Implementation of Zermelo's work of 1908 in Lestrade: Part I, Logic

M. Randall Holmes

September 6, 2019

1 Introduction

This document was originally titled as an essay on the proposition that mathematics is what can be done in Automath (as opposed to what can be done in ZFC, for example). Such an essay is still in in my mind, but this particular document has transformed itself into the large project of implementing Zermelo's two important set theory papers of 1908 in Lestrade, with the further purpose of exploring the actual capabilities of Zermelo's system of 1908 as a mathematical foundation, which we think are perhaps underrated.

This is a new version of this document in modules, designed to make it possible to work more efficiently without repeated execution of slow log files when they do not need to be revisited.

2 Introducing Lestrade

Text in Lestrade is a series of declarations and definitions of entities of various sorts, which fall into two large categories, *objects* and *constructions*.

There are five kinds of sorts of object: there is a sort `prop` of propositions, there is a sort `type` of type labels and there is a sort `obj` of "untyped mathematical objects" (which we will use as the sort of the sets in our implementation of Zermelo set theory and ZFC). Then there are two systems of parameterized sorts which can be expanded as the Lestrade text develops: for each proposition p there is a sort `that` p, inhabited by evidence for p (a proof of p is evidence, but we prefer the less definite term for the sort); for each type label τ there is a sort `in` τ (which we may call the type τ). The types are intended to be inhabited by typed mathematical objects.

Constructions are "functions" in the most general sense. A construction sort is of the form $[x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \Rightarrow \tau]$, where the τ_i of each input parameter x_i is an object or construction sort, τ is an object sort, and the output sort τ may depend on the values of any or all of the x_i's, while each τ_j may depend on any or all x_i's for $i < j$. In other words, constructions are dependently typed. We will see this at work in examples. Constructions are not exactly first-class objects in a Lestrade theory: we will see in our Zermelo development that (for example) a constructon taking sort `obj` to sort `prop` is a proper class rather than a set. A rule of inference in logic as implemented in Lestrade is a construction, while an implication is an actual proposition, thus an object. In general, we can think of constructions as proper class functions; we reserve the term "function" for objects implementing constructions in some locally appropriate sense.

We have just described basically the entire framework. All further concepts are introduced by specific declarations and definitions in the Lestrade environment.

This is less powerful than Automath, it should be observed. No construction produces a construction as output (though a construction can take a construction as input and produce an object as output and this feature can be used to encapsulate constructions in actual objects of appropriate types, which we would be more likely to call "functions". In Automath, functions are first class objects and function types are first class types, and we will see that this gives Automath strength which would make it difficult to replicate our implementation of (first-order) Zermelo set theory in Automath.

Nonetheless, philosophically it *is* Automath: it is a system of dependent type theory intended to implement mathematical objects and proofs about mathematical objects in parallel ways, the parallelism mediated by the Curry-Howard isomorphism. It might be characterized as more cautious Automath.

3 Developing logic

We will develop classical logic, but in a way which can readily be cut down to constructive logic.

```
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}



postulate -> p q prop

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



postulate V p q prop

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



postulate ?? prop

>> ??: prop {move 0}
```

Above are the declarations for basic operations of propositional logic. We introduce propositional variables usable as parameters in our declarations using the `declare` command, then introduce primitives of conjunction, implicatation, disjunction, and the False using the `postulate` command. That **??** is a constant rather than a variable parameter can be seen in its designation at `move 0` rather than `move 1`, the designation attached to the parameters `p` and `q`.

```
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}




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}
```

Above we declare the rules of deduction for conjunction as constructions. Parameters `pp`, `qq`, and `rr` representing evidence for p, q, and $p \wedge q$ respectively, are introduced. We then declare primitive functions which take proofs of p and q to a proof of $p \wedge q$, and which extract proofs of p and q from a proof of $p \wedge q$.

Notice that the dependently typed constructions `Conj`, `Simp1`, and `Simp2` also have p and q as implicit arguments (the values of the implicit arguments can be deduced from the types of the explicit arguments; this feature took work to install!)

```
Lestrade execution:

clearcurrent
```

```
declare p prop

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

```
declare q prop

>> q: prop {move 1}



declare rr that p & q

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



define Conjsymm rr : Conj(Simp2 rr,Simp1 \
   rr)

>> Conjsymm: [(.p_1:prop),(.q_1:prop),(rr_1:
>>      that (.p_1 & .q_1)) => ((Simp2(rr_1) Conj
>>      Simp1(rr_1)):that (.q_1 & .p_1))]
>>   {move 0}



open

   declare r prop

>>    r: prop {move 2}



   declare s prop

>>    s: prop {move 2}
```

```
   declare t prop

>>    t: prop {move 2}



   declare ss that r & s & t

>>    ss: that (r & (s & t)) {move 2}



   define conjtest ss : Conjsymm ss

>>    conjtest: [(.r_1:prop),(.s_1:prop),(.t_1:
>>         prop),(ss_1:that (.r_1 & (.s_1 & .t_1)))
>>         => (---:that ((.s_1 & .t_1) & .r_1))]
>>      {move 1}



   close
```

In this block of text we verify a rule of inference (from $p \wedge q$, infer $q \wedge p$) and present an example of its use.

The `clearcurrent` command clears all the variables currently declared, so p, q and `rr` have to be declared again.

The proof of the rule of inference is simply presented as a calculation: we will see examples below where proof development looks more the way we expect a proof to look.

The example is encapsulated in a local environment (indicated by the `open` and `close` commands). Notice that the parameters introduced locally to that environment are labelled `move 2`, and the proof `conjtest` constructed in the local environment is labelled `move 1`: it will be cleared at the next execution of the `clearcurrent` command, while `Conjsymm`, developed at move 0, is a permanently recorded theorem.

The reader who carefully examines the terms may notice that the type information reported by Lestrade for declarations includes the body of definitions only in the case of objects and constructions defined at move 0. It is possible to set Lestrade to show all definition bodies, but this makes the Lestrade output larger.

