

Implementation of Zermelo's work of 1908 in Lestrade: Part II, Axiomatics of Zermelo set theory

M. Randall Holmes

April 16, 2020

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 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.

This is the version being developed under Lestrade release 2.0, which still throws errors.

2 Basic concepts of set theory: the axioms of extensionality and pairing

In this section, we start to declare the basic notions and axioms of 1908 Zermelo set theory. The membership relation is declared. The axioms declared here are existence of the empty set, weak extensionality (atoms are allowed, following Zermelo's clear intentions in the 1908 paper), and pairing.

I have reedited this file to be a fairly direct implementation of Zermelo's axiomatics paper, currently just the first part discussing the axioms, but intended to include the development of theory of equivalence. The way it was initially written was a correct implementation of the axioms, but concepts were not presented in the same order. We will leave in the anachronistic demonstration of the basic property of the Kuratowski pair, which belongs at the same level of exposition. I will add comments in this pass corresponding to paragraph numbers in the Zermelo paper.

```
begin Lestrade execution
```

```
>>> comment load whatismath1
```

```
{move 1}
```

```
>>> clearcurrent
```

```
{move 1}
```

```
>>> declare x obj
```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 1}
```

```
>>> define /= x y : ~ (x = y)
```

```

=/> : [(x_1 : obj), (y_1 : obj) =>
      ({def} ~ (x_1 = y_1) : prop)]

```

```

=/> : [(x_1 : obj), (y_1 : obj) =>
      (--- : prop)]

```

```

{move 0}

```

```

>>> postulate E x y prop

```

```

E : [(x_1 : obj), (y_1 : obj) =>
      (--- : prop)]

```

```

{move 0}

```

```

>>> postulate 0 obj

```

```

0 : obj

```

```

{move 0}

```

```

>>> postulate Empty x that ~ (x E 0)

```

```

Empty : [(x_1 : obj) => (--- : that
      ~ (x_1 E 0))]

```

```

{move 0}

```

```

>>> define Isset x : (x = 0) V Exists \
    [y => y E x]

Isset : [(x_1 : obj) =>
    ({def} (x_1 = 0) V Exists ([y_3
    : obj) =>
    ({def} y_3 E x_1 : prop)]) : prop)]

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

{move 0}

>>> declare u1 obj

u1 : obj

{move 1}

>>> declare v1 obj

v1 : obj

{move 1}

>>> declare nonemptyev that u1 E v1

nonemptyev : that u1 E v1

{move 1}

```

```
>>> define Inhabited nonemptyev : Fixform \
      (Isset v1, Add2 (v1 = 0, Ei1 u1 nonemptyev))
```

```
Inhabited : [(u1_1 : obj), (v1_1
      : obj), (nonemptyev_1 : that .u1_1
      E .v1_1) =>
      ({def} Isset (.v1_1) Fixform (.v1_1
      = 0) Add2 .u1_1 Ei1 nonemptyev_1 : that
      Isset (.v1_1))]
```

```
Inhabited : [(u1_1 : obj), (v1_1
      : obj), (nonemptyev_1 : that .u1_1
      E .v1_1) => (--- : that Isset (.v1_1))]
```

```
{move 0}
```

```
>>> declare z obj
```

```
z : obj
```

```
{move 1}
```

```
>>> define <=< x y : Forall [z => (z E x) -> \
      z E y] & (Isset x) & Isset y
```

```
<=< : [(x_1 : obj), (y_1 : obj) =>
      ({def} Forall ([z_3 : obj) =>
      ({def} (z_3 E x_1) -> z_3 E y_1
      : prop)]) & Isset (x_1) & Isset
      (y_1) : prop)]
```

```

<=< : [(x_1 : obj), (y_1 : obj) =>
      (--- : prop)]

{move 0}

>>> define disjoint x y : ~ Exists [z => \
      (z E x) & z E y] & Isset x & Isset \
      y

disjoint : [(x_1 : obj), (y_1 : obj) =>
      ({def} ~ (Exists ([z_4 : obj] =>
        ({def} (z_4 E x_1) & z_4 E y_1
          : prop)))) & Isset (x_1) & Isset
      (y_1) : prop)]

disjoint : [(x_1 : obj), (y_1 : obj) =>
      (--- : prop)]

{move 0}
end Lestrade execution

```

We define the subset relation. Note that we stipulate that it only holds between sets, which means that the atoms do not sneak into the power sets, and the power set of an atom is the empty set.

The form of our definition of set agrees with what Zermelo says in the axiomatics paper: it is a relation only between sets, not between the atoms which might exist.

We further define the disjointness relation between sets.

```

begin Lestrade execution

```

```

>>> clearcurrent

```

```

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> declare subsev1 that x <=< y

subsev1 : that x <=< y

{move 1}

>>> declare subsev2 that y <=< z

```

```
subsev2 : that y <= z
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare u obj
```

```
u : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare uinev that u E x
```

```
uinev : that u E x
```

```
{move 3}
```

```
>>> define line1 uinev : Mp uinev, Ui \  
    u Simp1 subsev1
```

```
line1 : [(uinev_1 : that u E x) =>  
    (--- : that u E y)]
```



```
{move 2}
```

```
>>> define line2 uinev : Mp (line1 \
    uinev, Ui u Simp1 subsev2)
```

```
line2 : [(uinev_1 : that u E x) =>
    (--- : that u E z)]
```

```
{move 2}
```

```
>>> close
```

```
{move 2}
```

```
>>> define line3 u : Ded line2
```

```
line3 : [(u_1 : obj) => (--- : that
    (u_1 E x) -> u_1 E z)]
```

```
{move 1}
```

```
>>> close
```

```
{move 1}
```

```
>>> define Transsub subsev1 subsev2 : Fixform \
    (x <= z, (Ug line3) Conj (Simp1 \
    Simp2 subsev1) Conj (Simp2 Simp2 subsev2))
```

```
Transsub : [(x_1 : obj), (y_1 : obj), (z_1
```

```

: obj), (subsev1_1 : that .x_1 <=<=
.y_1), (subsev2_1 : that .y_1 <=<=
.z_1) =>
({def} (.x_1 <=<= .z_1) Fixform Ug
[(u_4 : obj) =>
  ({def} Ded ([uinev_5 : that
    u_4 E .x_1) =>
    ({def} uinev_5 Mp u_4 Ui Simp1
    (subsev1_1) Mp u_4 Ui Simp1
    (subsev2_1) : that u_4 E .z_1)]) : that
    (u_4 E .x_1) -> u_4 E .z_1)]) Conj
Simp1 (Simp1 (subsev1_1)) Conj
Simp2 (Simp2 (subsev2_1)) : that
.x_1 <=<= .z_1)]

```

```

Transsub : [(x_1 : obj), (y_1 : obj), (z_1
: obj), (subsev1_1 : that .x_1 <=<=
.y_1), (subsev2_1 : that .y_1 <=<=
.z_1) => (--- : that .x_1 <=<= .z_1)]

```

```

{move 0}
end Lestrade execution

```

We prove the transitive property of the subset relation.

```

begin Lestrade execution

>>> declare issetx that Isset x

issetx : that Isset (x)

{move 1}

```

```

>>> open

{move 2}

>>> declare u obj

u : obj

{move 2}

>>> open

{move 3}

>>> declare uinev that u E x

uinev : that u E x

{move 3}

>>> define line1 uinev : uinev

line1 : [(uinev_1 : that u E x) =>
  (--- : that u E x)]

{move 2}

>>> close

```

```

{move 2}

>>> define linea2 u : Ded line1

linea2 : [(u_1 : obj) => (--- : that
      (u_1 E x) -> u_1 E x)]

{move 1}

>>> close

{move 1}

>>> define Reflsubset issetx : Fixform \
      (x <= x, (Ug linea2) Conj issetx \
      Conj issetx)

Reflsubset : [(x_1 : obj), (issetx_1
      : that Isset (.x_1)) =>
      ({def} (.x_1 <= .x_1) Fixform Ug
      [(u_4 : obj) =>
        ({def} Ded [(uinev_5 : that
          u_4 E .x_1) =>
            ({def} uinev_5 : that u_4 E .x_1)]) : that
          (u_4 E .x_1) -> u_4 E .x_1)]) Conj
      issetx_1 Conj issetx_1 : that .x_1
      <= .x_1)]

Reflsubset : [(x_1 : obj), (issetx_1
      : that Isset (.x_1)) => (--- : that
      .x_1 <= .x_1)]

```

```

{move 0}
end Lestrade execution

```

We prove the reflexive property of the subset relation (as a relation on sets).

```

begin Lestrade execution

```

```

>>> declare ineq that x E y

```

```

ineq : that x E y

```

```

{move 1}

```

```

>>> declare subeq that y <= z

```

```

subeq : that y <= z

```

```

{move 1}

```

```

>>> define Mpsubs ineq subeq : Mp (ineq, Ui \
    x Simp1 subeq)

```

```

Mpsubs : [(x_1 : obj), (y_1 : obj), (z_1
    : obj), (ineq_1 : that x_1 E y_1), (subeq_1
    : that y_1 <= z_1) =>
    ({def} ineq_1 Mp x_1 Ui Simp1 (subeq_1) : that
    x_1 E z_1)]

```

```

Mpsubs : [(x_1 : obj), (y_1 : obj), (z_1
    : obj), (ineq_1 : that x_1 E y_1), (subeq_1

```

```

: that .y_1 <=<= .z_1) => (--- : that
.x_1 E .z_1)]

```

```

{move 0}
end Lestrade execution

```

This is the frequently useful rule of inference taking $x \in y$ and $y \subseteq z$ to $x \in z$.

```

begin Lestrade execution

```

```

>>> open

```

```

{move 2}

```

```

>>> declare X obj

```

```

X : obj

```

```

{move 2}

```

```

>>> open

```

```

{move 3}

```

```

>>> declare Xsetev that Isset X

```

```

Xsetev : that Isset (X)

```

```

{move 3}

```

```

>>> open

{move 4}

>>> declare u obj

u : obj

{move 4}

>>> open

{move 5}

>>> declare uinxev that u E X

uinxe v : that u E X

{move 5}

>>> define line1 uinxev : uinxev

line1 : [(uinxe v_1 : that
          u E X) => (--- : that
          u E X)]

{move 4}

>>> close

```

```

{move 4}

>>> define line2 u : Ded line1

line2 : [(u_1 : obj) => (---
      : that (u_1 E X) -> u_1
      E X)]

{move 3}

>>> close

{move 3}

>>> define line3 : Ug line2

line3 : that Forall ([(x''_2
      : obj) =>
      ({def} (x''_2 E X) -> x''_2
      E X : prop)])

{move 2}

>>> define line4 Xsetev : Fixform \
      (X <=< X, line3 Conj Xsetev Conj \
      Xsetev)

line4 : [(Xsetev_1 : that Isset
      (X)) => (--- : that X <=<
      X)]

```



```

{move 2}

>>> close

{move 2}

>>> define line5 X : Ded line4

line5 : [(X_1 : obj) => (--- : that
      Isset (X_1) -> X_1 <=< X_1)]

{move 1}

>>> close

{move 1}

>>> define Subsetrefl : Ug line5

Subsetrefl : [
  ({def} Ug ([X_2 : obj) =>
    ({def} Ded ([Xsetev_3 : that
      Isset (X_2)) =>
        ({def} (X_2 <=< X_2) Fixform
          Ug ([u_6 : obj) =>
            ({def} Ded ([uinxev_7
              : that u_6 E X_2) =>
                ({def} uinxev_7 : that
                  u_6 E X_2])) : that
              (u_6 E X_2) -> u_6 E X_2])) Conj
            Xsetev_3 Conj Xsetev_3 : that

```

```

      X_2 <= X_2)) : that Isset
      (X_2 -> X_2 <= X_2)) : that
Forall ((x''_2 : obj) =>
  ({def} Isset (x''_2) -> x''_2
  <= x''_2 : prop))))]

```

```

Subsetrefl : that Forall ((x''_2 : obj) =>
  ({def} Isset (x''_2) -> x''_2 <=
  x''_2 : prop]))

```

```

{move 0}
end Lestrade execution

```

I do not know why I proved reflexivity of the subset relation again, but I am going to leave it alone for now.

```

begin Lestrade execution

```

```

>>> define Zeroisset : Fixform (Isset \
  0, Add1 (Exists [x => x E 0], Refleq \
  0))

```

```

Zeroisset : [
  ({def} Isset (0) Fixform Exists
  ((x_4 : obj) =>
    ({def} x_4 E 0 : prop))) Add1
  Refleq (0) : that Isset (0))]

```

```

Zeroisset : that Isset (0)

```

```

{move 0}
end Lestrade execution

```

The empty set is a set.

```
begin Lestrade execution
```

```
>>> declare firstev that Isset x
```

```
firstev : that Isset (x)
```

```
{move 1}
```

```
>>> declare secondev that Isset y
```

```
secondev : that Isset (y)
```

```
{move 1}
```

```
>>> declare thirdev that ~ (x <= y)
```

```
thirdev : that ~ (x <= y)
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> define linec1 : Counterexample \  
      (Notconj (thirdev, Conj firstev \  
        secondev))
```

```

linec1 : that Exists ([ (z_2 : obj) =>
    ({def} ~ ((z_2 E x) -> z_2 E y) : prop)])

```

```

{move 1}

```

```

>>> open

```

```

{move 3}

```

```

>>> declare z1 obj

```

```

z1 : obj

```

```

{move 3}

```

```

>>> declare u1 obj

```

```

u1 : obj

```

```

{move 3}

```

```

>>> declare evu1 that ~ ((u1 E x) -> \
    u1 E y)

```

```

evu1 : that ~ ((u1 E x) -> u1
    E y)

```

```

{move 3}

```

```
>>> define linec2 u1 evu1 : Ei1 \
      u1, Conj (Notimp2 evu1, Notimp1 \
      evu1)
```

```
linec2 : [(u1_1 : obj), (evu1_1
      : that ~ ((u1_1 E x) -> u1_1
      E y)) => (--- : that Exists
      ((x'_2 : obj) =>
      ({def} (x'_2 E x) & ~ (x'_2
      E y) : prop)))]
```

```
{move 2}
```

```
>>> close
```

```
{move 2}
```

```
>>> define Subsetcounter1 : Eg linec1, linec2
```

```
Subsetcounter1 : that Exists ((x'_2
      : obj) =>
      ({def} (x'_2 E x) & ~ (x'_2
      E y) : prop))
```

```
{move 1}
```

```
>>> close
```

```
{move 1}
```

```
>>> define Subsetcounter firstev secondev \
      thirdev : Subsetcounter1
```

```

Subsetcounter : [(x_1 : obj), (y_1
: obj), (firstev_1 : that Isset
(x_1)), (seconde_1 : that Isset
(y_1)), (thirdev_1 : that ~ (x_1
<=& y_1)) =>
({def} Counterexample (thirdev_1
Notconj firstev_1 Conj seconde_1) Eg
[(u1_2 : obj), (evu1_2 : that
~ ((u1_2 E x_1) -> u1_2 E y_1)) =>
({def} u1_2 Ei1 Notimp2 (evu1_2) Conj
Notimp1 (evu1_2) : that Exists
[(x'_3 : obj) =>
({def} (x'_3 E x_1) & ~ (x'_3
E y_1) : prop))]] : that
Exists [(x'_2 : obj) =>
({def} (x'_2 E x_1) & ~ (x'_2
E y_1) : prop))]]]

```

```

Subsetcounter : [(x_1 : obj), (y_1
: obj), (firstev_1 : that Isset
(x_1)), (seconde_1 : that Isset
(y_1)), (thirdev_1 : that ~ (x_1
<=& y_1)) => (--- : that Exists
[(x'_2 : obj) =>
({def} (x'_2 E x_1) & ~ (x'_2
E y_1) : prop))]]]

```

```

{move 0}
end Lestrade execution

```

I don't think I used this result, but it is nice to have it in the library (existence of witnesses to failures of inclusion).

```

begin Lestrade execution

>>> declare setev1 that Isset x

setev1 : that Isset (x)

{move 1}

>>> declare setev2 that Isset y

setev2 : that Isset (y)

{move 1}

>>> declare extev [z => that (z E x) == \
      (z E y)]

extev : [(z_1 : obj) => (--- : that
      (z_1 E x) == z_1 E y)]

{move 1}

>>> postulate Ext setev1 setev2 extev \
      that x = y

Ext : [(x_1 : obj), (y_1 : obj), (setev1_1
      : that Isset (x_1)), (setev2_1
      : that Isset (y_1)), (extev_1
      : [(z_2 : obj) => (--- : that (z_2
      E x_1) == z_2 E y_1)]) =>
      (--- : that x_1 = y_1)]

```

```

{move 0}

>>> clearcurrent

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> declare setev that z E x

setev : that z E x

```



```

{move 1}

>>> declare setev2 that z E y

setev2 : that z E y

{move 1}

>>> declare extev1 [setev => that z E y]

extev1 : [(setev_1 : that z E x) =>
  (--- : that z E y)]

{move 1}

>>> declare extev2 [setev2 => that z E y]

extev2 : [(setev2_1 : that z E y) =>
  (--- : that z E y)]

{move 1}

>>> postulate Ext1 setev extev1, extev2 \
  that x = y

Ext1 : [(x_1 : obj), (y_1 : obj), (z_1
  : obj), (setev_1 : that z_1 E x_1), (extev1_1
  : [(setev_2 : that z_1 E x_1) =>
    (--- : that z_1 E y_1)]), (extev2_1
  : [(setev2_2 : that z_1 E y_1) =>
    (--- : that z_1 E y_1)]) =>

```

```
(--- : that .x_1 = .y_1)]
```

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

Above we have declared the membership relation $x \in y$, the empty set 0 and the axiom that it has no members, defined sets as elements and 0, and stated the weak axiom of extensionality: sets which have the same extension are equal.

The definition of “set” (and the possibility of objects which are not sets) is clearly stated in Zermelo’s axiomatics paper.

The alternative formulation `Ext1` is better in not involving logic primitives, which would add a little more burden to needed definitions. I should define one of these in terms of the other.

The rule of inference `Inhabited` from $x \in y$ to sethood of y is often useful.

```
begin Lestrade execution  
  
>>> declare sev1 that x <=< y  
  
sev1 : that x <=< y  
  
{move 1}  
  
>>> declare sev2 that y <=< x  
  
sev2 : that y <=< x  
  
{move 1}  
  
>>> open
```

```
{move 2}
```

```
>>> declare u obj
```

```
u : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare ineval that u E x
```

```
ineval : that u E x
```

```
{move 3}
```

```
>>> declare ineval2 that u E y
```

```
ineval2 : that u E y
```

```
{move 3}
```

```
>>> define dir1 ineval : Mpsubs \  
      ineval sev1
```

```
dir1 : [(ineval_1 : that u E x) =>  
      (--- : that u E y)]
```

```

{move 2}

>>> define dir2 ineva2 : Mpsubs \
      ineva2 sev2

dir2 : [(ineva2_1 : that u E y) =>
      (--- : that u E x)]

{move 2}

>>> close

{move 2}

>>> define bothways u : Dediff dir1, dir2

bothways : [(u_1 : obj) => (---
      : that (u_1 E x) == u_1 E y)]

{move 1}

>>> close

{move 1}

>>> define Antisymsub sev1 sev2 : Ext \
      (Simp1 (Simp2 sev1), Simp2 (Simp2 \
      sev1), bothways)

```

```

Antisymsub : [(x_1 : obj), (y_1
  : obj), (sev1_1 : that x_1 <=<=
    y_1), (sev2_1 : that y_1 <=<= x_1) =>
  ({def} Ext (Simp1 (Simp2 (sev1_1)), Simp2
    (Simp2 (sev1_1)), [(u_2 : obj) =>
      ({def} Dediff ([ineva1_3 : that
        u_2 E x_1) =>
          ({def} ineva1_3 Mpsubs sev1_1
            : that u_2 E y_1)], [(ineva2_3
              : that u_2 E y_1) =>
                ({def} ineva2_3 Mpsubs sev2_1
                  : that u_2 E x_1)]) : that
        (u_2 E x_1) == u_2 E y_1)]) : that
    x_1 = y_1)]

```

```

Antisymsub : [(x_1 : obj), (y_1
  : obj), (sev1_1 : that x_1 <=<=
    y_1), (sev2_1 : that y_1 <=<= x_1) =>
  (--- : that x_1 = y_1)]

```

```

{move 0}
end Lestrade execution

```

We prove that the subset relation is antisymmetric (which is an alternative way in which Zermelo states extensionality).

