B.3 Examples

B.3.1 A simple example

We recall the introductory example of Chapter 2, of a simple doubling function.

```
[N:Type(0)][Z:N][S:N->N];
[Even = [n:N]{Phi:N -> Prop}
             {evenZ:Phi Z}
             {evenSS:{k:N}(Phi k)->Phi (S (S k))}
                      Phi n];
Goal <f:N -> N> {n:N}(Even n) -> Even (f n);
Intros #;
Intros n;Refine S;Refine S;
Refine n;
Intros n hyp;Expand Even;
Intros ___;
Prf;
Refine evenSS;
Refine hyp;
Immed;
Undo 1;
Refine evenZ;Refine evenSS;
Save a_simple_example;

a_simple_example;
[plustwo = a_simple_example.1];
[proof = a_simple_example.2];
plustwo;Hnf VReg;
Normal VReg;
proof;Hnf VReg;
Normal VReg;
```

B.3.2 Division by two

Given the discussion in Chapter 4, we simply include the LEGO file here.

```
[divisionby2:Prop]; (* this is just a marker *)

Goal leqNat zero zero;
Refine ExIntro zero;Refine reflEQ;
Save zero_leq_zero;

Goal leqNat zero one;
Refine ExIntro one;Refine reflEQ;
Save zero_leq_one;

Goal {n,q,r|nat}{eq:EQ n (plus q r)}
                    EQ (succ n) (plus r (succ q));
intros;Refine pluscommutes;Refine respEQ succ;Immed;
Save div2_lemma1;

Goal {q,r|nat}{leq:leqNat q r}
                   leqNat (succ q) (succ r);
intros;Refine leq;
intros k eqk;Refine ExIntro k;
Refine pluslemmaS;Refine respEQ succ;Immed;
Save div2_lemma2;

[Div2Spec = [n:nat][p:nat#nat][q=p.1][r=p.2]
     and3 (EQ n (plus q r)) (leqNat q r) (leqNat r (succ q))];

Goal del2 (univPred|nat) (univRel|nat|unit) Div2Spec;

Refine compose_del2;
Refine +3 natrec_del2;

(* zero case *)
Refine pointwise_del2;
intros n u # #;
intros +2 __;Refine pair3;
```

```
Refine reflEQ (plus zero zero);
Refine zero_leq_zero;
Refine zero_leq_one;

(* successor case *)
Refine pointwise_del2;
intros k p #;
[q=p.1][r=p.2];
Refine (r,succ q);

intros _ ih;Refine pair3;
Refine div2_lemma1;Refine and3_out1 ih;
Refine and3_out3 ih;
Refine div2_lemma2;Refine and3_out2 ih;

Save Div2_del2;

[div2 = (Normal [n:nat](Div2_del2.1 n void))];
(* div2 = [n:nat]natiter (zero,zero)
                        ([b:nat#nat](b.2,succ b.1)) n
   div2 : nat->nat#nat *)

div2 eight;Normal VReg;
(* (succ (succ (succ (succ zero))),succ (succ (succ (succ zero)))) *)

div2 seven;Normal VReg;
(* (succ (succ (succ zero)),succ (succ (succ (succ zero)))) *)
```

B.3.3 Minimum finding in a list

We first give the LEGO script.

```
[minlist:Prop];

[A|Type];

[Inlist [a:A] = listrec false
```

```
               ([b:A][l:list A][phi:Prop]or(EQ a b) phi)];

[r:A->A->bool][R = [a,b:A]EQ tt (r a b)];
[reflR:refl R]
[transR:trans R]
[antisymR:{a,b|A}(R a b)->(R b a)->EQ a b];
[linearR:{a,b:A}or (R a b) (R b a)];

Goal {a,b|A}{notaleb:EQ ff (r a b)}R b a;
intros;orE linearR a b;
intros rab;Refine peano4bool;
Refine transEQ;Refine +2 symEQ;Immed;

intros rba;Immed;
Save linearRlemma;

[min [a,b:A] = if (r a b) a b];

[MinSpec [l:list A][f:A->A] =
           {a:A}and (Inlist (f a) (cons a l))
                    (Lelist R (f a) (cons a l))];

Goal del2 (univPred|(list A)) (univRel|(list A)|unit) MinSpec;
Refine compose_del2;
Refine +3 Listrec_del2;

(* base case *)
Refine pointwise_del2;
intros l u #;
Intros +1 __ a;
Refine pair;
Refine inl;
Refine reflEQ (I a);
Refine pair;
Refine reflR;
Intros;Immed;

(* step case *)
```

```
intros b;Refine pointwise_del2;
intros l f #;Refine [a:A]min a (f b);
Intros _ spec a;Refine spec b;
intros inlist lelist;
Refine boolIsInductive (r a (f b));

intros case1;
Refine case1 [bb:bool][ga = if bb a (f b)]
              and (Inlist ga (cons a (cons b l)))
                  (Lelist R ga (cons a (cons b l)));
Refine pair;Refine inl;Refine reflEQ;
Refine pair;Refine reflR;
Equiv Lelist R a (cons b l);
Refine LelistIsMonotone;Refine transR;Immed;

intros case2;
Refine case2 [bb:bool][ga = if bb a (f b)]
              and (Inlist ga (cons a (cons b l)))
                  (Lelist R ga (cons a (cons b l)));
Refine pair;Refine inr;Immed;
Refine pair;Refine linearRlemma;Immed;

Save MinAux_del2;
(*
[minaux = (Normal [l:list A]MinAux_del2.1 l void)];

minaux = [l:list A]listrec ([a:A]a)
                ([a:A][_:list A][f:A->A][b:A]
                        if (r b (f a)) b (f a))
                   l

minaux : (list A)->A->A
*)
Discharge A;
```

We now include the transcript of the dialogue with LEGO. We omit the proof of the initial lemma.

```
Goal
  ?0 : del2 (univPred|(list A)) (univRel|(list A)|unit) MinSpec
Refine by  compose_del2
  ?3 : Type
  ?7 : Rel (list A) ?3
  ?9 : del2 (univPred|(list A)) (univRel|(list A)|unit) ?7
  ?10 : del2 (univPred|(list A)) ?7 MinSpec
Refine 10 by  Listrec_del2
  ?9 : del2(univPred|(list A))(univRel|(list A)|unit)(nstarRel MinSpec)
  ?14 : {x:A}del2 (univPred|(list A)) MinSpec (cstarRel x MinSpec)
Refine by  pointwise_del2
  ?21 : {a:list A}{b:unit}<c:A->A>
        (univPred|(list A) a)->
        (univRel|(list A)|unit a b)->
         nstarRel MinSpec a c
  ?14 : {x:A}del2 (univPred|(list A)) MinSpec (cstarRel x MinSpec)
intros (2) l u #
  l : list A
  u : unit
  ?23 : A->A
  ?24 : (univPred|(list A) l)->
        (univRel|(list A)|unit l u)->
         nstarRel MinSpec l ?23
Intros (3) _ _ a
  h : univPred|(list A) l
  pre : univRel|(list A)|unit l u
  a : A
  ?25 : and (Inlist (?23 a) (cons a (nil A)))
            (Lelist R (?23 a) (cons a (nil A)))
Refine by  pair
  ?28 : Inlist (?23 a) (cons a (nil A))
  ?29 : Lelist R (?23 a) (cons a (nil A))
Refine by  inl
  ?32 : EQ (?23 a) a
  ?29 : Lelist R (?23 a) (cons a (nil A))
Refine by  reflEQ (I a)
  ?29 : Lelist R (I a) (cons a (nil A))
Refine by  pair
```

```
  ?37 : R (I a) a
  ?38 : true
Refine by  reflR
  ?38 : true
Intros (2)
  A1 : Prop
  H : A1
  ?40 : A1
Immediate
Discharge..  H A1
Discharge..  a pre h
Discharge..  u l
```

