Chapter 3

First-order and second-order deliverables

In this chapter, we consider the basic syntactic definitions in the study of this approach to program correctness. As indicated in the introduction, we wish to give an account of correct programs (with respect to some specification) in a way which makes clear the distinction between computational and logical behaviour. This leads to Definition 3.1.2 of deliverable below. Surely the most desirable feature of a design methodology is that it should be *compositional*, i.e. we should be able to describe the development of large programs in terms of suitably smaller sub-programs, and better still if this can be done in a syntax-directed way. Since our putative programming language is a simple type theory with primitive recursion over inductive types, our deliverable constructors should reflect this structure. It is at this point that we draw on the experience of using category theory in the semantics of formal systems, in particular λ-calculus and higher-order logic [53, for example]. Rather than formalise an interpretation of a typed λ-calculus inside ECC (considered as a logical framework *cf.* [34]), we develop combinators for the appropriate categorical constructions. This will involve some technicalities, notably the idea of a *semi-cartesian closed category*, due to Hayashi [37]. With these, we are able, more or less, to realise the aim of a compositional system for correct program development.

3.1 Definition and properties of first-order deliverables

We work relative to some well-formed context Γ in ECC.

3.1.1 Specifications

Informally, we consider a first approximation[1] to the idea of specification, given by a pair, consisting of

- a type, together with
- a predicate defined over that type.

The motivation is that the types carry computational information, while the predicates carry (computationally irrelevant) propositional information about the specification. Such an idea is not new: it lies at the heart of the so-called "theory of subsets" in Martin-Löf type theory [93,79][2], and the division of type information into "informative" and "non-informative" propositions underlies most of the studies of program extraction in type theory [81,82,38,39,79].

---

[1]There is, of course, a zero'th approximation, taking *the types themselves* to be specifications. Indeed, the use of Martin-Löf type theory as a programming language [70] is based on this reading of types.

[2]The reader should throughout have in mind the account of the subset theory in Martin-Löf type theory [79, Chapter 18], where a system of multiple judgments, as opposed to the Σ-types of ECC, conveys a similar idea.

Definition 3.1.1 A *specification* is a pair of terms s, S such that $\Gamma \vdash s : \mathit{Type}$, and $\Gamma, x{:}s \vdash Sx : \mathit{Prop}$.

We typically write $\mathcal{S}, \mathcal{T}, \mathcal{U}$ for specifications, and understand s, t, u (respectively S, T, U) as referring to their underlying type (resp. predicate). Formally, there is a type of specifications, namely $SPEC_1 =_{\text{def}} \Sigma s{:}\mathit{Type}.s \rightarrow \mathit{Prop}$, so that we may consider operations which construct specifications entirely within the framework of ECC (*cf.* the account of specifications and refinements in [61]). We will consider a category whose objects are the specifications, and whose morphisms are defined below. Specifications defined by *logically* equivalent predicates in general define distinct objects.

3.1.2 First-order deliverables

Having made a choice of objects, we must make an appropriate choice of morphism, in order to define a category. Following the intuition behind our definition of specification, the morphisms should reflect our concern with separating computational from propositional information. As indicated in the introduction, the appropriate notion of morphism consists of a pair (f, F), where f is a function between the underlying types, and F[3] is a proof that f respects the predicates. Formally, we make the following:

Definition 3.1.2 (Burstall [12,59]) *first-order deliverable*
Given specifications $\mathcal{S} = (s, S), \mathcal{T} = (t, T)$, a *first-order deliverable* is a term $\mathcal{F}$ such that

$$\Gamma \vdash \mathcal{F} : \Sigma f : s \longrightarrow t. \forall x : s. Sx \Longrightarrow T(fx).$$

[3]The reader hopefully will forgive this somewhat unfortunate notation, which is chosen to be consistent with that of specifications.

The motivation for such a definition goes right back to Hoare's original paper on axiomatic semantics [40], and, like the logic of triples which bears his name, expresses in a *formal* system the informal notion of a program, together with a certificate of some specified input/output behaviour. Of course, we are concerned with a functional language, rather than an imperative one, so there is no confusion over program variables and logical variables. In this framework, moreover, the proof and the program are linked as a pair. This definition uses the Σ-types in an essential way to capture this idea. We may use all the features of ECC to construct such pairs, but based on our intuitions about computational *vs.* propositional information, we insist upon a trivial extraction process: first projection π_1 from the Σ-type yields the underlying algorithm f. Indeed this accounts for the name "deliverables": they are what a software house should deliver to its customers, a program plus a proof in a box with the specification printed on the cover (the Σ-type). The customer can independently check the proof and then run the program, without the need for a complicated extraction process which may yield an unusual algorithm. Indeed, we even propose this method as a style for developing programs in the first place. We usually have a reasonable idea of the algorithm in advance, as opposed to its proof of termination, or even correctness, and we would like an understanding, which reflects our intuitions, of how to build up these deliverables from smaller ones, for example by refinement and composition, possibly with machine assistance.

Example As hinted in Chapter 2, the construction of Figure 2–1 is a simple example of a first-order deliverable, in this case from $(nat, Even)$ to $(nat, Even)$. The reader unfamiliar with machine-checked proofs may balk at the complicated formal machinery required to establish such a simple result. She may take comfort in the fact that all the complexity is localised in the propositional reasoning. The algorithm remains recognisable in the resulting pair. In more complex examples, we must *normalise* the first projection of a deliverable to recover the algorithm. Nonetheless, the categorical combinators which we develop below give us a schematic description of the algorithm within a deliverable.

In terms of LEGO, we make the following constructions to define the type of first-order deliverables within ECC:

```
Lego> Del1;
value = [s,t|Type][S:Pred s][T:Pred t][f:s->t]{x|s}(S x)->T (f x)
type  = {s,t|Type}(Pred s)->(Pred t)->(s->t)->Prop
Lego> del1;
value = [s,t|Type][S:Pred s][T:Pred t]<f:s->t>Del1 S T f
type  = {s,t|Type}(Pred s)->(Pred t)->Type
```

We have exploited the implicit syntax to enable us to suppress the argument types[4] in the predicate `Del1` and the type `del1`.

With an eye to the categorical aspects of this definition, we typically write [5]

$$\mathcal{S} \xrightarrow{\mathcal{F}} \mathcal{T} \in \mathbf{del}_1 \text{ or } (s, S) \xrightarrow{(f, F)} (t, T) \in \mathbf{del}_1$$

when $\Gamma \vdash f : s \longrightarrow t$, and $\Gamma \vdash F : \mathrm{Del}_1\ S\ T\ f$. Since s, t may be inferred by the typechecker, and we are in general not interested in F, save to know that it exists, we may even abuse our notation and write

$$S \xrightarrow{f} T \in \mathbf{del}_1.$$

This is in accordance with LEGO's implicit syntax, coupled with a certain laxity about the proof terms F.

---

[4]This accounts for our choice of notation: since the types `s,t` may be inferred from `S,T`, we relegate them notationally to the lower case.

[5]The reader should be careful to distinguish the category arrow $\xrightarrow{\quad\quad}$ from the type constructor arrow $\longrightarrow$.

Of course, at this stage, we need not have used the Σ-types to present these definitions. However, internalising the mathematical pair in a Σ-type allows us to represent operations which produce such function-proof pairs within the calculus. This gives us the possibility of developing a structure on these gadgets.

3.1.3 Equality of deliverables

Mathematicians and computer scientists typically think, at least informally, that operations are extensional. However, in type theory, we must in general be more circumspect, if we are to preserve the proof-theoretic properties upon which our implementations depend. An example of this is Martin-Löf's switch to an extensional equality type in his 1984 theory [71], which leads to an undecidable type-checking problem. Previous, and subsequent versions of his theory have used an intensional equality type [69,70,79]. Others have argued elsewhere for extensional systems [2, for example], on grounds of utility, but we do not take this view. The problem of adding even η-conversion to systems such as ECC, is still an active area of research, though a limited case of η-conversion, on the well-typed terms, appears to be permissible [94,95,31]. Adding Σ-types to a type theory raises the problem of surjective pairing: for systems such as ECC it is known that the Church-Rosser property for reduction fails [59, pp. 40–41].

Rather than concern ourselves with these delicate questions, our definitions above encourage us to make the following identifications between the terms of interest to us. In fact, we shall scarcely have need of these technicalities, except at a number of points below, in verifying that $\mathbf{del}_1$ is a category, moreover a category with certain structure.

Definition 3.1.3 *Equality of specifications*
Given specifications $(s, S), (t, T)$, we say $(s, S) = (t, T)$ if

$$s \simeq_{\beta\delta} t \text{ and } \lambda x{:}s.\ S\,x \simeq_{\beta\delta} \lambda x{:}t.\ T\,x.$$

This definition will have hardly any impact on the development of the subsequent material, but is included partly to underline our concern with extensionality in intensional systems such as ECC, and also to support the following definition of equality on morphisms, without which we cannot even define a category. Moreover, the definition of first-order deliverable only mentions the predicate part S of a specification $\mathcal{S}$ in the application $S\,x$, so our proposed structure on $\mathbf{del}_1$ cannot detect differences between η-expansions of S. Since this definition is based on the underlying conversion relation $\simeq_{\beta\delta}$, which is decidable, we are able to use the typechecker to test for equality.

The other natural definition of equality, based on logical equivalence, is not in general decidable. In the spirit of constructive mathematics, a non-trivial logical equivalence between specifications must be regarded as having some algorithmic content, which should be made explicit. As an example, we might consider two definitions of what it means to be a permutation of a word over some alphabet Λ:

Enumerative [107] Given two words $u = a_1 \ldots\ a_m, v = b_1 \ldots b_n$, we say $u \sim\ v$, if and only if $m = n$ and

$$\exists f{:}\{1,\ldots,n\} \xrightarrow{\sim} \{1,\ldots,n\}.\ \forall i \in \{1,\ldots,n\}.\ a_i = b_{f(i)}$$

Impredicative, higher-order definition [66] Permutation $\sim$ is the least congruence on Λ^* such that for all words $u, v \in \Lambda^*, u * v\ \sim\ v * u$. We can express this in ECC as follows:

$$\sim\,=_{\text{def}} \lambda u, v{:}\Lambda^*\!.\,\forall R{:}\Lambda^* \to \Lambda^* \to \mathit{Prop}.\,(\mathit{cong} R) \Rightarrow (\forall u, v{:}\Lambda^*\!.\ R(u{*}v)(v{*}u)) \Rightarrow Ruv$$

where $\mathit{cong} R$ expresses that the relation R is an equivalence relation and a congruence for the operations in Λ^*.

A proof of the equivalence of these two definitions involves the *construction*, on the one hand, of specific permutations of initial segments $\{1,\ldots,n\}$ of the natural numbers, and on the other, essentially, derivation trees of the proofs of propositions $u \sim v$. Appendix B contains a detailed treatment of the impredicative definition in LEGO. We developed this in the course of investigating proofs of sorting algorithms.