```

begin Lestrade execution

```

```

>>> clearcurrent

```

```

{move 1}

```

```

>>> declare x obj

```

```

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> postulate ; x y obj

; : [(x_1 : obj), (y_1 : obj) =>
    (--- : obj)]

{move 0}

>>> postulate Pair x y that Forall [z => \
    (z E x ; y) == (z = x) V z = y]

Pair : [(x_1 : obj), (y_1 : obj) =>
    (--- : that Forall ([z_2 : obj) =>
        ({def} (z_2 E x_1 ; y_1) == (z_2
            = x_1) V z_2 = y_1 : prop)))]

```

```

{move 0}

>>> define Usc x : x ; x

Usc : [(x_1 : obj) =>
      ({def} x_1 ; x_1 : obj)]

Usc : [(x_1 : obj) => (--- : obj)]

{move 0}

>>> define $ x y : (x ; x) ; (x ; y)

$ : [(x_1 : obj), (y_1 : obj) =>
     ({def} (x_1 ; x_1) ; x_1 ; y_1 : obj)]

$ : [(x_1 : obj), (y_1 : obj) =>
     (--- : obj)]

{move 0}
end Lestrade execution

```

Above we present the operation of unordered pairing and the axiom of pairing which determines the extension of the pair. We write $x ; y$ for $\{x, y\}$. We define the singleton operation, borrowing Rosser's notation $USC(x)$ for $\{x\}$.

We define the Kuratowski ordered pair, using the notation xy$ for (x, y) . This is of course a notion unknown to Zermelo, but it is a formal feature of his system even if he did not know about it.

Our treatment differs from Zermelo's in treating the singleton as a special

case of the unordered pair. He treats the two as separate constructions.

3 Developments from pairing, including the properties of the ordered pair

Herein we do some development work with unordered pairs, singletons, and Kuratowski ordered pairs. The results on Kuratowski ordered pairs are anachronistic, having nothing to do with Zermelo's development, and we do not make use of these in implementing Zermelo's proofs; lemmas provided about singletons and ordered pairs are used extensively, though it should be noted that strictly speaking Zermelo's well-ordering theorem proof does not actually depend on the axiom of pairing (pairs of objects taken from a set given in advance are provided by separation, and this is all that is actually needed in Zermelo's proof; we might at some point revise the development here to highlight this fact).

```
begin Lestrade execution
```

```
>>> clearcurrent
```

```
{move 1}
```

```
>>> declare x obj
```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```



```

{move 1}

>>> declare ineq that  $y \in x ; x$ 

ineq : that  $y \in x ; x$ 

{move 1}

>>> open

{move 2}

>>> define line1 :  $\text{Ui } (y, \text{Pair } x \ x)$ 

line1 : that  $(y \in x ; x) \implies (y = x) \vee y = x$ 

{move 1}

>>> define line2 :  $\text{Iff1 ineq line1}$ 

line2 : that  $(y = x) \vee y = x$ 

{move 1}

>>> define line3 :  $\text{Oridem line2}$ 

line3 : that  $y = x$ 

{move 1}

```

```

>>> close

{move 1}

>>> define Inusc1 inev : line3

Inusc1 : [(x_1 : obj), (y_1 : obj), (inev_1
      : that y_1 E x_1 ; x_1) =>
      ({def} Oridem (inev_1 Iff1 y_1 Ui
        .x_1 Pair x_1) : that y_1 = x_1)]

Inusc1 : [(x_1 : obj), (y_1 : obj), (inev_1
      : that y_1 E x_1 ; x_1) => (---
      : that y_1 = x_1)]

{move 0}

>>> clearcurrent

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> open

{move 2}

```

```

>>> define line1 : Add1 (x = x, Refleq \
    x)

line1 : that (x = x) V x = x

{move 1}

>>> define line2 : Iff2 (line1, Ui \
    (x, Pair x x))

line2 : that x E x ; x

{move 1}

>>> close

{move 1}

>>> define Inusc2 x : line2

Inusc2 : [(x_1 : obj) =>
    ({def} (x_1 = x_1) Add1 Refleq (x_1) Iff2
    x_1 Ui x_1 Pair x_1 : that x_1 E x_1
    ; x_1)]

Inusc2 : [(x_1 : obj) => (--- : that
    x_1 E x_1 ; x_1)]

{move 0}

```

```

>>> clearcurrent

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> open

{move 2}

>>> define scratch1 : Ui x (Pair x y)

scratch1 : that (x E x ; y) == (x = x) V x = y

{move 1}

>>> define scratch2 : Add1 (x = y, Refleq \
    x)

```

```

scratch2 : that (x = x) V x = y

{move 1}

>>> define scratch3 : Iff2 (scratch2, scratch1)

scratch3 : that x E x ; y

{move 1}

>>> close

{move 1}

>>> define Inpair1 x y : scratch3

Inpair1 : [(x_1 : obj), (y_1 : obj) =>
  ({def} (x_1 = y_1) Add1 Refleq (x_1) Iff2
    x_1 Ui x_1 Pair y_1 : that x_1 E x_1
    ; y_1)]

Inpair1 : [(x_1 : obj), (y_1 : obj) =>
  (--- : that x_1 E x_1 ; y_1)]

{move 0}

>>> clearcurrent

{move 1}

>>> declare x obj

```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> define scratch1 : Ui y (Pair x y)
```

```
scratch1 : that (y E x ; y) == (y = x) V y = y
```

```
{move 1}
```

```
>>> define scratch2 : Add2 (y = x, Refleq \
    y)
```

```
scratch2 : that (y = x) V y = y
```

```
{move 1}
```

```
>>> define scratch3 : Iff2 scratch2 \
```

```

scratch1

scratch3 : that y E x ; y

{move 1}

>>> close

{move 1}

>>> define Inpair2 x y : scratch3

Inpair2 : [(x_1 : obj), (y_1 : obj) =>
  ({def} (y_1 = x_1) Add2 Refleq (y_1) Iff2
    y_1 Ui x_1 Pair y_1 : that y_1 E x_1
    ; y_1)]

Inpair2 : [(x_1 : obj), (y_1 : obj) =>
  (--- : that y_1 E x_1 ; y_1)]

{move 0}

>>> clearcurrent

{move 1}

>>> declare x obj

x : obj

```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare ineq1 that y E Usc x
```

```
ineq1 : that y E Usc (x)
```

```
{move 3}
```

```
>>> declare ineq2 that y = x
```

```
ineq2 : that y = x
```

```
{move 3}
```

```
>>> define dir1 ineq1 : Inusc1 ineq1
```



```

dir1 : [(inev1_1 : that y E Usc
      (x)) => (--- : that y = x)]

{move 2}

>>> define line3 ineq2 : Eqsymm \
      ineq2

line3 : [(ineq2_1 : that y = x) =>
      (--- : that x = y)]

{move 2}

>>> define line4 : Fixform (x E Usc \
      x, Inusc2 x)

line4 : that x E Usc (x)

{move 2}

>>> declare z1 obj

z1 : obj

{move 3}

>>> define dir2 ineq2 : Subs (Eqsymm \
      ineq2, [z1 => z1 E Usc x], line4)

```

```

dir2 : [(inev2_1 : that y = x) =>
      (--- : that y E Usc (x))]

{move 2}

>>> define inuscone : Fixform ((y E Usc \
      x) == y = x, Dediff dir1, dir2)

inuscone : that (y E Usc (x)) ==
      y = x

{move 2}

>>> close

{move 2}

>>> define inuscone2 y : inuscone

inuscone2 : [(y_1 : obj) => (---
      : that (y_1 E Usc (x)) == y_1
      = x)]

{move 1}

>>> define one1 : Ug inuscone2

one1 : that Forall ([(x''_2 : obj) =>
      ({def} (x''_2 E Usc (x)) ==
      x''_2 = x : prop)])

```

```

{move 1}

>>> declare w obj

w : obj

{move 2}

>>> declare y2 obj

y2 : obj

{move 2}

>>> define one2 : Fixform (One [w => \
      w E Usc x], Ei (x, [w => Forall \
      [y2 => (y2 E Usc x) == y2 = w]], one1))

one2 : that One ([w_2 : obj) =>
  ({def} w_2 E Usc (x) : prop)])

{move 1}

>>> define one3 : Theax one2

one3 : that The (one2) E Usc (x)

{move 1}

```

```

>>> define one4 : Inusc1 one3

one4 : that The (one2) = x

{move 1}

>>> close

{move 1}

>>> define Theeltthm x : one2

Theeltthm : [(x_1 : obj) =>
  ({def} One ([(w_3 : obj) =>
    ({def} w_3 E Usc (x_1) : prop)]) Fixform
  Ei (x_1, [(w_3 : obj) =>
    ({def} Forall ([(y2_4 : obj) =>
      ({def} (y2_4 E Usc (x_1)) ==
        y2_4 = w_3 : prop)]) : prop)], Ug
    [(y_4 : obj) =>
      ({def} ((y_4 E Usc (x_1)) ==
        y_4 = x_1) Fixform Dediff ([(inev1_6
        : that y_4 E Usc (x_1)) =>
        ({def} Inusc1 (inev1_6) : that
        y_4 = x_1)], [(inev2_6 : that
        y_4 = x_1) =>
        ({def} Subs (Eqsymm (inev2_6), [(z1_7
        : obj) =>
        ({def} z1_7 E Usc (x_1) : prop)], (x_1
        E Usc (x_1)) Fixform Inusc2
        (x_1)) : that y_4 E Usc (x_1)))] : that
        (y_4 E Usc (x_1)) == y_4 = x_1)])) : that
  One ([(w_2 : obj) =>
    ({def} w_2 E Usc (x_1) : prop)])]]]

```

```
Theeltthm : [(x_1 : obj) => (--- : that
  One [(w_2 : obj) =>
    ({def} w_2 E Usc (x_1) : prop)])]]
```

```
{move 0}
```

```
>>> define Theelt x : Fixform (The (Theeltthm \
  x) = x, one4)
```

```
Theelt : [(x_1 : obj) =>
  ({def} (The (Theeltthm (x_1)) = x_1) Fixform
  Inusc1 (Theax (One [(w_6 : obj) =>
    ({def} w_6 E Usc (x_1) : prop)]) Fixform
  Ei (x_1, [(w_6 : obj) =>
    ({def} Forall [(y2_7 : obj) =>
      ({def} (y2_7 E Usc (x_1)) ==
        y2_7 = w_6 : prop)]) : prop)], Ug
  [(y_7 : obj) =>
    ({def} ((y_7 E Usc (x_1)) ==
      y_7 = x_1) Fixform Dediff [(inev1_9
        : that y_7 E Usc (x_1)) =>
      ({def} Inusc1 (inev1_9) : that
        y_7 = x_1)], [(inev2_9 : that
        y_7 = x_1) =>
      ({def} Subs (Eqsymm (inev2_9), [(z1_10
        : obj) =>
      ({def} z1_10 E Usc (x_1) : prop)], (x_1
        E Usc (x_1)) Fixform Inusc2
      (x_1)) : that y_7 E Usc (x_1)))] : that
      (y_7 E Usc (x_1)) == y_7 = x_1)])))] : that
  The (Theeltthm (x_1)) = x_1]
```

```
Theelt : [(x_1 : obj) => (--- : that
```

The (Theeltthm (x_1)) = x_1)]

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

We prove that $y \in \{x\}$ iff $y = x$, and that $(\theta y : y \in \{x\}) = x$. This involves careful manipulations of environments and forms of statements to avoid blowup.

We should also prove that if there is only one element in a set, it is the singleton of its element.

In the following block, we develop the operation which sends x and $\{x, y\}$ to y . It is not immediately clear (except to common sense) that there *is* such an operation. This might be useful for Zermelo's implementation of equivalence, later in this file. I'm of two minds as to whether it will actually be useful, but it was an interesting exercise building the proofs and definitions.

```
begin Lestrade execution
```

```
>>> clearcurrent
```

```
{move 1}
```

```
>>> declare x obj
```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```

```

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> goal that One [z => (z E x ; y) & (z = x) == \
    y = x]

that One ([ (z : obj) =>
    ({def} (z E x ; y) & (z = x) ==
    y = x : prop)])

{move 1}

>>> goal that Forall [z => ((z E x ; y) & (z = x) == \
    y = x) == z = y]

that Forall ([ (z : obj) =>
    ({def} ((z E x ; y) & (z = x) ==
    y = x) == z = y : prop)])

{move 1}

>>> open

{move 2}

>>> declare z1 obj

```

```
z1 : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare z2 obj
```

```
z2 : obj
```

```
{move 3}
```

```
>>> declare dir1 that (z1 E x ; y) & (z1 \
    = x) == y = x
```

```
dir1 : that (z1 E x ; y) & (z1
    = x) == y = x
```

```
{move 3}
```

```
>>> declare dir2 that z1 = y
```

```
dir2 : that z1 = y
```

```
{move 3}
```



```

>>> define line1 dir1 : Iff1 Simp1 \
    dir1, Ui z1, Pair x y

line1 : [(dir1_1 : that (z1 E x ; y) & (z1
    = x) == y = x) => (--- : that
    (z1 = x) V z1 = y)]

{move 2}

>>> open

{move 4}

>>> declare case1 that z1 = x

case1 : that z1 = x

{move 4}

>>> define line2 case1 : Iff1 \
    case1 Simp2 dir1

line2 : [(case1_1 : that z1
    = x) => (--- : that y = x)]

{move 3}

>>> define line3 case1 : Subs1 \
    Eqsymm line2 case1 case1

```

```

line3 : [(case1_1 : that z1
          = x) => (--- : that z1 = y)]

{move 3}

>>> declare case2 that z1 = y

case2 : that z1 = y

{move 4}

>>> define line4 case2 : case2

line4 : [(case2_1 : that z1
          = y) => (--- : that z1 = y)]

{move 3}

>>> close

{move 3}

>>> define line5 dir1 : Cases line1 \
      dir1 line3, line4

line5 : [(dir1_1 : that (z1 E x ; y) & (z1
          = x) == y = x) => (--- : that
          z1 = y)]

{move 2}

```

```

>>> define line6 : Conj Inpair2 \
      x y, Iffrefl (y = x)

line6 : that (y E x ; y) & (y = x) ==
y = x

{move 2}

>>> define line7 dir2 : Subs Eqsymm \
      dir2 [z2 => (z2 E x ; y) & (z2 \
        = x) == y = x] line6

line7 : [(dir2_1 : that z1 = y) =>
  (--- : that (z1 E x ; y) & (z1
    = x) == y = x)]

{move 2}

>>> close

{move 2}

>>> define line8 z1 : Dediff line5, line7

line8 : [(z1_1 : obj) => (--- : that
  ((z1_1 E x ; y) & (z1_1 = x) ==
    y = x) == z1_1 = y)]

{move 1}

```

```
>>> close
```

```
{move 1}
```

```
>>> define Theother1 x y : Ug line8
```

```
Theother1 : [(x_1 : obj), (y_1 : obj) =>
  ({def} Ug [(z1_2 : obj) =>
    ({def} Dediff [(dir1_3 : that
      (z1_2 E x_1 ; y_1) & (z1_2
        = x_1) == y_1 = x_1) =>
      ({def} Cases (Simp1 (dir1_3) Iff1
        z1_2 Ui x_1 Pair y_1, [(case1_4
          : that z1_2 = x_1) =>
          ({def} Eqsymm (case1_4 Iff1
            Simp2 (dir1_3)) Subs1 case1_4
              : that z1_2 = y_1)], [(case2_4
                : that z1_2 = y_1) =>
                ({def} case2_4 : that z1_2
                  = y_1)]) : that z1_2 = y_1)], [(dir2_3
                  : that z1_2 = y_1) =>
                ({def} Subs (Eqsymm (dir2_3), [(z2_4
                  : obj) =>
                  ({def} (z2_4 E x_1 ; y_1) & (z2_4
                    = x_1) == y_1 = x_1 : prop)], (x_1
                    Inpair2 y_1) Conj Iffrefl (y_1
                    = x_1)) : that (z1_2 E x_1
                    ; y_1) & (z1_2 = x_1) == y_1
                    = x_1)]) : that ((z1_2 E x_1
                    ; y_1) & (z1_2 = x_1) == y_1
                    = x_1) == z1_2 = y_1)]) : that
  Forall [(x''_2 : obj) =>
    ({def} ((x''_2 E x_1 ; y_1) & (x''_2
      = x_1) == y_1 = x_1) == x''_2
      = y_1 : prop)]))]
```

```

Theother1 : [(x_1 : obj), (y_1 : obj) =>
  (--- : that Forall ([(x''_2 : obj) =>
    ({def} ((x''_2 E x_1 ; y_1) & (x''_2
      = x_1) == y_1 = x_1) == x''_2
      = y_1 : prop)))]])

{move 0}

>>> declare w obj

w : obj

{move 1}

>>> define Theother2 x y : Fixform One \
  [z => ((z E x ; y) & (z = x) == \
    y = x)], Ei y, [w => Forall [z => \
    ((z E x ; y) & (z = x) == y = x) == \
    z = w]], Theother1 x y

Theother2 : [(x_1 : obj), (y_1 : obj) =>
  ({def} One ([(z_3 : obj) =>
    ({def} (z_3 E x_1 ; y_1) & (z_3
      = x_1) == y_1 = x_1 : prop)))] Fixform
  Ei (y_1, [(w_3 : obj) =>
    ({def} Forall ([(z_4 : obj) =>
      ({def} ((z_4 E x_1 ; y_1) & (z_4
        = x_1) == y_1 = x_1) == z_4
        = w_3 : prop))] : prop)], x_1
    Theother1 y_1) : that One ([(z_2
      : obj) =>
      ({def} (z_2 E x_1 ; y_1) & (z_2
        = x_1) == y_1 = x_1 : prop)))]])

```

```
Theother2 : [(x_1 : obj), (y_1 : obj) =>
  (--- : that One ([(z_2 : obj) =>
    ({def} (z_2 E x_1 ; y_1) & (z_2
      = x_1) == y_1 = x_1 : prop)])))]
```

```
{move 0}
```

```
>>> declare ispairev that z = x ; y
```

```
ispairev : that z = x ; y
```

```
{move 1}
```

```
>>> declare z1 obj
```

```
z1 : obj
```

```
{move 1}
```

```
>>> define Theother x ispairev : The (Theother2 \
  x y)
```

```
Theother : [(x_1 : obj), (.y_1 : obj), (.z_1
  : obj), (ispairev_1 : that .z_1
    = x_1 ; .y_1) =>
  ({def} The (x_1 Theother2 .y_1) : obj)]
```

```
Theother : [(x_1 : obj), (.y_1 : obj), (.z_1
  : obj), (ispairev_1 : that .z_1
```

```

      = x_1 ; .y_1) => (--- : obj)]

{move 0}

>>> open

{move 2}

>>> define it : Theother x ispairev

it : obj

{move 1}

>>> define line9 : Fixform ((it E x ; y) & (it \
      = x) == y = x, Theax (Theother2 \
      x y))

line9 : that (it E x ; y) & (it
      = x) == y = x

{move 1}

>>> define line10 : Iff1 Simp1 line9, Ui \
      it, Pair x y

line10 : that (it = x) V it = y

{move 1}

```

```

>>> open

{move 3}

>>> declare case1 that it = x

case1 : that it = x

{move 3}

>>> define line11 case1 : Iff1 case1 \
    Simp2 line9

line11 : [(case1_1 : that it = x) =>
    (--- : that y = x)]

{move 2}

>>> define line12 case1 : Subs1 \
    Eqsymm line11 case1 case1

line12 : [(case1_1 : that it = x) =>
    (--- : that it = y)]

{move 2}

>>> declare case2 that it = y

case2 : that it = y

```



```
{move 3}
```

```
>>> define line13 case2 : case2
```

```
line13 : [(case2_1 : that it = y) =>  
  (--- : that it = y)]
```

```
{move 2}
```

```
>>> close
```

```
{move 2}
```

```
>>> define line14 : Cases line10 line12, line13
```

```
line14 : that it = y
```

```
{move 1}
```

```
>>> close
```

```
{move 1}
```

```
>>> define Theother3 x ispairev : line14
```

```
Theother3 : [(x_1 : obj), (.y_1 : obj), (.z_1  
  : obj), (ispairev_1 : that .z_1  
  = x_1 ; .y_1) =>  
  ({def} Cases (Simp1 (((x_1 Theother  
  ispairev_1 E x_1 ; .y_1) & (x_1 Theother
```

```

ispairev_1 = x_1) == .y_1 = x_1) Fixform
Theax (x_1 Theother2 .y_1)) Iff1
x_1 Theother ispairev_1 Ui x_1 Pair
.y_1, [(case1_2 : that x_1 Theother
  ispairev_1 = x_1) =>
  ({def} Eqsymm (case1_2 Iff1 Simp2
    ((x_1 Theother ispairev_1 E x_1
      ; .y_1) & (x_1 Theother ispairev_1
        = x_1) == .y_1 = x_1) Fixform
    Theax (x_1 Theother2 .y_1))) Subs1
    case1_2 : that x_1 Theother ispairev_1
      = .y_1)], [(case2_2 : that
    x_1 Theother ispairev_1 = .y_1) =>
    ({def} case2_2 : that x_1 Theother
      ispairev_1 = .y_1)]) : that x_1
Theother ispairev_1 = .y_1)]

Theother3 : [(x_1 : obj), (.y_1 : obj), (.z_1
  : obj), (ispairev_1 : that .z_1
    = x_1 ; .y_1) => (--- : that x_1
      Theother ispairev_1 = .y_1)]

{move 0}

>>> define Theother4 x y : Theother3 x Refleq \
  (x ; y)

Theother4 : [(x_1 : obj), (y_1 : obj) =>
  ({def} x_1 Theother3 Refleq (x_1
    ; y_1) : that x_1 Theother Refleq
      (x_1 ; y_1) = y_1)]

Theother4 : [(x_1 : obj), (y_1 : obj) =>
  (--- : that x_1 Theother Refleq (x_1

```

```
; y_1) = y_1)]
```

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

Our aim in the next blocks of code is to characterize projections of the pair. x is the unique object which belongs to all elements of $x; y$. y is the unique object which belongs to exactly one element of $x; y$. These theorems allow us to prove that an ordered pair is determined by its projections.

