Implementing homotopy type theory in Lestrade

M. Randall Holmes

April 28, 2018

This document is both a text document discussing implementation of HOTT (homotopy type theory) in the Lestrade logical framework, and a file executable under Lestrade: most of the verbatim blocks consist of actual dialogue with the Lestrade Type Inspector.

p. 18: the form of Lestrade averts identification of types used as propositions and types used as "collections" of objects, though the possibility of observing analogy between them remains.

"equality by definition" should correspond to applications of the Lestrade rewriting feature.

We declare function types (section 1.2, p. 21). Lestrade distinguishes types of proofs and types of mathematical objects, so we will provide the distinct but analogous constructions in parallel. We could also collapse propositions for purposes of HOTT into types, but for the moment we will try this approach.

```
declare At type

>> At: type {move 1}


declare Bt type

>> Bt: type {move 1}


construct ->> At Bt type

>> ->>: [(At_1:type),(Bt_1:type) => (---:type)]
>>   {move 0}


declare Ap prop

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

```
declare Bp prop

>> Bp: prop {move 1}


construct -> Ap Bp prop

>> ->: [(Ap_1:prop),(Bp_1:prop) => (---:prop)]
>>   {move 0}
```

We now discuss the introduction of elements of function types. The anonymous nomeclature of λ-terms was not (yet) supported by Lestrade at the time this part of the text was written, though Lestrade output could contain anonymous function terms; during the writing of this file terms with bound variables representing functions and function sorts were made available, and do appear later in the file.

```
open

     declare at1 in At

>>      at1: in At {move 2}


     construct ft at1 in Bt

>>      ft: [(at1_1:in At) => (---:in Bt)]
>>        {move 1}


     close

construct Lambda ft in At ->> Bt

>> Lambda: [(.At_1:type),(.Bt_1:type),(ft_1:
>>        [(at1_2:in .At_1) => (---:in .Bt_1)])
>>        => (---:in (.At_1 ->> .Bt_1))]
>>   {move 0}


open

     declare ap1 that Ap
```

```
>>      ap1: that Ap {move 2}


     construct ded ap1 that Bp

>>      ded: [(ap1_1:that Ap) => (---:that Bp)]
>>        {move 1}


     close

construct Deduction ded that Ap -> Bp

>> Deduction: [(.Ap_1:prop),(.Bp_1:prop),(ded_1:
>>        [(ap1_2:that .Ap_1) => (---:that .Bp_1)])
>>        => (---:that (.Ap_1 -> .Bp_1))]
>>   {move 0}
```

We implement the computation rule of β-reduction. We need to implement a primitive notion of function application.

```
declare ft1 in At ->> Bt

>> ft1: in (At ->> Bt) {move 1}


declare at1 in At

>> at1: in At {move 1}


construct Apply ft1 at1 in Bt

>> Apply: [(.At_1:type),(.Bt_1:type),(ft1_1:
>>        in (.At_1 ->> .Bt_1)),(at1_1:in .At_1)
>>        => (---:in .Bt_1)]
>>   {move 0}


rewritec Beta ft, at1,  Apply(Lambda ft, at1), ft at1

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

```
>> Beta': that Beta''((Lambda(ft) Apply at1))
>>   {move 1}



>> Beta: [(.At_1:type),(.Bt_1:type),(ft_1:[(at1_2:
>>             in .At_1) => (---:in .Bt_1)]),
>>        (at1_1:in .At_1),(Beta''_1:[(Beta'''_3:
>>             in .Bt_1) => (---:prop)]),
>>        (Beta'_1:that Beta''_1((Lambda(ft_1)
>>        Apply at1_1))) => (---:that Beta''_1(ft_1(at1_1)))]
>>   {move 0}


open

     declare at2 in At

>>      at2: in At {move 2}


     define idt at2 :  at2

>>      idt: [(at2_1:in At) => (at2_1:in At)]
>>        {move 1}


     close

define betatest at1 : Apply (Lambda idt, at1)

>> betatest: [(.At_1:type),(at1_1:in .At_1)
>>        => (at1_1:in .At_1)]
>>   {move 0}
```

We implement the rule of `modus ponens`, analogous to function application.

```
declare fp1 that Ap -> Bp

>> fp1: that (Ap -> Bp) {move 1}
```

```
declare ap1 that Ap

>> ap1: that Ap {move 1}


construct Mp fp1 ap1 that Bp

>> Mp: [(.Ap_1:prop),(.Bp_1:prop),(fp1_1:that
>>        (.Ap_1 -> .Bp_1)),(ap1_1:that .Ap_1)
>>        => (---:that .Bp_1)]
>>   {move 0}


rewritec Betap ded, ap1, Mp(Deduction ded, ap1), ded ap1

>> Betap'': [(Betap'''_1:that Bp) => (---:prop)]
>>   {move 1}



>> Betap': that Betap''((Deduction(ded) Mp ap1))
>>   {move 1}



>> Betap: [(.Ap_1:prop),(.Bp_1:prop),(ded_1:
>>        [(ap1_2:that .Ap_1) => (---:that .Bp_1)]),
>>        (ap1_1:that .Ap_1),(Betap''_1:[(Betap'''_3:
>>             that .Bp_1) => (---:prop)]),
>>        (Betap'_1:that Betap''_1((Deduction(ded_1)
>>        Mp ap1_1))) => (---:that Betap''_1(ded_1(ap1_1)))]
>>   {move 0}


open

     declare ap2 that Ap

>>      ap2: that Ap {move 2}


     define idp ap2 :  ap2

>>      idp: [(ap2_1:that Ap) => (ap2_1:that
>>             Ap)]
>>       {move 1}
```

```
     close

define betatestp ap1 : Mp (Deduction idp, ap1)

>> betatestp: [(.Ap_1:prop),(ap1_1:that .Ap_1)
>>        => (ap1_1:that .Ap_1)]
>>   {move 0}
```

We introduce η-reduction, the rule that allows reduction of $(\lambda x : f(x))$ to f.

```
declare Ft in At ->> Bt

>> Ft: in (At ->> Bt) {move 1}


open

     declare ft2 in At ->> Bt

>>      ft2: in (At ->> Bt) {move 2}


     declare at2 in At

>>      at2: in At {move 2}


     define applyft at2:  Apply Ft, at2

>>      applyft: [(at2_1:in At) => ((Ft Apply
>>             at2_1):in Bt)]
>>        {move 1}


     close

rewritec Eta Ft, Lambda applyft, Ft

>> Eta'': [(Eta'''_1:in (At ->> Bt)) => (---:
>>        prop)]
>>   {move 1}
```

```
>> Eta': that Eta''(Lambda(applyft)) {move 1}


>> Eta: [(.At_1:type),(.Bt_1:type),(Ft_1:in
>>        (.At_1 ->> .Bt_1)),(Eta''_1:[(Eta'''_2:
>>             in (.At_1 ->> .Bt_1)) => (---:prop)]),
>>        (Eta'_1:that Eta''_1(Lambda([(at2_3:
>>             in .At_1) => ((Ft_1 Apply at2_3):
>>             in .Bt_1)]))
>>        ) => (---:that Eta''_1(Ft_1))]
>>   {move 0}


declare Fp that Ap -> Bp

>> Fp: that (Ap -> Bp) {move 1}


open

     declare fp2 that Ap -> Bp

>>      fp2: that (Ap -> Bp) {move 2}


     declare ap2 that Ap

>>      ap2: that Ap {move 2}


     define applyfp ap2:  Mp Fp, ap2

>>      applyfp: [(ap2_1:that Ap) => ((Fp Mp
>>             ap2_1):that Bp)]
>>        {move 1}


     close

rewritec Etap Fp, Deduction applyfp, Fp

>> Etap'': [(Etap'''_1:that (Ap -> Bp)) => (---:
>>        prop)]
```

```
>>   {move 1}
```

```
>> Etap': that Etap''(Deduction(applyfp)) {move
>>   1}
```

```
>> Etap: [(.Ap_1:prop),(.Bp_1:prop),(Fp_1:that
>>        (.Ap_1 -> .Bp_1)),(Etap''_1:[(Etap'''_2:
>>             that (.Ap_1 -> .Bp_1)) => (---:
>>             prop)]),
>>        (Etap'_1:that Etap''_1(Deduction([(ap2_3:
>>             that .Ap_1) => ((Fp_1 Mp ap2_3):
>>             that .Bp_1)]))
>>        ) => (---:that Etap''_1(Fp_1))]
>>   {move 0}
```

