Implementing Zermelo's axiomatics and proof of the well-ordering theorem in Lestrade, a dependent types theorem prover

M. Randall Holmes

Boise State Math Dept graduate seminar, 9/27/2019

Abstract

Over a few weeks in the summer, we implemented much of the content of Zermelo's important set theory papers of 1908, including the original axiomatization of Zermelo set theory, the precursor of our current set theory ZFC, and Zermelo's proof of the Well-Ordering Theorem, under our dependent type theory based theorem proving system Lestrade. We will give an overview of this work.

Zermelo's 1908 papers

The papers appearing in 1908 were a second (?) version of Zermelo's proof of the Well-Ordering theorem (if one has the axiom of choice, then every set is well-ordered) and a paper describing the axioms of something close to what we now call "Zermelo set theory", which with the addition of axioms of Replacement and Choice (around 1920) became the currently dominant system of set theory ZFC.

It is a reasonable premise that the exact purpose of Zermelo's axiomatics paper was to provide a firm foundation for his proof of the Well-Ordering Theorem. The paper does, however, include further investigations of the notion of transfinite cardinal number.

Lestrade is a theorem proving system which I have been developing for a number of years, belonging to a class of such systems based on dependent type theory. We will discuss what this means.

The motivations of Lestrade are actually ultimately in philosophy of mathematics, and I will indulge myself by briefly describing these. But it turns out to be recognizably a member of a known family of theorem proving systems, the descendants of the system Automath developed by N de Bruijn in the 1970's.

First philosophical point: the nature of functions

We learn in university math classes at a certain point that a function is "a set of ordered pairs in which no two elements have the same first projection". But we all know that that is not what we learned a function was at first. A function is initially presented to us as a rule or procedure for getting an output given an input. In the most basic case, a function is simply an expression with holes in it into which we are to insert the input to replace an unknown:

$$f(x) = x^2 + x + 1$$

can be taken as an archetypal example.

This bears on whether we think a function is an infinite object (infinity being an important consideration in philosophy of mathematics). The ordered pair definition of a function is appealing if one has no qualms about actual infinities: we simply tabulate the values of the function in every individual case.

A function understood as a machine which reacts to any given input to produce an output is compatible with the notion of a function as a finite object (with a potential infinity of possible outputs determined by a potential infinity of possible inputs). The expression $x^2 + x + 1$ looks finite!

Second philosophical point: proofs as mathematical objects

Lestrade was further intended to implement a certain view of proofs as being themselves mathematical objects. This view is often called the "Curry-Howard isomorphism".

Under this view, each proposition A is associated with a type `that` A (this is Lestrade notation, not standard) inhabited by proofs of or evidence for A.

Logical operations then correspond to operations on types (sometimes very familiar operations on types).

Conjunction ("and")

We have a proof of $A \wedge B$ iff we have a proof of A and a proof of B.

This can be handled by saying that a proof of $A \wedge B$ is a pair (a, b) where a is a proof of A and b is a proof of B, whence the type $\texttt{that}\ A \wedge B$ can simply be identified with $\texttt{that}\, A \times \texttt{that}\, B$.

The logical operation of conjunction corresponds to the type theory or set theory operation of cartesian product.

Implication

A typical way of proving $A \to B$ is to show that if we assume A, B must follow.

That is, if we are presented with a of type $\texttt{that}\ A$, we should have b of type $\texttt{that}\ B$. So we identify proofs of $A \to B$ with functions from $\texttt{that}\ A$ to $\texttt{that}\ B$ *: the type $\texttt{that}\ A \to B$ is implemented as $\texttt{that}\, B^{\texttt{that}\, A}$, and the logical operation of implication corresponds to the type theory or set theory construction of spaces of functions.

For example, for any A, the identity function which takes any a in $\texttt{that}\ A$ to a itself is a proof of $A \to A$.

*This is not exactly what we do in Lestrade, but it is very close.

Universal quantifier

A typical way of proving $(\forall x \in D : P(x))$ is to assume that t is an arbitrary element of D and show how to get a proof of $P(t)$.