```
begin Lestrade execution
```

```
>>> clearcurrent
```

```
{move 1}
```

```
>>> declare x obj
```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare z obj
```

```
z : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare ineq that z E x $ y
```

```
ineq : that z E x $ y
```

```
{move 3}
```

```
>>> open
```

```
{move 4}
```

```
>>> define line1 :  $\text{Ui } z \text{ (Pair } \backslash$   
x ; x (x ; y))
```

```
line1 : that (z E (x ; x) ; x ; y) ==  
(z = x ; x)  $\vee$  z = x ; y
```

```
{move 3}
```

```
>>> define line2 : Iff1 ineq \
```

```

line1

line2 : that (z = x ; x) V z = x ; y

{move 3}

>>> declare eqev1 that z = x ; x

eqev1 : that z = x ; x

{move 4}

>>> declare w obj

w : obj

{move 4}

>>> define dir1 eqev1 : Subs1 \
      (Eqsymm eqev1, Inusc2 x)

dir1 : [(eqev1_1 : that z = x ; x) =>
      (--- : that x E z)]

{move 3}

>>> declare eqev2 that z = x ; y

eqev2 : that z = x ; y

```

```

{move 4}

>>> define dir2 eqev2 : Subs1 \
      (Eqsymm eqev2, Inpair1 x y)

dir2 : [(eqev2_1 : that z = x ; y) =>
      (--- : that x E z)]

{move 3}

>>> define line3 : Cases line2 \
      dir1, dir2

line3 : that x E z

{move 3}

>>> close

{move 3}

>>> define scratch ineq : line3

scratch : [(ineq_1 : that z E x $ y) =>
      (--- : that x E z)]

{move 2}

>>> define scratch2 : Ded scratch

```

```

scratch2 : that (z E x $ y) ->
  x E z

{move 2}

>>> close

{move 2}

>>> define scratch3 z : scratch2

scratch3 : [(z_1 : obj) => (---
  : that (z_1 E x $ y) -> x E z_1)]

{move 1}

>>> close

{move 1}

>>> define Firstprojthm1 x y : Ug scratch3

Firstprojthm1 : [(x_1 : obj), (y_1
  : obj) =>
  ({def} Ug ([z_2 : obj) =>
    ({def} Ded ([inev_3 : that
      z_2 E x_1 $ y_1) =>
      ({def} Cases (inev_3 Iff1 z_2
        Ui (x_1 ; x_1) Pair x_1 ; y_1, [(eqev1_4
          : that z_2 = x_1 ; x_1) =>

```

```

      ({def} Eqsymm (eqev1_4) Subs1
      Inusc2 (x_1) : that x_1
      E z_2)], [(eqev2_4 : that
      z_2 = x_1 ; y_1) =>
      ({def} Eqsymm (eqev2_4) Subs1
      x_1 Inpair1 y_1 : that x_1
      E z_2)]) : that x_1 E z_2)]) : that
      (z_2 E x_1 $ y_1) -> x_1 E z_2)]) : that
Forall ([ (x''_2 : obj) =>
      ({def} (x''_2 E x_1 $ y_1) ->
      x_1 E x''_2 : prop))]))]

```

```

Firstprojthm1 : [(x_1 : obj), (y_1
: obj) => (--- : that Forall ([ (x''_2
: obj) =>
      ({def} (x''_2 E x_1 $ y_1) ->
      x_1 E x''_2 : prop))]))]

```

```
{move 0}
```

```
>>> clearcurrent
```

```
{move 1}
```

```
>>> declare x obj
```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```



```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare w obj
```

```
w : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare z obj
```

```
z : obj
```

```
{move 3}
```

```
>>> declare firstev that Forall \  
    [z => (z E x $ y) -> w E z]
```

```
firstev : that Forall ([ (z_2  
    : obj) =>  
    ({def} (z_2 E x $ y) -> w E z_2  
    : prop)])
```

{move 3}

```
>>> define line1 firstev : Ui (Usc \
      x, firstev)
```

```
line1 : [(firstev_1 : that Forall
  ([(z_3 : obj) =>
    ({def} (z_3 E x $ y) ->
      w E z_3 : prop)))] =>
  (--- : that (Usc (x) E x $ y) ->
    w E Usc (x))]
```

{move 2}

```
>>> define line2 firstev : Fixform \
      ((Usc x) E x $ y, Inpair1 (x ; x, x ; y))
```

```
line2 : [(firstev_1 : that Forall
  ([(z_3 : obj) =>
    ({def} (z_3 E x $ y) ->
      w E z_3 : prop)))] =>
  (--- : that Usc (x) E x $ y)]
```

{move 2}

```
>>> define line3 firstev : Mp (line2 \
      firstev, line1 firstev)
```

```
line3 : [(firstev_1 : that Forall
  ([(z_3 : obj) =>
    ({def} (z_3 E x $ y) ->
```

```

      w E z_3 : prop))))) =>
    (--- : that w E Usc (x))]
```

```
{move 2}
```

```
>>> define line4 firstev : Inusc1 \
      line3 firstev
```

```
line4 : [(firstev_1 : that Forall
  ([(z_3 : obj) =>
    ({def} (z_3 E x $ y) ->
      w E z_3 : prop)))])) =>
  (--- : that w = x)]
```

```
{move 2}
```

```
>>> close
```

```
{move 2}
```

```
>>> define line5 w : Ded line4
```

```
line5 : [(w_1 : obj) => (--- : that
  Forall ([(z_3 : obj) =>
    ({def} (z_3 E x $ y) -> w_1
      E z_3 : prop))] -> w_1 = x)]
```

```
{move 1}
```

```
>>> close
```

```
{move 1}
```

```
>>> define Firstprojthm2 x y : Ug line5
```

```
Firstprojthm2 : [(x_1 : obj), (y_1
: obj) =>
  ({def} Ug [(w_2 : obj) =>
    ({def} Ded [(firstev_3 : that
      Forall [(z_5 : obj) =>
        ({def} (z_5 E x_1 $ y_1) ->
          w_2 E z_5 : prop))]) =>
      ({def} Inusc1 (((Usc (x_1) E x_1
        $ y_1) Fixform (x_1 ; x_1) Inpair1
        x_1 ; y_1) Mp Usc (x_1) Ui
        firstev_3) : that w_2 = x_1)]) : that
      Forall [(z_4 : obj) =>
        ({def} (z_4 E x_1 $ y_1) ->
          w_2 E z_4 : prop))] -> w_2
      = x_1)]) : that Forall [(x''_2
: obj) =>
      ({def} Forall [(z_4 : obj) =>
        ({def} (z_4 E x_1 $ y_1) ->
          x''_2 E z_4 : prop))] -> x''_2
      = x_1 : prop)])])]
```

```
Firstprojthm2 : [(x_1 : obj), (y_1
: obj) => (--- : that Forall [(x''_2
: obj) =>
  ({def} Forall [(z_4 : obj) =>
    ({def} (z_4 E x_1 $ y_1) ->
      x''_2 E z_4 : prop))] -> x''_2
    = x_1 : prop)])])]
```

```
{move 0}
```

```
end Lestrade execution
```

At this point we have proved that x belongs to all (both) elements of (x, y) , and that any w which belongs to both elements of (x, y) is actually equal to x .

The corresponding result for y will be a bit harder. We first want to prove $(\exists!z : z \in (x, y) \wedge y \in z)$. Then we want to prove for any w that if $(\exists!z : z \in (x, y) \wedge w \in z)$, then $w = y$.

Expanding things a bit, for the first part we want to prove $(\exists z : (\forall w : w \in (x, y) \wedge y \in w) \leftrightarrow w = z)$.

To be exact, this w is $\{x, y\}$, so we want to prove $(\forall w : (w \in (x, y) \wedge y \in w) \leftrightarrow w = \{x, y\})$.

```
begin Lestrade execution
```

```
>>> clearcurrent
```

```
{move 1}
```

```
>>> declare x obj
```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare w obj
```

```
w : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare yinitinpairev that (w E x $ y) & y E w
```

```
yinitinpairev : that (w E x $ y) & y E w
```

```
{move 3}
```

```
>>> open
```

```
{move 4}
```

```
>>> define line1 : Simp1 yinitinpairev
```

```
line1 : that w E x $ y
```

```
{move 3}
```

```
>>> define line2 : Ui (w, Pair \  
  (x ; x, x ; y))
```

```
line2 : that (w E (x ; x) ; x ; y) ==
  (w = x ; x) V w = x ; y
```

```
{move 3}
```

```
>>> open
```

```
{move 5}
```

```
>>> declare casehyp1 that \
      w = x ; x
```

```
casehyp1 : that w = x ; x
```

```
{move 5}
```

```
>>> define line3 casehyp1 \
      : Subs1 (casehyp1, Simp2 \
      yinitinpairev)
```

```
line3 : [(casehyp1_1 : that
      w = x ; x) => (--- : that
      y E x ; x)]
```

```
{move 4}
```

```
>>> define line4 casehyp1 \
      : Inusc1 line3 casehyp1
```

```
line4 : [(casehyp1_1 : that
```

```

      w = x ; x) => (--- : that
      y = x)]

```

```

{move 4}

```

```

>>> declare q obj

```

```

q : obj

```

```

{move 5}

```

```

>>> define dir1 casehyp1 : Subs \
      (Eqsymm line4 casehyp1, [q => \
      w = x ; q], casehyp1)

```

```

dir1 : [(casehyp1_1 : that
      w = x ; x) => (--- : that
      w = x ; y)]

```

```

{move 4}

```

```

>>> declare casehyp2 that \
      w = x ; y

```

```

casehyp2 : that w = x ; y

```

```

{move 5}

```

```

>>> define dir2 casehyp2 : casehyp2

```



```

dir2 : [(casehyp2_1 : that
      w = x ; y) => (--- : that
      w = x ; y)]

```

```

{move 4}

```

```

>>> close

```

```

{move 4}

```

```

>>> define line5 : Iff1 line1 \
      line2

```

```

line5 : that (w = x ; x) V w = x ; y

```

```

{move 3}

```

```

>>> define line6 : Cases line5 \
      dir1, dir2

```

```

line6 : that w = x ; y

```

```

{move 3}

```

```

>>> close

```

```

{move 3}

```

```

>>> define Line6 yinitinpairev : line6

```

```

Line6 : [(yinitinpairev_1 : that
        (w E x $ y) & y E w) => (---
        : that w = x ; y)]

{move 2}

>>> declare isunorderedxy that w = x ; y

isunorderedxy : that w = x ; y

{move 3}

>>> declare q obj

q : obj

{move 3}

>>> define Line7 isunorderedxy : Subs \
      (Eqsymm isunorderedxy, [q => \
        (q E x $ y) & y E q], Conj \
        (Inpair2 (x ; x, x ; y), Inpair2 \
        x y))

Line7 : [(isunorderedxy_1 : that
        w = x ; y) => (--- : that (w E x $ y) & y E w)]

{move 2}

>>> close

```

```
{move 2}
```

```
>>> define line8 w : Dediff Line6, Line7
```

```
line8 : [(w_1 : obj) => (--- : that  
  ((w_1 E x $ y) & y E w_1) ==  
  w_1 = x ; y)]
```

```
{move 1}
```

```
>>> define line9 : Ug line8
```

```
line9 : that Forall ([(x''_2 : obj) =>  
  ({def} ((x''_2 E x $ y) & y E x''_2) ==  
  x''_2 = x ; y : prop)])
```

```
{move 1}
```

```
>>> declare q obj
```

```
q : obj
```

```
{move 2}
```

```
>>> define line10 : Fixform (One [q => \  
  (q E x $ y) & y E q], Ei1 (x ; y, line9))
```

```
line10 : that One ([(q_2 : obj) =>  
  ({def} (q_2 E x $ y) & y E q_2  
  : prop)])
```

```
{move 1}
```

```
>>> close
```

```
{move 1}
```

```
>>> define Secondprojthm1 x y : line10
```

```
Secondprojthm1 : [(x_1 : obj), (y_1  
  : obj) =>  
  ({def} One ([(q_3 : obj) =>  
    ({def} (q_3 E x_1 $ y_1) & y_1  
      E q_3 : prop)]) Fixform (x_1  
; y_1) Ei1 Ug ([(w_4 : obj) =>  
  ({def} Dediff ([(yinitinpairev_5  
    : that (w_4 E x_1 $ y_1) & y_1  
      E w_4) =>  
    ({def} Cases (Simp1 (yinitinpairev_5) Iff1  
      w_4 Ui (x_1 ; x_1) Pair x_1  
; y_1, [(casehyp1_6 : that  
  w_4 = x_1 ; x_1) =>  
    ({def} Subs (Eqsymm (Inusc1  
      (casehyp1_6 Subs1 Simp2 (yinitinpairev_5))), [(q_7  
        : obj) =>  
        ({def} w_4 = x_1 ; q_7  
          : prop)], casehyp1_6) : that  
      w_4 = x_1 ; y_1)], [(casehyp2_6  
        : that w_4 = x_1 ; y_1) =>  
        ({def} casehyp2_6 : that  
          w_4 = x_1 ; y_1)]) : that  
      w_4 = x_1 ; y_1)], [(isunorderedxy_5  
        : that w_4 = x_1 ; y_1) =>  
        ({def} Subs (Eqsymm (isunorderedxy_5), [(q_6  
          : obj) =>
```

```

      ({def} (q_6 E x_1 $ y_1) & y_1
        E q_6 : prop)], ((x_1
      ; x_1) Inpair2 x_1 ; y_1) Conj
      x_1 Inpair2 y_1) : that (w_4
      E x_1 $ y_1) & y_1 E w_4)]) : that
      ((w_4 E x_1 $ y_1) & y_1 E w_4) ==
      w_4 = x_1 ; y_1)]) : that One
      ((q_2 : obj) =>
      ({def} (q_2 E x_1 $ y_1) & y_1
      E q_2 : prop]]))]]

```

```

Secondprojthm1 : [(x_1 : obj), (y_1
: obj) => (--- : that One ((q_2
: obj) =>
      ({def} (q_2 E x_1 $ y_1) & y_1
      E q_2 : prop]]))]]

```

```

{move 0}
end Lestrade execution

```

We report that our text plan given just before the block of Lestrade code worked exactly to plan the proof. We still have the second part, to show that for any w that if $(\exists! z : z \in (x, y) \wedge w \in z)$, then $w = y$.

We used environment nesting carefully to avoid declaring anything in move 0 in this block other than `Secondprojthm1`.

```

begin Lestrade execution

```

```

>>> clearcurrent

```

```

{move 1}

```

```

>>> declare x obj

```

```

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> declare w obj

w : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> declare secondprojev that One [z => \
      (z E x $ y) & w E z]

secondprojev : that One ([ (z_2 : obj) =>
      ({def} (z_2 E x $ y) & w E z_2 : prop)])

{move 1}

```

```

>>> open

{move 2}

>>> declare u obj

u : obj

{move 2}

>>> declare wev that Witnesses secondprojev \
    u

wev : that secondprojev Witnesses u

{move 2}

>>> open

{move 3}

>>> define fact1 : Ui (u, wev)

fact1 : that ((u E x $ y) & w E u) ==
    u = u

{move 2}

>>> define fact2 : Iff2 (Refleq \

```

```

    u, fact1)

fact2 : that (u E x $ y) & w E u

{move 2}

>>> define fact3 : Simp1 fact2

fact3 : that u E x $ y

{move 2}

>>> define fact4 : Simp2 fact2

fact4 : that w E u

{move 2}

>>> define fact5 : Ui u ((x ; x) Pair \
    (x ; y))

fact5 : that (u E (x ; x) ; x ; y) ==
    (u = x ; x) V u = x ; y

{move 2}

>>> define fact6 : Iff1 fact3 fact5

fact6 : that (u = x ; x) V u = x ; y

```



```
{move 2}
```

```
>>> open
```

```
{move 4}
```

```
>>> declare casehyp1 that u = x ; x
```

```
casehyp1 : that u = x ; x
```

```
{move 4}
```

```
>>> declare casehyp2 that u = x ; y
```

```
casehyp2 : that u = x ; y
```

```
{move 4}
```

```
>>> define line1 casehyp1 : Inusc1 \  
      (Subs1 casehyp1 fact4)
```

```
line1 : [(casehyp1_1 : that  
          u = x ; x) => (--- : that  
          w = x)]
```

```
{move 3}
```

```
>>> define fact7 : Ui (x ; y, wev)
```

```
fact7 : that (((x ; y) E x $ y) & w E x ; y) ==
  (x ; y) = u
```

```
{move 3}
```

```
>>> define line2 casehyp1 : Subs1 \
  (line1 casehyp1, fact7)
```

```
line2 : [(casehyp1_1 : that
  u = x ; x) => (--- : that
  (((x ; y) E x $ y) & x E x ; y) ==
  (x ; y) = u)]
```

```
{move 3}
```

```
>>> define line3 casehyp1 : Subs1 \
  (casehyp1, line2 casehyp1)
```

```
line3 : [(casehyp1_1 : that
  u = x ; x) => (--- : that
  (((x ; y) E x $ y) & x E x ; y) ==
  (x ; y) = x ; x)]
```

```
{move 3}
```

```
>>> define line4 casehyp1 : Iff1 \
  (Conj (Inpair2 (x ; x, x ; y), Inpair1 \
  (x, y)), line3 casehyp1)
```

```
line4 : [(casehyp1_1 : that
  u = x ; x) => (--- : that
```

```

(x ; y) = x ; x)]

