ZFC in Lestrade

Randall Holmes

May 14, 2018

A quick and dirty declaration of all of ZFC under Lestrade. We begin with logical primitives.

Of course the file isn't that interesting unless one proves something with it.

```
Lestrade execution:

construct ?? prop

>> ??: prop {move 0}


declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


construct -> p q prop

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


declare x obj

>> x: obj {move 1}


declare pred [x=>prop]

>> pred: [(x_1:obj) => (---:prop)]
```

```
>>   {move 1}


construct Forall pred prop

>> Forall: [(pred_1:[(x_2:obj) => (---:prop)])
>>        => (---:prop)]
>>   {move 0}


declare pq that p -> q

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


declare pp that p

>> pp: that p {move 1}


construct Mp pq pp that q

>> Mp: [(.p_1:prop),(.q_1:prop),(pq_1:that (.p_1
>>        -> .q_1)),(pp_1:that .p_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}


declare univev that Forall pred

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

```
declare x2 obj

>> x2: obj {move 1}


construct Ui univev x2 that pred x2

>> Ui: [(.pred_1:[(x_2:obj) => (---:prop)]),
>>        (univev_1:that Forall(.pred_1)),(x2_1:
>>        obj) => (---:that .pred_1(x2_1))]
>>   {move 0}


declare univev2 [x=>that pred x]

>> univev2: [(x_1:obj) => (---:that pred(x_1))]
>>   {move 1}


construct Ug univev2 that Forall pred

>> Ug: [(.pred_1:[(x_2:obj) => (---:prop)]),
>>        (univev2_1:[(x_3:obj) => (---:that .pred_1(x_3))])
>>        => (---:that Forall(.pred_1))]
>>   {move 0}


declare maybe that (p-> ??)-> ??

>> maybe: that ((p -> ??) -> ??) {move 1}


construct Dneg maybe that p

>> Dneg: [(.p_1:prop),(maybe_1:that ((.p_1 ->
>>        ??) -> ??)) => (---:that .p_1)]
>>   {move 0}


define ~ p : p -> ??

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

```
define & p q : ~(p -> ~q)

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


define V p q : ~p -> q

>> V: [(p_1:prop),(q_1:prop) => ((~(p_1) ->
>>        q_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}


define Exists pred : ~ Forall [x => ~pred x]

>> Exists: [(pred_1:[(x_2:obj) => (---:prop)])
>>        => (~(Forall([(x_3:obj) => (~(pred_1(x_3)):
>>             prop)])))
>>        :prop)]
>>   {move 0}
```

The above logical primitives are sufficient, since our logic is classical. Of course derived rules for the defined logical operations are needed.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}
```

```
construct = x y prop

>> =: [(x_1:obj),(y_1:obj) => (---:prop)]
>>   {move 0}


construct E x y prop

>> E: [(x_1:obj),(y_1:obj) => (---:prop)]
>>   {move 0}


construct Refl x that x = x

>> Refl: [(x_1:obj) => (---:that (x_1 = x_1))]
>>   {move 0}


declare pred [x => prop]

>> pred: [(x_1:obj) => (---:prop)]
>>   {move 1}


declare eqev that x=y

>> eqev: that (x = y) {move 1}


declare predev that pred x

>> predev: that pred(x) {move 1}


construct Substitution pred, eqev predev that pred y

>> Substitution: [(pred_1:[(x_2:obj) => (---:
>>             prop)]),
>>        (.x_1:obj),(.y_1:obj),(eqev_1:that (.x_1
>>        = .y_1)),(predev_1:that pred_1(.x_1))
>>        => (---:that pred_1(.y_1))]
>>   {move 0}
```

```
construct Empty obj

>> Empty: obj {move 0}


construct notinempty x that ~(x E Empty)

>> notinempty: [(x_1:obj) => (---:that ~((x_1
>>        E Empty)))]
>>   {move 0}


construct the pred obj

>> the: [(pred_1:[(x_2:obj) => (---:prop)])
>>        => (---:obj)]
>>   {move 0}


declare unique that Exists [y=>Forall[x=> (pred x) <-> x=y]]

