Chapter 4

Examples

To illustrate the discussion of the previous chapter, we now turn to some examples of the use of deliverables in small-scale program development. The use of deliverables may seem heavy-handed and even counter-productive for the smaller examples we have considered, but we hope that the final example, a proof of the Chinese remainder theorem, will be some vindication of the methodology. Our choice of examples is to some extent dictated by chestnuts in the literature, by way of comparison with other approaches to formal program development.

The examples we consider here are all second-order deliverables. Chapter 2 contains a worked example of the doubling function, viewed as a recursive first-order deliverable on the natural numbers. We have not studied examples of first-order deliverables which are significantly more complicated than this. Few natural examples arise which do not specify some relationship between input and output. Furthermore, the case for using deliverables as a programming methodology hinges on the strengths and weaknesses of second-order deliverables.

4.1 Division by two

In a thorough illustration of the propositions-as-types paradigm, the authors of [79] give a careful account of division by two. They observe that this is the constructive content of the theorem

$$\forall n : \mathbb{N}\ \exists m : \mathbb{N}\,.(n = 2m) \vee (n = 2m + 1).$$

They give a mathematical proof by induction, and show how this gives rise to an algorithm which requires a certain amount of rephrasing to properly be considered a function from $\mathbb{N}$ to $\mathbb{N}$. Here, with the emphasis being that our choice of algorithm determines the correctness proof, rather than *vice versa*, we present a rather different development.

4.1.1 The mathematical specification

Rather than consider the original specification,

$$\forall n : \mathbb{N}\,.\exists m : \mathbb{N}\,.(n = 2m) \vee (n = 2m + 1)$$

we started from a slightly weaker one, namely

$$\forall n : \mathbb{N}\,.\exists m, m' : \mathbb{N}\,.(n = m + m') \wedge (m \leq m').$$

Our intended algorithm below meets the stronger specification, namely

$$\forall n : \mathbb{N}\,.\exists m, m' : \mathbb{N}\,.(n = m + m') \wedge (m \leq m' \leq m + 1),$$

which, isolating the witness m, certainly implies the original: the disjunction $(n = 2m) \vee (n = 2m + 1)$ becomes a pair of inequalities, since

$$(m \leq m' \leq m + 1) \quad \text{iff} \quad m' = m \vee m' = m + 1.$$

The second specification is instantiated by a function returning pairs of natural numbers; composed with first projection we obtain a function meeting the first specification. What is perhaps interesting to note, though this example is perhaps too artificially small, is that an attempt to derive an algorithm for division by two from the weak specification above using deliverables, led to the additional condition $m' \leq m + 1$, and hence to the complete development we give below.

4.1.2 The correctness proof of our intended algorithm

We consider the prototype algorithm in Standard ML [35]

```
(* ([n/2],[n+1/2]) = *)
fun fst (m,_) = m;
fun natiter z s n = if n=0 then z
                           else s (natiter z s (n-1));
val div2_aux = natiter (0,0) (fn p =>
                                 let val (q,r) = p
                                      in (r,q+1) end);
fun div2 n = fst (div2_aux n);
```

By induction, the function `div2_aux`, which in LEGO is the term

```
natiter (zero,zero) [p:nat#nat][q=p.1][r=p.2](r, succ p)
```

is easily shown to meet the second specification:

base case $(0 = 0 + 0) \wedge (0 \leq 0 \leq 1)$. Immediate.

step case Here, we suppose, by induction, that $(k = x + y) \wedge (x \leq y \leq x + 1)$, we must show that $(k + 1 = y + (x + 1)) \wedge (y \leq x + 1 \leq y + 1)$. This follows trivially from $(x + y) + 1 = y + (x + 1)$ and $x \leq y \Longrightarrow x + 1 \leq y + 1$.

These arithmetic lemmas are as inconsequential as those in [79], but their non-computational nature has been identified, and localised in the proof, rather than the algorithm.

4.1.3 The development in terms of deliverables

Since this example is so straightforward, we give the whole dialogue with LEGO. We first introduce the specification, and seek a function which satisfies it.

Let

$$Div2Spec =_{\text{def}} \lambda n{:}\mathbb{N}.\ \lambda p{:}\mathbb{N}\times\mathbb{N}.\ (n = \pi_1(p)+\pi_2(p))\wedge(\pi_1(p) \leq \pi_2(p) \leq \pi_1(p)+1).$$

We seek

$$\lambda n{:}\mathbb{N}.\ \lambda u{:}unit.\ true \xrightarrow{div_2} Div2Spec \quad [\lambda n{:}\mathbb{N}.\ true]$$

This gives us our first subgoal

$$1 \xrightarrow{?0} Div2Spec \quad [1]$$

```
Lego> [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))];
defn  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))
      Div2Spec : nat->(nat#nat)->Prop

Lego> Goal del2 (univPred|nat) (univRel|nat|unit) Div2Spec;
Goal
  ?0 : del2 (univPred|nat) (univRel|nat|unit) Div2Spec
```

We then decompose this subgoal, into two others

$$?0 \quad \rightsquigarrow \quad 1 \xrightarrow{?9} ?7 \quad [1], \quad ?7 \xrightarrow{?10} Div2Spec \quad [1]$$

Now subgoal ?10 is general enough to match against our natural number recursor, and in the process of matching, subgoals ?3,?7 are completely instantiated. We are left with two subgoals,

$$\begin{array}{rcl} ?9 & \rightsquigarrow & 1 \xrightarrow{?9} \mathit{Div2Spec}[0/n] \quad [1] \\ ?10 & \rightsquigarrow & \mathit{Div2Spec} \xrightarrow{?13} \mathit{Div2Spec}[n+1/n] \quad [1] \end{array}$$

```
Lego> Refine compose_del2;
Refine by  compose_del2
  ?3 : Type
  ?7 : Rel nat ?3
  ?9 : del2 (univPred|nat) (univRel|nat|unit) ?7
  ?10 : del2 (univPred|nat) ?7 Div2Spec

Lego> Refine +3 natrec_del2;
Refine 10 by  natrec_del2
  ?9 : del2 (univPred|nat) (univRel|nat|unit) (zstarRel Div2Spec)
  ?13 : del2 Nat Div2Spec (sstarRel Div2Spec)
```

We now concentrate on the base case ?9. We use the pointwise construction, to instantiate the base case with the value $(0, 0)$, in subgoals ?24, ?25 below. In fact we may exploit unification to do the instantiation, in the course of working on the propositional reasoning in subgoals ?23 . . . ?32. We use the obvious facts that $0 \leq 0$, and $0 \leq 1$. So we work on subgoal ?23.

```
Lego> Refine pointwise_del2;
Refine by  pointwise_del2
  ?20 : {a:nat}{b:unit}
        <c:nat#nat>(univPred|nat a)->(univRel|nat|unit a b)->
         zstarRel Div2Spec a c
  ?13 : del2 Nat Div2Spec (sstarRel Div2Spec)

Lego> intros n u # #;
intros (2) n u #
  n : nat
```

```
  u : unit
  ?24 : nat
  ?25 : nat
  ?23 : (univPred|nat n)->(univRel|nat|unit n u)->
         zstarRel Div2Spec n (?24,?25)
Lego> intros +2 __;Refine pair3;
intros (2) _ _
  h : univPred|nat n
  pre : univRel|nat|unit n u
  ?26 : zstarRel Div2Spec n (?24,?25)
Refine by  pair3
  ?30 : EQ zero (plus ?24 ?25)
  ?31 : leqNat ?24 ?25
  ?32 : leqNat ?25 (succ ?24)
Refine by  reflEQ (plus zero zero)
  ?31 : leqNat zero zero
  ?32 : leqNat zero (succ zero)
Refine by  zero_leq_zero
  ?32 : leqNat zero (succ zero)
Refine by  zero_leq_one
Discharge..  pre h
Discharge..  u n
```

This branch of the tree is now closed.

We return to the step case, subgoal `?13`. Again we use a pointwise construction. There is rather more involved logical reasoning in subgoals `?43` ... `?63`. This is after we have specified the pointwise value which solves subgoal `?42`, in this case

$$\lambda p.\ [q = \pi_1(p)][r = \pi_2(p)](r, q + 1).$$

```
  ?13 : del2 Nat Div2Spec (sstarRel Div2Spec)

Lego> Refine pointwise_del2;
Refine by  pointwise_del2
```

```
  ?40 : {a:nat}{b:nat#nat}<c:nat#nat>
        (Nat a)->(Div2Spec a b)->sstarRel Div2Spec a c
intros (2) k p #
  k : nat
  p : nat#nat
  ?42 : nat#nat
  ?43 : (Nat k)->(Div2Spec k p)->sstarRel Div2Spec k ?42
defn  q = p.1
      q : nat
defn  r = p.2
      r : nat
Lego> Refine (r,succ q);
Refine by  (r,succ q)
  ?43 : (Nat k)->(Div2Spec k p)->sstarRel Div2Spec k (r,succ q)
Lego> intros _ ih;Refine pair3;
intros (2) _ ih
  h : Nat k
  ih : Div2Spec k p
  ?44 : sstarRel Div2Spec k (r,succ q)
Refine by  pair3
  ?48 : EQ (succ k) (plus r (succ q))
  ?49 : leqNat r (succ q)
  ?50 : leqNat (succ q) (succ r)
```

At this point we use the easily proven lemma `div2_lemma1`, which proves that $k = q + r \implies k + 1 = r + (q + 1)$, and then use that part of the induction hypothesis `ih` which proves $k = q + r$.

```
Lego> Refine div2_lemma1;Refine and3_out1 ih;
Refine by  div2_lemma1
  ?54 : EQ k (plus q r)
  ?49 : leqNat r (succ q)
  ?50 : leqNat (succ q) (succ r)
Refine by  and3_out1 ih
  ?49 : leqNat r (succ q)
  ?50 : leqNat (succ q) (succ r)
```

Now we recover $r \leq q + 1$ from the induction hypothesis.

```
Lego> Refine and3_out3 ih;
Refine by  and3_out3 ih
  ?50 : leqNat (succ q) (succ r)
```

Lastly, to close the proof, we use another trivial lemma `div2_lemma2`, which proves that $q \leq r \implies q + 1 \leq r + 1$. We use the remaining part of our induction hypothesis to prove $q \leq r$. This is all we need.

```
Lego> Refine div2_lemma2;Refine and3_out2 ih;
Refine by  div2_lemma2
  ?63 : leqNat q r
Refine by  and3_out2 ih
Discharge..  ih h
Discharge..  r q p k
*** QED ***
```

We have finished the proof. So we save the result, and then print it. The complete term is almost too unwieldy to be readable, though we can discern a certain amount of structure from this pretty-printed version.

```
Lego> Save Div2_del2;
Div2_del2  saved
Lego> Div2_del2;
value = compose_del2
         (pointwise_del2
          ([n:nat][u:unit]
           ((zero,zero),
            [_:univPred|nat n][_:univRel|nat|unit n u]
            pair3 (reflEQ (plus zero zero))
                   zero_leq_zero
                   zero_leq_one)
         (natrec_del2
          (pointwise_del2
           ([k:nat][p:nat#nat][q=p.1][r=p.2]
            ((r,succ q),
```

```
              [_:Nat k][ih:Div2Spec k p]
              pair3 (div2_lemma1 (and3_out1 ih))
                    (and3_out3 ih)
                    (div2_lemma2 (and3_out2 ih))))))

type  = del2 (univPred|nat) (univRel|nat|unit) Div2Spec
```

However, we may extract the algorithm `div2_aux`, using first projection and then normalisation. We also use normalisation to actually compute with `div2_aux`.

```
Lego> [n:nat](Div2_del2.1 n void);
value = [n:nat]Div2_del2.1 n void
type  = nat->nat#nat

Lego> Normal VReg;Save div_aux;
[n:nat]natrecd  ([_:nat]nat#nat)
                (zero,zero)
                ([_:nat][b:nat#nat](b.2,succ b.1)) n

Lego> div2_aux eight;Normal VReg;
value = div2_aux eight
type  = nat#nat
(succ (succ (succ (succ zero))),succ (succ (succ (succ zero))))

Lego> div2_aux seven;Normal VReg;
value = div2_aux seven
type  = nat#nat
(succ (succ (succ zero)),succ (succ (succ (succ zero))))
```

4.2 Finding the minimum of a list

We now turn to an example which is treated several times in the literature, that of minimum finding in a list [81,96]. It makes non-trivial use of the semi-cartesian closed structure in the fibres $\mathbf{del}_2$.

