Sample interactions with a dependent type checker used as a theorem prover

Randall Holmes

10/24/2017

In this talk I'm going to discuss a logical framework (called Lestrade) implemented by a piece of software called the Lestrade Type Inspector (if I may be forgiven for exploiting literary associations of my name).

The framework Lestrade is a dependent type theory. Rather than explain discursively what this is (though there will be some discursive explanation) we will attempt to exhibit what it means by showing interactions with the system.

Things we talk about in Lestrade we call *entities* when we are being completely nonspecific.

Entities are divided into two species, *objects* and *functions*.

The species are further divided into sorts. We say sorts rather than types because the word "type" is reserved for specific sorts, as we will see shortly, but we may slip up.

There are five varieties of object sort.

There is a sort `prop` inhabited by propositions (yes, statements).

For each proposition p of sort `prop`, there is a sort (`that` p) inhabited by evidence for p (one might say, "proofs of p", but we are deliberately vaguer about this).

There is a sort `obj` inhabited by generic mathematical objects. In an implementation of ZFC, the sets would be of sort `obj` (presence of this sort supports "type-free" reasoning though an implementation of ZFC under Lestrade would not be type-free, as the proofs/evidence for propositions would be sorted as usual).

There is a sort `type` inhabited by "type labels" and for each type label τ an associated sort (`in` τ) inhabited by objects of type τ. For example, there might be a type label `Nat` and an object 2 of sort `in Nat`.

Function sorts are more complicated, and this is always the point in an exposition of Lestrade where something horrible and indigestible appears – along with a suggestion from me that one can pass over this quickly and refer back to it in the course of reading later examples which are kinder and gentler.

A Lestrade function f takes a fixed finite list of arguments $x_1, \ldots, x_n$, with each x_i being of an object or function type τ_i and any value $f(x_1, \ldots, x_n)$ being of an object sort τ (notice that all outputs of functions are objects). That much doesn't sound evil, but wait. Each τ_i may depend on any or all x_j's with $j < i$, and τ may depend on any or all of the x_i's.

Our notation for the type of f above (in which the x_i's are bound variables) is

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

We give an extended example of dependent typing which may also hint at other wickedness we are up to.

We consider a universally quantified statement $(\forall x \in \tau : \phi(x))$ and set out to determine the sort of the universal quantifier $\forall$. We begin with x, which is apparently of type τ. Backtrack a little: τ is of sort `type` and x is of sort `in` τ. Now $\phi(x)$ is a proposition, so ϕ is a function from `in` τ to `prop`. So $\forall$ is a function which takes a first argument τ (a domain of quantification) and a second argument which is a function from (`in` τ) to `prop` (note the dependence of its sort on the sort of the first argument) and returns a proposition (of sort `prop`).

We make these declarations under Lestrade.

```
declare tau type

>> tau: type {move 1}


declare x in tau

>> x: in tau {move 1}


open

     declare x1 in tau

>>      x1: in tau {move 2}


     construct phi x1  prop

>>      phi: [(x1_1:in tau) => (---:prop)]
>>        {move 1}


     close

construct Forall tau phi prop

>> Forall: [(tau_1:type),(phi_1:[(x1_2:in tau_1)
>>             => (---:prop)])
>>        => (---:prop)]
>>   {move 0}
```

On the preceding slide, I wrote the `declare` and `construct` lines, and the Lestrade Type Inspector wrote the replies (I can run a LaTeX file as a Lestrade script, and the Inspector will engage in proper dialogue when presented with Lestrade commands and echo the LaTeX context back into the file: it is cute!)

There is quite a lot going on on the slide; for the moment, notice the dependent type of `Forall`.

We perpetrate further horrors on the next slide.

```
open

     declare x1 in tau

>>       x1: in tau {move 2}


     construct univev x1 that phi x1

>>       univev: [(x1_1:in tau) => (---:that
>>                phi(x1_1))]
>>         {move 1}


     close

construct Ug tau phi, univev that Forall tau phi

>> Ug: [(tau_1:type),(phi_1:[(x1_2:in tau_1)
>>                => (---:prop)]),
>>          (univev_1:[(x1_3:in tau_1) => (---:that
>>                phi_1(x1_3))])
>>          => (---:that (tau_1 Forall phi_1))]
>>    {move 0}
```

In the previous slide, we present the definition of the universal generalization rule of logic as a dependently typed function, which is both a suitably horrid example of dependent typing, and something which furthers other parts of our agenda which are so far implicit.

We give a formal description of function sorts and how to sort a Lestrade term, which, as foreshadowed, is indigestible and horrible. Let $U[t/v]$ denote the result of replacing the variable v with t in U.

