Nahas Coq Tutorial in Lestrade (sort of)

Randall Holmes

Revised 1/31/2018. Much more use of Lestrade rewriting, so that it is computing with inductive types in much the same way Coq does, though with a different theoretical underpinning. Proved up to commutativity of or! 2/1/2018 added ability to extract information from or proofs.

The intention here is to compare Coq and Lestrade by paralleling the reasoning in Nahas's tutorial as far as I can in Lestrade. It is not guaranteed that I will get far!

Built-ins of Coq must be declared in Lestrade.

```
declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}


clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop
```

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


construct -> A B prop

>> ->: [(A_1:prop),(B_1:prop) => (---:prop)]
>>   {move 0}


declare aa that A

>> aa: that A {move 1}


declare ded [aa => that B]

>> ded: [(aa_1:that A) => (---:that B)]
>>   {move 1}


construct Deduction ded that A -> B

>> Deduction: [(.A_1:prop),(.B_1:prop),(ded_1:
>>        [(aa_2:that .A_1) => (---:that .B_1)])
>>        => (---:that (.A_1 -> .B_1))]
>>   {move 0}


declare cc that A -> B

>> cc: that (A -> B) {move 1}


declare aa2 that A

>> aa2: that A {move 1}
```

```
construct Mp cc aa2 that B

>> Mp: [(.A_1:prop),(.B_1:prop),(cc_1:that (.A_1
>>        -> .B_1)),(aa2_1:that .A_1) => (---:
>>        that .B_1)]
>>   {move 0}


clearcurrent

declare A prop

>> A: prop {move 1}


open

     declare aa that A

>>      aa: that A {move 2}


     define selfimp aa:aa

>>      selfimp: [(aa_1:that A) => (aa_1:that
>>             A)]
>>        {move 1}


     close

define myfirstproof A: Deduction selfimp

>> myfirstproof: [(A_1:prop) => (Deduction([(aa_2:
>>             that A_1) => (aa_2:that A_1)])
>>        :that (A_1 -> A_1))]
>>   {move 0}
```

Now we try the next proof.

```
clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}


open

     declare aa that A

>>      aa: that A {move 2}


     open

          declare cc that A -> B

>>           cc: that (A -> B) {move 3}


          define innerproof cc :  Mp cc aa

>>           innerproof: [(cc_1:that (A -> B))
>>                  => ((cc_1 Mp aa):that B)]
>>             {move 2}
```

This line corresponds to the pose command in the Coq proof.

```
        close

    define outerproof0 aa : Deduction innerproof

>>      outerproof0: [(aa_1:that A) => (Deduction([(cc_2:
>>                  that (A -> B)) => ((cc_2 Mp
>>                  aa_1):that B)])
>>              :that ((A -> B) -> B))]
>>        {move 1}


    close

define outerproof A B : Deduction outerproof0

>> outerproof: [(A_1:prop),(B_1:prop) => (Deduction([(aa_2:
>>             that A_1) => (Deduction([(cc_3:
>>                  that (A_1 -> B_1)) => ((cc_3
>>                  Mp aa_2):that B_1)])
>>             :that ((A_1 -> B_1) -> B_1))])
>>        :that (A_1 -> ((A_1 -> B_1) -> B_1)))]
>>   {move 0}
```

The next example doesn't work for us because we do not have automated goal directed proof in Lestrade. In general, Lestrade is more verbose than Coq, and lacks tactics. It's not that we can't prove the same theorems. And we can support a goal directed style of proof (the comments show our goals). Automation of goal directed reasoning is desirable!

```
% our goal is to prove  A -> (A -> B) -> (B -> C) -> C

clearcurrent

declare A prop
```

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

declare B prop

>> B: prop {move 1}

declare C prop

>> C: prop {move 1}

open

     declare aa that A

>>       aa: that A {move 2}

     % our goal is now # (A -> B) -> (B -> C) -> C

     open

          declare dd that A -> B

>>           dd: that (A -> B) {move 3}

          % our goal is now #(B -> C) -> C

          open

               declare ee that B->C

>>                ee: that (B -> C) {move 4}
```

```
              % now our goal is C.  To get C we need B.  To get B we need A.

              define ff:Mp dd aa

>>                ff: [((dd Mp aa):that B)]
>>                  {move 3}


              define gg ee: Mp ee ff

>>                gg: [(ee_1:that (B -> C))
>>                       => ((ee_1 Mp ff):that
>>                       C)]
>>                  {move 3}


              close

          define hh dd:  Deduction gg

>>           hh: [(dd_1:that (A -> B)) => (Deduction([(ee_2:
>>                       that (B -> C)) => ((ee_2
>>                       Mp (dd_1 Mp aa)):that
>>                       C)])
>>                  :that ((B -> C) -> C))]
>>             {move 2}


          close

     define ii aa: Deduction hh

>>      ii: [(aa_1:that A) => (Deduction([(dd_2:
>>                  that (A -> B)) => (Deduction([(ee_3:
>>                       that (B -> C)) => ((ee_3
>>                       Mp (dd_2 Mp aa_1)):that
>>                       C)])
```

```
>>                  :that ((B -> C) -> C))])
>>             :that ((A -> B) -> ((B -> C) ->
>>             C)))]
>>        {move 1}


     close

define anotherproof A B C:Deduction ii

>> anotherproof: [(A_1:prop),(B_1:prop),(C_1:
>>        prop) => (Deduction([(aa_2:that A_1)
>>             => (Deduction([(dd_3:that (A_1
>>                  -> B_1)) => (Deduction([(ee_4:
>>                       that (B_1 -> C_1)) =>
>>                       ((ee_4 Mp (dd_3 Mp aa_2)):
>>                       that C_1)])
>>                  :that ((B_1 -> C_1) -> C_1))])
>>             :that ((A_1 -> B_1) -> ((B_1 ->
>>             C_1) -> C_1)))])
>>        :that (A_1 -> ((A_1 -> B_1) -> ((B_1
>>        -> C_1) -> C_1))))]
>>   {move 0}
```