{move 3}

>>> define line5 casehyp1 : Inusc1 \
      (Subs1 (line4 casehyp1, Inpair2 \
x y))

line5 : [(casehyp1_1 : that
u = x ; x) => (--- : that
y = x)]

{move 3}

>>> define line6 casehyp1 : Subs1 \
      (Eqsymm line5 casehyp1, line1 \
casehyp1)

line6 : [(casehyp1_1 : that
u = x ; x) => (--- : that
w = y)]

{move 3}

>>> define line7 casehyp2 : (Subs1 \
casehyp2 fact4)

line7 : [(casehyp2_1 : that
u = x ; y) => (--- : that
w E x ; y)]

```

```
{move 3}
```

```
>>> define line8 casehyp2 : Iff1 \  
      (line7 casehyp2, Ui w (x Pair \  
      y))
```

```
line8 : [(casehyp2_1 : that  
      u = x ; y) => (--- : that  
      (w = x) V w = y)]
```

```
{move 3}
```

```
>>> open
```

```
{move 5}
```

```
>>> declare case1 that w = x
```

```
case1 : that w = x
```

```
{move 5}
```

```
>>> declare case2 that w = y
```

```
case2 : that w = y
```

```
{move 5}
```

```
>>> define dir2 case2 : case2
```

```

dir2 : [(case2_1 : that
  w = y) => (--- : that
  w = y)]

{move 4}

>>> define fact8 : Ui (x ; x, wev)

fact8 : that (((x ; x) E x $ y) & w E x ; x) ==
  (x ; x) = u

{move 4}

>>> define line9 case1 : Subs1 \
  (casehyp2, Subs1 (case1, fact8))

line9 : [(case1_1 : that
  w = x) => (--- : that
  (((x ; x) E x $ y) & x E x ; x) ==
  (x ; x) = x ; y)]

{move 4}

>>> define line10 case1 : Iff1 \
  (Conj (Inpair1 (x ; x, x ; y), Inusc2 \
  x), line9 case1)

line10 : [(case1_1 : that
  w = x) => (--- : that
  (x ; x) = x ; y)]

```

```
{move 4}
```

```
>>> define line11 case1 : Inusc1 \  
      (Subs1 (Eqsymm (line10 \  
        case1), Inpair2 x y))
```

```
line11 : [(case1_1 : that  
  w = x) => (--- : that  
  y = x)]
```

```
{move 4}
```

```
>>> define dir1 case1 : Subs1 \  
      (Eqsymm line11 case1, case1)
```

```
dir1 : [(case1_1 : that  
  w = x) => (--- : that  
  w = y)]
```

```
{move 4}
```

```
>>> close
```

```
{move 4}
```

```
>>> define line13 casehyp2 : Cases \  
      (line8 casehyp2, dir1, dir2)
```

```
line13 : [(casehyp2_1 : that  
  u = x ; y) => (--- : that  
  w = y)]
```

```

      {move 3}

    >>> close

    {move 3}

    >>> define line14 : Cases (fact6, line6, line13)

    line14 : that w = y

    {move 2}

    >>> close

    {move 2}

    >>> define line15 u wev : line14

    line15 : [(u_1 : obj), (wev_1
      : that secondprojev Witnesses u_1) =>
      (--- : that w = y)]

    {move 1}

    >>> define line16 : Eg (secondprojev, line15)

    line16 : that w = y

    {move 1}

```

```

>>> close

{move 1}

>>> define Secondprojthm2 x y w secondprojev \
      : line16

Secondprojthm2 : [(x_1 : obj), (y_1
  : obj), (w_1 : obj), (secondprojev_1
  : that One ([(z_3 : obj) =>
    ({def} (z_3 E x_1 $ y_1) & w_1
      E z_3 : prop])) =>
  ({def} secondprojev_1 Eg [(u_2 : obj), (wev_2
    : that secondprojev_1 Witnesses
    u_2) =>
    ({def} Cases (Simp1 (Refleq (u_2) Iff2
      u_2 Ui wev_2) Iff1 u_2 Ui (x_1
      ; x_1) Pair x_1 ; y_1, [(casehyp1_3
        : that u_2 = x_1 ; x_1) =>
        ({def} Eqsymm (Inusc1 (((x_1
          ; x_1) Inpair2 x_1 ; y_1) Conj
          x_1 Inpair1 y_1 Iff1 casehyp1_3
          Subs1 Inusc1 (casehyp1_3 Subs1
          Simp2 (Refleq (u_2) Iff2 u_2
          Ui wev_2)) Subs1 (x_1 ; y_1) Ui
          wev_2 Subs1 x_1 Inpair2 y_1)) Subs1
          Inusc1 (casehyp1_3 Subs1 Simp2
          (Refleq (u_2) Iff2 u_2 Ui
          wev_2)) : that w_1 = y_1)], [(casehyp2_3
          : that u_2 = x_1 ; y_1) =>
          ({def} Cases (casehyp2_3 Subs1
          Simp2 (Refleq (u_2) Iff2 u_2
          Ui wev_2) Iff1 w_1 Ui x_1 Pair
          y_1, [(case1_4 : that w_1
            = x_1) =>

```



```

      ({def} Eqsymm (Inusc1 (Eqsymm
      (((x_1 ; x_1) Inpair1
      x_1 ; y_1) Conj Inusc2 (x_1) Iff1
      casehyp2_3 Subs1 case1_4 Subs1
      (x_1 ; x_1) Ui wev_2) Subs1
      x_1 Inpair2 y_1)) Subs1
      case1_4 : that w_1 = y_1)], [(case2_4
      : that w_1 = y_1) =>
      ({def} case2_4 : that w_1
      = y_1)]) : that w_1 = y_1)]) : that
      w_1 = y_1)] : that w_1 = y_1)]

```

```

Secondprojthm2 : [(x_1 : obj), (y_1
: obj), (w_1 : obj), (secondprojev_1
: that One ([(z_3 : obj) =>
      ({def} (z_3 E x_1 $ y_1) & w_1
      E z_3 : prop)])) => (--- : that
      w_1 = y_1)]

```

```

      {move 0}
end Lestrade execution

```

This completes the proof of the characterizations of first and second projections. Now we prove that pairs are characterized exactly by their projections. It is worth noting that the size of the Lestrade proof is more accurately determined if one ignores Lestrade's responses in the dialogue and considers only the input lines. Another alternative would be to consider the size of the Lestrade terms saved at move 0. We are currently generating this text with a setting in the prover which suppresses display of proof terms (and more generally of the definitions of defined terms) except at move 0. At move 0, displayed proof terms/definitions can be quite large because all definitions at higher indexed moves are expanded out.

```

begin Lestrade execution

```

```

>>> clearcurrent

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> declare w obj

w : obj

{move 1}

>>> declare paireqev that (x $ y) = z $ w

```

```
paireqev : that (x $ y) = z $ w
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> define line1 : Firstprojthm1 x y
```

```
line1 : that Forall ([(x''_2 : obj) =>
  ({def} (x''_2 E x $ y) -> x E x''_2
  : prop)])
```

```
{move 1}
```

```
>>> define line2 : Subs1 paireqev line1
```

```
line2 : that Forall ([(x''_2 : obj) =>
  ({def} (x''_2 E z $ w) -> x E x''_2
  : prop)])
```

```
{move 1}
```

```
>>> define line3 : Firstprojthm2 z w
```

```
line3 : that Forall ([(x''_2 : obj) =>
  ({def} Forall ([(z_4 : obj) =>
    ({def} (z_4 E z $ w) -> x''_2
    E z_4 : prop)]) -> x''_2 = z : prop)])
```

```
{move 1}
```

```
>>> define line4 : Ui x line3
```

```
line4 : that Forall ([(z_3 : obj) =>
  ({def} (z_3 E z $ w) -> x E z_3
  : prop)]) -> x = z
```

```
{move 1}
```

```
>>> define line5 : Mp line2 line4
```

```
line5 : that x = z
```

```
{move 1}
```

```
>>> define line6 : Secondprojthm1 x y
```

```
line6 : that One ([(q_2 : obj) =>
  ({def} (q_2 E x $ y) & y E q_2
  : prop)])
```

```
{move 1}
```

```
>>> define line7 : Subs1 paireqev line6
```

```
line7 : that One ([(q_2 : obj) =>
  ({def} (q_2 E z $ w) & y E q_2
  : prop)])
```

```

{move 1}

>>> define line8 : Secondprojthm2 z w y line7

line8 : that y = w

{move 1}

>>> close

{move 1}

>>> define Pairseq paireqev : Conj (line5, line8)

Pairseq : [(x_1 : obj), (y_1 : obj), (z_1
: obj), (w_1 : obj), (paireqev_1
: that (x_1 $ y_1) = z_1 $ w_1) =>
({def} paireqev_1 Subs1 x_1 Firstprojthm1
y_1 Mp x_1 Ui z_1 Firstprojthm2
w_1 Conj Secondprojthm2 (z_1, w_1, y_1, paireqev_1
Subs1 x_1 Secondprojthm1 y_1) : that
(x_1 = z_1) & y_1 = w_1)]

Pairseq : [(x_1 : obj), (y_1 : obj), (z_1
: obj), (w_1 : obj), (paireqev_1
: that (x_1 $ y_1) = z_1 $ w_1) =>
(--- : that (x_1 = z_1) & y_1
= w_1)]

{move 0}

```

`end Lestrade execution`

The details of the implementation of the ordered pair take up quite a lot of space but it is an important feature of the system.

It is very interesting to observe that a definition of the pair local to the collection of relations from a given set to a given other set appears to be implicit in Zermelo's definition of correspondences; I'll be explicit about this in constructions to appear below in this document, when I add them.

`begin Lestrade execution`

`>>> declare s obj`

`s : obj`

`{move 1}`

`>>> declare t obj`

`t : obj`

`{move 1}`

`>>> declare u obj`

`u : obj`

`{move 1}`

`>>> open`

```

{move 2}

>>> declare dir1 that (s ; t) <=& \
    u

dir1 : that (s ; t) <=& u

{move 2}

>>> define lineal dir1 : Conj Mp Inpair1 \
    s t, Ui s Simp1 dir1, Mp Inpair2 \
    s t, Ui t Simp1 dir1

lineal : [(dir1_1 : that (s ; t) <=&
    u) => (--- : that (s E u) & t E u)]

{move 1}

>>> declare dir2 that (s E u) & t E u

dir2 : that (s E u) & t E u

{move 2}

>>> open

{move 3}

>>> declare x1 obj

```

```
x1 : obj
```

```
{move 3}
```

```
>>> open
```

```
{move 4}
```

```
>>> declare xev1 that x1 E s ; t
```

```
xev1 : that x1 E s ; t
```

```
{move 4}
```

```
>>> define linebb2 xev1 : Iff1 \  
      xev1, Ui x1, Pair s t
```

```
linebb2 : [(xev1_1 : that x1  
            E s ; t) => (--- : that  
            (x1 = s) V x1 = t)]
```

```
{move 3}
```

```
>>> open
```

```
{move 5}
```

```
>>> declare case1 that x1 \  
      = s
```



```

case1 : that x1 = s

{move 5}

>>> define linebb3 case1 : Subs1 \
      (Eqsymm case1, Simp1 dir2)

linebb3 : [(case1_1 : that
            x1 = s) => (--- : that
            x1 E u)]

{move 4}

>>> declare case2 that x1 \
      = t

case2 : that x1 = t

{move 5}

>>> define linea4 case2 : Subs1 \
      (Eqsymm case2, Simp2 dir2)

linea4 : [(case2_1 : that
            x1 = t) => (--- : that
            x1 E u)]

{move 4}

>>> close

```

```

{move 4}

>>> define linea5 xev1 : Cases \
      linebb2 xev1, linebb3, linea4

linea5 : [(xev1_1 : that x1
      E s ; t) => (--- : that
      x1 E u)]

{move 3}

>>> close

{move 3}

>>> define linea6 x1 : Ded linea5

linea6 : [(x1_1 : obj) => (---
      : that (x1_1 E s ; t) -> x1_1
      E u)]

{move 2}

>>> close

{move 2}

>>> define linebb7 dir2 : Fixform ((s ; t) <=& \
      u, Conj (Ug linea6, Conj (Inhabited \
      Inpair1 s t, Inhabited Simp1 dir2)))

```

```

linebb7 : [(dir2_1 : that (s E u) & t E u) =>
  (--- : that (s ; t) <=< u)]

{move 1}

>>> close

{move 1}

>>> define Pairsubs s t u : Dediff linea1, linebb7

Pairsubs : [(s_1 : obj), (t_1 : obj), (u_1
: obj) =>
  ({def} Dediff ([ (dir1_2 : that
    (s_1 ; t_1) <=< u_1) =>
    ({def} (s_1 Inpair1 t_1) Mp s_1
    Ui Simp1 (dir1_2) Conj (s_1 Inpair2
    t_1) Mp t_1 Ui Simp1 (dir1_2) : that
    (s_1 E u_1) & t_1 E u_1)], [(dir2_2
    : that (s_1 E u_1) & t_1 E u_1) =>
    ({def} ((s_1 ; t_1) <=< u_1) Fixform
    Ug [(x1_5 : obj) =>
      ({def} Ded ([ (xev1_6 : that
        x1_5 E s_1 ; t_1) =>
        ({def} Cases (xev1_6 Iff1
        x1_5 Ui s_1 Pair t_1, [(case1_7
        : that x1_5 = s_1) =>
        ({def} Eqsymm (case1_7) Subs1
        Simp1 (dir2_2) : that
        x1_5 E u_1)], [(case2_7
        : that x1_5 = t_1) =>
        ({def} Eqsymm (case2_7) Subs1
        Simp2 (dir2_2) : that

```

```

      x1_5 E u_1])) : that
      x1_5 E u_1])) : that (x1_5
      E s_1 ; t_1) -> x1_5 E u_1])) Conj
      Inhabited (s_1 Inpair1 t_1) Conj
      Inhabited (Simp1 (dir2_2)) : that
      (s_1 ; t_1) <=<= u_1])) : that
      ((s_1 ; t_1) <=<= u_1) == (s_1
      E u_1) & t_1 E u_1)]

```

```

Pairsubs : [(s_1 : obj), (t_1 : obj), (u_1
: obj) => (--- : that ((s_1 ; t_1) <=<=
u_1) == (s_1 E u_1) & t_1 E u_1)]

```

```

{move 0}

```

```

>>> open

```

```

{move 2}

```

```

>>> declare dir1 that Usc s <=<= t

```

```

dir1 : that Usc (s) <=<= t

```

```

{move 2}

```

```

>>> define lineas8 dir1 : Simp1 (Iff1 \
dir1, Pairsubs s s t)

```

```

lineas8 : [(dir1_1 : that Usc (s) <=<=
t) => (--- : that s E t)]

```

```

{move 1}

>>> declare dir2 that s E t

dir2 : that s E t

{move 2}

>>> define linea9 dir2 : Fixform (Usc \
    s <= t, Iff2 (Conj dir2 dir2, Pairsubs \
    s s t))

linea9 : [(dir2_1 : that s E t) =>
    (--- : that Usc (s) <= t)]

{move 1}

>>> close

{move 1}

>>> define Uscsubs s t : Dediff line8, line9

Uscsubs : [(s_1 : obj), (t_1 : obj) =>
    ({def} Dediff ([dir1_2 : that
        Usc (s_1) <= t_1) =>
        ({def} Simp1 (dir1_2 Iff1 Pairsubs
            (s_1, s_1, t_1)) : that s_1
            E t_1)], [dir2_2 : that s_1
            E t_1) =>
        ({def} (Usc (s_1) <= t_1) Fixform
            dir2_2 Conj dir2_2 Iff2 Pairsubs

```

```

      (s_1, s_1, t_1) : that Usc (s_1) <=<=
      t_1])) : that (Usc (s_1) <=<=
t_1) == s_1 E t_1)]

Uscsubs : [(s_1 : obj), (t_1 : obj) =>
  (--- : that (Usc (s_1) <=<= t_1) ==
s_1 E t_1)]

{move 0}

>>> define Pairinhabited s t : Ei s, [u => \
  u E s ; t], Inpair1 s t

Pairinhabited : [(s_1 : obj), (t_1
  : obj) =>
  ({def} Ei (s_1, [(u_2 : obj) =>
    ({def} u_2 E s_1 ; t_1 : prop)], s_1
  Inpair1 t_1) : that Exists ([u_2
    : obj) =>
    ({def} u_2 E s_1 ; t_1 : prop)])))]

Pairinhabited : [(s_1 : obj), (t_1
  : obj) => (--- : that Exists ([u_2
    : obj) =>
    ({def} u_2 E s_1 ; t_1 : prop)])))]

{move 0}
end Lestrade execution

```

This is a batch of axioms relating unordered pairs and singletons to subset which were brought to my attention by the actual Zermelo development.

```

begin Lestrade execution

  >>> clearcurrent

{move 1}

  >>> declare x obj

x : obj

{move 1}

  >>> declare sethyp that Isset x

sethyp : that Isset (x)

{move 1}

  >>> open

  {move 2}

  >>> declare W obj

W : obj

{move 2}

  >>> open

```

```

{move 3}

>>> declare absurdhyp that W E 0

absurdhyp : that W E 0

{move 3}

>>> define line1 absurdhyp : Giveup \
      (W E x, Mp absurdhyp Empty W)

line1 : [(absurdhyp_1 : that W E 0) =>
      (--- : that W E x)]

{move 2}

>>> close

{move 2}

>>> define lineb2 W : Ded line1

lineb2 : [(W_1 : obj) => (--- : that
      (W_1 E 0) -> W_1 E x)]

{move 1}

>>> close

{move 1}

```



```
>>> define Zeroissubset sethyp : Fixform \
      (0 <= x, Conj (Ug lineb2, Conj (Zeroisset, sethyp)))
```

```
Zeroissubset : [(x_1 : obj), (sethyp_1
  : that Isset (x_1)) =>
  ({def} (0 <= x_1) Fixform Ug [(W_4
    : obj) =>
    ({def} Ded [(absurdhyp_5 : that
      W_4 E 0) =>
      ({def} (W_4 E x_1) Giveup
        absurdhyp_5 Mp Empty (W_4) : that
        W_4 E x_1)]) : that (W_4
        E 0) -> W_4 E x_1)]) Conj Zeroisset
    Conj sethyp_1 : that 0 <= x_1)]
```

```
Zeroissubset : [(x_1 : obj), (sethyp_1
  : that Isset (x_1)) => (--- : that
    0 <= x_1)]
```

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

The empty set is a subset of every set.

```
begin Lestrade execution
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 1}
```

```

>>> define <=/= x y : (x <= y) & x /= \
      y

<=/= : [(x_1 : obj), (y_1 : obj) =>
      ({def} (x_1 <= y_1) & x_1 /= y_1
      : prop)]

<=/= : [(x_1 : obj), (y_1 : obj) =>
      (--- : prop)]

{move 0}
end Lestrade execution

```

4 The axiom scheme of separation

We now develop the signature axiom scheme of Zermelo set theory, which may be thought of as its solution to the “paradoxes of naive set theory”. An arbitrary predicate of untyped objects can be converted to a set, if restricted to an already given set.

Our development follows the order in the axiomatics paper. In Zermelo’s treatment, this is the third axiom, after extensionality and the axiom of elementary sets (empty set, singleton, and pairing). Zermelo does assert that the object witnessing an instance of separation is a subset of the bounding set, and so a set: we merely provide an additional axiom that $\{x \in A : \phi(x)\}$, from which the assertion that it is a subset of A can be proved.

