Development of the fundamental theorem of arithmetic

M. Randall Holmes

basic declarations of logic and arithmetic 5/23

In this file, I'm developing the fundamental theorem of arithmetic (every natural number has a unique prime factorization) in Lestrade as a demonstration. I'm using Lestrade rather than Automath because I have "literate programming" implemented for easy commenting. I ought to enable this for my Automath version as well.

We need to start by implementing logical and arithmetic primitives. I'll do this in a quick and dirty way, but I ought to develop a library that could be slotted in here easily.

1 Logic

Logic falls apart into propositional logic, logic of equality, and logic of quantifiers.

1.1 Propositional logic

We begin with propositional logic basics. In the first pass, I provide postulateive propositional logic primitives plus the double negation law.

When I need to add logical material later, I will add it in a subsequent block and make a note of what later development motivated it. I'll do similar things with later sections: the idea here is to get an idea of how the development process works.

```
Lestrade execution:
```

```
declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


postulate ?? prop

>> ??: prop {move 0}


postulate -> p q prop

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


postulate & p q prop

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


postulate V p q prop

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


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

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

```
declare pp that p

>> pp: that p {move 1}


define propfix p pp : pp

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


declare qq that q

>> qq: that q {move 1}


postulate Conjunction pp qq that p & q

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


declare rr that p & q

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


postulate Simplification1 rr that p

>> Simplification1: [(.p_1:prop),(.q_1:prop),
>>        (rr_1:that (.p_1 & .q_1)) => (---:that
>>        .p_1)]
>>   {move 0}
```

```
postulate Simplification2 rr that q

>> Simplification2: [(.p_1:prop),(.q_1:prop),
>>        (rr_1:that (.p_1 & .q_1)) => (---:that
>>        .q_1)]
>>   {move 0}


declare ss that p -> q

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


postulate Modusponens pp ss that q

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


declare ded [pp => that q]

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


postulate Deduction ded that p -> q

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


postulate Addition1 q pp that p V q
```

```
>> Addition1: [(q_1:prop),(.p_1:prop),(pp_1:
>>        that .p_1) => (---:that (.p_1 V q_1))]
>>   {move 0}


postulate Addition2 p qq that p V q

>> Addition2: [(p_1:prop),(.q_1:prop),(qq_1:
>>        that .q_1) => (---:that (p_1 V .q_1))]
>>   {move 0}


declare r prop

>> r: prop {move 1}


declare tt that p V q

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


declare case1 that p->r

>> case1: that (p -> r) {move 1}


declare case2 that q->r

>> case2: that (q -> r) {move 1}


postulate Cases tt case1 case2 that r

>> Cases: [(.p_1:prop),(.q_1:prop),(tt_1:that
>>        (.p_1 V .q_1)),(.r_1:prop),(case1_1:
>>        that (.p_1 -> .r_1)),(case2_1:that (.q_1
```

```
>>        -> .r_1)) => (---:that .r_1)]
>>   {move 0}


define ~ p : p -> ??

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


declare absurd that ??

>> absurd: that ?? {move 1}


postulate Exfalso p absurd that p

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


declare maybe that ~ ~ p

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


postulate Doublenegation maybe that p

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

The development of absolute difference below requires disjunctive syllogism.

```
Lestrade execution:
```

```
clearcurrent

open

     declare p prop

>>       p: prop {move 2}


     declare q prop

>>       q: prop {move 2}


     declare rr that p V q

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


     declare ss that ~p

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


     declare pp that p

>>       pp: that p {move 2}


     define pimpq p q ss:  Deduction [pp=> Exfalso q, Modusponens pp ss]

>>       pimpq: [(p_1:prop),(q_1:prop),(ss_1:
>>              that ~(p_1)) => (Deduction([(pp_2:
>>                   that p_1) => ((q_1 Exfalso
>>                   (pp_2 Modusponens ss_1)):that
>>                   q_1)])
>>              :that (p_1 -> q_1))]
```

```
>>        {move 1}


     declare qq that q

>>      qq: that q {move 2}


     define qimpq q : Deduction [qq => qq]

>>      qimpq: [(q_1:prop) => (Deduction([(qq_2:
>>                  that q_1) => (qq_2:that q_1)])
>>             :that (q_1 -> q_1))]
>>        {move 1}


     define ds1 rr ss : Cases (rr,pimpq p q ss,qimpq q)

>>      ds1: [(.p_1:prop),(.q_1:prop),(rr_1:
>>             that (.p_1 V .q_1)),(ss_1:that
>>             ~(.p_1)) => (Cases(rr_1,pimpq(.p_1,
>>             .q_1,ss_1),qimpq(.q_1)):that .q_1)]
>>        {move 1}


     close

declare P prop

>> P: prop {move 1}


declare Q prop

>> Q: prop {move 1}


declare Rr that P V Q
```

```
>> Rr: that (P V Q) {move 1}


declare Ss that ~P

>> Ss: that ~(P) {move 1}


define Ds1 Rr Ss : ds1 Rr Ss

>> Ds1: [(.P_1:prop),(.Q_1:prop),(Rr_1:that
>>        (.P_1 V .Q_1)),(Ss_1:that ~(.P_1)) =>
>>        (Cases(Rr_1,Deduction([(pp_2:that .P_1)
>>             => ((.Q_1 Exfalso (pp_2 Modusponens
>>             Ss_1)):that .Q_1)]),
>>        Deduction([(qq_3:that .Q_1) => (qq_3:
>>             that .Q_1)]))
>>        :that .Q_1)]
>>   {move 0}
```

The definition of definition by cases below requires symmetry of or. NOTE: you are cheating, you didn't prove this!

```
Lestrade execution:

clearcurrent

declare p prop

>> p: prop {move 1}


declare q prop

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

```
declare orev that p V q

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


postulate Symmor orev that q V p

>> Symmor: [(.p_1:prop),(.q_1:prop),(orev_1:
>>        that (.p_1 V .q_1)) => (---:that (.q_1
>>        V .p_1))]
>>   {move 0}
```

1.2 Logic of equality

Here we present equality (polymorphically for any type) and the basic rules of reflexivity and substitution which govern it. Other equality material will be added in a subsequent block as needed.

```
Lestrade execution:

clearcurrent

declare T type

>> T: type {move 1}


declare x in T

>> x: in T {move 1}


declare y in T
```

```
>> y: in T {move 1}


postulate = x y prop

>> =: [(.T_1:type),(x_1:in .T_1),(y_1:in .T_1)
>>        => (---:prop)]
>>   {move 0}


declare eqev that x=y

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


declare pred [x => prop]

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


declare predev that pred x

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


postulate Reflexivity x :  that x=x

>> Reflexivity: [(.T_1:type),(x_1:in .T_1) =>
>>        (---:that (x_1 = x_1))]
>>   {move 0}


postulate Substitution eqev pred, predev that pred y

>> Substitution: [(.T_1:type),(.x_1:in .T_1),
>>        (.y_1:in .T_1),(eqev_1:that (.x_1 =
>>        .y_1)),(pred_1:[(x_2:in .T_1) => (---:
```

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


define Subs eqev predev :  Substitution eqev pred, predev

>> Subs: [(.T_1:type),(.x_1:in .T_1),(.y_1:in
>>        .T_1),(eqev_1:that (.x_1 = .y_1)),(.pred_1:
>>        [(x_2:in .T_1) => (---:prop)]),
>>        (predev_1:that .pred_1(.x_1)) => (Substitution(eqev_1,
>>        .pred_1,predev_1):that .pred_1(.y_1))]
>>   {move 0}
```

adding more theorems about equality for subsequent developments.

```
Lestrade execution:

clearcurrent

declare T type

>> T: type {move 1}


declare x in T

>> x: in T {move 1}


declare y in T

>> y: in T {move 1}
```

```
declare u in T

>> u: in T {move 1}


declare eqev that x=y

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


define Symmeq eqev :  Substitution eqev [u => u=x] Reflexivity x

>> Symmeq: [(.T_1:type),(.x_1:in .T_1),(.y_1:
>>        in .T_1),(eqev_1:that (.x_1 = .y_1))
>>        => (Substitution(eqev_1,[(u_2:in .T_1)
>>             => ((u_2 = .x_1):prop)]
>>        ,Reflexivity(.x_1)):that (.y_1 = .x_1))]
>>   {move 0}
```

1.3 Logic of quantifiers

Here we present the logic of quantifiers, again polymorphically for any type.

```
Lestrade execution:

clearcurrent

declare T type

>> T: type {move 1}


declare x in T

>> x: in T {move 1}
```

```
declare pred [x => prop]

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


postulate Forall pred prop

>> Forall: [(.T_1:type),(pred_1:[(x_2:in .T_1)
>>             => (---:prop)])
>>        => (---:prop)]
>>   {move 0}


postulate Exists pred prop

>> Exists: [(.T_1:type),(pred_1:[(x_2:in .T_1)
>>             => (---:prop)])
>>        => (---:prop)]
>>   {move 0}


declare existsev that pred x

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


postulate Existint pred, existsev that Exists pred

>> Existint: [(.T_1:type),(pred_1:[(x_2:in .T_1)
>>             => (---:prop)]),
>>        (.x_1:in .T_1),(existsev_1:that pred_1(.x_1))
>>        => (---:that Exists(pred_1))]
>>   {move 0}


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