We try a bigger theorem in the forward style (but commented suitably for backward).

```
% Theorem:  A -> (A -> B) -> (A -> B -> C) -> C

clearcurrent

declare A prop

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

```
declare B prop

>> B: prop {move 1}


declare C prop

>> C: prop {move 1}


open

     declare aa that A

>>      aa: that A {move 2}


     % now our goal is (A -> B) -> (A -> B -> C) -> C

     open

          declare dd that A->B

>>           dd: that (A -> B) {move 3}


          % now our goal is (A -> B -> C) -> C

          open

               declare ee that A->(B->C)

>>                ee: that (A -> (B -> C)) {move
>>                  4}


               define ff ee: Mp  ee aa
```

```
>>                ff: [(ee_1:that (A -> (B ->
>>                       C))) => ((ee_1 Mp aa):
>>                       that (B -> C))]
>>                  {move 3}


             define bb ee: Mp dd aa

>>                bb: [(ee_1:that (A -> (B ->
>>                       C))) => ((dd Mp aa):that
>>                       B)]
>>                  {move 3}


             define cc ee: Mp (ff ee, bb ee)

>>                cc: [(ee_1:that (A -> (B ->
>>                       C))) => ((ff(ee_1) Mp
>>                       bb(ee_1)):that C)]
>>                  {move 3}


             close

          define gg dd:Deduction cc

>>           gg: [(dd_1:that (A -> B)) => (Deduction([(ee_2:
>>                       that (A -> (B -> C)))
>>                       => (((ee_2 Mp aa) Mp
>>                       (dd_1 Mp aa)):that C)])
>>                  :that ((A -> (B -> C)) ->
>>                  C))]
>>             {move 2}


          close

     define hh aa :Deduction gg
```

```
>>      hh: [(aa_1:that A) => (Deduction([(dd_2:
>>                    that (A -> B)) => (Deduction([(ee_3:
>>                         that (A -> (B -> C)))
>>                         => (((ee_3 Mp aa_1) Mp
>>                         (dd_2 Mp aa_1)):that
>>                         C)])
>>                    :that ((A -> (B -> C)) ->
>>                    C))])
>>               :that ((A -> B) -> ((A -> (B ->
>>               C)) -> C)))]
>>         {move 1}


     close

define hugeproof A B C: Deduction hh

>> hugeproof: [(A_1:prop),(B_1:prop),(C_1:prop)
>>         => (Deduction([(aa_2:that A_1) => (Deduction([(dd_3:
>>                    that (A_1 -> B_1)) => (Deduction([(ee_4:
>>                         that (A_1 -> (B_1 ->
>>                         C_1))) => (((ee_4 Mp
>>                         aa_2) Mp (dd_3 Mp aa_2)):
>>                         that C_1)])
>>                    :that ((A_1 -> (B_1 -> C_1))
>>                    -> C_1))])
>>               :that ((A_1 -> B_1) -> ((A_1 ->
>>               (B_1 -> C_1)) -> C_1)))])
>>         :that (A_1 -> ((A_1 -> B_1) -> ((A_1
>>         -> (B_1 -> C_1)) -> C_1))))]
>>    {move 0}
```

One might try comparing the Lestrade proof object with the proof exhibited in the Nahas example. There is enough here to see that the philosophy is similar. Things are a little different because proofs of implications are not

actually functions in the Lestrade logic (the constructor `Deduction` is applied to the relevant function to yield the proof), and applications of modus ponens are similarly not exactly function applications.

Other global differences have to do with the fact that we do not quantify over `prop` in Lestrade (we *could* but our style discourages it), so I have been generating functions from proposition names to proofs of particular propositions rather than proofs of universally quantified statements over propositions.

```
% example for Morgan

% Prove (A-> (B-> C)) -> (B ->(A -> C))

clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}


declare C prop

>> C: prop {move 1}


open

     declare line1 that A -> (B -> C)

>>       line1: that (A -> (B -> C)) {move 2}


     % goal is now B -> (A -> C)
```

```
     open

          declare line2 that B

>>           line2: that B {move 3}


          % goal is A -> C

          open

               declare line3 that A

>>              line3: that A {move 4}


               % goal is C

               define line4 line3 : Mp line1 line3

>>              line4: [(line3_1:that A) =>
>>                     ((line1 Mp line3_1):that
>>                     (B -> C))]
>>                {move 3}


               define line5 line3 : Mp(line4 line3, line2)

>>              line5: [(line3_1:that A) =>
>>                     ((line4(line3_1) Mp line2):
>>                     that C)]
>>                {move 3}


               close

          define line6 line2 : Deduction line5
```

```
>>           line6: [(line2_1:that B) => (Deduction([(line3_2:
>>                       that A) => (((line1 Mp
>>                       line3_2) Mp line2_1):
>>                       that C)])
>>                  :that (A -> C))]
>>             {move 2}


          close

     define line7 line1 : Deduction line6

>>      line7: [(line1_1:that (A -> (B -> C)))
>>             => (Deduction([(line2_2:that B)
>>                  => (Deduction([(line3_3:that
>>                       A) => (((line1_1 Mp line3_3)
>>                       Mp line2_2):that C)])
>>                  :that (A -> C))])
>>             :that (B -> (A -> C)))]
>>        {move 1}


     close

define Morganexample A B C : Deduction line7

>> Morganexample: [(A_1:prop),(B_1:prop),(C_1:
>>        prop) => (Deduction([(line1_2:that (A_1
>>             -> (B_1 -> C_1))) => (Deduction([(line2_3:
>>                  that B_1) => (Deduction([(line3_4:
>>                       that A_1) => (((line1_2
>>                       Mp line3_4) Mp line2_3):
>>                       that C_1)])
>>                  :that (A_1 -> C_1))])
>>             :that (B_1 -> (A_1 -> C_1)))])
>>        :that ((A_1 -> (B_1 -> C_1)) -> (B_1
>>        -> (A_1 -> C_1))))]
```

```
>>   {move 0}
```

Now we consider inductive types. Lestrade does not have disjoint unions or induction types as a primitive, so we must be more verbose.