This closes the branch corresponding to the base case. The step case now follows.

```
  ?14 : {x:A}del2 (univPred|(list A)) MinSpec (cstarRel x MinSpec)
intros (1) b
  b : A
  ?41 : del2 (univPred|(list A)) MinSpec (cstarRel b MinSpec)
Refine by  pointwise_del2
  ?48 : {a:list A}{b'2:A->A}<c:A->A>
        (univPred|(list A) a)->
        (MinSpec a b'2)->cstarRel b MinSpec a c
intros (2) l f #
  l : list A
  f : A->A
  ?50 : A->A
  ?51 : (univPred|(list A) l)->
        (MinSpec l f)->cstarRel b MinSpec l ?50
Refine by  [a:A]min a (f b)
  ?51 : (univPred|(list A) l)->
        (MinSpec l f)->cstarRel b MinSpec l ([a:A]min a (f b))
Intros (3) _ spec a
  h : univPred|(list A) l
  spec : MinSpec l f
  a : A
  ?52 : and (Inlist (([a:A]min a (f b)) a) (cons a (cons b l)))
```

```
            (Lelist R (([a:A]min a (f b)) a) (cons a (cons b l)))
Refine by  spec b
  ?54 : (Inlist (f b) (cons b l))->
        (Lelist R (f b) (cons b l))->
        and (Inlist (min a (f b)) (cons a (cons b l)))
            (Lelist R (min a (f b)) (cons a (cons b l)))
intros (2) inlist lelist
  inlist : Inlist (f b) (cons b l)
  lelist : Lelist R (f b) (cons b l)
  ?55 : and (Inlist (min a (f b)) (cons a (cons b l)))
            (Lelist R (min a (f b)) (cons a (cons b l)))
```

We now consider the cases, according as $r\,a\,(fb) = tt, ff$.

```
Refine by  boolIsInductive (r a (f b))
  ?57 : (EQ tt (r a (f b)))->
         and (Inlist (min a (f b)) (cons a (cons b l)))
             (Lelist R (min a (f b)) (cons a (cons b l)))
  ?58 : (EQ ff (r a (f b)))->
         and (Inlist (min a (f b)) (cons a (cons b l)))
             (Lelist R (min a (f b)) (cons a (cons b l)))
intros (1) case1
  case1 : EQ tt (r a (f b))
  ?59 : and (Inlist (min a (f b)) (cons a (cons b l)))
             (Lelist R (min a (f b)) (cons a (cons b l)))
Refine by  case1 ([bb:bool][ga=if bb a (f b)]
           and (Inlist ga (cons a (cons b l)))
               (Lelist R ga (cons a (cons b l))))
  ?66 : and (Inlist (if tt a (f b)) (cons a (cons b l)))
            (Lelist R (if tt a (f b)) (cons a (cons b l)))
Refine by  pair
  ?69 : Inlist (if tt a (f b)) (cons a (cons b l))
  ?70 : Lelist R (if tt a (f b)) (cons a (cons b l))
Refine by  inl
  ?73 : EQ (if tt a (f b)) a
  ?70 : Lelist R (if tt a (f b)) (cons a (cons b l))
Refine by  reflEQ
  ?70 : Lelist R (if tt a (f b)) (cons a (cons b l))
```

```
Refine by  pair
  ?78 : R (if tt a (f b)) a
  ?79 : and (R (if tt a (f b)) b)
            (Lelist R (if tt a (f b)) l)
Refine by  reflR
  ?79 : and (R (if tt a (f b)) b)
            (Lelist R (if tt a (f b)) l)
Equiv
  ?79 : Lelist R a (cons b l)
Refine by  LelistIsMonotone
  ?85 : trans R
  ?86 : A
  ?89 : R a ?86
  ?90 : Lelist R ?86 (cons b l)
Refine by  transR
  ?86 : A
  ?89 : R a ?86
  ?90 : Lelist R ?86 (cons b l)
Immediate
Discharge..  case1
  b : A
  l : list A
  f : A->A
  h : univPred|(list A) l
  spec : MinSpec l f
  a : A
  inlist : Inlist (f b) (cons b l)
  lelist : Lelist R (f b) (cons b l)
  ?58 : (EQ ff (r a (f b)))->
         and (Inlist (min a (f b)) (cons a (cons b l)))
             (Lelist R (min a (f b)) (cons a (cons b l)))
intros (1) case2
  case2 : EQ ff (r a (f b))
  ?91 : and (Inlist (min a (f b)) (cons a (cons b l)))
            (Lelist R (min a (f b)) (cons a (cons b l)))
Refine by  case2 ([bb:bool][ga=if bb a (f b)]
           and (Inlist ga (cons a (cons b l)))
               (Lelist R ga (cons a (cons b l))))
```

```
  ?98 : and (Inlist (if ff a (f b)) (cons a (cons b l)))
            (Lelist R (if ff a (f b)) (cons a (cons b l)))
Refine by  pair
  ?101 : Inlist (if ff a (f b)) (cons a (cons b l))
  ?102 : Lelist R (if ff a (f b)) (cons a (cons b l))
Refine by  inr
  ?105 : or (EQ (if ff a (f b)) b) (Inlist (if ff a (f b)) l)
  ?102 : Lelist R (if ff a (f b)) (cons a (cons b l))
Immediate
  ?102 : Lelist R (if ff a (f b)) (cons a (cons b l))
Refine by  pair
  ?108 : R (if ff a (f b)) a
  ?109 : and (R (if ff a (f b)) b)
             (Lelist R (if ff a (f b)) l)
Refine by  linearRlemma
  ?112 : EQ ff (r a (if ff a (f b)))
  ?109 : and (R (if ff a (f b)) b)
             (Lelist R (if ff a (f b)) l)
Immediate
Discharge..  case2
Discharge..  lelist inlist
Discharge..  a spec h
Discharge..  f l
Discharge..  b
*** QED ***
MinAux_del2  saved
```

Here is the final result

```
Lego> MinAux_del2;
value = compose_del2
         (pointwise_del2
          ([l:list A][u:unit]
           (I,
            [_:univPred|(list A) l][_:univRel|(list A)|unit l u]
            [a:A]pair (inl (reflEQ (I a)))
                      (pair (reflR (I a))
                             ([A1:Prop][H:A1]H)))
```

```
         (Listrec_del2
            MinSpec
              ([b:A]pointwise_del2
                     ([l:list A][f:A->A]([a:A]min a (f b),
                      [_:ONEL l][spec:MinSpec l f][a:A]
                      spec b
                       (and (Inlist (([c:A]min c (f b)) a)
                                    (cons a (cons b l)))
                            (Lelist R (([c:A]min c (f b)) a)
                                     (cons a (cons b l))))
                       ([inlist:Inlist (f b) (cons b l)]
                        [lelist:Lelist R (f b) (cons b l)]
                        boolIsInductive (r a (f b))
                          (and (Inlist (min a (f b))
                                  (cons a (cons b l)))
                               (Lelist R (min a (f b))
                                  (cons a (cons b l))))
                          ([case1:EQ tt (r a (f b))]
                           case1 ([bb:bool][ga=if bb a (f b)]
                            and (Inlist ga (cons a (cons b l)))
                                (Lelist R ga (cons a (cons b l))))
                            (pair (inl (reflEQ (if tt a (f b))))
                               (pair (reflR (if tt a (f b)))
                            (LelistIsMonotone R transR case1 lelist))))
                          ([case2:EQ ff (r a (f b))]
                           case2 ([bb:bool][ga=if bb a (f b)]
                            and (Inlist ga (cons a (cons b l)))
                                (Lelist R ga (cons a (cons b l))))
                            (pair (inr inlist)
                               (pair (linearRlemma case2) lelist)))))

type  = del2 (univPred|(list A)) (univRel|(list A)|unit) MinSpec

Lego> [minaux = (Normal [l:list A]MinAux_del2.1 l void)];
[l:list A]listrecd ([_:list A]A->A) ([t:A]t)
            ([a:A][_:list A][r'4:A->A][a'5:A]
               boolrecd ([_:bool]A) a'5 (r'4 a) (r a'5 (r'4 a))) l
```