```
>> Ei: [(.T_1:type),(x_1:in .T_1),(.pred_1:[(x_2:
>>             in .T_1) => (---:prop)]),
>>        (existsev_1:that .pred_1(x_1)) => (Existint(.pred_1,
>>        existsev_1):that Exists(.pred_1))]
>>   {move 0}


declare univev that Forall pred

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


declare y in T

>> y: in T {move 1}


postulate Ui univev y that pred y

>> Ui: [(.T_1:type),(.pred_1:[(x_2:in .T_1)
>>             => (---:prop)]),
>>        (univev_1:that Forall(.pred_1)),(y_1:
>>        in .T_1) => (---:that .pred_1(y_1))]
>>   {move 0}


declare univev2 [x => that pred x]

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


postulate Ug univev2 that Forall pred

>> Ug: [(.T_1:type),(.pred_1:[(x_2:in .T_1)
>>             => (---:prop)]),
>>        (univev2_1:[(x_3:in .T_1) => (---:that
```

```
>>              .pred_1(x_3))])
>>        => (---:that Forall(.pred_1))]
>>   {move 0}


declare p prop

>> p: prop {move 1}


declare existsev2 that Exists pred

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


declare witnessev [x,existsev => that p]

>> witnessev: [(x_1:in T),(existsev_1:that pred(x_1))
>>        => (---:that p)]
>>   {move 1}


postulate Eg existsev2 witnessev that p

>> Eg: [(.T_1:type),(.pred_1:[(x_2:in .T_1)
>>              => (---:prop)]),
>>        (existsev2_1:that Exists(.pred_1)),(.p_1:
>>        prop),(witnessev_1:[(x_3:in .T_1),(existsev_3:
>>             that .pred_1(x_3)) => (---:that
>>             .p_1)])
>>        => (---:that .p_1)]
>>   {move 0}
```

Here we present definite description. That the description operator takes the evidence for the supporting propositions as arguments prevents us from having to worry about default behavior of the operator when the supporting

propositions do not hold.

```
Lestrade execution:

clearcurrent

declare T type

>> T: type {move 1}


declare x in T

>> x: in T {move 1}


declare pred [x => prop]

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


declare existsev that Exists pred

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


declare y in T

>> y: in T {move 1}


declare xev that pred x

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


declare yev that pred y
```

```
>> yev: that pred(y) {move 1}


declare uniqueev [x,y,xev,yev => that x=y]

>> uniqueev: [(x_1:in T),(y_1:in T),(xev_1:that
>>        pred(x_1)),(yev_1:that pred(y_1)) =>
>>        (---:that (x_1 = y_1))]
>>   {move 1}


postulate The existsev uniqueev in T

>> The: [(.T_1:type),(.pred_1:[(x_2:in .T_1)
>>             => (---:prop)]),
>>        (existsev_1:that Exists(.pred_1)),(uniqueev_1:
>>        [(x_3:in .T_1),(y_3:in .T_1),(xev_3:
>>             that .pred_1(x_3)),(yev_3:that
>>             .pred_1(y_3)) => (---:that (x_3
>>             = y_3))])
>>        => (---:in .T_1)]
>>   {move 0}


postulate Theax existsev uniqueev that pred The existsev uniqueev

>> Theax: [(.T_1:type),(.pred_1:[(x_2:in .T_1)
>>             => (---:prop)]),
>>        (existsev_1:that Exists(.pred_1)),(uniqueev_1:
>>        [(x_3:in .T_1),(y_3:in .T_1),(xev_3:
>>             that .pred_1(x_3)),(yev_3:that
>>             .pred_1(y_3)) => (---:that (x_3
>>             = y_3))])
>>        => (---:that .pred_1((existsev_1 The
>>        uniqueev_1)))]
>>   {move 0}
```

Added later, on the occasion of defining absolute difference, the postulateion of definition by cases. Note that the block structure of Lestrade is used here to prevent the namespace from being cluttered with nonce definitions of Functions used in this proof. This has the further effect that the final proof objects are quite large, since the nonce defined notions are expanded out.

There is an interesting detour in the proof where I exploit symmetry to avoid proving uniqueness in the same way twice (by reusing the function defined in the first case). Look at the functions with names starting with `Uline`.

```
Lestrade execution:

clearcurrent

open

     declare T type

>>       T: type {move 2}


     declare p prop

>>       p: prop {move 2}


     declare q prop

>>       q: prop {move 2}


     declare casesev that p V q

>>       casesev: that (p V q) {move 2}
```

```
     declare a in T

>>      a: in T {move 2}


     declare b in T

>>      b: in T {move 2}


     declare overlap that (p & q) -> a=b

>>      overlap: that ((p & q) -> (a = b)) {move
>>        2}


     open

          open

               declare revev that q&p

>>                revev: that (q & p) {move
>>                  4}


               define unrevev revev: Conjunction(Simplification2 revev, \
               Simplification1 revev)

>>                unrevev: [(revev_1:that (q
>>                       & p)) => ((Simplification2(revev_1)
>>                       Conjunction Simplification1(revev_1)):
>>                       that (p & q))]
>>                  {move 3}


               define revline1 revev :  Modusponens (unrevev revev,overlap)
```

```
>>                revline1: [(revev_1:that (q
>>                       & p)) => ((unrevev(revev_1)
>>                       Modusponens overlap):
>>                       that (a = b))]
>>                  {move 3}


               define revline2 revev :  Symmeq(revline1 revev)

>>                revline2: [(revev_1:that (q
>>                       & p)) => (Symmeq(revline1(revev_1)):
>>                       that (b = a))]
>>                  {move 3}


               close

          define overlap2  :  Deduction revline2

>>           overlap2: [(Deduction([(revev_1:
>>                       that (q & p)) => (Symmeq(((Simplification2(revev_1)
>>                       Conjunction Simplification1(revev_1))
>>                       Modusponens overlap)):
>>                       that (b = a))])
>>                    :that ((q & p) -> (b = a)))]
>>               {move 2}


          open

               declare x in T

>>                x: in T {move 4}


               define casepred x: (p& x=a)V (q &x=b)
```

```
>>                casepred: [(x_1:in T) => (((p
>>                       & (x_1 = a)) V (q & (x_1
>>                       = b))):prop)]
>>                  {move 3}


               close

          declare x in T

>>           x: in T {move 3}


          declare pp that p

>>           pp: that p {move 3}


          define casepred1 x pp:  \
               Addition1(q&x=b,Conjunction(pp,Reflexivity a))

>>           casepred1: [(x_1:in T),(pp_1:that
>>                  p) => (((q & (x_1 = b)) Addition1
>>                  (pp_1 Conjunction Reflexivity(a))):
>>                  that ((p & (a = a)) V (q &
>>                  (x_1 = b))))]
>>             {move 2}


          define casepred11 pp: \
               Existint(casepred,casepred1 a pp)

>>           casepred11: [(pp_1:that p) => (Existint([(x_2:
>>                       in T) => (((p & (x_2
>>                       = a)) V (q & (x_2 = b))):
>>                       prop)]
>>                  ,(a casepred1 pp_1)):that
>>                  Exists([(x_3:in T) => (((p
```

```
>>                      & (x_3 = a)) V (q & (x_3
>>                      = b))):prop)]))
>>                ]
>>           {move 2}


        declare qq that q

>>         qq: that q {move 3}


        define casepred2 x qq:  \
             Addition2(p&x=a,Conjunction(qq,Reflexivity b))

>>         casepred2: [(x_1:in T),(qq_1:that
>>                q) => (((p & (x_1 = a)) Addition2
>>                (qq_1 Conjunction Reflexivity(b))):
>>                that ((p & (x_1 = a)) V (q
>>                & (b = b))))]
>>           {move 2}


        define casepred21 qq:  \
             Existint(casepred,casepred2 b qq)

>>         casepred21: [(qq_1:that q) => (Existint([(x_2:
>>                      in T) => (((p & (x_2
>>                      = a)) V (q & (x_2 = b))):
>>                      prop)]
>>                ,(b casepred2 qq_1)):that
>>                Exists([(x_3:in T) => (((p
>>                      & (x_3 = a)) V (q & (x_3
>>                      = b))):prop)]))
>>                ]
>>           {move 2}


        close
```

```
     define Existsifthenelse p q casesev a, b  : \
          Cases casesev (Deduction casepred11)\
          (Deduction casepred21)

>>      Existsifthenelse: [(p_1:prop),(q_1:prop),
>>             (casesev_1:that (p_1 V q_1)),(.T_1:
>>             type),(a_1:in .T_1),(b_1:in .T_1)
>>             => (Cases(casesev_1,Deduction([(pp_4:
>>                  that p_1) => (Existint([(x_5:
>>                       in .T_1) => (((p_1 &
>>                       (x_5 = a_1)) V (q_1 &
>>                       (x_5 = b_1))):prop)]
>>                  ,((q_1 & (a_1 = b_1)) Addition1
>>                  (pp_4 Conjunction Reflexivity(a_1)))):
>>                  that Exists([(x_6:in .T_1)
>>                       => (((p_1 & (x_6 = a_1))
>>                       V (q_1 & (x_6 = b_1))):
>>                       prop)]))
>>                  ]),
>>             Deduction([(qq_8:that q_1) => (Existint([(x_9:
>>                       in .T_1) => (((p_1 &
>>                       (x_9 = a_1)) V (q_1 &
>>                       (x_9 = b_1))):prop)]
>>                  ,((p_1 & (b_1 = a_1)) Addition2
>>                  (qq_8 Conjunction Reflexivity(b_1)))):
>>                  that Exists([(x_10:in .T_1)
>>                       => (((p_1 & (x_10 = a_1))
>>                       V (q_1 & (x_10 = b_1))):
>>                       prop)]))
>>                  ]))
>>             :that Exists([(x_11:in .T_1) =>
>>                  (((p_1 & (x_11 = a_1)) V (q_1
>>                  & (x_11 = b_1))):prop)]))
>>             ]
>>        {move 1}
```