This suggests the implementation of a proof of $(\forall x \in D : P(x))$ as a function taking each $d \in D$ to a proof of $P(d)$.

In set theoretical terms, we are identifying

$$\texttt{that}\,(\forall x \in D : P(x))$$

with $\Pi_{d \in D}(\texttt{that}\,P(d))$, the infinite cartesian product of the sets $\texttt{that}\,P(d)$ indexed by the domain D.

In type theory, the implementation would be the same: this would be a theory with dependent types, as the type $\texttt{that}\,P(d)$ of the output for input d depends on the value of d.

We describe how this looks in Lestrade.

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}
```

In a Lestrade environment, the `declare` command introduces parameters of given types to be supplied to postulated operations. `postulate` introduces objects and constructions which we actually postulate.

So the text here declares an operation (intended, it appears, to be conjunction) which takes two propositions as input and outputs a proposition. `prop`, the type of propositions, is a primitive of Lestrade.

```
Lestrade execution:


declare pp that p

>> pp: that p {move 1}



declare qq that q

>> qq: that q {move 1}



declare rr that p & q

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

Here we declare some parameters which will be used as input to constructions postulated on the next slide. Note that for any proposition p, `that` p is a type (intended to be inhabited by evidence that p is true, which may be understood as proofs of p but need not be so understood).

Note that Lestrade does understand use of & as an infix.

Lestrade execution:

```
postulate Conj pp qq that p & q

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

```
postulate Simp1 rr that p

>> Simp1: [(.p_1:prop),(.q_1:prop),(rr_1:that
>>      (.p_1 & .q_1)) => (---:that .p_1)]
>>   {move 0}
```

```
postulate Simp2 rr that q

>> Simp2: [(.p_1:prop),(.q_1:prop),(rr_1:that
>>      (.p_1 & .q_1)) => (---:that .q_1)]
>>   {move 0}
```

On this slide we postulate the rules of inference for the conjunction connective, the familiar rules

$$\frac{\begin{array}{c} A \\ B \end{array}}{A \wedge B} \text{ by conjunction}$$

$$\frac{A \wedge B}{A} \text{ by simplification(1)}$$

$$\frac{A \wedge B}{B} \text{ by simplification(2)}$$

Implicit arguments

If one looks carefully at the declarations of the rules of inference for conjunction, one sees that their explicit arguments are not the only arguments they have: the rule `Conj` of conjunction actually has four parameters, p, q, pp, and qq. But p and q can be left implicit because they can be deduced from the types of the explicitly given parameters.

Implementing this was a fair amount of work but made the system much more practical.

Lestrade execution:

```
postulate -> p q prop

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



declare ss that p -> q

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



postulate Mp pp ss that q

>> Mp: [(.p_1:prop),(pp_1:that .p_1),(.q_1:prop),
>>       (ss_1:that (.p_1 -> .q_1)) => (---:that
>>       .q_1)]
>>   {move 0}
```

Here we declare the implication connective and the rule of modus ponens. The parameters declared earlier are still available to us.

Lestrade execution:

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

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

```
postulate Ded ded that p -> q
```

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

On this slide we declare our primitive technique of proving implications, which is usually called the deduction theorem.

We declare the parameter `ded` as a construction taking a proof of p as input and having output a proof of q. `[pp => that q]` is a notation for a construction type. If we have such a construction, we should believe $p \to q$, and that is what we postulate via the declaration of `Ded`.

Notice that we do not actually identify the proof of $p \to q$ with the construction (function) witnessing it. There are reasons for this.

```
Lestrade execution:


declare D type

>> D: type {move 1}


declare d in D

>> d: in D {move 1}


declare pred [d => prop] \



>> pred: [(d_1:in D) => (---:prop)]
>>   {move 1}
```

We set up for the declaration of the universal quantifier and its rules. We introduce as parameters a type D, an object d of type D, and a predicate `pred` of type D objects (a function taking an element d of type D to a proposition).