It should be noted, re the discussion of currying on p. 23, that Lestrade functions of more than one variable do not follow either of the indicated paradigms: they are functions of multiple inputs, which are not construed as making up a composite object (there is no input tuple). However, Lestrade functions are not strictly speaking mathematical objects for Lestrade. We can implement currying in our object types of functions, and when we have products we will be able to implement the other approach.

We consider the problem of universes. HOTT wants each type to be an element of a sort called a universe. This does violence to Lestrade. So, universes for us are sorts. Universes form a sequence (which we do not have the ability to index).

We begin with a retraction of `type` onto the types used as universes.

```
declare Tt type

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

```
construct Univ Tt type

>> Univ: [(Tt_1:type) => (---:type)]
>>   {move 0}
```

```
rewritec Uretract Tt, Univ (Univ Tt), Univ Tt
```

```
>> Uretract'': [(Uretract'''_1:type) => (---:
>>        prop)]
>>   {move 1}
```

```
>> Uretract': that Uretract''(Univ(Univ(Tt)))
>>   {move 1}
```

```
>> Uretract: [(Tt_1:type),(Uretract''_1:[(Uretract'''_2:
>>             type) => (---:prop)]),
>>        (Uretract'_1:that Uretract''_1(Univ(Univ(Tt_1))))
>>        => (---:that Uretract''_1(Univ(Tt_1)))]
>>   {move 0}
```

We have an order on universes.

```
declare Tta type

>> Tta: type {move 1}


declare Ttb type

>> Ttb: type {move 1}


construct << Tta Ttb prop

>> <<: [(Tta_1:type),(Ttb_1:type) => (---:prop)]
>>   {move 0}


declare order1 that Tt << Tta

>> order1: that (Tt << Tta) {move 1}


declare order2 that Tta << Ttb

>> order2: that (Tta << Ttb) {move 1}
```

```
construct Utrans order1 order2 that Tt << Ttb

>> Utrans: [(.Tt_1:type),(.Tta_1:type),(order1_1:
>>        that (.Tt_1 << .Tta_1)),(.Ttb_1:type),
>>        (order2_1:that (.Tta_1 << .Ttb_1)) =>
>>        (---:that (.Tt_1 << .Ttb_1))]
>>   {move 0}


rewritec Ordertag1 Tta, Ttb, (Univ Tta) << Ttb, Tta << Ttb

>> Ordertag1'': [(Ordertag1'''_1:prop) => (---:
>>        prop)]
>>   {move 1}



>> Ordertag1': that Ordertag1''((Univ(Tta) <<
>>   Ttb)) {move 1}



>> Ordertag1: [(Tta_1:type),(Ttb_1:type),(Ordertag1''_1:
>>        [(Ordertag1'''_2:prop) => (---:prop)]),
>>        (Ordertag1'_1:that Ordertag1''_1((Univ(Tta_1)
>>        << Ttb_1))) => (---:that Ordertag1''_1((Tta_1
>>        << Ttb_1)))]
>>   {move 0}


rewritec Ordertag2 Tta, Ttb, Tta << Univ Ttb, Tta << Ttb

>> Ordertag2'': [(Ordertag2'''_1:prop) => (---:
>>        prop)]
>>   {move 1}



>> Ordertag2': that Ordertag2''((Tta << Univ(Ttb)))
>>   {move 1}



>> Ordertag2: [(Tta_1:type),(Ttb_1:type),(Ordertag2''_1:
```

```
>>        [(Ordertag2'''_2:prop) => (---:prop)]),
>>        (Ordertag2'_1:that Ordertag2''_1((Tta_1
>>        << Univ(Ttb_1)))) => (---:that Ordertag2''_1((Tta_1
>>        << Ttb_1)))]
>>   {move 0}
```

construct Umin type

```
>> Umin: type {move 0}
```

rewritec Mintag Univ Umin, Umin

```
>> Mintag'': [(Mintag'''_1:type) => (---:prop)]
>>   {move 1}



>> Mintag': that Mintag''(Univ(Umin)) {move
>>   1}



>> Mintag: [(Mintag''_1:[(Mintag'''_2:type)
>>             => (---:prop)]),
>>        (Mintag'_1:that Mintag''_1(Univ(Umin)))
>>        => (---:that Mintag''_1(Umin))]
>>   {move 0}
```

construct Uminismin Tt that Umin << Tt

```
>> Uminismin: [(Tt_1:type) => (---:that (Umin
>>        << Tt_1))]
>>   {move 0}
```

construct maxu Tt Tta type

```
>> maxu: [(Tt_1:type),(Tta_1:type) => (---:type)]
>>   {move 0}
```

rewritec Maxtag1 Tt Tta Univ(Tt maxu Tta) Tt maxu Tta

```
>> Maxtag1'': [(Maxtag1'''_1:type) => (---:prop)]
>>   {move 1}



>> Maxtag1': that Maxtag1''(Univ((Tt maxu Tta)))
>>   {move 1}



>> Maxtag1: [(Tt_1:type),(Tta_1:type),(Maxtag1''_1:
>>        [(Maxtag1'''_2:type) => (---:prop)]),
>>        (Maxtag1'_1:that Maxtag1''_1(Univ((Tt_1
>>        maxu Tta_1)))) => (---:that Maxtag1''_1((Tt_1
>>        maxu Tta_1)))]
>>   {move 0}


rewritec Maxtag2 Tt Tta (Univ Tt) maxu Tta Tt maxu Tta

>> Maxtag2'': [(Maxtag2'''_1:type) => (---:prop)]
>>   {move 1}



>> Maxtag2': that Maxtag2''((Univ(Tt) maxu Tta))
>>   {move 1}



>> Maxtag2: [(Tt_1:type),(Tta_1:type),(Maxtag2''_1:
>>        [(Maxtag2'''_2:type) => (---:prop)]),
>>        (Maxtag2'_1:that Maxtag2''_1((Univ(Tt_1)
>>        maxu Tta_1))) => (---:that Maxtag2''_1((Tt_1
>>        maxu Tta_1)))]
>>   {move 0}


rewritec Maxtag3 Tt Tta Tt maxu Univ Tta Tt maxu Tta

>> Maxtag3'': [(Maxtag3'''_1:type) => (---:prop)]
>>   {move 1}



>> Maxtag3': that Maxtag3''((Tt maxu Univ(Tta)))
```

```
>>   {move 1}



>> Maxtag3: [(Tt_1:type),(Tta_1:type),(Maxtag3''_1:
>>        [(Maxtag3'''_2:type) => (---:prop)]),
>>        (Maxtag3'_1:that Maxtag3''_1((Tt_1 maxu
>>        Univ(Tta_1)))) => (---:that Maxtag3''_1((Tt_1
>>        maxu Tta_1)))]
>>   {move 0}