```
     open

          open

               declare x in T

>>                x: in T {move 4}


               define casepred x: (p& x=a)V (q &x=b)

>>                x: [(x_1:in T) => (((p
>>                       & (x_1 = a)) V (q & (x_1
>>                       = b))):prop)]
>>                  {move 3}


               close

          declare u in T

>>           u: in T {move 3}


          declare v in T

>>           v: in T {move 3}


          declare uev that casepred u

>>           uev: that casepred(u) {move 3}


          declare vev that casepred v

>>           vev: that casepred(v) {move 3}
```

```
          open

               declare uev1 that p&u=a

>>                uev1: that (p & (u = a)) {move
>>                  4}


               declare uev2 that q&u=b

>>                uev2: that (q & (u = b)) {move
>>                  4}


               define uline1 uev1:  Simplification1 uev1

>>                uline1: [(uev1_1:that (p &
>>                       (u = a))) => (Simplification1(uev1_1):
>>                       that p)]
>>                  {move 3}


               define uline2 uev1:  Simplification2 uev1

>>                uline2: [(uev1_1:that (p &
>>                       (u = a))) => (Simplification2(uev1_1):
>>                       that (u = a))]
>>                  {move 3}


               open

                    declare vev1 that p&v=a

>>                     vev1: that (p & (v =
>>                       a)) {move 5}
```

```
                    declare vev2 that q&v=b

>>                     vev2: that (q & (v =
>>                       b)) {move 5}


                    define vline1 vev1: Simplification1 vev1

>>                     vline1: [(vev1_1:that
>>                            (p & (v = a))) =>
>>                            (Simplification1(vev1_1):
>>                            that p)]
>>                       {move 4}


                    define vline2 vev1:  Simplification2 vev1

>>                     vline2: [(vev1_1:that
>>                            (p & (v = a))) =>
>>                            (Simplification2(vev1_1):
>>                            that (v = a))]
>>                       {move 4}


                    declare w in T

>>                     w: in T {move 5}


                    define vline3 vev1:  \
                         Substitution(uline2 uev1,[w=>u=w],Reflexivity u)

>>                     vline3: [(vev1_1:that
>>                            (p & (v = a))) =>
>>                            (Substitution(uline2(uev1),
>>                            [(w_2:in T) => ((u
>>                                 = w_2):prop)]
```

```
>>                          ,Reflexivity(u)):
>>                          that (u = a))]
>>                    {move 4}


                  define vline4 vev1:  Subs(Symmeq(vline2 vev1),vline3 vev1)

>>                   vline4: [(vev1_1:that
>>                          (p & (v = a))) =>
>>                          ((Symmeq(vline2(vev1_1))
>>                          Subs vline3(vev1_1)):
>>                          that (u = v))]
>>                    {move 4}


                  define vline5 vev2 :  Simplification1 vev2

>>                   vline5: [(vev2_1:that
>>                          (q & (v = b))) =>
>>                          (Simplification1(vev2_1):
>>                          that q)]
>>                    {move 4}


                  define vline6 vev2 :  Simplification2 vev2

>>                   vline6: [(vev2_1:that
>>                          (q & (v = b))) =>
>>                          (Simplification2(vev2_1):
>>                          that (v = b))]
>>                    {move 4}


                  define vline7 vev2 : Conjunction(uline1 uev1,vline5 vev2)

>>                   vline7: [(vev2_1:that
>>                          (q & (v = b))) =>
>>                          ((uline1(uev1) Conjunction
```

```
>>                          vline5(vev2_1)):
>>                          that (p & q))]
>>                    {move 4}


                  define vline8 vev2 : Modusponens(vline7 vev2,overlap)

>>                   vline8: [(vev2_1:that
>>                          (q & (v = b))) =>
>>                          ((vline7(vev2_1)
>>                          Modusponens overlap):
>>                          that (a = b))]
>>                    {move 4}


                  define vline9 vev2 :  \
                       Substitution(uline2 uev1,[w => u=w],Reflexivity u)

>>                   vline9: [(vev2_1:that
>>                          (q & (v = b))) =>
>>                          (Substitution(uline2(uev1),
>>                          [(w_2:in T) => ((u
>>                               = w_2):prop)]
>>                          ,Reflexivity(u)):
>>                          that (u = a))]
>>                    {move 4}


                  define vline10 vev2 :  Subs(vline8 vev2,vline9 vev2)

>>                   vline10: [(vev2_1:that
>>                          (q & (v = b))) =>
>>                          ((vline8(vev2_1)
>>                          Subs vline9(vev2_1)):
>>                          that (u = b))]
>>                    {move 4}
```

```
                    define vline11 vev2 :  \
                         Subs(Symmeq(vline6 vev2),vline10 vev2)

>>                     vline11: [(vev2_1:that
>>                            (q & (v = b))) =>
>>                            ((Symmeq(vline6(vev2_1))
>>                            Subs vline10(vev2_1)):
>>                            that (u = v))]
>>                       {move 4}


                    close

                define uline3 uev1 :  \
                    Cases(vev,Deduction vline4,Deduction vline11)

>>                 uline3: [(uev1_1:that (p &
>>                        (u = a))) => (Cases(vev,
>>                        Deduction([(vev1_2:that
>>                            (p & (v = a))) =>
>>                            ((Symmeq(Simplification2(vev1_2))
>>                            Subs Substitution(uline2(uev1_1),
>>                            [(w_4:in T) => ((u
>>                                 = w_4):prop)]
>>                            ,Reflexivity(u))):
>>                            that (u = v))]),
>>                        Deduction([(vev2_5:that
>>                            (q & (v = b))) =>
>>                            ((Symmeq(Simplification2(vev2_5))
>>                            Subs (((uline1(uev1_1)
>>                            Conjunction Simplification1(vev2_5))
>>                            Modusponens overlap)
>>                            Subs Substitution(uline2(uev1_1),
>>                            [(w_8:in T) => ((u
>>                                 = w_8):prop)]
>>                            ,Reflexivity(u)))):
>>                            that (u = v))]))
>>                       :that (u = v))]
```

```
>>                  {move 3}


                close

           declare Uev that p & u=a

>>            Uev: that (p & (u = a)) {move 3}


           define Uline3 u v vev Uev :  uline3 Uev

>>            Uline3: [(u_1:in T),(v_1:in T),
>>                   (vev_1:that ((p & (v_1 = a))
>>                   V (q & (v_1 = b)))),(Uev_1:
>>                   that (p & (u_1 = a))) => (Cases(vev_1,
>>                   Deduction([(vev1_2:that (p
>>                        & (v_1 = a))) => ((Symmeq(Simplification2(vev1_2))
>>                        Subs Substitution(Simplification2(Uev_1),
>>                        [(w_4:in T) => ((u_1
>>                             = w_4):prop)]
>>                        ,Reflexivity(u_1))):that
>>                        (u_1 = v_1))]),
>>                   Deduction([(vev2_5:that (q
>>                        & (v_1 = b))) => ((Symmeq(Simplification2(vev2_5))
>>                        Subs (((Simplification1(Uev_1)
>>                        Conjunction Simplification1(vev2_5))
>>                        Modusponens overlap)
>>                        Subs Substitution(Simplification2(Uev_1),
>>                        [(w_8:in T) => ((u_1
>>                             = w_8):prop)]
>>                        ,Reflexivity(u_1)))):
>>                        that (u_1 = v_1))]))
>>                   :that (u_1 = v_1))]
>>                {move 2}


           close
```

```
     open

          declare x in T

>>           x: in T {move 3}


          define casepred x: (p& x=a)V (q &x=b)

>>           casepred: [(x_1:in T) => (((p &
>>                  (x_1 = a)) V (q & (x_1 = b))):
>>                  prop)]
>>             {move 2}


          close

     declare U in T

>>      U: in T {move 2}


     declare V2 in T

>>      V2: in T {move 2}


     declare Uevx that casepred U

>>      Uevx: that casepred(U) {move 2}


     declare Vev that casepred V2

>>      Vev: that casepred(V2) {move 2}
```

```
     declare Uev2 that p & U=a

>>       Uev2: that (p & (U = a)) {move 2}


     define Uline4 casesev a, b, overlap U V2 Vev Uev2 : \
         Uline3 U V2 Vev Uev2

>>       Uline4: [(.p_1:prop),(.q_1:prop),(casesev_1:
>>              that (.p_1 V .q_1)),(.T_1:type),
>>              (a_1:in .T_1),(b_1:in .T_1),(overlap_1:
>>              that ((.p_1 & .q_1) -> (a_1 = b_1))),
>>              (U_1:in .T_1),(V2_1:in .T_1),(Vev_1:
>>              that ((.p_1 & (V2_1 = a_1)) V (.q_1
>>              & (V2_1 = b_1)))),(Uev2_1:that
>>              (.p_1 & (U_1 = a_1))) => (Cases(Vev_1,
>>              Deduction([(vev1_2:that (.p_1 &
>>                   (V2_1 = a_1))) => ((Symmeq(Simplification2(vev1_2))
>>                   Subs Substitution(Simplification2(Uev2_1),
>>                   [(w_4:in .T_1) => ((U_1 =
>>                        w_4):prop)]
>>                   ,Reflexivity(U_1))):that (U_1
>>                   = V2_1))]),
>>              Deduction([(vev2_5:that (.q_1 &
>>                   (V2_1 = b_1))) => ((Symmeq(Simplification2(vev2_5))
>>                   Subs (((Simplification1(Uev2_1)
>>                   Conjunction Simplification1(vev2_5))
>>                   Modusponens overlap_1) Subs
>>                   Substitution(Simplification2(Uev2_1),
>>                   [(w_8:in .T_1) => ((U_1 =
>>                        w_8):prop)]
>>                   ,Reflexivity(U_1)))):that
>>                   (U_1 = V2_1))]))
>>              :that (U_1 = V2_1))]
>>         {move 1}


     declare Uev3 that q & U=b
```

```
>>      Uev3: that (q & (U = b)) {move 2}


     define Uline5  casesev a, b, overlap U V2 Vev Uev3 : \
          Uline4(Symmor casesev, b, a, overlap2, \
          U, V2, Symmor Vev, Uev3)