B.3.4 Insert sort

We present the script for the proof, rather than the dialogue with the proof-checker. Large examples cease to be intelligible when directly quoted. We begin with the machine-checked proof of Lemma 4.3.12.

```
[insertsortdel:Prop];

(* an oddity: the exchange lemma and its generalisation *)
[s,t,u|Type];
[S|Pred s][T|Pred t];
[P|Rel t s][Q|Rel s u][R|Rel t u];

[liftPred [t,s|Type][S:Pred s] = [y:t][x:s]S x:Rel t s];

Goal (SubRel (composeRel P Q) R)->
       (del2 S  (liftPred T) Q)->
         del2 T (andRel P (liftPred S)) R;
Intros sub FF #;[f = FF.1][F = FF.2];
Intros y x;Refine f;Immed;
Intros y _ x _;Refine sub;
Intros;Refine ex_y;Refine +2 F;
Refine fst pre;Refine snd pre;Immed;
Save exchange_del2;

Discharge P;

[N,P|Rel t s][Q|Rel s u][R|Rel t u];

Goal {sidecondition:SubRel (composeRel P Q) R}
     {FF:del2 S (op N) Q}
      del2 T (andRel N (andRel P (liftPred S))) R;
Intros __#;[f = FF.1][F = FF.2];
intros y x;Refine f;Immed;
Intros;
[hypN = fst pre];
[hypP = fst (snd pre)];
```

```
[hypS = snd (snd pre)];
Refine sidecondition;
Intros;Refine ex_y;Refine +2 F;Immed;
Save generalised_exchange_del2;

Discharge s;

(* Now the proof begins in earnest *)
[A|Type];
[le:A -> A -> bool][Le = [a,b:A]EQ tt (le a b)];
[reflLe:refl Le]
[transLe:trans Le]
[antisymLe:{a,b|A}(Le a b)->(Le b a)->EQ a b];
[linearLe:{a,b:A}or (Le a b) (Le b a)];

Goal {a,b|A}{notaleb:EQ ff (le a b)}Le b a;
intros;orE linearLe a b;
intros aleb;Refine peano4bool;
Refine transEQ;Refine +2 symEQ;Immed;
intros blea;Immed;
Save linearLelemma;

[InsertSortSpec = andRel (Perm|A) (liftPred (Sorted Le))];
[ONEL = univPred|(list A)];
[ONELU = univRel|(list A)|unit];

Goal del2 ONEL ONELU InsertSortSpec;

Refine compose_del2;
Refine +3 Listrec_del2;
Refine pointwise_del2;

(* base case *)
intros l u #;
intros +1 __;Refine pair;
Refine reflPerm;Refine nilSorted Le;

(* step case *)
```

```
intros a;Refine compose_del2;
Refine +3 functional_del2
          [_:list A][p:A#(list A)]cons p.1 p.2;

[Phi_a = [m:list A][p:A#(list A)][n = cons p.1 p.2]
             and (Perm (cons a m) n) (Sorted Le n) ];

Refine exchange_del2;Immed;

(* now consider the side condition *)
(* ?58 : SubRel (composeRel (Perm|A) Phi_a) Phi_a *)
Intros l n hyp;Refine hyp;
intros m perm_lm phi_a_mn;
[sorted_n = fst phi_a_mn];
[perm_mn = snd phi_a_mn];
Refine pair;
Refine transPerm;
Refine +1 consclPerm;
Immed;

(* return to reasoning: now we can use depListrec_del2 *)
Refine compose_del2;
Refine +3 depListrec_del2;

(* base case *)
Refine pointwise_del2;
intros m n # #;
intros +2 __;Refine pair;
Refine reflPerm;
Refine pair;Refine top;Refine top;

(* step case *)
intros b;Refine pointwise_del2;
intros h p #;[c = p.1][k = p.2];
Refine if (le b c) (b,cons c k) (c,cons b k);
intros sorted_bh pre;
(* split the hypotheses *)
[lelist_bh = fst sorted_bh : Lelist Le b h];
```

```
[sorted_h = snd sorted_bh : Sorted Le h];
[perm_ah_ck  = fst pre:Perm (cons a h) (cons c k)];
[sorted_ck = snd pre: Sorted Le (cons c k)];
[lelist_ck = fst sorted_ck : Lelist Le c k];
[sorted_k = snd sorted_ck : Sorted Le k];
```

We now consider cases in the proof of Lemma 4.3.11. The first case is the simple case (i) of the lemma.

```
(* Now consider cases *)
Refine boolIsInductive (le b c);

intros blec;EQrepl (symEQ blec);
Refine pair;

Refine transPerm;
Refine +1 transposePerm|A|a|b|(nil A)|(nil A);
Refine consclPerm;Immed;
Refine pair;Refine pair;Immed;
Refine LelistIsMonotone Le transLe;Immed;

intros notblec;EQrepl (symEQ notblec);
Refine pair;

Refine transPerm;
Refine +1 transposePerm|A|a|b|(nil A)|(nil A);
Refine transPerm;
Refine +2 transposePerm|A|b|c|(nil A)|(nil A);
Refine consclPerm;Immed;

Refine pair;Refine pair;Immed;
Refine linearLelemma notblec;Refine pair;Immed;
```

We then treat case (iii), using Lemma 4.3.10 and finally (ii), using Lemma 4.3.8. This concludes the proof.

```
(* ?185 : Lelist Le b k *)
```

```
Refine boolIsInductive (le a b);

intros aleb;
Refine PermPreservesLelist;
Refine +2 SortedImpliesLelist;
Refine +1 heredPermlemma;
Refine +2 SortedPermsHaveEqualHeads
            Le reflLe antisymLe
            ? sorted_ck perm_ah_ck
            [a:A]Perm (cons a ?) ?;
Immed;
Refine pair;
Refine LelistIsMonotone Le transLe;
Immed;

intros notaleb;
Refine heredLelistlemma;
Refine +1 PermPreservesLelist;
Refine +2 perm_ah_ck;
Refine pair;
Refine linearLelemma notaleb;
Immed;

Save InsertSort_del2;

[insertsort = (Normal [l:list A]InsertSort_del2.1 l void)];
```

B.3.5 The Chinese remainder theorem

Technical preliminaries

The success or failure of many constructive proofs in type theory is highly sensitive to the choice of representation. We discuss some of the main ideas we employed in structuring the proof of Theorem 4.4.1.

Both $Matrix_\iota$ and $Matrix_\epsilon$ are instances of what we term a *generalised Kronecker* δ matrix, in that they represent (the conjunction of) a matrix of propositions $(\Phi_{i,j})$,where, for $1 \leq i, j \leq n$,

$$\Phi_{i,j} =_{\text{def}} \begin{cases} \phi_i & i = j \\ \psi_{i,j} & i \neq j \end{cases}$$

for some propositions ϕ_i, $\psi_{i,j}$. In fact, we may further parametrise the functions ϕ, ψ by allowing them to vary along vectors $\vec{a} =_{\text{def}} \{a_1, \ldots, a_n\}$ and $\vec{b} =_{\text{def}} \{b_1, \ldots, b_n\}$, yielding the following definition.

Definition B.3.1 *Generalised Kronecker* δ
Suppose given types α, β and relations $\phi, \psi{:}\alpha \longrightarrow \beta \longrightarrow Prop$. Given two vectors $\vec{a} =_{\text{def}} \{a_1, \ldots, a_n\},\ (a_i{:}\alpha),\ \vec{b} =_{\text{def}} \{b_1, \ldots, b_n\},\ (b_i{:}\beta)$, we define the *generalised Kronecker* δ to be the matrix of propositions $Kronecker\ \phi\ \psi\ (\vec{a}, \vec{b})$, where

$$(Kronecker\ \phi\ \psi\ (\vec{a}, \vec{b}))_{i,j} =_{\text{def}} \begin{cases} \phi(a_i, b_i) & i = j \\ \psi(a_i, b_j) & i \neq j \end{cases}$$

We consider the following instances of this general construction:

$Matrix_\iota$ we take, for $m, p{:}nat$,

- $\phi(m, p) =_{\text{def}} Coprime(m, p)$
- $\psi(m, p) =_{\text{def}} p \equiv 0 \pmod{m}$

$Matrix_\epsilon$ we take, for $m, b{:}nat$,

- $\phi(m, b) =_{\text{def}} b \equiv 1 \pmod{m}$
- $\psi(m, b) =_{\text{def}} b \equiv 0 \pmod{m}$

The strength of the generalised Kronecker δ is that we may define it by primitive recursion over the *list* of pairs of values from the vectors $\vec{a}, \vec{b}$, using a *zip* function, again definable primitive recursively.