construct Maxorder1 Tt Tta that Tt << Tt maxu Tta

>> Maxorder1: [(Tt_1:type),(Tta_1:type) => (---:
>>        that (Tt_1 << (Tt_1 maxu Tta_1)))]
>>   {move 0}


construct  Maxorder2 Tt Tta that Tta << Tt maxu Tta

>> Maxorder2: [(Tt_1:type),(Tta_1:type) => (---:
>>        that (Tta_1 << (Tt_1 maxu Tta_1)))]
>>   {move 0}


declare order3 that Tt << Ttb

>> order3: that (Tt << Ttb) {move 1}


declare order4 that Tta << Ttb

>> order4: that (Tta << Ttb) {move 1}


construct Maxorder3 order3 order4 that (Tt maxu Tta) << Ttb

>> Maxorder3: [(.Tt_1:type),(.Ttb_1:type),(order3_1:
>>        that (.Tt_1 << .Ttb_1)),(.Tta_1:type),
>>        (order4_1:that (.Tta_1 << .Ttb_1)) =>
>>        (---:that ((.Tt_1 maxu .Tta_1) << .Ttb_1))]
>>   {move 0}
```

We have a next universe

```
construct Nextu Tt type

>> Nextu: [(Tt_1:type) => (---:type)]
>>   {move 0}


rewritec Nextutag1, Tt, Univ(Nextu Tt), Nextu Tt

>> Nextutag1'': [(Nextutag1'''_1:type) => (---:
>>        prop)]
>>   {move 1}



>> Nextutag1': that Nextutag1''(Univ(Nextu(Tt)))
>>   {move 1}



>> Nextutag1: [(Tt_1:type),(Nextutag1''_1:[(Nextutag1'''_2:
>>             type) => (---:prop)]),
>>        (Nextutag1'_1:that Nextutag1''_1(Univ(Nextu(Tt_1))))
>>        => (---:that Nextutag1''_1(Nextu(Tt_1)))]
>>   {move 0}


rewritec Nextutag2, Tt, Nextu(Univ Tt), Tt

>> Nextutag2'': [(Nextutag2'''_1:type) => (---:
>>        prop)]
>>   {move 1}



>> Nextutag2': that Nextutag2''(Nextu(Univ(Tt)))
>>   {move 1}



>> Nextutag2: [(Tt_1:type),(Nextutag2''_1:[(Nextutag2'''_2:
>>             type) => (---:prop)]),
>>        (Nextutag2'_1:that Nextutag2''_1(Nextu(Univ(Tt_1))))
>>        => (---:that Nextutag2''_1(Tt_1))]
>>   {move 0}
```

```
construct Nextorder Tt that Tt << Nextu Tt

>> Nextorder: [(Tt_1:type) => (---:that (Tt_1
>>        << Nextu(Tt_1)))]
>>   {move 0}
```

Each universe is mapped into the types.

```
declare x in Univ Tt

>> x: in Univ(Tt) {move 1}
```

```
construct Utype Tt x type

>> Utype: [(Tt_1:type),(x_1:in Univ(Tt_1)) =>
>>        (---:type)]
>>   {move 0}
```

```
construct Uprop Tt x prop

>> Uprop: [(Tt_1:type),(x_1:in Univ(Tt_1)) =>
>>        (---:prop)]
>>   {move 0}
```

Each inhabitant of a universe maps upward into any universe later in the order, and its image has the same associated type.

```
construct Raiseu Tt Tta order1 x in Univ Tta

>> Raiseu: [(Tt_1:type),(Tta_1:type),(order1_1:
>>        that (Tt_1 << Tta_1)),(x_1:in Univ(Tt_1))
>>        => (---:in Univ(Tta_1))]
>>   {move 0}
```

```
rewritec Embed1 Tt, Tta, order1, Utype (Tta, Raiseu Tt Tta order1 x), Utype (Tt, x)

>> Embed1'': [(Embed1'''_1:type) => (---:prop)]
>>   {move 1}
```

```
>> Embed1': that Embed1''((Tta Utype Raiseu(Tt,
>>   Tta,order1,x))) {move 1}
```

```
>> Embed1: [(Tt_1:type),(Tta_1:type),(order1_1:
>>        that (Tt_1 << Tta_1)),(Embed1''_1:[(Embed1'''_2:
>>             type) => (---:prop)]),
>>        (.x_1:in Univ(Tt_1)),(Embed1'_1:that
>>        Embed1''_1((Tta_1 Utype Raiseu(Tt_1,
>>        Tta_1,order1_1,.x_1)))) => (---:that
>>        Embed1''_1((Tt_1 Utype .x_1)))]
>>   {move 0}
```

```
rewritec Embedp1 Tt, Tta, order1, Uprop (Tta, Raiseu Tt Tta order1 x), Uprop (Tt, x)
```

```
>> Embedp1'': [(Embedp1'''_1:prop) => (---:prop)]
>>   {move 1}
```

```
>> Embedp1': that Embedp1''((Tta Uprop Raiseu(Tt,
>>   Tta,order1,x))) {move 1}
```

```
>> Embedp1: [(Tt_1:type),(Tta_1:type),(order1_1:
>>        that (Tt_1 << Tta_1)),(Embedp1''_1:[(Embedp1'''_2:
>>             prop) => (---:prop)]),
>>        (.x_1:in Univ(Tt_1)),(Embedp1'_1:that
>>        Embedp1''_1((Tta_1 Uprop Raiseu(Tt_1,
>>        Tta_1,order1_1,.x_1)))) => (---:that
>>        Embedp1''_1((Tt_1 Uprop .x_1)))]
>>   {move 0}
```

Each type has a minimal universe that it lives in. The minimal universe that a universe lives in is the next universe.