Definition 3.1.4 *Equality of deliverables*

Given

$$(s, S) \underset{(g,G)}{\overset{(f,F)}{\rightrightarrows}} (t, T) \in \mathbf{del}_1,$$

we say $(f, F) = (g, G)$ if

$$\lambda x{:}s.\ f x \simeq_{\beta\delta} \lambda x{:}s.\ g x$$

and

$$\lambda x{:}s.\ \lambda h{:}S\,x.\ F x h \simeq_{\beta\delta} \lambda x{:}s.\ \lambda h{:}S\,x.\ G x h.$$

In terms of LEGO we can define a polymorphic function — which we might call "extensionalisation" — from first-order deliverables to first-order deliverables, mapping (f, F) to $(\lambda x{:}s.\ f x, \lambda h{:}S\,x.\ F x h)$:

```
Lego> ext_del1;
value = [s,t|Type][S|Pred s][T|Pred t][FF:del1 S T][f=FF.1][F=FF.2]
        ([x:s]f x,[x|s][h:S x]F h)
type  = {s,t|Type}{S|Pred s}{T|Pred t}(del1 S T)->del1 S T
```

The definition of equality of deliverables $\mathcal{F}, \mathcal{G}$ then amounts to convertibility of the extensionalisations of $\mathcal{F}, \mathcal{G}$. This seems to be the minimal extension of the basic conversion relation which ensures good categorical properties. We are now in a position to exploit Lemma 2.2.1, to enable us to determine equality of deliverables within LEGO.

3.1.4 Semi-structure in categories

The use of cartesian closed categories to give models of the simply-typed λ-calculus is by now very familiar in computer science [21,53, for example], as are various equational presentations of the structure of a cartesian closed category. The basic type and term constructors are defined by adjunctions. In this analysis, the unit and counit of the adjunction defining the arrow type correspond, loosely, to η and β conversion, respectively (and similarly for the product type constructor). An earlier account of the ideas in this thesis attempted to give an account of a cartesian closed structure on $\mathbf{del}_1$ [13], but we have abandoned the clumsy definitions required in favour of the (hopefully) smoother treatment below. As we discussed above, the absence of η-conversion and surjective pairing in ECC forces some extra technical difficulty upon us. However, models of various typed λ-calculi without η-conversion and surjective pairing can be given a rigorous semantic account in terms of *semi-adjunctions*, introduced by Hayashi in [37][6]. Essentially, the equations defining an adjunction are relaxed sufficiently that, under suitable conditions, there is a notion of counit which corresponds appropriately to β-conversion, without a unit corresponding to η-conversion. Since we follow Hayashi's treatment closely, we recall here the main definitions of that paper.

Definition 3.1.5 (Semi-functor) Given two categories $\mathbf{C}, \mathbf{D}$, a *semi-functor* between them is "a functor which need not preserve identities": that is to say, we are given an assignment F of objects of $\mathbf{D}$ to objects of $\mathbf{C}$, and an assignment, also called F, of arrows of $\mathbf{D}$ to arrows of $\mathbf{C}$, such that

$$A \xrightarrow{f} B \in \mathbf{C} \quad \text{implies}\; FA \xrightarrow{Ff} FB \quad \in \mathbf{D}$$

[6] I am grateful to Bart Jacobs for introducing me to this concept.

and

$$F(A \xrightarrow{f} B \xrightarrow{g} C) \;=\; FA \xrightarrow{Ff} FB \xrightarrow{Fg} FC \in \mathbf{D}.$$

In a *semi*-adjunction, we replace the usual natural bijection on homsets with a pair of maps, which need not be mutual inverses. However, we still require them to behave "naturally".

Definition 3.1.6 (Semi-adjunction) Given two categories $\mathbf{C}, \mathbf{D}$, and two semi-functors F, G between them, with

$$\begin{tikzcd} \mathbf{C} \arrow[r, shift left, "F"] & \mathbf{D} \arrow[l, shift left, "G"] \end{tikzcd},$$

a *semi-adjunction* is given by two families σ, τ of maps, indexed by the objects of $\mathbf{C}, \mathbf{D}$, such that for every

$$C' \xrightarrow{f} C \;\in \mathbf{C} \;\text{ and }\quad D \xrightarrow{g} D' \;\in \mathbf{D},$$

the following diagram

$$\begin{tikzcd} \mathbf{D}(FC, D) \arrow[r, shift left, "\sigma_{C,D}"] \arrow[d, "Ff;(\_);g"'] & \mathbf{C}(C, GD) \arrow[l, shift left, "\tau_{C,D}"] \arrow[d, "f;(\_);Gg"] \\ \mathbf{D}(FC', D') \arrow[r, shift left, "\sigma_{C',D'}"] & \mathbf{C}(C', GD') \arrow[l, shift left, "\tau_{C',D'}"] \end{tikzcd}$$

commutes. We say F (resp G) is left (resp. right) *semi-adjoint* to G (resp F).

The beauty of this definition lies in the following lemma.

Lemma 3.1.1 (Hayashi) *Suppose F, G above are in fact functors. Then σ, τ define families of inverse isomorphisms giving an adjunction between F and G.*

So this concept subsumes the whole of our ordinary understanding of adjunctions, and hence all the constructions of universal gadgets defined by adjunctions[7]. In particular, in a category $\mathbf{C}$:

a semi-terminal object is given by a right semi-adjoint to the functor $\mathbf{C} \xrightarrow{!} \mathbf{1}$;

semi-products are given by a right semi-adjoint $\times$ to the diagonal functor $\mathbf{C} \xrightarrow{\Delta} \mathbf{C} \times \mathbf{C}$;

semi-exponentials are given, for each $C \in \mathbf{C}$, by a right semi-adjoint to the semi-functor $C \times (\_)$;

If $\mathbf{C}$ has all the above structure, we say $\mathbf{C}$ is a *semi-cartesian closed* category, or *semi-ccc*. Hayashi's development leads to the following main results, generalising previous accounts of models of the λ-calculus with β-conversion only.

Proposition 3.1.1 *Semi-cccs are sound and complete for interpretations of the $\lambda\beta$-calculus.*

Proposition 3.1.2 *Semi-cccs can be presented algebraically. Given a category $\mathbf{C}$, and operations $1, !, \times, <, >, \pi_l, \pi_r, (\_)^{(-)}, \Lambda, ev$ satisfying the following equations (i)-(vi), then $\mathbf{C}$ may be given a structure of a semi-ccc.*

(i) *Given $X \xrightarrow{f} Y$ and $X \xrightarrow{g} Z$, then $X \xrightarrow{<f,g>} Y \times Z$. Moreover, these operations are semi-functorial, in the sense that, given also $W \xrightarrow{h} X$, then*

$$W \xrightarrow{h} X \xrightarrow{<f,g>} Y \times Z \;=\; W \xrightarrow{<hf,hg>} Y \times Z.$$

(ii) $X \times Y \xrightarrow{\pi_l} X$ *and* $X \times Y \xrightarrow{\pi_r} Y$*, with* $<f,g>\pi_l = f$ *and* $<f,g>\pi_r = g$

[7]However, by contrast with adjunctions, structure defined by a semi-adjunction is *not* in general unique.

(iii) *Given* $X \times Y \xrightarrow{k} Z$*, then* $X \xrightarrow{\Lambda(k)} Z^Y$. *Given* $W \xrightarrow{h} X$*, then*

$$W \xrightarrow{h} X \xrightarrow{\Lambda(k)} Z^Y = \Lambda(W \times Y \xrightarrow{<\pi_l h, \pi_r>} X \times Y \xrightarrow{k} Z).$$

(iv) $Z^Y \times Y \xrightarrow{ev} Z$. *Given* $X \times Y \xrightarrow{k} Z$, $W \xrightarrow{h} X$*, and* $W \xrightarrow{h'} Y$*, then*

$$W \xrightarrow{<\Lambda(k)h, h'>} Z^Y \times Y \xrightarrow{ev} Z = W \xrightarrow{<h, h'>} X \times Y \xrightarrow{k} Z.$$

(v) 1 *is an object of* $\mathbf{C}$*, and* $X \xrightarrow{!_X} 1$*, with*

$$W \xrightarrow{h} X \xrightarrow{!_X} 1 = W \xrightarrow{!_W} 1.$$

(vi)

$$Z^Y \times Y \xrightarrow{ev} Z = Z^Y \times Y \xrightarrow{<\pi_l, \pi_r>} Z^Y \times Y \xrightarrow{ev} Z.$$

Much of the rest of this section is dedicated to a proof of the following theorem:

Theorem 3.1.1 $\boldsymbol{del}_1$ *is semi-cartesian closed.*

Remark We should note that the structure of a semi-ccc is not instrumental in the use of deliverables as a means of developing proven programs. What the results below reflect is the understanding that we may construct at least the skeleton of a proof of some program phrase alongside the development of the phrase itself. The categorical structure reflects syntactic constructions on terms which enforce the correctness of programs.

3.1.5 Identities and composition

By considering our slight modification of the underlying conversion on the terms of ECC, to remedy the failure of surjective pairing and η-conversion in the calculus, we fixed a notion of equality on arrows. With this definition, it is now trivial to establish that specifications, together with first-order deliverables as morphisms, form a category, denoted by $\mathbf{del}_1$.

The identities are given simply by

$$(s, S) \xrightarrow{(\lambda x{:}s.\ x,\ \lambda x{:}s.\ \lambda h{:}Sx.\ h)} (s, S) \in \mathbf{del}_1;$$

in LEGO, we have

```
Lego> id_del1;
value = [s|Type][S:Pred s]([x:s]x,[x|s][p:S x]p)
type  = {s|Type}{S:Pred s}del1 S S .
```

Composition is given by

$$\mathcal{S} \xrightarrow{(f, F)} \mathcal{T} \xrightarrow{(g, G)} \mathcal{U} = \mathcal{S} \xrightarrow{(\lambda x{:}s.\ g(fx),\ \lambda x{:}s.\ \lambda h{:}Sx.\ G(fx)(Fxh))} \mathcal{U};$$

in LEGO, we have

```
Lego> compose_del1;
value = [s,t,u|Type][S|Pred s][T|Pred t][U|Pred u]
        [FF:del1 S T][GG:del1 T U]
        [f=FF.1][F=FF.2][g=GG.1][G=GG.2]
        ([x:s]g (f x), [x|s][h:S x]G (F h): <f:s->u>Del1 S U f)

type  = {s,t,u|Type}{S|Pred s}{T|Pred t}{U|Pred u}
        (del1 S T)->(del1 T U)->del1 S U .
```

We now obtain, by immediate appeal to the typechecker, that these definitions do indeed yield the structure of a category on $\mathbf{del}_1$[8].