```
clearcurrent

construct False prop

>> False: prop {move 0}


declare absurd that False

>> absurd: that False {move 1}


declare absurdfun [absurd => prop]

>> absurdfun: [(absurd_1:that False) => (---:
>>        prop)]
>>   {move 1}


declare absurd2 that False

>> absurd2: that False {move 1}


construct Absurdity absurdfun, absurd2 that absurdfun absurd2

>> Absurdity: [(absurdfun_1:[(absurd_2:that
>>             False) => (---:prop)]),
>>        (absurd2_1:that False) => (---:that
>>        absurdfun_1(absurd2_1))]
>>   {move 0}
```

False is a statement with no proofs. If we are given any property of proofs of False and any proof of False we get a proof that it has this property.

```
construct True prop

>> True: prop {move 0}


construct thus that True

>> thus: that True {move 0}


declare thus2 that True

>> thus2: that True {move 1}


declare thusfun [thus2 => prop]

>> thusfun: [(thus2_1:that True) => (---:prop)]
>>   {move 1}


declare indeed that thusfun thus

>> indeed: that thusfun(thus) {move 1}


declare thus3 that True

>> thus3: that True {move 1}


construct Onlythus thusfun, indeed thus3 that thusfun thus3
```

```
>> Onlythus: [(thusfun_1:[(thus2_2:that True)
>>             => (---:prop)]),
>>        (indeed_1:that thusfun_1(thus)),(thus3_1:
>>        that True) => (---:that thusfun_1(thus3_1))]
>>   {move 0}
```

`True` is a statement with only one proof, `thus`. Given any property of proofs of `True`, a proof that `thus` has this property, and a proof of `True`, we get a proof that the last given proof of `True` has this property.

```
construct bool type

>> bool: type {move 0}


construct true in bool

>> true: in bool {move 0}


construct false in bool

>> false: in bool {move 0}


declare b in bool

>> b: in bool {move 1}


declare boolfun [b=>prop]

>> boolfun: [(b_1:in bool) => (---:prop)]
>>   {move 1}
```

```
declare boolcase1 that boolfun true

>> boolcase1: that boolfun(true) {move 1}


declare boolcase2 that boolfun false

>> boolcase2: that boolfun(false) {move 1}


declare b2 in bool

>> b2: in bool {move 1}


construct Oneortheother boolfun, boolcase1 boolcase2 b2 that boolfun b2

>> Oneortheother: [(boolfun_1:[(b_2:in bool)
>>             => (---:prop)]),
>>        (boolcase1_1:that boolfun_1(true)),(boolcase2_1:
>>        that boolfun_1(false)),(b2_1:in bool)
>>        => (---:that boolfun_1(b2_1))]
>>   {move 0}
```

bool is a type containing the objects true and false and no other objects. Any property of bools which holds of both true and false holds of any element of bool.

Now we try proving the Nahas theorems.

```
clearcurrent

declare p prop

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

```
declare pp that p

>> pp: that p {move 1}


define fixpropform p pp:pp

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

`fixpropform(p,pp)` is defined as `pp` and will be typed as `that p` (literally) if `that p` is equivalent via definition or rewriting to the type of `pp` found by the type checker. This helps ensure that defined (or computed) types are displayed as desired.

```
define Truecanbeproven : thus

>> Truecanbeproven: [(thus:that True)]
>>   {move 0}


define ~p : p -> False

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


% we aim to prove ~False

open

     declare absurd that False

>>      absurd: that False {move 2}
```

```
     open

          declare absurd2 that False

>>           absurd2: that False {move 3}


          define absurdid absurd2 : False

>>           absurdid: [(absurd2_1:that False)
>>                  => (False:prop)]
>>             {move 2}


          close

     %%  if the following line is used to define absurdid
     %%  instead of the block above a bug ensues.  This is weird.
     %% This suggests that more coercion of expansion is needed.
     % Or some kind of fix to matching.

     % define absurdid absurd : False

     define absurdconc absurd: Absurdity absurdid, absurd

>>      absurdconc: [(absurd_1:that False) =>
>>             (Absurdity([(absurd2_2:that False)
>>                  => (False:prop)]
>>             ,absurd_1):that False)]
>>        {move 1}


     % for error diagnosis

     define absurdid2 absurd : False
```

```
>>      absurdid2: [(absurd_1:that False) =>
>>             (False:prop)]
>>        {move 1}


     define absurdconc2 absurd: Absurdity absurdid2, absurd

>>      absurdconc2: [(absurd_1:that False)
>>             => (Absurdity(absurdid2,absurd_1):
>>             that absurdid2(absurd_1))]
>>        {move 1}


     close

declare nono that False

>> nono: that False {move 1}


define test nono : absurdconc2 nono

>> test: [(nono_1:that False) => (Absurdity([(absurd_2:
>>             that False) => (False:prop)]
>>        ,nono_1):that False)]
>>   {move 0}


%% we can evaluate absurdconc2
% but we cannot use it as an argument to Deduction for some reason.  Matching?

define Falsecannotbeproven0 : Deduction absurdconc

>> Falsecannotbeproven0: [(Deduction([(absurd_1:
>>             that False) => (Absurdity([(absurd2_2:
>>                  that False) => (False:prop)]
>>             ,absurd_1):that False)])
>>        :that (False -> False))]
```

```
>>   {move 0}


%% fixpropform gives us the desired form
% of the statement proved, unlike the previous

define Falsecannotbeproven : fixpropform(~False,Deduction absurdconc)

>> Falsecannotbeproven: [((~(False) fixpropform
>>        Deduction([(absurd_1:that False) =>
>>             (Absurdity([(absurd2_2:that False)
>>                  => (False:prop)]
>>             ,absurd_1):that False)]))
>>        :that ~(False))]
>>   {move 0}


%% Tracing the errors, I conclude that it is not really a bug.
%% One really does want to force expansion:  there is no reason
%% to expect a constant to match a term with bound variables in it,
%which is what the matching algorithm is trying to do when this crashes.