>> unique: that Exists([(y_1:obj) => (Forall([(x_2:
>>             obj) => ((pred(x_2) <-> (x_2 =
>>             y_1)):prop)])
>>        :prop)])
>>   {move 1}


construct The1 pred, unique that pred the pred

>> The1: [(pred_1:[(x_2:obj) => (---:prop)]),
>>        (unique_1:that Exists([(y_3:obj) =>
>>             (Forall([(x_4:obj) => ((pred_1(x_4)
>>                  <-> (x_4 = y_3)):prop)])
>>             :prop)]))
>>        => (---:that pred_1(the(pred_1)))]
>>   {move 0}


declare notunique that ~Exists [y=>Forall[x=>(pred x) <-> x=y]]

>> notunique: that ~(Exists([(y_1:obj) => (Forall([(x_2:
>>             obj) => ((pred(x_2) <-> (x_2 =
>>             y_1)):prop)])
>>        :prop)]))
>>   {move 1}
```

```
construct The2 pred, notunique that (the pred) = Empty

>> The2: [(pred_1:[(x_2:obj) => (---:prop)]),
>>        (notunique_1:that ~(Exists([(y_3:obj)
>>             => (Forall([(x_4:obj) => ((pred_1(x_4)
>>                  <-> (x_4 = y_3)):prop)])
>>             :prop)]))
>>        ) => (---:that (the(pred_1) = Empty))]
>>   {move 0}


open

     declare z obj

>>      z: obj {move 2}


     declare in1 that z E x

>>      in1: that (z E x) {move 2}


     construct ext1 in1 that z E y

>>      ext1: [(.z_1:obj),(in1_1:that (.z_1
>>             E x)) => (---:that (.z_1 E y))]
>>        {move 1}


     declare in2 that z E y

>>      in2: that (z E y) {move 2}


     construct ext2 in2 that z E x

>>      ext2: [(.z_1:obj),(in2_1:that (.z_1
>>             E y)) => (---:that (.z_1 E x))]
>>        {move 1}


     close
```

```
construct Extensionality ext1, ext2 that x=y

>> Extensionality: [(.x_1:obj),(.y_1:obj),(ext1_1:
>>        [(.z_2:obj),(in1_2:that (.z_2 E .x_1))
>>             => (---:that (.z_2 E .y_1))]),
>>        (ext2_1:[(.z_3:obj),(in2_3:that (.z_3
>>             E .y_1)) => (---:that (.z_3 E .x_1))])
>>        => (---:that (.x_1 = .y_1))]
>>   {move 0}


construct Setof x pred obj

>> Setof: [(x_1:obj),(pred_1:[(x_2:obj) => (---:
>>             prop)])
>>        => (---:obj)]
>>   {move 0}


declare inev that (y E x) & pred y

>> inev: that ((y E x) & pred(y)) {move 1}


construct Comp1 inev that y E Setof x pred

>> Comp1: [(.y_1:obj),(.x_1:obj),(.pred_1:[(x_2:
>>             obj) => (---:prop)]),
>>        (inev_1:that ((.y_1 E .x_1) & .pred_1(.y_1)))
>>        => (---:that (.y_1 E (.x_1 Setof .pred_1)))]
>>   {move 0}


declare inev2 that y E Setof x pred

>> inev2: that (y E (x Setof pred)) {move 1}


construct Comp2 inev2 that y E x

>> Comp2: [(.y_1:obj),(.x_1:obj),(.pred_1:[(x_2:
>>             obj) => (---:prop)]),
>>        (inev2_1:that (.y_1 E (.x_1 Setof .pred_1)))
>>        => (---:that (.y_1 E .x_1))]
>>   {move 0}
```

```
construct Comp3 inev2 that pred y

>> Comp3: [(.y_1:obj),(.x_1:obj),(.pred_1:[(x_2:
>>             obj) => (---:prop)]),
>>        (inev2_1:that (.y_1 E (.x_1 Setof .pred_1)))
>>        => (---:that .pred_1(.y_1))]
>>   {move 0}
```

Basic primitives, equality, membership, set builder notation defined, along with their basic rules of inference.

The definite description operator appears here because it is a logical operation and belongs right after equality. Its use is that it allows us to state the Axiom of Replacement in a much simpler way below, using (Lestrade) functions; a functional binary relation can readily be converted to a function using definite description. In an unexpected interlock, I had to declare the empty set since I use it as the default object.