4.2.1 The mathematical specification

Suppose R is a decidable total order on some type A, which for the purposes of this example contains a maximal element a_0 (this avoids having to consider exceptions for the case of the nil list). We distinguish between R, a *boolean* valued function, and the *relation* $\lambda a, b{:}A.\ R\,a\,b = tt$, denoted $\leq_R$. Then we may specify the minimum of a list as follows:

$$\forall l{:}listA.l \neq nil \Rightarrow \exists m{:}A.m \in l \wedge (\forall a{:}A.a \in l \Rightarrow m \leq_R a).$$

We abbreviate the second conjunct to $m \preceq_R l$. We may easily express a solution to this specification as follows:

```
fun min a b = if (R a b) then a else b;

fun minelemaux nil = (fn a => a) |
    minelemaux (b :: l) = (fn a => (min a (minelemaux l b));

fun minelem nil = a_0 |
    minelem a::l = minelemaux l a;
```

We have explicitly curried the function definition of [96][1]

```
fun minelem (a::nil) = a
 |  minelem(a::b::l) = min a (minelem (b::l));
```

since the type system of ECC is too strict to allow such a definition based on pattern matching. Our definition is by recursion on the first argument l of `minelemaux`; the corresponding proof is by induction on l. We seek to verify that `minelemaux` meets the following specification:

$$\forall l{:}listA.\exists f{:}A \rightarrow A.\forall a{:}A.\ f a \in a :: l \wedge f a \preceq_R a :: l;$$

[1] Sannella in fact treats the *maximum* of the list.

The proof for `minelem` follows by composition with a (suitably relativised) deliverable for application.

4.2.2 The correctness proof of our intended algorithm

We just consider the verification of `minelemaux`. As indicated, the proof is by induction.

Base case

$$\exists f{:}A \to A.\forall a{:}A.\ fa \in a :: nil \land fa \preceq_R a :: nil.$$

The condition $fa \in a :: nil$ forces us to choose $f =_{\text{def}} \lambda a{:}A.\ a$. For a reflexive R, the second conjunct is then satisfied.

Step case Suppose

$$\exists f{:}A \to A.\forall a{:}A.\ fa \in a :: k \land fa \preceq_R a :: k.$$

Then for $b \in A$, taking $g =_{\text{def}} \lambda a{:}A.\ min\ a\ (f\ b)$, we obtain for all $a \in A$, by cases on $R\ a\ (f\ b)$:

- $R\ a\ (f\ b) = true$, and hence $min\ a\ (f\ b) = a$. Now $a \in a :: b :: k$, and $a \preceq_R a :: b :: k$, since $a \leq_R a$, and $a \leq_R (f\ b) \preceq_R b :: k$, by hypothesis, and the transitivity of $\leq_R$.
- $R\ a\ (f\ b) = false$, and hence $min\ a\ (f\ b) = f\ b$. We have $fb \in b :: k$, by hypothesis, and hence $fb \in a :: b :: k$. Again, by hypothesis, $f\ b \preceq_R b :: k$, and $fb \leq_R a$. Hence $fb \preceq_R a :: b :: k$, and we are done.

4.2.3 The development in terms of deliverables

We give a partial dialogue with the proof-checker, in which we quote significant subgoals.

Let A be an arbitrary type, equipped with a boolean-valued linear ordering r.

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

Let

$$MinSpec =_{\mathrm{def}} \lambda l{:}listA.\ \lambda f{:}A \longrightarrow A.\ \forall a{:}A.\ fa \in a :: l \wedge fa \preceq_R a :: l.$$

We seek an arrow

$$\lambda l{:}listA.\ \lambda u{:}unit.\ true \xrightarrow{minelem} MinSpec \quad [\lambda l{:}listA.\ true]$$

```
Lego> [min [a,b:A] = if (r a b) a b];
Lego> [Inlist [a:A] =
             listrec false ([b:A][l:list A][p:Prop]or (EQ a b) p)];

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

Lego> Goal del2 (univPred|(list A)) (univRel|(list A)|unit) MinSpec;
```

yielding

```
  ?0 : del2 (univPred|(list A)) (univRel|(list A)|unit) MinSpec
```

We use composition to enable us to exploit our recursion combinator for second-order deliverables over lists of Section 3.2.7, to package up the proof by induction.

```
Refine compose_del2;Refine +3 Listrec_del2;
```

yielding

```
  ?9 : del2 (univPred|(list A))
                        (univRel|(list A)|unit) (nstarRel MinSpec)
  ?14 : {x:A}del2 (univPred|(list A)) MinSpec (cstarRel x MinSpec)
```

Subgoal `?9` is the base case of our induction. We resolve it using a pointwise construction. The propositional reasoning is relatively straightforward in this case, and the unification is almost strong enough to resolve the "value" goal `?23`. In fact, we must assist the unifier, by using the explicit refinement `Refine reflEQ (I a);` below. This is sufficient to allow us to completely close this branch of the derivation.

```
Lego> Refine 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

Lego> intros l u #;Intros +1 __ 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)))

Lego> Refine pair;
```

This gives us the two conjuncts

```
  ?28 : Inlist (?23 a) (cons a (nil A))
  ?29 : Lelist R (?23 a) (cons a (nil A))
```

which we resolve with

```
Lego> Refine inl;Refine reflEQ (I a);
  ?29 : Lelist R (?23 a) (cons a (nil A))
Lego> Refine pair;Refine reflR;Refine top;
Discharge..  a pre h
Discharge..  u l
  ?14 : {x:A}del2 (univPred|(list A)) MinSpec (cstarRel x MinSpec)
```

This is our step case. We extend the context with a free variable of type A, and then continue with another pointwise construction. In this case we give the function explicitly. We are left with a logical goal `?51`.

```
Lego> intros b;Refine 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

Lego> intros l f #;Refine [a:A]min a (f b);
  ?51 : (univPred|(list A) l)->
        (MinSpec l f)->
         cstarRel b MinSpec l ([a:A]min a (f b))
```

As in the mathematical proof above, we split the subgoal into two cases, `?57` and `?58`, after splitting the hypothesis `spec : MinSpec l f`, to obtain each conjunct. In this case we name them `inlist` and `lelist` respectively.

```
Lego> Intros _ spec a;Refine spec b;
Lego> intros inlist lelist;
Lego> Refine 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)))
```

We give the code for the two cases, according as $a \leq_R f\,b$ or not, which mirrors our informal argument above, but suppress the intermediate output.

```
Lego> intros case1;
Lego> 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)));
Lego> Refine pair;Refine inl;Refine reflEQ;
Lego> Refine pair;Refine reflR;
Lego> Equiv Lelist R a (cons b l);
Lego> Refine LelistIsMonotone;Refine transR;Immed;
Immediate
Discharge..  case1
```

The second case requires the use of an explicit lemma

```
[linearRlemma = ... : {a,b|A}(EQ ff (r a b))->R b a];
```

which transfers the boolean value $r\,a\,(f\,b) = \mathit{ff}$ into the *proposition* $f\,b \leq_R a$, using the linearity of the ordering $\leq_R$.

```
Lego> intros case2;
Lego> 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)));
Lego> Refine pair;Refine inr;Immed;
Lego> Refine pair;Refine linearRlemma;Immed;
Immediate
Discharge..  case2
Discharge..  lelist inlist
Discharge..  a spec h
Discharge..  f l
Discharge..  b
*** QED ***
```

We now save the term. The curious reader may look in Appendix B, where it is printed in full. Again, to extract the algorithm, we use first projection and then normalise the result.

```
Lego> Save MinAux_del2;
Lego> [l:list A]MinAux_del2.1 l void;Normal VReg;

[l:list A]listrecd ([_:list A]A->A)
                   ([t:A]t)
                   ([b:A][_:list A][f:A->A]
                    [a:A]boolrecd ([_:bool]A) a (f b)
                                  (r a (f b)))
                     l
```

4.3 Insert sort

We chose insert sort, since it is expressible naturally within primitive recursion, as opposed to more efficient algorithms such as Hoare's quicksort, which has a natural expression only in terms of general recursion.

4.3.1 The mathematical specification

Informally, this is straightforward enough. For every list l, over some type α which carries a decidable linear ordering, there exists a sorted list m which is a permutation of l. In a formal treatment, we must give explicit representations of the notions of "sortedness" and "permutation". We use the impredicative definition of the permutation relation, written $\sim$, mentioned in Section 3.1. The predicate $Sorted$ is defined by recursion:

$$\frac{}{Sorted(nil)} \qquad \frac{Sorted\,(l) \quad a \preceq l}{Sorted\,(a :: l)}\,[a{:}\alpha, l{:}list\,\alpha]$$

where

$$\frac{}{a \preceq nil \quad [a{:}\alpha]} \qquad \frac{a \preceq l \quad a \leq b}{a \preceq b :: l}\,[a, b{:}\alpha, l{:}list\,\alpha]$$

As we observed in the section on dependent list recursion, $Sorted$ may be seen as an instance of the $depListof$ constructor, while the predicate $\lambda l{:}list\,\alpha.\ a \preceq l$, for $a{:}\alpha$, is just an instance of $Listof$.

We obtain immediately by induction a number of trivial, but useful, lemmas about them. We remark in passing that some of these proved useful in the example of minimum finding, so we were able to benefit from some *reuse* of lemmas.

Lemma 4.3.1 $\forall a{:}\alpha.\forall l{:}list\,\alpha.\ Sorted\,(a :: l) \Longrightarrow Sorted\,(l)$

Lemma 4.3.2 $\forall a{:}\alpha.\forall l{:}list\,\alpha.\ Sorted\,(a :: l) \Longrightarrow a \preceq l$

Lemma 4.3.3 $\forall a, b{:}\alpha.\forall l{:}list\,\alpha.\ a \preceq b :: l \Longrightarrow a \preceq l$

Induction enables us to extend these results to

Lemma 4.3.4 $\forall a{:}\alpha.\forall l, m{:}list\,\alpha.a \preceq l@m \Leftrightarrow (a \preceq m) \wedge (a \preceq l)$

Lemma 4.3.5 $\forall a{:}\alpha.\forall l, m{:}list\,\alpha.Sorted\,(l@m) \Longrightarrow Sorted\,(m) \wedge Sorted\,(l)$

Lemma 4.3.6 $\forall a, b{:}\alpha.\forall l{:}list\,\alpha.a \leq b \wedge b \preceq l \Longrightarrow a \preceq l$

Moreover, given the Definition 3.1.3 of permutation earlier, with its obvious elimination rule, we can prove a derived elimination rule for predicates.