>>      Uline5: [(.p_1:prop),(.q_1:prop),(casesev_1:
>>             that (.p_1 V .q_1)),(.T_1:type),
>>             (a_1:in .T_1),(b_1:in .T_1),(overlap_1:
>>             that ((.p_1 & .q_1) -> (a_1 = b_1))),
>>             (U_1:in .T_1),(V2_1:in .T_1),(Vev_1:
>>             that ((.p_1 & (V2_1 = a_1)) V (.q_1
>>             & (V2_1 = b_1)))),(Uev3_1:that
>>             (.q_1 & (U_1 = b_1))) => (Uline4(Symmor(casesev_1),
>>             b_1,a_1,Deduction([(revev_2:that
>>                  (.q_1 & .p_1)) => (Symmeq(((Simplification2(revev_2)
>>                  Conjunction Simplification1(revev_2))
>>                  Modusponens overlap_1)):that
>>                  (b_1 = a_1))]),
>>             U_1,V2_1,Symmor(Vev_1),Uev3_1):
>>             that (U_1 = V2_1))]
>>        {move 1}


     define Uniqueifthenelse casesev a,b, overlap U V2 Uevx Vev: \
         Cases (Uevx,Deduction(Uline4 (casesev, a, b, overlap, U, V2, Vev)),\
         Deduction( Uline5  (casesev, a, b, overlap, U, V2, Vev)))

>>      Uniqueifthenelse: [(.p_1:prop),(.q_1:
>>             prop),(casesev_1:that (.p_1 V .q_1)),
>>             (.T_1:type),(a_1:in .T_1),(b_1:
>>             in .T_1),(overlap_1:that ((.p_1
>>             & .q_1) -> (a_1 = b_1))),(U_1:in
>>             .T_1),(V2_1:in .T_1),(Uevx_1:that
>>             ((.p_1 & (U_1 = a_1)) V (.q_1 &
>>             (U_1 = b_1)))),(Vev_1:that ((.p_1
```

```
>>             & (V2_1 = a_1)) V (.q_1 & (V2_1
>>             = b_1)))) => (Cases(Uevx_1,Deduction([(Uev2_2:
>>                  that (.p_1 & (U_1 = a_1)))
>>                  => (Uline4(casesev_1,a_1,b_1,
>>                  overlap_1,U_1,V2_1,Vev_1,Uev2_2):
>>                  that (U_1 = V2_1))]),
>>             Deduction([(Uev3_3:that (.q_1 &
>>                  (U_1 = b_1))) => (Uline5(casesev_1,
>>                  a_1,b_1,overlap_1,U_1,V2_1,
>>                  Vev_1,Uev3_3):that (U_1 =
>>                  V2_1))]))
>>             :that (U_1 = V2_1))]
>>        {move 1}


     define ifthenelse casesev a b overlap : \
          The (Existsifthenelse p q casesev a b,\
          Uniqueifthenelse(casesev,a,b,overlap))

>>      ifthenelse: [(.p_1:prop),(.q_1:prop),
>>             (casesev_1:that (.p_1 V .q_1)),
>>             (.T_1:type),(a_1:in .T_1),(b_1:
>>             in .T_1),(overlap_1:that ((.p_1
>>             & .q_1) -> (a_1 = b_1))) => ((Existsifthenelse(.p_1,
>>             .q_1,casesev_1,a_1,b_1) The [(U_3:
>>                  in .T_1),(V2_3:in .T_1),(Uevx_3:
>>                  that ((.p_1 & (U_3 = a_1))
>>                  V (.q_1 & (U_3 = b_1)))),(Vev_3:
>>                  that ((.p_1 & (V2_3 = a_1))
>>                  V (.q_1 & (V2_3 = b_1))))
>>                  => (Uniqueifthenelse(casesev_1,
>>                  a_1,b_1,overlap_1,U_3,V2_3,
>>                  Uevx_3,Vev_3):that (U_3 =
>>                  V2_3))])
>>             :in .T_1)]
>>        {move 1}
```

```
     define ifthenelseax casesev a b overlap :\
          Theax (Existsifthenelse p q casesev a b,\
          Uniqueifthenelse(casesev,a,b,overlap))

>>      ifthenelseax: [(.p_1:prop),(.q_1:prop),
>>             (casesev_1:that (.p_1 V .q_1)),
>>             (.T_1:type),(a_1:in .T_1),(b_1:
>>             in .T_1),(overlap_1:that ((.p_1
>>             & .q_1) -> (a_1 = b_1))) => ((Existsifthenelse(.p_1,
>>             .q_1,casesev_1,a_1,b_1) Theax [(U_3:
>>                  in .T_1),(V2_3:in .T_1),(Uevx_3:
>>                  that ((.p_1 & (U_3 = a_1))
>>                  V (.q_1 & (U_3 = b_1)))),(Vev_3:
>>                  that ((.p_1 & (V2_3 = a_1))
>>                  V (.q_1 & (V2_3 = b_1))))
>>                  => (Uniqueifthenelse(casesev_1,
>>                  a_1,b_1,overlap_1,U_3,V2_3,
>>                  Uevx_3,Vev_3):that (U_3 =
>>                  V2_3))])
>>             :that ((.p_1 & ((Existsifthenelse(.p_1,
>>             .q_1,casesev_1,a_1,b_1) The [(U_5:
>>                  in .T_1),(V2_5:in .T_1),(Uevx_5:
>>                  that ((.p_1 & (U_5 = a_1))
>>                  V (.q_1 & (U_5 = b_1)))),(Vev_5:
>>                  that ((.p_1 & (V2_5 = a_1))
>>                  V (.q_1 & (V2_5 = b_1))))
>>                  => (Uniqueifthenelse(casesev_1,
>>                  a_1,b_1,overlap_1,U_5,V2_5,
>>                  Uevx_5,Vev_5):that (U_5 =
>>                  V2_5))])
>>             = a_1)) V (.q_1 & ((Existsifthenelse(.p_1,
>>             .q_1,casesev_1,a_1,b_1) The [(U_7:
>>                  in .T_1),(V2_7:in .T_1),(Uevx_7:
>>                  that ((.p_1 & (U_7 = a_1))
>>                  V (.q_1 & (U_7 = b_1)))),(Vev_7:
>>                  that ((.p_1 & (V2_7 = a_1))
>>                  V (.q_1 & (V2_7 = b_1))))
>>                  => (Uniqueifthenelse(casesev_1,
```

```
>>                   a_1,b_1,overlap_1,U_7,V2_7,
>>                   Uevx_7,Vev_7):that (U_7 =
>>                   V2_7))])
>>              = b_1))))]
>>        {move 1}


     close

declare P prop

>> P: prop {move 1}


declare Q prop

>> Q: prop {move 1}


declare Casesev that P V Q

>> Casesev: that (P V Q) {move 1}


declare  Tt  type

>> Tt: type {move 1}


declare A in Tt

>> A: in Tt {move 1}


declare B in Tt

>> B: in Tt {move 1}
```

```
declare Overlap that (P & Q) -> A = B

>> Overlap: that ((P & Q) -> (A = B)) {move
>>   1}


define Ifthenelse Casesev A B Overlap :  ifthenelse Casesev A B Overlap

>> Ifthenelse: [(.P_1:prop),(.Q_1:prop),(Casesev_1:
>>        that (.P_1 V .Q_1)),(.Tt_1:type),(A_1:
>>        in .Tt_1),(B_1:in .Tt_1),(Overlap_1:
>>        that ((.P_1 & .Q_1) -> (A_1 = B_1)))
>>        => ((Cases(Casesev_1,Deduction([(pp_5:
>>             that .P_1) => (Existint([(x_6:in
>>                  .Tt_1) => (((.P_1 & (x_6 =
>>                  A_1)) V (.Q_1 & (x_6 = B_1))):
>>                  prop)]
>>             ,((.Q_1 & (A_1 = B_1)) Addition1
>>             (pp_5 Conjunction Reflexivity(A_1)))):
>>             that Exists([(x_7:in .Tt_1) =>
>>                  (((.P_1 & (x_7 = A_1)) V (.Q_1
>>                  & (x_7 = B_1))):prop)]))
>>             ]),
>>        Deduction([(qq_9:that .Q_1) => (Existint([(x_10:
>>                  in .Tt_1) => (((.P_1 & (x_10
>>                  = A_1)) V (.Q_1 & (x_10 =
>>                  B_1))):prop)]
>>             ,((.P_1 & (B_1 = A_1)) Addition2
>>             (qq_9 Conjunction Reflexivity(B_1)))):
>>             that Exists([(x_11:in .Tt_1) =>
>>                  (((.P_1 & (x_11 = A_1)) V
>>                  (.Q_1 & (x_11 = B_1))):prop)]))
>>             ]))
>>        The [(U_12:in .Tt_1),(V2_12:in .Tt_1),
>>             (Uevx_12:that ((.P_1 & (U_12 =
>>             A_1)) V (.Q_1 & (U_12 = B_1)))),
>>             (Vev_12:that ((.P_1 & (V2_12 =
```