Proof

[8]Indeed, we only need the modified equality to prove the identity laws. The conversion relation is sufficient to establish associativity of composition.

```
Lego> leftidentity_del1;
value = [s,t|Type][S|Pred s][T|Pred t][F|del1 S T]
        reflEQ (compose_del1 (id_del1 S) F)
type  = {s,t|Type}{S|Pred s}{T|Pred t}{F|del1 S T}
        EQ (compose_del1 (id_del1 S) F) (ext_del1 F)
Lego> rightidentity_del1;
value = [s,t|Type][S|Pred s][T|Pred t][F|del1 S T]
        reflEQ (compose_del1 F (id_del1 T))
type  = {s,t|Type}{S|Pred s}{T|Pred t}{F|del1 S T}
        EQ (compose_del1 F (id_del1 T)) (ext_del1 F)
Lego> associativity_del1;
value = [s,t,u,v|Type][S|Pred s][T|Pred t][U|Pred u][V|Pred v]
        [F|del1 S T][G|del1 T U][H|del1 U V]
        reflEQ (compose_del1 F (compose_del1 G H))
type  = {s,t,u,v|Type}{S|Pred s}{T|Pred t}{U|Pred u}{V|Pred v}
        {F|del1 S T}{G|del1 T U}{H|del1 U V}
        EQ (compose_del1 F (compose_del1 G H))
              (compose_del1 (compose_del1 F G) H).
```

■

Example One could imagine two very simple first-order deliverables, both with underlying algorithm $x \mapsto x + 1$, which respectively transform even numbers into odds and odds into evens. Their composition gives us another representation, with the same normal form, of the simple example of $x \mapsto x + 2$ above.

3.1.6 Semi-Terminal object

$$Unit =_{\text{def}} (unit, \lambda u : unit.u = ()^{9}) : SPEC_1$$

defines a trivial specification, where the unit type is defined in LEGO by

[9] Here, () is the unique term of type $unit$.

```
[unit:Type(0)];
[void:unit];
[unitrec:{u:unit}{C:unit->Type}(C void) -> (C u)];
[[C:unit->Type][d:C void]
    unitrec void C d ==> d];
```

in Pollack's notation for inductive types. We obtain, for any $S : SPEC_1$ the deliverable

$$!_S =_{\text{def}} (\lambda x{:}s.\ (), \lambda x{:}s.\ \lambda p{:}S\,x.\ \mathit{reflEQ}()) : \mathbf{del}_1\ S\ \mathit{Unit}.$$

This satisfies Hayashi's condition (v) above.

Proof

```
Lego> Unit;
value = [u:unit]EQ void u
type  = Pred unit

Lego> shriek_del1;
value = [s|Type][S1:Pred s]([_:s]void,[x|s][_:S1 x]reflEQ void
                            :<f:s->unit>Del1 S1 Unit f)
type  = {s|Type}{S:Pred s}del1 S Unit

Lego> hayashi5;
value = [v,w|Type][V|Pred v][W|Pred w]
         [_|del1 V W] reflEQ (shriek_del1 V)
type  = {v,w|Type}{V|Pred v}{W|Pred w}{KK|del1 V W}
         EQ (shriek_del1 V) (compose_del1 KK (shriek_del1 W)).
```

■

3.1.7 Binary semi-products

To obtain semi-products, we use the non-dependent Σ-type as the underlying type, with conjunction at the predicate level[10]:

$$\mathcal{S} \times \mathcal{T} =_{\text{def}} (s \times t, \lambda p{:}s \times t.\ (S(\pi_1 p)) \wedge (T(\pi_2 p)).$$

In LEGO, we have the following:

```
Lego> Product_del1;
value = [s,t|Type][S:Pred s][T:Pred t][xy:s#t]and (S xy.1) (T xy.2)
type  = {s,t|Type}(Pred s)->(Pred t)->(s#t)->Prop
Lego> pair_fun;
value = [s,t,u|Type][f:s->t][g:s->u][x:s](f x,g x)
type  = {s,t,u|Type}(s->t)->(s->u)->s->t#u
Lego> pair_del1;
value = [s,t,u|Type][S|Pred s][T|Pred t][U|Pred u]
        [FF:del1 S T][GG:del1 S U][f=FF.1][F=FF.2][g=GG.1][G=GG.2]
        (pair_fun f g,[x|s][p:S x]pair (F p) (G p))
type  = {s,t,u|Type}{S|Pred s}{T|Pred t}{U|Pred u}
        (del1 S T)->(del1 S U)->del1 S (Product_del1 T U).
Lego> pi1_del1;
value = [t,u|Type][T1:Pred t][U1:Pred u]
        ([yz:t#u]yz.1,[x|t#u][h:Product_del1 T1 U1 x]fst h
         :<f:(t#u)->t>Del1 (Product_del1 T1 U1) T1 f)
type  = {t,u|Type}{T:Pred t}{U:Pred u}del1 (Product_del1 T U) T
Lego> pi2_del1;
value = [t,u|Type][T1:Pred t][U1:Pred u]
        ([yz:t#u]yz.2,[x|t#u][h:Product_del1 T1 U1 x]snd h
         :<f:(t#u)->u>Del1 (Product_del1 T1 U1) U1 f)
type  = {t,u|Type}{T:Pred t}{U:Pred u}del1 (Product_del1 T U) U
```

That we indeed have a semi-product structure now follows:

---

[10]This is hardly surprising, given the Curry-Howard correspondence.

```
Lego> hayashi1;
value = [s,t,v,w|Type][S|Pred s][T|Pred t][V|Pred v][W|Pred w]
        [GG|del1 W S][HH|del1 W T][KK|del1 V W]
        reflEQ (compose_del1 KK (pair_del1 GG HH))
type  = {s,t,v,w|Type}{S|Pred s}{T|Pred t}{V|Pred v}{W|Pred w}
        {GG|del1 W S}{HH|del1 W T}{KK|del1 V W}
        EQ (compose_del1 KK (pair_del1 GG HH))
           (pair_del1 (compose_del1 KK GG) (compose_del1 KK HH))
Lego> hayashi2i;
value = [s,t,w|Type][S|Pred s][T|Pred t][W|Pred w]
        [GG|del1 W S][HH|del1 W T]
        reflEQ (compose_del1 (pair_del1 GG HH) (pi1_del1 S T))
type  = {s,t,w|Type}{S|Pred s}{T|Pred t}{W|Pred w}
        {GG|del1 W S}{HH|del1 W T}
    EQ (compose_del1 (pair_del1 GG HH) (pi1_del1 S T)) (ext_del1 GG)
Lego> hayashi2ii;
value = [s,t,w|Type][S|Pred s][T|Pred t][W|Pred w]
        [GG|del1 W S][HH|del1 W T]
        reflEQ (compose_del1 (pair_del1 GG HH) (pi2_del1 S T))
type  = {s,t,w|Type}{S|Pred s}{T|Pred t}{W|Pred w}
        {GG|del1 W S}{HH|del1 W T}
    EQ (compose_del1 (pair_del1 GG HH) (pi2_del1 S T)) (ext_del1 HH)
```

3.1.8 Binary semi-coproducts

In fact we have the additional structure of a semi-coproduct, the obvious algebraic definition of which we do not give here, thanks to the datatypes mechanism. For completeness we record the LEGO definitions. We omit the proofs, however, since they are of only marginally greater difficulty than those for the semi-product above, save for the use of the recursion combinator in the proof that the case construct respects the disjunction of predicates. Logically, the construction is given by

$$\mathcal{S} + \mathcal{T} =_{\text{def}} (s + t, \lambda c{:}s + t.\ (\exists x{:}s.\ (c = \textbf{inl}\ x) \wedge (Sx)) \vee (\exists y{:}t.\ (c = \textbf{inr}\ y) \wedge (Ty))$$

where $\exists, \wedge, \vee$ are the defined constructs in the embedded higher-order logic, and we assume the existence of a sum $+$ of types, together with constructors **inl**, **inr** etc[11].

```
[sum_type:([tau:Type]tau->tau->tau)Type];
[inl_type:{A,B|Type}A->(sum_type A B)];
[inr_type:{A,B|Type}B->(sum_type A B)];
[when:{A,B|Type}{C:(sum_type A B)->Type}
  ({a:A}C (inl_type a))->({b:B}C (inr_type b))->{c:sum_type A B}C c];

[[A,B|Type][C:(sum_type A B)->Type]
 [d:{a:A}C (inl_type a)][e:{b:B}C (inr_type b)][a:A][b:B]
 when C d e (inl_type a) ==> d a ||
 when C d e (inr_type b) ==> e b];

[case [A,B,C|Type][f:A->C][g:B->C] = when ([_:sum_type A B]C) f g];

[Sum_del1 = [s,t|Type][S:Pred s][T:Pred t][c:sum_type s t]
            or (Ex [x:s]and(EQ (inl_type x) c) (S x))
               (Ex [y:t]and(EQ (inr_type y) c) (T y))];
```

Remark In the above definition, we wrote
`[sum_type:([tau:Type]tau->tau->tau) Type]`
for the sum type-constructor. This device, due to H.Goguen, using an η-expansion of the ambiguous expression "`Type->Type->Type` " allows us to define a sum type at all type levels, using Pollack's typical ambiguity translation, in such a way that

$$\frac{\Gamma \vdash s : \mathit{Type}_i \quad \Gamma \vdash t : \mathit{Type}_i}{\Gamma \vdash s + t : \mathit{Type}_i} \quad (i \in \omega).$$

[11] Datatype definitions follow almost *verbatim* those in Martin-Löf type theory. For an eloquent account, see [79].

```
Lego> case_del1;
value = [s,t|Type][S:Pred s][T:Pred t][u|Type][U:Pred u]
        [FF:del1 S U][GG:del1 T U][f=FF.1][F=FF.2][g=GG.1][G=GG.2]
        ([c:sum_type s t]case f g c,
         [x|sum_type s t][h:Sum_del1 S T x]
         when ([c:sum_type s t](Sum_del1 S T c)->U (case f g c)) ...)

type  = {s,t|Type}{S:Pred s}{T:Pred t}{u|Type}{U:Pred u}
        (del1 S U)->(del1 T U)->del1 (Sum_del1 S T) U
```

3.1.9 Semi-exponentials

The notion of first-order deliverable is based on the predicate Del_1 on terms of arrow type. Precisely this predicate defines the specification which yields a semi-exponential object in $\mathbf{del}_1$. λ-abstraction and evaluation then follow from those operations in the underlying type theory.

```
Lego> lambda_del1;
value = [s,t,u|Type][S|Pred s][T|Pred t][U|Pred u]
        [FF:del1 (Product_del1 S T) U][f=FF.1][F=FF.2]
        ([x:s][y:t]f (x,y),[x|s][h:S x][y|t][h1:T y]F (pair h h1)
         :<f:s->t->u>Del1 S (Del1 T U) f)
type  = {s,t,u|Type}{S|Pred s}{T|Pred t}{U|Pred u}
        (del1 (Product_del1 S T) U)->del1 S (Del1 T U)

Lego> evdel1;
value = [t,u|Type][T1:Pred t][U1:Pred u]
        ([p:(t->u)#t][h=p.1][y=p.2]h y,
         [p|(t->u)#t][hyp:Product_del1 (Del1 T1 U1) T1 p]
            fst hyp (snd hyp))
type  = {t,u|Type}{T:Pred t}{U:Pred u}
                  del1 (Product_del1 (Del1 T U) T) U

Lego> hayashi3;
value = [s,t,u,w|Type][S|Pred s][T|Pred t][U|Pred u][W|Pred w]
        [FF|del1 (Product_del1 S T) U][GG|del1 W S][HH|del1 W T]
```

```
        reflEQ ...
type  = {s,t,u,w|Type}{S|Pred s}{T|Pred t}{U|Pred u}{W|Pred w}
        {FF|del1 (Product_del1 S T) U}{GG|del1 W S}{HH|del1 W T}
        EQ (compose_del1
             (pair_del1 (compose_del1 GG (lambda_del1 FF)) HH)
             (evdel1 T U))
           (compose_del1 (pair_del1 GG HH) FF)

Lego> hayashi4;
value = [s,t,u,w|Type][S|Pred s][T|Pred t][U|Pred u][W|Pred w]
        [FF|del1 (Product_del1 S T) U][GG|del1 W S]
        reflEQ (compose_del1 GG (lambda_del1 FF))
type  = {s,t,u,w|Type}{S|Pred s}{T|Pred t}{U|Pred u}{W|Pred w}
        {FF|del1 (Product_del1 S T) U}{GG|del1 W S}
        EQ (compose_del1 GG (lambda_del1 FF))
           (lambda_del1
             (compose_del1
               (pair_del1 (compose_del1 (pi1_del1 W T) GG)
                          (pi2_del1 W T))
                FF))
```