```

begin Lestrade execution

>>> clearcurrent

{move 1}

```

```

>>> declare A obj

A : obj

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> declare pred [x => prop]

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

{move 1}

>>> postulate Set A pred obj

Set : [(A_1 : obj), (pred_1 : [(x_2
      : obj) => (--- : prop)]) =>
      (--- : obj)]

{move 0}

>>> postulate Separation A pred that Forall \
      [x => (x E Set A pred) == (x E A) & pred \
      x]

```

```

Separation : [(A_1 : obj), (pred_1
  : [(x_2 : obj) => (--- : prop)]) =>
  (--- : that Forall ([(x_2 : obj) =>
    ({def} (x_2 E A_1 Set pred_1) ==
      (x_2 E A_1) & pred_1 (x_2) : prop)]))]

{move 0}
end Lestrade execution

```

We present the axiom of separation and the constructor implementing it. Like the deduction theorem, this is a constructor taking constructions to objects. Its argument of type $[x:obj \Rightarrow prop]$ is a general predicate of objects, and may be thought of as a proper class.

The fact that any property of objects however formulated generates a set when restricted to a previously given set implements Zermelo's intention. We do not thereby automatically find ourselves in a second order theory, because we have not provided ourselves with quantifiers over proper classes. We could declare quantifiers over proper classes easily enough, but we have not done so. It is worth noting that in Automath (at least in later versions) quantification over any type, including function types such as the type of predicates of sets we are considering here, is automatically provided: as soon as one axiomatizes Zermelo set theory along these lines in Automath, one has thereby axiomatized second order Zermelo set theory, which is a bit stronger. Weakness in a logical framework can be an advantage.

```

begin Lestrade execution

>>> postulate Separation2 A pred that \
      Isset (Set A pred)

Separation2 : [(A_1 : obj), (pred_1
  : [(x_2 : obj) => (--- : prop)]) =>
  (--- : that Isset (A_1 Set pred_1))]

```

```

    {move 0}
end Lestrade execution

```

We provide the additional axiom that $\{x \in A : \phi(x)\}$ is always a set (which is only relevant to empty extensions). Like `Scthm2`, this is implicit in Zermelo's statement of his axioms.

```

begin Lestrade execution

>>> declare sillyeq that x = Set A pred

sillyeq : that x = A Set pred

{move 1}

>>> define Separation3 sillyeq : Separation2 \
    A pred

Separation3 : [(A_1 : obj), (x_1
    : obj), (pred_1 : [(x_2 : obj) =>
        (--- : prop)]), (sillyeq_1
    : that .x_1 = .A_1 Set .pred_1) =>
    ({def} .A_1 Separation2 .pred_1 : that
    Isset (.A_1 Set .pred_1))]

Separation3 : [(A_1 : obj), (x_1
    : obj), (pred_1 : [(x_2 : obj) =>
        (--- : prop)]), (sillyeq_1
    : that .x_1 = .A_1 Set .pred_1) =>
    (--- : that Isset (.A_1 Set .pred_1))]

```

```

{move 0}
end Lestrade execution

```

This is a tricky “theorem” which allows the deduction that x is a set from the proof of $x = x$ if x happens to be ultimately defined using the separation constructor, without the user needing to specify the predicate defining the set. This is a diabolical perhaps unintended use of the implicit argument mechanism.

```

begin Lestrade execution

```

```

>>> define Separation4 sillyeq : Separation \
      A pred

```

```

Separation4 : [(A_1 : obj), (x_1
  : obj), (.pred_1 : [(x_2 : obj) =>
    (--- : prop)])], (sillyeq_1
  : that x_1 = A_1 Set .pred_1) =>
  ({def} A_1 Separation .pred_1 : that
    Forall ([(x_2 : obj) =>
      ({def} (x_2 E A_1 Set .pred_1) ==
        (x_2 E A_1) & .pred_1 (x_2) : prop)]))]

```

```

Separation4 : [(A_1 : obj), (x_1
  : obj), (.pred_1 : [(x_2 : obj) =>
    (--- : prop)])], (sillyeq_1
  : that x_1 = A_1 Set .pred_1) =>
  (--- : that Forall ([(x_2 : obj) =>
    ({def} (x_2 E A_1 Set .pred_1) ==
      (x_2 E A_1) & .pred_1 (x_2) : prop)]))]

```

```

{move 0}
end Lestrade execution

```

This is a tricky “theorem” which allows the instance of separation defining x to be extracted from the proof of $x = x$. This is a diabolical perhaps unintended use of the implicit argument mechanism.

```
begin Lestrade execution
```

```
>>> declare X7 obj
```

```
X7 : obj
```

```
{move 1}
```

```
>>> declare Y7 obj
```

```
Y7 : obj
```

```
{move 1}
```

```
>>> declare Z7 obj
```

```
Z7 : obj
```

```
{move 1}
```

```
>>> declare xinyev that X7 E Y7
```

```
xinyev : that X7 E Y7
```

```
{move 1}
```

```

>>> declare pred7 [Z7 => prop]

pred7 : [(Z7_1 : obj) => (--- : prop)]

{move 1}

>>> declare univev that Forall [Z7 => \
      (Z7 E Y7) -> pred7 Z7]

univev : that Forall ([ (Z7_2 : obj) =>
      ({def} (Z7_2 E Y7) -> pred7 (Z7_2) : prop)])

{move 1}

>>> define Univcheat xinyev univev : Mp \
      xinyev, Ui X7 univev

Univcheat : [(X7_1 : obj), (.Y7_1
      : obj), (xinyev_1 : that .X7_1 E .Y7_1), (.pred7_1
      : [(Z7_2 : obj) => (--- : prop)]), (univev_1
      : that Forall ([ (Z7_3 : obj) =>
      ({def} (Z7_3 E .Y7_1) -> .pred7_1
      (Z7_3) : prop)])) =>
      ({def} xinyev_1 Mp .X7_1 Ui univev_1
      : that .pred7_1 (.X7_1))]

Univcheat : [(X7_1 : obj), (.Y7_1
      : obj), (xinyev_1 : that .X7_1 E .Y7_1), (.pred7_1
      : [(Z7_2 : obj) => (--- : prop)]), (univev_1
      : that Forall ([ (Z7_3 : obj) =>
      ({def} (Z7_3 E .Y7_1) -> .pred7_1

```



```

      (Z7_3) : prop]])) => (---
: that .pred7_1 (.X7_1))]
```

```

{move 0}
end Lestrade execution
```

This is another implicit argument trick. From evidence for $x \in y$ and $(\forall z : z \in y \rightarrow \phi(z))$, get evidence for $\phi(x)$. The advantage is that the second parameter may be a complex defined notion which is only universal when expanded: the implicit argument mechanism handles the expansion without the user's attention being needed.

```

begin Lestrade execution
```

```

>>> declare ineq7 that X7 E Set Y7, pred7
```

```

ineq7 : that X7 E Y7 Set pred7
```

```

{move 1}
```

```

>>> define Separation5 ineq7 : Iff1 ineq7, Ui \
      X7, Separation4 Refleq Set Y7, pred7
```

```

Separation5 : [(X7_1 : obj), (Y7_1
: obj), (.pred7_1 : [(Z7_2 : obj) =>
      (--- : prop)]), (ineq7_1 : that
.X7_1 E .Y7_1 Set .pred7_1) =>
({def} ineq7_1 Iff1 .X7_1 Ui Separation4
(Refleq (.Y7_1 Set .pred7_1)) : that
(X7_1 E .Y7_1) & .pred7_1 (.X7_1))]
```

```

Separation5 : [(X7_1 : obj), (Y7_1
```

```

: obj), (.pred7_1 : [(Z7_2 : obj) =>
  (--- : prop)]), (inev7_1 : that
.X7_1 E .Y7_1 Set .pred7_1) => (---
: that (.X7_1 E .Y7_1) & .pred7_1
(.X7_1))]
```

```

{move 0}
end Lestrade execution
```

This is a tricky method to get a proof of $a \in A \wedge \phi(a)$ from a proof of $a \in \{x \mid \phi(x)\}$. The numbers attached to the various flavors of separation are arbitrary, basically in order of discovery of the need for them.

```

begin Lestrade execution

>>> declare y obj

y : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> declare Aisset that Isset A

Aisset : that Isset (A)
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare X obj
```

```
X : obj
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare Xinev that X E (Set \  
    A pred)
```

```
Xinev : that X E A Set pred
```

```
{move 3}
```

```
>>> define line1 Xinev : Simp1 Iff1 \  
    Xinev, Ui X, Separation A pred
```

```
line1 : [(Xinev_1 : that X E A Set  
    pred) => (--- : that X E A)]
```

```

{move 2}

>>> close

{move 2}

>>> define line2 X : Ded line1

line2 : [(X_1 : obj) => (--- : that
      (X_1 E A Set pred) -> X_1 E A)]

{move 1}

>>> close

{move 1}

>>> define Sepsub A pred, Aisset : Fixform \
      ((Set A pred) <=< A, Conj (Ug line2, Conj \
      (Separation2 A pred, Aisset)))

Sepsub : [(A_1 : obj), (pred_1 : [(x_2
      : obj) => (--- : prop)]), (Aisset_1
      : that Isset (A_1)) =>
      ({def} ((A_1 Set pred_1) <=< A_1) Fixform
      Ug ([X_4 : obj) =>
      ({def} Ded ([Xinev_5 : that
      X_4 E A_1 Set pred_1) =>
      ({def} Simp1 (Xinev_5 Iff1
      X_4 Ui A_1 Separation pred_1) : that
      X_4 E A_1)]) : that (X_4
      E A_1 Set pred_1) -> X_4 E A_1)]) Conj
      (A_1 Separation2 pred_1) Conj Aisset_1

```

```
      : that (A_1 Set pred_1) <=<= A_1)]
```

```
Sepsub : [(A_1 : obj), (pred_1 : [(x_2
      : obj) => (--- : prop)])], (Aisset_1
      : that Isset (A_1)) => (--- : that
      (A_1 Set pred_1) <=<= A_1)]
```

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

This uses the implicit argument mechanism to extract a proof that $\{x \in A : \phi(x)\}$ is a subset of A (if A is a set) from the proof that $\{x \in A : \phi(x)\}$ is equal to itself. The magic is that this works if the form used for $\{x : \phi(x)\}$ is a definition from which we do not want to extract the predicate.

```
begin Lestrade execution
```

```
>>> declare eqev that (Set A pred) = Set \
      A pred
```

```
eqev : that (A Set pred) = A Set pred
```

```
{move 1}
```

```
>>> define Sepsub2 Aisset eqev : Sepsub \
      A pred, Aisset
```

```
Sepsub2 : [(A_1 : obj), (.pred_1
      : [(x_2 : obj) => (--- : prop)])], (Aisset_1
      : that Isset (.A_1)), (eqev_1
      : that (.A_1 Set .pred_1) = .A_1
      Set .pred_1) =>
```

```

({def} Sepsub (.A_1, .pred_1, Aisset_1) : that
(.A_1 Set .pred_1) <=<= .A_1)]

```

```

Sepsub2 : [(A_1 : obj), (.pred_1
: [(x_2 : obj) => (--- : prop)]), (Aisset_1
: that Isset (.A_1)), (eqev_1
: that (.A_1 Set .pred_1) = .A_1
Set .pred_1) => (--- : that (.A_1
Set .pred_1) <=<= .A_1)]

```

```

{move 0}
end Lestrade execution

```

This uses the implicit argument mechanism to extract a proof that $\{x \in A : \phi(x)\}$ is a subset of A (if A is a set) from the proof that $\{x \in A : \phi(x)\}$ is equal to itself. The magic is that this works if the form used for $\{x : \phi(x)\}$ is a definition from which we do not want to extract the predicate.

```

begin Lestrade execution

```

```

>>> clearcurrent

```

```

{move 1}

```

```

>>> declare M obj

```

```

M : obj

```

```

{move 1}

```

```

>>> declare M1 obj

```

```

M1 : obj

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> define Complement M M1 : Set M [x => \
      ~ (x E M1)]

Complement : [(M_1 : obj), (M1_1
      : obj) =>
      ({def} M_1 Set [(x_2 : obj) =>
      ({def} ~ (x_2 E M1_1) : prop)] : obj)]

Complement : [(M_1 : obj), (M1_1
      : obj) => (--- : obj)]

{move 0}

>>> define Compax M M1 : Fixform (Forall \
      [x => (x E Complement M M1) == (x E M) & ~ (x E M1)], Separation \
      M [x => ~ (x E M1)])

Compax : [(M_1 : obj), (M1_1 : obj) =>
      ({def} Forall [(x_3 : obj) =>
      ({def} (x_3 E M_1 Complement M1_1) ==
      (x_3 E M_1) & ~ (x_3 E M1_1) : prop)]) Fixform

```

```

M_1 Separation [(x_3 : obj) =>
  ({def} ~ (x_3 E M1_1) : prop)] : that
Forall ([(x_2 : obj) =>
  ({def} (x_2 E M_1 Complement M1_1) ==
    (x_2 E M_1) & ~ (x_2 E M1_1) : prop)))]

Compax : [(M_1 : obj), (M1_1 : obj) =>
  (--- : that Forall ([(x_2 : obj) =>
    ({def} (x_2 E M_1 Complement M1_1) ==
      (x_2 E M_1) & ~ (x_2 E M1_1) : prop)))]

{move 0}
end Lestrade execution

```

Above we implement the relative complement and its defining axiom.

```

begin Lestrade execution

  >>> clearcurrent

{move 1}

  >>> declare x obj

  x : obj

  {move 1}

  >>> declare y obj

  y : obj

```



```

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> define ** x y : Set x [z => z E y]

** : [(x_1 : obj), (y_1 : obj) =>
      ({def} x_1 Set [(z_2 : obj) =>
                      ({def} z_2 E y_1 : prop)] : obj)]

** : [(x_1 : obj), (y_1 : obj) =>
      (--- : obj)]

{move 0}
end Lestrade execution

begin Lestrade execution

>>> clearcurrent

{move 1}

>>> declare T obj

T : obj

```

```

{move 1}

>>> declare A obj

A : obj

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> declare B obj

B : obj

{move 1}

>>> define Intersection T A : Set A [x => \
      Forall [B => (B E T) -> x E B]]

Intersection : [(T_1 : obj), (A_1
      : obj) =>
      ({def} A_1 Set [(x_2 : obj) =>
      ({def} Forall [(B_3 : obj) =>
      ({def} (B_3 E T_1) -> x_2
      E B_3 : prop)]) : prop)] : obj)]

```

```
Intersection : [(T_1 : obj), (A_1  
  : obj) => (--- : obj)]
```

```
{move 0}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare ineq that A E T
```

```
ineq : that A E T
```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare u obj
```

```
u : obj
```

```
{move 3}
```

```
>>> open
```

```
{move 4}
```

```
>>> declare hyp1 that u E Intersection \
      T A
```

```
hyp1 : that u E T Intersection
      A
```

```
{move 4}
```

```
>>> declare x1 obj
```

```
x1 : obj
```

```
{move 4}
```

```
>>> declare B1 obj
```

```
B1 : obj
```

```
{move 4}
```

```
>>> declare hyp2 that Forall \
      [B1 => (B1 E T) -> u E B1]
```

```
hyp2 : that Forall ([ (B1_2
      : obj) =>
      ({def} (B1_2 E T) -> u E B1_2
      : prop)])
```

```
{move 4}
```

```
>>> define line1 hyp2 : Ui A hyp2
```

```
line1 : [(hyp2_1 : that Forall
  ([(B1_3 : obj) =>
    ({def} (B1_3 E T) ->
      u E B1_3 : prop)))] =>
  (--- : that (A E T) ->
    u E A)]
```

```
{move 3}
```

```
>>> define line2 hyp2 : Mp ineq \
  line1 hyp2
```

```
line2 : [(hyp2_1 : that Forall
  ([(B1_3 : obj) =>
    ({def} (B1_3 E T) ->
      u E B1_3 : prop)))] =>
  (--- : that u E A)]
```

```
{move 3}
```

```
>>> define line3 hyp2 : Conj \
  (line2 hyp2, hyp2)
```

```
line3 : [(hyp2_1 : that Forall
  ([(B1_3 : obj) =>
    ({def} (B1_3 E T) ->
      u E B1_3 : prop)))] =>
  (--- : that (u E A) & Forall
    ([(B1_3 : obj) =>
      ({def} (B1_3 E T) ->
        u E B1_3 : prop)))]))]
```

```
{move 3}
```

```
>>> define line4 hyp2 : Fixform \
      (u E Intersection T A, Iff2 \
      (line3 hyp2, Ui (u, Separation \
      A [x1 => Forall [B1 => (B1 \
      E T) -> x1 E B1]])))
```

```
line4 : [(hyp2_1 : that Forall
      ([B1_3 : obj) =>
      ({def} (B1_3 E T) ->
      u E B1_3 : prop)))] =>
      (--- : that u E T Intersection
      A)]
```

```
{move 3}
```

```
>>> define line5 hyp1 : Simp2 \
      (Iff1 (hyp1, Ui (u, Separation \
      A [x1 => Forall [B1 => (B1 \
      E T) -> x1 E B1]])))
```

```
line5 : [(hyp1_1 : that u E T Intersection
      A) => (--- : that Forall
      ([B1_2 : obj) =>
      ({def} (B1_2 E T) ->
      u E B1_2 : prop)))]
```

```
{move 3}
```

```
>>> close
```

```
{move 3}
```

```
>>> define bothways u : Dediff line5, line4
```

```
bothways : [(u_1 : obj) => (---  
  : that (u_1 E T Intersection  
    A) == Forall ([(B1_3 : obj) =>  
      ({def} (B1_3 E T) -> u_1  
        E B1_3 : prop))]])]
```

```
{move 2}
```

```
>>> close
```

```
{move 2}
```

```
>>> define Intax1 ineq : Ug bothways
```

```
Intax1 : [(ineq_1 : that A E T) =>  
  (--- : that Forall ([(x''_2  
    : obj) =>  
      ({def} (x''_2 E T Intersection  
        A) == Forall ([(B1_4 : obj) =>  
          ({def} (B1_4 E T) -> x''_2  
            E B1_4 : prop))] : prop))]])]
```

```
{move 1}
```

```
>>> close
```

```
{move 1}
```

```

>>> define Intax T A : Ded Intax1

Intax : [(T_1 : obj), (A_1 : obj) =>
  ({def} Ded ([(inev_2 : that A_1
    E T_1) =>
      ({def} Ug ([(u_3 : obj) =>
        ({def} Dediff ([(hyp1_4 : that
          u_3 E T_1 Intersection A_1) =>
            ({def} Simp2 (hyp1_4 Iff1
              u_3 Ui A_1 Separation [(x1_8
                : obj) =>
                  ({def} Forall ([ (B1_9
                    : obj) =>
                      ({def} (B1_9 E T_1) ->
                        x1_8 E B1_9 : prop))]) : prop))] : that
                Forall ([ (B1_5 : obj) =>
                  ({def} (B1_5 E T_1) ->
                    u_3 E B1_5 : prop))]))], [(hyp2_4
                  : that Forall ([ (B1_6 : obj) =>
                    ({def} (B1_6 E T_1) ->
                      u_3 E B1_6 : prop))])) =>
                  ({def} (u_3 E T_1 Intersection
                    A_1) Fixform inev_2 Mp A_1
                    Ui hyp2_4 Conj hyp2_4 Iff2
                    u_3 Ui A_1 Separation [(x1_8
                      : obj) =>
                        ({def} Forall ([ (B1_9
                          : obj) =>
                            ({def} (B1_9 E T_1) ->
                              x1_8 E B1_9 : prop))]) : prop]] : that
                    u_3 E T_1 Intersection A_1))] : that
                (u_3 E T_1 Intersection A_1) ==
                Forall ([ (B1_5 : obj) =>
                  ({def} (B1_5 E T_1) ->
                    u_3 E B1_5 : prop))]))]) : that
                Forall ([ (x''_3 : obj) =>

```



```

      ({def} (x''_3 E T_1 Intersection
A_1) == Forall ([ (B1_5 : obj) =>
      ({def} (B1_5 E T_1) ->
        x''_3 E B1_5 : prop))]) : prop)))])) : that
(A_1 E T_1) -> Forall ([ (x''_3
: obj) =>
      ({def} (x''_3 E T_1 Intersection
A_1) == Forall ([ (B1_5 : obj) =>
      ({def} (B1_5 E T_1) -> x''_3
        E B1_5 : prop))]) : prop)))]

```

```

Intax : [(T_1 : obj), (A_1 : obj) =>
  (--- : that (A_1 E T_1) -> Forall
    [(x''_3 : obj) =>
      ({def} (x''_3 E T_1 Intersection
A_1) == Forall ([ (B1_5 : obj) =>
      ({def} (B1_5 E T_1) -> x''_3
        E B1_5 : prop))]) : prop)))]

```

```

{move 0}
end Lestrade execution

```

Above we develop the set intersection operation and prove the natural symmetric form of its associated comprehension axiom (without the asymmetric special role of A).