% define Falsecannotbeproven2 : fixpropform(~False,Deduction absurdconc2)

clearcurrent

% prove that True implies True

declare tt that True

>> tt: that True {move 1}


declare absurd that False

>> absurd: that False {move 1}
```

```
define zorch tt :thus

>> zorch: [(tt_1:that True) => (thus:that True)]
>>   {move 0}


define Trueimpliestrue :  Deduction zorch

>> Trueimpliestrue: [(Deduction(zorch):that
>>        (True -> True))]
>>   {move 0}


define zorch2 absurd:thus

>> zorch2: [(absurd_1:that False) => (thus:that
>>        True)]
>>   {move 0}


define Falseimpliestrue:  Deduction zorch2

>> Falseimpliestrue: [(Deduction(zorch2):that
>>        (False -> True))]
>>   {move 0}


% I did it in the way they deprecate

define zorch3 absurd:  absurd

>> zorch3: [(absurd_1:that False) => (absurd_1:
>>        that False)]
>>   {move 0}


define Falseimpliesfalse : Deduction zorch3
```

```
>> Falseimpliesfalse: [(Deduction(zorch3):that
>>        (False -> False))]
>>   {move 0}


% true does not imply false

open

     declare wow that True->False

>>      wow: that (True -> False) {move 2}


     define wowwee wow : Mp wow thus

>>      wowwee: [(wow_1:that (True -> False))
>>             => ((wow_1 Mp thus):that False)]
>>        {move 1}


     close

define Truedoesnotimplyfalse : fixpropform(~(True->False),Deduction wowwee)

>> Truedoesnotimplyfalse: [((~((True -> False))
>>        fixpropform Deduction([(wow_1:that (True
>>             -> False)) => ((wow_1 Mp thus):
>>             that False)]))
>>        :that ~((True -> False)))]
>>   {move 0}


%% We codify negation introduction.
%% Note use of fixpropform to get the result
%to display as negative rather than conditional.

clearcurrent
```

```
declare p prop

>> p: prop {move 1}


declare pp that p

>> pp: that p {move 1}


declare contra [pp => that False]

>> contra: [(pp_1:that p) => (---:that False)]
>>   {move 1}


define Negintro contra:fixpropform(~p, Deduction contra)

>> Negintro: [(.p_1:prop),(contra_1:[(pp_2:that
>>              .p_1) => (---:that False)])
>>        => ((~(.p_1) fixpropform Deduction(contra_1)):
>>        that ~(.p_1))]
>>   {move 0}


% we aim to prove A -> (~A -> C)