3.1.10 Semi-pullbacks and the internalisation of equality

It is perhaps tempting, at first sight, to imagine that $\mathbf{del}_1$ has more structure than that outlined above, given the evident (and naïve) set-theoretic character of the foregoing, in particular, even the existence of all finite semi-limits and semi-colimits. However, this appears not to be the case. Since equality of arrows is defined using the underlying conversion relation of the calculus, slightly modified, the construction of pullbacks, or equalisers, would require us to internalise this notion, which is known not to be possible: representing the convertibility relation as a proposition would lead to absurdity, in the presence of a non-empty context, for example one containing an assumption that all terms were interconvertible. It is possible to define an object which ought to define the vertex of a pullback square in $\mathbf{del}_1$, *viz.*

$$(s \times t, \lambda p{:}s \times t.\ S(\pi_1(p)) \wedge (EQ_u\ f(\pi_1(p))\ g(\pi_2(p))) \wedge T(\pi_2(p))),^{12}$$

given a cone of the form

$$\begin{tikzcd} & (t,T) \arrow[d, "(g.G)"] \\ (s,S) \arrow[r, "(f,F)"'] & (u,U) \end{tikzcd}$$

together with projection maps (given essentially by π_1, π_2), and even show that the mediating arrow

$$(\lambda z{:}v.\ (pz, qz), \lambda z{:}v.\ \lambda h{:}Vz.\ conj\ (Pzh)\ (\mathit{reflEQ}(pz, qz))\ (Qzh))^{13}$$

from a commuting square

$$\begin{tikzcd} (v,V) \arrow[r, "(q,Q)"] \arrow[d, "(p,P)"'] & (t,T) \arrow[d, "(g,G)"] \\ (s,S) \arrow[r, "(f,F)"'] & (u,U) \end{tikzcd}$$

is semi-functorial in $(f, F), (g, G)$. But one cannot pass from an arbitrary proof of $EQ_u\ f(\pi_1(p))\ g(\pi_2(p))$ to the knowledge that $f(\pi_1(p)) \simeq_{\beta\delta} g(\pi_2(p))$. That is to say, the proposed limit cone is not even a cone.

Similar remarks apply to any attempt to characterise the monomorphisms, epimorphisms, and idempotents in $\mathbf{del}_1$. This suggests that perhaps other notions of equality on arrows might make for a smoother definition, but apart from

---

[12]Here, EQ is Leibniz' equality defined in Chapter 2.

[13]$conj$ is the term which, given proofs p_1, p_2, p_3 of three propositions ϕ, ψ, χ, returns the proof of the conjunction $\phi \wedge \psi \wedge \chi$.

a few remarks on an extension of the idea of deliverables to considering types together with partial equivalence relations in Section 6.1, I have not pursued this problem.

3.1.11 A factorisation system on $\mathbf{del}_1$

Every function — that is to say a term of arrow type — gives rise to a deliverable, very much in the manner of the assignment rule of Hoare logic [40] or Dijkstra's predicate transformer for assignment[22,25]. Namely, for

$$f : s \longrightarrow t, P : t \longrightarrow Prop$$

we obtain

$$(f, \lambda x{:}s.\lambda h{:}f^{\star}P.h){:}\mathbf{del}_1\ f^{\star}P\ P, \text{ where } f^{\star}P =_{\text{def}} \lambda x{:}s.P(fx).$$

We call these deliverables *trivial*, since they come with vacuous proofs of correctness. On the algorithmic side, we may distinguish those deliverables which embody a trivial algorithm — the identity. These correspond to propositional reasoning. In a suitable sense, all of $\mathbf{del}_1$ lies between these two extremes.

Remark Every first-order deliverable factorises as a trivial deliverable followed by propositional reasoning.

Proof Obvious.

$$\mathcal{S} \xrightarrow{(f,F)} \mathcal{T} = \mathcal{S} \xrightarrow{(id,F)} f^{\star}\mathcal{T} \xrightarrow{(f,id)} \mathcal{T}$$

■

3.1.12 A consequence rule

Logical implication induces a pointwise ordering $\subset$ on predicates, for which we have the following consequence rule, in the manner of Hoare logic:

$$\frac{\mathcal{S} \subset \mathcal{S}' \quad \mathcal{S}' \xrightarrow{(f,F')} \mathcal{T}' \quad \mathcal{T}' \subset \mathcal{T}}{\mathcal{S} \xrightarrow[(f,F)]{} \mathcal{T}}$$

In general, we will not have need of it, in view of the above remark, which incorporates propositional reasoning into the general framework of deliverables.

3.1.13 Pointwise construction

A basic combinator in the theory of deliverables constructs a function-proof pair from a function which returns value-proof pairs[14]. Mendler, in his thesis [74], calls such gadgets "pointwise designs": for each argument value x, the pointwise existence of a value y (of type t) satisfying some property Tv, yields a deliverable with codomain (t, T). In detail,

$$\frac{\mathcal{F}\colon \Pi x{:}s.\ \Sigma y{:}t.\ (Sx) \Longrightarrow (Ty)}{\mathcal{S} \xrightarrow{(f,F)} \mathcal{T}}$$

where $f =_{\text{def}} \lambda x{:}s.\ \pi_1(\mathcal{F}x)$, and $F =_{\text{def}} \lambda x{:}s.\ \lambda h{:}Sx.\ \pi_2(\mathcal{F}x)h$.

3.1.14 Inductively defined types

Provided we accept a weak definition of inductive type, it is relatively straightforward to add inductive types to the categorical structure developed so far.

---

[14]This just corresponds to Howard's observation, emphasised by Martin-Löf, that given a strong interpretation of the existential quantifier as Σ-type, the axiom of choice becomes constructively valid [42,70,71].

Categorical accounts of inductive types, via initial algebras, impose extra equalities on the iterator, due to the uniqueness clause in the definition of initial algebra. As with the semi-structures above, we only have existence, and not uniqueness, of the relevant universal arrows.

The basic idea is very simple: we add inductive types at the $Type$ level, with a strong[15] elimination rule, yielding a simply typed recursor at the $Type$ level, and the usual induction principle at the $Prop$ level. This type is then paired with the identically true predicate. The elimination rule for first-order deliverables is easily derived, by packaging primitive recursion at the type level with induction at the predicate level. We illustrate this general idea by considering the case of natural numbers and lists.

Natural numbers

We assume, as in Martin-Löf type theory the existence of a type of natural numbers, with two constructors, zero and successor. This yields the well-formed context

$$nat{:}\,Type, 0{:}nat, S{:}nat \longrightarrow nat$$

We typically abbreviate S to "$+1$" in informal mathematical language. We extend this context with a dependent elimination constant $natrecd$[16],

$$natrecd{:}\Pi C{:}nat \longrightarrow Type.\ \Pi z{:}C0.\ \Pi s{:}(\Pi k{:}nat.\ \Pi ih{:}Ck.\ C(k+1)).\ \Pi n{:}nat.\ Cn$$

---

[15] *Strong*, that is, because we can eliminate over *types*, and not merely propositions.

[16] "*d*" for dependent.

together with reduction rules defining the δ-redices[17] (in some context where C, z, s, n have the appropriate types):

- $\mathit{natrecd}\ C\ z\ s\ 0 \leadsto z$;
- $\mathit{natrecd}\ C\ z\ s\ (n+1) \leadsto s\ n\ (\mathit{natrecd}\ C\ z\ s\ n)$.

This is precisely expressed in LEGO as follows[18] :

```
[nat:Type(0)];
[zero:nat];
[succ:nat -> nat];
[natrecd:{C:nat->Type}
         {z:C zero}{s:{k:nat}{ih:C k}C (succ k)}{n:nat}C n];
[[n:nat][C:nat->Type][z:C zero][s:{k:nat}{ih:C k}C (succ k)]
   natrecd C z s zero ==> d
|| natrecd  C z s (succ n) ==> s n (natrecd C z s n)].
```

This yields a derived iterator and primitive recursor

$$\mathit{natiter}: \Pi\alpha{:}\mathit{Type}.\ \alpha \longrightarrow (\alpha \longrightarrow \alpha) \longrightarrow \alpha$$

and

$$\mathit{natrec}: \Pi\alpha{:}\mathit{Type}.\ \alpha \longrightarrow (\mathit{nat} \longrightarrow \alpha \longrightarrow \alpha) \longrightarrow \alpha$$

where[19]

$$\mathit{natiter}\ z\ s\ 0 \quad =_{\mathrm{def}} \quad z$$

---

[17] *cf.* Martin-Löf type theory, or Gödel's earlier system **T** of functionals [104]. We essentially use this language of primitive recursion in all finite types as our programming language.

[18] *cf.* the discussion in Chapter 2 of adding arbitrary reductions in LEGO.

[19] The type α is, of course, inferred by the typechecker.

$$\begin{aligned}
natiter\ z\ s\ (n+1) &=_{\text{def}} s\ (natiter\ z\ s\ n)\\
natrec\ z\ s\ 0 &=_{\text{def}} z\\
natrec\ z\ s\ (n+1) &=_{\text{def}} s\ n\ (natrec\ z\ s\ n),
\end{aligned}$$

and an induction principle

$$natind : \Pi\Phi{:}nat \longrightarrow Type.\ \Pi z{:}\Phi(0).\ \Pi s{:}(\Pi k{:}nat.\ \Pi ih{:}\Phi(k).\ \Phi(k+1)).\ \Pi n{:}nat.\ \Phi n.$$

Our methodology suggests we examine derived induction principles for the iterator and recursor, since we are interested in programs, in this case of the form $natiter\ z\ s$ or $natrec\ z\ s$, together with proven propositions about them. For the iterator, we obtain

$$\frac{\Phi(z) \quad \forall y{:}\alpha.\ \Phi(y) \Longrightarrow \Phi(s\ y)}{\forall n{:}nat.\ \Phi(natiter\ z\ s\ n)}[z{:}\alpha,\ s{:}\alpha \longrightarrow \alpha]$$

and for the recursor

$$\frac{\Phi(z) \quad \forall k{:}nat.\ \forall y{:}\alpha.\ \Phi(k) \Longrightarrow \Phi(s\ k\ y)}{\forall n{:}nat.\ \Phi(natrec\ z\ s\ n)}[z{:}\alpha,\ s{:}nat \longrightarrow \alpha \longrightarrow \alpha].$$

Since we are interested in building new deliverables from less complex ones, we could of course use the dependent eliminator $natrecd$ to construct terms of type $\mathbf{del}_1$, but in doing so we violate the separation of proofs from programs which distinguishes our approach. So we package recursion at the $Type$ level with induction at the $Prop$ level in a pair.