```
declare Pp prop
```

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


construct Uof Tt type

>> Uof: [(Tt_1:type) => (---:type)]
>>   {move 0}


rewritec Uoftag Tt, Univ(Uof Tt), Uof Tt

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



>> Uoftag': that Uoftag''(Univ(Uof(Tt))) {move
>>   1}



>> Uoftag: [(Tt_1:type),(Uoftag''_1:[(Uoftag'''_2:
>>             type) => (---:prop)]),
>>        (Uoftag'_1:that Uoftag''_1(Univ(Uof(Tt_1))))
>>        => (---:that Uoftag''_1(Uof(Tt_1)))]
>>   {move 0}


construct Uofp Pp type

>> Uofp: [(Pp_1:prop) => (---:type)]
>>   {move 0}


rewritec Uofptag Pp, Univ(Uofp Pp), Uofp Pp

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



>> Uofptag': that Uofptag''(Univ(Uofp(Pp)))
>>   {move 1}
```

```
>> Uofptag: [(Pp_1:prop),(Uofptag''_1:[(Uofptag'''_2:
>>             type) => (---:prop)]),
>>        (Uofptag'_1:that Uofptag''_1(Univ(Uofp(Pp_1))))
>>        => (---:that Uofptag''_1(Uofp(Pp_1)))]
>>   {move 0}


construct Inuof Tt in Uof Tt

>> Inuof: [(Tt_1:type) => (---:in Uof(Tt_1))]
>>   {move 0}


construct Inuofp Pp in Uofp Pp

>> Inuofp: [(Pp_1:prop) => (---:in Uofp(Pp_1))]
>>   {move 0}


rewritec Embed2 Tt, Utype (Uof Tt, Inuof Tt), Tt

>> Embed2'': [(Embed2'''_1:type) => (---:prop)]
>>   {move 1}



>> Embed2': that Embed2''((Uof(Tt) Utype Inuof(Tt)))
>>   {move 1}



>> Embed2: [(Tt_1:type),(Embed2''_1:[(Embed2'''_2:
>>             type) => (---:prop)]),
>>        (Embed2'_1:that Embed2''_1((Uof(Tt_1)
>>        Utype Inuof(Tt_1)))) => (---:that Embed2''_1(Tt_1))]
>>   {move 0}


rewritec Embedp2 Pp, Uprop (Uofp Pp, Inuofp Pp), Pp

>> Embedp2'': [(Embedp2'''_1:prop) => (---:prop)]
>>   {move 1}



>> Embedp2': that Embedp2''((Uofp(Pp) Uprop
```

```
>>   Inuofp(Pp))) {move 1}



>> Embedp2: [(Pp_1:prop),(Embedp2''_1:[(Embedp2'''_2:
>>             prop) => (---:prop)]),
>>        (Embedp2'_1:that Embedp2''_1((Uofp(Pp_1)
>>        Uprop Inuofp(Pp_1)))) => (---:that Embedp2''_1(Pp_1))]
>>   {move 0}


construct Uofismin Tt x that (Uof (Utype Tt x)) << Tt

>> Uofismin: [(Tt_1:type),(x_1:in Univ(Tt_1))
>>        => (---:that (Uof((Tt_1 Utype x_1))
>>        << Tt_1))]
>>   {move 0}


construct Uofisminp Tt x that (Uofp(Uprop Tt x))<< Tt

>> Uofisminp: [(Tt_1:type),(x_1:in Univ(Tt_1))
>>        => (---:that (Uofp((Tt_1 Uprop x_1))
>>        << Tt_1))]
>>   {move 0}


rewritec Uofu Tt, Uof(Univ Tt), Nextu Tt

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



>> Uofu': that Uofu''(Uof(Univ(Tt))) {move 1}



>> Uofu: [(Tt_1:type),(Uofu''_1:[(Uofu'''_2:
>>             type) => (---:prop)]),
>>        (Uofu'_1:that Uofu''_1(Uof(Univ(Tt_1))))
>>        => (---:that Uofu''_1(Nextu(Tt_1)))]
>>   {move 0}


rewritec Uoffunction  Tt Tta Uof(Tt ->> Tta) (Uof Tt) maxu Uof Tta
```

```
>> Uoffunction'': [(Uoffunction'''_1:type) =>
>>        (---:prop)]
>>   {move 1}



>> Uoffunction': that Uoffunction''(Uof((Tt
>>   ->> Tta))) {move 1}



>> Uoffunction: [(Tt_1:type),(Tta_1:type),(Uoffunction''_1:
>>        [(Uoffunction'''_2:type) => (---:prop)]),
>>        (Uoffunction'_1:that Uoffunction''_1(Uof((Tt_1
>>        ->> Tta_1)))) => (---:that Uoffunction''_1((Uof(Tt_1)
>>        maxu Uof(Tta_1))))]
>>   {move 0}


declare Ppa prop

>> Ppa: prop {move 1}


rewritec Uofimp  Pp Ppa Uofp(Pp-> Ppa) (Uofp Pp) maxu Uofp Ppa

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



>> Uofimp': that Uofimp''(Uofp((Pp -> Ppa)))
>>   {move 1}



>> Uofimp: [(Pp_1:prop),(Ppa_1:prop),(Uofimp''_1:
>>        [(Uofimp'''_2:type) => (---:prop)]),
>>        (Uofimp'_1:that Uofimp''_1(Uofp((Pp_1
>>        -> Ppa_1)))) => (---:that Uofimp''_1((Uofp(Pp_1)
>>        maxu Uofp(Ppa_1))))]
>>   {move 0}
```

I added provisions to compute the minimal universes of function and implication types.

We also will want the negative assertion that given a universe, the next universe is not less than or equal to it in the order on universes.

I also need to consider the handling of `prop` relative to universes: this has now been handled above by providing embeddings of universes into `prop` as well as into `type`.

Something fun to think about is where `that Tt << Tta` lives in the universe structure. Probably unproblematically at the bottom, but it is still entertaining.

We may need max and min operations on the universe order when we start actually typing constructions in Martin-Löf type theory.

That this is all quite elaborate does not tell against Lestrade. It witnesses my thinking about HoTT, which is terribly baroque. Lestrade is extremely economical in its native primitives: HoTT is not. That Lestrade can implement HoTT tells in its favor (I fully expect that it can, but all the declarations must be written!). That HoTT has a quite elaborate declaration says something accurate about HoTT.

This is of course rather more elaborate because Lestrade does not support subtyping. But subtyping is problematic, always.

It can also be noted that at some points in the above, Lestrade actually uses rewrite rules to type check.