```
Lestrade execution:


postulate Forall pred prop

>> Forall: [(.D_1:type),(pred_1:[(d_2:in .D_1)
>>          => (---:prop)])
>>       => (---:prop)]
>>    {move 0}



declare univev that Forall pred

>> univev: that Forall(pred) {move 1}



declare d2 in D

>> d2: in D {move 1}



postulate Ui univev d2 that pred d2

>> Ui: [(.D_1:type),(.pred_1:[(d_2:in .D_1)
>>          => (---:prop)]),
>>       (univev_1:that Forall(.pred_1)),(d2_1:
>>       in .D_1) => (---:that .pred_1(d2_1))]
>>    {move 0}
```

On this slide, we declare the universal quantifier and the rule of universal instantiation.

`Forall pred` would more usually be written $(\forall x \in D : \texttt{pred}(x))$.

`univev` is a parameter, evidence that $(\forall x \in D : \texttt{pred}(x))$.

The rule `Ui` takes as input evidence that $(\forall x \in D : \texttt{pred}(x))$ and a $d \in D$ and outputs evidence that $\texttt{pred}(d)$, which is quite familiar.

```
Lestrade execution:


declare univev2 [d => that pred d] \




>> univev2: [(d_1:in D) => (---:that pred(d_1))]
>>   {move 1}



postulate Ug univev2 that Forall pred

>> Ug: [(.D_1:type),(.pred_1:[(d_2:in .D_1)
>>         => (---:prop)]),
>>      (univev2_1:[(d_3:in .D_1) => (---:that
>>         .pred_1(d_3))])
>>      => (---:that Forall(.pred_1))]
>>   {move 0}
```

On this slide we complete our sample logic declarations by declaring the rule of universal generalization. Given a construction taking each $d \in D$ to evidence for $\texttt{pred}(d)$ (notice the dependent typing), we obtain evidence that $(\forall d \in D : \texttt{pred}(d))$, as we would expect.

Again, the evidence for the universal statement is not identified with the construction (function) as in our abstract description of the Curry-Howard isomorphism. We might suggest our reasons for this briefly later, or might not.

We present a proof in first-order logic as a sample interaction with Lestrade. We prove

$$(\forall x : P(x)) \wedge (\forall x : P(x) \rightarrow Q(x)) \rightarrow (\forall x : Q(x)).$$

```
Lestrade execution:

clearcurrent


declare D type

>> D: type {move 1}



declare d in D

>> d: in D {move 1}



declare P [d => prop] \




>> P: [(d_1:in D) => (---:prop)]
>>   {move 1}



declare Q [d => prop] \




>> Q: [(d_1:in D) => (---:prop)]
>>   {move 1}
```

Here we clear the parameters which have been accumulating (`clearcurrent`) and declare parameters for the intended proof.

```
Lestrade execution:


open

   declare d3 in D

>>    d3: in D {move 2}



   declare initialev that (Forall P) & Forall \
      [d3 =>(P d3) -> (Q d3)] \




>>    initialev: that (Forall(P) & Forall([(d3_1:
>>         in D) => ((P(d3_1) -> Q(d3_1)):prop)]))
>>      {move 2}
```

Here we open a local environment in which we will define a function from which we can make a proof of the conditional theorem that we aim to prove.

`initialev` is of the right type to be the input to such a function. The variable `d3` is not actually a parameter: it is provided for use as a bound variable in the definition of `initialev`.

```
Lestrade execution:


   define line1 initialev: Simp1 initialev


>>    line1: [(initialev_1:that (Forall(P) &
>>         Forall([(d3_2:in D) => ((P(d3_2) ->
>>            Q(d3_2)):prop)]))
>>         ) => (---:that Forall(P))]
>>      {move 1}



   define line2 initialev: Simp2 initialev


>>    line2: [(initialev_1:that (Forall(P) &
>>         Forall([(d3_2:in D) => ((P(d3_2) ->
>>            Q(d3_2)):prop)]))
>>         ) => (---:that Forall([(d3_4:in D)
>>            => ((P(d3_4) -> Q(d3_4)):prop)]))
>>         ]
>>      {move 1}
```

Here we extract the two "conjuncts" of `initialev` (as functions of `initialev`).