clearcurrent

declare A prop

>> A: prop {move 1}


declare C prop

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

```
open

     declare aa that A

>>       aa: that A {move 2}


     % now our goal is ~A -> C

     open

          declare notaa that ~A

>>           notaa: that ~(A) {move 3}


          define line2 notaa :  Mp notaa aa

>>           line2: [(notaa_1:that ~(A)) =>
>>                  ((notaa_1 Mp aa):that False)]
>>             {move 2}


          open

               declare absurd2 that False

>>                absurd2: that False {move
>>                  4}


               define cc absurd2:C

>>                cc: [(absurd2_1:that False)
>>                       => (C:prop)]
>>                  {move 3}
```

```
               close

          define gotcha notaa:Absurdity cc, line2 notaa

>>           gotcha: [(notaa_1:that ~(A)) =>
>>                  (Absurdity([(absurd2_2:that
>>                       False) => (C:prop)]
>>                  ,line2(notaa_1)):that C)]
>>             {move 2}


          close

     define line3 aa : Deduction gotcha

>>      line3: [(aa_1:that A) => (Deduction([(notaa_2:
>>                  that ~(A)) => (Absurdity([(absurd2_3:
>>                       that False) => (C:prop)]
>>                  ,(notaa_2 Mp aa_1)):that C)])
>>             :that (~(A) -> C))]
>>        {move 1}


     close

define Absurd2 A C: Deduction line3

>> Absurd2: [(A_1:prop),(C_1:prop) => (Deduction([(aa_2:
>>             that A_1) => (Deduction([(notaa_3:
>>                  that ~(A_1)) => (Absurdity([(absurd2_4:
>>                       that False) => (C_1:prop)]
>>                  ,(notaa_3 Mp aa_2)):that C_1)])
>>             :that (~(A_1) -> C_1))])
>>        :that (A_1 -> (~(A_1) -> C_1)))]
>>   {move 0}
```

```
% now we prove theorems about bool

% we need definitions by cases on bool

clearcurrent

declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


declare b in bool

>> b: in bool {move 1}


construct Boolmatch p q b : prop

>> Boolmatch: [(p_1:prop),(q_1:prop),(b_1:in
>>        bool) => (---:prop)]
>>   {move 0}


%% these declarations force
%% equality of Boolmatch p q true with p and Boolmatch p q false with q,
%% and moreover force calculations with
%% these equations where appropriate:  Lestrade will
%%  simplify the left hand terms
% in expanding definitions and verifying matches.

%% This has the effect of creating something equivalent to
%% Coq simp, which gets called automatically
```

```
% whereever usable.

rewritec Boolmatchtrue p q, Boolmatch p q true, p

>> Boolmatchtrue'': [(Boolmatchtrue'''_1:prop)
>>        => (---:prop)]
>>   {move 1}



>> Boolmatchtrue': that Boolmatchtrue''(Boolmatch(p,
>>   q,true)) {move 1}



>> Boolmatchtrue: [(p_1:prop),(q_1:prop),(Boolmatchtrue''_1:
>>        [(Boolmatchtrue'''_2:prop) => (---:prop)]),
>>        (Boolmatchtrue'_1:that Boolmatchtrue''_1(Boolmatch(p_1,
>>        q_1,true))) => (---:that Boolmatchtrue''_1(p_1))]
>>   {move 0}


rewritec Boolmatchfalse p q, Boolmatch p, q, false, q

>> Boolmatchfalse'': [(Boolmatchfalse'''_1:prop)
>>        => (---:prop)]
>>   {move 1}



>> Boolmatchfalse': that Boolmatchfalse''(Boolmatch(p,
>>   q,false)) {move 1}



>> Boolmatchfalse: [(p_1:prop),(q_1:prop),(Boolmatchfalse''_1:
>>        [(Boolmatchfalse'''_2:prop) => (---:
>>             prop)]),
```

```
>>        (Boolmatchfalse'_1:that Boolmatchfalse''_1(Boolmatch(p_1,
>>        q_1,false))) => (---:that Boolmatchfalse''_1(q_1))]
>>   {move 0}


declare boolfact1 that Boolmatch p q true

>> boolfact1: that Boolmatch(p,q,true) {move
>>   1}


define Boolmatch1 p q boolfact1:fixpropform(p,boolfact1)

>> Boolmatch1: [(p_1:prop),(q_1:prop),(boolfact1_1:
>>        that Boolmatch(p_1,q_1,true)) => ((p_1
>>        fixpropform boolfact1_1):that p_1)]
>>   {move 0}


declare pp that p

>> pp: that p {move 1}


define Boolmatch2 q pp : fixpropform(Boolmatch p q true,pp)

>> Boolmatch2: [(q_1:prop),(.p_1:prop),(pp_1:
>>        that .p_1) => ((.p_1 fixpropform pp_1):
>>        that Boolmatch(.p_1,q_1,true))]
>>   {move 0}


declare boolfact2 that Boolmatch p q false

>> boolfact2: that Boolmatch(p,q,false) {move
>>   1}
```

```
define Boolmatch3 p q boolfact2 : fixpropform(q,boolfact2)

>> Boolmatch3: [(p_1:prop),(q_1:prop),(boolfact2_1:
>>        that Boolmatch(p_1,q_1,false)) => ((q_1
>>        fixpropform boolfact2_1):that q_1)]
>>   {move 0}


declare qq that q

>> qq: that q {move 1}


define Boolmatch4 p qq : fixpropform(Boolmatch p q false,qq)

>> Boolmatch4: [(p_1:prop),(.q_1:prop),(qq_1:
>>        that .q_1) => ((.q_1 fixpropform qq_1):
>>        that Boolmatch(p_1,.q_1,false))]
>>   {move 0}


define Istrue b : Boolmatch True False b

>> Istrue: [(b_1:in bool) => (Boolmatch(True,
>>        False,b_1):prop)]
>>   {move 0}


define trueistrue  :fixpropform(Istrue true, thus)

>> trueistrue: [((Istrue(true) fixpropform thus):
>>        that Istrue(true))]
>>   {move 0}


% I should prove ~Istrue(false) as well.

declare trueisfalse that Istrue false
```

```
>> trueisfalse: that Istrue(false) {move 1}


define Trueisnotfalse : Negintro [trueisfalse => trueisfalse]

>> Trueisnotfalse: [(Negintro([(trueisfalse_1:
>>             that Istrue(false)) => (trueisfalse_1:
>>             that Istrue(false))])
>>        :that ~(Istrue(false)))]
>>   {move 0}


declare b1 in bool

>> b1: in bool {move 1}


declare b2 in  bool

>> b2: in bool {move 1}


% this is the equality relation on booleans, not his function eqb.

define eqb0 b1 b2 :  Boolmatch (Istrue b1,~(Istrue b1),b2)

>> eqb0: [(b1_1:in bool),(b2_1:in bool) => (Boolmatch(Istrue(b1_1),
>>        ~(Istrue(b1_1)),b2_1):prop)]
>>   {move 0}


clearcurrent

declare T type

>> T: type {move 1}