We introduce the predicate $Nat =_{\text{def}} \lambda n{:}nat.\ true$ on the natural numbers. For each constructor of the type, we obtain a corresponding deliverable:

$$\frac{}{Zero =_{\text{def}} (\lambda u{:}unit.\ 0,\ \lambda u{:}unit.\ \lambda h{:}\,Unit\ u.\ \top){:}\,Unit \longrightarrow Nat}$$

$$\frac{}{Succ =_{\text{def}} (\lambda n{:}nat.\ S\,n,\ \lambda n{:}nat.\ \lambda h{:}Nat\,n.\ \top){:}Nat \longrightarrow Nat}$$

For the iterator, we obtain

$$\frac{Unit \xrightarrow{(z,Z)} (t,T) \quad (t,T) \xrightarrow{(s,S)} (t,T)}{Natiter\,(z,Z)\;(s,S){:}(nat,\;Nat) \longrightarrow (t,T)}$$

and for the recursor

$$\frac{Unit \xrightarrow{(z,Z)} (t,T) \quad Nat \xrightarrow{(s,S)} (t,T)^{(t,T)}}{Natrec\,(z,Z)\;(s,S){:}(nat,\;Nat) \longrightarrow (t,T)}$$

where the function component of $Natiter\ (z,Z)\ (s,S)$ (respectively $Natrec$) is $natiter\ (z())\ s$ (respectively $natrec$), and the proof component is obtained from the appropriate derived induction principle above.

These terms are easily obtained by refinement in LEGO:

```
Lego> natiter_del1;
value = [t|Type][T|Pred t][ZZ:del1 Unit T][SS:del1 T T]
        [z=ZZ.1 void][Z=ZZ.2][s=SS.1][S=SS.2]
        (natiter z s,
         natind ([m:nat](Nat m)->T (natiter z s m)) ... )
type  = {t|Type}{T|Pred t}(del1 Unit T)->(del1 T T)->del1 Nat T
Lego> natrec_del1;
value = [t|Type][T|Pred t][ZZ:del1 Unit T][SS:del1 Nat (Del1 T T)]
        [z=ZZ.1 void][Z=ZZ.2][s=SS.1][S=SS.2]
        (natrec z s,
         natind ([m:nat](Nat m)->T (natrec z s m)) ...)
type  = {t|Type}{T|Pred t}
        (del1 Unit T)->(del1 Nat (Del1 T T))->del1 Nat T
```

Example As an example of the use of these combinators, we present a correctness proof of a doubling function, given by

$$double =_{\text{def}} \lambda n{:}nat.\ natiter\,0\ (\lambda k{:}nat.\ k+2)\ n.$$

Suppose we wish to show that $double\ n$ is even for all natural numbers n. Posed in terms of deliverables, we seek a term of type $\mathbf{del}_1\ Nat\ Even$, whose function component is $double$.

Using the rule for $Natiter$ above, the problem reduces to finding:

base case $Unit \xrightarrow{(z,Z)} (nat, Even)$. We take $z =_{\text{def}} \lambda u{:}unit.\ 0$, and use the proof that 0 is even from Chapter 2;

step case $(nat, Even) \xrightarrow{(s,S)} (nat, Even)$. We simply use the first-order deliverable we constructed in the sample derivation at the end of Chapter 2.

We thus obtain a non-trivial recursive first-order deliverable[20].

Lists

In much the same way as above, we may define combinators for deliverables over a type of lists. As before, we extend the context with a type constructor, in this case $list{:}\,Type_i \longrightarrow Type_i$, which we may express in LEGO using Goguen's trick of Section 3.1.8 as

```
[list:([tau:Type]tau->tau)Type];
```

together with constructors the usual nil and $cons$, and a dependent eliminator $listrecd$. Again we derive an iterator $listiter$, a primitive recursor $listrec$ and an induction combinator $listind$.

When it comes to considering the derived induction principles for the iterator and recursor, however, we now have the freedom to specify recursions over lists of elements satisfying some predicate, rather than over all lists of the parameter type a. That is, given some specification (a, A), we obtain a derived specification $List\ (a, A) =_{\text{def}} (list\ a,\ Listof\ A)$, where

$$\begin{aligned} Listof\ A\ (nil\ a) &=_{\text{def}} true \\ Listof\ A\ (cons\ x\ l) &=_{\text{def}} (A\,x) \wedge (Listof\ A\ l) \end{aligned}$$

[20]This example is originally due to Burstall.

defines $Listof\ A$ by primitive recursion.

We may then proceed in the same way as above, obtaining constructors

$$\frac{}{\overline{Nil =_{\text{def}} (\lambda u{:}unit.\ nil\ a,\ \lambda u{:}unit.\ \lambda h{:}\,Unit\ u.\ \top){:}\,Unit \longrightarrow Listof\ A}}$$

and

$$\frac{}{\overline{Cons{:}A \longrightarrow (Listof\ A)^{Listof\ A}}}$$

with function component,

$$\lambda x{:}a.\ \lambda l{:}list\ a.\ cons\ x\ l$$

and proof component

$$\lambda x{:}a.\ \lambda p{:}A\ x.\ \lambda l{:}list\ a.\ \lambda q{:}Listof\ A\ l.\ pair\ p\ q.$$

Likewise, we package together recursion and an appropriate derived induction principle, to obtain the following rules for constructing deliverables: an iterator,

$$\frac{Unit \xrightarrow{(n,N)} (t,T) \quad (a,A) \xrightarrow{(c,C)} (t,T)^{(t,T)}}{Listiter\ (n,N)\ (c,C){:}\ (list\ a,\ Listof\ A) \longrightarrow (t,T)};$$

and a recursor

$$\frac{Unit \xrightarrow{(n,N)} (t,T) \quad (a,A) \xrightarrow{(c,C)} (list\ a,\ Listof\ A)^{(t,T)^{(t,T)}}}{Listrec\ (n,N)\ (c,C){:}\ (list\ a,\ Listof\ A) \longrightarrow (t,T)}.$$

In LEGO we have the following:

```
Lego> listiter_del1;
value = [s,t|Type][S|Pred s][T|Pred t]
        [NN:del1 Unit T][CC:del1 S (Del1 T T)]
        [n=NN.1 void][N=NN.2][c=CC.1][C=CC.2]
```

```
        (listiter n c,
         listind ([l:list s](Listof S l)->T (listiter n c l)) ...)
type  = {s,t|Type}{S|Pred s}{T|Pred t}
        (del1 Unit T)->(del1 S (Del1 T T))->del1 (Listof S) T
Lego> listrec_del1;
value = [s,t|Type][S|Pred s][T|Pred t]
        [NN:del1 Unit T][CC:del1 S (Del1 (Listof S) (Del1 T T))]
        [n=NN.1 void][N=NN.2][c=CC.1][C=CC.2]
        (listrec n c,
         listind ([l:list s](Listof S l)->T (listrec n c l)) ...)
type  = {s,t|Type}{S|Pred s}{T|Pred t}
        (del1 Unit T)->(del1 S (Del1 (Listof S) (Del1 T T)))
        ->del1 (Listof S) T
```

3.2 Second-order deliverables

The system which we have described above amounts to a functional version of the well known invariants used in proofs of imperative programs. Unfortunately the specification makes no connection between the input and the output of the function. All we say is that if the input satisfies property S then the output satisfies property T, but there is no *relation* between them. For example we might specify that a sorting function takes lists to ordered lists, but we cannot specify that the output is a permutation of the input. The function might always produce the empty list, which is indeed sorted, but not very interesting. As a matter of fact the classical invariant proofs have the same weakness, masked by the tacit assumption that some variable which is carried through the computation does not change its value. To enforce the constraint that the output bear some relation to the input, we need to develop a compositional theory in which relations are the basic objects of study, with a notion of arrow which respects relations, rather than predicates.

3.2.1 A thought experiment

Suppose we are given some Π^0_2 specification $\forall x{:}s.\ \exists z{:}u.\ R(x, z)$, and we wish to find some function $f{:}s \longrightarrow u$ which satisfies it. In what sense may we refine such specifications by composition? Suppose we wish to instantiate f via the composition $f = g; h$ of two functions

$$s \xrightarrow{\quad g \quad} t \xrightarrow{\quad h \quad} u$$

where t is some intermediate type. Then, following our intuition in the case of predicates, we anticipate some intermediate specification $Q(x, y) \quad [x{:}s\ , y{:}t]$, such that g solves

$$\forall x{:}s.\ \exists y{:}t.\ Q(x, y),$$

and h solves

$$\forall x{:}s.\ (\exists y{:}t.\ Q(x,y)) \Longrightarrow \exists z{:}t.\ R(x,z).$$

This last is logically equivalent to

$$\forall x{:}s.\ \forall y{:}t.\ Q(x,y) \Longrightarrow \exists z{:}t.\ R(x,z).$$

But now we are left in something of a quandary: h, our intended solution, makes no reference to the intermediate value of y. Also, we have introduced an asymmetry between the rôles of g and h. A remedy, which underlies the definitions 3.2.2, and 3.2.3 below, is to separate the rôles of the independent parameter x and the dependent variables y, z.

We consider relations such as Q, R as the objects of study, for a fixed type s, but allowing the types t, u to vary. The following provides an appropriate notion of morphism which re-establishes a symmetry between Q and R, and their corresponding instantiations g, h.

Definition 3.2.1 An arrow from $Q(x,y) \quad [x{:}s\ , y{:}t]$ to $R(x,z) \quad [x{:}s\ , z{:}u]$ consists of the following data:

- a function $f{:}s \longrightarrow t \longrightarrow u$, that is to say a function of *two* arguments; this recovers the missing dependence we observed above;
- a proof $F{:}\forall x{:}s.\ \forall y{:}t.\ Q(x,y) \Longrightarrow R(x,(f\,x\,y))$.

The composition of two such gadgets

$$P(x,w) \xrightarrow{(g,G)} Q(x,y) \xrightarrow{(h,H)} R(x,z)$$

is definable as [21]

$$(\lambda x, w{:}s, r.\ h\,x\,(g\,x\,w),\quad \lambda x, w{:}s, r.\ \lambda p{:}P(x,w).\ H\,x\,(g\,x\,w)\,(G\,x\,w\,p)).$$

[21] We employ an informal alphabetical convention: x is of type s, y of type t, z of type u, and hence w is of type r.

We have now established a definition which respects the symmetry of source and target in our previous analysis of the decomposition of the Π^0_2 specification $\forall x{:}s.\ \exists z{:}u.\ R(x, z)$. In so doing, we have generalised the notion of specification, and our old specification corresponds in this new setting to choosing $r =_{\mathrm{def}} unit$, $P(x, w) =_{\mathrm{def}} true$, and $f_{old} =_{\mathrm{def}} \lambda x{:}s.\ f_{new}\, x\, ()$.