```
Lestrade execution:

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 rr that p -> q

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



postulate Mp pp rr that q

>> Mp: [(.p_1:prop),(pp_1:that .p_1),(.q_1:prop),
>>      (rr_1:that (.p_1 -> .q_1)) => (---:that
>>      .q_1)]
>>   {move 0}



open

   declare pp1 that p

>>    pp1: that p {move 2}



   postulate ded pp1 that q

>>    ded: [(pp1_1:that p) => (---:that q)]
>>      {move 1}



   close

postulate Ded ded that p -> q

>> Ded: [(.p_1:prop),(.q_1:prop),(ded_1:[(pp1_2:
>>         that .p_1) => (---:that .q_1)])
>>      => (---:that (.p_1 -> .q_1))]
>>   {move 0}
```

In this block we present the primitive rules for implication. The rule of modus ponens requires no special comment: it is developed in much the same way as the rules for conjunction above.

The rule `Ded` which implements the deduction theorem is a more subtle object, and the way it is developed involves subtle use of the Lestrade environment system.

What `Ded` does is readily described: it takes implicit arguments p and q (deducible from the type of the explicit argument) and a construction of evidence for q from evidence for p, and returns a proof of $p \rightarrow q$.

It makes an important point about the difference between Lestrade and Automath (and other Automath relatives) that a construction of type $[p : \texttt{prop} \Rightarrow \texttt{that}\, q]$ is not itself a proof of (or even evidence for) $p \rightarrow q$: such a construction is actually in itself a rule of inference rather than an implication. That we can infer q from p (a rule of inference, and a construction) is cast to a proof of $p \rightarrow q$ (an entity) by the constructor `Ded`.

To explain how it is done, we need to outline the story of Lestrade environments. At any given point, we are committed to objects in a move i (and all moves j for $j < i$) and may declare variable parameters in move $i + 1$. The system variable i is usually 0: the `open` command has the effect of creating a new empty move $i+2$ and then incrementing i; the `close` command has the effect of erasing all declarations and definitions in move $i + 1$[1] and decrementing i. i cannot be decremented below 0. The `clearcurrent` command has the effect of clearing move $i + 1$ without decrementing i: this can be used to clear move 1, which would otherwise get cluttered. Anything declared in move 0 is permanent.

The `declare` command is used to introduce new objects in move $i + 1$, which serve as parameters in declarations and definitions.

The `postulate` command, the mechanism for introduction of primitive notions and axioms, declares an identifier as a constant in move i of a given type, either an object if it is presented with no arguments, or a construction if it is presented with a list of arguments from move $i+1$ appearing in the order in which they are declared (the order condition is a cheap way to ensure that type dependencies are coherent). As we have seen above, some arguments may be left implicit, if they can be deduced from the types of the explicitly given arguments.

The `define` command defines an identifier as equivalent to an object term in move i (if it has no arguments) or as a construction in move i if it is presented with move $i + 1$ arguments in the correct order. Of course this only works if the term types correctly. This command is the vehicle for introduction of defined notions and theorems.

Declaration information recorded in type i will expand any definitions in type $i + 1$ on which the declaration or definition depended, because such information may disappear when move $i + 1$ is cleared.

[1] We will see later that it is possible to save declarations in move $i + 1$ so that one can return to (that version of) move $i + 1$ if desired.

Notice that constructions are declared in prefix form, but once declared can be used in infix form. Lestrade treats most arity 2 operators as infix in output; it understands mixfix notation $x_1 f x_2 x_n \ldots$ for a construction f with arity greater than 2, but will never present output in this way.

This is only a brief account, enough to support the origin story of `Ded`. We execute `open`, so we are in move 1 instead of move 0. We introduce a move 2 parameter `pp1` representing evidence for p. We declare a construction `ded` which takes `pp1` as input and outputs evidence for q: this is a move 1 object. We then execute `close`, so that `ded` appears at move 1 while we are in move 0. Since `ded` is not a defined object, it is viewed as a variable parameter of the appropriate type, which allows us to use the `postulate` command to declare `Ded` with the expected behavior. It is a subtle point that primitive notions and axioms declared in move $i + 1$ using `postulate` become variable parameters when we transition to move i.

```
Lestrade execution:



open

   declare pp2 that p

>>    pp2: that p {move 2}




   declare ded2 [pp2 => that q] \





>>    ded2: [(pp2_1:that p) => (---:that q)]
>>      {move 2}




   postulate Ded2 ded2 that p-> q
```

```
>>    Ded2: [(ded2_1:[(pp2_2:that p) => (---:
>>            that q)])
>>         => (---:that (p -> q))]
>>      {move 1}
```

```
   close
```

Here we illustrate how the deduction theorem construction could be declared without a local environment (though we package the whole experiment in a local environment so that the redundant construction we declare can be discarded). The key move here is that ded2 is declared directly with a construction type. Notice that Ded2 has the same dependent type as Ded. If the open and close were removed, this block would declare the deduction theorem as a primitive construction in move 0, functionally equivalent to Ded already declared above.

