Deliverables: a categorical approach to program development in type theory

Rod Burstall and James McKinna[1]
Laboratory for the Foundations of Computer Science
University of Edinburgh[2]

August 15, 1992

1 Introduction

This paper outlines a method for constructing a program together with a proof of its correctness with respect to a given specification. The technology used is type theory, in fact an extended version of Calculus of Constructions Luo's ECC [22] as implemented in Pollack's 'Lego' system [24]; here *program* means a primitive recursive description of a function using simply typed lambda calculus with higher order functions and data types such as numbers and lists.

Given a precondition and a postcondition we consider pairs consisting of

- a program
- a proof that the program satisfies the postcondition given the precondition.

We call such a pair together with the pre- and postconditions a *deliverable*, since it is what a software house should ideally deliver to its client instead of just a program.

Now we observe that various operations, composition, pairing, iteration and so on, can be used to combine deliverables, and that these operations can be implemented in ECC. Consider for example composition. If (f, p) is a deliverable from *pre* to *post* and $(f\prime, p\prime)$ is a deliverable from *post* to *post′* then their composition is a deliverable from *pre* to *post′* thus

- $f \circ f\prime$, the composition of the functions
- a proof using p and $p\prime$ that $f \circ f\prime$ satisfies *post′* given *pre*.

[1]The authors gratefully acknowledge the support of the EC Logical Frameworks BRA and the SERC

[2]J.C.M.B., King's Buildings, Mayfield Rd., Edinburgh EH9 3JZ, UK;
e-mail `rb@dcs.ed.ac.uk, jhm@dcs.ed.ac.uk`

In fact the combining operations (excluding iteration) are exactly those of a cartesian closed category whose objects are the pre- and postconditions and whose arrows are deliverables. This is comforting since it assures us that we have a complete set of combinators. They enable us to build a complicated program together with its correctness proof. Since the resulting expression of the calculus denotes a pair we can extract the program trivially by evaluating the first element of the pair.

We have used the Lego system to define these combinators and use them to create the deliverables for several programs as examples.

However in many cases a precondition and a postcondition do not satisfactorily specify a desired program. For example when developing a sorting program the precondition might ask for an arbitrary list and the postcondition demand that the result be a sorted list. But we would get little thanks for a program which always returned the nil list, even though this is sorted. We need to specify that the output list should also be a permutation of the input one.

If we try to replace the pre- and postconditions by a single statement specifying a relation between input and output there do not seem to be convenient combination operators; so we discard that approach. Thinking syntactically, we could allow the pre- and postconditions to share a free variable. For example using a free variable l for lists

$$pre(x) = x = l$$

$$post(y) = isPermutation(y, l)$$

This ties the pre- and postconditions together, but such free variables should have a definite scope.

We can think, more semantically, that l satisfies some base condition (it must be a list) and the pre- and post conditions are relations whose first argument is l. Thus we can talk of pre- and post conditions *relative to* l. Now a deliverable relative to l is a program which takes the precondition to the postcondition for all l. We will refer to these as *second order* specifications and deliverables in contrast to the original *first order* ones.

How should we represent specifications and deliverables in type theory? We choose the following:

A first order specification is a predicate over a type, thus

- a type, s;
- a function, S, from s to $Prop$ (the type of propositions);

and a first order deliverable from specification (s, S) to specification (t, T) is a program paired with a proof thus

- a function, $f : s \to t$;
- a proof, $\phi : \forall x : s.Sx \to T(fx)$ where $\forall$ is the $\prod$ quantifier on propositions.

Now we can generalise these to work relative to a first order specification (u, U), which corresponds to the free variable l in our example. Call u the base type; now the second order specification becomes a relation over the base type and another type, and the second order deliverable must respect the pre- and postrelations. We leave the details to the body of the paper.