Remark We might complement this discussion by remarking that the incorporation of x as a value which remains constant through the computation of f is reminiscent of the use of "ghost" variables in the classical invariant proofs[22]. To overcome the limitation of Hoare's logic in providing no connection between the initial and final values of the program variables, these "ghost" variables recorded the initial state, and remained unchanged, indeed even unmentioned, by the program. Here, we have the advantage that not only are our programs f allowed to mention x, but the scope of x is also delimited by the universal quantifier in the proof F.

3.2.2 Basic definitions

In view of the above discussion, we relativise specifications and first-order deliverables to depend on some input type s. Indeed, by observing that we may uniformly impose some condition S on the input parameter $x{:}s$, without affecting the notion of composition, we arrive at Burstall's definition of a "second-order" deliverable.

Definition 3.2.2 *relativised specification*
Suppose $\Gamma \vdash s : \mathit{Type}$, $\Gamma \vdash S : s \longrightarrow \mathit{Prop}$. Then a *relativised specification with respect to* (s, S) is given by a pair of terms t, R, such that

[22] This seems to have been a guiding intuition for Burstall [private communication].

- $\Gamma \vdash t : \mathit{Type}$, and
- $\Gamma \vdash R : s \longrightarrow t \longrightarrow \mathit{Prop}$.

Definition 3.2.3 (Burstall) *second-order deliverable*

Suppose $\Gamma \vdash s : \mathit{Type}$, $\Gamma \vdash S : s \longrightarrow \mathit{Prop}$. Given two relativised specifications (t, Q) and (u, R), a *second-order deliverable over* (s,S) is a term $\mathcal{F}$ such that

$$\Gamma \vdash \mathcal{F} : \Sigma f : s \longrightarrow t \longrightarrow u.\forall x : s.S(x) \Longrightarrow \forall y : t.Q(x, y) \Longrightarrow R(x, (fxy)).$$

We define $\mathrm{Del}_2\ S\ Q\ R$ to be the predicate

$$\lambda f{:}s \longrightarrow t \longrightarrow u.\ \forall x : s.Sx \Longrightarrow \forall y : t.Q(x, y) \Longrightarrow R(x, (fxy)).$$

Definition 3.2.3 embodies the idea that, for each $x{:}s$ such that Sx holds, (fx, Fx) is a first-order deliverable from Qx to Rx, where $\mathcal{F} =_{\mathrm{def}} (f, F)$. We may make this precise with the following constructions.

Proposition 3.2.1 (Family construction) *Suppose that we are given*

- $\Gamma \vdash s : \mathit{Type}$,
- $\Gamma \vdash S : s \longrightarrow \mathit{Prop}$, *and*
- *two relativised specifications* $(t, Q), (u, R)$ *with respect to* (s, S).

If

- $\Gamma, x{:}s \vdash f_x : t \longrightarrow u$, *and*
- $\Gamma, x{:}s, h{:}Sx \vdash F_{x,h} : \mathit{Del}_1\ (Qx)\ (Rx)\ f_x$

then

$$(\lambda x{:}s.\ \lambda y{:}t.\ f_x\, y,\ \lambda x{:}s.\ \lambda h{:}Sx.\ \lambda y{:}t.\ \lambda p{:}Q\, x\, y.\ F_{x,h}\, y\, p)$$

defines a second-order deliverable over (s, S).

Proposition 3.2.2 (Relativisation of first-order deliverables) *Every second-order deliverable arises in this way.*

Proof As constructions in LEGO, we obtain

```
Lego> family_of_del1_to_del2;
value = [s,t,u|Type][S|Pred s][P|Rel s t][Q|Rel s u]
        [family:{x:s}<fx:t->u>(S x)->Del1 (P x) (Q x) fx]
        ([x:s][y:t]((family x)).1 y,
         [x|s][h:S x][y|t][p:P x y]((family x)).2 h p)
type  = {s,t,u|Type}{S|Pred s}{P|Rel s t}{Q|Rel s u}
        ({x:s}<fx:t->u>(S x)->Del1 (P x) (Q x) fx)->del2 S P Q
Lego> del2_to_family_of_del1;
value = [s,t,u|Type][S|Pred s][P|Rel s t][Q|Rel s u]
        [FF|del2 S P Q][x:s][f=FF.1][F=FF.2]
        (f x, F:<fx:t->u>(S x)->Del1 (P x) (Q x) fx)
type  = {s,t,u|Type}{S|Pred s}{P|Rel s t}{Q|Rel s u}
        (del2 S P Q)->{x:s}<fx:t->u>(S x)->Del1 (P x) (Q x) fx.
```

■

These suggest that the study of second-order deliverables amounts to the study of first-order deliverables in an extended context. In particular we expect to obtain, for a given specification $\mathcal{S} =_{\text{def}} (s, S)$, a category structure on the second-order deliverables over $\mathcal{S}$. We make a similar definition of equality on second-order deliverables to that given in Section 3.1 above, the details of which are left to the reader. Then we can indeed define identities and composition. Composition is given as in Definition 3.2.1 in the thought experiment above. We call this category $\mathbf{del}_2\mathcal{S}$.

Lemma 3.2.1 *Identities and composition in $\mathbf{del}_2\mathcal{S}$ are given by the following LEGO terms:*

```
Lego> id_del2;
```

```
value = [s,t|Type][S:Pred s][R:Rel s t]
        ([_:s][y:t]y,[x|s][_:S x][y|t][p:R x y]p)
type  = {s,t|Type}{S:Pred s}{R:Rel s t}del2 S R R.
Lego> compose_del2;
value = [s,t,u,v|Type][S|Pred s][P|Rel s t][Q|Rel s u][R|Rel s v]
        [FF:del2 S P Q][GG:del2 S Q R][f=FF.1][F=FF.2][g=GG.1][G=GG.2]
        ([x:s][y:t]g x (f x y),
         [x|s][h:S x][y|t][p:P x y]G h (F h p):<f:s->t->v>Del2 S P R f)
type  = {s,t,u,v|Type}{S|Pred s}{P|Rel s t}{Q|Rel s u}{R|Rel s v}
        (del2 S P Q)->(del2 S Q R)->del2 S P R.
```

Proof Immediate from the above constructions. The proofs of the identity and composition laws are similarly trivial. For details, see Appendix B. ■

If (f, F) is a second-order deliverable over (s, S), we typically write

$$(t, Q) \xrightarrow{(f,F)} (u, R) \quad [(s, S)]\ , \text{ or even } Q \xrightarrow{(f,F)} R \quad [S],$$

since, as usual, the types s, t, u may be inferred by the typechecker. Our notation is intended to indicate that we are considering deliverables relative to some assumption defined by the specification (s, S). This notation is deliberately intended to echo the style of contexts in Martin-Löf type theory. We shall return to this idea, in rather greater detail, in Chapter 5.

3.2.3 Each $\mathbf{del}_2\mathcal{S}$ is a semi-ccc

As in Section 3.1, we work relative to some context Γ. We have seen how a second-order deliverable (f, F) over (s, S) in context Γ may be viewed as arising from a first-order deliverable in the extended context $\Gamma, x{:}s, h{:}Sx$. The conditions of definitions 3.2.2, and 3.2.3 are intended to enforce a hierarchy of dependencies in this extended context. The type t must not depend on x or h. The relation R, considered as a predicate on t in context $\Gamma, x{:}s$ does not depend on h. The function component f may not depend on h, but the proof component F may do so.

Given these conditions, we may lift the structure in $\mathbf{del}_1$, by observing that the various constructions of Section 3.1 respect this hierarchy of dependencies. The predicates concerned need to be modified to include an explicit hypothesis Sx. We arrive at the following result.

Theorem 3.2.1 *For each specification $\mathcal{S}$, $\mathbf{del}_2\mathcal{S}$ has the structure of a semi-ccc.*

Proof We merely sketch some of the constructions, on the basis of the above informal intuition. Full details are left to Appendix B.

Semi-terminal object This is given by the relativised specification

$$1_S =_{\text{def}} (unit,\ \lambda x{:}s.\ \lambda u{:}unit.\ true).$$

The shriek map $!_{(t,Q)}$, from some relativised specification (t, Q) to 1_S, has function component $\lambda x{:}s.\ \lambda y{:}t.\ ()$, and proof component

$$\lambda x{:}s.\,\lambda h{:}Sx.\,\lambda y{:}t.\,\lambda p{:}P\,x\,w.\ \top^{23}.$$

Semi-products Suppose we are given two relativised specifications $(t, Q), (u, R)$.

Then we may form the relativised specification

$$(t, Q) \times (u, R) =_{\text{def}} (t \times u,\ \lambda x{:}s.\ \lambda p{:}t \times u.\ (Qx(\pi_1 p)) \wedge (Rx(\pi_2 p)))$$

---

[23] Here, $\top$ is the term corresponding to $true$-introduction,

$$\top =_{\text{def}} \lambda\phi{:}Prop.\ \lambda p{:}\phi.\ p$$

of type $true =_{\text{def}} \Pi\phi{:}Prop.\ \phi \Rightarrow \phi$. In LEGO,

```
Lego> top;
value = [A|Prop][a:A]a
type = {A|Prop}A->A
```

This defines a semi-product object in $\mathbf{del}_2(s, S)$. The pairing map is given by

$$\frac{P \xrightarrow{(f,F)} Q \quad [S] \quad P \xrightarrow{(g,G)} R \quad [S]}{P \xrightarrow[(<f,g>,\ <F,G>)]{} Q \times R \quad [S]}$$

where

- $< f, g > =_{\text{def}} \lambda x{:}s.\ \lambda w{:}r.\ \mathbf{pair}_{t\times u}(f\,x\,w,\ g\,x\,w)$, and
- $< F, G > =_{\text{def}} \lambda x{:}s.\ \lambda h{:}S x.\ \lambda w{:}r.\ \lambda p{:}P\,x\,w.\ pair(F\,x\,h\,w\,p)(G\,x\,h\,w\,p)$ [24].

Projections are similarly straightforward to define:

$$P \times Q \xrightarrow{\pi} P \quad [S]$$

has function component $\lambda x{:}s.\ \lambda yz{:}t \times u.\ \pi_1(yz)$, and proof component [25]

$$\lambda x{:}s.\ \lambda h{:}S x.\ \lambda yz{:}t \times u.\ \lambda p{:}(P\,x\,\pi_1(yz)) \wedge (Q\,x\,\pi_2(yz)).\ fst\,p.$$

---

[24]Here, $pair$ is the term corresponding to $\wedge$-introduction,

$$pair =_{\text{def}} \lambda \phi, \psi{:}Prop.\ \lambda p{:}\phi.\ \lambda q{:}\psi.\ \lambda \chi{:}Prop.\ \lambda r{:}\phi \Rightarrow \psi \Rightarrow \chi.\ r\,p\,q$$

of type $\Pi\phi, \psi{:}Prop.\ \phi \Rightarrow \psi \Rightarrow \phi \wedge \psi$. In LEGO,

```
Lego> pair;
value = [A,B|Prop][a:A][b:B][C:Prop][h:A->B->C]h a b
type = {A,B|Prop}A->B->and A B
```

In the above λ-terms, we have suppressed arguments to $pair$ in accordance with LEGO's implicit syntax.

