(!LOGFILE);
closeOut(!LOGFILE);
LOGFILE:=openOut("LTXTs\\"^y^".tex");
Readfile();flushOut(!LOGFILE);
closeOut(!LOGFILE);closeIn(!READFILE);LOGFILE:=openOut("default");
SAVED:=(x,(!CONTEXT,!CONTEXTNAMES,!CONTEXTS))
::(drop x (!SAVED)));


(*
```

`Showdec` will show the declaration of a name. `Showdecs` will show all declarations (hit return to get the next one). `Showmoves` will show names of available next moves.

Above is the implementation of the command interface and the file reading commands. The legacy declaration commands are implemented by text manipulation (at the token level, without parsing or display which might be dangerous) producing `Declare` or `Posit` commands as appropriate.

The items in the definition of `Readline` provide a quick and dirty summary of the commands available in the interface or in executable log files. The `Readbook` and `Readtex` files should not for the moment be run in log files.

It is worth noting that Lestrade does not have very many commands. The entries in the definition of `Readline` allow one to review the small collection of commands.