```
Lestrade execution:

clearcurrent


declare p prop

>> p: prop {move 1}



declare q prop

>> q: prop {move 1}



open

   declare rr that p & q
```

```
>>    rr: that (p & q) {move 2}



   define line1 rr: Conjsymm rr

>>    line1: [(rr_1:that (p & q)) => (---:that
>>         (q & p))]
>>      {move 1}



   close

define Consymmimp p q: Ded line1

>> Consymmimp: [(p_1:prop),(q_1:prop) => (Ded([(rr_2:
>>         that (p_1 & q_1)) => (Conjsymm(rr_2):
>>         that (q_1 & p_1))])
>>      :that ((p_1 & q_1) -> (q_1 & p_1)))]
>>   {move 0}
```

Here is an example of use of `Ded`. Do notice that `line1` and `Conjsymm` have different types: the implicit type mechanism obscures the true type of `Conjsymm`, which is not suitable for use as an input to `Ded`.

```
Lestrade execution:

clearcurrent


declare p prop

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

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

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



declare absurd that ??

>> absurd: that ?? {move 1}



postulate Giveup p absurd that p

>> Giveup: [(p_1:prop),(absurd_1:that ??) =>
>>      (---:that p_1)]
>>   {move 0}



declare maybe that ~ ~ p

>> maybe: that ~(~(p)) {move 1}



postulate Dneg maybe that p

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



declare q prop
```

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



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

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

Here we define negation $\neg p$ as $p \rightarrow \bot$ ($\bot$ being the absurd), introduce the rule that anything follows from the absurd, and introduce the primitive rule of double negation which makes the logic classical. All of our other primitive constructions of logic are constructive.

We incidentally define the biconditional.

```
Lestrade execution:

clearcurrent


declare p prop

>> p: prop {move 1}



declare q prop

>> q: prop {move 1}



declare r prop
```

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



declare pp that p

>> pp: that p {move 1}



declare qq that q

>> qq: that q {move 1}



declare rr that p V q

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



postulate Add1 q pp that p V q

>> Add1: [(q_1:prop),(.p_1:prop),(pp_1:that
>>      .p_1) => (---:that (.p_1 V q_1))]
>>   {move 0}



postulate Add2 p qq that p V q

>> Add2: [(p_1:prop),(.q_1:prop),(qq_1:that
>>      .q_1) => (---:that (p_1 V .q_1))]
>>   {move 0}
```

```
declare case1 [pp => that r] \
```

```
>> case1: [(pp_1:that p) => (---:that r)]
>>   {move 1}
```

```
declare case2 [qq => that r] \
```

```
>> case2: [(qq_1:that q) => (---:that r)]
>>   {move 1}
```

```
postulate Cases rr case1, case2 that r

>> Cases: [(.p_1:prop),(.q_1:prop),(rr_1:that
>>      (.p_1 V .q_1)),(.r_1:prop),(case1_1:[(pp_2:
>>         that .p_1) => (---:that .r_1)]),
>>      (case2_1:[(qq_3:that .q_1) => (---:that
>>         .r_1)])
>>      => (---:that .r_1)]
>>   {move 0}
```

Here we develop the primitive rules of disjunction, the two rules of addition and the rule of proof by cases.

Here we use the approach of declaring the two constructions which implement the cases directly as construction variables, rather than the indirect though philosophically interesting approach using local environments exhibited in the development of the deduction theorem.

4 Propositional logic lemmas

This space is reserved for proofs of lemmas in propositional logic as we require them for proofs below.

```
Lestrade execution:

clearcurrent


declare p prop

>> p: prop {move 1}



declare pp that p

>> pp: that p {move 1}



define Fixform p pp : pp

>> Fixform: [(p_1:prop),(pp_1:that p_1) => (pp_1:
>>      that p_1)]
>>   {move 0}
```

Fixform is a device for controlling the way that Lestrade expresses defined concepts in reported type information; we will see its use on many occasions below.

```
Lestrade execution:

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 qq that q

>> qq: that q {move 1}



declare iffev that p==q

>> iffev: that (p == q) {move 1}



define Iff1 pp iffev : Mp pp Simp1 iffev


>> Iff1: [(.p_1:prop),(pp_1:that .p_1),(.q_1:
>>      prop),(iffev_1:that (.p_1 == .q_1)) =>
>>      ((pp_1 Mp Simp1(iffev_1)):that .q_1)]
>>   {move 0}
```

```
define Iff2 qq iffev : Mp qq Simp2 iffev


>> Iff2: [(.q_1:prop),(qq_1:that .q_1),(.p_1:
>>      prop),(iffev_1:that (.p_1 == .q_1)) =>
>>      ((qq_1 Mp Simp2(iffev_1)):that .p_1)]
>>   {move 0}
```

Rules of modus ponens for the biconditional

```
Lestrade execution:


declare dir1 [pp => that q] \




>> dir1: [(pp_1:that p) => (---:that q)]
>>   {move 1}



declare dir2 [qq => that p] \




>> dir2: [(qq_1:that q) => (---:that p)]
>>   {move 1}
```

```
define Dediff dir1, dir2 : Fixform(p==q,Conj \
   (Ded dir1,Ded dir2))

>> Dediff: [(.p_1:prop),(.q_1:prop),(dir1_1:
>>      [(pp_2:that .p_1) => (---:that .q_1)]),
>>      (dir2_1:[(qq_3:that .q_1) => (---:that
>>         .p_1)])
>>      => (((.p_1 == .q_1) Fixform (Ded(dir1_1)
>>      Conj Ded(dir2_1))):that (.p_1 == .q_1))]
>>   {move 0}



define Dediffdud dir1, dir2 : Conj (Ded dir1, \
   Ded dir2)

>> Dediffdud: [(.p_1:prop),(.q_1:prop),(dir1_1:
>>      [(pp_2:that .p_1) => (---:that .q_1)]),
>>      (dir2_1:[(qq_3:that .q_1) => (---:that
>>         .p_1)])
>>      => ((Ded(dir1_1) Conj Ded(dir2_1)):that
>>      ((.p_1 -> .q_1) & (.q_1 -> .p_1)))]
>>   {move 0}



define Iffrefl p: Dediff [pp => pp] \
   [pp => pp] \




>> Iffrefl: [(p_1:prop) => (Dediff([(pp_2:that
>>         p_1) => (pp_2:that p_1)]
>>      ,[(pp_3:that p_1) => (pp_3:that p_1)])
>>      :that (p_1 == p_1))]
>>   {move 0}
```