In order to define a *total* zip function, since partial functions are not available to us in ECC, we must pay a price. This is where the subscript checks arise. The zip function we define returns a list of length the minimum of the lengths of $\vec{a}, \vec{b}$. *zip* is defined by recursion on the first list argument, producing a function from lists to lists, itself defined by primitive recursion. In LEGO,

```
[zip = [A,B|Type][l:list A][m:list B]
       listiter ([_:list B]nil (A#B))
                ([a:A][z:(list B)->list (A#B)]
                 [k:list B]listrec (nil (A#B))
                           ([b:B][n:list B][_:list (A#B)]
                                  cons (a,b) (z n)) k)
                l m : {A,B|Type}(list A)->(list B)->list (A#B)
```

In all the instances of zip considered in the proof, $|\vec{a}| = |\vec{b}|$. For such vectors, we may formulate the following induction principle for zip.

Lemma B.3.1 *zip induction*
Suppose given types α, β, and a three-place relation

$$\Phi{:}\alpha \longrightarrow \beta \longrightarrow (\alpha \times \beta) \longrightarrow \mathit{Prop}$$

Then the following rule is derivable:

$$\frac{\Phi\ \mathit{nil}_\alpha\ \mathit{nil}_\beta\ (\mathit{nil}_\alpha,\ \mathit{nil}_\beta) \qquad \begin{array}{c}\forall a{:}\alpha.\ \forall b{:}\beta.\ \forall l{:}\mathit{list}\,\alpha.\ \forall m{:}\mathit{list}\,\beta \\ |l| = |m| \Longrightarrow \Phi\ l\ m\ (\mathit{zip}\ l\ m) \Longrightarrow \\ \Phi\ (a :: l)\ (b :: m)\ ((a,b) :: (\mathit{zip}\ l\ m))\end{array}}{\forall l{:}\mathit{list}\,\alpha.\ \forall m{:}\mathit{list}\,\beta.\ |l| = |m| \Longrightarrow \Phi\ l\ m\ (\mathit{zip}\ l\ m)}$$

Now the Kronecker matrix is definable in LEGO as follows:

```
[Kronecker [A,B|Type][Phi,Psi:Rel A B][as:list A][bs:list B] =

    depListof ([ab:A#B][a = ab.1][b = ab.2][m:list (A#B)]
               and (Phi a b)
```

```
                    (Listof ([cd:A#B][c = cd.1][d = cd.2]
                                 and (Psi a d) (Psi c b)) m))

               (zip as bs)];
```

Using zip induction, we may derive "fold/unfold" characterisation of

$$Kronecker\ \phi\ \psi\ (a :: \vec{a})\ (b :: \vec{b})$$

```
[Kronecker_unfold = ...
     : {A,B|Type}{Phi,Psi|Rel A B}{a|A}{b|B}
       {l|list A}{m|list B}(EQUAL_LENGTH l m)->
       (Kronecker Phi Psi (cons (a,b) (zip l m)))->
        and4 (Phi a b) (Kronecker Phi Psi (zip l m))
              (Listof (Psi a) m) (Listof (op Psi b) l)];
[Kronecker_fold = ...
     : {A,B|Type}{Phi,Psi|Rel A B}{a|A}{b|B}(Phi a b)->
       {l|list A}{m|list B}(EQUAL_LENGTH l m)->
       (Listof (Psi a) m)->(Listof ([c:A]Psi c b) l)->
       (Kronecker Phi Psi (zip l m))->
        Kronecker Phi Psi (cons (a,b) (zip l m))];
```

Once we have established all these technical details, the proof of the initialisation step is very compact.

The proof

The complete proof rests on a very large number of largely trivial arithmetic lemmas, together with some others for the manipulations of the recursive propositions we have considered in this particular formalisation of a proof of Theorem 4.4.1. We do not include them here. We merely give the main constructions of the deliverables (ϵ, E), (ι, I) and (σ, Σ). We also include a pointwise construction of a deliverable for the initial step, to contrast the size of proof required, if we adopt the powerful structuring available in the deliverables approach.

```
(* CRT preliminaries *)
[crt_init:Prop];

(* the specification of the initialisation step *)
[INIT_SPEC [l,m:list zed] = Kronecker Coprime Divides (zip l m)];

Goal {p|zed}{l,m|list zed}
     {coprime_pl:Listof (Coprime p) l}
     {eq:EQUAL_LENGTH l m}{init_lm:INIT_SPEC l m}
     INIT_SPEC l (maplist (mult_zed p) m);

intros;
Refine zip_induction_eq
       ([l,m:list zed][n:list (zed#zed)]
        {coprime_pl:Listof (Coprime p) l}
         {matrix:Kronecker Coprime Divides n}
          INIT_SPEC l (maplist (mult_zed p) m));
Immed;

intros;Immed;
intros;Refine Kronecker_unfold eq1 matrix;
intros coprime_ab matrix_lm
       listof_divides_a listof_divides_b;
Refine Kronecker_fold;
Refine multCoprime;Refine symCoprime;
Refine fst coprime_pl1;Refine coprime_ab;
Refine transEQ eq1;Refine maplistPreservesLength;
Refine maplistPreservesListof ? listof_divides_a;
Intros q div;Refine multDivides;Immed;
Refine ListofPreservesSubPred ? listof_divides_b;
Intros q div;Refine multDivides;Immed;
Refine phi_lmn;Refine snd coprime_pl1;
Immed;

Save maplist_multp_preserves_INIT_SPEC;

(* this is the devious initialisation function *)
```

```
[Bar [ns:list zed]
    = listrec (nil zed)
         ([n:zed][ns:list zed][product:list zed]
           cons (PiList ns) (maplist (mult_zed n) product)) ns];

Goal {ns|list zed}EQUAL_LENGTH ns (Bar ns);
intros;
Refine listind
     [ns:list zed]EQ (listlength ns) (listlength (Bar ns));
Immed;
Refine reflEQ;
intros;Refine respEQ succ;Refine transEQ ih;
Refine maplistPreservesLength;
Save BarPreservesLength;

(* statement of the initial step of the CRT *)

[ONE_CRT [mrs:list (zed#zed)][u:unit] = true];
[PCM [mrs:list (zed#zed)]
   = [ms = maplist (pi1|zed|zed) mrs]PairwiseCoprime ms];
[SUBSCRIPT_CHECK_1 [ms:list zed][mbars:list zed]
     = EQUAL_LENGTH ms mbars];
[CRT_INIT_SPEC [mrs:list (zed#zed)][mbars:list zed]
    =   [ms = maplist (pi1|zed|zed) mrs]
          (andRel INIT_SPEC SUBSCRIPT_CHECK_1) ms mbars];

Goal del1 PCM PairwiseCoprime;
Refine functional_del1;
Save PCM_TO_PC;

Goal del2 PCM ONE_CRT CRT_INIT_SPEC;

Refine compose_del2;
Refine +3 pullback_del2_along_del1 PCM_TO_PC;
Refine +3 depListrec_del2;

(* nil case *)
Refine pointwise_del2;
```

```
intros l u #;Refine nil zed;
intros;Refine pair;Intros;Immed;Refine reflEQ;

(* step case *)
intros p;Refine pointwise_del2;
intros l m #;Refine cons (PiList l) (maplist (mult_zed p) m);

intros hyp pre;
[coprime_pl = fst hyp:Listof (Coprime p) l];
[init_lm = fst pre:INIT_SPEC l m];
[equal_lengths = snd pre:EQUAL_LENGTH l m];

Refine pair;
(* so we can use the equal lengths twice *)
Refine +1 respEQ succ;
Refine Kronecker_fold;
Refine listwise_coprime_implies_coprime_to_product;
Refine coprime_pl;
Refine ?+4;
Refine listwise_divides_multiples;
Refine listwise_divides_product;
Refine maplist_multp_preserves_INIT_SPEC;
Refine coprime_pl;
Refine equal_lengths;Refine init_lm;
Refine maplistPreservesLength;Immed;

Save recursive_CRT_INIT_del2;
```