Lemma 4.3.7 *Let S be a property of lists (over α) such that $S(l@m) \Leftrightarrow Sl \wedge Sm$. Then*

$$\frac{l \sim m \quad Sl}{Sm}$$

Hence, as a corollary,

Lemma 4.3.8 *For all* $a{:}\alpha$, $l, m{:}list\,\alpha$

$$\frac{l \sim m \quad a \preceq l}{a \preceq m}.$$

From which we obtain

Lemma 4.3.9 *For all* $a, b{:}\alpha$, $l, m{:}list\,\alpha$,

$$\frac{Sorted\,(a :: l) \quad Sorted\,(b :: m) \quad a :: l \ \sim \ b :: m}{a \leq b}$$

Proof $a \leq a$, by reflexivity of $\leq$, and $a \preceq l$, by Lemma 4.3.2, since $Sorted\,(a :: l)$. So, by Lemma 4.3.8, we obtain $a \preceq b :: m$, since $a :: l \ \sim \ b :: m$. Hence, by definition of $\preceq$, $a \leq b$. ■

Hence, as a corollary, by symmetry of the argument, and antisymmetry of $\leq$,

Lemma 4.3.10

$$\frac{Sorted\,(a :: l) \quad Sorted\,(b :: m) \quad a :: l \ \sim \ b :: m}{a = b}$$

4.3.2 A correctness proof for insert sort

Here is the ML prototype for the sort that we wrote, with a view to proving correct:

```
fun listrec n c l = fold ( fn (a,m) => c a m) l n;
val swapcons = fn b => (fn p =>
                        let val (c,m) = p
                         in  (max(c,b), (min(c,b):: m))
                          end);
fun scons a l = let val (b,m) = listrec (a,nil) swapcons  l
                 in b::m
                  end;
val sort = listrec nil scons;
```

and here is its translation into LEGO code:

```
[swapcons = [A|Type][b:A][p:A # (list A)]
            ((min p.1 p.2),cons (max p.1 p.2) m)];

[scons = [A|Type][a:A][l:list A]
         [p = listrec (a,nil) swapcons  l](cons p.1 p.2)];

[sort = listrec nil scons];
```

We anticipate two levels of induction in the correctness proof for this function, since sort is defined by nested recursion. We recall the derived induction principles for recursive program phrases which we used to give an account of recursive deliverables:

$$\text{for } \phi{:}(list\ \alpha) \longrightarrow \beta \longrightarrow Prop \quad n{:}\beta, \quad c{:}\alpha \longrightarrow (list\ \alpha) \longrightarrow \beta \longrightarrow \beta$$

$$\dfrac{\phi\ nil\ n \qquad \begin{matrix} \left[\phi\ l\ b\ \ [a{:}\alpha,\ l{:}list\ \alpha,\ b{:}\beta]\right] \\ \vdots \\ \phi\ \ (a{::}l)\ \ (c\ a\ l\ b) \end{matrix}}{\forall l{:}list\ \alpha.\phi\ l\ (listrec\ n\ c\ l)}$$

a consequence of which is the following induction principle for sorted lists:

$$\text{for } \phi{:}(list\ \alpha) \longrightarrow \beta \longrightarrow Prop \quad n{:}\beta, \quad c{:}\alpha \longrightarrow (list\ \alpha) \longrightarrow \beta \longrightarrow \beta$$

$$\dfrac{\phi\ nil\ n \qquad \begin{matrix} \left[Sorted\ (a{::}l),\ \ \phi\ l\ b\ \ [a{:}\alpha,\ l{:}list\ \alpha,\ b{:}\beta]\right] \\ \vdots \\ \phi\ \ (a{::}l)\ \ (c\ a\ l\ b) \end{matrix}}{\forall l{:}list\ \alpha.\ \ Sorted\ \ l \Longrightarrow \phi\ l\ (listrec\ n\ c\ l)}\ \text{Sorted List Induction}$$

(the only property of $Sorted$ relevant here is that of Lemma 4.3.1). This induction principle exemplifies the motivation for second-order deliverables: namely that a relation ϕ between lists holds, *only* on the condition that one of them is sorted.

The correctness proof proceeds by application of the first induction principle, for the outermost recursion: this produces as subgoals

(a) $Sorted(nil)$, $nil \sim nil$

(b) $\forall a : \alpha . \forall l, m{:}list\,\alpha.\ (Sorted\,(m) \wedge l \sim m) \Longrightarrow$
$(Sorted\,(\text{scons}\ a\ m) \wedge a :: l \sim (\text{scons}\ a\ m))$.

Now (a) is immediate, and (b) reduces to

$$\forall a : \alpha . \forall m{:}list\,\alpha . Sorted\,(m) \Longrightarrow (Sorted\,(\text{scons}\ a\ m) \wedge a :: m \sim (\text{scons}\ a\ m))$$

$\forall$-introduction extends the context with the assumption $[a{:}\alpha]$, and in this extended context we define the relativised specification

$$\psi_a \equiv \lambda m, n{:}list\,\alpha . Sorted\,(n) \wedge a :: m \sim n.$$

Now we may apply Sorted List Induction to yield the following subgoals in the new context:

(c) $a :: nil \sim a :: nil \wedge Sorted\,(a :: nil)$

(d) $\forall b : \alpha . \forall m, n{:}list\,\alpha.\ Sorted\,(b :: m) \wedge \psi_a m n \Longrightarrow \psi_a (b :: m)(\text{swapcons}\ b\ m)$.

(c) is trivial, and (d) reduces to

$$\forall b, c : \alpha . \forall m, n{:}list\,\alpha . Sorted\,(b :: m) \wedge Sorted\,(c :: n) \wedge a :: m \sim c :: n \Longrightarrow$$

$$Sorted\,(\min\,(b, c) :: \max\,(b, c) :: n) \wedge a :: b :: m \sim \min\,(b, c) :: \max\,(b, c) :: n$$

This last goal rests, apart from some trivial properties of $\sim$, on the following:

Lemma 4.3.11 *Suppose $a, b, c \in \alpha$, and $m, n \in list\,\alpha$ such that:*

- $Sorted\,(c :: n)$*;*
- $c :: n \sim a :: m$*;*
- $b \preceq m$.

Then $\max(b, c) \preceq n$.

Proof We have to consider cases, in order to calculate $\max(b, c)$, according as:

(i) $b \leq c$;

(ii) $c \leq b, b \leq a$;

(iii) $c \leq b, a \leq b$.

We must at this point take account of the hypothesis that $\leq$ is *linear* (to exhaust the cases).

(i) $b \leq c \Longrightarrow \max(b, c) = c \preceq n$, by Lemma 4.3.2

(ii) $c \leq b \Longrightarrow \max(b, c) = b$. By definition of $\preceq$, $b \leq a, b \preceq m \Longrightarrow b \preceq a :: m$. $c :: n \sim a :: m \Longrightarrow b \preceq c :: n$. Hence $b \preceq n$.

(iii) $a \leq b, b \preceq m \Longrightarrow \mathit{Sorted}\,(a :: m)$. Hence by Lemma 4.3.10, $a = c$. Hence $m \sim n$, and so $b \preceq n$, by Lemma 4.3.8.

■

4.3.3 The proof recast in terms of deliverables

We have a double recursion in the definition of our program, which translates into a nested list recursion at the level of deliverables. The base case of each induction is resolved with a simple pointwise construction. So too is the inner step case for the verification of the `swapcons` function, with propositional reasoning based on the above lemmas.

However, there is a point of considerable delicacy in the course of this proof. There is a stage at which we must shift from a second-order deliverable over 1, the identically true predicate on lists, defined by the outermost recursion, to the inner recursion on second-order deliverables defined over the predicate $Sorted$. It appears that we must eliminate the *input* parameter l using the permutation relation, which moves the condition $Sorted(m)$ from being a condition on a *result* to being a condition on the *input* to $scons$. This is the reduction in case (b) above, from

$$\forall a : \alpha. \forall l, m{:}list\,\alpha.(Sorted\ (m) \wedge l \sim m) \Longrightarrow (Sorted\ (\text{scons}\ a\ m) \wedge a :: l \sim (\text{scons}\ a\ m)$$

to

$$\forall a : \alpha. \forall m{:}list\,\alpha. Sorted\ (m) \Longrightarrow (Sorted\ (\text{scons}\ a\ m) \wedge a :: m \sim (\text{scons}\ a\ m)$$

This shift allows us to apply the principle of Sorted List Induction.

This is rather inconvenient, and moreover it seems that only *ad hoc* solutions exist to resolve the difficulty. One solution, which violates our the clean mathematical structure of deliverables, is simply to project out the function and proof, perform the logical manipulations which underlie this shift, and then repackage them in a Σ-type. This is not only ugly, but extremely hard to understand if we are attempting a top-down derivation.

The alternative we considered is the following rule, for which it seems very difficult to give any intuitive justification. Accordingly, we simply state it, and observe that it indeed solves this technical problem. We provisionally call the rule *exchange*, because it exchanges the rôles of dependent variable and parameter in a deliverable, subject to a side condition on specifications. A more general statement is possible than that which we give here, but it turns out that this formulation is sufficient to serve the purposes of this particular proof.

Lemma 4.3.12 (Exchange Lemma) *Suppose (s, S) and (t, T) are specifications, and that (u, Q) is a relativised specification with respect to (s, S), and (u, R) is a relativised specification with respect to (t, T). Suppose further that P is a relation on s and t,* *such that $P \otimes Q \subseteq R$, where $\otimes$ is relational composition. Then the following rule is derivable:*

$$\frac{\pi^{\star}T \xrightarrow{(\lambda x{:}s.\ \lambda y{:}t.\ M, \ldots)} Q \quad [S]}{(P \cap \pi^{\star}S) \xrightarrow[(\lambda y{:}t.\ \lambda x{:}s.\ M, \ldots)]{} R \quad [T]}$$

where $\pi^{\star}T =_{\mathrm{def}} \lambda x{:}s.\ \lambda y{:}t.\ T\,y$, and $\pi^{\star}S =_{\mathrm{def}} \lambda y{:}t.\ \lambda x{:}s.\ S\,x$ are T and S "lifted" to the level of relativised specifications.

Proof See Appendix B. ■

Remark The astute reader will see the interest in applying this rule in the case

$$S =_{\mathrm{def}} \mathit{Sorted}, \quad T =_{\mathrm{def}} \lambda m{:}\mathit{list}\,\alpha.\ \mathit{true}, \quad P =_{\mathrm{def}} \sim \mathit{(permutation)}$$

as we shall see below.

4.3.4 The completed development in terms of deliverables

We relegate the details to Appendix B, as this proof is probably too long to be intelligible. We outline the significant refinement steps.

The specification for sorting is given by

```
[InsertSortSpec = andRel (Perm|A) (liftPred (Sorted Le))];
```

where, as in the previous example, `Le` is the associated propositional level ordering, based on a boolean-valued order function `le`:

```
[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)];
```

The predicate `Sorted` and the relation `Lelist` are defined, as indicated above, using the $depListof$ and $Listof$ constructors:

```
[Lelist = [A|Type][Le:Rel A A][a:A]Listof (Le a)];
[Sorted = [A|Type][Le:Rel A A]depListof (Lelist Le)];
```

We make the abbreviations `ONEL` and `ONELU` for the terminal predicate and relation in the top-level goal

$$1 \xrightarrow{?0} InsertSortSpec \quad [1]$$

```
[ONEL = univPred|(list A)];
[ONELU = univRel|(list A)|unit];
```

The outermost recursion uses the $Listrec_2$ combinator, which we apply after splitting the initial goal with the composition operator.

```
Refine compose_del2;
Refine +3 Listrec_del2;
```

The base case of this recursion/induction is resolved by a pointwise construction. Unification solves the value subgoal `?23`, when we solve the propositional subgoal `?28 : Perm|A (nil A) ?23`. Moreover, we can prove that any permutation of the nil list must be equal to nil, hence we obtain the correct instantiation `?23`$\rightsquigarrow$ nil. Also, the nil list is trivially sorted.