We might try to characterise the first order deliverables as a cartesian closed category whose objects are specifications and whose arrows are deliverables; this is almost correct, but we have to modify the CCC to a *semi*-cartesian closed category, a mild generalisation due to Hayashi [12]. The second order deliverables are then characterised using a fibration whose fibres are categories of first order deliverables. In fact this structure constitutes a model of type theory.

The paper includes some illustrative examples of program development using deliverables and brief mention of others, of which the most substantial is an algorithm for the Chinese Remainder theorem.

We also discuss the relation to earlier work on extracting programs from proofs, particularly the work of Martin-Löf as developed by Nördström, Petersson and Smith [29]. They define a translation from a type theory, say S, to an underlying type theory U such that the types in S correspond to our specifications and the unary terms in S correspond to our deliverables. A somewhat similar approach is taken by Paulin-Mohring in her work on program extraction using the realisability interpretation [30].

The virtue of our method is that it can quite easily be coded up in an existing system, and there is no difficulty about extracting the program from the proof, since they are built together but as separate components of a pair; normalising the first component gives us the program. We have mentioned operations for combining deliverables, and this might suggest a bottom-up approach; but we can just as well start with a specification of the desired program and derive the corresponding deliverable top-down by refinement in Lego.

The weakness of our method is that we have to use a combinatory notation which is less perspicuous than the usual λ notation of type theory. Since deliverables form a model of type theory, this might possibly be rectified by extending ECC with an extra kind whose types would be specifications and whose terms would be deliverables, thus following the insight of Martin-Löf; but we have not yet worked out this approach.

The basic approach was proposed in an unpublished talk by Burstall at the Båstad Workshop on type theory in 1988. A brief account by the present authors appeared as [4], and a full treatment of work so far is in McKinna's thesis [25].

2 Review of ECC and a simple example

Luo's Extended Calculus of Constructions, ECC [21, 22], is a rich type theory containing Coquand and Huet's Calculus of Constructions [6, 7] as a subsystem, together with strong Σ-types and a cumulative hierarchy of predicative universes, much as in the systems considered by Martin-Löf and his collaborators [26, 27, 29]. In Martin-Löf systems, *all* types may be read as propositions, and in Coquand and Huet's original system, all propositions may be read as types. ECC avoids this blurring of distinctions, giving us access to full intuitionistic higher-order logic at a propositional level, together with a predicative environment for computation and abstract mathematics. It is precisely this ability to distinguish propositional from computational information within a single framework which underlies our approach to program development.

ECC is built out of a calculus of terms with the following official syntax

$$\begin{array}{rcl} T & ::= & \kappa \mid V \mid \\ & & \Pi V{:}T.T \mid \lambda V{:}T.T \mid TT \mid \\ & & \Sigma V{:}T.T \mid \mathbf{pair}_T(T,T) \mid \pi_1(T) \mid \pi_2(T) \end{array}$$

where V ranges over some infinite collection of variables, and κ ranges over *Prop* and Type_i $(i \in \omega)$, the so-called *kinds* of ECC. *Prop* is an impredicative universe, as in Coquand and Huet's original systems [6], intended to contain propositions, while the Type_i are predicative universes much like a set-theoretic hierarchy (and very similar to the $\mathbf{U}_i$ of some versions of Martin-Löf's theories [29, 27]). Substitution for free occurrences of variables is defined in the usual way. Terms are identified up to renaming of bound variables. The basic conversion relation $\simeq_\beta$ is defined on all terms, and as usual is the congruence closure of the familiar reductions:

(β) $(\lambda x{:}A.M)N \rhd M[N/x]$, and

(σ) $\pi_i(\mathbf{pair}_T(M_1, M_2)) \rhd M_i (i = 1, 2)$.

LEGO is Pollack's implementation of a typechecker and refinement proof system for ECC and a number of related systems, based on earlier ideas of Huet, de Bruijn and others [3, 6].