For comparison, here is the pointwise construction of a proof of the theorem, in which we build the `Bar` algorithm into the statement of the theorem.

```
(* statement of the initial step of the CRT *)
[CRT_INIT = {mrs:list (zed#zed)}
            [ms = maplist (pi1|zed|zed) mrs]
            {pairwise_coprime_ms : PairwiseCoprime ms}
            [mbars = Bar ms]
            INIT_SPEC ms mbars];
```

```
(* here's the proof *)
Goal CRT_INIT;

Intros __;[ms = maplist (pi1|zed|zed) mrs][mbars = Bar ms];
Refine listrec [mrs:list (zed#zed)]
               [ms =  maplist (pi1|zed|zed) mrs]
               {pairwise_coprime_ms:PairwiseCoprime ms}
               [mbars = Bar ms] INIT_SPEC ms mbars;Immed;

Intros;Immed;

intros mr;[m = mr.1];
intros k;
[ks = maplist (pi1|zed|zed) k][kbars = Bar ks];
intros ih pairwise_coprime_mks;
[m_coprime_to_ks = fst pairwise_coprime_mks
                 : Listof (Coprime m) ks];
[pairwise_coprime_ks = snd pairwise_coprime_mks
                     : PairwiseCoprime ks];

Refine listind
  [k:list (zed#zed)][ks =  maplist (pi1|zed|zed) k]
   {pairwise_coprime_ks:PairwiseCoprime ks}
    [kbars = Bar ks]
     {init_kkbars:INIT_SPEC ks kbars}
     {pairwise_coprime_mks:PairwiseCoprime (cons m ks)}
  INIT_SPEC (cons m ks)
      (cons (PiList ks) (maplist (mult_zed m) kbars));
Immed;

intros ___;Refine pair;Refine pair;
Intros __;Refine gen;
Refine zero_zed;Refine one_zed;Refine reflEQ;
Intros;Immed;Intros;Immed;

(* ?31 : ... *)

(* ?34 : INIT_SPEC (zip ks kbars) *)
```

```
Refine +1 ih;Immed;

(* back to ?31 : ... *)

intros ns;[n = ns.1];
intros l;
[ls = maplist (pi1|zed|zed) l][lbars = Bar ls];
intros ih1 pairwise_coprime_nls
       init_kkbars pairwise_coprime_mnls;
[coprime_mn = fst (fst pairwise_coprime_mnls)
   :Coprime m n];
[coprime_m_ls = snd (fst pairwise_coprime_mnls)
   :Listof (Coprime m) ls];
[coprime_n_ls = fst (snd pairwise_coprime_mnls)
   :Listof (Coprime n) ls];
[pairwise_coprime_ls = snd (snd pairwise_coprime_mnls)
   :PairwiseCoprime ls];
(* we now have got to ?56 *)

Refine Kronecker_unfold Coprime Divides ? init_kkbars;
Refine maplistPreservesLength;
Refine BarPreservesLength;

intros coprime_n_Pi_ls init_lnlbars two_listsof;
[listof_divides_n_nls = fst two_listsof
    :Listof (Divides n) (maplist (mult_zed n) lbars)];
[ls_listwise_divide_Pi_ls = snd two_listsof
    :Listof ([a:zed]Divides a (PiList ls)) ls];

Refine Kronecker_fold;
Refine multCoprime;Refine coprime_mn;
Refine listwise_coprime_implies_coprime_to_product;
Refine coprime_m_ls;

Refine respEQ succ;
Refine maplistPreservesLength;
Refine maplistPreservesLength;
Refine BarPreservesLength;
```

```
Refine listwise_divides_multiples;

Refine pair;Refine multDivides';Refine reflDivides;

Refine ListofPreservesSubPred
        ([a:zed]Divides a (PiList ls))
         ([a:zed]Divides a (PiList (cons n ls)));
Intros __;Refine multDivides;Immed;
Refine listwise_divides_product;

Refine Kronecker_fold;
Refine multCoprime;
Refine symCoprime;
Refine coprime_mn;
Refine listwise_coprime_implies_coprime_to_product;
Refine coprime_n_ls;

Refine maplistPreservesLength;
Refine maplistPreservesLength;
Refine BarPreservesLength;

Refine maplistPreservesListof;
Refine +2 listwise_divides_multiples;
Intros __;Refine multDivides;Immed;

Refine ListofPreservesSubPred
        ([a:zed]Divides a (PiList ls))
         ([a:zed]Divides a (PiList (cons m ls)));
Intros __;Refine multDivides;Immed;
Refine listwise_divides_product;

Equiv INIT_SPEC ls
   (maplist (mult_zed m) (maplist (mult_zed n) lbars));
Refine maplist_multn_preserves_INIT_SPEC;
Refine coprime_m_ls;
Refine maplistPreservesLength;
Refine BarPreservesLength;
```

```
Refine init_lnlbars;

Save crt_init_proof; (* phew!!! *)
```