The step case is a little more complicated. We first introduce the parameter $a : A$ in the $Listrec_2$ rule. Recall that the algorithm we considered above contains a `let` construct.

```
fun scons a l = let val (b,m) = listrec (a,nil) swapcons  l
                in b::m
                 end;
```

Now composition of deliverables explicates the `let` construct. For the second component of this composition, we just use the second-order deliverable analogue of the trivial construction of Subsection 3.1.11. We then fold this function into a relativised specification, with which we pursue the inner recursion/induction.

```
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)];
```

We are now in a position to exploit our Exchange Lemma 4.3.12. The side condition

```
?58 : SubRel (composeRel (Perm|A) Phi_a) Phi_a
```

is solved by observing that the $Sorted$ conjunct in ϕ_a is unchanged by composition with the $\sim$ relation, while $\sim$ is itself *transitive*, and hence closed under composition with itself. We must also use the fact that $\sim$ is closed under $cons$:

```
[conscl = [A|Type][a:A][S:Rel (list A) (list A)]
                       [m,n:list A]S (cons a m) (cons a n)];
[consclPerm = ... : {A|Type}{a:A}SubRel Perm (conscl a Perm)];
```

As indicated above, we are now in a position to use the $depListrec_2$ combinator. Again, we preface its application with an appeal to composition.

```
Refine compose_del2;
Refine +3 depListrec_del2;
```

The proof concludes by considering a pointwise construction in the base case of the induction, and then an account of the verification of the `swapcons` function. As in Lemma 4.3.11 above, we distinguish three cases, by induction on the possible boolean values of the expressions `(le b c)` and `(le a b)`. A more refined analysis of recursion/induction might allow us to consider this inductive step in terms of combinators for deliverables. We have not pursued this, however.

4.4 The Chinese remainder theorem

We now turn to a more mathematical example in the rich and complex field of number theory. The Chinese remainder theorem, which we will state and prove for the ring of integers $\mathbb{Z}$, requires the use of some technical machinery to formalise it in a convenient way. The achievement in applying the deliverables methodology to the theorem is considerable in terms of managing the complexity of the formal proof.

4.4.1 The mathematical specification

The theorem concerns the solution of simultaneous congruences to pairwise coprime moduli.

Theorem 4.4.1 (Chinese remainder theorem)

$$\forall n \forall m_1 \ldots m_n \forall r_1 \ldots r_n (\bigwedge_{i=1}^{n} 0 \leq r_i \leq m_i) \wedge (\bigwedge_{i \neq j} \gcd(m_i, m_j) = 1) \Longrightarrow$$
$$\exists x \quad \bigwedge_{i=1}^{n} x \equiv r_i \pmod{m_i} \wedge 0 \leq x \leq \prod_{i=1}^{n} m_i$$

We begin by making a few remarks on the specification, and its reification via the algorithm we give below.

Firstly, the quantifier string (where the range of quantification is implicitly defined to be the integers) is of variable length, determined by the outermost quantifier $\forall n$. In formalising this specification, we are faced with two choices:

- a literal interpretation $\forall n{:}\mathbb{N}\,.\,\forall \vec{m}{:}\,\mathbb{Z}^n.\,\forall \vec{r}{:}\,\mathbb{Z}^n.\;\ldots$, where

$$\mathbb{Z}^n =_{\mathrm{def}} natrec\ (\lambda k{:}\mathbb{N}\,.\,Type)\ unit\ (\lambda k{:}\mathbb{N}\,.\lambda\tau{:}\,Type.\ \mathbb{Z}\times\tau)\ n,$$

- the computationally more useful $\forall \vec{mr}{:}list\ \ \mathbb{Z}^2\ldots$[2], where the outermost quantifier is now implicit in the *length* of the list. This considerably simplifies the recursions involved, since the type of the input remains constant throughout the recursion, rather than depending on n. This is in accordance with our choice of simple types in our programming language. However, the price we pay or this choice is the introduction of a number of extraneous "subscript checks" into the proof. This seems unavoidable, even if we take the vector approach above, though we prefer to have this information at the propositional level.

Secondly, we require — largely for the simplification of this presentation, since the Euclidean algorithm has a natural description in terms of general, rather than primitive recursion — the following easy construction:

Lemma 4.4.1 (Euclidean algorithm)

$$\forall m,n{:}\,\mathbb{Z}.\exists a,b{:}\,\mathbb{Z}.am+bn=\gcd(m,n).$$

Moreover, $\gcd(m,n)=1$ *iff* $\exists a,b{:}\,\mathbb{Z}.am+bn=1.$

We assume this without proof, in the form of a second-order deliverable

$$1 \xrightarrow{Euclid} (\mathbb{Z}\times\mathbb{Z},\ \lambda m,n,a,b{:}\,\mathbb{Z}.\ am+bn=1)\quad [(\mathbb{Z}\times\mathbb{Z},\ Coprime)]$$

[2]We use the notation $\vec{mr}$ to indicate that we take a list of $(modulus, remainder)$ pairs as our basic input. When we wish to distinguish the moduli or remainders as separate lists, we will use the notation $\vec{m}$,$\vec{r}$.

where $Coprime$ is the predicate $\lambda p.\ \gcd(\pi_1(p), \pi_2(p)) = 1$, defined on the type of pairs of integers.

Thirdly, there is an issue as to how to represent the integers in our development. This touches on the whole area of data abstraction, which I certainly do not have time to include in this thesis. A possible topic for future research is to explore the use of the categorical techniques developed in this thesis in the context of Luo's work on data refinement in type theory [61], and the work of Hoare and Power on category theory and data refinement [41,88]. Here we confine ourselves to using the familiar representation $\mathbb{Z} = \mathbb{N}^{\,2}/\sim$, where $(a, b) \sim (c, d)$ iff $a + d = c + b$. In terms of deliverables, we only consider the underlying algorithm defined on lists of pair of natural numbers. The proof that it respects the equivalence relation $\sim$ is computationally irrelevant, as far as correctness is concerned.

Lastly, there is a question as to the computational relevance of the lemma above. It turns out that in the proof we present here of the Chinese remainder theorem, taken from [47], there are both relevant and irrelevant applications of the lemma. I do not see how, in a system based on extraction, one could mark the different instances of *the same lemma* , in such a way as to reflect the two uses.

We turn to the proof of Theorem 4.4.1 above.

Proof Firstly, $\{m_1, \ldots, m_n\}$ pairwise coprime implies $\forall 1 \leq i \leq n. \gcd(m_i, p_i) = 1$, where $p_i =_{\text{def}} \prod_{j \neq i} m_j$. The proof of this is a computationally irrelevant application of the Euclidean algorithm. It moreover has the combinatorial advantage of reducing $\frac{1}{2}n(n-1)$ conditions to n conditions, at the cost of calculating the products p_i.

Then we exploit the Euclidean algorithm to calculate a_i, b_i $(1 \leq i \leq n)$ such that $\forall 1 \leq i \leq n.\ a_i m_i + b_i p_i = 1$.

Finally, we compute $x = \sum_{i=1}^{n} r_i b_i p_i$. Using the above properties of b_i, p_i, we establish that x is the desired simultaneous solution.

In other words, an algorithm for solving the simultaneous congruences is given by the composition of the following three steps:

initialisation $\iota : [(m_1, r_1), \ldots, (m_n, r_n)] \mapsto [(m_1, \prod_{j\neq 1} m_j, r_1), \ldots, (m_n, \prod_{j\neq n} m_j, r_n)]$, the significant component of which takes the list of moduli $\vec{m}$ and applies the following function defined in primitive recursion:

$$\begin{aligned} nil &\mapsto nil \\ m :: ms &\mapsto (\Pi ms) :: (maplist\ (\times_{\mathbb{Z}} m)\ ms) \end{aligned}$$

where $\Pi l =_{\text{def}} listrec\ 1\ (\lambda b, k, r.\ b \times_{\mathbb{Z}} r)\ l$ simply multiplies all the elements in a list together.

listwise euclid $\epsilon\ :\ [(m_1, p_1, r_1), \ldots, (m_n, p_1, r_n)]\ \mapsto\ [(b_1, p_1, r_1), \ldots, (b_n, p_n, r_n)]$, where

$$b_i =_{\text{def}} \pi_2(Euclid\ (m_i, p_i))$$

This is simply $maplist\ \lambda a{:}\ \mathbb{Z}^3.\ (\pi_2(Euclid\ (\pi_1 a, \pi_2 a)), \pi_2 a, \pi_3 a)$.

summation $\sigma : [(b_1, p_1, r_1), \ldots, (b_n, p_n, r_n)] \mapsto \sum_{i=1}^{n} b_i p_i r_i$, which is just

$$listrec\ 0\ (\lambda b, k, r.\ \pi_1(b) \times_{\mathbb{Z}} \pi_2(b) \times_{\mathbb{Z}} \pi_3(b) + r).$$

Properly speaking, the theorem also requires that x can be computed within the range $0 \leq x \leq \prod_{i=1}^{n} m_i$, but this step is computationally trivial, and we shall not pursue it here. ■

Remark Clearly, the above algorithm is hardly of optimal complexity, but, for the purposes of this thesis, this is of no concern.

4.4.2 A development in terms of deliverables

We have been deliberately obtuse in our presentation of the above algorithm, emphasising the rôle of the vectors $\vec{m}$, $\vec{r}$ as "ghost" variables. Of course, we believe it is more appropriate to discuss the logical properties of the above stages in terms of second-order deliverables. The persistence of the $\vec{mr}$, both as arguments to the computation and parameters in the proof, directly echoes the discussion in Section 3.2.1.

Accordingly, we wish to extract the above steps ι, ϵ, σ as the function components of three second-order deliverables:

initialisation the corresponding proof I we seek is of the proposition

$$\forall \vec{mr}{:}list\,\mathbb{Z}^2.\ \forall u{:}unit.\ PairwiseCoprime(\vec{m}) \Longrightarrow true \Longrightarrow Matrix_\iota(\vec{mr},\ \iota(\vec{mr},\ u))$$

where $Matrix_\iota$ is a representation of the relativised specification

$$\lambda \vec{m}, \vec{p}.\ \forall 1 \le i \le n.\ \gcd(m_i, p_i) = 1 \wedge \forall j \ne i.\ p_j \equiv 0 \pmod{m_i}.$$

That is, we wish to show

$$1 \xrightarrow{(\iota,\ I)} Matrix_\iota \quad [PairwiseCoprime]$$

listwise euclid here, the proof E we seek is of the proposition

$$\forall \vec{mr}{:}list\ \mathbb{Z}^2.\ \forall \vec{p}{:}list\ \mathbb{Z}.\ true \Longrightarrow Matrix_\iota(\vec{mr},\ \vec{p}) \Longrightarrow Matrix_\epsilon(\vec{mr},\ \epsilon(\vec{mr},\ \vec{p})).$$

The relativised specification $Matrix_\epsilon$ is a representation of

$$\lambda \vec{m}, \vec{b}.\ \forall 1 \le i, j \le n.\ b_j \equiv \delta_{ij} \pmod{m_i}.^3$$

---

[3] δ_{ij} is the Kronecker δ symbol:

$$\delta_{ij} =_{\text{def}} \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

The second-order deliverable we seek to derive is

$$Matrix_\iota \xrightarrow{(\epsilon,\, E)} Matrix_\epsilon \quad [1]$$

summation Finally, the summation step is

$$Matrix_\epsilon \xrightarrow{(\sigma,\, \Sigma)} Vector_\sigma \quad [1]$$

where $Vector_\sigma$ represents

$$\lambda \vec{m} \vec{r}.\ \lambda x{:}\ \mathbb{Z}.\ x \equiv r_j\, \delta_{ij} \pmod{m_i}$$

which establishes that x is indeed the solution to the simultaneous congruence.