A function sort is of the form

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

where each x_i is a variable (bound in the term), the sort of each x_i is τ_i, which may be an object or function sort, and τ is an object sort. Each x_i may occur in τ_j only if $i < j$. These notations appear in Lestrade output but not in user input.

A Lestrade object term is either atomic, in which case its type can be looked up, or it is an object term of the form $f(t_1, \ldots, t_n)$, where each t_i is an object or function term, and f is an atomic function term of sort

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

(notice the agreement in number of arguments). The sort of t_1 must match the sort τ_1 (up to definitional expansion and rewriting): if $n = 1$ the type of $f(t_1)$ is then $\tau[t_1/x_1]$; otherwise it is the sort (if any) of $f^*(t_2, \ldots, t_n)$ where f^* is postulated with sort

$$[(x_2 : \tau_2[t_1/x_1]), \ldots, (x_n, \tau_n[t_1/x_1]) \Rightarrow (- : \tau[t_1/x_1])].$$

A Lestrade function term can be atomic, in which case its sort can be looked up. Function terms occurring in applied position are always atomic: if a nonatomic functional term replaces a variable in applied position, the nonatomic term is eliminated by definitional expansion as part of the substitution process.

It can be "curried": if f is an arity n function and $m < n$, $f(t_1, \ldots, t_m)$ denotes

$$[x_{m+1} \ldots, x_n \Rightarrow f(t_1, \ldots, t_m, x_{m+1}, \ldots, x_n)].$$

Such terms occur only as arguments.

It can be an anonymous function term with variable binding: $[x_1, \ldots, x_n \Rightarrow T]$, which we will explain when we have discussed the Lestrade declaration context and definitions. Such terms occur only as arguments.

An object sort term is of one of the forms `prop`, `type`, `obj`, `that` p where p is an object term of sort `prop`, or `in` τ where τ is an object term of sort `type`.

A function sort term can be of the form

$$[x_1, \ldots, x_n \Rightarrow \tau],$$

representing the sort

$$[(x_1 : \tau_1^*), \ldots, (x_n : \tau_n^*) \Rightarrow \tau^*],$$

where τ_i is the sort of x_i (which is a variable in the local environment), τ^* is the result of replacing each x_i with x_i^* in τ, and each τ_i^* is the result of replacing x_j with x_j^* for each $j < i$ in τ_i.

The Lestrade declaration environment is designed to support the definition of atomic function identifiers in the parameterized style

$$f(x_1, \ldots, x_n) = T,$$

where the x_i's are variable parameters and T is an object term.

The declaration environment contains identifiers with associated sort and definition information. The set of declarations is partitioned into moves indexed by natural numbers: the moves at any particular point are indexed by $0, 1, \ldots, i, i+1$, where i is a parameter. There are always at least two moves. Move i is called the "last move" and move $i+1$ is called the "next move".

The underlying idea behind the moves is that entities in any move are variable relative to entities at all previous moves. Entities declared at move 0 are things to which the user is committed. When i has a value higher than 0, we have temporarily fixed all objects declared at moves 1 to i and are varying those declared at move $i+1$. The way we use the variables declared at move $i+1$ is as parameters in declarations or definitions of new identifiers to be declared at move i.

Relativity of variables is not unusual in ordinary mathematics: a term $ax+by$ in a calculus text probably has x, y in effect at move 2 and a, b at move 1.

Here are Lestrade commands which make global changes to the declaration environment.

The `open` command increments i, creates a new empty next move, and causes the former next move to become the last move.

The `close` command does nothing if $i = 0$; if I is positive it decrements i and discards all declarations at the last move. The former last move becomes the next move.

The `clearcurrent` command clears all declarations in the next move without modifying i: this is needed as an independent command to be able to clear declarations at move 1.

A *variable* is precisely an identifier which is declared but not defined in the next move.

Declarations in each move are sorted in the order of declaration. The argument list to any declaration command must have its variables in the order in which they were declared (this is a cheap way to enforce dependency restrictions).

The `declare` command, whose operands are a fresh identifier and a sort term, declares a new variable (in the next move, of course) with the given sort.

```
declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}
```

The `construct` command, whose operands are a fresh identifier f, an argument list $x_1, \ldots, x_n$, and an object sort τ, declares the new identifier f as a function with sort

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

where each τ_i is the type recorded for the variable x_i. The declaration of f is recorded at the last move. If the argument list is null, f is declared as an object of type τ at the last move. This is how to declare axioms and primitive notions under Lestrade. In any instance of the `construct` command, any variable on which the sort of any argument depends or on which the object sort operand depends must appear in the argument list.

```
construct ?? prop

>> ??: prop {move 0}


construct & p q prop

>> &: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}


construct -> p q prop

>> ->: [(p_1:prop),(q_1:prop) => (---:prop)]
>>   {move 0}
```

The `define` command, whose operands are a fresh identifier f, an argument list $x_1, \ldots, x_n$, and an object term T, in effect defines the new identifier as the function satisfying $f(x_1, \ldots, x_n) = T$. The sort information recorded for f (at the last move) is

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

the actual sort of f is obtained by replacing T with $-$, but this is a convenient way to store the definition body. The system also uses $[(x_1 : \tau_1), \ldots, (x_n : \tau_n) \Rightarrow (T : \tau)]$ as an anonymous internal notation for f.