Sketch development of the HoTT dependent type construction.

```
clearcurrent

declare Tt type

>> Tt: type {move 1}


declare A in Univ Tt

>> A: in Univ(Tt) {move 1}


declare a in Utype Tt A

>> a: in (Tt Utype A) {move 1}


declare B [a => in Univ Tt]

>> B: [(a_1:in (Tt Utype A)) => (---:in Univ(Tt))]
>>   {move 1}
```

```
construct Depfun A B type

>> Depfun: [(.Tt_1:type),(A_1:in Univ(.Tt_1)),
>>        (B_1:[(a_2:in (.Tt_1 Utype A_1)) =>
>>             (---:in Univ(.Tt_1))])
>>        => (---:type)]
>>   {move 0}


construct Forall A B prop

>> Forall: [(.Tt_1:type),(A_1:in Univ(.Tt_1)),
>>        (B_1:[(a_2:in (.Tt_1 Utype A_1)) =>
>>             (---:in Univ(.Tt_1))])
>>        => (---:prop)]
>>   {move 0}


declare F in Depfun A B

>> F: in (A Depfun B) {move 1}


declare Fp that Forall A B

>> Fp: that (A Forall B) {move 1}


declare a2 in Utype Tt A

>> a2: in (Tt Utype A) {move 1}


construct Applyd F a2 in Utype Tt B a2

>> Applyd: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),
>>        (.B_1:[(a_2:in (.Tt_1 Utype .A_1)) =>
>>             (---:in Univ(.Tt_1))]),
>>        (F_1:in (.A_1 Depfun .B_1)),(a2_1:in
>>        (.Tt_1 Utype .A_1)) => (---:in (.Tt_1
>>        Utype .B_1(a2_1)))]
>>   {move 0}


define Applydx Tt F a2 :  Applyd F a2
```

```
>> Applydx: [(Tt_1:type),(.A_1:in Univ(Tt_1)),
>>        (.B_1:[(a_2:in (Tt_1 Utype .A_1)) =>
>>             (---:in Univ(Tt_1))]),
>>        (F_1:in (.A_1 Depfun .B_1)),(a2_1:in
>>        (Tt_1 Utype .A_1)) => ((F_1 Applyd a2_1):
>>        in (Tt_1 Utype .B_1(a2_1)))]
>>   {move 0}
```

```
construct Ui Fp a2 that Uprop Tt B a2

>> Ui: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),(.B_1:
>>        [(a_2:in (.Tt_1 Utype .A_1)) => (---:
>>             in Univ(.Tt_1))]),
>>        (Fp_1:that (.A_1 Forall .B_1)),(a2_1:
>>        in (.Tt_1 Utype .A_1)) => (---:that
>>        (.Tt_1 Uprop .B_1(a2_1)))]
>>   {move 0}
```

```
declare f [a => in Utype Tt B a]

>> f: [(a_1:in (Tt Utype A)) => (---:in (Tt
>>        Utype B(a_1)))]
>>   {move 1}
```

```
declare fp [a => that Uprop Tt B a]

>> fp: [(a_1:in (Tt Utype A)) => (---:that (Tt
>>        Uprop B(a_1)))]
>>   {move 1}
```

```
construct Lambdad f in Depfun A B

>> Lambdad: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),
>>        (.B_1:[(a_2:in (.Tt_1 Utype .A_1)) =>
>>             (---:in Univ(.Tt_1))]),
>>        (f_1:[(a_3:in (.Tt_1 Utype .A_1)) =>
>>             (---:in (.Tt_1 Utype .B_1(a_3)))])
>>        => (---:in (.A_1 Depfun .B_1))]
>>   {move 0}
```

```
construct Ug fp that Forall A B
```

```
>> Ug: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),(.B_1:
>>        [(a_2:in (.Tt_1 Utype .A_1)) => (---:
>>             in Univ(.Tt_1))]),
>>        (fp_1:[(a_3:in (.Tt_1 Utype .A_1)) =>
>>             (---:that (.Tt_1 Uprop .B_1(a_3)))])
>>        => (---:that (.A_1 Forall .B_1))]
>>   {move 0}


rewritec Depuniverse A B, Uof (Depfun A B) Univ Tt

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



>> Depuniverse': that Depuniverse''(Uof((A Depfun
>>   B))) {move 1}



>> Depuniverse: [(.Tt_1:type),(A_1:in Univ(.Tt_1)),
>>        (B_1:[(a_2:in (.Tt_1 Utype A_1)) =>
>>             (---:in Univ(.Tt_1))]),
>>        (Depuniverse''_1:[(Depuniverse'''_3:
>>             type) => (---:prop)]),
>>        (Depuniverse'_1:that Depuniverse''_1(Uof((A_1
>>        Depfun B_1)))) => (---:that Depuniverse''_1(Univ(.Tt_1)))]
>>   {move 0}


rewritec Depuniversep A B, Uofp (Forall A B) Univ Tt

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



>> Depuniversep': that Depuniversep''(Uofp((A
>>   Forall B))) {move 1}
```

```
>> Depuniversep: [(.Tt_1:type),(A_1:in Univ(.Tt_1)),
>>        (B_1:[(a_2:in (.Tt_1 Utype A_1)) =>
>>             (---:in Univ(.Tt_1))]),
>>        (Depuniversep''_1:[(Depuniversep'''_3:
>>             type) => (---:prop)]),
>>        (Depuniversep'_1:that Depuniversep''_1(Uofp((A_1
>>        Forall B_1)))) => (---:that Depuniversep''_1(Univ(.Tt_1)))]
>>   {move 0}
```

I have now constructed the dependent function types and universally quantified propositions of HoTT. One thing I started doing with these types is taking advantage of the implicit argument feature; earlier functions in the current draft show all arguments because I was carefully thinking about them, but a lot of them are deducible.

It is a notable point that I built the type directly rather than building its tag in some universe. A style in which a tag is built in `Univ Tt` is a possible alternative.

The exact minimal universe to which `Depfun A B` belongs is a feature of it, because the codomain of B is a feature of B in type theory.

We commence building product (and conjunction) types.

```
clearcurrent

construct Unit type

>> Unit: type {move 0}


construct unit in Unit

>> unit: in Unit {move 0}


declare A type

>> A: type {move 1}


declare B type

>> B: type {move 1}


construct ** A B type
```

```
>> **: [(A_1:type),(B_1:type) => (---:type)]
>>   {move 0}


declare a in A

>> a: in A {move 1}


declare  b in B

>> b: in B {move 1}


construct ; a b in A ** B

>> ;: [(.A_1:type),(a_1:in .A_1),(.B_1:type),
>>        (b_1:in .B_1) => (---:in (.A_1 ** .B_1))]
>>   {move 0}


declare C type

>> C: type {move 1}


declare g [a, b => in C]

>> g: [(a_1:in A),(b_1:in B) => (---:in C)]
>>   {move 1}


construct recprod g in (A ** B) ->> C

>> recprod: [(.A_1:type),(.B_1:type),(.C_1:type),
>>        (g_1:[(a_2:in .A_1),(b_2:in .B_1) =>
>>             (---:in .C_1)])
>>        => (---:in ((.A_1 ** .B_1) ->> .C_1))]
>>   {move 0}


rewritec Paireval a b g, Apply (recprod g, a ; b), g a b

>> Paireval'': [(Paireval'''_1:in C) => (---:
>>        prop)]
```

```
>>   {move 1}



>> Paireval': that Paireval''((recprod(g) Apply
>>   (a ; b))) {move 1}



>> Paireval: [(.A_1:type),(a_1:in .A_1),(.B_1:
>>        type),(b_1:in .B_1),(.C_1:type),(g_1:
>>        [(a_2:in .A_1),(b_2:in .B_1) => (---:
>>             in .C_1)]),
>>        (Paireval''_1:[(Paireval'''_3:in .C_1)
>>             => (---:prop)]),
>>        (Paireval'_1:that Paireval''_1((recprod(g_1)
>>        Apply (a_1 ; b_1)))) => (---:that Paireval''_1((a_1
>>        g_1 b_1)))]
>>   {move 0}


define P1 A B : recprod  [a,b=>a]

>> P1: [(A_1:type),(B_1:type) => (recprod([(a_2:
>>             in A_1),(b_2:in B_1) => (a_2:in
>>             A_1)])
>>        :in ((A_1 ** B_1) ->> A_1))]
>>   {move 0}


define P2 A B : recprod  [a,b=>b]

>> P2: [(A_1:type),(B_1:type) => (recprod([(a_2:
>>             in A_1),(b_2:in B_1) => (b_2:in
>>             B_1)])
>>        :in ((A_1 ** B_1) ->> B_1))]
>>   {move 0}


declare x in A ** B

>> x: in (A ** B) {move 1}


define p1 x : Apply (recprod [a,b=>a],x)
```

```
>> p1: [(.A_1:type),(.B_1:type),(x_1:in (.A_1
>>        ** .B_1)) => ((recprod([(a_2:in .A_1),
>>             (b_2:in .B_1) => (a_2:in .A_1)])
>>        Apply x_1):in .A_1)]
>>    {move 0}


define p2 x : Apply (recprod [a,b=>b],x)

>> p2: [(.A_1:type),(.B_1:type),(x_1:in (.A_1
>>        ** .B_1)) => ((recprod([(a_2:in .A_1),
>>             (b_2:in .B_1) => (b_2:in .B_1)])
>>        Apply x_1):in .B_1)]
>>    {move 0}


define applytest a b :  Apply(recprod [a,b=>a],a;b)

>> applytest: [(.A_1:type),(a_1:in .A_1),(.B_1:
>>        type),(b_1:in .B_1) => (a_1:in .A_1)]
>>    {move 0}


rewritec Uofprod A B Uof(A ** B), (Uof A) maxu Uof B

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



>> Uofprod': that Uofprod''(Uof((A ** B))) {move
>>    1}



>> Uofprod: [(A_1:type),(B_1:type),(Uofprod''_1:
>>        [(Uofprod'''_2:type) => (---:prop)]),
>>        (Uofprod'_1:that Uofprod''_1(Uof((A_1
>>        ** B_1)))) => (---:that Uofprod''_1((Uof(A_1)
>>        maxu Uof(B_1))))]
>>    {move 0}


rewritec Techfix A B, Utype((Uof A)maxu Uof B,Inuof(A**B)), A**B

>> Techfix'': [(Techfix'''_1:type) => (---:prop)]
```

```
>>   {move 1}



>> Techfix': that Techfix''(((Uof(A) maxu Uof(B))
>>   Utype Inuof((A ** B)))) {move 1}



>> Techfix: [(A_1:type),(B_1:type),(Techfix''_1:
>>        [(Techfix'''_2:type) => (---:prop)]),
>>        (Techfix'_1:that Techfix''_1(((Uof(A_1)
>>        maxu Uof(B_1)) Utype Inuof((A_1 ** B_1)))))
>>        => (---:that Techfix''_1((A_1 ** B_1)))]
>>   {move 0}
```