The deliverable for the Chinese remainder theorem is then obtained as the composition

$$(\iota,\, I); (\epsilon,\, E); (\sigma,\, \Sigma).$$

In fact, this is not quite the whole story. The relativised specifications $Matrix_\iota$, $Matrix_\epsilon$, $Vector_\sigma$ are based on a complex recursion on their list arguments. This allows us to use the dependent list recursion deliverable constructor $depListrec_2$, but, as we noted in the introduction above, we must enhance the matrices with explicit subscript checks, to the effect that the output list has the same length as the input list. This is a technical consideration forced upon us by the use of lists to represent the quantifier string $\forall \vec{m} \forall \vec{r}$ in the Chinese remainder theorem.

The complete development is shown in Appendix B. We have only really exploited the power of recursive deliverables in the initialisation step. This is partly due to time considerations, and partly on practical grounds. Certainly, we achieved a dramatic reduction in the size of the proof script we needed in the case of $(\iota,\, I)$, comparing a recursive development with a pointwise construction. This is largely thanks to the work invested in the representation of the matrices of propositions.

In the cases of the second and third steps, we had only time to consider pointwise constructions, but found that the scripts were still not very long. The complex recursion used in the definition of the matrices has proved an obstruction, albeit temporary we hope, to a recursive development of (ϵ, E) and (σ, Σ). In any case, it appears that most of the complexity of this proof lies in the initialisation step. This is reflected in the quadratic recursion in the definition of ι, compared to the linear algorithms ϵ, σ.

Chapter 5

Abstract deliverables

From a categorical perspective, the construction of first-order deliverables is just a particular instance, modulo the considerations of semi-structure as indicated in Chapter 3, of Grothendieck's construction for an indexed category, whose base is the base theory of types and functions, and whose fibres are given by the predicates. We discuss this abstract categorical formulation of the notion of deliverable, and examine a particular instance which illustrates these ideas.

Abstracting away from the details of syntax, we obtain a perspective on the idea of deliverables which is applicable across a wide spectrum of semantic frameworks, based on the hyperdoctrines of Lawvere. This allows us to re-interpret our calculations of Chapter 3, with the added simplification of an extensional view of functions and proofs. This means that we do not have to agonise over the correct notion of equality of deliverables, or the non-uniqueness of structure defined by semi-adjunctions.

Specialising this analysis to the particularly simple and well-understood hyperdoctrine given by the subobjects in a topos, further structure emerges. We use the semantics of dependent type theories, developed in recent years in categorical logic, to show how a subsystem of ECC may be interpreted. Thus the wheel turns full circle: we use (a fragment of) the basic type theory to speak *about* the structure of the composite objects we call deliverables, which we originally defined *within* the basic type theory.

5.1 Construction of first order abstract deliverables

If we examine the syntactic definitions we made in Chapter 3, we see that the only structure we have exploited is an underlying (semi-)cartesian closed structure of types and terms corresponding to our programming language, together with the system of predicates, or *assertions* we may make about such programs, defined internally as $Prop$-valued functions. The type-theoretic properties of ECC allow us to define deliverables using the simple type structure of the programs, and the logic of quantifiers. The abstract categorical structure which is sufficient to describe these constructions was elaborated more than twenty years ago by Lawvere [54,55].

5.1.1 Hyperdoctrines

Definition 5.1.1 (Lawvere [54,55]) *hyperdoctrine*
A *hyperdoctrine* is an indexed category $\mathbf{H}$ over a cartesian closed base category $\mathbf{C}$, with cartesian closed fibres, called the *attributes* of $\mathbf{C}$, together with the following additional structure :

Quantification for each $C \xrightarrow{f} D$ in $\mathbf{C}$, the pullback functor $f^{\star}{:}\mathbf{H}[D] \longrightarrow \mathbf{H}[C]$ has left and right adjoints, denoted $\exists_f, \forall_f$;

Coherence the Beck-Chevalley condition holds for $\exists_f, \forall_f$, which intuitively means substitution commutes with the quantifiers; moreover we require that $f^{\star}$ preserves exponentiation in the fibres.

Remark We may think of $\mathbf{C}$ as some representation of a (functional) programming language, so its arrows correspond to the denotations of program phrases. The attributes of $\mathbf{C}$ may then be thought of as an abstract notion of predicates on $\mathbf{C}$, since they are indexed by the objects of $\mathbf{C}$. Morphisms in the fibres $\mathbf{H}[C]$ may then be regarded as *proofs* of entailments. The additional structure given by the quantifiers and coherence ensures that this logical language obeys the usual rules for substitution with respect to the logical operations.

Definition 5.1.2 *first-order abstract deliverables*
Given an indexed category $\mathbf{H}$, Grothendieck's construction applied to $\mathbf{H}$ yields the *first-order abstract deliverables* over $\mathbf{C}$.

Grothendieck's construction defines a category $\mathcal{G}(\mathbf{H})$, together with a functor $p{:}\mathcal{G}(\mathbf{H}) \longrightarrow \mathbf{C}$, which is a fibration (see Appendix A for background material on fibrations). The objects of $\mathcal{G}(\mathbf{H})$ are pairs (C, ϕ), where C is an object of $\mathbf{C}$, and ϕ an object of $\mathbf{H}[C]$. The morphisms in $\mathcal{G}(\mathbf{H})$ are also pairs, where

$$(C, \phi) \xrightarrow{(f, F)} (D, \psi)$$

is an arrow in $\mathcal{G}(\mathbf{H})$ if

- $f{:}C \longrightarrow D$ in $\mathbf{C}$, and
- $F{:}\phi \longrightarrow f^{\star}\psi$ in $\mathbf{H}[C]$.

Theorem 5.1.1 *If $\mathbf{H}$ is a hyperdoctrine, then $\mathcal{G}(\mathbf{H})$ is cartesian closed.*

Proof The constructions of Section 3.1 give a syntactic description of a putative proof of this theorem. We use the cartesian closed structure of $\mathbf{C}$ in the first components of objects in $\mathcal{G}(\mathbf{H})$, and the cartesian closed structure in the fibres, together with the quantifiers, is used in defining the predicates (these are the second components of objects in $\mathcal{G}(\mathbf{H})$). We exploit the coherence of the quantifiers and exponentiation with respect to substitution in the definition of hyperdoctrine to reflect exactly the "logical" constructions of Section 3.1. Indeed, the uniqueness of the categorical structure defined by *bona fide* adjunctions smooths over some of the technical niceties we encountered in ECC.

However, the theorem as it stands is in fact false. The so-called "canonical isomorphisms" in the definition of an indexed category obstruct the proof. The cartesian closed structure is only defined up to these isomorphisms.

Nonetheless, in the model we consider below there is enough degeneracy to allow the construction to proceed smoothly. This is because we work with *posets* of attributes. ∎

In a precisely similar way, we may define second-order abstract deliverables.

Definition 5.1.3 *second-order abstract deliverables*
Given a hyperdoctrine $\mathbf{H}$, and an object (C, ϕ) of $\mathcal{G}(\mathbf{H})$, we define a category of *second-order abstract deliverables over* (C, ϕ), whose

- objects are pairs, of an object D in $\mathbf{C}$, and an attribute $\psi \in \mathbf{H}[C \times D]$,
- arrows are also pairs, where $(D, \psi) \xrightarrow{(f, F)} (E, \chi)$ if
  - $C \times D \xrightarrow{f} E$ in $\mathbf{C}$, and
  - $(\pi_C^\star \phi) \wedge \psi \xrightarrow{F} {<}\pi_C, f{>}^\star \chi$ in $\mathbf{H}[C \times D]$.

Just as we have an abstract counterpart, up to isomorphism, to Theorem 3.1.1 in the above, we may state counterparts to Theorems 3.2.1, 3.2.2 for second-order deliverables in this abstract setting. We leave the formulation, and the consequent wrestling with canonical isomorphisms, to the patient reader.

5.1.2 A programming language with assertions

Scott, in his essay [100], suggests that the understanding of λ-calculus using cartesian closed categories $\mathbf{C}$ should be conservatively extended via the Yoneda embedding to considering the λ-calculus, *qua* theory of functions, as a subtheory of the higher-order logic of the topos $\mathbf{Sets}^{\mathbf{C}^{op}}$. In this chapter, we extend this idea, by observing that precisely those features — subobject classifier, representation of predicates — which make toposes work as models of higher-order logic, allow us to give an interpretation of a subsystem of ECC. This is interesting on two grounds: it gives a new model construction for $\mathcal{CC}^+$; and enables us to use $\mathcal{CC}^+$ as a language for deliverables defined over a model of higher-order logic. This last is probably the most interesting application of the ideas of this thesis. It suggests that we should be able to define complex deliverables using the usual apparatus of λ and Π, rather than the clumsy categorical combinators used in the previous chapters. Moreover, we may rather more directly apply our technology for theorem-proving in typed λ-calculi to the development of programs. The implementation of this idea remains a subject for future research, as I have not had time to investigate the details of such an approach.

5.2 A topos-theoretic model

We now turn to a particular instance of the foregoing analysis, and elaborate a model, the basic ideas of which were first sketched by Burstall in July 1990. Related constructions are discussed, in rather more abstract detail, in [76]. Although the construction we give is valid for an arbitrary topos, and, certainly in view of Scott's position, we should demand that it be so, we exemplify it with the concrete example of the category **Sets**, of sets and functions.

5.2.1 Introduction

This section concerns a naïve construction on the category of sets which gives rise to a model of the Calculus of Constructions (with type constants), $\mathcal{CC}^+$[60]. $\mathcal{CC}^+$ can be viewed as a subsystem of ECC. It was introduced to resolve a conjecture of Luo [60] concerning the conservativity of the Calculus of Constructions over Church's simple theory of types [14][1]. It turns out that if we interpret the type of individuals in Church's system as a term of type $Prop$ in $\mathcal{CC}$, then the Calculus fails to be conservative. What we require is the ability to interpret the type of individuals as some constant of type $Type$. $\mathcal{CC}^+$ is the minimal extension of $\mathcal{CC}$ by the ability to assume a context of type constants. The relationship with topos theory, here exemplified by the category **Sets** of sets and functions, is made by observing that the internal language of a topos is essentially Church's system, with extensionality and a strong form of proof-irrelevance. The language is parametrised in exactly the same way as intended in $\mathcal{CC}^+$, namely we have constant symbols for every object (i.e. *type*) of the topos.

---

[1] We follow the Nijmegen notation, by referring to this system as λHOL[77,3].

We consider this model interesting because the semantics we give for the type theory interprets types in the theory as *specifications*, in the sense of Chapter 3. That is, we interpret a type as a pair, consisting of a set, together with a subset of it. A term of functional type is then one which respects the subsets, i.e. a first-order deliverable. Moreover, we have a model-theoretic connection with Luo's conservativity result. Luo used the projection of Berardi and Mohring [7,81], which essentially erases dependencies at the level of types (while preserving those at the level of propositions), to show the conservativity of $\mathcal{CC}^+$ over λHOL. Here, we see a similar phenomenon: the types of λHOL are the carrier sets of our interpretation below, so we obtain a mapping from $\mathcal{CC}^+$ to λHOL from the first projection functor, which simply maps the pairs onto their carrier sets.