We now return to the other two steps of the proof.

```
(* crt_step.l *)

(* step case of the CRT *)
[crt_step:Prop];

[STEP_SPEC [mrs:list(zed#zed)][xs:list zed]
    = Kronecker
        ([mr:zed#zed][x:zed][m = mr.1][r = mr.2]Mod m x r)
         ([mr:zed#zed][x:zed][m = mr.1]Divides m x)
          (zip mrs xs)];
[SUBSCRIPT_CHECK_2 [mrs:list(zed#zed)][xs:list zed]
        = EQUAL_LENGTH mrs xs];
[CRT_STEP_SPEC = andRel STEP_SPEC SUBSCRIPT_CHECK_2];

(* the Euclidean algorithm as a deliverable *)

[EUCLID:del2 ([mn:zed#zed][m = mn.1][n = mn.2]Coprime m n)
             (univRel|(zed#zed)|unit)
             ([mn:zed#zed][ab:zed#zed]
              [m = mn.1][n = mn.2][a = ab.1][b = ab.2]
              EQ_zed (plus_zed (mult_zed m a)
                               (mult_zed n b)) one_zed)];

[euclid = [m,n:zed]EUCLID.1 (m,n) void];

[Euclid [m,n|zed][p = euclid m n]
        [a = p.1][b = p.2][coprime_mn:Coprime m n] =
        EUCLID.2 |(m,n) coprime_mn |void top
          :EQ_zed (plus_zed (mult_zed m a)
                            (mult_zed n b)) one_zed];
```

```
[crt_step_fn = [mrm:(zed#zed)#zed]
               [m = mrm.1.1][r = mrm.1.2][mbar = mrm.2]
                mult_zed r (mult_zed mbar (euclid m mbar).2)];

(* remaining arithmetic and list lemmas which we need *)

[coprime_euclid:{m,r,n|zed}(Coprime m n)->
                  Mod m (crt_step_fn ((m,r),n)) r];

Goal {m|zed}{mrs|list(zed#zed)}{mbars|list zed}
      {listof:Listof (Divides m) mbars}
        Listof (Divides m)
             (maplist crt_step_fn (zip mrs mbars));

intros;Refine zip_induction
       ([mrs:list (zed#zed)][mbars:list zed]
        [n:list ((zed#zed)#zed)]
        {listof:Listof (Divides m) mbars}
          Listof (Divides m) (maplist crt_step_fn n));
Immed;
Intros;Immed;
Intros;Immed;
intros;
[divides_mb = fst listof1];
[listof_divides = snd listof1:Listof (Divides m) m1];
Refine pair;
Refine multDivides;Refine multDivides';Immed;
Refine phi_lmn;Immed;

Save crt_lemma69;

Goal {mr|zed#zed}{mbar|zed}{mrs|list (zed#zed)}
     {listof:Listof (op Divides mbar) (maplist (pi1|zed|zed) mrs)}
      Listof ([p:zed#zed]Divides p.1 (crt_step_fn (mr,mbar))) mrs;

intros;
Refine listind [mrs:list (zed#zed)]
  {listof:Listof (op Divides mbar) (maplist (pi1|zed|zed) mrs)}
```

```
     Listof ([p:zed#zed]Divides p.1 (crt_step_fn (mr,mbar))) mrs;
Immed;
intros;Immed;
intros;
[divides_bmbar = fst listof1:Divides b.1 mbar];
[listof_divides_k = snd listof1
    :Listof (op Divides mbar) (maplist (pi1|zed|zed) k)];
Refine pair;
Refine multDivides;Refine multDivides';Immed;
Refine ih;Immed;

Save crt_lemma70;


(* with which we resolve the step case *)

Goal del2 (univPred|(list(zed#zed))) CRT_INIT_SPEC CRT_STEP_SPEC;

Refine pointwise_del2;
intros mrs mbars #;[ms = maplist (pi1|zed|zed) mrs];
Refine maplist crt_step_fn (zip mrs mbars);

intros _ crt_init_spec;
[kronecker_delta = fst crt_init_spec:INIT_SPEC ms mbars];
[equal_lengths = snd crt_init_spec:EQUAL_LENGTH ms mbars];
[equal_lengths' : EQUAL_LENGTH mrs mbars
    = transEQ (maplistPreservesLength|?|?|?|mrs) equal_lengths];
Refine pair;

Refine zip_induction_eq
    ([mrs:list(zed#zed)][mbars:list zed]
     [n:list((zed#zed)#zed)]
     [ms = maplist (pi1|zed|zed) mrs]
     {matrix:INIT_SPEC ms mbars}
      STEP_SPEC mrs (maplist crt_step_fn n));Immed;

intros mr mbar mrs mbars ___;
[m = mr.1][r = mr.2];
```

```
Refine Kronecker_unfold ? matrix;
Refine transEQ ? eq;Refine symEQ;
Refine maplistPreservesLength;

intros coprime_mmbar matrix_lm
       listof_divides_m listof_divides_mbar;
Refine Kronecker_fold;
Refine coprime_euclid coprime_mmbar;
Refine transEQ;
Refine +1 zipPreservesEqualLength eq;
Refine maplistPreservesLength;

(* ?69 *) Refine crt_lemma69;Refine listof_divides_m;
(* ?70 *) Refine crt_lemma70;Refine listof_divides_mbar;

Refine phi_lmn matrix_lm;

(* boring subscript checking *)
Refine transEQ;
Refine +1 zipPreservesEqualLength equal_lengths';
Refine maplistPreservesLength;

Save pointwise_CRT_STEP_del2;

(* crt.l *)

(* The Chinese remainder theorem, January 1992 *)
[crt:Prop];
(* Finally, here is the theorem *)

[CRT_SPEC = [mrs:list (zed#zed)][x:zed]
      Listof ([mr:zed#zed][m = mr.1][r = mr.2]Mod m x r) mrs];

[CRT = {mrs:list (zed#zed)}
       [ms = maplist (pi1|zed|zed) mrs]
       {pairwise_coprime:PairwiseCoprime ms}
       <x:zed>CRT_SPEC mrs x];
```

```
Goal del2 (univPred|(list(zed#zed))) CRT_STEP_SPEC CRT_SPEC;

Refine pointwise_del2;

intros mrs xs #;Refine SigmaList xs;
intros _ crt_step_spec;
[step_spec = fst crt_step_spec];
[subscript_check = snd crt_step_spec];

Refine zip_induction_eq
       ([mrs:list(zed#zed)][xs:list zed]
        [n:list ((zed#zed)#zed)]
        {matrix_of_remainders : Kronecker
           ([mr:zed#zed][x:zed][m = mr.1][r = mr.2]Mod m x r)
           ([mr:zed#zed][x:zed][m = mr.1]Divides m x) n}
        CRT_SPEC mrs (SigmaList xs));Immed;

intros mr x mrs xs ___;
[m = mr.1][r = mr.2];
Refine Kronecker_unfold eq matrix_of_remainders;

intros mod_mxr matrix listof_divides_mxs
       listof_divides_msx;Refine pair;

Equiv Mod m (SigmaList (cons x xs1)) (plus_zed zero_zed r);
Refine plus_zed_commutes;
Refine plusRespMod;Refine mod_mxr;
Refine divides_implies_is_zero_modulo;
Refine listwise_divides_implies_divides_sum;
Immed;

Equiv Listof ([a:zed#zed]Mod a.1 (SigmaList (cons x xs1)) a.2) mrs1;
Refine ListofPreservesSubPred2 ?
         listof_divides_msx (phi_lmn matrix);
intros pq divides_p mod_pq;
[p = pq.1][q = pq.2];
Equiv Mod p (SigmaList (cons x xs1)) (plus_zed zero_zed q);
```

```
Refine plusRespMod;
Refine divides_implies_is_zero_modulo;
Refine divides_p;
Refine mod_pq;

Save pointwise_CRT_del2;

(* these are the undischarged assumptions on which it depends

(* datatypes *)
  unit : Type(0)
  void : unit
  unitrecd : {u:unit}{C:unit->Type}(C void)->C u

  nat : Type(0)
  zero : nat
  succ : nat->nat
  natrecd : {C:nat->Type}(C zero)->
            ({n:nat}(C n)->C (succ n))->{a:nat}C a

  bool : Type(0)
  tt : bool
  ff : bool
  boolrecd : {C:bool->Type}(C tt)->(C ff)->{b:bool}C b

  list : Type(0)->Type(0)
  nil : {A:Type(0)}list A
  cons : {A|Type(0)}A->(list A)->list A
  listrecd : {A|Type(0)}{C:(list A)->Type(0)}
              (C (nil A))->
               ({b:A}{k:list A}(C k)->C (cons b k))->
               {l:list A}C l

(* ideal arithmetic *)

  multCoprime : {p,m,n|zed}(Coprime p m)->
                (Coprime p n)->Coprime p (mult_zed m n)
```

```
(* the Euclidean algorithm,
     considered as a second-order deliverable *)
  EUCLID : del2 ([mn:zed#zed]Coprime mn.1 mn.2)
                (univRel|(zed#zed)|unit)
                ([mn,ab:zed#zed]
                  EQ_zed (plus_zed (mult_zed mn.1 ab.1)
                                   (mult_zed mn.2 ab.2))
                          one_zed)

  coprime_euclid : {m,r,n|zed}(Coprime m n)->
                       Mod m (crt_step_fn ((m,r),n)) r
*)
```

Finally, we present the final composition of the three stages of the proof. The last two steps must be pulled back along a trivial deliverable, to ensure correct matching of the types of each term.

```
[CRT_iota_del2 = recursive_CRT_INIT_del2];

Goal del1 PCM (univPred|(list (zed#zed)));
Refine logical_del1; Refine univPredI;
Save PCM_to_1;

[CRT_epsilon_del2
       = pullback_del2_along_del1 PCM_to_1
            pointwise_CRT_STEP_del2        ];

[CRT_sigma_del2
       = pullback_del2_along_del1 PCM_to_1
            pointwise_CRT_del2             ];

[FinalCRT_del2 = compose_del2
                   CRT_iota_del2
                     (compose_del2
                         CRT_epsilon_del2
                            CRT_sigma_del2)
               : del2 PCM ONE_CRT CRT_SPEC ];
```

Bibliography

Note: throughout, "LNM" and "LNCS" refer to the series Lecture Notes in Mathematics, respectively Lecture Notes in Computer Science, published by Springer-Verlag.

Department of Computer Science technical reports are available from Lorraine Edgar. Requests by e-mail should be sent to `lme@dcs.ed.ac.uk`.

LEGO is available by anonymous ftp.

```
ftp ftp.dcs.ed.ac.uk
Name: anonymous
Password: < enter your e-mail address >
cd export/lego
get README
```

Read `README`, which gives full details of how to set up the system. The user's manual [65] is in the ftp directory, along with examples. Any questions regarding LEGO should be directed to Randy Pollack, `rap@dcs.ed.ac.uk`.

[1] P.Aczel, *An Introduction to Inductive Definitions*, in: The Handbook of Mathematical Logic, ed. J.Barwise, North-Holland, Amsterdam 1977.

[2] R.Backhouse, P.Chisholm, and G.Malcolm, *Do-it-yourself Type Theory*, notes for the International Summer School on Constructive Methods in Computer Science, Marktoberdorf 1988.