```
>>              A_1)) V (.Q_1 & (V2_12 = B_1))))
>>              => (Cases(Uevx_12,Deduction([(Uev2_13:
>>                   that (.P_1 & (U_12 = A_1)))
>>                   => (Cases(Vev_12,Deduction([(vev1_14:
>>                        that (.P_1 & (V2_12 =
>>                        A_1))) => ((Symmeq(Simplification2(vev1_14))
>>                        Subs Substitution(Simplification2(Uev2_13),
>>                        [(w_16:in .Tt_1) => ((U_12
>>                             = w_16):prop)]
>>                        ,Reflexivity(U_12))):
>>                        that (U_12 = V2_12))]),
>>                   Deduction([(vev2_17:that (.Q_1
>>                        & (V2_12 = B_1))) =>
>>                        ((Symmeq(Simplification2(vev2_17))
>>                        Subs (((Simplification1(Uev2_13)
>>                        Conjunction Simplification1(vev2_17))
>>                        Modusponens Overlap_1)
>>                        Subs Substitution(Simplification2(Uev2_13),
>>                        [(w_20:in .Tt_1) => ((U_12
>>                             = w_20):prop)]
>>                        ,Reflexivity(U_12)))):
>>                        that (U_12 = V2_12))]))
>>                   :that (U_12 = V2_12))]),
>>              Deduction([(Uev3_21:that (.Q_1
>>                   & (U_12 = B_1))) => (Cases(Symmor(Vev_12),
>>                   Deduction([(vev1_22:that (.Q_1
>>                        & (V2_12 = B_1))) =>
>>                        ((Symmeq(Simplification2(vev1_22))
>>                        Subs Substitution(Simplification2(Uev3_21),
>>                        [(w_24:in .Tt_1) => ((U_12
>>                             = w_24):prop)]
>>                        ,Reflexivity(U_12))):
>>                        that (U_12 = V2_12))]),
>>                   Deduction([(vev2_25:that (.P_1
>>                        & (V2_12 = A_1))) =>
>>                        ((Symmeq(Simplification2(vev2_25))
>>                        Subs (((Simplification1(Uev3_21)
>>                        Conjunction Simplification1(vev2_25))
```

```
>>                        Modusponens Deduction([(revev_27:
>>                             that (.Q_1 & .P_1))
>>                             => (Symmeq(((Simplification2(revev_27)
>>                             Conjunction Simplification1(revev_27))
>>                             Modusponens Overlap_1)):
>>                             that (B_1 = A_1))]))
>>                        Subs Substitution(Simplification2(Uev3_21),
>>                        [(w_29:in .Tt_1) => ((U_12
>>                             = w_29):prop)]
>>                        ,Reflexivity(U_12)))):
>>                        that (U_12 = V2_12))]))
>>                   :that (U_12 = V2_12))]))
>>              :that (U_12 = V2_12))])
>>         :in .Tt_1)]
>>    {move 0}


define Ifthenelseax Casesev A B Overlap : \
     propfix((P&(Ifthenelse Casesev A B Overlap)=A) V  \
     Q & (Ifthenelse Casesev A B Overlap)=B,\
     ifthenelseax Casesev A B Overlap)

>> Ifthenelseax: [(.P_1:prop),(.Q_1:prop),(Casesev_1:
>>        that (.P_1 V .Q_1)),(.Tt_1:type),(A_1:
>>        in .Tt_1),(B_1:in .Tt_1),(Overlap_1:
>>        that ((.P_1 & .Q_1) -> (A_1 = B_1)))
>>        => ((((.P_1 & (Ifthenelse(Casesev_1,
>>        A_1,B_1,Overlap_1) = A_1)) V (.Q_1 &
>>        (Ifthenelse(Casesev_1,A_1,B_1,Overlap_1)
>>        = B_1))) propfix (Cases(Casesev_1,Deduction([(pp_5:
>>             that .P_1) => (Existint([(x_6:in
>>                  .Tt_1) => (((.P_1 & (x_6 =
>>                  A_1)) V (.Q_1 & (x_6 = B_1))):
>>                  prop)]
>>             ,((.Q_1 & (A_1 = B_1)) Addition1
>>             (pp_5 Conjunction Reflexivity(A_1)))):
>>             that Exists([(x_7:in .Tt_1) =>
>>                  (((.P_1 & (x_7 = A_1)) V (.Q_1
```

```
>>                & (x_7 = B_1))):prop)]))
>>           ]),
>>       Deduction([(qq_9:that .Q_1) => (Existint([(x_10:
>>                in .Tt_1) => (((.P_1 & (x_10
>>                = A_1)) V (.Q_1 & (x_10 =
>>                B_1))):prop)]
>>           ,((.P_1 & (B_1 = A_1)) Addition2
>>           (qq_9 Conjunction Reflexivity(B_1)))):
>>           that Exists([(x_11:in .Tt_1) =>
>>                (((.P_1 & (x_11 = A_1)) V
>>                (.Q_1 & (x_11 = B_1))):prop)]))
>>           ]))
>>       Theax [(U_12:in .Tt_1),(V2_12:in .Tt_1),
>>           (Uevx_12:that ((.P_1 & (U_12 =
>>           A_1)) V (.Q_1 & (U_12 = B_1)))),
>>           (Vev_12:that ((.P_1 & (V2_12 =
>>           A_1)) V (.Q_1 & (V2_12 = B_1))))
>>           => (Cases(Uevx_12,Deduction([(Uev2_13:
>>                that (.P_1 & (U_12 = A_1)))
>>                => (Cases(Vev_12,Deduction([(vev1_14:
>>                     that (.P_1 & (V2_12 =
>>                     A_1))) => ((Symmeq(Simplification2(vev1_14))
>>                     Subs Substitution(Simplification2(Uev2_13),
>>                     [(w_16:in .Tt_1) => ((U_12
>>                          = w_16):prop)]
>>                     ,Reflexivity(U_12))):
>>                     that (U_12 = V2_12))]),
>>                Deduction([(vev2_17:that (.Q_1
>>                     & (V2_12 = B_1))) =>
>>                     ((Symmeq(Simplification2(vev2_17))
>>                     Subs (((Simplification1(Uev2_13)
>>                     Conjunction Simplification1(vev2_17))
>>                     Modusponens Overlap_1)
>>                     Subs Substitution(Simplification2(Uev2_13),
>>                     [(w_20:in .Tt_1) => ((U_12
>>                          = w_20):prop)]
>>                     ,Reflexivity(U_12)))):
>>                     that (U_12 = V2_12))]))
```

```
>>                  :that (U_12 = V2_12))]),
>>             Deduction([(Uev3_21:that (.Q_1
>>                  & (U_12 = B_1))) => (Cases(Symmor(Vev_12),
>>                  Deduction([(vev1_22:that (.Q_1
>>                       & (V2_12 = B_1))) =>
>>                       ((Symmeq(Simplification2(vev1_22))
>>                       Subs Substitution(Simplification2(Uev3_21),
>>                       [(w_24:in .Tt_1) => ((U_12
>>                            = w_24):prop)]
>>                       ,Reflexivity(U_12))):
>>                       that (U_12 = V2_12))]),
>>                  Deduction([(vev2_25:that (.P_1
>>                       & (V2_12 = A_1))) =>
>>                       ((Symmeq(Simplification2(vev2_25))
>>                       Subs (((Simplification1(Uev3_21)
>>                       Conjunction Simplification1(vev2_25))
>>                       Modusponens Deduction([(revev_27:
>>                            that (.Q_1 & .P_1))
>>                            => (Symmeq(((Simplification2(revev_27)
>>                            Conjunction Simplification1(revev_27))
>>                            Modusponens Overlap_1)):
>>                            that (B_1 = A_1))]))
>>                       Subs Substitution(Simplification2(Uev3_21),
>>                       [(w_29:in .Tt_1) => ((U_12
>>                            = w_29):prop)]
>>                       ,Reflexivity(U_12)))):
>>                       that (U_12 = V2_12))]))
>>                  :that (U_12 = V2_12))]))
>>             :that (U_12 = V2_12))]))
>>        :that ((.P_1 & (Ifthenelse(Casesev_1,
>>        A_1,B_1,Overlap_1) = A_1)) V (.Q_1 &
>>        (Ifthenelse(Casesev_1,A_1,B_1,Overlap_1)
>>        = B_1))))]
>>   {move 0}
```

Create a version of definition by cases where the two alternatives are

exclusive.