The rules for $\mathcal{CC}^+$

Contexts

$$\frac{}{\circ\ \mathbf{valid}}$$

$$\frac{\Gamma \vdash A\ \mathbf{type}}{\Gamma, x{:}A\ \mathbf{valid}}$$

Type formation

$$\frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash \kappa\ \mathbf{type}}$$

where κ is a constant. This rule is really to do with the allowable signatures — we take some λHOL signature and produce a $\mathcal{CC}^+$ signature — which is more of a semantic consideration in our particular model $\mathcal{B}$. Nonetheless, it is important to stress that the object set(s) of λHOL have syntactic counterparts at the **type** level. This is the nub of Luo's original conjecture on the conservativity of $\mathcal{CC}^+$ over λHOL.

$$(\Pi F)\ \frac{\Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash \Pi x{:}A.B\ \mathbf{type}}$$

$$(Prop)\ \frac{\Gamma\ \mathbf{valid}}{\Gamma \vdash Prop\ \mathbf{type}}$$

$$(Prf)\ \frac{\Gamma \vdash \phi{:}Prop}{\Gamma \vdash Prf(\phi)\ \mathbf{type}}$$

Basic typing

$$(var)\ \frac{\Gamma, x{:}A, \Delta\ \mathbf{valid}}{\Gamma, x{:}A, \Delta \vdash x{:}A}$$

$$(con)\ \frac{\Gamma \vdash \kappa\ \mathbf{type}}{\Gamma, x{:}\kappa \vdash c{:}\kappa}$$

where c is a constant, specified by some signature in higher-order logic.

$$(\Pi I)\ \frac{\Gamma, x{:}A \vdash b{:}B}{\Gamma \vdash \lambda x{:}A.b{:}\,\Pi x{:}A.B}$$

$$(\Pi E)\ \frac{\Gamma \vdash M{:}\Pi x{:}A.B \quad \Gamma \vdash a{:}A}{\Gamma \vdash M a{:}B[a/x]}$$

$$(\forall F)\ \frac{\Gamma, x{:}A \vdash \phi{:}Prop}{\Gamma \vdash \forall x{:}A.\phi{:}Prop}$$

$$(\forall I)\ \frac{\Gamma, x{:}A \vdash p{:}Prf(\phi)}{\Gamma \vdash \Lambda x{:}A.p{:}Prf(\forall x{:}A.\phi)}$$

$$(\forall E)\ \frac{\Gamma \vdash p{:}Prf(\forall x{:}A.\phi) \quad \Gamma \vdash a{:}A}{\Gamma \vdash p\,a{:}Prf(\phi[a/x])}$$

Conversion

$$\frac{\Gamma \vdash M{:}A \quad \Gamma \vdash B\ \mathbf{type} \quad A \simeq B}{\Gamma \vdash M{:}B}$$

We take the conversion relation $\simeq$ here to be β-conversion. Although η holds in toposes, instead we will consider an explicit assumption of extensionality.

Substitution (derivable in the presence of β-conversion and the above Π rules)

$$\frac{\Gamma, x{:}A \vdash B\ \mathbf{type} \quad \Gamma, x{:}A \vdash b{:}B \quad \Gamma \vdash a{:}A}{\Gamma \vdash b[a/x]{:}B[a/x]}$$

5.2.2 Modelling dependent types (following Hyland, Pitts)

In [45], a general framework for modelling dependent type theories using fibrations [6,46] was developed, with emphasis on the Calculus of Constructions as a particular case. In [84], a refinement was made essentially to the presentations of the data of the model (via the so-called "categories with fibrations"), which we shall employ here. For the interested reader, Appendix A contains some introductory material on fibrations in general, though this is not essential to our development. Our treatment closely follows that of [84].

5.2.3 Categories with fibrations

In the case of simply-typed equational theories, categorical logic provides a very simple semantic paradigm: types correspond to objects, and terms to morphisms (where the codomain corresponds to the type of the term, and the domain to the context in which the term is well-typed). Substitution then corresponds to composition. A well-known example is simply-typed λ-calculus in cartesian closed categories [53, for example]. Capturing the type-theoretic idea of dependent type requires more sophisticated machinery, as there is the obvious interaction between contexts and the well-formedness of types, as well as the well-typedness of terms.

Definition 5.2.1 (Pitts) *category with fibrations*
Let $\mathcal{B}$ be a category with a terminal object. We say that $\mathcal{B}$ is a *category with fibrations* if, for every X an object of $\mathcal{B}$, we have a collection $Fib(X)$ such that:

- for every X an object of $\mathcal{B}$, and A in $Fib(X)$, there is an object $X \bullet A$ of $\mathcal{B}$, together with an arrow $d_A : X \bullet A \longrightarrow X$ in $\mathcal{B}$ (called a *display* map), and

- for every $f : Y \longrightarrow X$ an arrow of $\mathcal{B}$, and A in $Fib(X)$, there is a $f^\star A$ in $Fib(Y)$, together with an arrow $f \bullet A : Y \bullet f^\star A \longrightarrow X \bullet A$ such that

$$\begin{tikzcd} Y \bullet f^\star A \arrow[r, "f \bullet A"] \arrow[d, "d_{f^\star A}"'] \arrow[dr, phantom, "\lrcorner"] & X \bullet A \arrow[d, "d_A"] \\ Y \arrow[r, "f"'] & X \end{tikzcd}$$

  is a pullback in $\mathcal{B}$ (we do not require that $\mathcal{B}$ has all pullbacks).

5.2.4 Interpreting types and terms

Given a category $\mathcal{B}$ with fibrations, the base $\mathcal{B}$ is used to model contexts and well-typed substitutions, while an element of $Fib(\Gamma)$ models the types which are well-formed in context Γ, i.e. those A such that $\Gamma \vdash A$ **type** is derivable. From the rule for valid contexts

$$\frac{\Gamma \vdash A \ \mathbf{type}}{\Gamma, x{:}A \ \mathbf{valid}}$$

we see that the operation $\bullet$ corresponds to context extension. Now, to model the judgment $\Gamma \vdash a : A$, where $\Gamma \vdash A$ **type** , we take *sections* s of d_A (i.e. $s; d_A = id$)

$$\begin{tikzcd}
X \arrow[r, "s"] \arrow[dr, "id_X"'] & X \bullet A \arrow[d, "d_A"] \\
& X
\end{tikzcd}$$

This extends the simply-typed case, where context extension ($\bullet$) is given simply by cartesian product ($\times$) on objects, with display maps the product projections. In this situation every term-in-context (morphism) gives rise to a section of a display map simply by pairing with the appropriate identity.

In particular, we may give a semantics for the rule

$$(var)\frac{\Gamma, x{:}A, \Delta \ \textbf{valid}}{\Gamma, x{:}A, \Delta \vdash x{:}A}$$

as follows: given $[\![\Gamma]\!] \in \mathcal{B}$, $[\![A]\!] \in Fib([\![\Gamma]\!])$, $[\![\Delta]\!] \in Fib([\![\Gamma]\!] \bullet [\![A]\!])$, i.e. data for the hypothesis, we seek a section

$$\begin{tikzcd}
[\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!] \arrow[r, "s"] \arrow[dr, "id"'] & [\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!] \bullet A' \arrow[d, "d_{A'}"] \\
& [\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!]
\end{tikzcd}$$

where A' is the object in $Fib([\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!])$ corresponding to $\Gamma, x{:}A, \Delta \vdash A$ **type** . The (essentially) unique s is obtained as the mediating arrow to the following pullback

$$\begin{tikzcd}
[\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!] \arrow[r, dotted, "\exists! s"] \arrow[dr, "id"'] & [\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!] \bullet A' \arrow[r] \arrow[d, "d_{A'}"] \arrow[dr, phantom, "\lrcorner", very near start] & [\![\Gamma]\!] \bullet A \arrow[d, "d_A"] \\
& [\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!] \arrow[r, "f_\Delta ; d_A"'] & [\![\Gamma]\!]
\end{tikzcd}$$

where $A' = (f_\Delta; d_A)^\star A$ and $f_\Delta : [\![\Gamma]\!] \bullet [\![A]\!] \bullet [\![\Delta]\!] \longrightarrow [\![\Gamma]\!] \bullet [\![A]\!]$ is just a composite of the d, defined by induction on the length of Δ.

The semantics of substitution is a straightforward generalisation of this. The existence of $f^\star A$ and the pullback squares in the definition ensures that we may obtain new types by substituting terms for variables in types, and that these are well-behaved with respect to substitution(in the language of fibrations, we are simply stipulating that a certain class of cartesian arrows exist). The soundness of the rule

$$\frac{\Gamma, x{:}A \vdash B \ \textbf{type} \quad \Gamma, x{:}A \vdash b{:}B \quad \Gamma \vdash a{:}A}{\Gamma \vdash b[a/x]{:}B[a/x]}$$

requires that

$$\frac{\Gamma, x{:}A \vdash B \ \textbf{type} \quad \Gamma \vdash a{:}A}{\Gamma \vdash B[a/x] \ \textbf{type}}$$

which is obtained as $s^\star[\![B]\!]$, where s is the section determined by $\Gamma \vdash a{:}A$. The interpretation of $\Gamma \vdash b[a/x]{:}B[a/x]$ is then given by the unique section t defined by

$$\begin{tikzcd}
\Gamma \arrow[r, dotted, "\exists! t"] \arrow[dr, "id"'] & \Gamma \bullet s^\star B \arrow[r, "s \bullet B"] \arrow[d] \arrow[dr, phantom, "\lrcorner"] & \Gamma \bullet A \bullet B \arrow[d, "d_B"] \\
 & \Gamma \arrow[r, "s"'] \arrow[dr, "id"'] & \Gamma \bullet A \arrow[d, "d_A"] \\
 & & \Gamma
\end{tikzcd}$$

where $t; s \bullet B = s; u : \Gamma \longrightarrow \Gamma \bullet A \bullet B$, and

$$u = [\![\Gamma, x{:}A \vdash b : B]\!] : \Gamma \bullet A \longrightarrow \Gamma \bullet A \bullet B.$$

This follows from the fact u is a section of d_B, so $s; u; d_B = s$, and hence we obtain t as the mediating morphism to the above pullback square.

To give a model of $\mathcal{CC}^+$, following [45], is to give two subfibrations $\mathcal{F}_{type}$, $\mathcal{F}_{prop}$ of $cod_{\mathcal{B}}$, corresponding to the types and propositions, together with enough structure to reflect the relation between terms of type $Prop$ and those types of the form $Prf(A)$ in the theory: we should be able to form dependent product (Π) types, over both propositions and types, that propositions may be objectified as terms of a type $Prop$ which is moreover impredicatively closed under Π.

5.2.5 Dependent products in a category with fibrations

To give structure for dependent Π in a fibration is to give a right adjoint to substitution. In the context of a category with fibrations, this means for each X, A in $Fib(X)$, B in $Fib(X \bullet A)$, there exists an object ΠAB in $Fib(X)$ (hence the soundness of the rule (ΠF)), and a bijection

$$\lambda : \mathcal{B}/X \bullet A(f \bullet A, d_B) \cong \mathcal{B}/X(f, d_{\Pi AB}) : \rho$$

natural in each f, d_B. This is to say, given $f : Y \longrightarrow X$ in $\mathcal{B}$, $A \in Fib(X)$, $B \in Fib(X \bullet A)$, the dotted arrows should be in one-to-one correspondence:

$$\begin{tikzcd}
X \bullet A \bullet B \arrow[dr, "d_B"'] & Y \bullet f^\star A \arrow[l, dotted, "\alpha = \rho(\beta)"'] \arrow[r, "d_{f^\star A}"] \arrow[d, "f \bullet A"] \arrow[dr, phantom, "\lrcorner", very near start] & Y \arrow[r, dotted, "\beta = \lambda(\alpha)"] \arrow[d, "f"] & X \bullet \Pi AB \arrow[dl, "d_{\Pi AB}"] \\
 & X \bullet A \arrow[r, "d_A"'] & X &
\end{tikzcd}$$

In addition to naturality (which we have yet to define), we require that the construction of Π respects substitution — the so-called Beck-Chevalley condition [49,101,45,46]. We relegate discussion of these technical issues to Appendix A below.