```
Lestrade execution:

declare z obj

>> z: obj {move 1}


construct pair x y obj

>> pair: [(x_1:obj),(y_1:obj) => (---:obj)]
>>   {move 0}


construct pair1 x y that x E pair x y

>> pair1: [(x_1:obj),(y_1:obj) => (---:that
>>        (x_1 E (x_1 pair y_1)))]
>>   {move 0}


construct pair2 x y that y E pair x y

>> pair2: [(x_1:obj),(y_1:obj) => (---:that
>>        (y_1 E (x_1 pair y_1)))]
>>   {move 0}
```

```
construct pair3 x y z that ((z E pair x y) -> ((z = x) V z = y))

>> pair3: [(x_1:obj),(y_1:obj),(z_1:obj) =>
>>        (---:that ((z_1 E (x_1 pair y_1)) ->
>>        ((z_1 = x_1) V (z_1 = y_1))))]
>>   {move 0}


construct Pow x obj

>> Pow: [(x_1:obj) => (---:obj)]
>>   {move 0}


declare inev3 that z E y

>> inev3: that (z E y) {move 1}


declare inev4 [z,inev3 => that z E x]

>> inev4: [(z_1:obj),(inev3_1:that (z_1 E y))
>>        => (---:that (z_1 E x))]
>>   {move 1}


construct Pow1 inev4 that y E Pow x

>> Pow1: [(.y_1:obj),(.x_1:obj),(inev4_1:[(z_2:
>>             obj),(inev3_2:that (z_2 E .y_1))
>>             => (---:that (z_2 E .x_1))])
>>        => (---:that (.y_1 E Pow(.x_1)))]
>>   {move 0}


declare inev5 that y E Pow x

>> inev5: that (y E Pow(x)) {move 1}


declare inev6 that z E y

>> inev6: that (z E y) {move 1}
```

```
construct Pow2 inev5 inev6 that z E x

>> Pow2: [(.y_1:obj),(.x_1:obj),(inev5_1:that
>>        (.y_1 E Pow(.x_1))),(.z_1:obj),(inev6_1:
>>        that (.z_1 E .y_1)) => (---:that (.z_1
>>        E .x_1))]
>>   {move 0}


construct Union x obj

>> Union: [(x_1:obj) => (---:obj)]
>>   {move 0}


declare inev7 that z E y

>> inev7: that (z E y) {move 1}


declare inev8 that y E x

>> inev8: that (y E x) {move 1}


construct Union1 inev7 inev8  that z E Union x

>> Union1: [(.z_1:obj),(.y_1:obj),(inev7_1:that
>>        (.z_1 E .y_1)),(.x_1:obj),(inev8_1:that
>>        (.y_1 E .x_1)) => (---:that (.z_1 E
>>        Union(.x_1)))]
>>   {move 0}


declare inev9 that z E Union x

>> inev9: that (z E Union(x)) {move 1}


construct Union2 inev9 that Exists[y => (z E y) & (y E x)]

>> Union2: [(.z_1:obj),(.x_1:obj),(inev9_1:that
>>        (.z_1 E Union(.x_1))) => (---:that Exists([(y_2:
>>             obj) => (((.z_1 E y_2) & (y_2 E
>>             .x_1)):prop)]))
>>        ]
```

```
>>   {move 0}


construct N obj

>> N: obj {move 0}


construct N0 that Empty E N

>> N0: that (Empty E N) {move 0}


declare inev10 that x E N

>> inev10: that (x E N) {move 1}


construct N1 inev10 that (pair x x) E N

>> N1: [(.x_1:obj),(inev10_1:that (.x_1 E N))
>>        => (---:that ((.x_1 pair .x_1) E N))]
>>   {move 0}


open

     declare x1 obj

>>      x1: obj {move 2}


     construct  I  x1 prop

>>      I: [(x1_1:obj) => (---:prop)]
>>        {move 1}


     construct  I1 that I Empty

>>      I1: that I(Empty) {move 1}


     declare i1 that I x1

>>      i1: that I(x1) {move 2}
```

```
     construct I2 x1 i1 that I pair x1 x1

>>      I2: [(x1_1:obj),(i1_1:that I(x1_1))
>>             => (---:that I((x1_1 pair x1_1)))]
>>        {move 1}