The deduction theorem for biconditionals. Notice the use of `Fixform` to force the reported type to be `that p == q` rather than `that (p -> q) & (q -> p)`. The version `Dediffdud` is provided to make this point. Of course `Fixform` only works here because the desired form is definitionally equivalent to the default form; Lestrade does check this (not by any special provision in the software, but automatically in the course of type checking the term).

```
Lestrade execution:

clearcurrent


declare p prop

>> p: prop {move 1}



declare pp that p

>> pp: that p {move 1}



declare porpev that p V p

>> porpev: that (p V p) {move 1}
```

The idempotent property of disjunction. This (below) demonstrates that from $p \vee p$ we can deduce p.

```
Lestrade execution:
```

```
define Oridem porpev : Cases porpev [pp => \
      pp] \
   , [pp=> pp] \
```

```
>> Oridem: [(.p_1:prop),(porpev_1:that (.p_1
>>      V .p_1)) => (Cases(porpev_1,[(pp_2:that
>>         .p_1) => (pp_2:that .p_1)]
>>      ,[(pp_3:that .p_1) => (pp_3:that .p_1)])
>>      :that .p_1)]
>>   {move 0}
```

```
clearcurrent
```

```
declare p prop

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

```
declare pp that p

>> pp: that p {move 1}
```

```
declare adabsurdum [pp => that ??] \
```

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

```
define Negintro adabsurdum : Fixform (~ p, \
   Ded adabsurdum)

>> Negintro: [(.p_1:prop),(adabsurdum_1:[(pp_2:
>>         that .p_1) => (---:that ??)])
>>      => ((~(.p_1) Fixform Ded(adabsurdum_1)):
>>      that ~(.p_1))]
>>   {move 0}
```

Here is an another example of using `Fixform`: `Negintro` is actually a special case of the deduction theorem; what `Fixform` does is force the surface expression of the implication $p \rightarrow \perp$ proved to $\neg p$.

We give the classical rules of inference from the negation of an implication.

```
Lestrade execution:

clearcurrent


declare p prop

>> p: prop {move 1}



declare q prop

>> q: prop {move 1}



declare notimp that ~(p -> q)

>> notimp: that ~((p -> q)) {move 1}
```

```
open

   declare qev that q

>>    qev: that q {move 2}



   open

      declare pev that p

>>       pev: that p {move 3}



      define idqev pev: qev

>>       idqev: [(pev_1:that p) => (---:that
>>            q)]
>>         {move 2}



      close

   define line1 qev : Mp( Ded idqev,notimp)


>>    line1: [(qev_1:that q) => (---:that ??)]
>>      {move 1}



   close
```

```
define Notimp1 notimp: Negintro line1

>> Notimp1: [(.p_1:prop),(.q_1:prop),(notimp_1:
>>      that ~((.p_1 -> .q_1))) => (Negintro([(qev_2:
>>         that .q_1) => ((Ded([(pev_3:that .p_1)
>>            => (qev_2:that .q_1)])
>>         Mp notimp_1):that ??)])
>>      :that ~(.q_1))]
>>   {move 0}



open

   declare negpev that ~p

>>    negpev: that ~(p) {move 2}



   open

      declare pev that p

>>       pev: that p {move 3}



      define line2 pev: Giveup q, Mp pev \
         negpev

>>       line2: [(pev_1:that p) => (---:that
>>            q)]
>>         {move 2}
```

```
      close

   define line3 negpev: Mp (Ded line2,notimp)


>>    line3: [(negpev_1:that ~(p)) => (---:that
>>         ??)]
>>      {move 1}



   close

define Notimp2 notimp: Dneg(Negintro line3)


>> Notimp2: [(.p_1:prop),(.q_1:prop),(notimp_1:
>>      that ~((.p_1 -> .q_1))) => (Dneg(Negintro([(negpev_2:
>>         that ~(.p_1)) => ((Ded([(pev_3:that
>>            .p_1) => ((.q_1 Giveup (pev_3 Mp
>>            negpev_2)):that .q_1)])
>>         Mp notimp_1):that ??)]))
>>      :that .p_1)]
>>   {move 0}



declare notconjev that ~(p & q)

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



declare qev that q

>> qev: that q {move 1}
```

```
open

   declare pev that p

>>    pev: that p {move 2}



   define line4 pev: Mp(Conj pev qev,notconjev)


>>    line4: [(pev_1:that p) => (---:that ??)]
>>      {move 1}



   close

define Notconj notconjev qev : Negintro line4


>> Notconj: [(.p_1:prop),(.q_1:prop),(notconjev_1:
>>      that ~((.p_1 & .q_1))),(qev_1:that .q_1)
>>      => (Negintro([(pev_2:that .p_1) => (((pev_2
>>         Conj qev_1) Mp notconjev_1):that ??)])
>>      :that ~(.p_1))]
>>   {move 0}
```

Above we present the rule of inference which gives us $\neg p$ if we have $\neg(p \wedge q)$ and q.

Now we develop the law of excluded middle.