My recursor for product types is not restricted to a universe, but it is also not an object. The recursors that they define, existing in dependent product types, are readily defined directly using our "class recursor", and our class recursor does nothing more for us than the uniform existence of their product recursors does for them.

Now to define the induction operator for products (which asserts that anything defined on all pairs is thereby defined on the entire product)

The universe rule for products, implemented as a rewrite rule, creates a problem in the typing of localized product induction constructors below. Adding `Techfix` removes the problem. Disabling the rule temporarily (which is something the Lestrade user model supports) would also work, but I'd just as soon have the rule active in this script.

```
open

     declare x1 in A ** B

>>      x1: in (A ** B) {move 2}


     construct Tprodfun x1 type

>>      Tprodfun: [(x1_1:in (A ** B)) => (---:
>>             type)]
>>        {move 1}


     declare a1 in A
```

```
>>      a1: in A {move 2}


     declare b1 in B

>>      b1: in B {move 2}


     construct Tpairsfun a1 b1 in Tprodfun (a1;b1)

>>      Tpairsfun: [(a1_1:in A),(b1_1:in B)
>>             => (---:in Tprodfun((a1_1 ; b1_1)))]
>>        {move 1}


     close

declare xx in A ** B

>> xx: in (A ** B) {move 1}


construct Prodinduction Tpairsfun, xx in Tprodfun xx

>> Prodinduction: [(.A_1:type),(.B_1:type),(.Tprodfun_1:
>>        [(x1_2:in (.A_1 ** .B_1)) => (---:type)]),
>>        (Tpairsfun_1:[(a1_3:in .A_1),(b1_3:in
>>             .B_1) => (---:in .Tprodfun_1((a1_3
>>             ; b1_3)))]),
>>        (xx_1:in (.A_1 ** .B_1)) => (---:in
>>        .Tprodfun_1(xx_1))]
>>   {move 0}
```

Again, we see no reason to localize `Typeinduction` to a universe, though we certainly can construct all such localized versions. Our `Typeinduction` is not an object, and does nothing more for us than uniform existence of the localized induction functions does in the version of the HoTT book before me.

However, in the next block we localize it, just to show that we can.

```
open

     construct Tt type
```

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


     declare x1 in A ** B

>>      x1: in (A ** B) {move 2}


     construct Tprodfun2  x1 in Univ Tt

>>      Tprodfun2: [(x1_1:in (A ** B)) => (---:
>>             in Univ(Tt))]
>>        {move 1}


     construct ubound that Uof (A**B) << Tt

>>      ubound: that (Uof((A ** B)) << Tt) {move
>>        1}


     declare a1 in A

>>      a1: in A {move 2}


     declare b1 in B

>>      b1: in B {move 2}


     construct Tpairsfun2 a1 b1 in Utype Tt  (Tprodfun2 (a1;b1))

>>      Tpairsfun2: [(a1_1:in A),(b1_1:in B)
>>             => (---:in (Tt Utype Tprodfun2((a1_1
>>             ; b1_1))))]
>>        {move 1}


     close

define typetest A B Tt ubound : Utype (Tt ,Raiseu (Uof(A**B),Tt,ubound,Inuof (A**B)))

>> typetest: [(A_1:type),(B_1:type),(Tt_1:type),
>>        (ubound_1:that (Uof((A_1 ** B_1)) <<
>>        Tt_1)) => ((A_1 ** B_1):type)]
```

```
>>   {move 0}


construct Prodinduction2 ubound Tpairsfun2 \
      in Depfun (Raiseu (Uof(A**B),Tt,ubound,Inuof (A**B)),Tprodfun2)

>> Prodinduction2: [(.A_1:type),(.B_1:type),
>>        (.Tt_1:type),(ubound_1:that (Uof((.A_1
>>        ** .B_1)) << .Tt_1)),(.Tprodfun2_1:[(x1_2:
>>             in (.A_1 ** .B_1)) => (---:in Univ(.Tt_1))]),
>>        (Tpairsfun2_1:[(a1_3:in .A_1),(b1_3:
>>             in .B_1) => (---:in (.Tt_1 Utype
>>             .Tprodfun2_1((a1_3 ; b1_3))))])
>>        => (---:in (Raiseu(Uof((.A_1 ** .B_1)),
>>        .Tt_1,ubound_1,Inuof((.A_1 ** .B_1)))
>>        Depfun .Tprodfun2_1))]
>>   {move 0}
```

The gruesome lesson here was that typing this was tricky: we originally had to disable the rewrite rule for universes of products for Lestrade to be able to verify the type inference. But it *was* able to do so, and it is part of the Lestrade user model that the user can cherry-pick the rewrite rules that she wants to use in a particular context. Some investigation of the reasons why it failed (technically instructive!) enabled us to install an additional rewrite rule `Techfix` above which allows this proof to coexist with the rewrite rule for universe of products. There are commands for disabling and reenabling rewrite rules but they are a bit awkward at the moment.

The use of an open/close block made development much easier!

All of this needs to be constructed on the propositional side of things. I'll take a simpler approach (explicit projections) to recursion for conjunction types, and I'll provide a version of induction on product types with proposition output. Other variations are probably not needed.

```
clearcurrent

declare A type

>> A: type {move 1}


declare B type

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

```
declare p prop

>> p: prop {move 1}


declare q prop

>> q: prop {move 1}


declare pp that p

>> pp: that p {move 1}


declare qq that q

>> qq: that q {move 1}


construct & p q prop

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


declare rr that p & q

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


construct 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}


construct Simplification1 rr that p

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

```
construct Simplication2 rr that q

>> Simplication2: [(.p_1:prop),(.q_1:prop),(rr_1:
>>        that (.p_1 & .q_1)) => (---:that .q_1)]
>>   {move 0}


open

     construct Tt type

>>      Tt: type {move 1}


     declare x1 in A ** B

>>      x1: in (A ** B) {move 2}


     construct Tprodfun2  x1 in Univ Tt

>>      Tprodfun2: [(x1_1:in (A ** B)) => (---:
>>             in Univ(Tt))]
>>        {move 1}


     construct ubound that Uof (A**B) << Tt

>>      ubound: that (Uof((A ** B)) << Tt) {move
>>        1}


     declare a1 in A

>>      a1: in A {move 2}


     declare b1 in B

>>      b1: in B {move 2}


     construct Tpairsfunp a1 b1 that Uprop Tt  (Tprodfun2 (a1;b1))
```

```
>>      Tpairsfunp: [(a1_1:in A),(b1_1:in B)
>>             => (---:that (Tt Uprop Tprodfun2((a1_1
>>             ; b1_1))))]
>>       {move 1}
```