[3] H.Barendregt, *λ-calculi with types*, survey article in: The Handbook of Logic in Computer Science, eds. S.Abramsky, D.M.Gabbay, and T.S.Maibaum, Oxford University Press, forthcoming.

[4] E.Bishop, *Foundations of Constructive Mathematics*, Ergebnisse der Mathematik und ihrer Grenzgebiete, Series 3, no. 6, Springer-Verlag, 1985.

[5] J.Bell, *Toposes and Local set theories*, Oxford University Press, 1990.

[6] J.Bénabou, *Fibred categories and the foundations of naïve category theory*, JSL, 1985.

[7] S. Berardi, *Type Dependence and Constructive Mathematics*, Ph.D. thesis, Dipartimento di Informatica, Torino, Italy 1990.

[8] E.Bishop, *Foundations of Constructive Analysis*, McGraw-Hill, 1967.

[9] C.Böhm and A.Berarducci, *Automatic synthesis of typed λ-programs on term algebras*, in: Theoretical Computer Science, Vol. 39, North-Holland, Amsterdam, 1985.

[10] R.Boileau and A.Joyal, *La logique des topos*, Annals of Pure and Applied Logic, North-Holland, 1981.

[11] N.G. de Bruijn, *A survey of the project AUTOMATH*, in: [103].

[12] R.M.Burstall, *An approach to Program Specification and Development in Constructions*, talk given at the Workshop on Programming Logic, Båstad, Sweden, May 1989. See also the discussion in [78, pp. 96–99].

[13] R.M.Burstall and J.H.McKinna, *Deliverables: an approach to program development in Constructions*, in [44], also available as a University of Edinburgh technical report ECS-LFCS-91-133.

[14] A.Church, *A simple theory of types*, Journal of Symbolic Logic, Vol. 5, 1940.

[15] R.Constable *et al.*, *Implementing Mathematics with the NuPrl Proof Development System*, Prentice-Hall, New Jersey, 1986.

[16] R.Constable and S.Fraser Smith, *Partial Objects in Constructive Type Theory*, in: Proceedings of the Second LICS Symposium, IEEE, 1987.

[17] T.Coquand and G.Huet, *Constructions: a Higher-order Proof system for mechanizing mathematics*, in: Proceedings *EUROCAL* '85, LNCS 203, Springer-Verlag, 1985.

[18] T.Coquand and G.Huet, *The Theory of Constructions*, in: Information and Computation, Vol. 76, nos. 2/3, Academic Press, 1988.

[19] T.Coquand, *Metamathematical Investigations of a Calculus of Constructions*, in: [43].

[20] T.Coquand and C.Paulin-Mohring, *Inductively defined types*, in: [43].

[21] P-L.Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman Research Notes in Theoretical Computer Science, Pitman, London, 1986.

[22] E.W.Dijkstra, *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*, in: Communications of the ACM, Vol. 18, 1975.

[23] E.W.Dijkstra *A Discipline of Programming*, Prentice-Hall, New Jersey, 1976.

[24] E.W.Dijkstra, *Selected writings on Computing*, Springer-Verlag, 1982.

[25] E.W.Dijkstra, C.S.Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

[26] T.Ehrard, *A Categorical Semantics of Constructions*, in: Proceedings of the Third LICS Symposium, IEEE, 1988.

[27] S.Feferman, *Formal Theories for Transfinite Iterations of Generalised Inductive definitions and some subsystems of Analysis*, in: Intuitionism and Proof Theory, ed.s A.Kino, J.Myhill and R.E.Vesley, North-Holland, Amsterdam 1970.

[28] M.P.Fourman, *The logic of topoi*, in: The Handbook of Mathematical Logic, ed. J.Barwise, North-Holland, 1977.

[29] P.J.Freyd and A.Scedrov, *Categories and Allegories*, North-Holland, Amsterdam, 1990.

[30] P.A.Gardner, *Representing Logics in Type Theory*, Ph.D. thesis, University of Edinburgh, 1992.

[31] H.Geuvers, *The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi*, draft, December 1991.

[32] J-Y.Girard, *Interpretation fonctionelle et élimination des coupures dans l'arithmétique de l'ordre supérieure*, thesis, University of Paris VII, 1972.

[33] H.Goguen and Z.Luo, *Inductive data types: Well-ordering types revisited*, manuscript submitted to Proceedings of the Second Workshop on Logical Frameworks, 1991.

[34] R.Harper, F.Honsell and G.D.Plotkin, *A Framework for Defining Logics*, Journal of the ACM, to appear.

[35] R.Harper, R.J.G.Milner and M.Tofte, *The Definition of Standard ML, Version 3*, Technical Report ECS-LFCS-89-91, LFCS, Dept. of Computer Science, University of Edinburgh, 1989.

[36] R.Harper and R.A.Pollack, *Type checking with universes*, in: Theoretical Computer Science, Vol. 89, North-Holland, Amsterdam, 1991.

[37] S.Hayashi, *Adjunction of semifunctors: categorical structures in nonextensional lambda calculus*, in Theoretical Computer Science, Vol. 41, North-Holland, Amsterdam, 1985.

[38] S.Hayashi, *ATT: Optimised Curry-Howard isomorphism for Program Extraction*, talk given at First Annual BRA Workshop on Logical Frameworks, Antibes, May 1990.

[39] S.Hayashi, *Singleton, Union and Intersection Types for Program Extraction*, in: Proceedings of TACS '91, Sendai, Japan, Springer LNCS 526, Springer-Verlag, 1991.

[40] C.A.R.Hoare, *An axiomatic basis for computer programming*, in: Communications of the ACM, Vol. 12, 1969.

[41] C.A.R.Hoare, *Data refinement in a categorical setting*, Draft, Oxford 1987.

[42] W.A.Howard, *The "formulae-as-types" notion of construction*, in: [103].

[43] G.Huet, T.Coquand, C.Paulin-Mohring *et al.*, *The Calculus of Constructions, Version 4.10, Documentation and user's manual*, Rapports Techniques no.110, Projet Formel, INRIA-Rocquencourt, Paris, August 1989.

[44] G.Huet and G.Plotkin, eds. *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks, Antibes, May 1990,*, distributed electronically to participating BRA sites, January 1991.

[45] J.M.E.Hyland and A.M.Pitts, *The Theory of Constructions: Categorical Semantics and Topos-theoretic models*, in: Proceedings of the AMS Conference on Categories in Computer Science, Boulder, Colorado, 1986.

[46] J.M.E.Hyland, E.P.Robinson and G.Rosolini, *The discrete objects in the effective topos*, Proceedings of the London Mathematical Society, Vol. 60, pp.1–36, Jan 1991.

[47] K.Ireland and M.Rosen, *A classical introduction to modern number theory*, second edition, Springer Graduate Texts in Mathematics no. 84, Springer-Verlag, 1990.

[48] B.P.F.Jacobs, *Categorical Type Theory*, proefschrift, University of Nijmegen, 1991.

[49] P.T.Johnstone, *Topos Theory*, Academic Press, London, 1977.

[50] P.T.Johnstone and R.Paré, eds., *Indexed Categories and their Applications*, Springer LNM 661, Springer-Verlag, 1978.

[51] A.Kock and G.Wraith, *Elementary Toposes*, Aarhus Lecture Notes no. 30, Aarhus Universitet, Denmark, 1971.

[52] G.Kreisel, *Functions, Ordinals, Species*, in: Logic, Philosophy and Methodology of Science III, eds. B.van Rootselaar and J.Staal, North-Holland, Amsterdam, 1982.

[53] J.Lambek and P.J.Scott, *An Introduction to Higher-Order Categorical Logic*, Cambridge Studies in Advanced Mathematics no. 7, Cambridge University Press, Cambridge, England, 1986.

[54] F.W.Lawvere, *Adjointness in foundations*, Dialectica 23, 1969.

[55] F.W.Lawvere, *Equality in hyperdoctrines and the comprehension schema as an adjoint functor*, in: Proceedings of the AMS symposium on Applications of Category Theory, AMS, Providence R.I., 1970.