```
Lestrade execution:

clearcurrent
```

```
declare p prop

>> p: prop {move 1}



open

   declare nona that ~(p V ~p)

>>    nona: that ~((p V ~(p))) {move 2}



   open

      declare thatp that p

>>       thatp: that p {move 3}



      define line1 thatp: Add1(~p,thatp)


>>       line1: [(thatp_1:that p) => (---:that
>>            (p V ~(p)))]
>>         {move 2}



      define line2 thatp: Mp (line1 thatp, \
         nona)

>>       line2: [(thatp_1:that p) => (---:that
>>            ??)]
>>        {move 2}



      close

   define line3 nona: Negintro line2

>>    line3: [(nona_1:that ~((p V ~(p)))) =>
>>         (---:that ~(p))]
>>      {move 1}



   define line4 nona: Add2(p,line3 nona)


>>    line4: [(nona_1:that ~((p V ~(p)))) =>
>>         (---:that (p V ~(p)))]
>>      {move 1}



   define line5 nona: Mp(line4 nona,nona)


>>    line5: [(nona_1:that ~((p V ~(p)))) =>
>>         (---:that ??)]
>>      {move 1}



   close

define Excmid p : Dneg(Negintro line5)

>> Excmid: [(p_1:prop) => (Dneg(Negintro([(nona_2:
>>         that ~((p_1 V ~(p_1)))) => (((p_1 Add2
>>         Negintro([(thatp_3:that p_1) => (((~(p_1)
>>            Add1 thatp_3) Mp nona_2):that ??)]))
>>         Mp nona_2):that ??)]))
>>      :that (p_1 V ~(p_1)))]
>>   {move 0}
```

We prove the rules of disjunctive syllogism.

```
Lestrade execution:

clearcurrent


declare p prop

>> p: prop {move 1}



declare q prop

>> q: prop {move 1}



declare orev that p V q

>> orev: that (p V q) {move 1}



declare negpev that ~p

>> negpev: that ~(p) {move 1}
```

```
declare negqev that ~q

>> negqev: that ~(q) {move 1}


open

   declare negpev2 that ~p

>>    negpev2: that ~(p) {move 2}


   declare negqev2 that ~q

>>    negqev2: that ~(q) {move 2}


   open

      declare thatp that p

>>       thatp: that p {move 3}


      declare thatq that q

>>       thatq: that q {move 3}


      define line1 thatp: Mp thatp negpev2
```

```
>>       line1: [(thatp_1:that p) => (---:that
>>            ??)]
>>         {move 2}



      define line2 thatp: Mp thatp negpev


>>       line2: [(thatp_1:that p) => (---:that
>>            ??)]
>>         {move 2}



      define line3 thatq: Mp thatq negqev2


>>       line3: [(thatq_1:that q) => (---:that
>>            ??)]
>>         {move 2}



      define line4 thatq: Mp thatq negqev


>>       line4: [(thatq_1:that q) => (---:that
>>            ??)]
>>         {move 2}



      close

   define line5 negpev2: Cases orev, line1, \
      line4
```

```
>>    line5: [(negpev2_1:that ~(p)) => (---:
>>         that ??)]
>>      {move 1}



   define line6 negqev2:Cases orev,line2, \
      line3

>>    line6: [(negqev2_1:that ~(q)) => (---:
>>         that ??)]
>>      {move 1}



   close

define Ds1 orev negqev: Dneg (Negintro line5)


>> Ds1: [(.p_1:prop),(.q_1:prop),(orev_1:that
>>      (.p_1 V .q_1)),(negqev_1:that ~(.q_1))
>>      => (Dneg(Negintro([(negpev2_2:that ~(.p_1))
>>         => (Cases(orev_1,[(thatp_3:that .p_1)
>>            => ((thatp_3 Mp negpev2_2):that
>>            ??)]
>>         ,[(thatq_4:that .q_1) => ((thatq_4
>>            Mp negqev_1):that ??)])
>>         :that ??)]))
>>      :that .p_1)]
>>   {move 0}



define Ds2 orev negpev: Dneg(Negintro line6)


>> Ds2: [(.p_1:prop),(.q_1:prop),(orev_1:that
>>        (.p_1 V .q_1)),(negpev_1:that ~(.p_1))
>>        => (Dneg(Negintro([(negqev2_2:that ~(.q_1))
>>           => (Cases(orev_1,[(thatp_3:that .p_1)
>>              => ((thatp_3 Mp negpev_1):that ??)]
>>           ,[(thatq_4:that .q_1) => ((thatq_4
>>              Mp negqev2_2):that ??)])
>>           :that ??)]))
>>        :that .q_1)]
>>     {move 0}
```

We enjoyed the sharing of environments in this proof. The reader will notice increasing size of the definition bodies of objections and constructions defined at move 0: recall that notions defined at moves of positive index are eliminated from these definitions by suitable expansions, which as we will see later can make definition bodies infeasibly large if we do not exercise care.

5 Quantification

In this section, we introduce the universal and existential quantifiers over untyped objects (sort `obj`) which will be the sets and atoms of our theory.