```
Lestrade execution:

open

     declare nulloverlap  that ~(P&Q)

>>      nulloverlap: that ~((P & Q)) {move 2}


     open

          declare impossible that P&Q

>>           impossible: that (P & Q) {move
>>             3}


          define consequence impossible : Exfalso(A=B,Modusponens impossible nullo

>>           consequence: [(impossible_1:that
>>                  (P & Q)) => (((A = B) Exfalso
>>                  (impossible_1 Modusponens
>>                  nulloverlap)):that (A = B))]
>>             {move 2}


          close

     define Overlap2 nulloverlap :  Deduction consequence

>>      Overlap2: [(nulloverlap_1:that ~((P
>>             & Q))) => (Deduction([(impossible_2:
>>                  that (P & Q)) => (((A = B)
>>                  Exfalso (impossible_2 Modusponens
>>                  nulloverlap_1)):that (A =
>>                  B))])
```

```
>>             :that ((P & Q) -> (A = B)))]
>>       {move 1}


     define ifthenelse2 nulloverlap:  Ifthenelse Casesev A B Overlap2 nulloverlap

>>      ifthenelse2: [(nulloverlap_1:that ~((P
>>             & Q))) => (Ifthenelse(Casesev,A,
>>             B,Overlap2(nulloverlap_1)):in Tt)]
>>       {move 1}


     define ifthenelseax2 nulloverlap:  Ifthenelseax Casesev A B Overlap2 nullover

>>      ifthenelseax2: [(nulloverlap_1:that
>>             ~((P & Q))) => (Ifthenelseax(Casesev,
>>             A,B,Overlap2(nulloverlap_1)):that
>>             ((P & (Ifthenelse(Casesev,A,B,Overlap2(nulloverlap_1))
>>             = A)) V (Q & (Ifthenelse(Casesev,
>>             A,B,Overlap2(nulloverlap_1)) =
>>             B))))]
>>       {move 1}


     close

declare Nulloverlap that ~(P&Q)

>> Nulloverlap: that ~((P & Q)) {move 1}


define Ifthenelse2 Casesev A B Nulloverlap: ifthenelse2 Nulloverlap

>> Ifthenelse2: [(.P_1:prop),(.Q_1:prop),(Casesev_1:
>>       that (.P_1 V .Q_1)),(.Tt_1:type),(A_1:
>>       in .Tt_1),(B_1:in .Tt_1),(Nulloverlap_1:
>>       that ~((.P_1 & .Q_1))) => (Ifthenelse(Casesev_1,
>>       A_1,B_1,Deduction([(impossible_2:that
```

```
>>             (.P_1 & .Q_1)) => (((A_1 = B_1)
>>             Exfalso (impossible_2 Modusponens
>>             Nulloverlap_1)):that (A_1 = B_1))]))
>>        :in .Tt_1)]
>>   {move 0}


define Ifthenelseax2 Casesev A B Nulloverlap : propfix((P & (Ifthenelse2 Casesev A

>> Ifthenelseax2: [(.P_1:prop),(.Q_1:prop),(Casesev_1:
>>        that (.P_1 V .Q_1)),(.Tt_1:type),(A_1:
>>        in .Tt_1),(B_1:in .Tt_1),(Nulloverlap_1:
>>        that ~((.P_1 & .Q_1))) => ((((.P_1 &
>>        (Ifthenelse2(Casesev_1,A_1,B_1,Nulloverlap_1)
>>        = A_1)) V (.Q_1 & (Ifthenelse2(Casesev_1,
>>        A_1,B_1,Nulloverlap_1) = B_1))) propfix
>>        Ifthenelseax(Casesev_1,A_1,B_1,Deduction([(impossible_2:
>>             that (.P_1 & .Q_1)) => (((A_1 =
>>             B_1) Exfalso (impossible_2 Modusponens
>>             Nulloverlap_1)):that (A_1 = B_1))]))
>>        ):that ((.P_1 & (Ifthenelse2(Casesev_1,
>>        A_1,B_1,Nulloverlap_1) = A_1)) V (.Q_1
>>        & (Ifthenelse2(Casesev_1,A_1,B_1,Nulloverlap_1)
>>        = B_1))))]
>>   {move 0}
```

2 Arithmetic primitives

In this section we will introduce a basic treatment of the natural numbers. Basic algebraic axioms will be supplied without proof for now, since our aim is at a different level.

2.1 True primitives of arithmetic

In this subsection we introduce the actual primitives of arithmetic, from which the items we postulate in the next subsection actually could be developed.

We will start the natural numbers with one, which seems appropriate in number theory.

We begin with the basic declarations of the natural number type, the first natural number 1, the successor operation, and the third and fourth Peano axioms.

```
Lestrade execution:

clearcurrent

postulate Nat type

>> Nat: type {move 0}


postulate 1 in Nat

>> 1: in Nat {move 0}


declare n in Nat

>> n: in Nat {move 1}


postulate Succ n in Nat

>> Succ: [(n_1:in Nat) => (---:in Nat)]
>>   {move 0}


declare m in Nat
```

```
>> m: in Nat {move 1}


define =/= n m : ~ (n=m)

>> =/=: [(n_1:in Nat),(m_1:in Nat) => (~((n_1
>>        = m_1)):prop)]
>>   {move 0}


postulate Axiom3 n that (Succ n) =/= 1

>> Axiom3: [(n_1:in Nat) => (---:that (Succ(n_1)
>>        =/= 1))]
>>   {move 0}


declare equalsuccs that (Succ n) = Succ m

>> equalsuccs: that (Succ(n) = Succ(m)) {move
>>   1}


postulate Axiom4 equalsuccs that n=m

>> Axiom4: [(.n_1:in Nat),(.m_1:in Nat),(equalsuccs_1:
>>        that (Succ(.n_1) = Succ(.m_1))) => (---:
>>        that (.n_1 = .m_1))]
>>   {move 0}
```

We introduce our primitive form of induction. The fact that the induction step appears before the basis is dictated by the needs of the implicit argument deduction feature of Lestrade: one can reliably deduce the predicate from the form of the induction step, but not always from the form of the basis step.

One should notice a very close formal analogy between iteration and induction.

```
Lestrade execution:

clearcurrent

declare m in Nat

>> m: in Nat {move 1}


declare pred [m => prop]

>> pred: [(m_1:in Nat) => (---:prop)]
>>   {move 1}


declare indhyp that pred m

>> indhyp: that pred(m) {move 1}


declare indstep [m, indhyp => that pred (Succ m)]

>> indstep: [(m_1:in Nat),(indhyp_1:that pred(m_1))
>>        => (---:that pred(Succ(m_1)))]
>>   {move 1}


declare basis that pred 1

>> basis: that pred(1) {move 1}


declare n in Nat

>> n: in Nat {move 1}


postulate Induction indstep, basis n that pred n
```

```
>> Induction: [(.pred_1:[(m_2:in Nat) => (---:
>>              prop)]),
>>        (indstep_1:[(m_3:in Nat),(indhyp_3:that
>>             .pred_1(m_3)) => (---:that .pred_1(Succ(m_3)))]),
>>        (basis_1:that .pred_1(1)),(n_1:in Nat)
>>        => (---:that .pred_1(n_1))]
>>   {move 0}
```

We introduce iteration.

The full function `Iterate` is quite elaborate, as it allows iteration to pass through a series of types indexed by natural numbers. The function `Simpleiter` captures simple iteration of a function with the same domain and codomain. We provide `Iterate` because its close parallelism with the induction primitive is worthy of note. We might even find occasion to use it.

It should be noted that `Simpleiter(f,s,n)` for positive n translates to $f^{n-1}(s)$, the application of the function f to the starting point s $n-1$ times rather than n times, while `Simpleiter(f,s,1)` $= s$. This is perhaps odd, but a natural consequence of starting the natural numbers with 1.

Note the use of the `propfix` function defined above to coerce the output types of the equational identities `Simpleiterstart` and `Simpleiternext` into forms in terms of `Simpleiter` (they would otherwise have rather obscure forms in terms of `Iterate`).

```
Lestrade execution:

clearcurrent

declare m in Nat

>> m: in Nat {move 1}


declare T  [m => type]

>> T: [(m_1:in Nat) => (---:type)]
```

```
>>   {move 1}


declare s in T m

>> s: in T(m) {move 1}


declare stepfun [m,s => in T Succ m]

>> stepfun: [(m_1:in Nat),(s_1:in T(m_1)) =>
>>        (---:in T(Succ(m_1)))]
>>   {move 1}


declare start in T 1

>> start: in T(1) {move 1}


declare n in Nat

>> n: in Nat {move 1}


postulate Iterate stepfun, start n in T n

>> Iterate: [(.T_1:[(m_2:in Nat) => (---:type)]),
>>        (stepfun_1:[(m_3:in Nat),(s_3:in .T_1(m_3))
>>             => (---:in .T_1(Succ(m_3)))]),
>>        (start_1:in .T_1(1)),(n_1:in Nat) =>
>>        (---:in .T_1(n_1))]
>>   {move 0}


postulate Iteratestart stepfun, start \
     that (Iterate stepfun, start 1) = start
```

```
>> Iteratestart: [(.T_1:[(m_2:in Nat) => (---:
>>             type)]),
>>        (stepfun_1:[(m_3:in Nat),(s_3:in .T_1(m_3))
>>             => (---:in .T_1(Succ(m_3)))]),
>>        (start_1:in .T_1(1)) => (---:that (Iterate(stepfun_1,
>>        start_1,1) = start_1))]
>>   {move 0}


postulate Iteratenext stepfun, start n \
     that (Iterate stepfun, start, Succ n) \
     = stepfun(n,Iterate stepfun, start n)

>> Iteratenext: [(.T_1:[(m_2:in Nat) => (---:
>>             type)]),
>>        (stepfun_1:[(m_3:in Nat),(s_3:in .T_1(m_3))
>>             => (---:in .T_1(Succ(m_3)))]),
>>        (start_1:in .T_1(1)),(n_1:in Nat) =>
>>        (---:that (Iterate(stepfun_1,start_1,
>>        Succ(n_1)) = (n_1 stepfun_1 Iterate(stepfun_1,
>>        start_1,n_1))))]
>>   {move 0}


declare Tt type

>> Tt: type {move 1}


declare x in Tt

>> x: in Tt {move 1}


declare stepfun2 [x => in Tt]