```

```
declare t1 in T

>> t1: in T {move 1}


declare t2 in T

>> t2: in T {move 1}


% the universal quantifier

declare pred [t1 => prop]

>> pred: [(t1_1:in T) => (---:prop)]
>>   {move 1}


construct Forall pred prop

>> Forall: [(.T_1:type),(pred_1:[(t1_2:in .T_1)
>>             => (---:prop)])
>>        => (---:prop)]
>>   {move 0}


declare uproof that Forall pred

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


declare t3 in T

>> t3: in T {move 1}


construct Uinst uproof t3 that pred t3
```

```
>> Uinst: [(.T_1:type),(.pred_1:[(t1_2:in .T_1)
>>             => (---:prop)]),
>>        (uproof_1:that Forall(.pred_1)),(t3_1:
>>        in .T_1) => (---:that .pred_1(t3_1))]
>>   {move 0}


declare gproof [t1 => that pred t1]

>> gproof: [(t1_1:in T) => (---:that pred(t1_1))]
>>   {move 1}


construct Ugen gproof that Forall pred

>> Ugen: [(.T_1:type),(.pred_1:[(t1_2:in .T_1)
>>             => (---:prop)]),
>>        (gproof_1:[(t1_3:in .T_1) => (---:that
>>             .pred_1(t1_3))])
>>        => (---:that Forall(.pred_1))]
>>   {move 0}


% the case constructor for bool over a general type

declare b in bool

>> b: in bool {move 1}


construct Ifthenelse t1 t2 b : in T

>> Ifthenelse: [(.T_1:type),(t1_1:in .T_1),(t2_1:
>>        in .T_1),(b_1:in bool) => (---:in .T_1)]
>>   {move 0}
```

```
rewritec Ifthenelse1 t1 t2, Ifthenelse (t1, t2, true), t1

>> Ifthenelse1'': [(Ifthenelse1'''_1:in T) =>
>>        (---:prop)]
>>   {move 1}



>> Ifthenelse1': that Ifthenelse1''(Ifthenelse(t1,
>>   t2,true)) {move 1}



>> Ifthenelse1: [(.T_1:type),(t1_1:in .T_1),
>>        (t2_1:in .T_1),(Ifthenelse1''_1:[(Ifthenelse1'''_2:
>>             in .T_1) => (---:prop)]),
>>        (Ifthenelse1'_1:that Ifthenelse1''_1(Ifthenelse(t1_1,
>>        t2_1,true))) => (---:that Ifthenelse1''_1(t1_1))]
>>   {move 0}


rewritec Ifthenelse2 t1 t2, Ifthenelse t1 t2 false t2

>> Ifthenelse2'': [(Ifthenelse2'''_1:in T) =>
>>        (---:prop)]
>>   {move 1}



>> Ifthenelse2': that Ifthenelse2''(Ifthenelse(t1,
>>   t2,false)) {move 1}



>> Ifthenelse2: [(.T_1:type),(t1_1:in .T_1),
>>        (t2_1:in .T_1),(Ifthenelse2''_1:[(Ifthenelse2'''_2:
>>             in .T_1) => (---:prop)]),
>>        (Ifthenelse2'_1:that Ifthenelse2''_1(Ifthenelse(t1_1,
```

```
>>        t2_1,false))) => (---:that Ifthenelse2''_1(t2_1))]
>>   {move 0}


%% enhanced versions of Ifthenelse with the output type depending on the
%% boolean parameter, combined with an Ifthenelse construction for types,
% recommend themselves.

declare b1 in bool

>> b1: in bool {move 1}


declare b2 in bool

>> b2: in bool {move 1}


define eqb b1 b2 : \
   Ifthenelse (Ifthenelse true false b1,Ifthenelse false true b1,b2)

>> eqb: [(b1_1:in bool),(b2_1:in bool) => (Ifthenelse(Ifthenelse(true,
>>        false,b1_1),Ifthenelse(false,true,b1_1),
>>        b2_1):in bool)]
>>   {move 0}


%% the proof here is literally the same as
%% {\tt Trueisnotfalse}:  Lestrade determines by
% computation that it proves the same thing.

define Noteqbtruefalse : fixpropform(~(Istrue(true eqb false)),Trueisnotfalse)

>> Noteqbtruefalse: [((~(Istrue((true eqb false)))
>>        fixpropform Trueisnotfalse):that ~(Istrue((true
>>        eqb false))))]
>>   {move 0}
```

```
define Boolrefl1 :  fixpropform(Istrue(true eqb true),trueistrue)

>> Boolrefl1: [((Istrue((true eqb true)) fixpropform
>>        trueistrue):that Istrue((true eqb true)))]
>>   {move 0}


define Boolrefl2 :  fixpropform(Istrue(false eqb false),trueistrue)

>> Boolrefl2: [((Istrue((false eqb false)) fixpropform
>>        trueistrue):that Istrue((false eqb false)))]
>>   {move 0}


define Boolrefl3 b1 : \
   Oneortheother [b1 => Istrue(b1 eqb b1)] Boolrefl1 Boolrefl2 b1

>> Boolrefl3: [(b1_1:in bool) => (Oneortheother([(b1_2:
>>             in bool) => (Istrue((b1_2 eqb b1_2)):
>>             prop)]
>>        ,Boolrefl1,Boolrefl2,b1_1):that Istrue((b1_1
>>        eqb b1_1)))]
>>   {move 0}


% here I prove a theorem with an actual universal quantifier (over bool)

define Boolrefl : Ugen Boolrefl3

>> Boolrefl: [(Ugen(Boolrefl3):that Forall([(b1_2:
>>             in bool) => (Istrue((b1_2 eqb b1_2)):
>>             prop)]))
>>        ]
>>   {move 0}


% prove thm_eqb_a_t
```

```
declare xxx that Istrue(true)

>> xxx: that Istrue(true) {move 1}


define Eqtrueimptrue1 : \
   fixpropform(Istrue(true eqb true) -> \
   Istrue(true),Deduction [xxx=>trueistrue])

>> Eqtrueimptrue1: [(((Istrue((true eqb true))
>>        -> Istrue(true)) fixpropform Deduction([(xxx_1:
>>             that Istrue(true)) => (trueistrue:
>>             that Istrue(true))]))
>>        :that (Istrue((true eqb true)) -> Istrue(true)))]
>>   {move 0}


declare xxx2 that Istrue(false)

>> xxx2: that Istrue(false) {move 1}


define Eqtrueimptrue2 :  \
   fixpropform(Istrue(true eqb false) ->\
    Istrue(false),Deduction[xxx2=>xxx2])

>> Eqtrueimptrue2: [(((Istrue((true eqb false))
>>        -> Istrue(false)) fixpropform Deduction([(xxx2_1:
>>             that Istrue(false)) => (xxx2_1:
>>             that Istrue(false))]))
>>        :that (Istrue((true eqb false)) -> Istrue(false)))]
>>   {move 0}


define  Eqtrueimptrue : \
   Ugen(Oneortheother ([b1 => Istrue(b1 eqb true) \
   -> Istrue(b1)], Eqtrueimptrue1, Eqtrueimptrue2))
```

```
>> Eqtrueimptrue: [(Ugen([(b2_1:in bool) =>
>>             (Oneortheother([(b1_2:in bool)
>>                  => ((Istrue((b1_2 eqb true))
>>                  -> Istrue(b1_2)):prop)]
>>             ,Eqtrueimptrue1,Eqtrueimptrue2,
>>             b2_1):that (Istrue((b2_1 eqb true))
>>             -> Istrue(b2_1)))])
>>        :that Forall([(b2_3:in bool) => ((Istrue((b2_3
>>             eqb true)) -> Istrue(b2_3)):prop)]))
>>        ]
>>   {move 0}
```