The LEGO syntax for terms in the language of ECC is given by

$$\begin{array}{l} \mathtt{T} ::= \mathtt{Prop} \mid \mathtt{Type}(i)\ (i \in \omega) \mid \mathtt{V} \\ \quad \{\mathtt{V{:}T}\}\mathtt{T} \mid [\mathtt{V{:}T}]\mathtt{T} \mid \mathtt{TT} \mid \\ \quad \mathtt{<V{:}T>T} \mid \mathtt{(T,T)} \mid \mathtt{T.1} \mid \mathtt{T.2} \end{array}$$

where the correspondence with the official syntax is given in Table 1. As usual, the non-dependent Π-types are denoted by an infix $\rightarrow$.

The syntax has recently been extended to provide arbitrary extensions to the conversion relation. As an example, we may add a type of natural numbers, by declaring appropriate constants in an initial context

$$\begin{array}{rcl} \Gamma_{nat} & = & nat{:}\mathit{Type}_0, 0{:}nat, S{:}nat \rightarrow nat, \\ & & natrecd{:}\Pi C{:}nat \rightarrow \mathit{Type}.\ (C0) \rightarrow (\Pi k{:}nat.\ (Ck) \rightarrow (C(Sk))) \rightarrow \Pi n{:}nat.\ Cn \end{array}$$

`{V:T}T`	corresponds to	$\Pi x\colon T.\ T$,
`[V:T]T`	to	$\lambda x{:}T.\ T$,
`<V:T>T`	to	$\Sigma V{:}T.T$,
`(T,T)`	to	$\mathbf{pair}_{\Sigma x:A.B}(T,T)$,
`T.1`	to	$\pi_1(T)$,
and `T.2`	to	$\pi_2(T)$.

Table 1: Comparison between syntax of LEGO and ECC

together with reductions

$$\begin{array}{rcl} \mathit{natrecd}\ C\ z\ s\ 0 & \rhd & z \\ \mathit{natrecd}\ C\ z\ s\ (Sk) & \rhd & s\ k\ (\mathit{natrecd}\ C\ z\ s\ k) \end{array}$$

for C, z, s, k of the appropriate types.

In this context, we may derive the following term

$$\Gamma_{nat}, n{:}\mathit{nat} \vdash \forall\Phi{:}\mathit{nat} \to \mathit{Prop}.\Phi(0) \Rightarrow (\forall k{:}\mathit{nat}.\Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(n) : \mathit{Prop}$$

which represents impredicatively the (informal) proposition that n is an *even* natural number. That is, we define even numbers to be those n which satisfy *all* predicates satisfied by 0 and closed under successor of successor (the impredicativity, of course, lies in the fact that "evenness" is just such a predicate). Moreover, in this representation, the judgment

$$\begin{array}{lll} \Gamma_{nat} & \vdash & \lambda\Phi{:}\mathit{nat} \to \mathit{Prop}. \\ & & \quad \lambda z{:}\Phi(0)\,. \\ & & \quad\quad \lambda s{:}\forall k{:}\mathit{nat}.\,\Phi(k) \Rightarrow \Phi(S(Sk))\,. \\ & & \quad\quad\quad z \\ & & : \ \forall\Phi{:}\mathit{nat} \to \mathit{Prop}.\Phi(0) \Rightarrow (\forall k{:}\mathit{nat}.\Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(0) \end{array}$$

represents a proof that 0 is even.

Example In the same way, one could define a predicate representing "oddness",

$$\mathit{Odd} =_{\mathrm{def}} \lambda n{:}\mathit{nat}.\ \forall\Phi{:}\mathit{nat} \to \mathit{Prop}.\Phi(S0) \Rightarrow (\forall k{:}\mathit{nat}.\Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(n)$$

being one such representation. One might then show that the successor function S transforms even numbers into odds, and odds to evens, yielding terms sEO of type $\forall n{:}\mathit{nat}.\ \mathit{Even}(n) \to \mathit{Odd}(Sn)$ and sOE of type $\forall n{:}\mathit{nat}.\ \mathit{Odd}(n) \to \mathit{Even}(Sn)$ (the patient reader may care to experiment with the system by doing just such a thing).