The following development makes use of the reasoning in Russell's paradox to show that for every set there is some object not belonging to it.

```

begin Lestrade execution

```

```

  >>> clearcurrent

```

```

{move 1}

```

```

  >>> declare x1 obj

```

```
x1 : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 1}
```

```
>>> define Russell x1 : Set x1 [y => \
      ~ (y E y)]
```

```
Russell : [(x1_1 : obj) =>
  ({def} x1_1 Set [(y_2 : obj) =>
    ({def} ~ (y_2 E y_2) : prop)] : obj)]
```

```
Russell : [(x1_1 : obj) => (--- : obj)]
```

```
{move 0}
```

```
>>> define Russellax x1 : Fixform (Forall \
  [y => (y E Russell x1) == (y E x1) & ~ (y E y)], Separation \
  x1 [y => ~ (y E y)])
```

```
Russellax : [(x1_1 : obj) =>
  ({def} Forall ([y_3 : obj) =>
    ({def} (y_3 E Russell (x1_1)) ==
      (y_3 E x1_1) & ~ (y_3 E y_3) : prop)]) Fixform
```

```

x1_1 Separation [(y_3 : obj) =>
  ({def} ~ (y_3 E y_3) : prop)] : that
Forall ([(y_2 : obj) =>
  ({def} (y_2 E Russell (x1_1)) ==
    (y_2 E x1_1) & ~ (y_2 E y_2) : prop)))]

Russellax : [(x1_1 : obj) => (---
  : that Forall ([(y_2 : obj) =>
    ({def} (y_2 E Russell (x1_1)) ==
      (y_2 E x1_1) & ~ (y_2 E y_2) : prop)))]

{move 0}

>>> open

{move 2}

>>> declare x obj

x : obj

{move 2}

>>> open

{move 3}

>>> declare rhyp1 that (Russell \
  x) E x

rhyp1 : that Russell (x) E x

```

```
{move 3}
```

```
>>> open
```

```
{move 4}
```

```
>>> declare rhyp2 that (Russell \  
x) E Russell x
```

```
rhyp2 : that Russell (x) E Russell  
(x)
```

```
{move 4}
```

```
>>> open
```

```
{move 5}
```

```
>>> declare y1 obj
```

```
y1 : obj
```

```
{move 5}
```

```
>>> define line1 : Ui (Russell \  
x, Russellax x)
```

```
line1 : that (Russell (x) E Russell  
(x)) == (Russell (x) E x) & ~ (Russell
```

```

(x) E Russell (x))

{move 4}

>>> define linea1 : Ui (Russell \
    x, Separation x [y1 => ~ (y1 \
        E y1)])

linea1 : that (Russell (x) E x Set
    [(y1_4 : obj) =>
        ({def} ~ (y1_4 E y1_4) : prop)]) ==
    (Russell (x) E x) & ~ (Russell
    (x) E Russell (x))

{move 4}

>>> define line2 : Iff1 rhyp2 \
    linea1

line2 : that (Russell (x) E x) & ~ (Russell
    (x) E Russell (x))

{move 4}

>>> define line3 : Simp2 line2

line3 : that ~ (Russell (x) E Russell
    (x))

{move 4}

```

```

>>> define line4 : Mp rhyp2 \
    line3

line4 : that ??

{move 4}

>>> close

{move 4}

>>> define line5 rhyp2 : line4

line5 : [(rhyp2_1 : that Russell
    (x) E Russell (x)) =>
    (--- : that ??)]

{move 3}

>>> define line6 : Negintro line5

line6 : that ~ (Russell (x) E Russell
    (x))

{move 3}

>>> define line7 : Ui (Russell \
    x, Russellax x)

line7 : that (Russell (x) E Russell

```

```

(x)) == (Russell (x) E x) & ~ (Russell
(x) E Russell (x))

```

```

{move 3}

```

```

>>> declare z obj

```

```

z : obj

```

```

{move 4}

```

```

>>> define linea7 : Ui (Russell \
      x, Separation x [z => ~ (z E z)])

```

```

linea7 : that (Russell (x) E x Set
  [(z_4 : obj) =>
    ({def} ~ (z_4 E z_4) : prop)]) ==
  (Russell (x) E x) & ~ (Russell
  (x) E Russell (x))

```

```

{move 3}

```

```

>>> define line8 : Iff2 (Conj \
      (rhyp1, line6), linea7)

```

```

line8 : that Russell (x) E x Set
  [(z_3 : obj) =>
    ({def} ~ (z_3 E z_3) : prop)]

```

```

{move 3}

```

```

>>> define line9 : Mp line8 line6

line9 : that ??

{move 3}

>>> close

{move 3}

>>> define notin rhyp1 : line9

notin : [(rhyp1_1 : that Russell
  (x) E x) => (--- : that ??)]

{move 2}

>>> define Notin1 : Negintro notin

Notin1 : that ~ (Russell (x) E x)

{move 2}

>>> define Enotin1 : Ei1 (Russell \
  x, Notin1)

Enotin1 : that Exists ([ (x'_2
  : obj) =>
  ({def} ~ (x'_2 E x) : prop)])

```



```

      {move 2}

    >>> close

    {move 2}

    >>> define Notin2 x : Notin1

    Notin2 : [(x_1 : obj) => (--- : that
      ~ (Russell (x_1) E x_1)))]

    {move 1}

    >>> define Enotin x : Enotin1

    Enotin : [(x_1 : obj) => (--- : that
      Exists ([(x'_2 : obj) =>
        ({def} ~ (x'_2 E x_1) : prop)))]))]

    {move 1}

    >>> close

    {move 1}

    >>> define Notin x1 : Notin2 x1

    Notin : [(x1_1 : obj) =>
      ({def} Negintro ([(rhyp1_2 : that
        Russell (x1_1) E x1_1) =>

```

```

({def} rhyp1_2 Conj Negintro ([rhyp2_6
  : that Russell (x1_1) E Russell
  (x1_1)) =>
  ({def} rhyp2_6 Mp Simp2 (rhyp2_6
    Iff1 Russell (x1_1) Ui x1_1
    Separation [(y1_11 : obj) =>
      ({def} ~ (y1_11 E y1_11) : prop)]) : that
    ??)]) Iff2 Russell (x1_1) Ui
  x1_1 Separation [(z_6 : obj) =>
    ({def} ~ (z_6 E z_6) : prop)] Mp
  Negintro ([rhyp2_4 : that Russell
    (x1_1) E Russell (x1_1)) =>
    ({def} rhyp2_4 Mp Simp2 (rhyp2_4
      Iff1 Russell (x1_1) Ui x1_1
      Separation [(y1_9 : obj) =>
        ({def} ~ (y1_9 E y1_9) : prop)]) : that
        ??)]) : that ??)]) : that
  ~ (Russell (x1_1) E x1_1))]
```

```

Notin : [(x1_1 : obj) => (--- : that
  ~ (Russell (x1_1) E x1_1))]
```

```
{move 0}
```

```
>>> define Uenotin : Ug Enotin
```

```

Uenotin : [
  ({def} Ug [(x_2 : obj) =>
    ({def} Russell (x_2) Ei1 Negintro
      [(rhyp1_4 : that Russell (x_2) E x_2) =>
        ({def} rhyp1_4 Conj Negintro
          [(rhyp2_8 : that Russell
            (x_2) E Russell (x_2)) =>
              ({def} rhyp2_8 Mp Simp2 (rhyp2_8
                Iff1 Russell (x_2) Ui x_2
```

```

      Separation [(y1_13 : obj) =>
        ({def} ~ (y1_13 E y1_13) : prop))] : that
      ??)] Iff2 Russell (x_2) Ui
x_2 Separation [(z_8 : obj) =>
  ({def} ~ (z_8 E z_8) : prop)] Mp
Negintro ([rhyp2_6 : that
  Russell (x_2) E Russell
  (x_2)) =>
  ({def} rhyp2_6 Mp Simp2 (rhyp2_6
  Iff1 Russell (x_2) Ui x_2
  Separation [(y1_11 : obj) =>
    ({def} ~ (y1_11 E y1_11) : prop))] : that
  ??)] : that ??)] : that
Exists ([x'_3 : obj) =>
  ({def} ~ (x'_3 E x_2) : prop)])))] : that
Forall ([x'_2 : obj) =>
  ({def} Exists ([x'_3 : obj) =>
    ({def} ~ (x'_3 E x'_2) : prop)] : prop)])))]

```

```

Uenotin : that Forall ([x'_2 : obj) =>
  ({def} Exists ([x'_3 : obj) =>
    ({def} ~ (x'_3 E x'_2) : prop)] : prop)]

```

```

{move 0}
end Lestrade execution

```

By a diagonalization similar to that in the Russell argument, we are able to uniformly select an element from the complement of each set.

The use of the definitions `linea1` and `linea7` (which eliminate the need to define $\widehat{\text{Russellax}}$) are a test of the matching capabilities of Lestrade. But the formulation of something like `Russellax` for a defined set construction is probably a good idea.

I believe I may use the constructions here to implement some of Zermelo's constructions where he speaks generally of choosing something not in a set.

5 The axioms of power set and union

In this section, we introduce the axioms of power set and union, which allow construction of more specific sets.

```
begin Lestrade execution
```

```
>>> clearcurrent
```

```
{move 1}
```

```
>>> declare x obj
```

```
x : obj
```

```
{move 1}
```

```
>>> declare y obj
```

```
y : obj
```

```
{move 1}
```

```
>>> declare z obj
```

```
z : obj
```

```
{move 1}
```

```
>>> postulate  $\mathcal{S}c$  x obj
```

```

Sc : [(x_1 : obj) => (--- : obj)]

{move 0}

>>> postulate Scthm x : that Forall [z => \
      (z E Sc x) == z <=< x]

Scthm : [(x_1 : obj) => (--- : that
      Forall ([z_2 : obj) =>
        ({def} (z_2 E Sc (x_1)) ==
          z_2 <=< x_1 : prop)))]])

{move 0}
end Lestrade execution

```

Here is the declaration of the power set operation (for which we use a variant of Rosser's notation $SC(x)$) and its main axiom.

```

begin Lestrade execution

>>> open

{move 2}

>>> declare X obj

X : obj

{move 2}

```

```

>>> open

{move 3}

>>> declare Xisset that Isset X

Xisset : that Isset (X)

{move 3}

>>> define line1 : Ui X Subsetrefl

line1 : that Isset (X) -> X <=<=
      X

{move 2}

>>> define line2 Xisset : Xisset \
      Mp line1

line2 : [(Xisset_1 : that Isset
      (X)) => (--- : that X <=<=
      X)]

{move 2}

>>> define line3 : Scthm X

line3 : that Forall ([(z_2 : obj) =>
      ({def} (z_2 E Sc (X)) ==

```

```

      z_2 <=<= X : prop)])

{move 2}

>>> define line4 : Ui X line3

line4 : that (X E Sc (X)) ==
  X <=<= X

{move 2}

>>> define linea5 Xisset : line2 \
  Xisset Iff2 line4

linea5 : [(Xisset_1 : that Isset
  (X)) => (--- : that X E Sc
  (X)))]

{move 2}

>>> declare v obj

v : obj

{move 3}

>>> define line6 Xisset : Fixform \
  (Isset Sc X, Add2 ((Sc X) = 0, Ei \
  (X, [v => v E (Sc X)], linea5 \
  Xisset)))

```

```

line6 : [(Xisset_1 : that Isset
          (X)) => (--- : that Isset
          (Sc (X))))]

{move 2}

>>> close

{move 2}

>>> define line7 X : Ded line6

line7 : [(X_1 : obj) => (--- : that
          Isset (X_1) -> Isset (Sc (X_1))))]

{move 1}

>>> define line7 X : Ded line5

line7 : [(X_1 : obj) => (--- : that
          Isset (X_1) -> X_1 E Sc (X_1))]

{move 1}

>>> close

{move 1}

>>> define Scofsetisset : Ug line7

```



```

Scofsetisset : [
  ({def} Ug ([X_2 : obj) =>
    ({def} Ded ([Xisset_3 : that
      Isset (X_2)) =>
        ({def} Isset (Sc (X_2)) Fixform
          (Sc (X_2) = 0) Add2 Ei (X_2, [(v_6
            : obj) =>
              ({def} v_6 E Sc (X_2) : prop)], Xisset_3
            Mp X_2 Ui Subsetrefl Iff2 X_2
            Ui Scthm (X_2)) : that Isset
              (Sc (X_2))))]) : that Isset
                (X_2 -> Isset (Sc (X_2))))]) : that
  Forall ([(x''_2 : obj) =>
    ({def} Isset (x''_2) -> Isset
      (Sc (x''_2)) : prop]]))]
```

```

Scofsetisset : that Forall ([(x''_2
  : obj) =>
    ({def} Isset (x''_2) -> Isset (Sc
      (x''_2)) : prop)])
```

```

{move 0}
end Lestrade execution
```

The power set of a set is a set.

```

begin Lestrade execution
```

```

>>> define Inownpowerset : Ug linea7
```

```

Inownpowerset : [
  ({def} Ug ([X_2 : obj) =>
    ({def} Ded ([Xisset_3 : that
```

```

      Isset (X_2)) =>
      ({def} Xisset_3 Mp X_2 Ui Subsetrefl
      Iff2 X_2 Ui Scthm (X_2) : that
      X_2 E Sc (X_2))]) : that
      Isset (X_2) -> X_2 E Sc (X_2))]) : that
Forall ([ (x''_2 : obj) =>
      ({def} Isset (x''_2) -> x''_2
      E Sc (x''_2) : prop))])

```

```

Inownpowerset : that Forall ([ (x''_2
      : obj) =>
      ({def} Isset (x''_2) -> x''_2 E Sc
      (x''_2) : prop)])

```

```

      {move 0}
end Lestrade execution

```

Each set belongs to its own power set.

```

begin Lestrade execution

```

```

>>> postulate Sc2 x : that Isset Sc x

```

```

Sc2 : [(x_1 : obj) => (--- : that
      Isset (Sc (x_1)))]

```

```

      {move 0}
end Lestrade execution

```

This is an additional axiom implicit in Zermelo's treatment but natural in any case: the power set of an atom is empty by the axioms given, but we further specify that it is the empty set. The axiom is stated in the convenient general form that all power sets are sets (which is what Zermelo

actually says), but the case of atoms (and the empty set itself) is the only case in which it is actually needed. Careful reading of Zermelo's axiom may reveal that he says that power sets are actually sets, which would fully justify this.

begin Lestrade execution

```
>>> declare w obj
```

```
w : obj
```

```
{move 1}
```

```
>>> postulate Union x obj
```

```
Union : [(x_1 : obj) => (--- : obj)]
```

```
{move 0}
```

```
>>> postulate Uthm x : that Forall [z => \
      (z E Union x) == Exists [w => (z E w) & w E x]]
```

```
Uthm : [(x_1 : obj) => (--- : that
  Forall ([z_2 : obj) =>
    ({def} (z_2 E Union (x_1)) ==
      Exists ([w_4 : obj) =>
        ({def} (z_2 E w_4) & w_4 E x_1
          : prop)]) : prop)))]
```

```
{move 0}
```

```
>>> postulate Uthm2 x : that Isset Union \
      x
```

```
Uthm2 : [(x_1 : obj) => (--- : that
      Isset (Union (x_1)))]
```

```
{move 0}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare unioninhyp that z E Union \
      y
```

```
unioninhyp : that z E Union (y)
```

```
{move 2}
```

```
>>> declare unionsubshyp that y <=& \
      x
```

```
unionsubshyp : that y <=& x
```

```
{move 2}
```

```
>>> define line1 unioninhyp : Iff1 \
      unioninhyp, Ui z Uthm y
```

```
line1 : [(unioninhyp_1 : that z E Union
```

```

(y)) => (--- : that Exists ([w_2
: obj) =>
({def} (z E w_2) & w_2 E y : prop)))]

{move 1}

>>> open

{move 3}

>>> declare w1 obj

w1 : obj

{move 3}

>>> declare wev that (z E w1) & w1 \
E y

wev : that (z E w1) & w1 E y

{move 3}

>>> define line2 wev : Mpsubs Simp2 \
wev unionsubshyp

line2 : [(w1_1 : obj), (wev_1
: that (z E w1_1) & w1_1
E y) => (--- : that w1_1 E x)]

```

```
{move 2}
```

```
>>> define line3 wev : Conj Simp1 \
      wev line2 wev
```

```
line3 : [(w1_1 : obj), (wev_1
      : that (z E .w1_1) & .w1_1
      E y) => (--- : that (z E .w1_1) & .w1_1
      E x)]
```

```
{move 2}
```

```
>>> define line4 wev : Ei1 w1 line3 \
      wev
```

```
line4 : [(w1_1 : obj), (wev_1
      : that (z E .w1_1) & .w1_1
      E y) => (--- : that Exists
      ([x'_2 : obj) =>
      ({def} (z E x'_2) & x'_2
      E x : prop)))]
```

```
{move 2}
```

```
>>> close
```

```
{move 2}
```

```
>>> define line5 unioninhyp unionsubshyp \
      : Eg (line1 unioninhyp, line4)
```

```
line5 : [(unioninhyp_1 : that z E Union
```

```

(y)), (unionsubshyp_1 : that
y <=< x) => (--- : that Exists
([(x'_2 : obj) =>
  ({def} (z E x'_2) & x'_2 E x : prop)])))]

```

```

{move 1}

```

```

>>> define line6 unioninhyp unionsubshyp \
      : Iff2 (line5 unioninhyp unionsubshyp, Ui \
z Uthm x)

```

```

line6 : [(unioninhyp_1 : that z E Union
(y)), (unionsubshyp_1 : that
y <=< x) => (--- : that z E Union
(x)))]

```

```

{move 1}

```

```

>>> close

```

```

{move 1}

```

```

>>> declare uihyp that z E Union y

```

```

uihyp : that z E Union (y)

```

```

{move 1}

```

```

>>> declare ushyp that y <=< x

```

```

ushyp : that y <=< x

```

```

{move 1}

>>> define Unionmonotone uihyp ushyp : line6 \
    uihyp ushyp

Unionmonotone : [(x_1 : obj), (y_1
    : obj), (z_1 : obj), (uihyp_1
    : that .z_1 E Union (.y_1)), (ushyp_1
    : that .y_1 <=< .x_1) =>
    ({def} uihyp_1 Iff1 .z_1 Ui Uthm (.y_1) Eg
    [(w1_3 : obj), (wev_3 : that
        (.z_1 E .w1_3) & .w1_3 E .y_1) =>
        ({def} .w1_3 Ei1 Simp1 (wev_3) Conj
        Simp2 (wev_3) Mpsubs ushyp_1 : that
        Exists [(x'_4 : obj) =>
            ({def} (.z_1 E x'_4) & x'_4
            E .x_1 : prop))]] Iff2
    .z_1 Ui Uthm (.x_1) : that .z_1 E Union
    (.x_1))]

Unionmonotone : [(x_1 : obj), (y_1
    : obj), (z_1 : obj), (uihyp_1
    : that .z_1 E Union (.y_1)), (ushyp_1
    : that .y_1 <=< .x_1) => (--- : that
    .z_1 E Union (.x_1))]

{move 0}

>>> define ++ x y : Union (x ; y)

++ : [(x_1 : obj), (y_1 : obj) =>
    ({def} Union (x_1 ; y_1) : obj)]

```



```

++ : [(x_1 : obj), (y_1 : obj) =>
      (--- : obj)]

```

```

{move 0}

```

```

>>> goal that (z E x ++ y) == (z E x) V z E y

```

```

that (z E x ++ y) == (z E x) V z E y

```

```

{move 1}

```

```

>>> open

```

```

{move 2}

```

```

>>> declare dir1 that z E x ++ y

```

```

dir1 : that z E x ++ y

```

```

{move 2}

```

```

>>> define linec1 dir1 : Iff1 dir1, Ui \
      z, Uthm (x ; y)