>> stepfun2: [(x_1:in Tt) => (---:in Tt)]
>>   {move 1}
```

```
declare start2 in Tt

>> start2: in Tt {move 1}


declare y in Nat

>> y: in Nat {move 1}


define Simpleiter stepfun2, start2 y: Iterate [n,x=>stepfun2 x] start2 y

>> Simpleiter: [(.Tt_1:type),(stepfun2_1:[(x_2:
>>             in .Tt_1) => (---:in .Tt_1)]),
>>        (start2_1:in .Tt_1),(y_1:in Nat) =>
>>        (Iterate([(n_4:in Nat),(x_4:in .Tt_1)
>>             => (stepfun2_1(x_4):in .Tt_1)]
>>        ,start2_1,y_1):in .Tt_1)]
>>   {move 0}


define Simpleiterstart stepfun2, start2:  \
     propfix((Simpleiter stepfun2, start2 1) \
     = start2,Iteratestart [n,x=>stepfun2 x] start2)

>> Simpleiterstart: [(.Tt_1:type),(stepfun2_1:
>>        [(x_2:in .Tt_1) => (---:in .Tt_1)]),
>>        (start2_1:in .Tt_1) => (((Simpleiter(stepfun2_1,
>>        start2_1,1) = start2_1) propfix Iteratestart([(n_4:
>>             in Nat),(x_4:in .Tt_1) => (stepfun2_1(x_4):
>>             in .Tt_1)]
>>        ,start2_1)):that (Simpleiter(stepfun2_1,
>>        start2_1,1) = start2_1))]
>>   {move 0}
```

```
define Simpleiternext stepfun2,start2,y :  \
     propfix((Simpleiter stepfun2,start2, Succ y) \
     = stepfun2(Simpleiter stepfun2, start2,y),\
     Iteratenext [n,x=>stepfun2 x] start2 y)

>> Simpleiternext: [(.Tt_1:type),(stepfun2_1:
>>        [(x_2:in .Tt_1) => (---:in .Tt_1)]),
>>        (start2_1:in .Tt_1),(y_1:in Nat) =>
>>        (((Simpleiter(stepfun2_1,start2_1,Succ(y_1))
>>        = stepfun2_1(Simpleiter(stepfun2_1,start2_1,
>>        y_1))) propfix Iteratenext([(n_4:in
>>             Nat),(x_4:in .Tt_1) => (stepfun2_1(x_4):
>>             in .Tt_1)]
>>        ,start2_1,y_1)):that (Simpleiter(stepfun2_1,
>>        start2_1,Succ(y_1)) = stepfun2_1(Simpleiter(stepfun2_1,
>>        start2_1,y_1))))]
>>   {move 0}
```

Our immediate purpose in postulating iteration is to define addition and multiplication.

```
Lestrade execution:

clearcurrent

declare m in Nat

>> m: in Nat {move 1}


declare n in Nat

>> n: in Nat {move 1}


define + m n :  Simpleiter Succ, Succ m, n
```

```
>> +: [(m_1:in Nat),(n_1:in Nat) => (Simpleiter(Succ,
>>        Succ(m_1),n_1):in Nat)]
>>   {move 0}


define * m n :  Simpleiter +(m), m, n

>> *: [(m_1:in Nat),(n_1:in Nat) => (Simpleiter([(n_2:
>>             in Nat) => ((m_1 + n_2):in Nat)]
>>        ,m_1,n_1):in Nat)]
>>   {move 0}
```

2.2 Algebra axioms (actually provable)

We begin with axioms 7 through 9 of the usual presentations of first order Peano arithmetic, which we actually prove, since they are just cases of the definition of iteration. Our presentation of the last axiom is different from the usual formulations, but we do know secretly that addition is commutative.

```
Lestrade execution:

clearcurrent

declare x in Nat

>> x: in Nat {move 1}


declare y in Nat

>> y: in Nat {move 1}


declare z in Nat
```

```
>> z: in Nat {move 1}


define Addone x : propfix((x+1)=Succ x,Simpleiterstart(Succ,Succ x))

>> Addone: [(x_1:in Nat) => ((((x_1 + 1) = Succ(x_1))
>>        propfix Simpleiterstart(Succ,Succ(x_1))):
>>        that ((x_1 + 1) = Succ(x_1)))]
>>   {move 0}


define Addnext x y : propfix((x+Succ y)=Succ(x+y), \
      Simpleiternext(Succ,Succ x,y))

>> Addnext: [(x_1:in Nat),(y_1:in Nat) => ((((x_1
>>        + Succ(y_1)) = Succ((x_1 + y_1))) propfix
>>        Simpleiternext(Succ,Succ(x_1),y_1)):
>>        that ((x_1 + Succ(y_1)) = Succ((x_1
>>        + y_1))))]
>>   {move 0}


define Multone x : propfix((x*1)=x,Simpleiterstart(+(x),x))

>> Multone: [(x_1:in Nat) => ((((x_1 * 1) =
>>        x_1) propfix Simpleiterstart([(n_2:in
>>             Nat) => ((x_1 + n_2):in Nat)]
>>        ,x_1)):that ((x_1 * 1) = x_1))]
>>   {move 0}


define Multnext x y :  propfix((x*Succ y)=x+x*y, \
      Simpleiternext(+(x),x,y))

>> Multnext: [(x_1:in Nat),(y_1:in Nat) => ((((x_1
>>        * Succ(y_1)) = (x_1 + (x_1 * y_1)))
>>        propfix Simpleiternext([(n_2:in Nat)
>>             => ((x_1 + n_2):in Nat)]
```

```
>>        ,x_1,y_1)):that ((x_1 * Succ(y_1)) =
>>        (x_1 + (x_1 * y_1))))]
>>   {move 0}
```

In the next block, we present algebraic axioms without proof (but these are in fact provable from the axioms given so far, as is well-known, and some of this could be filled in later, certainly if these preamble sections were to be used as a general purpose library).

It is useful to remember that Lestrade does not have order of operations: all binary operations have the same precedence and group to the right. Thus `x*y+z` parses as `x*(y+z)` in the statement of the distributive law below, but we leave the redundant parentheses in place for the comfort of the reader. In Lestrade output, all parentheses are shown.

The multiplication cancellation property is simpler because natural numbers are taken to be positive integers. But we will see that this choice also causes complexities.

```
Lestrade execution:

postulate Commadd x y that (x+y)=y+x

>> Commadd: [(x_1:in Nat),(y_1:in Nat) => (---:
>>        that ((x_1 + y_1) = (y_1 + x_1)))]
>>   {move 0}


postulate Assocadd x y z that ((x+y)+z)=x+y+z

>> Assocadd: [(x_1:in Nat),(y_1:in Nat),(z_1:
>>        in Nat) => (---:that (((x_1 + y_1) +
>>        z_1) = (x_1 + (y_1 + z_1))))]
>>   {move 0}


postulate Commult x y that (x*y)=y*x
```

```
>> Commult: [(x_1:in Nat),(y_1:in Nat) => (---:
>>        that ((x_1 * y_1) = (y_1 * x_1)))]
>>   {move 0}


postulate Assocmult x y z that ((x*y)*z)=x*y*z

>> Assocmult: [(x_1:in Nat),(y_1:in Nat),(z_1:
>>        in Nat) => (---:that (((x_1 * y_1) *
>>        z_1) = (x_1 * (y_1 * z_1))))]
>>   {move 0}


postulate Dist x y z that (x*(y+z))=(x*y)+x*z

>> Dist: [(x_1:in Nat),(y_1:in Nat),(z_1:in
>>        Nat) => (---:that ((x_1 * (y_1 + z_1))
>>        = ((x_1 * y_1) + (x_1 * z_1))))]
>>   {move 0}


declare addcancel that (x+z)=y+z

>> addcancel: that ((x + z) = (y + z)) {move
>>   1}


postulate Addcancel addcancel that x=y

>> Addcancel: [(.x_1:in Nat),(.z_1:in Nat),(.y_1:
>>        in Nat),(addcancel_1:that ((.x_1 + .z_1)
>>        = (.y_1 + .z_1))) => (---:that (.x_1
>>        = .y_1))]
>>   {move 0}


declare multcancel that (x*z)=y*z
```

```
>> multcancel: that ((x * z) = (y * z)) {move
>>   1}


postulate Multcancel multcancel that x=y

>> Multcancel: [(.x_1:in Nat),(.z_1:in Nat),
>>        (.y_1:in Nat),(multcancel_1:that ((.x_1
>>        * .z_1) = (.y_1 * .z_1))) => (---:that
>>        (.x_1 = .y_1))]
>>   {move 0}
```

adding variations of things in the previous block needed later

```
Lestrade execution:

clearcurrent

open

     declare x in Nat

>>      x: in Nat {move 2}


     declare y in Nat

>>      y: in Nat {move 2}


     declare z in Nat

>>      z: in Nat {move 2}


     declare eqev1 that (x+y)=x+z
```

```
>>      eqev1: that ((x + y) = (x + z)) {move
>>        2}


     define line1 eqev1 : Subs (Commadd x y,eqev1)

>>      line1: [(.x_1:in Nat),(.y_1:in Nat),
>>             (.z_1:in Nat),(eqev1_1:that ((.x_1
>>             + .y_1) = (.x_1 + .z_1))) => (((.x_1
>>             Commadd .y_1) Subs eqev1_1):that
>>             ((.y_1 + .x_1) = (.x_1 + .z_1)))]
>>        {move 1}


     define line2 eqev1 :  Subs (Commadd x z,line1 eqev1)