```
Lestrade execution:


   open

      declare d2 in D

>>        d2: in D {move 3}
```

We open another local environment, suitable for constructing the function from which we will make the proof of our conclusion ($\forall x \in D : Q(x)$). This will be a function taking $d_2 \in S$ as input and returning a proof of $Q(d_2)$.

A brief summary of what the local environments (introduced with `open` and closed with `close`) do: when we `declare` parameters and then `define` expressions in terms of those parameters in the local environment, Lestrade at the same time defines functions of parameters of those types which can be referenced outside the local environment when it is closed and its local declarations disappear.

```
Lestrade execution:


      define line3 d2: Ui line1 initialev \
         d2

>>       line3: [(d2_1:in D) => (---:that P(d2_1))]
>>         {move 2}



      define line4 d2: Ui line2 initialev \
         d2

>>       line4: [(d2_1:in D) => (---:that (P(d2_1)
>>            -> Q(d2_1)))]
>>         {move 2}
```

`line3` is the proof that $P(d_2)$ by universal instantiation from our assumption that $(\forall x \in D : P(x))$.

`line4` is the proof that $P(d_2) \rightarrow Q(d_2)$, similarly by universal instantiation.

```
Lestrade execution:


      define line5 d2: Mp line3 d2 line4 \
         d2

>>       line5: [(d2_1:in D) => (---:that Q(d2_1))]
>>         {move 2}



      close

   define line6 initialev: Ug line5

>>    line6: [(initialev_1:that (Forall(P) &
>>         Forall([(d3_2:in D) => ((P(d3_2) ->
>>            Q(d3_2)):prop)]))
>>         ) => (---:that Forall(Q))]
>>      {move 1}
```

`line5` draws the conclusion $Q(d_2)$ by modus ponens from `line3` and `line4`.

We close the local environment in which `line5` was defined as an expression in d_2: but in the containing environment, `line5` is still visible as a function of a parameter d_2, and so we can apply universal generalization to this function to get `line6`, a proof that $(\forall x \in D : Q(x))$.

```
Lestrade execution:


   close

define Thetheorem P, Q: Ded line6

>> Thetheorem: [(.D_1:type),(P_1:[(d_2:in .D_1)
>>         => (---:prop)]),
>>      (Q_1:[(d_3:in .D_1) => (---:prop)])
>>      => (Ded([(initialev_5:that (Forall(P_1)
>>         & Forall([(d3_6:in .D_1) => ((P_1(d3_6)
>>            -> Q_1(d3_6)):prop)]))
>>         ) => (Ug([(d2_7:in .D_1) => (((Simp1(initialev_5)
>>            Ui d2_7) Mp (Simp2(initialev_5)
>>            Ui d2_7)):that Q_1(d2_7))])
>>         :that Forall(Q_1))])
>>      :that ((Forall(P_1) & Forall([(d3_11:in
>>         .D_1) => ((P_1(d3_11) -> Q_1(d3_11)):
>>         prop)]))
>>      -> Forall(Q_1)))]
>>   {move 0}
```

We then close the local environment in which `line6` was defined, but `line6` is still visible in the containing environment as a function taking `initialev` of type `that` $(\forall x \in D : P(x)) \wedge (\forall y \in D : P(y) \rightarrow Q(y))$ as input and giving output of type `that` $(\forall z \in D : Q(z))$.

Application of the deduction theorem to this function gives the desired proof of the conditional theorem. Notice that this is actually a function of the parameters P, Q, the predicates involved, and so can be invoked with different predicates as arguments.

For objects defined unconditionally (not in a local environment), Lestrade will display the actual body of a defined function (it could display such terms at all levels, but the display would be much more verbose). It is rather dense, but a term is presented representing the mathematical object we constructed to witness the proof of the theorem.

This ends the first part of the talk, in which I try to give an impression of what the flavor of work under the Lestrade theorem prover is like. There is a lot more to say about this: there is technical description on my web site.

In the second part, I am going to show you highlights from the work I did over the summer. The slides here will provide landmarks for a high level tour of the actual documents I produced in the summer, which I will display.

The documents we will look at (and indeed the very slides you are looking at) have a dual nature. Their LaTeX source is also executable by Lestrade, which identifies labelled verbatim blocks in the LaTeX document as Lestrade codes, runs these blocks and inserts Lestrade's responses to the commands it finds in the file.

This means that Lestrade proof scripts can be very nicely commented: in fact, we can write something like a mathematical paper in parallel with the proof development in Lestrade.

Lestrade feedback is rather verbose, so the documents are large.

The first document, logical preliminaries

I'll skim through this one. It should look familiar after our development of selected logical primitives in the first part. One notable difference is that in our first part we present quantification over a type D (which can be supplied as a parameter). In the Zermelo development all objects are of the fixed type `obj`, and we develop quantifiers over this type (designed to serve as the global type for a theory like ZFC in having only one type of object).

It is worth noting that we started with a development of logical primitives for constructive logic, to which we added the law of excluded middle (which we do use). This means that our set of primitives is not economical as the reductions of basic connectives to a set of one or two which are familiar to us do not work constructively.

As a matter of file development, we then proved those lemmas in logic (and in later portions of the document) which we actually turned out to need in the development of still later parts of the document. A fully developed logical preliminaries document would probably have a larger suite of theorems of propositional and first order logic: we proved the ones we used as needed.

Maybe take a look at logic of equality, pp. 37, 42, 44.

The second document: 1908 Zermelo set theory axiomatics

In this file we developed all the content of the first part of Zermelo's 1908 paper on axiomatics, and we are starting to develop his second part on the theory of cardinality.

We note that we follow the clear sense of the paper in allowing there to be many non-sets with no elements. Zermelo is quite clear that he is allowing this.

We have an extensive anachronistic section in which we introduce the Kuratowski ordered pair and prove its basic properties. We may visit this in the document, time permitting. Students may be well aware that verifying the properties of the Kuratowski pair from first principles is a large task. People at my last talk in the logic seminar will be aware that Zermelo carries out his investigations of cardinality in the second half of this paper and his proof of the well ordering theorem in the other paper without using an ordered pair at all!

I will note just as an aside (it would take too much time to really explain) that it requires care to get the axiom of separation to have the correct strength.

Under Automath*, the parent system of Lestrade, a formalization of Zermelo in the straightforward way essentially automatically gives a formalization of second order Zermelo set theory, in which one can quantify over proper classes. Lestrade's type system is just enough weaker than Automath's type system that the distinction between a scheme of separation in the modern style and a second-order axiom can be drawn.

Zermelo, by all accounts, would have preferred the second-order axiom!

*The same danger might exist with the widely used modern descendant of Automath, Coq, but I am not certain of this.

The document includes the practical application of Russell's paradox, that $\{x \in A : x \notin x\}$ is not an element of A for any set A.

The document uses Zermelo's original formulation of the natural numbers and the axiom of infinity. I might say a brief word about the difference.

The section under development on cardinal equivalence will present interesting difficulties, as Zermelo does not have a definition of the ordered pair as a set. I am only at the beginning of implementing this work.

The proof of the well-ordering theorem