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

>> ~: [(p_1:prop) => ((p_1 -> ??):prop)]
>>   {move 0}


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

>> <->: [(p_1:prop),(q_1:prop) => (((p_1 ->
>>         q_1) & (q_1 -> p_1)):prop)]
>>   {move 0}
```

We describe the computation of the term

$$f(t_1, \ldots, t_n).$$

It will only compute if it sorts successfully. If $n = 1$ the value is $T[t_1/x_1]$; otherwise the value is the value computed for $f^*(t_2, \ldots, t_n)$ where f^* is postulated with internal notation

$$[(x_2 : \tau_2[t_1/x_1]), \ldots, (x_n : \tau_n[t_1/x_1])$$

$$\Rightarrow (T[t_1/x_1] : \tau[t_1/x_1])].$$

If the argument list is null, f is defined as synonymous with T. In any instance of the `define` command, any variable on which the sort of any argument depends or on which the object term operand or its sort depends must appear in the argument list.

We can report that a term $[x_1, \ldots, x_n \Rightarrow T]$ will refer to the function f with internal information

$$[(x_1^* : \tau_1^*), \ldots, (x_n^* : \tau_n^*) \Rightarrow (T^* : \tau^*)] :$$

as in user-entered function sort terms, starring signals that x_i^* replaces x_i in T, in τ, and in τ_j for $j > i$. Note that bound variables in either function or function sort terms must be variables declared at the next move.

It is an interesting point that the original implementation of Lestrade did not allow user-entered terms with bound variables at all, and that, though it could be a bit long winded, all applications of such terms can be implemented with parameterized declarations and definitions of atomic function terms.

A very important note is that defined identifiers at the next move must be eliminated from sort information saved to the last move due to a `construct` or `define` command, by definitional expansion where possible, or by replacement with the anonymous function term form in the case of functions in argument position. The reason for this is that if the next move were closed, the reference of any such identifiers would be lost.

Some examples of this process can be seen in the displays in the last proof at the end of these slides: as more moves are closed toward the end of the proof, more concepts defined in those moves are expanded out.

We continue with a little development of logic. We have already declared a selection of propositional connectives. We will declare rules of inference for these connectives as dependently typed functions and carry out a baby proof or two.

```
declare pp that p

>> pp: that p {move 1}


declare qq that q

>> qq: that q {move 1}


declare pq that p & q

>> pq: that (p & q) {move 1}


construct Conjunction pp qq that p & q

>> Conjunction: [(.p_1:prop),(pp_1:that .p_1),
>>         (.q_1:prop),(qq_1:that .q_1) => (---:
>>         that (.p_1 & .q_1))]
>>   {move 0}


construct Simplification1 pq that p

>> Simplification1: [(.p_1:prop),(.q_1:prop),
>>         (pq_1:that (.p_1 & .q_1)) => (---:that
>>         .p_1)]
>>   {move 0}


construct Simplification2 pq that q

>> Simplification2: [(.p_1:prop),(.q_1:prop),
>>         (pq_1:that (.p_1 & .q_1)) => (---:that
>>         .q_1)]
```

```
>>   {move 0}


define Conjcomm pq : \
    Conjunction(Simplification2 pq, Simplification1 pq)

>> Conjcomm: [(.p_1:prop),(.q_1:prop),(pq_1:
>>        that (.p_1 & .q_1)) => ((Simplification2(pq_1)
>>        Conjunction Simplification1(pq_1)):that
>>        (.q_1 & .p_1))]
>>   {move 0}
```

We present the primitive rules for conjunction, and define a rule of inference `Conjcomm`.

We note without having time to comment at too much length on a practical feature of Lestrade. Officially the function `Conjunction` must have four arguments, p, q, pp, qq and if you look at its sort information, it does.

However, the need for the arguments p and q can be deduced from the explicitly given arguments, and the values of p and q can be deduced from the sorts of arguments supplied to `Conjunction`: Lestrade supports "implicit argument inference", not requiring the user to explicitly supply arguments that it can infer.

Next we present rules for implication.