```

```

linec1 : [(dir1_1 : that z E x ++
      y) => (--- : that Exists ([(w_2
      : obj) =>
      ({def} (z E w_2) & w_2 E x ; y : prop)))]

```

```
{move 1}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare w83 obj
```

```
w83 : obj
```

```
{move 3}
```

```
>>> declare wev83 that (z E w83) & w83 \
      E x ; y
```

```
wev83 : that (z E w83) & w83 E x ; y
```

```
{move 3}
```

```
>>> define linec2 wev83 : Iff1 Simp2 \
      wev83, Ui w83, Pair x y
```

```
linec2 : [(w83_1 : obj), (wev83_1
      : that (z E w83_1) & w83_1
      E x ; y) => (--- : that (w83_1
      = x) V w83_1 = y)]
```

```
{move 2}
```

```
>>> open
```

```

{move 4}

>>> declare case1 that w83 = x

case1 : that w83 = x

{move 4}

>>> declare case2 that w83 = y

case2 : that w83 = y

{move 4}

>>> define linec3 case1 : Add1 \
      (z E y, Subs1 case1 Simp1 wev83)

linec3 : [(case1_1 : that w83
      = x) => (--- : that (z E x) V z E y)]

{move 3}

>>> define linec4 case2 : Add2 \
      (z E x, Subs1 case2 Simp1 wev83)

linec4 : [(case2_1 : that w83
      = y) => (--- : that (z E x) V z E y)]

```

```

{move 3}

>>> close

{move 3}

>>> define linec5 wev83 : Cases \
      linec2 wev83, linec3, linec4

linec5 : [(w83_1 : obj), (wev83_1
      : that (z E w83_1) & w83_1
      E x ; y) => (--- : that (z E x) V z E y)]

{move 2}

>>> close

{move 2}

>>> define linec6 dir1 : Eg linec1 \
      dir1, linec5

linec6 : [(dir1_1 : that z E x ++
      y) => (--- : that (z E x) V z E y)]

{move 1}

>>> declare dir2 that (z E x) V z E y

dir2 : that (z E x) V z E y

```

```
{move 2}
```

```
>>> open
```

```
{move 3}
```

```
>>> declare case1 that  $z \in x$ 
```

```
case1 : that  $z \in x$ 
```

```
{move 3}
```

```
>>> declare case2 that  $z \in y$ 
```

```
case2 : that  $z \in y$ 
```

```
{move 3}
```

```
>>> define linec7 : Inpair1 x y
```

```
linec7 : that  $x \in x$  ; y
```

```
{move 2}
```

```
>>> define linec8 : Inpair2 x y
```

```
linec8 : that  $y \in x$  ; y
```

```

{move 2}

>>> declare z1 obj

z1 : obj

{move 3}

>>> define linec9 case1 : Ei x, [z1 \
    => (z E z1) & z1 E x ; y], Conj \
    (case1, linec7)

linec9 : [(case1_1 : that z E x) =>
    (--- : that Exists ([z1_2
        : obj) =>
        ({def} (z E z1_2) & z1_2
            E x ; y : prop)))]

{move 2}

>>> define linec10 case2 : Ei y, [z1 \
    => (z E z1) & z1 E x ; y], Conj \
    (case2, linec8)

linec10 : [(case2_1 : that z E y) =>
    (--- : that Exists ([z1_2
        : obj) =>
        ({def} (z E z1_2) & z1_2
            E x ; y : prop)))]

{move 2}

```

```

>>> close

{move 2}

>>> define linec11 dir2 : Cases dir2, linec9, linec10

linec11 : [(dir2_1 : that (z E x) V z E y) =>
  (--- : that Exists ([z1_2 : obj) =>
    ({def} (z E z1_2) & z1_2 E x ; y : prop)))]

{move 1}

>>> define linec12 dir2 : Iff2 linec11 \
  dir2, Ui z, Uthm (x ; y)

linec12 : [(dir2_1 : that (z E x) V z E y) =>
  (--- : that z E Union (x ; y))]

{move 1}

>>> close

{move 1}

>>> define Binaryunion x y z : Dediff \
  linec6, linec12

Binaryunion : [(x_1 : obj), (y_1
  : obj), (z_1 : obj) =>
  ({def} Dediff ([dir1_2 : that
    z_1 E x_1 ++ y_1) =>

```

```

({def} dir1_2 Iff1 z_1 Ui Uthm
(x_1 ; y_1) Eg [(w83_3 : obj), (wev83_3
: that (z_1 E w83_3) & w83_3
E x_1 ; y_1) =>
({def} Cases (Simp2 (wev83_3) Iff1
.w83_3 Ui x_1 Pair y_1, [(case1_4
: that w83_3 = x_1) =>
({def} (z_1 E y_1) Add1
case1_4 Subs1 Simp1 (wev83_3) : that
(z_1 E x_1) V z_1 E y_1)], [(case2_4
: that w83_3 = y_1) =>
({def} (z_1 E x_1) Add2
case2_4 Subs1 Simp1 (wev83_3) : that
(z_1 E x_1) V z_1 E y_1)]) : that
(z_1 E x_1) V z_1 E y_1)] : that
(z_1 E x_1) V z_1 E y_1)], [(dir2_2
: that (z_1 E x_1) V z_1 E y_1) =>
({def} Cases (dir2_2, [(case1_4
: that z_1 E x_1) =>
({def} Ei (x_1, [(z1_5 : obj) =>
({def} (z_1 E z1_5) & z1_5
E x_1 ; y_1 : prop)], case1_4
Conj x_1 Inpair1 y_1) : that
Exists ([z1_5 : obj) =>
({def} (z_1 E z1_5) & z1_5
E x_1 ; y_1 : prop)))]], [(case2_4
: that z_1 E y_1) =>
({def} Ei (y_1, [(z1_5 : obj) =>
({def} (z_1 E z1_5) & z1_5
E x_1 ; y_1 : prop)], case2_4
Conj x_1 Inpair2 y_1) : that
Exists ([z1_5 : obj) =>
({def} (z_1 E z1_5) & z1_5
E x_1 ; y_1 : prop)))]]) Iff2
z_1 Ui Uthm (x_1 ; y_1) : that
z_1 E Union (x_1 ; y_1))] : that
(z_1 E x_1 ++ y_1) == (z_1 E x_1) V z_1
E y_1)]

```



```

Binaryunion : [(x_1 : obj), (y_1
    : obj), (z_1 : obj) => (--- : that
    (z_1 E x_1 ++ y_1) == (z_1 E x_1) V z_1
    E y_1)]

```

```

{move 0}
end Lestrade execution

```

Here we declare the set union operation and its defining theorem, and define binary union. Various utilities need to be developed, for example the theorem `Unionmonotone` needed in the proof below that a subset of a partition is a partition.

6 The axiom of choice

Here we state the axiom of choice in its original form: each partition has a choice set.

```

begin Lestrade execution

```

```

    >>> clearcurrent

```

```

{move 1}

```

```

    >>> declare x obj

```

```

    x : obj

```

```

    {move 1}

```

```

    >>> declare y obj

```

```
y : obj
```

```
{move 1}
```

```
>>> declare z obj
```

```
z : obj
```

```
{move 1}
```

```
>>> declare w obj
```

```
w : obj
```

```
{move 1}
```

```
>>> define Ispartition x : (Forall [y => \
      (y E x) -> Exists [z => z E y]]) & Forall \
      [y => (y E (Union x)) -> One [z => \
      (y E z) & z E x]]
```

```
Ispartition : [(x_1 : obj) =>
  ({def} Forall ([y_3 : obj) =>
    ({def} (y_3 E x_1) -> Exists
      [(z_5 : obj) =>
        ({def} z_5 E y_3 : prop)]) : prop)]) & Forall
  ([y_3 : obj) =>
    ({def} (y_3 E Union (x_1)) ->
      One ([z_5 : obj) =>
        ({def} (y_3 E z_5) & z_5 E x_1
          : prop)]) : prop)]) : prop)]
```

```

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

{move 0}

>>> open

    {move 2}

    >>> declare partev that Ispartition \
        x

partev : that Ispartition (x)

    {move 2}

    >>> declare subpartev that y <=< x

subpartev : that y <=< x

    {move 2}

    >>> goal that Ispartition y

that Ispartition (y)

    {move 2}

```

```

>>> declare x17 obj

x17 : obj

{move 2}

>>> declare z17 obj

z17 : obj

{move 2}

>>> goal that Forall [z17 => (z17 \
      E y) -> Exists [x17 => x17 E z17]]

that Forall ([ (z17 : obj) =>
  ({def} (z17 E y) -> Exists ([ (x17
    : obj) =>
    ({def} x17 E z17 : prop)]) : prop)])

{move 2}

>>> open

      {move 3}

>>> declare z1 obj

z1 : obj

```

```

{move 3}

>>> open

{move 4}

>>> declare ineq that z1 E y

ineq : that z1 E y

{move 4}

>>> define line1 ineq : Mpsubs \
      ineq, subpartev

line1 : [(ineq_1 : that z1
      E y) => (--- : that z1 E x)]

{move 3}

>>> define line2 ineq : Mp line1 \
      ineq, Ui z1 Simp1 partev

line2 : [(ineq_1 : that z1
      E y) => (--- : that Exists
      ([(z_2 : obj) =>
      ({def} z_2 E z1 : prop)])))]

{move 3}

```

```

>>> close

{move 3}

>>> define line3 z1 : Ded line2

line3 : [(z1_1 : obj) => (---
  : that (z1_1 E y) -> Exists
    ([(z_3 : obj) =>
      ({def} z_3 E z1_1 : prop)]))]

{move 2}

>>> close

{move 2}

>>> define line4 partev subpartev : Ug \
  line3

line4 : [(partev_1 : that Ispartition
  (x)), (subpartev_1 : that y <=<=
  x) => (--- : that Forall ([(x''_2
  : obj) =>
    ({def} (x''_2 E y) -> Exists
      ([(z_4 : obj) =>
        ({def} z_4 E x''_2 : prop)])) : prop)]))]

{move 1}

>>> goal that Forall [z17 => (z17 \
  E Union y) -> One [x17 => (z17 \

```

$E\ x17) \ \& \ x17\ E\ y]]$

```
that Forall ([ (z17 : obj) =>
  ({def} (z17 E Union (y)) ->
    One ([ (x17 : obj) =>
      ({def} (z17 E x17) & x17 E y : prop)]) : prop)])
```

{move 2}

>>> open

{move 3}

>>> declare z1 obj

z1 : obj

{move 3}

>>> open

{move 4}

>>> declare thehyp that z1 E Union \
y

thetyp : that z1 E Union (y)

{move 4}

```
>>> define line5 thehyp : Unionmonotone \
    thehyp subpartev
```

```
line5 : [(thehyp_1 : that z1
    E Union (y)) => (--- : that
    z1 E Union (x))]
```

```
{move 3}
```

```
>>> define line6 thehyp : Mp \
    line5 thehyp, Ui z1 Simp2 partev
```

```
line6 : [(thehyp_1 : that z1
    E Union (y)) => (--- : that
    One ([(z_2 : obj) =>
        ({def} (z1 E z_2) & z_2
        E x : prop)))]]
```

```
{move 3}
```

```
>>> declare w1 obj
```

```
w1 : obj
```

```
{move 4}
```

```
>>> goal that Forall [w1 => \
    ((z1 E w1) & w1 E x) == \
    (z1 E w1) & w1 E y]
```

```
that Forall ([(w1 : obj) =>
```



```

      ({def} ((z1 E w1) & w1
E x) == (z1 E w1) & w1
E y : prop)])

```

```

{move 4}

```

```

>>> open

```

```

{move 5}

```

```

>>> declare w2 obj

```

```

w2 : obj

```

```

{move 5}

```

```

>>> open

```

```

{move 6}

```

```

>>> declare dir1 that (z1 \
      E w2) & w2 E x

```

```

dir1 : that (z1 E w2) & w2
      E x

```

```

{move 6}

```

```

>>> define line7 dir1 : Simp2 \
      dir1

```

```

line7 : [(dir1_1 : that
          (z1 E w2) & w2 E x) =>
          (--- : that w2 E x)]

```

```

{move 5}

```

```

>>> define line8 dir1 : Iff1 \
      thehyp, Ui z1 Uthm y

```

```

line8 : [(dir1_1 : that
          (z1 E w2) & w2 E x) =>
          (--- : that Exists
            ([(w_2 : obj) =>
              ({def} (z1 E w_2) & w_2
                E y : prop)]))]

```

```

{move 5}

```

```

>>> define line9 dir1 : Ui \
      z1 Simp2 partev

```

```

line9 : [(dir1_1 : that
          (z1 E w2) & w2 E x) =>
          (--- : that (z1 E Union
            (x)) -> One ([(z_3
              : obj) =>
                ({def} (z1 E z_3) & z_3
                  E x : prop)]))]

```

```

{move 5}

```

```

>>> define line10 dir1 \

```

```

      : Unionmonotone thehyp \
subpartev

```

```

line10 : [(dir1_1 : that
  (z1 E w2) & w2 E x) =>
  (--- : that z1 E Union
    (x)))]

```

```

{move 5}

```

```

>>> define line11 dir1 \
      : Mp line10 dir1, line9 \
      dir1

```

```

line11 : [(dir1_1 : that
  (z1 E w2) & w2 E x) =>
  (--- : that One ([(z_2
    : obj) =>
    ({def} (z1 E z_2) & z_2
      E x : prop)))])]

```

```

{move 5}

```

```

>>> open

```

```

{move 7}

```

```

>>> declare w3 obj

```

```

w3 : obj

```

```
{move 7}
```

```
>>> declare u obj
```

```
u : obj
```

```
{move 7}
```

```
>>> declare whyp3 that \
  Forall [u => ((z1 \
    E u) & u E x) == \
    u = w3]
```

```
whyp3 : that Forall
  ([ (u_2 : obj) =>
    ({def} ((z1 E u_2) & u_2
      E x) == u_2 = w3
    : prop)])
```

```
{move 7}
```

```
>>> define line12 whyp3 \
  : Iff1 dir1, Ui w2 \
  whyp3
```

```
line12 : [(w3_1 : obj), (whyp3_1
  : that Forall ([ (u_3
    : obj) =>
    ({def} ((z1
      E u_3) & u_3
      E x) == u_3 = w3_1
    : prop)])) =>
  (--- : that w2 = w3_1)]
```

```
{move 6}
```

```
>>> open
```

```
{move 8}
```

```
>>> declare w4 obj
```

```
w4 : obj
```

```
{move 8}
```

```
>>> declare whyp4 \  
      that (z1 E w4) & w4 \  
      E y
```

```
whyp4 : that (z1  
  E w4) & w4 E y
```

```
{move 8}
```

```
>>> define line13 \  
      whyp4 : Mpsubs Simp2 \  
      whyp4 subpartev
```

```
line13 : [(w4_1  
  : obj), (whyp4_1  
  : that (z1 E .w4_1) & .w4_1  
  E y) => (---  
  : that .w4_1 E x)]
```

```
{move 7}
```

```
>>> define line14 \  
      whyp4 : Iff1 (Conj \  
      Simp1 whyp4 line13 \  
      whyp4, Ui w4 whyp3)
```

```
line14 : [(w4_1  
      : obj), (whyp4_1  
      : that (z1 E .w4_1) & .w4_1  
      E y) => (---  
      : that .w4_1 = w3)]
```

```
{move 7}
```

```
>>> define line15 \  
      whyp4 : Subs1 line14 \  
      whyp4 Simp2 whyp4
```

```
line15 : [(w4_1  
      : obj), (whyp4_1  
      : that (z1 E .w4_1) & .w4_1  
      E y) => (---  
      : that w3 E y)]
```

```
{move 7}
```

```
>>> define line16 \  
      whyp4 : Subs1 (Eqsymm \  
      line12 whyp3, line15 \  
      whyp4)
```

```

line16 : [(w4_1
          : obj), (whyp4_1
          : that (z1 E .w4_1) & .w4_1
          E y) => (---
          : that w2 E y)]

```

```

{move 7}

```

```

>>> close

```

```

{move 7}

```

```

>>> define line17 whyp3 \
      : Eg line8 dir1 line16

```

```

line17 : [(w3_1 : obj), (whyp3_1
      : that Forall ([u_3
      : obj) =>
      ({def} ((z1
      E u_3) & u_3
      E x) == u_3 = .w3_1
      : prop)))] =>
      (--- : that w2 E y)]

```

```

{move 6}

```

```

>>> close

```

```

{move 6}

```

```

>>> define line18 dir1 \
      : Eg line11 dir1 line17

```

```

line18 : [(dir1_1 : that
            (z1 E w2) & w2 E x) =>
            (--- : that w2 E y)]

```

```

{move 5}

```

```

>>> define line19 dir1 \
      : Conj Simp1 dir1, line18 \
      dir1

```

```

line19 : [(dir1_1 : that
            (z1 E w2) & w2 E x) =>
            (--- : that (z1 E w2) & w2
            E y)]

```

```

{move 5}

```

```

>>> declare dir2 that (z1 \
                        E w2) & w2 E y

```

```

dir2 : that (z1 E w2) & w2
           E y

```

```

{move 6}

```

```

>>> define line20 dir2 \
      : Conj (Simp1 dir2, Mpsubs \
      Simp2 dir2 subpartev)

```

```

line20 : [(dir2_1 : that

```



```

(z1 E w2) & w2 E y) =>
(--- : that (z1 E w2) & w2
E x)]

```

```

{move 5}

```

```

>>> close

```

```

{move 5}

```

```

>>> define line21 w2 : Dediff \
line19, line20

```

```

line21 : [(w2_1 : obj) =>
(--- : that ((z1 E w2_1) & w2_1
E x) == (z1 E w2_1) & w2_1
E y)]

```

```

{move 4}

```

```

>>> close

```

```

{move 4}

```

```

>>> define line22 thehyp : Ug \
line21

```

```

line22 : [(thhyp_1 : that
z1 E Union (y)) => (---
: that Forall ([x''_2
: obj) =>
({def} ((z1 E x''_2) & x''_2

```

```

      E x) == (z1 E x''_2) & x''_2
      E y : prop)))]

```

```

{move 3}

```

```

>>> define line23 thehyp : Onequiv \
      line6 thehyp line22 thehyp

```

```

line23 : [(thehyp_1 : that
      z1 E Union (y)) => (---
      : that One ([(x''_2 : obj) =>
      ({def} (z1 E x'''_2) & x'''_2
      E y : prop)))]

```

```

{move 3}

```

```

>>> close

```

```

{move 3}

```

```

>>> define line24 z1 : Ded line23

```

```

line24 : [(z1_1 : obj) => (---
      : that (z1_1 E Union (y)) ->
      One ([(x'''_3 : obj) =>
      ({def} (z1_1 E x'''_3) & x'''_3
      E y : prop)))]