I'll give an overview of this proof.

Zermelo's aim is to prove that every set can be well-ordered (a well-ordering being a linear order of a set under which each nonempty subset of the set has a minimal element).

Zermelo does not have an ordered pair at his disposal. He can however implement a well-ordering as the set of its final segments. This method of implementing relations as sets will work for any linear order and in fact for a wider class of relations.

He states the Axiom of Choice in the form "Every collection of pairwise disjoint sets has a choice set".

However, this is not the form of choice he uses in his argument for the well-ordering theorem. Instead, he postulates a set M along with a method for choosing a distinguished element from each nonempty subset of M. Only after his main proof does he argue (via references to results in the axiomatics paper) that the Axiom as he states it allows him to assume that he can choose elements from nonempty subsets of M in this way. We have fully implemented the main proof (that given a method of choosing an element from each nonempty subset of M we can produce a well-ordering of M); we have not yet completed the proof that AC implies that there is such a method.

For any subset A of M, we define A' as $A \setminus \{a\}$, where a is the distinguished member of A ($\emptyset$ is assigned a distingished member too in our formalism, which is of no interest: $\emptyset' = \emptyset$).

We define a Θ-chain as a set C of subsets of M with the properties that it contains M as an element, contains A' as an element if it contains A, and contains $\bigcap D$ as an element for each nonempty subset D of C.

The intersection of all Θ-chains is the collection of final segments of a well-ordering of M.

Doesn't that seem seductively easy? It requires proof.

One needs to prove that the intersection of all Θ-chains is a Θ-chain.

One then proves that the intersection of all Θ-chains is a linear order. One proves this by showing that for any fixed B in the intersection of all Θ-chains, the collection of D in the intersection of all Θ-chains such that either $B \subseteq D$ or $D \subseteq B$ is itself a Θ-chain and so is in fact the entire intersection of all Θ-chains... Pause to get your mind around this impredicative wonder!

The minimal element of a nonempty subset of M is then the distinguished element of the intersection of all elements of the intersection of all Θ-chains which include the given nonempty set as a subset. Isn't that obvious?

We have some remarks about the experience of proving this.

First, Zermelo's method for choosing a distinguished element of each set, which he postulates at the outset of his main argument, is not presented as a set theoretical object at all (we would in modern terms call this a choice function). Lestrade supports this style: we declare a construction as a parameter of our argument (which we will eventually show that we can implement using AC when we complete this work). Here Lestrade matched Zermelo's style of reasoning well.

Second, the issue of blowup of size of proofs when they are formalized. It is the general rumor that formalized proofs are inconveniently large. Indeed, writing formalized proofs is tedious, resembling computer programming more than argument in the usual sense. However, from the beginnings of the Automath project, workers have observed that the actual blowup factor is much less than folklore suggests: folklore proposes 20; actual attempts to measure it in practice suggest 4 or 5 as the blowup factor (in a suitable metric on *size* ; of course this is not a measure of *effort*.)

There is a lot more experience with this now. When I first started working in this area, if you named a famous theorem it had very likely never been formalized. At this point, if you name a famous theorem, it has probably been verified under some theorem proving system.

But this patticular proof, which takes a few pages in the source, in fact blows up enormously. The problem is the impredicative character of the argument. Zermelo's argument is seductively brief (not really very much longer than my summary above). But every verification that something is a Θ-chain involves verification of three claims, two of which are logically quite complex. When I formalized the proof, it came out as quite large (the hundreds of pages of the document are not a good measure of this, as Lestrade feedback is somewhat verbose, but the blowup factor is still clearly much larger than usual).

I have done other formalizations with Lestrade which appear to bear out the experience of other workers that one is usually looking at a linear blowup factor which is not as large as one expects (though still productive of tedium!) It would be interesting to revisit the argument and see if there is some way I could prove suitable lemmas which would vastly reduce the size of the text. Certainly Zermelo's argument is convincing on careful reading and not all that long...but it is also quite clear where a claim made in a sentence or two can conceal an enormous amount of checking.

Future activities

Other formalizations might be interesting. For example, I am tempted to see what a formalization of the content I present from Spivak's calculus book in Math 314 would look like in Lestrade.

I am interested in applications of formalized proof systems in teaching logic. Lestrade is actually fairly well adapted to presenting the style of formal reasoning I teach in classes, but I have never tried to present a logic lab using this software. The Automath group did teach logic using their software, which is quite similar in its underlying philosophy and probably less user-friendly.

Lestrade looks like the type declaration system of a functional programming language. I am contemplating what it would look like if it actually *were* a functional programming language, with the additional feature that one could present formal proofs (presumably of correctness of pieces of one's programs) in the same environment. There are current language projects along these lines, similar in some ways and different in others.