     close

declare inev11 that x E N

>> inev11: that (x E N) {move 1}


construct N3 I, I1,I2, inev11 that I x

>> N3: [(I_1:[(x1_2:obj) => (---:prop)]),
>>        (I1_1:that I_1(Empty)),(I2_1:[(x1_3:
>>             obj),(i1_3:that I_1(x1_3)) => (---:
>>             that I_1((x1_3 pair x1_3)))]),
>>        (.x_1:obj),(inev11_1:that (.x_1 E N))
>>        => (---:that I_1(.x_1))]
>>   {move 0}
```

Here are all the specific provisions for sets in Zermelo set theory. Unordered pairs, power sets, and unions are provided. I used the Zermelo implementation of $\mathbb{N}$.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare P obj

>> P: obj {move 1}
```

```
declare p obj

>> p: obj {move 1}


declare inev12 that p E P

>> inev12: that (p E P) {move 1}


declare inhabited [p,inev12 => that Exists[x=>x E p]]

>> inhabited: [(p_1:obj),(inev12_1:that (p_1
>>        E P)) => (---:that Exists([(x_2:obj)
>>             => ((x_2 E p_1):prop)]))
>>        ]
>>   {move 1}


declare q obj

>> q: obj {move 1}


declare inev13 that q E P

>> inev13: that (q E P) {move 1}


declare inev14 that x E p

>> inev14: that (x E p) {move 1}


declare inev15 that x E q

>> inev15: that (x E q) {move 1}


declare disjoint [p,q,x,inev12,inev13,inev14,inev15 => that p=q]

>> disjoint: [(p_1:obj),(q_1:obj),(x_1:obj),
>>        (inev12_1:that (p_1 E P)),(inev13_1:
>>        that (q_1 E P)),(inev14_1:that (x_1
>>        E p_1)),(inev15_1:that (x_1 E q_1))
>>        => (---:that (p_1 = q_1))]
```

```
>>   {move 1}


declare C obj

>> C: obj {move 1}


declare y obj

>> y: obj {move 1}


declare z obj

>> z: obj {move 1}


define Oneintersect p C:  Exists[y=>((y E p) & y E C)\
    & Forall[z=>((z E p) & (z E C))->z=y]]

>> Oneintersect: [(p_1:obj),(C_1:obj) => (Exists([(y_2:
>>             obj) => ((((y_2 E p_1) & (y_2 E
>>             C_1)) & Forall([(z_3:obj) => ((((z_3
>>                  E p_1) & (z_3 E C_1)) -> (z_3
>>                  = y_2)):prop)]))
>>             :prop)])
>>        :prop)]
>>   {move 0}


construct Choice P,inhabited,disjoint \
     that Exists[C => Forall[p=>(p E P) -> Oneintersect p C]]

>> Choice: [(P_1:obj),(inhabited_1:[(p_2:obj),
>>             (inev12_2:that (p_2 E P_1)) =>
>>             (---:that Exists([(x_3:obj) =>
>>                  ((x_3 E p_2):prop)]))
>>             ]),
>>        (disjoint_1:[(p_4:obj),(q_4:obj),(x_4:
>>             obj),(inev12_4:that (p_4 E P_1)),
>>             (inev13_4:that (q_4 E P_1)),(inev14_4:
>>             that (x_4 E p_4)),(inev15_4:that
>>             (x_4 E q_4)) => (---:that (p_4
>>             = q_4))])
>>        => (---:that Exists([(C_5:obj) => (Forall([(p_6:
```

```
>>                  obj) => (((p_6 E P_1) -> (p_6
>>                  Oneintersect C_5)):prop)])
>>             :prop)]))
>>        ]
>>   {move 0}


clearcurrent

declare P obj

>> P: obj {move 1}


declare p obj

>> p: obj {move 1}


declare inev12 that p E P

>> inev12: that (p E P) {move 1}


declare inp that ~(p = Empty)

>> inp: that ~((p = Empty)) {move 1}


construct Chooselocal P inev12 inp obj

>> Chooselocal: [(P_1:obj),(.p_1:obj),(inev12_1:
>>        that (.p_1 E P_1)),(inp_1:that ~((.p_1
>>        = Empty))) => (---:obj)]
>>   {move 0}


construct Choselocally P inev12 inp that (Chooselocal P inev12 inp) E p

>> Choselocally: [(P_1:obj),(.p_1:obj),(inev12_1:
>>        that (.p_1 E P_1)),(inp_1:that ~((.p_1
>>        = Empty))) => (---:that (Chooselocal(P_1,
>>        inev12_1,inp_1) E .p_1))]
>>   {move 0}
```