```
     close

construct Prodinductionp ubound Tpairsfunp \
      that Forall (Raiseu (Uof(A**B),Tt,ubound,Inuof (A**B)),Tprodfun2)

>> Prodinductionp: [(.A_1:type),(.B_1:type),
>>        (.Tt_1:type),(ubound_1:that (Uof((.A_1
>>        ** .B_1)) << .Tt_1)),(.Tprodfun2_1:[(x1_2:
>>             in (.A_1 ** .B_1)) => (---:in Univ(.Tt_1))]),
>>        (Tpairsfunp_1:[(a1_3:in .A_1),(b1_3:
>>             in .B_1) => (---:that (.Tt_1 Uprop
>>             .Tprodfun2_1((a1_3 ; b1_3))))])
>>        => (---:that (Raiseu(Uof((.A_1 ** .B_1)),
>>        .Tt_1,ubound_1,Inuof((.A_1 ** .B_1)))
>>        Forall .Tprodfun2_1))]
>>   {move 0}
```

```
rewritec Uofand p q Uofp(p & q) (Uofp p) maxu Uofp q

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



>> Uofand': that Uofand''(Uofp((p & q))) {move
>>   1}



>> Uofand: [(p_1:prop),(q_1:prop),(Uofand''_1:
>>        [(Uofand'''_2:type) => (---:prop)]),
>>        (Uofand'_1:that Uofand''_1(Uofp((p_1
>>        & q_1)))) => (---:that Uofand''_1((Uofp(p_1)
>>        maxu Uofp(q_1))))]
>>   {move 0}
```

we develop the induction principle for the unit type.

```
declare xunit1 in Unit

>> xunit1: in Unit {move 1}


declare  utype [xunit1 => type]

>> utype: [(xunit1_1:in Unit) => (---:type)]
>>   {move 1}


declare uvalue in utype unit

>> uvalue: in utype(unit) {move 1}


declare xunit in Unit

>> xunit: in Unit {move 1}


construct Unitind utype, uvalue, xunit in utype xunit

>> Unitind: [(utype_1:[(xunit1_2:in Unit) =>
>>             (---:type)]),
>>        (uvalue_1:in utype_1(unit)),(xunit_1:
>>        in Unit) => (---:in utype_1(xunit_1))]
>>   {move 0}
```

This is not localized in a universe; we leave that as an exercise. We do not bother to declare the recursor for the unit type, which is as the book says not useful.

We now develop dependent product constructions.

```
clearcurrent

declare Tt type

>> Tt: type {move 1}


declare A in Univ Tt

>> A: in Univ(Tt) {move 1}
```

```
declare a in Utype Tt A

>> a: in (Tt Utype A) {move 1}


declare B  [a => in Univ Tt]

>> B: [(a_1:in (Tt Utype A)) => (---:in Univ(Tt))]
>>   {move 1}


construct Depsum A B : type

>> Depsum: [(.Tt_1:type),(A_1:in Univ(.Tt_1)),
>>         (B_1:[(a_2:in (.Tt_1 Utype A_1)) =>
>>              (---:in Univ(.Tt_1))])
>>         => (---:type)]
>>   {move 0}


declare b in Utype Tt (B a)

>> b: in (Tt Utype B(a)) {move 1}


construct ;;  a b in Depsum A B

>> ;;: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),(a_1:
>>         in (.Tt_1 Utype .A_1)),(.B_1:[(a_2:in
>>              (.Tt_1 Utype .A_1)) => (---:in
>>              Univ(.Tt_1))]),
>>         (b_1:in (.Tt_1 Utype .B_1(a_1))) =>
>>         (---:in (.A_1 Depsum .B_1))]
>>   {move 0}


declare xx in Depsum A B

>> xx: in (A Depsum B) {move 1}


declare Tt2 [xx => in Univ Tt]

>> Tt2: [(xx_1:in (A Depsum B)) => (---:in Univ(Tt))]
```

```
>>   {move 1}


declare g  [a,b=>in Utype Tt Tt2 (a ;; b)]

>> g: [(a_1:in (Tt Utype A)),(b_1:in (Tt Utype
>>        B(a_1))) => (---:in (Tt Utype Tt2((a_1
>>        ;; b_1))))]
>>   {move 1}


rewritec Uofdepsum Tt, Uof(Depsum A B), Univ Tt

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



>> Uofdepsum': that Uofdepsum''(Uof((A Depsum
>>   B))) {move 1}



>> Uofdepsum: [(Tt_1:type),(Uofdepsum''_1:[(Uofdepsum'''_2:
>>             type) => (---:prop)]),
>>        (.A_1:in Univ(Tt_1)),(.B_1:[(a_3:in
>>             (Tt_1 Utype .A_1)) => (---:in Univ(Tt_1))]),
>>        (Uofdepsum'_1:that Uofdepsum''_1(Uof((.A_1
>>        Depsum .B_1)))) => (---:that Uofdepsum''_1(Univ(Tt_1)))]
>>   {move 0}


define Depfunx Tt A B : Depfun A B

>> Depfunx: [(Tt_1:type),(A_1:in Univ(Tt_1)),
>>        (B_1:[(a_2:in (Tt_1 Utype A_1)) => (---:
>>             in Univ(Tt_1))])
>>        => ((A_1 Depfun B_1):type)]
>>   {move 0}


rewritec Techfix2 Tt A B, Utype Tt Inuof(A Depsum B),  A Depsum B

>> Techfix2'': [(Techfix2'''_1:type) => (---:
>>        prop)]
```

```
>>   {move 1}



>> Techfix2': that Techfix2''((Tt Utype Inuof((A
>>   Depsum B)))) {move 1}



>> Techfix2: [(Tt_1:type),(A_1:in Univ(Tt_1)),
>>        (B_1:[(a_2:in (Tt_1 Utype A_1)) => (---:
>>             in Univ(Tt_1))]),
>>        (Techfix2''_1:[(Techfix2'''_3:type)
>>             => (---:prop)]),
>>        (Techfix2'_1:that Techfix2''_1((Tt_1
>>        Utype Inuof((A_1 Depsum B_1))))) =>
>>        (---:that Techfix2''_1((A_1 Depsum B_1)))]
>>   {move 0}
```

```
construct recprodd Tt2, g in Depfunx (Tt,Inuof (Depsum A B),Tt2)

>> recprodd: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),
>>        (.B_1:[(a_2:in (.Tt_1 Utype .A_1)) =>
>>             (---:in Univ(.Tt_1))]),
>>        (Tt2_1:[(xx_3:in (.A_1 Depsum .B_1))
>>             => (---:in Univ(.Tt_1))]),
>>        (g_1:[(a_4:in (.Tt_1 Utype .A_1)),(b_4:
>>             in (.Tt_1 Utype .B_1(a_4))) =>
>>             (---:in (.Tt_1 Utype Tt2_1((a_4
>>             ;; b_4))))])
>>        => (---:in Depfunx(.Tt_1,Inuof((.A_1
>>        Depsum .B_1)),Tt2_1))]
>>   {move 0}
```

Severe technical problems at this point. `Depfunx` was needed because the implicit type inference mechanism doesn't use rewriting, so it cannot infer the type argument `Tt` of `Depfun` from `Inuof (Depsum A B)` (which rewrites to `Univ Tt`). `Techfix2` was needed for the same reasons as the earlier `Techfix`.