Now we need to implement the "inductive" definition of or in the Coq document.

```
clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}


construct v A B prop

>> v: [(A_1:prop),(B_1:prop) => (---:prop)]
>>   {move 0}


declare aa that A
```

```
>> aa: that A {move 1}


declare bb that B

>> bb: that B {move 1}


construct Orl B aa that A v B

>> Orl: [(B_1:prop),(.A_1:prop),(aa_1:that .A_1)
>>        => (---:that (.A_1 v B_1))]
>>   {move 0}


construct Orr A bb that A v B

>> Orr: [(A_1:prop),(.B_1:prop),(bb_1:that .B_1)
>>        => (---:that (A_1 v .B_1))]
>>   {move 0}


declare aorb that A v B

>> aorb: that (A v B) {move 1}


declare orproofprop [aorb => prop]

>> orproofprop: [(aorb_1:that (A v B)) => (---:
>>        prop)]
>>   {move 1}


declare leftorprop [aa => that orproofprop (Orl B aa)]

>> leftorprop: [(aa_1:that A) => (---:that orproofprop((B
>>        Orl aa_1)))]
```

```
>>   {move 1}


declare rightorprop [bb => that orproofprop (Orr A bb)]

>> rightorprop: [(bb_1:that B) => (---:that
>>        orproofprop((A Orr bb_1)))]
>>   {move 1}


declare aorb2 that A v B

>> aorb2: that (A v B) {move 1}


construct Orproofsexhausted orproofprop, \
   leftorprop, rightorprop, aorb2 that orproofprop aorb2

>> Orproofsexhausted: [(.A_1:prop),(.B_1:prop),
>>        (orproofprop_1:[(aorb_2:that (.A_1 v
>>             .B_1)) => (---:prop)]),
>>        (leftorprop_1:[(aa_3:that .A_1) => (---:
>>             that orproofprop_1((.B_1 Orl aa_3)))]),
>>        (rightorprop_1:[(bb_4:that .B_1) =>
>>             (---:that orproofprop_1((.A_1 Orr
>>             bb_4)))]),
>>        (aorb2_1:that (.A_1 v .B_1)) => (---:
>>        that orproofprop_1(aorb2_1))]
>>   {move 0}


%% this says that any property which holds of
%% all Orl B aa's and all Orr A bb's holds of all proofs of A v B;
%% the intention is that there are no other proofs.  Some functions
%% which extract information from the innards of Orl and Orr
% constructions are needed to cash this out.

% our goal is to prove A -> A v B
```

```
clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}


open

     declare aa that A

>>       aa: that A {move 2}


     define aorbx1 aa : Orl B aa

>>       aorbx1: [(aa_1:that A) => ((B Orl aa_1):
>>              that (A v B))]
>>         {move 1}


     close

define Leftor A B :  Deduction aorbx1

>> Leftor: [(A_1:prop),(B_1:prop) => (Deduction([(aa_2:
>>              that A_1) => ((B_1 Orl aa_2):that
>>              (A_1 v B_1))])
>>         :that (A_1 -> (A_1 v B_1)))]
>>   {move 0}
```

```
% similarly, we prove B -> A v B

open

     declare bb that B

>>       bb: that B {move 2}


     define aorbx2 bb : Orr A bb

>>       aorbx2: [(bb_1:that B) => ((A Orr bb_1):
>>              that (A v B))]
>>         {move 1}


     close

define Rightor A B :  Deduction aorbx2

>> Rightor: [(A_1:prop),(B_1:prop) => (Deduction([(bb_2:
>>              that B_1) => ((A_1 Orr bb_2):that
>>              (A_1 v B_1))])
>>         :that (B_1 -> (A_1 v B_1)))]
>>   {move 0}
```