5.2.6 Propositions and types in $\mathcal{CC}^+$

In $\mathcal{CC}^+$, there are two levels, corresponding to types and propositions, and each may depend on the other as well as themselves. So we have two fibrations $\mathcal{F}_{type}$, $\mathcal{F}_{prop}$, corresponding to two judgments $\Gamma \vdash A{:}Prop$ and $\Gamma \vdash A\ \mathbf{type}$, and moreover each fibration is closed under Π. To relate the two, and allow the mixed quantifications, including, as a special case, the impredicative

$$\frac{\Gamma, p{:}Prop \vdash \phi(p){:}Prop}{\Gamma \vdash \forall p{:}Prop.\phi(p){:}Prop}$$

we stipulate that $\mathcal{F}_{prop}$ is fully faithfully included in $\mathcal{F}_{type}$, and that the products in $\mathcal{F}_{type}$ restrict to products in $\mathcal{F}_{prop}$. Moreover, we demand that terms (arrows; sections) of type $Prop$ in the fibration $\mathcal{F}_{type}$ be reflected back to types in (objects in the fibres of) the fibration $\mathcal{F}_{prop}$.

Accordingly, there should be an object Ω in the $Fib(1)$ corresponding to the type $Prop$. A term of this type (section of d_Ω) gives rise to a type in $\mathcal{F}_{prop}$ by pulling back the "generic proposition" along the given section. The generic proposition T is the interpretation of $x : Prop \vdash Prf(x)\ \mathbf{type}$, an object in $Fib(1{\bullet}\Omega)$, whose display map d_T should be an object in $\mathcal{F}_{prop}$. Moreover every proposition (object in $\mathcal{F}_{prop}$; display map corresponding to such) should arise in this way.

5.3 The model

We now turn to the details of our topos-theoretic model. As above, we present a base category $\mathcal{B}$, together with data for two categories with fibrations over $\mathcal{B}$. We use the category **Sets**, of ordinary sets and functions, to illustrate the construction. It should be clear, however, that we make no use of the category-theoretic structure of **Sets** other than its finite limit structure, the cartesian closed structure and the fact that the notion of "subset", on which the construction clearly rests, is representable. In other words this construction may be applied to any *topos* $\mathcal{E}$ [49,51,53] to yield a model of $\mathcal{CC}^+$.

5.4 Definition of $\mathcal{B}$

$\mathcal{B}$, which corresponds to the category of first-order abstract deliverables for the hyperdoctrine defined on **Sets**, whose fibre over an object X is given by the poset of subobjects of X, is defined as follows:

- The objects of $\mathcal{B}$ are pairs $X = (X_0, X_1)$, where X_0 is a set and X_1 a subset of X_0. In the sequel, we shall refer to X_0 as the *carrier* of (X_0, X_1), and X_1 as its *distinguished subset*, or even its *predicate*.

- The arrows of $\mathcal{B}$ with domain (Y_0, Y_1) and codomain (X_0, X_1) are functions $f : Y_0 \longrightarrow X_0$ such that $fY_1 \subseteq X_1$.

The identity on (X_0, X_1) is just given by the identity on X_0, and composition of arrows is just composition of functions: if $f : Y_0 \longrightarrow X_0$ and $g : Z_0 \longrightarrow Y_0$ satisfy $fY_1 \subseteq X_1$. and $gZ_1 \subseteq Y_1$, then $(f \circ g)Z_1 = f(gZ_1) \subseteq fY_1 \subseteq X_1$. Moreover, each $f : Y_0 \longrightarrow X_0$ has a unique restriction to Y_1, which we will denote throughout as f_1.

5.4.1 Properties of $\mathcal{B}$

Lemma 5.4.1 *$\mathcal{B}$ has finite limits*

Proof It is straightforward to show that the terminal object 1 of $\mathcal{B}$ is given by ($\{\star\}$,$\{\star\}$). $\mathcal{B}$ also has all pullbacks, and they are inherited from **Sets**: given

$W = (W_0, W_1)$, $X = (X_0, X_1)$, $Y = (Y_0, Y_1)$, $Z = (Z_0, Z_1)$,

$$\begin{tikzcd} W \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, phantom, "\lrcorner", very near start] & Z \arrow[d, "g"] \\ Y \arrow[r, "f"'] & X \end{tikzcd}$$

is a pullback in $\mathcal{B}$ iff

$$\begin{tikzcd} W_0 \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, phantom, "\lrcorner", very near start] & Z_0 \arrow[d, "g"] \\ Y_0 \arrow[r, "f"'] & X_0 \end{tikzcd} \qquad \begin{tikzcd} W_1 \arrow[r, "q_1"] \arrow[d, "p_1"'] \arrow[dr, phantom, "\lrcorner", very near start] & Z_1 \arrow[d, "g_1"] \\ Y_1 \arrow[r, "f_1"'] & X_1 \end{tikzcd}$$

are pullbacks in **Sets** iff

$$\begin{tikzcd} W_0 \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, phantom, "\lrcorner", very near start] & Z_0 \arrow[d, "g"] \\ Y_0 \arrow[r, "f"'] & X_0 \end{tikzcd}$$

is a pullback, and $W_1 \cong p^{-1}Y_1 \cap q^{-1}Z_1$, as may be readily checked. ■

Remark We may take

$$X \times Y = (X_0 \times Y_0, (X_0 \times Y_1) \cap (X_1 \times Y_0))$$

with projections

$$\pi_X = \pi_{X_0} : X_0 \times Y_0 \longrightarrow X_0 \quad \pi_Y = \pi_{Y_0} : X_0 \times Y_0 \longrightarrow Y_0$$

as a distinguished choice of product diagram in $\mathcal{B}$.

Lemma 5.4.2 *$\mathcal{B}$ is cartesian closed*

Proof Given $X = (X_0, X_1)$ and $Y = (Y_0, Y_1)$ the object

$$X^Y = (X_0^{Y_0}, \{\phi{:}Y_0 \longrightarrow X_0 \mid \forall y{:}Y_0.y \in Y_1 \Rightarrow \phi y \in X_1\}),$$

together with the evaluation map $ev_Y = ev_{Y_0} : Y_0 \times X_0^{Y_0} \longrightarrow X_0$ inherited from **Sets**, define an exponential in $\mathcal{B}$. Firstly, we observe that ev_{Y_0} does indeed yield an arrow in $\mathcal{B}$:

$$y \in Y_1, \phi \in \{\phi{:}Y_0 \longrightarrow X_0 \mid \forall y{:}Y_0.y \in Y_1 \Rightarrow \phi y \in X_1\} \Rightarrow ev_{Y_0}(y, \phi) \equiv \phi y \in X_1.$$

Secondly, given $f : Z \times Y \longrightarrow X$ in $\mathcal{B}$, the expression

$$\lambda(f)(z) = \lambda y{:}Y_0.f(z, y) : Z \longrightarrow X^Y$$

obviously yields the unique mediating arrow $id_Z \times \lambda(f)$ to ev_Y. Chasing the predicates through the adjunction defining X^Y in **Sets**, we see that it is indeed an arrow of $\mathcal{B}$: by definition of f,

$$z \in Z_1 \Rightarrow \forall y{:}Y_0.y \in Y_1 \Rightarrow f(z, y) \in X_1$$

hence

$$z \in Z_1 \Rightarrow \lambda(f)(z) \in (X^Y)_1$$

■

5.4.2 Definition of the category with fibration $\mathcal{F}_{type}$ over $\mathcal{B}$

Proceeding along the lines indicated in the initial section, we will define, for each Γ an object of B, a collection $Fib_T(\Gamma)$, together with an operation $\bullet_T$. Given $\Gamma = (X_0, X_1)$, $Fib_T(\Gamma)$ consists of the pairs (A_0, A_1) where A_0 is a set and $A_1 \subseteq X_0 \times A_0$. So these are precisely an instance of an abstract notion of relativised specification, *cf.* Definition 3.2.2.

We define context extension by

$$\Gamma \bullet_T A = (X_0 \times A_0, (X_1 \times A_0) \cap A_1)$$

together with the display map $d_A : \Gamma \bullet_T A \longrightarrow \Gamma$ given by π_{X_0} on the carriers (which is readily seen to be an arrow of B).

Proposition 5.4.1 (Burstall) *Every morphism* $X \bullet_T A \xrightarrow{f} X \bullet_T B$ *over* X *(i.e.* $f; d_B = d_A$*) is of the form* $< id_X,\ g >$*, where* g *is an abstract second-order deliverable from* A *to* B *over* X.

Proof The condition $f; d_B = d_A$ readily implies that $f =< id_X,\ g >$ for some g, since the underlying morphisms of d_A, d_B are just the projections onto X. By the definition of morphism in $\mathcal{B}$, this g then satisfies

$$\forall x{:}X_0.\ \forall a{:}A_0.\ (x \in X_1 \wedge (x, a) \in A_1) \Longrightarrow (x,\ g(x, a)) \in B_1$$

But this is exactly as required. ■

5.4.3 Substitution

Definition of $f^\star$ Given $f : Y \longrightarrow X$ in B, and A in $Fib_T(X)$, let

$$f^\star A = (A_0, \{(y, a) \in Y_0 \times A_0 \mid (fy, a) \in A_1\}) = (A_0, (f \times id_{A_0})^{-1} A_1)$$

Definition of $f \bullet_T$ The map $f \times id_{A_0} : Y_0 \times A_0 \longrightarrow X_0 \times A_0$ on carriers does indeed define an arrow

$$f \bullet_T A{:}Y \bullet_T f^\star A \longrightarrow X \bullet_T A$$

in B, since

$$\begin{aligned}
(f \times id_{A_1})(Y \bullet_T f^\star A)_1 &= (f \times id_{A_1})((Y_1 \times A_0) \cap (f \times id_{A_1})^{-1} A_1) \\
&\subseteq (fY_1 \times A_0) \cap A_1 \\
&\subseteq (X_1 \times A_0) \cap A_1 \\
&= (X \bullet_T A)_1
\end{aligned}$$

Moreover,

$$\begin{tikzcd}
Y \bullet_T f^\star A \arrow[r, "f \bullet_T A"] \arrow[d, "d_{f^\star A}"'] \arrow[dr, phantom, "\lrcorner", very near start] & X \bullet_T A \arrow[d, "d_A"] \\
Y \arrow[r, "f"'] & X
\end{tikzcd}$$

is a pullback in $\mathcal{B}$, since

$$\begin{tikzcd}
Y_0 \times A_0 \arrow[r, "f \times id_{A_0}"] \arrow[d, "\pi_{Y_0}"'] & X_0 \times A_0 \arrow[d, "\pi_{X_0}"] \\
Y_0 \arrow[r, "f"'] & X_0
\end{tikzcd}$$

is a pullback in **Sets**, and

$$(Y \bullet_T f^\star A)_1 \equiv \pi_{X_0}^{-1} X_1 \cap (f \times id_{A_0})^{-1} A_1$$

This does not, however, complete the description of $\mathcal{F}_{type}$, since we must close under isomorphism the class of display maps which $Fib_T()$ gives rise to. This is essentially a technical consideration. Formally, the objects of $Fib_T((X_0, X_1))$ should be triples $(A_0, A, A_1 \hookrightarrow A)$, where

$$\begin{tikzcd}
& A \arrow[dl] \arrow[dr] & \\
A_0 & & X_0
\end{tikzcd}$$

is a given product diagram in $\mathcal{B}$. The definition then requires a choice of pullbacks in $\mathcal{B}$, so this is not completely satisfactory.