What is the behaviour of the composite function $x \mapsto x + 2$? We may use the proofs sEO and sOE, together with *modus ponens*, to derive a term $ssEE$ of type $\forall n{:}nat.\ Even(n) \to Even(n+2)$.

Now the Σ-types of the calculus allow us to pair together these fragments of code with their associated proof terms, for example

$$(S, sEO) : \Sigma f{:}nat \to nat.\ \forall n{:}nat.\ Even(n) \to Odd(fn)$$

and we may see the *pair* $(\lambda n{:}nat.\ n+2,\ ssEE)$ arising as a composite construction from the *pairs* (S, sEO) and (S, sOE).

The pair $(\lambda n{:}nat.\ n+2,\ ssEE)$ of type

$$\Sigma f{:}nat \to nat.\ \forall n{:}nat.\ Even(n) \to Even(fn)$$

is a very simple instance of the general idea below of a *first-order deliverable*, in this case from $(nat, Even)$ to $(nat, Even)$.

3 Definition of first-order deliverables

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

Definition 3.1 *specification*
A *specification* is a pair of terms s, S such that $\Gamma \vdash s : Type$, and $\Gamma, x{:}s \vdash Sx : Prop$.

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 $SPEC_1 =_{\mathrm{def}} \Sigma s{:}Type.s \to Prop$, so that we may consider operations which construct specifications entirely within the framework of ECC (*cf.* the account of specifications and refinements in [23]).

We will consider a category whose objects are the specifications. Specifications defined by *logically* equivalent predicates in general define distinct objects. The appropriate notion of morphism between specifications consists of a pair (f, ϕ), where f is a function between the underlying types, and ϕ is a proof that f respects the predicates. We call these morphisms *first-order deliverables*.

Definition 3.2 *first-order deliverable*
Given specifications $\mathcal{S} = (s, S), \mathcal{T} = (t, T)$, a *first-order deliverable* is a term $\mathcal{F}$ such that

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

The motivation for such a definition goes right back to Hoare's original paper on axiomatic semantics [14], 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 lid (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 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.

In terms of LEGO, we define a predicate Del_1 and a type constructor $\mathbf{del}_1$ which describe first-order deliverables within ECC (compare Definition 3.2 above):

```
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
```

(This is the output from the typechecker after giving the definitions). We have exploited LEGO's type inference mechanism [33] to enable us to suppress the argument types[3] in the predicate `Del1` and the type `del1`.

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

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

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

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

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

[3] 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.

operations which produce such function-proof pairs within the calculus. This gives us the possibility of developing a structure on these morphisms.

Definition 3.3 *equality of specifications*
Given specifications $(s, S), (t, T)$, we say $(s, S) = (t, T)$ if

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

Definition 3.4 *equality of deliverables*
Given

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

we say $(f, \phi) = (g, \psi)$ if

$$\lambda x{:}s.\ fx \simeq_{\beta\delta} \lambda x{:}s.\ gx$$

and

$$\lambda x{:}s.\ \lambda h{:}S\,x.\ \phi x h \simeq_{\beta\delta} \lambda x{:}s.\ \lambda h{:}S\,x.\ \psi x h.$$

This definition of equality of deliverables seems to be the minimal extension of the basic conversion relation which ensures good categorical properties.

3.1 Semi-structure in categories

The use of cartesian closed categories to give models of the simply-typed λ-calculus is by now familiar in computer science [8, 20, 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). 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 [12]. 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. A detailed treatment is given in McKinna's thesis [25].

Definition 3.5 *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

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

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". The beauty of this idea lies in the following lemma.

Lemma 3.1 (Hayashi) *Suppose F, G above are in fact functors. Then a semi-adjunction between F and G yields 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[4]. In particular, in a category $\mathbf{C}$, we may define the notions of *semi-terminal object*,*semi-product* and *semi-exponential* exactly by analogy with the usual cartesian closed structure. 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 *Semi-cccs are sound and complete for interpretations of the $\lambda\beta$-calculus.*

Proposition 3.2 *Semi-cccs can be presented algebraically.*

In [25], the second author used LEGO to prove the following theorem:

Theorem 3.1 ***del***$_1$ *is semi-cartesian closed.*

We sketch some of the ideas in the discussion below.