The approach to proof by cases exhibited in the following proof is very strange. We show that for any aa proving A and any bb proving B, Orl B aa and Orr A bb have the property that $B \vee A$ (which of course says nothing about these proofs). But these are all the proofs of $A \vee B$, so if we have any proof of $A \vee B$, it has the property $B \vee A$ (that is, we get a proof of $B \vee A$. This is very peculiar! But it does justify the rule of proof by cases quite generally.

We probably want more powerful tools for defining functions on proofs of $A \vee B$, so that we can do program extraction, for example. But we do not need such tools to do propositional logic, even constructive propositional

logic.

```
% now prove A v B -> B v A

open

     %% commonproperty must be contained in a block so that
     %% it is automatically expanded.  Otherwise the "matching bug"
     % seen earlier happens.

     open

          declare aorb that A v B

>>           aorb: that (A v B) {move 3}


          define commonproperty aorb : B v A

>>           commonproperty: [(aorb_1:that (A
>>                  v B)) => ((B v A):prop)]
>>             {move 2}


          close

     declare aa that A

>>      aa: that A {move 2}


     define commcase1 aa : Orr B aa

>>      commcase1: [(aa_1:that A) => ((B Orr
>>             aa_1):that (B v A))]
>>        {move 1}
```

```
     declare bb that B

>>       bb: that B {move 2}


     define commcase2 bb : Orl A bb

>>       commcase2: [(bb_1:that B) => ((A Orl
>>              bb_1):that (B v A))]
>>         {move 1}


     declare aorb2 that A v B

>>       aorb2: that (A v B) {move 2}


     define commproof aorb2 :  \
   Orproofsexhausted commonproperty,\
   commcase1, commcase2, aorb2

>>       commproof: [(aorb2_1:that (A v B)) =>
>>              (Orproofsexhausted([(aorb_2:that
>>                   (A v B)) => ((B v A):prop)]
>>              ,commcase1,commcase2,aorb2_1):that
>>              (B v A))]
>>         {move 1}


     close

define Orcomm A B : Deduction commproof

>> Orcomm: [(A_1:prop),(B_1:prop) => (Deduction([(aorb2_2:
>>              that (A_1 v B_1)) => (Orproofsexhausted([(aorb_3:
>>                   that (A_1 v B_1)) => ((B_1
>>                   v A_1):prop)]
>>              ,[(aa_4:that A_1) => ((B_1 Orr
```

```
>>                   aa_4):that (B_1 v A_1))]
>>              ,[(bb_5:that B_1) => ((A_1 Orl
>>                   bb_5):that (B_1 v A_1))]
>>              ,aorb2_2):that (B_1 v A_1))])
>>         :that ((A_1 v B_1) -> (B_1 v A_1)))]
>>   {move 0}


% build the or destructor

clearcurrent

declare A prop

>> A: prop {move 1}


declare B prop

>> B: prop {move 1}


declare aa that A

>> aa: that A {move 1}


declare bb that B

>> bb: that B {move 1}


declare aorb that A v B

>> aorb: that (A v B) {move 1}


construct Ifthenelset aorb :  prop
```

```
>> Ifthenelset: [(.A_1:prop),(.B_1:prop),(aorb_1:
>>        that (.A_1 v .B_1)) => (---:prop)]
>>   {move 0}


rewritec Ifthenelset1  B aa Ifthenelset (Orl B aa), A

>> Ifthenelset1'': [(Ifthenelset1'''_1:prop)
>>        => (---:prop)]
>>   {move 1}



>> Ifthenelset1': that Ifthenelset1''(Ifthenelset((B
>>   Orl aa))) {move 1}



>> Ifthenelset1: [(B_1:prop),(.A_1:prop),(aa_1:
>>        that .A_1),(Ifthenelset1''_1:[(Ifthenelset1'''_2:
>>             prop) => (---:prop)]),
>>        (Ifthenelset1'_1:that Ifthenelset1''_1(Ifthenelset((B_1
>>        Orl aa_1)))) => (---:that Ifthenelset1''_1(.A_1))]
>>   {move 0}


rewritec Ifthenelset2 A bb Ifthenelset (Orr A bb) , B

>> Ifthenelset2'': [(Ifthenelset2'''_1:prop)
>>        => (---:prop)]
>>   {move 1}



>> Ifthenelset2': that Ifthenelset2''(Ifthenelset((A
>>   Orr bb))) {move 1}
```

```
>> Ifthenelset2: [(A_1:prop),(.B_1:prop),(bb_1:
>>        that .B_1),(Ifthenelset2''_1:[(Ifthenelset2'''_2:
>>             prop) => (---:prop)]),
>>        (Ifthenelset2'_1:that Ifthenelset2''_1(Ifthenelset((A_1
>>        Orr bb_1)))) => (---:that Ifthenelset2''_1(.B_1))]
>>   {move 0}


construct Ordestruct aorb that Ifthenelset aorb

>> Ordestruct: [(.A_1:prop),(.B_1:prop),(aorb_1:
>>        that (.A_1 v .B_1)) => (---:that Ifthenelset(aorb_1))]
>>   {move 0}


rewritec Ordestruct1 B aa Ordestruct (Orl B aa), aa

>> Ordestruct1'': [(Ordestruct1'''_1:that Ifthenelset((B
>>        Orl aa))) => (---:prop)]
>>   {move 1}



>> Ordestruct1': that Ordestruct1''(Ordestruct((B
>>   Orl aa))) {move 1}



>> Ordestruct1: [(B_1:prop),(.A_1:prop),(aa_1:
>>        that .A_1),(Ordestruct1''_1:[(Ordestruct1'''_2:
>>             that Ifthenelset((B_1 Orl aa_1)))
>>             => (---:prop)]),
>>        (Ordestruct1'_1:that Ordestruct1''_1(Ordestruct((B_1
>>        Orl aa_1)))) => (---:that Ordestruct1''_1(aa_1))]
>>   {move 0}
```

```
rewritec Ordestruct2 A bb  Ordestruct (Orr A bb), bb

>> Ordestruct2'': [(Ordestruct2'''_1:that Ifthenelset((A
>>        Orr bb))) => (---:prop)]
>>   {move 1}



>> Ordestruct2': that Ordestruct2''(Ordestruct((A
>>   Orr bb))) {move 1}



>> Ordestruct2: [(A_1:prop),(.B_1:prop),(bb_1:
>>        that .B_1),(Ordestruct2''_1:[(Ordestruct2'''_2:
>>             that Ifthenelset((A_1 Orr bb_1)))
>>             => (---:prop)]),
>>        (Ordestruct2'_1:that Ordestruct2''_1(Ordestruct((A_1
>>        Orr bb_1)))) => (---:that Ordestruct2''_1(bb_1))]
>>   {move 0}


construct Orbranch aorb : in bool

>> Orbranch: [(.A_1:prop),(.B_1:prop),(aorb_1:
>>        that (.A_1 v .B_1)) => (---:in bool)]
>>   {move 0}


rewritec Orbranch1 B aa Orbranch (Orl B aa), true

>> Orbranch1'': [(Orbranch1'''_1:in bool) =>
>>        (---:prop)]
>>   {move 1}
```

```
>> Orbranch1': that Orbranch1''(Orbranch((B
>>   Orl aa))) {move 1}



>> Orbranch1: [(B_1:prop),(.A_1:prop),(aa_1:
>>        that .A_1),(Orbranch1''_1:[(Orbranch1'''_2:
>>             in bool) => (---:prop)]),
>>        (Orbranch1'_1:that Orbranch1''_1(Orbranch((B_1
>>        Orl aa_1)))) => (---:that Orbranch1''_1(true))]
>>   {move 0}


rewritec Orbranch2 A bb Orbranch (Orr A bb), false

>> Orbranch2'': [(Orbranch2'''_1:in bool) =>
>>        (---:prop)]
>>   {move 1}



>> Orbranch2': that Orbranch2''(Orbranch((A
>>   Orr bb))) {move 1}



>> Orbranch2: [(A_1:prop),(.B_1:prop),(bb_1:
>>        that .B_1),(Orbranch2''_1:[(Orbranch2'''_2:
>>             in bool) => (---:prop)]),
>>        (Orbranch2'_1:that Orbranch2''_1(Orbranch((A_1
>>        Orr bb_1)))) => (---:that Orbranch2''_1(false))]
>>   {move 0}
```

The availability of a destructor is important in Coq, which is supposed to support extraction of programs from proofs. In my usual Lestrade theories, no such function would be desired, as I support proof irrelevance: there is

at most one proof of any proposition! But Lestrade is designed to support many viewpoints. Notice that correct typing of the two rewrite rules for `Ordestruct` requires rewriting of types.

The typing of the destructor here is of course very tricky and quite amusing to watch working. A second destructor `Orbranch` extracts the flavor (left or right), while the first destructor extracts the proof of A or B.