[25]As above, fst is a term corresponding to $\wedge$-elimination,

$$fst =_{\text{def}} \lambda \phi, \psi{:}Prop.\ \lambda c{:}\phi \wedge \psi.\ c\,\phi\,(\lambda p{:}\phi.\ \lambda q{:}\psi.\ p)$$

of type $\Pi\phi, \psi{:}Prop.\ \phi \wedge \psi \Rightarrow \phi$. In LEGO,

```
Lego> fst;
```

We omit the proof that this algebraic gadgetry does indeed define a semi-product structure in the sense of Proposition 3.1.2.

Semi-exponentials Just as the predicate Del_1 defined a semi-exponential object in the category $\mathbf{del}_1$, we may define a semi-exponential object in $\mathbf{del}_2\ (s, S)$, using the relativised specification to which Del_2 gives rise. More precisely, suppose $(r, P), (t, Q), (u, R)$ are relativised specifications. We obtain a relativised specification

$$R^Q =_{\mathrm{def}} (t \longrightarrow u,\ \lambda x{:}s.\ \lambda f{:}t \longrightarrow u.\ \forall y{:}t. Q\,x\,y \Longrightarrow R\,x\,(fy)).$$

If $\Gamma \vdash f : s \longrightarrow t \longrightarrow u$, and $\Gamma \vdash F : \mathrm{Del}_2\, S\, Q\, R\, f$, then $\Gamma, x{:}s \vdash Fx : R^Q\, fx$. Moreover, given

$$P \times Q \xrightarrow{\mathcal{F}} R \quad [S]$$

we obtain

$$P \xrightarrow{\Lambda(\mathcal{F})} R^Q \quad [S]$$

by currying in the obvious way: if $\mathcal{F} =_{\mathrm{def}} (f, F)$, then $\Lambda(\mathcal{F}) =_{\mathrm{def}} (\hat{f}, \hat{F})$, where

$$\hat{f} =_{\mathrm{def}} \lambda x{:}s.\ \lambda w{:}r.\ \lambda y{:}t.\ f\,x\,(w, y)$$

and

$$\hat{F} =_{\mathrm{def}} \lambda x{:}s. \lambda h{:}Sx. \lambda w{:}r. \lambda p{:}P\,x\,w. \lambda y{:}t. \lambda q{:}Q\,x\,y.\ F\,x\,h\,(w, y)\,(\mathit{pair}\,p\,q).$$

We may similarly define an evaluation map, details of which are left to the imaginative reader: it is rather less taxing to develop this construction by

```
value = [A,B|Prop][p:and A B]p A ([a:A][b:B]a)
type = {A,B|Prop}(and A B)->A
```

Again, we have suppressed arguments to fst in accordance with LEGO's implicit syntax.

refinement in LEGO. Likewise, the proofs that these data meet Hayashi's conditions for a semi-exponential are best dealt with by refinement. See Appendix B.

■

3.2.4 $\mathbf{del}_2$: an indexed category over $\mathbf{del}_1$

The categorically minded reader now asks herself what relationships exist between the various categories $\{\mathbf{del}_2\mathcal{S} | \mathcal{S} \in SPEC_1\}$, and to what extent we may elaborate upon the structure of this collection. In particular, she may ask what is the relationship between $\mathbf{del}_2\mathcal{S}$ and $\mathbf{del}_2\mathcal{T}$, given a first-order deliverable from $\mathcal{S}$ to $\mathcal{T}$. A moment's pause should convince her that composition in $\mathbf{del}_1$ should induce an operation on second-order deliverables, since they somehow are no more than first-order deliverables, except that they are defined in an extended context. In other words, we are groping towards the following theorem:

Theorem 3.2.2 *$\boldsymbol{del}_2$ is an indexed category [50,6] over $\boldsymbol{del}_1$, whose fibres are semi-cccs, with semi-cc structure strictly preserved by reindexing along arrows in $\boldsymbol{del}_1$.*

3.2.5 Pullback functors

The above theorem depends on the existence of pullback functors, which translate, or reindex, data between the categories $\mathbf{del}_2\mathcal{S}$. The obvious definition works, and moreover trivially respects the equality of objects and arrows, so we do indeed have pullback functors — and they are functors, not merely semi-functors, since identities and composition are preserved. It is then a straightforward, and tedious, task to verify that these operations compose, and strictly preserve the structure in each fibre.

Definition 3.2.4 *pullback along a first-order deliverable*

Suppose we are given specifications $\mathcal{S}, \mathcal{T}$, and a first-order deliverable $\mathcal{S} \xrightarrow{(k,K)} \mathcal{T}$. We define an operation of *pullback along* (k, K), where we abuse notation in the standard way by employing the same symbol for the operation on objects and arrows, as follows: given a relativised specification $\mathcal{Q} =_{\mathrm{def}} (u, Q)$ with respect to $\mathcal{T}$, let

$$(k, K)^{\star}\mathcal{Q} =_{\mathrm{def}} \lambda x{:}s.\ \lambda z{:}u.\ Q\,(kx)\,z;$$

moreover, given a relativised specification $\mathcal{R} =_{\mathrm{def}} (v, R)$, and a second-order deliverable

$$\mathcal{Q} \xrightarrow{(f,F)} \mathcal{R} \quad [\mathcal{T}]$$

we define $(k, K)^{\star}(f, F)$ to be the pair

$$(\lambda x{:}s.\ \lambda z{:}u.\ f\,(k\,x)\,z,\ \lambda x{:}s.\ \lambda h{:}S x.\ \lambda z{:}u.\ \lambda p{:}\mathcal{Q}\,(k\,x)\,z.\ F\,(k\,x)\,(K\,x\,h)\,z\,p).$$

Lemma 3.2.2 $(k, K)^{\star}\mathcal{Q}$ *is a relativised specification with respect to* $\mathcal{S}$*. Moreover,* $(k, K)^{\star}(f, F)$ *is a second-order deliverable from* $(k, K)^{\star}\mathcal{Q}$ *to* $(k, K)^{\star}\mathcal{R}$*.*

Proof Only the latter property requires any checking, but it is readily seen to be the case, for example by appeal to the typechecker:

```
Lego> pullback_del2_along_del1;
value = [s,t,u,v|Type][S|Pred s][T|Pred t][P|Rel s u][Q|Rel s v]
        [KK:del1 T S][k=KK.1][K=KK.2][FF:del2 S P Q][f=FF.1][F=FF.2]
        (compose f k, [y|t][h:T y][z|u][p:P (k y) z)]F (K h) pre)
type  = {s,t,u,v|Type}{S|Pred s}{T|Pred t}{P|Rel s u}{Q|Rel s v}
        {KK:del1 T S}[k=KK.1](del2 S P Q)->
        del2 T ([y:t][z:u]P (k y) z) ([y:t][a:v]Q (k y) a)
```

■

Lemma 3.2.3 $(k, K)^{\star}$ *preserves identities and composition.*

Proof Using the equality Lemma 2.2.1,

```
Lego> Goal EQ (pullback_del2_along_del1 KK (id_del2 S P))
              ([k=KK.1]id_del2 V ([x:v][y:t]P(k x) y));
Lego> Refine reflEQ;
*** QED ***
```

The case of composition is proved in exactly the same way. ■

So, indeed, we do have the the existence of functors between the fibres $\mathbf{del}_2$. That they are a satisfactory notion of reindexing requires us to show that they obey the condition $\mathcal{H}^\star; \mathcal{K}^\star \equiv (\mathcal{K}; \mathcal{H})^\star$. In fact, more is true. We have the following:

Lemma 3.2.4 *The reindexing is* strict, *in the sense that*

$$\mathcal{H}^\star; \mathcal{K}^\star = (\mathcal{K}; \mathcal{H})^\star.$$

Proof By inspection, using the definition of the composition of first-order deliverables. ■

We now turn to the remainder of Theorem 3.2.2, namely that the pullback functors preserve the structure of a semi-ccc in each fibre. As above, we find that the structure is preserved strictly. We examine only the case of exponentials, the cases of products and terminal object being exactly similar, and rather easier.

Lemma 3.2.5 *In the notation of Theorem 3.2.1 above, with $\mathcal{V} \xrightarrow{\mathcal{K}} \mathcal{S}$, we have*

$$\mathcal{K}^\star(R^Q) = (\mathcal{K}^\star R)^{\mathcal{K}^\star Q}.$$

Proof Straightforward typechecking. ■

Proposition 3.2.3 *Suppose $(r, P), (t, Q), (u, R)$ are relativised specifications with respect to $\mathcal{S}$. Given*

$$P \times Q \xrightarrow{\mathcal{F}} R \quad [S] \quad \textit{and} \quad \mathcal{V} \xrightarrow{\mathcal{K}} \mathcal{S}$$

we have $\Lambda(\mathcal{K}^\star\mathcal{F}) \equiv \mathcal{K}^\star\Lambda(\mathcal{F})$.

Proof Using the Equality Lemma 2.2.1. The only difficulty lies in having to coerce the above two terms into the same type, *viz.*

$$\mathbf{del}_2\ \mathcal{V}\ (\mathcal{K}^\star\mathcal{P}) \times (\mathcal{K}^\star\mathcal{Q})\ \mathcal{K}^\star\mathcal{R}.$$

But this is straightforward. ∎

Remark Since categorical structure defined by semi-adjunctions is not in general unique, we may ask what other structures of a semi-ccc we may put on $\mathbf{del}_2\mathcal{S}$. Hayashi's paper does not even define a notion of functor between semi-cccs which preserves structure. The above theorem is fortunate in not requiring us to develop this concept in any greater generality than the strict preservation we have observed. It turns out that for another choice of product and exponential object, in which we incorporate an extra hypothesis of the form Sx, we again obtain a semi-ccc structure. But now the pullback functors do not preserve this structure on the nose. Indeed they only preserve the structure in a *lax* sense, the laxity arising from the obvious ordering on predicates and relations. The exact sense of laxity would be difficult to make precise here. However, if this work were to be extended to consider refinement of specifications, as in for example Power's categorical analysis of data refinement [88], then we might expect appropriate lax notions to become important.

3.2.6 $\mathbf{del}_2$ has $\mathbf{del}_1$-indexed sums and products

As a consequence of this theorem, we might hope, in the light of [102], to give a semantics for Martin-Löf type theory in terms of deliverables. In particular, we would expect to interpret *dependent* products and sums of specifications. This would seem to be part of the development of the subset theory in [79]. Since all the structure is defined by semi-adjunctions, however, rather than adjunctions as in Seely's account of an extensional theory, we defer discussing such an idea — and the corollary that we may use a language of dependent types to describe and manipulate deliverables — to Chapter 5. In particular, we avoid discussing a technical difficulty in the definition of the structure of dependent products, which arises once again from the absence of surjective pairing. Namely, to give the structure of dependent Π, we must define right semi-adjoints to the weakening functors between fibres [48]. In order to be able to do this, we must restrict ourselves, in the absence of surjective pairing, to those relations defined over Σ-types, such that

$$\lambda p.\lambda z.Rpz \simeq_{\beta\delta} \lambda p.\lambda z.R(\pi_1(p), \pi_2(p))z.$$

3.2.7 Second-order deliverables for natural numbers and lists

In the context of second-order deliverables, the situation regarding inductive types is less well understood. We do not regard this section as giving a definitive account, but the examples of the next chapter suggest that we have a usable set of combinators for reasoning about recursive programs.