```
Lestrade execution:

clearcurrent


declare x obj

>> x: obj {move 1}



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



postulate Forall pred prop

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



postulate Exists pred prop

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

Here are the basic declarations for the universal and existential quantifiers (which are treated as separate primitives because they are separate primitives in constructive logic).

```
Lestrade execution:


declare univev that Forall pred

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



postulate Ui x univev that pred x

>> Ui: [(x_1:obj),(.pred_1:[(x_2:obj) => (---:
```

```
>>         prop)]),
>>      (univev_1:that Forall(.pred_1)) => (---:
>>      that .pred_1(x_1))]
>>   {move 0}
```

```
declare univev2 [x => that pred x] \
```

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

```
postulate 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 y obj

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

```
postulate = x y prop

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

```
postulate Refleq x that x=x

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

```
define Ugtest : Ug Refleq

>> Ugtest: [(Ug(Refleq):that Forall([(x_2:obj)
>>         => ((x_2 = x_2):prop)]))
>>      ]
>>   {move 0}
```

Here are initial declarations for equality and an example of universal generalization. The equality relation is declared, the rule of reflexivity presented, and the theorem $(\forall x : x = x)$ proved.

```
Lestrade execution:
```

```
declare existsev that pred x

>> existsev: that pred(x) {move 1}
```

```
postulate Ei x pred, existsev that Exists \
   pred

>> Ei: [(x_1:obj),(pred_1:[(x_2:obj) => (---:
>>         prop)]),
>>      (existsev_1:that pred_1(x_1)) => (---:
>>      that Exists(pred_1))]
>>   {move 0}
```

```
define Ei1 x existsev : Ei x pred, existsev


>> Ei1: [(x_1:obj),(.pred_1:[(x_2:obj) => (---:
>>         prop)]),
>>      (existsev_1:that .pred_1(x_1)) => (Ei(x_1,
>>      .pred_1,existsev_1):that Exists(.pred_1))]
>>   {move 0}



define Ei2 pred, existsev : Ei x pred, existsev


>> Ei2: [(pred_1:[(x_2:obj) => (---:prop)]),
>>      (.x_1:obj),(existsev_1:that pred_1(.x_1))
>>      => (Ei(.x_1,pred_1,existsev_1):that Exists(pred_1))]
>>   {move 0}



declare existsev2 that Exists pred

>> existsev2: that Exists(pred) {move 1}



declare r prop

>> r: prop {move 1}



declare witnessev [x,existsev => that r] \
>> witnessev: [(x_1:obj),(existsev_1:that pred(x_1))
>>      => (---:that r)]
>>   {move 1}
```

```
postulate Eg existsev2 witnessev that r

>> Eg: [(.pred_1:[(x_2:obj) => (---:prop)]),
>>      (existsev2_1:that Exists(.pred_1)),(.r_1:
>>      prop),(witnessev_1:[(x_3:obj),(existsev_3:
>>         that .pred_1(x_3)) => (---:that .r_1)])
>>      => (---:that .r_1)]
>>   {move 0}
```

```
declare test obj

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

```
define Witnesses existsev2 test : pred test


>> Witnesses: [(.pred_1:[(x_2:obj) => (---:prop)]),
>>      (existsev2_1:that Exists(.pred_1)),(test_1:
>>      obj) => (.pred_1(test_1):prop)]
>>   {move 0}
```

These are the declarations for the rules for the existential quantifier. The rule of existential instantiation comes in various flavors, because the ability of

Lestrade to deduce implicit arguments in higher-order situations is nonzero but limited. `Witnesses` is a gadget for extracting an instance of an existential statement whose surface form might not appear to be existential (so the user doesn't have to look up a definition).

```
Lestrade execution:

clearcurrent


declare x obj

>> x: obj {move 1}



declare y obj

>> y: obj {move 1}



declare eqev that x=y

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



declare pred [x=>prop] \




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

```
declare predev that pred x

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



postulate Subs eqev pred, predev that pred \
   y

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



define Subs1 eqev, predev : Subs eqev pred, \
   predev

>> Subs1: [(.x_1:obj),(.y_1:obj),(eqev_1:that
>>      (.x_1 = .y_1)),(.pred_1:[(x_2:obj) =>
>>         (---:prop)]),
>>      (predev_1:that .pred_1(.x_1)) => (Subs(eqev_1,
>>      .pred_1,predev_1):that .pred_1(.y_1))]
>>   {move 0}
```

The declaration of the rule of substitution (in two flavors) completes the logic of equality. The two flavors are present because the implicit argument feature is not perfectly dependable where constructions are involved in matching, so one will sometimes want Subs with the explicitly given predicate.

6 Lemmas of first order logic with equality

This is a space in which lemmas of first order logic with equality will be developed as needed below. Substitution comes in a flavor with the predicate left implicit, but this cannot always be done.

```
Lestrade execution:

clearcurrent


declare x obj

>> x: obj {move 1}



declare y obj

>> y: obj {move 1}



declare z obj

>> z: obj {move 1}



declare eqev1 that x=y

>> eqev1: that (x = y) {move 1}



declare eqev2 that y=z

>> eqev2: that (y = z) {move 1}
```

```
define Eqsymm eqev1 : Subs eqev1 [z=>z=x] \
   Refleq x

>> Eqsymm: [(.x_1:obj),(.y_1:obj),(eqev1_1:that
>>      (.x_1 = .y_1)) => (Subs(eqev1_1,[(z_2:
>>         obj) => ((z_2 = .x_1):prop)]
>>      ,Refleq(.x_1)):that (.y_1 = .x_1))]
>>   {move 0}



declare w obj

>> w: obj {move 1}



define Eqtrans eqev1 eqev2 : Subs1 eqev2 \
   eqev1

>> Eqtrans: [(.x_1:obj),(.y_1:obj),(eqev1_1:
>>      that (.x_1 = .y_1)),(.z_1:obj),(eqev2_1:
>>      that (.y_1 = .z_1)) => ((eqev2_1 Subs1
>>      eqev1_1):that (.x_1 = .z_1))]
>>   {move 0}



declare a17 obj

>> a17: obj {move 1}



declare b17 obj
```

```
>> b17: obj {move 1}


declare diffev that ~(a17 =b17)