[56] D.Leivant, *Reasoning about functional programs and complexity classes associated with type disciplines*, in: Proceedings of the 24th IEEE symposium on Foundations of Computer Science, 1983.

[57] Z.Luo, *An Higher-order Calculus and Theory Abstraction*, Technical report ECS-LFCS-88-57, Department of Computer Science, University of Edinburgh, 1988.

[58] Z.Luo, *ECC, an Extended Calculus of Constructions*, in: Proceedings of the Fourth IEEE Conference on Logic in Computer Science, Asilomar, California, 1989.

[59] Z.Luo, *An Extended Calculus of Constructions*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh, June 1990.

[60] Z.Luo, *A problem of adequacy: conservativity of Calculus of Constructions over higher-order logic*, Technical Report ECS-LFCS-90-121, Department of Computer Science, University of Edinburgh, October 1990.

[61] Z.Luo, *Program Specification and Data Refinement in Type Theory*, Technical Report ECS-LFCS-90-131, Department of Computer Science, University of Edinburgh, January 1991.

[62] Z.Luo, *An Higher-order Calculus and Theory Abstraction*, Information and Computation, Vol. 90, No. 1, 1991.

[63] Z.Luo, *A unifying theory of dependent types I*, Technical Report ECS-LFCS-91-154, Department of Computer Science, University of Edinburgh, 1991.

[64] Z.Luo, R.A.Pollack and P.Taylor, *How To Use LEGO (A Preliminary User's Manual)* LFCS Technical Note LFCS-TN-27, October 1989.

[65] The Edinburgh LEGO club, *LEGO User's Manual*, LFCS Technical Report, forthcoming.

[66] J.H.McKinna, *On permutation*, manuscript, Edinburgh, 1991.

[67] Saunders Mac Lane, *Categories for the Working Mathematician*, Springer Graduate Texts in Mathematics, no. 5, Springer-Verlag, 1971.

[68] D.MacQueen, *Using dependent types to express modular structure*, in: Proceedings POPL 13 , 1986.

[69] P.Martin-Löf, *An Intuitionistic Theory of Types: Predicative part*, in: Logic Colloquium 73, North-Holland, Amsterdam, 1975.

[70] P.Martin-Löf, *Constructive Mathematics and Computer Programming*, in: proceedings of the Conference on Logic, Philosophy and Methodology of Science VI, 1979, North-Holland, Amsterdam, 1982.

[71] P.Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, Naples, 1984.

[72] P.Martin-Löf, *On the meaning of the logical constants, and the justification of the logical laws*, Technical Report 2, Scuola di Specializzazione in Logica Matematica, Università di Siena, 1985.

[73] I.Mason, *Hoare's Logic in the LF*, Technical Report ECS-LFCS-87-32, Department of Computer Science, University of Edinburgh, 1987.

[74] M.Mendler, *The Logic of Design*, Ph.D. thesis, University of Edinburgh, forthcoming, 1992.

[75] E.Moggi, *The partial λ-calculus*, Ph.D. thesis, University of Edinburgh, available as LFCS report ECS-LFCS-88-63, 1988.

[76] E.Moggi, *A category-theoretic account of program modules*, in: Mathematical Structures in Computer Science, Vol.1, Cambridge University Press, 1991.

[77] H.Barendregt, E.Barendsen, W.Dekkers, H.Geuvers, B.P.F.Jacobs, lecture notes for the Summer School in λ-calculus, Nijmegen, 1991.

[78] P.Dybjer, L.Hallnäs, B.Nordström, K.Petersson, and J.Smith, editors, *Proceedings of the workshop on Programming Logic*, Programming Methodology Group Report no. 54, University of Göteborg and Chalmers University of Technology, May 1989.

[79] B.Nordström, K.Petersson, and J.Smith, *Programming in Martin-Löf's type theory*, Oxford University Press, 1990.

[80] C-E.Ore, *The ECC extended with inductive types*, draft, Edinburgh, 1989.

[81] C.Paulin-Mohring, *Extracting F_ω's programs from proofs in the Calculus of Constructions*, in: Proceedings POPL89, ACM, 1989.

[82] C.Paulin-Mohring and B.Werner, *Extracting and Executing Programs developed in the Inductive Constructions System: a Progress Report*, in: [44].

[83] D.Pavlovič, *Predicates and Fibrations*, proefschrift, University of Utrecht, 1990.

[84] A.M.Pitts, *Categorical Semantics of Dependent Types*, talk given at SRI International, Menlo Park, California, June 1989, and at the Logic Colloquium, Berlin, 1990.

[85] R.A.Pollack, *The theory of LEGO*, manuscript, Edinburgh, 1989.

[86] R.A.Pollack, *Implicit Syntax*, in: [44].

[87] R.A.Pollack, *Implicit Syntax*, draft of a paper given at the LogFIT Summer School in Proof Theory, Leeds, UK, July 1990.

[88] A.J.Power, *An algebraic formulation of data refinement*, in: Proceedings of MFPS '89, Tulane University, Louisiana, Springer LNCS 442, Springer-Verlag, 1990.

[89] D.Prawitz, *Natural Deduction: a Proof-Theoretic Study*, Almqvist and Wiksell, Stockholm, 1965.

[90] J.C.Reynolds, *Types, abstraction and parametric polymorphism*, in: Information Processing '83, ed. R.E.A.Mason, North-Holland, 1983.

[91] J.C.Reynolds and Qing-Ming Ma, *Parametric polymorphism revisited*, paper presented at the LMS Symposium on Category Theory and Computer Science, Durham, 1991.

[92] A.Salvesen and J.Smith, *On the strength of the subset type in Martin-Löf's type theory*, in: Proceedings of the Third LICS Symposium, IEEE, 1988.

[93] A.Salvesen, *On Information Discharging and Retrieval in Martin-Löf's type theory*, Ph.D. thesis, Institute of Informatics, University of Oslo, 1989.

[94] A.Salvesen, *The Church-Rosser Property for the LF with $\beta\eta$-reduction*, talk given at the First Annual BRA Workshop on Logical Frameworks, Antibes, May 1990.

[95] A.Salvesen, *The Church-Rosser Property for Pure Type Systems with $\beta\eta$-reduction*, manuscript, November 1991.

[96] D.Sannella, *Formal specification of ML programs*, LFCS technical report ECS-LFCS-86-15, Dept. of Computer Science, University of Edinburgh, 1986.

[97] D.Sannella and A.Tarlecki, *Towards formal development of ML programs: foundations and methodology*, LFCS technical report ECS-LFCS-89-71, Dept. of Computer Science, University of Edinburgh, 1989.

[98] D.S.Scott, *Constructive validity*, in: Proceedings of the Symposium on Automatic Demonstration, IRIA, Rennes, Springer LNM 125, Springer-Verlag, 1970.

[99] D.S.Scott, *Identity and Existence in Intuitionistic Logic*, in: Proceedings of the LMS Symposium on Applications of Sheaves, Durham 1977, Springer LNM 753, Springer-Verlag, 1977.

[100] D.S.Scott, *Relating Theories of the λ-calculus*, in: [103].

[101] R.A.G.Seely, *Hyperdoctrines, Natural Deduction and the Beck Condition*, in: Zeitschrift für Mathematische Logik und Grundlagen, Vol. 29, 1983.

[102] R.A.G.Seely, *Locally Cartesian Closed Categories and Type Theory*, in: Mathematical Proceedings of the Cambridge Philosophical Society, Vol. 95, 1984.

[103] J.P.Seldin and J.R.Hindley, eds., *To H.B.Curry, essays in Combinatory Logic, λ-calculus and Formalism*, Academic Press, 1980.

[104] S.Stenlund, *Combinators, λ-calculus and Proof Theory*, D.Reidel, Dordrecht, 1972.

[105] T.Streicher, *Correctness and completeness of a categorical semantics of Constructions*, thesis, Passau, 1989.

[106] P.Taylor, `diagrams3.tex`, available from `pt@doc.ic.ac.uk`.

[107] A.S.Troelstra and D. van Dalen *Constructivism in Mathematics I+II*, North-Holland, Amsterdam, 1988.

[108] P.Wadler, *Theorems for Free!*, in: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, London, ACM, 1989.