```

```

{move 2}

```

```

>>> close

```

```
{move 2}
```

```
>>> define line25 partev subpartev \  
      : Ug line24
```

```
line25 : [(partev_1 : that Ispartition  
            (x)), (subpartev_1 : that y <=<=  
            x) => (--- : that Forall ([ (x''_2  
            : obj) =>  
            ({def} (x''_2 E Union (y)) ->  
            One ([ (x'''_4 : obj) =>  
            ({def} (x''_2 E x'''_4) & x'''_4  
            E y : prop]) : prop]) )]]]
```

```
{move 1}
```

```
>>> close
```

```
{move 1}
```

```
>>> declare partev2 that Ispartition x
```

```
partev2 : that Ispartition (x)
```

```
{move 1}
```

```
>>> declare subpartev2 that y <=<= x
```

```
subpartev2 : that y <=<= x
```

```

{move 1}

>>> define Subpartition partev2 subpartev2 \
      : Fixform (Ispartition y, Conj (line4 \
      partev2 subpartev2, line25 partev2 subpartev2))

Subpartition : [(x_1 : obj), (y_1
      : obj), (partev2_1 : that Ispartition
      (x_1)), (subpartev2_1 : that
      .y_1 <=<= .x_1) =>
      ({def} Ispartition (y_1) Fixform
      Ug ([z1_4 : obj) =>
          ({def} Ded ([inev_5 : that
              z1_4 E .y_1) =>
              ({def} ine5 Mpsubs subpartev2_1
              Mp z1_4 Ui Simp1 (partev2_1) : that
              Exists ([z_6 : obj) =>
                  ({def} z_6 E z1_4 : prop)))])) : that
      (z1_4 E .y_1) -> Exists ([z_6
          : obj) =>
          ({def} z_6 E z1_4 : prop)))])) Conj
      Ug ([z1_4 : obj) =>
          ({def} Ded ([thehyp_5 : that
              z1_4 E Union (.y_1)) =>
              ({def} thehyp_5 Unionmonotone
              subpartev2_1 Mp z1_4 Ui Simp2
              (partev2_1) Onequiv Ug ([w2_7
                  : obj) =>
                  ({def} Dediff ([dir1_8
                      : that (z1_4 E w2_7) & w2_7
                      E .x_1) =>
                      ({def} Simp1 (dir1_8) Conj
                      thehyp_5 Unionmonotone
                      subpartev2_1 Mp z1_4 Ui
                      Simp2 (partev2_1) Eg
                      [(w3_10 : obj), (whyp3_10
                          : that Forall [(u_12

```

```

      : obj) =>
      ({def} ((z1_4
      E u_12) & u_12 E .x_1) ==
      u_12 = .w3_10 : prop))))) =>
      ({def} thehyp_5 Iff1
      z1_4 Ui Uthm (.y_1) Eg
      [(.w4_11 : obj), (whyp4_11
      : that (z1_4 E .w4_11) & .w4_11
      E .y_1) =>
      ({def} Eqsymm (dir1_8
      Iff1 w2_7 Ui whyp3_10) Subs1
      Simp1 (whyp4_11) Conj
      Simp2 (whyp4_11) Mpsubs
      subpartev2_1 Iff1
      .w4_11 Ui whyp3_10
      Subs1 Simp2 (whyp4_11) : that
      w2_7 E .y_1)] : that
      w2_7 E .y_1)] : that
      (z1_4 E w2_7) & w2_7
      E .y_1)], [(dir2_8
      : that (z1_4 E w2_7) & w2_7
      E .y_1) =>
      ({def} Simp1 (dir2_8) Conj
      Simp2 (dir2_8) Mpsubs
      subpartev2_1 : that (z1_4
      E w2_7) & w2_7 E .x_1)))] : that
      ((z1_4 E w2_7) & w2_7 E .x_1) ==
      (z1_4 E w2_7) & w2_7 E .y_1))] : that
      One ([x'''_6 : obj) =>
      ({def} (z1_4 E x'''_6) & x'''_6
      E .y_1 : prop)))])) : that
      (z1_4 E Union (.y_1)) -> One
      ([x'''_6 : obj) =>
      ({def} (z1_4 E x'''_6) & x'''_6
      E .y_1 : prop)))])) : that
      Ispartition (.y_1)]

```

```

Subpartition : [(x_1 : obj), (y_1
  : obj), (partev2_1 : that Ispartition
    (x_1)), (subpartev2_1 : that
      y_1 <=<= x_1) => (--- : that Ispartition
        (y_1))]

```

```

{move 0}
end Lestrade execution

```

We prove above that a subset of a partition is a partition.

```

begin Lestrade execution

```

```

>>> postulate Ac that Forall [x => (Ispartition \
  x) -> Exists [y => (y <=<= Union \
    x) & Forall [z => (z E x) -> \
      One [w => (w E y) & w E z]]]]

```

```

Ac : that Forall ([x_2 : obj) =>
  ({def} Ispartition (x_2) -> Exists
    ([y_4 : obj) =>
      ({def} (y_4 <=<= Union (x_2)) & Forall
        ([z_6 : obj) =>
          ({def} (z_6 E x_2) -> One
            ([w_8 : obj) =>
              ({def} (w_8 E y_4) & w_8
                E z_6 : prop)]) : prop)]) : prop)]) : prop)]) : prop)])

```

```

{move 0}

```

```

>>> declare partx that Ispartition x

```

```

partx : that Ispartition (x)

```

```

{move 1}

>>> define Product partx : Set Sc (Union \
      x) [y => Forall [z => (z E x) -> \
        One [w => (w E y) & w E z]]]

Product : [(x_1 : obj), (partx_1
  : that Ispartition (x_1)) =>
  ({def} Sc (Union (x_1)) Set [(y_2
    : obj) =>
    ({def} Forall ([z_3 : obj) =>
      ({def} (z_3 E x_1) -> One
        ((w_5 : obj) =>
          ({def} (w_5 E y_2) & w_5
            E z_3 : prop))] : prop))] : obj))]

Product : [(x_1 : obj), (partx_1
  : that Ispartition (x_1)) => (---
  : obj)]

{move 0}
end Lestrade execution

```

Examples of use of this axiom are needed. I should add the development of binary product.

7 The axiom of infinity

The axiom of infinity is introduced in the original form used by Zermelo. 0 is implemented as \emptyset and the successor operation is implemented as the singleton operation.

```

begin Lestrade execution

  >>> clearcurrent

{move 1}

  >>> declare x obj

x : obj

{move 1}

  >>> declare pred [x => prop]

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

{move 1}

  >>> define inductive pred : (Forall [x => \
    pred x -> pred Usc x]) & pred 0

inductive : [(pred_1 : [(x_2 : obj) =>
  (--- : prop)]) =>
  ({def} Forall ([x_3 : obj) =>
    ({def} pred_1 (x_3) -> pred_1
      (Usc (x_3)) : prop)]) & pred_1
  (0) : prop)]

inductive : [(pred_1 : [(x_2 : obj) =>
  (--- : prop)]) => (--- : prop)]

```



```

{move 0}

>>> postulate N obj

N : obj

{move 0}

>>> postulate Nax1 that inductive [x => \
      x E N]

Nax1 : that inductive ([x_2 : obj) =>
      ({def} x_2 E N : prop)])

{move 0}

>>> declare predindv that inductive pred

predindv : that inductive (pred)

{move 1}

>>> declare isnatev that x E N

isnatev : that x E N

{move 1}

>>> postulate Nax2 predindv isnatev : that \
      x E N

```

```

Nax2 : [(x_1 : obj), (.pred_1 : [(x_2
      : obj) => (--- : prop)])], (predindiv_1
      : that inductive (.pred_1)), (isnatev_1
      : that .x_1 E N) => (--- : that .x_1
      E N)]

```

```

{move 0}
end Lestrade execution

```

Natural numbers are defined as those objects which have all inductive properties, and it is declared that the collection of natural numbers is a set (and that belonging to this set is an inductive property). We cannot prove that having all inductive properties is an inductive property, because we have not equipped ourselves with second order quantification.

This is not exactly the same as Zermelo's development: he simply asserts the existence of a set \mathbb{N}_0 membership in which is inductive, then defines \mathbb{N} as the intersection of all inductive subsets of \mathbb{N}_0 , and shows that the latter set is uniquely determined by this procedure. But this approach is equivalent, and one is asserting the existence of a definite object.

Some declarations related to arithmetic and finite sets should appear here.

8 Commencing the theory of equivalence

This completes the development of the axioms of 1908 Zermelo set theory under Lestrade. It remains to develop the theory of equivalence following the Zermelo paper.

```

begin Lestrade execution

```

```

>>> clearcurrent

```

```

{move 1}

```

```

>>> declare x obj

```

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> declare A obj

A : obj

{move 1}

>>> declare B obj

B : obj

{move 1}

```

>>> declare disjev that A disjoint B

disjev : that A disjoint B

{move 1}

>>> define product disjev : Set Sc (A ++ \
    B) [z => Exists [x => (x E A) & Exists \
        [y => (y E B) & z = x ; y]]]

product : [(A_1 : obj), (B_1 : obj), (disjev_1
    : that A_1 disjoint B_1) =>
    ({def} Sc (A_1 ++ B_1) Set [(z_2
        : obj) =>
        ({def} Exists [(x_3 : obj) =>
            ({def} (x_3 E A_1) & Exists
                [(y_5 : obj) =>
                    ({def} (y_5 E B_1) & z_2
                        = x_3 ; y_5 : prop)]) : prop)]) : prop))] : obj)]

product : [(A_1 : obj), (B_1 : obj), (disjev_1
    : that A_1 disjoint B_1) => (---
    : obj)]

{move 0}
end Lestrade execution

```

Above I saved myself a little work by defining binary product independently of the infinitary product defined with AC. The missing ingredient would be the proof of equivalence of disjointness of A, B with $\{A, B\}$ being a partition (which would probably be good for me).

```
begin Lestrade execution
```

```
>>> declare F obj
```

```
F : obj
```

```
{move 1}
```

```
>>> declare s obj
```

```
s : obj
```

```
{move 1}
```

```
>>> declare t obj
```

```
t : obj
```

```
{move 1}
```

```
>>> define Equivalent disjev : Exists \  
  [F => (F <=<= product disjev) & Forall \  
    [s => (s E A ++ B) -> One [t => \  
      (t E F) & s E t]]]
```

```
Equivalent : [(A_1 : obj), (B_1  
  : obj), (disjev_1 : that A_1 disjoint  
  B_1) =>  
  ({def} Exists ([F_2 : obj] =>  
    ({def} (F_2 <=<= product (disjev_1)) & Forall  
      [(s_4 : obj) =>
```

```

      ({def} (s_4 E .A_1 ++ .B_1) ->
      One ([ (t_6 : obj) =>
      ({def} (t_6 E F_2) & s_4
      E t_6 : prop)]) : prop)]) : prop)]) : prop)]

```

```

Equivalent : [(A_1 : obj), (B_1
: obj), (disjev_1 : that .A_1 disjoint
.B_1) => (--- : prop)]

```

```

{move 0}

```

```

>>> define Mapping disjev F : (F <= \
product disjev) & Forall [s => (s E A ++ \
B) -> One [t => (t E F) & s E t]]

```

```

Mapping : [(A_1 : obj), (B_1 : obj), (disjev_1
: that .A_1 disjoint .B_1), (F_1
: obj) =>
({def} (F_1 <= product (disjev_1)) & Forall
[(s_3 : obj) =>
({def} (s_3 E .A_1 ++ .B_1) ->
One ([ (t_5 : obj) =>
({def} (t_5 E F_1) & s_3 E t_5
: prop)]) : prop)]) : prop)]

```

```

Mapping : [(A_1 : obj), (B_1 : obj), (disjev_1
: that .A_1 disjoint .B_1), (F_1
: obj) => (--- : prop)]

```

```

{move 0}

```

```

>>> declare ismap that Mapping disjev \
F

```

```

ismap : that disjev Mapping F

{move 1}

>>> declare c obj

c : obj

{move 1}

>>> declare d obj

d : obj

{move 1}

>>> define corresponds ismap c d : (c ; d) E F

corresponds : [(A_1 : obj), (B_1
  : obj), (.disjev_1 : that A_1 disjoint
  B_1), (F_1 : obj), (ismap_1
  : that .disjev_1 Mapping F_1), (c_1
  : obj), (d_1 : obj) =>
  ({def} (c_1 ; d_1) E F_1 : prop)]

corresponds : [(A_1 : obj), (B_1
  : obj), (.disjev_1 : that A_1 disjoint
  B_1), (F_1 : obj), (ismap_1
  : that .disjev_1 Mapping F_1), (c_1

```

```

      : obj), (d_1 : obj) => (--- : prop)]

{move 0}

>>> declare infield that s E A ++ B

infield : that s E A ++ B

{move 1}

>>> open

{move 2}

>>> define line1 : Mp infield, Ui \
      s Simp2 ismap

line1 : that One ([ (t_2 : obj) =>
      ({def} (t_2 E F) & s E t_2 : prop)])

{move 1}

>>> define theimage : The line1

theimage : obj

{move 1}

>>> define theimagefact : Fixform ((theimage \
      E F) & s E theimage, Theax line1)

```



```

theimagefact : that (theimage E F) & s E theimage

{move 1}

>>> declare u obj

u : obj

{move 2}

>>> goal that One [u => (u E theimage) & ~ (u = s)]

that One ([ (u : obj) =>
  ({def} (u E theimage) & ~ (u = s) : prop)])

{move 2}

>>> define line2 : Fixform theimage \
  E product disjev, Mpsubs Simp1 theimagefact, Simp1 \
  ismap

line2 : that theimage E product (disjev)

{move 1}

>>> define line3 : Simp2 Iff1 line2, Ui \
  theimage Separation4 Refleq product \
  disjev

```

```

line3 : that Exists ([(x_2 : obj) =>
  ({def} (x_2 E A) & Exists ([(y_4
    : obj) =>
      ({def} (y_4 E B) & theimage
        = x_2 ; y_4 : prop)]) : prop)])

{move 1}

>>> open

      {move 3}

>>> declare u1 obj

u1 : obj

      {move 3}

>>> declare witnessev1 that Witnesses \
      line3 u1

witnessev1 : that line3 Witnesses
u1

      {move 3}

>>> define line4 witnessev1 : Simp1 \
      witnessev1

line4 : [(u1_1 : obj), (witnessev1_1

```

```

      : that line3 Witnesses .u1_1) =>
      (--- : that .u1_1 E A)]

{move 2}

>>> define line5 witnessev1 : Simp2 \
      witnessev1

line5 : [(u1_1 : obj), (witnessev1_1
      : that line3 Witnesses .u1_1) =>
      (--- : that Exists [(y_2
      : obj) =>
      ({def} (y_2 E B) & theimage
      = .u1_1 ; y_2 : prop)))]

{move 2}

>>> open

      {move 4}

>>> declare v1 obj

v1 : obj

      {move 4}

>>> declare witnessev2 that Witnesses \
      line5 witnessev1 v1

witnessev2 : that line5 (witnessev1) Witnesses

```

```

v1

{move 4}

>>> define line6 witnessev2 : Simp1 \
    witnessev2

line6 : [(v1_1 : obj), (witnessev2_1
    : that line5 (witnessev1) Witnesses
    .v1_1) => (--- : that .v1_1
    E B)]

{move 3}

>>> define line7 witnessev2 : Simp2 \
    witnessev2

line7 : [(v1_1 : obj), (witnessev2_1
    : that line5 (witnessev1) Witnesses
    .v1_1) => (--- : that theimage
    = u1 ; .v1_1)]

{move 3}

>>> define line8 witnessev2 : Iff1 \
    Subs1 line7 witnessev2 Simp2 \
    theimagefact, Ui s, Pair u1 \
    v1

line8 : [(v1_1 : obj), (witnessev2_1
    : that line5 (witnessev1) Witnesses
    .v1_1) => (--- : that (s = u1) V s = .v1_1)]

```

```
{move 3}
```

```
>>> open
```

```
{move 5}
```

```
>>> declare case1 that s = u1
```

```
case1 : that s = u1
```

```
{move 5}
```

```
>>> declare u2 obj
```

```
u2 : obj
```

```
{move 5}
```

```
>>> goal that One [u2 => \  
    (u2 E theimage) & ~ (u2 \  
    = s)]
```

```
that One ([u2 : obj) =>  
    ({def} (u2 E theimage) & ~ (u2  
    = s) : prop)])
```

```
{move 5}
```

```
>>> declare case2 that s = v1
```

```

case2 : that s = v1

{move 5}

>>> goal that One [u2 => \
    (u2 E theimage) & ~ (u2 \
    = s)]

that One ([u2 : obj) =>
    ({def} (u2 E theimage) & ~ (u2
    = s) : prop)])

{move 5}

>>> close

{move 4}

>>> close

{move 3}

>>> close

{move 2}

>>> close

{move 1}

```

```
end Lestrade execution
```

Above is Zermelo’s definition of the “immediate equivalence” of disjoint sets A and B . We also define the notion of a mapping from A to B , where A, B are disjoint, realized as a subset F of the binary product of A and B defined above with the property that each element of $A \cup B$ belongs to exactly one element of F , and the notion of correspondence of c in one of the sets with a d in the other set via F .

```
begin Lestrade execution
```

```
>>> define Mappings disjev : Set (Sc \
      product disjev, Mapping (disjev))
```

```
Mappings : [(A_1 : obj), (B_1 : obj), (disjev_1
      : that A_1 disjoint B_1) =>
      ({def} Sc (product (disjev_1)) Set
      [(F_2 : obj) =>
        ({def} disjev_1 Mapping F_2 : prop)] : obj)]
```

```
Mappings : [(A_1 : obj), (B_1 : obj), (disjev_1
      : that A_1 disjoint B_1) => (---
      : obj)]
```

```
{move 0}
```

```
end Lestrade execution
```

Here we define the set of all mappings witnessing the equivalence of disjoint A and B . This set is nonempty iff $A \sim B$ holds. Note the use of the curried abstraction `Mapping(disjev)` as an argument.

```
begin Lestrade execution
```

```
>>> comment paragraph 17 page 23

{move 1}

>>> clearcurrent

{move 1}

>>> declare x obj

x : obj

{move 1}

>>> declare y obj

y : obj

{move 1}

>>> declare z obj

z : obj

{move 1}

>>> declare A obj

A : obj
```



```
{move 1}
```

```
>>> declare B obj
```

```
B : obj
```

```
{move 1}
```

```
>>> declare disjev that A disjoint B
```

```
disjev : that A disjoint B
```

```
{move 1}
```

```
>>> declare Phi obj
```

```
Phi : obj
```

```
{move 1}
```

```
>>> declare equivv that Mapping disjev \  
    Phi
```

```
equivv : that disjev Mapping Phi
```

```
{move 1}
```

```
>>> declare M obj
```

```
M : obj
```

```
{move 1}
```

```
>>> declare subsetev that M <=< A
```

```
subsetev : that M <=< A
```

```
{move 1}
```

```
>>> open
```

```
{move 2}
```

```
>>> declare x1 obj
```

```
x1 : obj
```

```
{move 2}
```

```
>>> declare y1 obj
```

```
y1 : obj
```

```
{move 2}
```

```
>>> declare z1 obj
```

```
z1 : obj
```

```
{move 2}
```

```
>>> define Phi1 : Set (Phi, [z1 \
    => Exists [x1 => (x1 E M) & x1 \
    E z1]])
```

```
Phi1 : obj
```

```
{move 1}
```

```
>>> define N1 : Set (B, [y1 => Exists \
    [z1 => (z1 E Phi1) & y1 E z1]])
```

```
N1 : obj
```

```
{move 1}
```

```
>>> comment show that M and N1 are \
    disjoint
```

```
{move 2}
```

```
>>> comment show that nothing is in \
    both M and N1
```

```
{move 2}
```

```
>>> open
```

```

{move 3}

>>> declare z2 obj

z2 : obj

{move 3}

>>> declare sillyhyp that Exists \
      [z2 => (z2 E M) & z2 E N1]

sillyhyp : that Exists ([ (z2_2
      : obj) =>
      ({def} (z2_2 E M) & z2_2 E N1
      : prop)])

{move 3}

>>> comment goal is now a contradiction

{move 3}

>>> declare z3 obj

z3 : obj

{move 3}

>>> declare localev that (z3 E M) & z3 \
      E N1

```

```

localev : that (z3 E M) & z3 E N1

{move 3}

>>> define line171 localev : Simp1 \
      localev

line171 : [(z3_1 : obj), (localev_1
      : that (.z3_1 E M) & .z3_1
      E N1) => (--- : that .z3_1
      E M)]

{move 2}

>>> define line172 localev : Mpsubs \
      (line171 localev, subsetev)

line172 : [(z3_1 : obj), (localev_1
      : that (.z3_1 E M) & .z3_1
      E N1) => (--- : that .z3_1
      E A)]

{move 2}

>>> close

{move 2}

>>> comment show that M is a set

```

```

{move 2}

>>> comment show that N1 is a set

{move 2}

>>> close

{move 1}

>>> comment show that Phi is a mapping \
      from M to N1

{move 1}
end Lestrade execution

```