>> diffev: that ~((a17 = b17)) {move 1}


declare funsies [a17 => obj] \



>> funsies: [(a17_1:obj) => (---:obj)]
>>   {move 1}


declare eqev17 that a17 = b17

>> eqev17: that (a17 = b17) {move 1}


define bothsides funsies, eqev17: Subs eqev17 \
   [b17 => funsies a17 = funsies b17] \
   Refleq funsies a17

>> bothsides: [(funsies_1:[(a17_2:obj) => (---:
>>           obj)]),
>>       (.a17_1:obj),(.b17_1:obj),(eqev17_1:that
>>       (.a17_1 = .b17_1)) => (Subs(eqev17_1,[(b17_3:
>>           obj) => ((funsies_1(.a17_1) = funsies_1(b17_3)):
>>           prop)]
>>       ,Refleq(funsies_1(.a17_1))):that (funsies_1(.a17_1)
>>      = funsies_1(.b17_1)))]
>>   {move 0}



open

   declare sillyhyp that b17=a17

>>    sillyhyp: that (b17 = a17) {move 2}



   define linec1 sillyhyp: Mp(Eqsymm sillyhyp, \
      diffev)

>>    linec1: [(sillyhyp_1:that (b17 = a17))
>>          => (---:that ??)]
>>      {move 1}



   close

define Negeqsymm diffev: Negintro linec1


>> Negeqsymm: [(.a17_1:obj),(.b17_1:obj),(diffev_1:
>>      that ~((.a17_1 = .b17_1))) => (Negintro([(sillyhyp_2:
>>         that (.b17_1 = .a17_1)) => ((Eqsymm(sillyhyp_2)
>>         Mp diffev_1):that ??)])
>>      :that ~((.b17_1 = .a17_1)))]
>>   {move 0}
```

Here are the expected symmetry and transitivity properties of equality, plus the ability to apply the same construction to both sides of an equation and symmetry for inequality. The exact lemmas present here and everywhere in this section are driven by what we turned out to need in the development of Zermelo's axiomatics and proof of the well-ordering theorem.

We prove the classical theorem that $\neg(\forall x : \phi) \rightarrow (\exists x : \neg\phi)$ (actually in the form of a rule of inference).

```
Lestrade execution:

clearcurrent


declare x obj

>> x: obj {move 1}



declare pred [x => prop] \




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



declare negunivev that ~(Forall pred)

>> negunivev: that ~(Forall(pred)) {move 1}



open

   declare z obj

>>    z: obj {move 2}
```

```
   declare negexistev that ~(Exists [z => \
         ~(pred z)]) \



>>    negexistev: that ~(Exists([(z_1:obj) =>
>>         (~(pred(z_1)):prop)]))
>>      {move 2}



   open

      declare y obj

>>       y: obj {move 3}



      open

         declare absurdhyp that ~(pred y)


>>          absurdhyp: that ~(pred(y)) {move
>>            4}



         define line1 absurdhyp: Ei1 y absurdhyp


>>          line1: [(absurdhyp_1:that ~(pred(y)))
>>               => (---:that Exists([(x_3:obj)
>>                  => (~(pred(x_3)):prop)]))
>>               ]
>>            {move 3}



         define line2 absurdhyp: Mp line1 \
            absurdhyp negexistev

>>          line2: [(absurdhyp_1:that ~(pred(y)))
>>               => (---:that ??)]
>>            {move 3}



         close

      define line3 y: Dneg Negintro line2


>>       line3: [(y_1:obj) => (---:that pred(y_1))]
>>         {move 2}



      close

   define line4 negexistev : Mp (Ug line3, \
      negunivev)

>>    line4: [(negexistev_1:that ~(Exists([(z_2:
>>            obj) => (~(pred(z_2)):prop)]))
>>         ) => (---:that ??)]
>>      {move 1}



   close
```

```
define Counterexample negunivev: Dneg (Negintro \
   line4)

>> Counterexample: [(.pred_1:[(x_2:obj) => (---:
>>         prop)]),
>>      (negunivev_1:that ~(Forall(.pred_1)))
>>      => (Dneg(Negintro([(negexistev_5:that
>>         ~(Exists([(z_6:obj) => (~(.pred_1(z_6)):
>>            prop)]))
>>         ) => ((Ug([(y_7:obj) => (Dneg(Negintro([(absurdhyp_8:
>>               that ~(.pred_1(y_7))) => (((y_7
>>               Ei1 absurdhyp_8) Mp negexistev_5):
>>               that ??)]))
>>            :that .pred_1(y_7))])
>>         Mp negunivev_1):that ??)]))
>>      :that Exists([(z_11:obj) => (~(.pred_1(z_11)):
>>         prop)]))
>>      ]
>>   {move 0}
```

7 Definite description

In this section we define the uniqueness quantifier and definite description.