>>      line2: [(.x_1:in Nat),(.y_1:in Nat),
>>             (.z_1:in Nat),(eqev1_1:that ((.x_1
>>             + .y_1) = (.x_1 + .z_1))) => (((.x_1
>>             Commadd .z_1) Subs line1(eqev1_1)):
>>             that ((.y_1 + .x_1) = (.z_1 + .x_1)))]
>>        {move 1}


     define line3 eqev1 : Addcancel (line2 eqev1)

>>      line3: [(.x_1:in Nat),(.y_1:in Nat),
>>             (.z_1:in Nat),(eqev1_1:that ((.x_1
>>             + .y_1) = (.x_1 + .z_1))) => (Addcancel(line2(eqev1_1)):
>>             that (.y_1 = .z_1))]
>>        {move 1}


     close

declare X in Nat
```

```
>> X: in Nat {move 1}


declare Y in Nat

>> Y: in Nat {move 1}


declare Z in Nat

>> Z: in Nat {move 1}


declare Eqev1 that (X+Y)=(X+Z)

>> Eqev1: that ((X + Y) = (X + Z)) {move 1}


define  Addcancel2 Eqev1 :  line3 Eqev1

>> Addcancel2: [(.X_1:in Nat),(.Y_1:in Nat),
>>        (.Z_1:in Nat),(Eqev1_1:that ((.X_1 +
>>        .Y_1) = (.X_1 + .Z_1))) => (Addcancel(((.X_1
>>        Commadd .Z_1) Subs ((.X_1 Commadd .Y_1)
>>        Subs Eqev1_1))):that (.Y_1 = .Z_1))]
>>   {move 0}
```

A further block of arithmetic axioms (which can actually be presented as theorems with additional work) concern order.

```
Lestrade execution:

clearcurrent

declare x in Nat
```

```
>> x: in Nat {move 1}


declare y in Nat

>> y: in Nat {move 1}


declare z in Nat

>> z: in Nat {move 1}


define < x y : Exists [z=>(x+z)=y]

>> <: [(x_1:in Nat),(y_1:in Nat) => (Exists([(z_2:
>>             in Nat) => (((x_1 + z_2) = y_1):
>>             prop)])
>>        :prop)]
>>   {move 0}


define > x y : y < x

>> >: [(x_1:in Nat),(y_1:in Nat) => ((y_1 <
>>        x_1):prop)]
>>   {move 0}


define <= x y: (x<y) V x=y

>> <=: [(x_1:in Nat),(y_1:in Nat) => (((x_1
>>        < y_1) V (x_1 = y_1)):prop)]
>>   {move 0}


define >= x y: (x>y)V x=y
```

```
>> >=: [(x_1:in Nat),(y_1:in Nat) => (((x_1
>>        > y_1) V (x_1 = y_1)):prop)]
>>   {move 0}


declare less1 that x<y

>> less1: that (x < y) {move 1}


declare less2 that y<z

>> less2: that (y < z) {move 1}


declare less3 that (x+z)<y+z

>> less3: that ((x + z) < (y + z)) {move 1}


declare less4 that (x*z)<y*z

>> less4: that ((x * z) < (y * z)) {move 1}


postulate Transless less1 less2 that x<z

>> Transless: [(.x_1:in Nat),(.y_1:in Nat),(less1_1:
>>        that (.x_1 < .y_1)),(.z_1:in Nat),(less2_1:
>>        that (.y_1 < .z_1)) => (---:that (.x_1
>>        < .z_1))]
>>   {move 0}


postulate Addmono1 less3 that x<y

>> Addmono1: [(.x_1:in Nat),(.z_1:in Nat),(.y_1:
>>        in Nat),(less3_1:that ((.x_1 + .z_1)
```

```
>>        < (.y_1 + .z_1))) => (---:that (.x_1
>>        < .y_1))]
>>   {move 0}


postulate Addmono2 z less1 that  (x+z)<y+z

>> Addmono2: [(z_1:in Nat),(.x_1:in Nat),(.y_1:
>>        in Nat),(less1_1:that (.x_1 < .y_1))
>>        => (---:that ((.x_1 + z_1) < (.y_1 +
>>        z_1)))]
>>   {move 0}


postulate Multmono1 less4 that x<y

>> Multmono1: [(.x_1:in Nat),(.z_1:in Nat),(.y_1:
>>        in Nat),(less4_1:that ((.x_1 * .z_1)
>>        < (.y_1 * .z_1))) => (---:that (.x_1
>>        < .y_1))]
>>   {move 0}


postulate Multmono2 z less1 that (x*z)<y*z

>> Multmono2: [(z_1:in Nat),(.x_1:in Nat),(.y_1:
>>        in Nat),(less1_1:that (.x_1 < .y_1))
>>        => (---:that ((.x_1 * z_1) < (.y_1 *
>>        z_1)))]
>>   {move 0}


postulate Trich1 x y :  that (x=y) V (x<y) V (y<x)

>> Trich1: [(x_1:in Nat),(y_1:in Nat) => (---:
>>        that ((x_1 = y_1) V ((x_1 < y_1) V (y_1
>>        < x_1))))]
>>   {move 0}
```

```
postulate Trich2 x y : that ~((x<y) & (y<x))

>> Trich2: [(x_1:in Nat),(y_1:in Nat) => (---:
>>        that ~(((x_1 < y_1) & (y_1 < x_1))))]
>>   {move 0}
```

We put the well-ordering principle in its own box. The exact form of this may be modified as we work with it.

```
Lestrade execution:

clearcurrent

declare x in Nat

>> x: in Nat {move 1}


declare y in Nat

>> y: in Nat {move 1}


declare z in Nat

>> z: in Nat {move 1}


declare pred [x => prop]

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

```
declare existsev that Exists pred

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


postulate Wop existsev\
    that Exists [y => (pred y) & Forall[z => (z<y) -> ~pred z]]

>> Wop: [(.pred_1:[(x_2:in Nat) => (---:prop)]),
>>        (existsev_1:that Exists(.pred_1)) =>
>>        (---:that Exists([(y_3:in Nat) => ((.pred_1(y_3)
>>             & Forall([(z_4:in Nat) => (((z_4
>>                  < y_3) -> ~(.pred_1(z_4))):
>>                  prop)]))
>>             :prop)]))
>>        ]
>>   {move 0}
```

Now we need to work on subtraction, division, and the Euclidean algorithm.

Here I'll try to work out the definition of subtraction.

```
Lestrade execution:

clearcurrent

declare x in Nat

>> x: in Nat {move 1}


declare y in Nat

>> y: in Nat {move 1}
```

```
declare orderev that x>y

>> orderev: that (x > y) {move 1}


open

     declare z1 in Nat

>>       z1: in Nat {move 2}


     declare z2 in Nat

>>       z2: in Nat {move 2}


     declare u in Nat

>>       u: in Nat {move 2}


     declare testev1 that (y+z1) = x

>>       testev1: that ((y + z1) = x) {move 2}


     declare testev2 that (y+z2) = x

>>       testev2: that ((y + z2) = x) {move 2}


     define testev z1 z2  testev1 testev2 : \
         Addcancel2(Subs (Symmeq testev2,testev1))

>>       testev: [(z1_1:in Nat),(z2_1:in Nat),
>>              (testev1_1:that ((y + z1_1) = x)),
>>              (testev2_1:that ((y + z2_1) = x))
```

```
>>             => (Addcancel2((Symmeq(testev2_1)
>>             Subs testev1_1)):that (z1_1 = z2_1))]
>>        {move 1}


     close

define Subtract1 orderev:  The (orderev,testev)

>> Subtract1: [(.x_1:in Nat),(.y_1:in Nat),(orderev_1:
>>        that (.x_1 > .y_1)) => ((orderev_1 The
>>        [(z1_3:in Nat),(z2_3:in Nat),(testev1_3:
>>             that ((.y_1 + z1_3) = .x_1)),(testev2_3:
>>             that ((.y_1 + z2_3) = .x_1)) =>
>>             (Addcancel2((Symmeq(testev2_3)
>>             Subs testev1_3)):that (z1_3 = z2_3))])
>>        :in Nat)]
>>   {move 0}


define Subtractax orderev:  \
    propfix((y+Subtract1 orderev) = x,\
    Theax(orderev,testev))

>> Subtractax: [(.x_1:in Nat),(.y_1:in Nat),
>>        (orderev_1:that (.x_1 > .y_1)) => ((((.y_1
>>        + Subtract1(orderev_1)) = .x_1) propfix
>>        (orderev_1 Theax [(z1_3:in Nat),(z2_3:
>>             in Nat),(testev1_3:that ((.y_1
>>             + z1_3) = .x_1)),(testev2_3:that
>>             ((.y_1 + z2_3) = .x_1)) => (Addcancel2((Symmeq(testev2_3)
>>             Subs testev1_3)):that (z1_3 = z2_3))]))
>>        :that ((.y_1 + Subtract1(orderev_1))
>>        = .x_1))]
>>   {move 0}
```

The preceding block defines $x - y$ as a function of a single explicit argument, evidence that $x > y$ (from this the arguments x and y are immediately deduced). The next block defines the absolute difference $|x - y|$ as a function of evidence that x and y are distinct.

```
Lestrade execution:

clearcurrent

declare x in Nat

>> x: in Nat {move 1}


declare y in Nat

>> y: in Nat {move 1}


declare diffev that x =/= y

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


define Lessormore diffev :  Ds1 Trich1 x y diffev

>> Lessormore: [(.x_1:in Nat),(.y_1:in Nat),
>>        (diffev_1:that (.x_1 =/= .y_1)) => (((.x_1
>>        Trich1 .y_1) Ds1 diffev_1):that ((.x_1
>>        < .y_1) V (.y_1 < .x_1)))]
>>   {move 0}
```