5.4.4 Dependent products

Define, given a context $\Gamma = (X_0, X_1)$, with $A \in \mathit{Fib}(\Gamma)$ and $B \in \mathit{Fib}(\Gamma \bullet A)$

$\Pi AB = (B_0^{A_0}, \{(x, \phi) \in X_0 \times B_0^{A_0} \mid \forall a{:}A_0 . x \in X_1 \Rightarrow (x, a) \in A_1 \Rightarrow (x, a, \phi a) \in B_1\})$

The natural isomorphism

$$\mathcal{B}/X \bullet A(f \bullet A, d_B) \cong \mathcal{B}/X(f, d_{\Pi AB})$$

which ensures we have a dependent product is given, as might reasonably be expected, by the usual exponential adjunction on the carriers, and chasing the predicates through the bijection: given $\phi : Y \bullet f^\star A \longrightarrow X \bullet A \bullet B$ such that $\phi; d_B = f \bullet A$, we must have $\phi = < f \times id_{A_0}, g >: Y_0 \times A_0 \longrightarrow (X_0 \times A_0) \times B_0$, with $g : Y_0 \times A_0 \longrightarrow B_0$; then $\psi =< f, \lambda(g) >: Y_0 \longrightarrow X_0 \times {B_0}^{A_0}$, moreover such that given $y \in Y_1, a \in A_0, (y, a) \in A_1$, we have $\psi y = (fy, \lambda a{:}A.g(y, a))$ and hence $\forall a{:}A.fy \in X_1 \Rightarrow (fy, a) \in A_1 \Rightarrow (fy, a, g(y, a)) \in B_1$.

Verification of the Beck-Chevalley condition consists of a long and tedious chase, but may be intuitively justified by our knowledge that Beck-Chevalley holds in **Sets** for each of the notions $\longrightarrow, \forall, \Rightarrow$ of exponential.

The collection of propositions is a type

The object $i\Omega = (\Omega, \Omega)$, where $\Omega = \{\mathit{true}, \mathit{false}\}$, considered as an element of $\mathit{Fib}_T(1)$, defines a type of propositions in the empty context, as we shall see below.

5.4.5 Definition of the category with fibrations $\mathcal{F}_{prop}$ over $\mathcal{B}$

As before, we obtain $\mathcal{F}_{prop}$ as a subfibration of $\mathcal{B}^2 \xrightarrow{cod} \mathcal{B}$ via collections $\mathit{Fib}_P(\Gamma)$, together with an operation $\Gamma \bullet_P (-)$. Given $\Gamma = (X_0, X_1)$, we take $\mathit{Fib}_P(\Gamma) = \mathcal{P}(X_0)$, the collection of subsets of X_0. For $A \in \mathit{Fib}_P(\Gamma)$, we define $\Gamma \bullet_T A = (X_0, X_1 \cap A)$, with the display map just given by the identity.

5.4.6 $\mathcal{F}_{prop}$ is included in $\mathcal{F}_{type}$

The inclusion Prf is induced by the mapping $A \mapsto A' = (1, A \times 1)$, defined on the collections $Fib_P(\Gamma)$, with its obvious extension to

$$\Gamma \bullet_P A \mapsto (X_0 \times 1, (X_1 \times 1) \cap (A \times 1)) \cong \Gamma \bullet_T A'$$

This then defines a mapping on the objects of $\mathcal{F}_{prop}$ (which are simply the displays $d_{(-)}$) to the objects of $\mathcal{F}_{type}$, which it is straightforward to show extends to a functor from $\mathcal{F}_{prop}$ to $\mathcal{F}_{type}$. Indeed, under this inclusion, $\mathcal{F}_{prop}$ is isomorphic to the category with fibrations defined by the subcollections of the $Fib_P(\Gamma)$ whose first component is 1. Moreover, this property is preserved by all the constructions we have considered, for obvious reasons. We are thus able to define products and substitution in $\mathcal{F}_{prop}$ by restriction of the constructions in $\mathcal{F}_{type}$.

5.4.7 Propositions yield types

Given a context Γ, whose interpretation is X in $\mathcal{B}$, and a judgment $\Gamma \vdash \phi{:}Prop$, we are given a section $f : X \longrightarrow X \bullet Prop \equiv (X_0 \times \Omega, X_1 \times \Omega)$ of $d_{Prop} : X \bullet_T Prop \longrightarrow X$. Such an arrow is exactly $< id_{X_0}, \chi >$, where $\chi : X_0 \longrightarrow \Omega$. This arrow classifies some subset of X_0, namely $A = \{x{:}X_0 \mid \chi(x)\}$, and hence an element of $Fib_P(X)$. In this way, terms of type $Prop$ yield types. In particular, we have the following denotation of T, the generic proposition:

$$[\![T]\!] = [\![p{:}Prop \vdash Prf(p)\ \mathbf{type}\ ]\!] = (1, \{(\phi, u){:}\Omega \times 1 \mid \phi = true\}) = (1, \{(\phi, u){:}\Omega \times 1 \mid \phi\})$$

Moreover, every proposition is a pullback of d_T along some map — indeed the classifying map in **Sets** of the subset representing the proposition.

5.4.8 Dependent products

It is straightforward, and somewhat tedious, to check that the calculation of products over propositions in $\mathcal{F}_{prop}$ agrees with that in $\mathcal{F}_{type}$. That is, we show that

$$\frac{\Gamma \vdash A : Prop \quad \Gamma, x{:}A \vdash B(x) : Prop}{[\![Prf(\forall x{:}A\ B)]\!] \simeq [\![\Pi p{:}Prf(A)\ Prf(B)]\!] \text{ in } Fib_T([\![\Gamma]\!])}$$

But this is trivial: both sides reduce to

$$\{* : 1 | A \Longrightarrow B\}$$

5.5 Consistency of the model

To show the model is consistent, we must exhibit a type with no inhabiting closed term. Accordingly, following our logical and type-theoretic intuition, we look at the denotation of the absurd proposition $\perp \equiv \forall x{:}Prop.x{:}Prop$, or rather $Prf(\perp) \cong \Pi x{:}Prop.Prf(x)$ **type** . From our earlier calculations,

$$[\![\circ \vdash \Pi x{:}Prop.\ Prf(x)\ \textbf{type}\ ]\!] \cong (1, \{u{:}1 | \forall\phi{:}\Omega.\phi\}) \cong (1, \emptyset)$$

since $\forall\phi{:}\Omega.\ \phi$ is the false proposition in $\mathbf{Sets}$, classifying the empty subset of a given set. Clearly the above object can have no sections of its projection to 1, since any such map (in $\mathbf{Sets}$) would have to factor through the empty set, which is impossible. Hence $Prf(\perp)$ has no inhabiting terms, and the model is a logically consistent one.

5.6 Proof-irrelevance

The model presented above is proof-irrelevant, in the sense that, given any two proofs of a given proposition, we may judge them equal (where we understand the interpretation of equality judgments as being given by equality of objects/arrows in the model). In more detail, suppose Γ is a well-formed context, with $[\![\Gamma]\!] = (X_0, X_1)$, and $\Gamma \vdash \phi{:}Prop$. Then $[\![Prf(\phi)]\!]$ is a subset P of X_0, and if $\Gamma \vdash p, q{:}Prf(\phi)$, then $[\![p]\!]$, $[\![q]\!]$ are sections of $id_{X_0} : (X_0, X_1 \cap P) \longrightarrow (X_0, X_1)$. But any such map must be the identity on carriers, to the extent to which it exists at all. Hence $[\![p]\!] = [\![q]\!]$.

We can express this as an axiom in $\mathcal{CC}^+$ in the obvious way

$$ProofIrrelevance{:}\forall\phi{:}Prop.\ \forall p, q{:}\phi.\ EQ_\phi\ p\ q$$

using Leibniz' equality, which is definable in $\mathcal{CC}^+$ in exactly the same way as in ECC.

In the interests of completeness, we should perhaps also add an axiom to reflect that, in the logic of sets, all true propositions are essentially identified as the one-point set, considered as a subobject of 1:

$$Collapse{:}\forall\phi{:}Prop.\ \phi \Longrightarrow EQ_{Prop}\ \phi\ true$$

5.6.1 Extensionality

The higher-order logic of **Sets** is extensional, in the sense that for any sets S, T, and functions $f, g{:}S \longrightarrow T$,

$$(\forall x{:}S.\ fx = gx) \Longrightarrow f = g$$

Pursuing a complete axiomatisation of the model, we would like to formulate this (ξ)-rule as an axiom

$$\mathit{Extensionality}{:}\forall S, T{:}\mathit{Type}.\ \forall f, g{:}S \longrightarrow T.\ (\forall x{:}S.\ fx = gx) \Longrightarrow f = g$$

But we do not have a symbol Type in this language which represents the judgment $\Gamma \vdash A\ \mathbf{type}$. Moreover, even if we consider $\mathcal{CC}^+$ as a subsystem of ECC, we are still unable to quantify over the types, without the assumption of universes. So we must be content to regard $\mathit{Extensionality}$ as a schematic axiom in the absence of enough structure in the topos to define the hierarchy of type universes.

5.7 A model of Luo's ECC

If we are prepared to make certain assumptions about the underlying theory of sets, then it is possible to extend this model to give a model of the whole of Luo's ECC. In fact, we already have a semantics for the Σ-types, under no further assumptions.

5.7.1 Sums

Following Hyland and Pitts[45], we specify sums by giving a *left* adjoint to substitution (this defines a weak sum, akin to the existential quantifier), together with an orthogonality condition, which gives the second projection from the sum. In terms of categories with fibrations, given a context $\Gamma = (X_0, X_1)$, with $A \in \mathit{Fib}(\Gamma)$ and $B \in \mathit{Fib}(\Gamma \bullet A)$, we define the object in $\mathit{Fib}(\Gamma)$, ΣAB whose underlying type is $A_0 \times B_0$ and whose predicate is

$$\{(x, a, b) \in X_0 \times A_0 \times B_0 \mid x \in X_1 \wedge (x, a) \in A_1 \wedge (x, a, b) \in B_1\}$$

Details that this indeed defines a Σ-type are left to the reader.

5.7.2 Type universes

Typical of set-theoretic assumptions when modelling universes in type theory [59] is that there exist strongly inaccessible cardinals $\kappa_1 < \kappa_2 < \ldots \kappa_n < \ldots$. The intention is that the nth inaccessible should code the nth universe. It seems intuitively straightforward that by restricting the fibration $\mathcal{F}_{type}$ to consist only of those (pairs of) sets in V_{κ_n}, we may interpret $Type_n$. The choice of strong inaccessibles ensures that all our constructions, Π,Σ, etc. do not take us outside κ_n. Thus we obtain a hierarchy of fibrations corresponding to the type levels, with fully faithful inclusions between them. Finally, we obtain "$Type_j\colon Type_{j+1}$", from $V_{\kappa_j} \in V_{\kappa_{j+1}}$. The details are very similar to [*ibid.* pp. 135–7].