```
Lestrade execution:

clearcurrent


declare x obj

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

```
declare y obj

>> y: obj {move 1}



declare pred [x => prop] \




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



define One pred : Exists [x => Forall [y \
         => (pred y) == y=x] \
      ] \




>> One: [(pred_1:[(x_2:obj) => (---:prop)])
>>      => (Exists([(x_3:obj) => (Forall([(y_4:
>>            obj) => ((pred_1(y_4) == (y_4 =
>>            x_3)):prop)])
>>         :prop)])
>>      :prop)]
>>   {move 0}



declare oneev that One pred

>> oneev: that One(pred) {move 1}
```

```
postulate The oneev : obj

>> The: [(.pred_1:[(x_2:obj) => (---:prop)]),
>>      (oneev_1:that One(.pred_1)) => (---:obj)]
>>   {move 0}



postulate Theax oneev that pred (The oneev)


>> Theax: [(.pred_1:[(x_2:obj) => (---:prop)]),
>>      (oneev_1:that One(.pred_1)) => (---:that
>>      .pred_1(The(oneev_1)))]
>>   {move 0}



declare pred2 [x => prop] \




>> pred2: [(x_1:obj) => (---:prop)]
>>   {move 1}



declare uniqueev that One pred

>> uniqueev: that One(pred) {move 1}



declare impliesev that Forall [x=> (pred \
      x) == pred2 x] \
>> impliesev: that Forall([(x_1:obj) => ((pred(x_1)
>>      == pred2(x_1)):prop)])
>>   {move 1}



goal that One pred2

>> Goal: that One(pred2)

open

   declare x1 obj

>>    x1: obj {move 2}



   declare y1 obj

>>    y1: obj {move 2}



   declare thisisit that Forall[y1=> (pred \
         y1) == y1=x1] \




>>    thisisit: that Forall([(y1_1:obj) => ((pred(y1_1)
>>         == (y1_1 = x1)):prop)])
>>      {move 2}
```

```
   goal that Forall[y1=> (pred2 y1) == y1=x1] \



>>    Goal: that Forall([(y1_29:obj) => ((pred2(y1_29)
>>         == (y1_29 = x1)):prop)])
>>

   open

      declare y2 obj

>>       y2: obj {move 3}



      open

         declare dir1 that pred2 y2

>>          dir1: that pred2(y2) {move 4}



         define line1 dir1 : Iff2 dir1, Ui \
            y2 impliesev

>>          line1: [(dir1_1:that pred2(y2))
>>               => (---:that pred(y2))]
>>            {move 3}



         define line2 dir1: Iff1 line1 dir1, \
            Ui y2 thisisit
```

```
>>          line2: [(dir1_1:that pred2(y2))
>>               => (---:that (y2 = x1))]
>>            {move 3}



         declare dir2 that y2 = x1

>>          dir2: that (y2 = x1) {move 4}



         define line3 dir2: Iff2 dir2, Ui \
            y2 thisisit

>>          line3: [(dir2_1:that (y2 = x1))
>>               => (---:that pred(y2))]
>>            {move 3}



         define line4 dir2: Iff1 line3 dir2, \
            Ui y2 impliesev

>>          line4: [(dir2_1:that (y2 = x1))
>>               => (---:that pred2(y2))]
>>            {move 3}



         close

      define line5 y2: Dediff line2, line4


>>       line5: [(y2_1:obj) => (---:that (pred2(y2_1)
>>            == (y2_1 = x1)))]
>>         {move 2}



      close

   define line6 thisisit : Ug line5

>>    line6: [(.x1_1:obj),(thisisit_1:that Forall([(y1_2:
>>            obj) => ((pred(y1_2) == (y1_2 =
>>            .x1_1)):prop)]))
>>         => (---:that Forall([(y2_11:obj) =>
>>            ((pred2(y2_11) == (y2_11 = .x1_1)):
>>            prop)]))
>>         ]
>>      {move 1}



   define line7 thisisit: Fixform(One pred2, \
      Ei1 x1 line6 thisisit)

>>    line7: [(.x1_1:obj),(thisisit_1:that Forall([(y1_2:
>>            obj) => ((pred(y1_2) == (y1_2 =
>>            .x1_1)):prop)]))
>>         => (---:that One(pred2))]
>>      {move 1}



   close

define Onequiv uniqueev impliesev: Eg uniqueev \
   line7

>> Onequiv: [(.pred_1:[(x_2:obj) => (---:prop)]),
>>      (uniqueev_1:that One(.pred_1)),(.pred2_1:
>>      [(x_3:obj) => (---:prop)]),
>>      (impliesev_1:that Forall([(x_4:obj) =>
>>         ((.pred_1(x_4) == .pred2_1(x_4)):prop)]))
>>      => ((uniqueev_1 Eg [(.x1_7:obj),(thisisit_7:
>>         that Forall([(y1_8:obj) => ((.pred_1(y1_8)
>>            == (y1_8 = .x1_7)):prop)]))
>>         => ((One(.pred2_1) Fixform (.x1_7 Ei1
>>         Ug([(y2_12:obj) => (Dediff([(dir1_13:
>>               that .pred2_1(y2_12)) => (((dir1_13
>>               Iff2 (y2_12 Ui impliesev_1))
>>               Iff1 (y2_12 Ui thisisit_7)):that
>>               (y2_12 = .x1_7))]
>>            ,[(dir2_16:that (y2_12 = .x1_7))
>>               => (((dir2_16 Iff2 (y2_12 Ui
>>               thisisit_7)) Iff1 (y2_12 Ui impliesev_1)):
>>               that .pred2_1(y2_12))])
>>            :that (.pred2_1(y2_12) == (y2_12
>>            = .x1_7)))]))
>>         ):that One(.pred2_1))])
>>      :that One(.pred2_1))]
>>   {move 0}
```

Here we develop the universal quantifier and definite description operator. Note that the definite description operator and its axiom are for us primitives.