```
rewritec Paireval2 a, b, Applyd (recprodd Tt2, g,a;;b), g a, b

>> Paireval2'': [(Paireval2'''_1:in (Tt Utype
>>        Tt2((a ;; b)))) => (---:prop)]
```

```
>>   {move 1}
```

```
>> Paireval2': that Paireval2''((recprodd(Tt2,
>>   g) Applyd (a ;; b))) {move 1}
```

```
>> Paireval2: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),
>>        (a_1:in (.Tt_1 Utype .A_1)),(.B_1:[(a_2:
>>             in (.Tt_1 Utype .A_1)) => (---:
>>             in Univ(.Tt_1))]),
>>        (b_1:in (.Tt_1 Utype .B_1(a_1))),(.Tt2_1:
>>        [(xx_3:in (.A_1 Depsum .B_1)) => (---:
>>             in Univ(.Tt_1))]),
>>        (Paireval2''_1:[(Paireval2'''_4:in (.Tt_1
>>             Utype .Tt2_1((a_1 ;; b_1)))) =>
>>             (---:prop)]),
>>        (.g_1:[(a_5:in (.Tt_1 Utype .A_1)),(b_5:
>>             in (.Tt_1 Utype .B_1(a_5))) =>
>>             (---:in (.Tt_1 Utype .Tt2_1((a_5
>>             ;; b_5))))]),
>>        (Paireval2'_1:that Paireval2''_1((recprodd(.Tt2_1,
>>        .g_1) Applyd (a_1 ;; b_1)))) => (---:
>>        that Paireval2''_1((a_1 .g_1 b_1)))]
>>   {move 0}
```

To get `Paireval2` to work correctly, it was necessary for `Tt2` to be an explicit argument of `g`; something to do with implicit argument inference.

We attack the induction rule, again not in a localized version.

```
declare Ttfun [xx => type]

>> Ttfun: [(xx_1:in (A Depsum B)) => (---:type)]
>>   {move 1}
```

```
declare Ttprop [xx => prop]

>> Ttprop: [(xx_1:in (A Depsum B)) => (---:prop)]
>>   {move 1}
```

```
declare Ttpairfun [a,b => in Ttfun(a;;b)]

>> Ttpairfun: [(a_1:in (Tt Utype A)),(b_1:in
>>        (Tt Utype B(a_1))) => (---:in Ttfun((a_1
>>        ;; b_1)))]
>>   {move 1}


declare Ttpairprop [a,b => that Ttprop(a;;b)]

>> Ttpairprop: [(a_1:in (Tt Utype A)),(b_1:in
>>        (Tt Utype B(a_1))) => (---:that Ttprop((a_1
>>        ;; b_1)))]
>>   {move 1}


declare xxx in Depsum A B

>> xxx: in (A Depsum B) {move 1}


construct Depsumind Ttfun, Ttpairfun, xxx in Ttfun xxx

>> Depsumind: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),
>>        (.B_1:[(a_2:in (.Tt_1 Utype .A_1)) =>
>>             (---:in Univ(.Tt_1))]),
>>        (Ttfun_1:[(xx_3:in (.A_1 Depsum .B_1))
>>             => (---:type)]),
>>        (Ttpairfun_1:[(a_4:in (.Tt_1 Utype .A_1)),
>>             (b_4:in (.Tt_1 Utype .B_1(a_4)))
>>             => (---:in Ttfun_1((a_4 ;; b_4)))]),
>>        (xxx_1:in (.A_1 Depsum .B_1)) => (---:
>>        in Ttfun_1(xxx_1))]
>>   {move 0}


construct Depsumindp Ttprop, Ttpairprop, xxx that Ttprop xxx

>> Depsumindp: [(.Tt_1:type),(.A_1:in Univ(.Tt_1)),
>>        (.B_1:[(a_2:in (.Tt_1 Utype .A_1)) =>
>>             (---:in Univ(.Tt_1))]),
>>        (Ttprop_1:[(xx_3:in (.A_1 Depsum .B_1))
>>             => (---:prop)]),
>>        (Ttpairprop_1:[(a_4:in (.Tt_1 Utype
>>             .A_1)),(b_4:in (.Tt_1 Utype .B_1(a_4)))
>>             => (---:that Ttprop_1((a_4 ;; b_4)))]),
```

```
>>        (xxx_1:in (.A_1 Depsum .B_1)) => (---:
>>        that Ttprop_1(xxx_1))]
>>   {move 0}
```

Localizing this is available as a rather dire exercise. Generally, I should argue that the nonlocal forms of recursion and inductive principles are harmless. I could also try building the local version using the global primitive.

We follow with the existential quantifier.

```
clearcurrent

declare Tt type

>> Tt: type {move 1}


declare A in Univ Tt

>> A: in Univ(Tt) {move 1}


declare a in Utype Tt A

>> a: in (Tt Utype A) {move 1}


declare B [a => in Univ Tt]

>> B: [(a_1:in (Tt Utype A)) => (---:in Univ(Tt))]
>>   {move 1}


construct Exists A B prop

>> Exists: [(.Tt_1:type),(A_1:in Univ(.Tt_1)),
>>        (B_1:[(a_2:in (.Tt_1 Utype A_1)) =>
>>             (---:in Univ(.Tt_1))])
>>        => (---:prop)]
>>   {move 0}


declare existsev that Uprop Tt B(a)

>> existsev: that (Tt Uprop B(a)) {move 1}
```

```
construct Ei Tt A B, existsev that Exists A B

>> Ei: [(Tt_1:type),(A_1:in Univ(Tt_1)),(B_1:
>>        [(a_2:in (Tt_1 Utype A_1)) => (---:in
>>             Univ(Tt_1))]),
>>        (.a_1:in (Tt_1 Utype A_1)),(existsev_1:
>>        that (Tt_1 Uprop B_1(.a_1))) => (---:
>>        that (A_1 Exists B_1))]
>>   {move 0}


declare existsev2 that Exists A B

>> existsev2: that (A Exists B) {move 1}


declare a1 in Utype Tt A

>> a1: in (Tt Utype A) {move 1}


declare witnessev that Uprop Tt B(a1)

>> witnessev: that (Tt Uprop B(a1)) {move 1}


declare Pp prop

>> Pp: prop {move 1}


declare givenw [a1,witnessev => that Pp]

>> givenw: [(a1_1:in (Tt Utype A)),(witnessev_1:
>>        that (Tt Uprop B(a1_1))) => (---:that
>>        Pp)]
>>   {move 1}


construct Eg Tt A B, existsev2, givenw : that Pp

>> Eg: [(Tt_1:type),(A_1:in Univ(Tt_1)),(B_1:
>>        [(a_2:in (Tt_1 Utype A_1)) => (---:in
>>             Univ(Tt_1))]),
```

```
>>        (existsev2_1:that (A_1 Exists B_1)),
>>        (.Pp_1:prop),(givenw_1:[(a1_3:in (Tt_1
>>             Utype A_1)),(witnessev_3:that (Tt_1
>>             Uprop B_1(a1_3))) => (---:that
>>             .Pp_1)])
>>        => (---:that .Pp_1)]
>>   {move 0}
```

This is of course also a dependent product construction, but these traditional constructions should suffice, as we are less interested in manipulating evidence for propositions than we are in manipulating typed mathematical objects.