We take as our guiding motivation the derived induction principles of the last section. Since we now work in the relativised case, these will be subtly altered by the presence of the induction variable.

This means, for the case of natural numbers, that we now examine proofs of statements of the form[26]:

$$\forall n{:}nat.\ R\ n\ (natrec\ z\ s\ n)$$

where, for some type t, $z{:}t$ and $s{:}nat \longrightarrow t \longrightarrow t$. A proof of this, by induction, yields

$$R\ 0\ z \quad \text{and} \quad \forall k{:}nat.\ \forall y{:}t.\ R\ k\ y \Longrightarrow R\ (k+1)\ (s\ k\ y)$$

[26]We only consider $natrec$, since $natiter$ is a degenerate instance of it.

as the requisite hypotheses in the base and step cases. We may now recognise the second hypothesis as the logical component of some second-order deliverable, whose function component is s.

The question arises as to how to view the first hypothesis $R\ 0\ z$. Do we regard it as part of some first or second-order deliverable? In a sense, neither, in the choice we have made in the current version of deliverables. If we examine the derived rule of induction again, but this time rephrased as

$$\frac{\forall k{:}nat.\ \forall y{:}t.\ R\ k\ y \Longrightarrow R\ (k+1)\ (s\ k\ y)}{\forall n{:}nat.\ \forall z{:}t.\ R\ 0\ z \Longrightarrow R\ n\ (natrec\ z\ s\ n)}$$

this isolates how we currently view recursions at the second-order level. Namely, we see the function which recursively applies s to an *arbitrary* initial value z as the function component of some second-order deliverable, whose proof component is the proof by induction of the conclusion

$$\forall n{:}nat.\ \forall z{:}t.\ R\ 0\ z \Longrightarrow R\ n\ (natrec\ z\ s\ n).$$

As observed above, the hypothesis in the step case of induction arises as the proof component of a second-order deliverable

$$R \xrightarrow{(s,S)} (+1)^{\star}R \quad [1_{nat}]$$

where $(+1)^{\star}R$, otherwise written $R[n+1/n]$, is the relation

$$\lambda n{:}nat.\ \lambda y{:}t.\ R\ (n+1)\ y.$$

In like manner, we write $0^{\star}R$, or $R[0/n]$, for the relation

$$\lambda n{:}nat.\ \lambda y{:}t.\ R\ 0\ y.$$

We thus obtain the second-order deliverable constructor for *nat* recursions as the following derived rule

$$\frac{R \xrightarrow{(s,S)} (+1)^{\star}R \quad [1_{nat}]}{Natrec_2\ (s,S){:}0^{\star}R \longrightarrow R \quad [1_{nat}]}$$

where $Natrec_2$ has function component

$$\lambda n{:}nat.\ \lambda z{:}t.\ natrec\ z\ s\ n.$$

The principle reason for making this choice of representation is a pragmatic one, based partly on experience, and on the behaviour of unification in the typechecker. If we were to mimic the construction of first-order deliverables by induction, we would expect some rule with one hypothesis for each constructor of the datatype, such as for example

$$\frac{1 \xrightarrow{(z,Z)} 0^{\star}R \quad [1_{nat}] \quad R \xrightarrow{(s,S)} (+1)^{\star}R \quad [1_{nat}]}{Natrec_2'\ (z,Z)\ (s,S){:}1 \to R \quad [1]}.$$

We would typically apply such a rule in a top-down proof, to a subgoal of the form

```
del2 ?n ?m R
```

In a top-down development, where we may construct deliverables using all the constructions described above, we would like the instantiation of `?m` to be both as general as possible, to allow for subsequent development, and yet to allow unification to constrain `?m` to make the application valid. Our choice of the above rule for $Natrec_2$ seems to achieve this. We do not regard this choice as necessarily definitive, however: it merely represents our present view.

3.2.8 Lists

We may extend this analysis to the case of lists, where, as in the case of first-order deliverables, we find the richer structure of lists reflected in a richer collection of predicates and relations.

Firstly, if we work in the fibre $\mathbf{del}_2\ 1_{list\ a}$, then we obtain in exactly the same way as above, the following derived rule:

$$\frac{\Pi x{:}a.\ R \xrightarrow{\mathcal{F}} (cons\ x)^{\star}R \quad [1_{list\ a}]}{Listrec_2\ \mathcal{F}{:}(nil)^{\star}R \rightarrow R \quad [1_{list\ a}]}$$

where

$$(cons\ x)^{\star}R =_{\text{def}} \lambda l{:}list\ a.\ \lambda y{:}t.\ R\ (cons\ x\ l)\ y$$

and

$$(nil)^{\star}R =_{\text{def}} \lambda l{:}list\ a.\ \lambda y{:}t.\ R\ (nil\ a)\ y.$$

The LEGO term we obtain is:

```
Lego> listrec_del2;
value = [s,t|Type][R|Rel (list s) t]
        [family:{x:s}del2 (univPred|(list s)) R (cstarRel x R)]
        [c=[x:s]((family x)).1][C=[x:s]((family x)).2]
        ([l:list s][n:t]listrec n c l,
         [l|list s][h:true][n|t][nR:nstarRel R l n]
         listind ([m:list s]R m (listrec n c m)) ...)
type  = {s,t|Type}{R|Rel (list s) t}
        ({x:s}del2 (univPred|(list s)) R (cstarRel x R))->
         del2 (univPred|(list s)) (nstarRel R) R
```

where

```
Lego> nstarRel;
value = [s,t|Type][R:Rel (list s) t][_:list s][y:t]R (nil s) y
type  = {s,t|Type}(Rel (list s) t)->Rel (list s) t
Lego> cstarRel;
value = [s,t|Type][x:s][R:Rel (list s) t][l:list s][y:t]R (cons x l) y
type  = {s,t|Type}s->(Rel (list s) t)->Rel (list s) t
```

So we indeed find `[l:list s][n:t]listrec n c l` as the function component, and a suitable induction over lists,

```
... listind ([m:list s]R m (listrec n c m)) ...
```

as proof component.

But already in this rule we find something new: the outermost Π binding. That is to say, the rule has as its premise a *dependent* family of second-order deliverables. This phenomenon arises from the parameter type a of the lists in question. The rule is susceptible to the same criticisms as the rule for $Natrec_2$ above, but also the criticism that we have accorded a different status to the parameter type. In particular, it does not seem to be constrained by any predicate A we might impose on a. Our justification, as above, is essentially pragmatic. We have found this rule to be a useful construction, as in the example of minimum finding in the next chapter.

This is not to say, however, that we cannot obtain forms of this rule in which the input list is not further constrained. We may, for example, consider the predicate $\mathit{Listof}\ A$, for some predicate A on a. In fact, since we are now considering second-order deliverables, where we can take into account relations which depend on both the input variable and the result of some computation step, we may extend this predicate to a dependent version, which we call $depListof\ A$, defined as follows:

$$\begin{aligned} \mathit{depListof}\ \Phi\ (\mathit{nil}\ a) &=_{\mathrm{def}} \mathit{true} \\ \mathit{depListof}\ \Phi\ (\mathit{cons}\ x\ l) &=_{\mathrm{def}} (\Phi\ x\ l) \wedge (\mathit{depListof}\ \Phi\ l) \end{aligned}$$

Here Φ is some *relation* between values of the variable x varying over the parameter type a, and lists over a. An example is the predicate $Sorted$, for which we take $\Phi\ x\ l =_{\mathrm{def}} x \preceq l$, the relation that x is less than each element of the list l. This is discussed in more detail in the examples in the next chapter.

This introduces an extra hypothesis into the induction scheme we must consider. Suppose we wish to prove

$$\forall l{:}\mathit{list}\ a.\ (\mathit{depListof}\ \Phi\ l) \Longrightarrow R\ l\ (\mathit{listrec}\ n\ c\ l)$$

where $n{:}t$, $c{:}a \longrightarrow t \longrightarrow t$. A proof by induction generates the following hypotheses for each constructor.

base case $true \Longrightarrow R\ nil\ n$, which reduces logically to $R\ nil\ n$. As with the rules for $Natrec_2$, we shall fold this assumption into the rule as the initial relation in the second-order deliverable we eventually derive.

step case Formally, we obtain

$$\forall x{:}a.\ \forall l{:}list\ a.\ ((depListof\ \ \Phi\ l) \Rightarrow R\ l\ (listrec\ n\ c\ l)) \Longrightarrow \\ ((depListof\ \ \Phi\ (x :: l)) \Rightarrow R\ (x :: l)\ (c\ x\ l\ (listrec\ n\ c\ l))).$$

Two simplifications present themselves. The first is to replace the explicit mention of $(listrec\ n\ c\ l)$ by an additional universally quantified parameter y. The second is to observe that $depListof\ \ \Phi\ (cons\ x\ l) \Longrightarrow depListof\ \ \Phi\ l$. Combining these, we obtain as an induction hypothesis in the step case

$$\forall x{:}a.\ \forall l{:}list\ a.\ \forall y{:}t.\ (depListof\ \ \Phi\ (x :: l)) \Longrightarrow R\ l\ y \Longrightarrow R\ (x :: l)\ (c\ x\ l\ y).$$

In this form, we see the logical part of a second-order deliverable emerge.

We thus obtain the following derived rule, which yields a second-order deliverable with function component $listrec$ from a dependent family of second-order deliverables:

$$\frac{\mathcal{F}{:}\Pi x{:}a.\ R \rightarrow (cons\ x)^{\star}R \quad [(cons\ x)^{\star}depListof\ \ \Phi]}{depListrec_2\ \mathcal{F}{:}nil^{\star}R \rightarrow R \quad [depListof\ \ \Phi]}$$

of which we shall see examples in the next Chapter. In LEGO, it is represented by the following term:

```
Lego> depListrec_del2;
value = [A,B|Type][Phi:Rel A (list A)][R:Rel (list A) B]
        [F:{a:A}del2 (cstarPred a (depListof Phi)) R (cstarPred a R)]
        ([l:list A][b:B]listrec b ([a:A][k:list A][r:B]((F a)).1 k r) l,
```

```
         [l|list A][lhyp:depListof Phi l][n|B][basehyp:R (nil A) n]
         listind
            ([m:list A](depListof Phi m)->
               R m (listrec n ([a:A][k:list A][r:B]((F a)).1 k r) m))
         ...)
type  = {A,B|Type}{Phi:Rel A (list A)}{R:Rel (list A) B}
        ({a:A}del2 (cstarPred a (depListof Phi)) R (cstarRel a R))->
          del2 (depListof Phi) (nstarRel R) R
```