```
clearcurrent

declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


declare pp that p

>> pp: that p {move 1}


declare pq that p -> q

>> pq: that (p -> q) {move 1}


construct Mp  pp pq that q

>> Mp: [(.p_1:prop),(pp_1:that .p_1),(.q_1:prop),
>>        (pq_1:that (.p_1 -> .q_1)) => (---:that
>>        .q_1)]
>>   {move 0}
```

```
declare ded  [pp => that q]

>> ded: [(pp_1:that p) => (---:that q)]
>>   {move 1}


construct Deduction ded that p->q

>> Deduction: [(.p_1:prop),(.q_1:prop),(ded_1:
>>        [(pp_2:that .p_1) => (---:that .q_1)])
>>        => (---:that (.p_1 -> .q_1))]
>>   {move 0}
```

Modus ponens and the deduction theorem should be recognizable.

The declaration of `ded` with an explicit function sort is slick: a little open/close block could also do it.

We present a quick proof motivated by `Conjcomm`.

```
declare pandq that p & q

>> pandq: that (p & q) {move 1}


define Conjcomm2 p q : Deduction [pandq => Conjcomm pandq]

>> Conjcomm2: [(p_1:prop),(q_1:prop) => (Deduction([(pandq_2:
>>              that (p_1 & q_1)) => (Conjcomm(pandq_2):
>>              that (q_1 & p_1))])
>>         :that ((p_1 & q_1) -> (q_1 & p_1)))]
>>   {move 0}
```

Note the use of `[pandq => Conjcomm pandq]` as an argument. It's worth noting that this is not the same function as `Conjcomm`: the latter function has two additional implicit arguments. It would be possible to introduce the function `[pandq => Conjcomm pandq]` using an open/close block, but the use of an anonymous function term is more efficient.

We present an extended proof of a theorem of propositional logic.

```
clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}


declare C prop

>> C: prop {move 1}


% prove ((A -> B) & (B -> C)) -> (A -> C)

open

     declare hyp1 that (A -> B) & (B -> C)

>>       hyp1: that ((A -> B) & (B -> C)) {move
>>         2}


     define line1 hyp1 : Simplification1 hyp1

>>       line1: [(hyp1_1:that ((A -> B) & (B
>>              -> C))) => (Simplification1(hyp1_1):
>>              that (A -> B))]
>>         {move 1}


     define line2 hyp1 : Simplification2 hyp1
```

```
>>      line2: [(hyp1_1:that ((A -> B) & (B
>>             -> C))) => (Simplification2(hyp1_1):
>>             that (B -> C))]
>>        {move 1}


     % now suppose A to prove A -> C

     open

          declare hyp2 that A

>>           hyp2: that A {move 3}


          define line3 hyp2 : Mp hyp2 (line1 hyp1)

>>           line3: [(hyp2_1:that A) => ((hyp2_1
>>                  Mp line1(hyp1)):that B)]
>>             {move 2}


          define line4 hyp2 : Mp (line3 hyp2, line2 hyp1)

>>           line4: [(hyp2_1:that A) => ((line3(hyp2_1)
>>                  Mp line2(hyp1)):that C)]
>>             {move 2}


          close

     define line5 hyp1 : Deduction line4

>>      line5: [(hyp1_1:that ((A -> B) & (B
>>             -> C))) => (Deduction([(hyp2_2:
```

```
>>                   that A) => (((hyp2_2 Mp line1(hyp1_1))
>>                   Mp line2(hyp1_1)):that C)])
>>              :that (A -> C))]
>>         {move 1}


     close

define Transimp A B C : Deduction line5

>> Transimp: [(A_1:prop),(B_1:prop),(C_1:prop)
>>         => (Deduction([(hyp1_2:that ((A_1 ->
>>              B_1) & (B_1 -> C_1))) => (Deduction([(hyp2_3:
>>                   that A_1) => (((hyp2_3 Mp
>>                   Simplification1(hyp1_2)) Mp
>>                   Simplification2(hyp1_2)):that
>>                   C_1)])
>>              :that (A_1 -> C_1))])
>>         :that (((A_1 -> B_1) & (B_1 -> C_1))
>>         -> (A_1 -> C_1)))]
>>   {move 0}
```

Notice that Lestrade's move structure supports a quite standard style of reasoning under nested hypotheses.

It is also worth noticing the automatic expansion of concepts defined at higher-indexed moves as moves are closed: recall that a defined function (or object) declared at the next move must be eliminated by definitional expansion from sort information posted to the last move by a `declare` or `construct` command. Tracking the appearance of the functions `line1` through `line5` in Lestrade output at different move levels will illustrate this.