The Axiom of Choice. We give a lengthy implementation along standard lines. The last few lines give a primitive construction which implements local choice from nonempty elements p of a set P, without attention to whether its first argument P has disjoint elements. If P is a partition, this function combined with Separation will build a choice set for P, and we believe this construction gives no additional global information. As in the case of the Axiom of Separation below, the fact that *function* is the primitive notion of Lestrade makes life simpler in the second approach. We do note that in spite of our efforts at localization the second approach allows definition of a global choice function: to choose an element from any set A in a uniform way, apply the uniform choice function to the first level of the cumulative hierarchy which has A as an element as P and A as p. However, global choice is a conservative extension of ZFC: our stronger formulation does not allow any new theorems of set theory to be proved. It would not be equivalent if we passed to a theory with classes.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj

>> y: obj {move 1}


declare f [x=> obj]

>> f: [(x_1:obj) => (---:obj)]
>>   {move 1}


declare A obj

>> A: obj {move 1}


construct Image f, A obj

>> Image: [(f_1:[(x_2:obj) => (---:obj)]),
>>         (A_1:obj) => (---:obj)]
>>   {move 0}
```

```
construct Image1 f, A \
     that Forall[x=>(x E Image f, A) <-> Exists[y=>(y E A) &(f y) = x]]

>> Image1: [(f_1:[(x_2:obj) => (---:obj)]),
>>         (A_1:obj) => (---:that Forall([(x_3:
>>              obj) => (((x_3 E Image(f_1,A_1))
>>              <-> Exists([(y_4:obj) => (((y_4
>>                   E A_1) & (f_1(y_4) = x_3)):
>>                   prop)]))
>>              :prop)]))
>>         ]
>>    {move 0}
```

This provides the Axiom of Replacement. Since we formulate it in terms of functions, we needed to provide the definite description operator above to allow demonstration of the usual form using functional binary relations.

A good question which someone asked is, how do we implement schemes? This looks like a single axiom, but it actually is equivalent to a scheme in an ordinary treatment. The reason is that we don't have facilities to quantify over the sort of functions from objects to objects which participates in the axiom `Image` which implements Replacement. We are asserting this for each function f from objects to objects, but not universally quantifying it. We *could* add quantifiers over this type to our language, in which case we would in effect be doing Morse-Kelley set theory. But we have to do it explicitly. This is an issue with the old system Automath, which has a confusion of functions , objects, and sorts in a crucial way which is in many ways very convenient but which causes it to be automatically possible to define quantification on any sort whatsoever. I think it is an interesting question whether a reasonably fluent implementation of ZFC in Automath which is not also an implementation of Morse-Kelley is even possible.

Similar remarks apply to the use of general functions from objects to propositions in the implementation of the Separation Axiom.

```
Lestrade execution:

clearcurrent

declare x obj

>> x: obj {move 1}


declare y obj
```

```
>> y: obj {move 1}


declare z obj

>> z: obj {move 1}


construct Foundation x \
that Exists[y => (y E x) & Forall[z=> (z E y) -> ~(z E x)]]

>> Foundation: [(x_1:obj) => (---:that Exists([(y_2:
>>             obj) => (((y_2 E x_1) & Forall([(z_3:
>>                  obj) => (((z_3 E y_2) -> ~((z_3
>>                  E x_1))):prop)]))
>>             :prop)]))
>>        ]
>>   {move 0}
```

This provides the axiom of foundation.

I'm considering continuing this file with a proof of the Well-Ordering Theorem. This would be in a style not using pairing, just for fun.