3.2 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

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

```
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

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

We may now verify that these definitions do indeed yield the structure of a category on $\mathbf{del}_1$[5].

3.3 Semi-Terminal object

$$Unit =_{\text{def}} (unit, \lambda u : unit.u = ()^6) : SPEC_1$$

defines a trivial specification, where we postulate the existence of a unit type in the same way as other inductive types. We then obtain, for any specification (s, S) the deliverable

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

3.4 Binary semi-products

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

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

That we indeed have a semi-product structure now follows straightforwardly from this definition.

3.5 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 on values, and implication introduction and elimination at the level of proofs.

$$\mathcal{S} \Longrightarrow \mathcal{T} =_{\text{def}} (s \to t, \lambda f{:}s \to t.\ \mathrm{Del}_1\ S\ T\ f)$$

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

[6]Here, () is the unique term of type $unit$. We write `void` in LEGO.

3.6 Trivial deliverables

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 [14] or Dijkstra's predicate transformer for assignment[9]. Namely, for

$$f : s \longrightarrow t, P : t \longrightarrow \mathit{Prop}$$

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.

3.7 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:

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

In fact, this rule is subsumed by composition, since any logical entailment gives rise to a deliverable whose function component is the identity.

3.8 Pointwise construction

A basic combinator in the theory of deliverables constructs a function-proof pair from a function which returns value-proof pairs[7]. Mendler, in his thesis [28], 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,

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

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

[7] This just corresponds to Howard's observation that, given a strong interpretation of the existential quantifier as Σ-type, the axiom of choice becomes constructively valid [15].

3.9 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. 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[8] 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.

3.9.1 Natural numbers

Recall from Section 2 that we may postulate the existence of a type of natural numbers, with two constructors, zero and successor. This yields the well-formed context

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

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

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

together with reduction rules defining the δ-redexes[9] (in some context where C, z, s, n have the appropriate types):

- $natrecd\ C\ z\ s\ 0 \rhd z$;
- $natrecd\ C\ z\ s\ (n+1) \rhd s\ n\ (natrecd\ C\ z\ s\ n)$.

This is precisely expressed in LEGO as follows:

```
[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)].
```

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

[9] *cf.* Martin-Löf type theory, or Gödel's earlier system **T** of functionals [38]. We essentially use this language of primitive recursion in all finite types as our programming language.

This yields a derived iterator and primitive recursor

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

and

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

where[10]

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

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:

$$\overline{\overline{Zero =_{\text{def}} (\lambda u{:}unit.\ 0,\ \lambda u{:}unit.\ \lambda h{:}Unit\ u.\ \top^{11}){:}Unit \longrightarrow Nat}}$$

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

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

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 the recursor *Natrec* analogously, 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. For example, here is the significant structure of *Natiter*, with the uninformative details of the proof component suppressed:

```
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
```

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 Section 2;

step case $(nat, Even) \xrightarrow{(s,S)} (nat, Even)$. We simply use the first-order deliverable which we constructed by composition at the end of Section 2, namely $(\lambda n{:}nat.\; n+2,\; ssEE)$.

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

3.9.2 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 $list{:}Type_0 \longrightarrow Type_0$, 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 $\mathit{List}\,(a, A) =_{\mathrm{def}} (\mathit{list}\ a,\ \mathit{Listof}\ A)$, where

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

defines $\mathit{Listof}\ A$ by primitive recursion.

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

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

and

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

with function component,

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

and proof component

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

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

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

and analogously a recursor.

4 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. Recalling the example in the introduction, 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. This gives us a categorical explanation of the idea that the pre- and postconditions are linked by having a free variahle in common, that is they are over the same context.

4.1 A thought experiment

Suppose we are given some 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

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

This last is logically equivalent to

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

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 4.2, and 4.3 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.

Definition 4.1 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 $\phi{:}\forall x{:}s.\, \forall y{:}t.\, Q(x,y) \Longrightarrow R(x,(f\,x\,y))$.

The composition of two such arrows

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

is definable as

$$(\lambda x, w{:}s, r.\; h\,x\,(g\,x\,w),\quad \lambda x, w{:}s, r.\; \lambda p{:}P(x,w).\; \psi\,x\,(g\,x\,w)\,(\phi\,x\,w\,p)).$$

We have now established a definition which respects the symmetry of source and target in our previous analysis of the decomposition of the 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 $r =_{\mathrm{def}} \mathit{unit}$, $P(x,w) =_{\mathrm{def}} \mathit{true}$, and $f_{old} =_{\mathrm{def}} \lambda x{:}s.\, f_{new}\,x\,()$.

4.2 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 the definition of a "second-order" deliverable.

Definition 4.2 *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

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

Definition 4.3 *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 4.3 embodies the idea that, for each $x{:}s$ such that Sx holds, $(fx, \phi x)$ is a first-order deliverable from Qx to Rx, where $\mathcal{F} =_{\mathrm{def}} (f,\phi)$. This suggests 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 $\mathcal{S} =_{\mathrm{def}} (s,S)$, 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 above, the details of which are left to the reader. Then we can indeed define identities and composition, given by

Identities The identity morphism from (t, Q) to itself over (s, S) is given by

$$(\lambda x{:}s.\ \lambda y{:}t.\ y,\ \lambda x{:}s.\ \lambda h{:}S\,x.\ \lambda y{:}t.\ \lambda q{:}Q\,x\,y.\ q)$$

Composition This is defined much as in the thought experiment above (4.1): the composition of

$$P(x, w) \xrightarrow{(f, \phi)} Q(x, y) \xrightarrow{(g, \psi)} R(x, z)$$

over (s, S) is definable as

$$(\lambda x{:}s.\ \lambda w{:}r.\ g\,x\,(f\,x\,w),\quad \lambda x{:}s.\ \lambda h{:}S\,x.\ \lambda w{:}r.\ \lambda p{:}P\,x\,w).\ \psi\,x\,h\,(f\,x\,w)\ (\phi\,x\,h\,w\,p)).$$

This gives us a category $\mathbf{del}_2\mathcal{S}$.

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

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

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.

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

As in Section 3, we work relative to some context Γ. We have seen how a second-order deliverable (f, ϕ) 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 4.2, and 4.3 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 ϕ may do so.

Given these conditions, we may lift the structure in $\mathbf{del}_1$, by observing that the various constructions of Section 3 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 4.1 *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 in McKinna's thesis [25].

Semi-terminal object This is given by the relativised specification

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

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

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

Semi-products Suppose we are given two relativised specifications $(t,Q),(u,R)$. Then we may form the relativised specification

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

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

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

where $<f,g>$ and $<\phi,\psi>$ are appropriate pairing operations on values and proofs respectively. Projections are similarly straightforward to define.

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

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

If $\Gamma\vdash f : s\longrightarrow t\longrightarrow u$, and $\Gamma\vdash\phi : \mathrm{Del}_2\,S\,Q\,R\,f$, then $\Gamma, x{:}s\vdash \phi x : R^Q\,fx$. 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 $\mathcal{F} =_{\rm def} (f,\phi)$, then $\Lambda(\mathcal{F}) =_{\rm def} (\hat{f},\hat{\phi})$, where $\hat{f}$ and $\hat{\phi}$ are appropriate currying operations on values and proofs respectively. 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 refinement in LEGO. Likewise, the proofs that these data meet Hayashi's conditions for a semi-exponential are best dealt with by refinement.

■

4.4 $\mathbf{del}_2$: 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 4.2 *$\mathbf{del}_2$ is an indexed category [19, 1] over $\mathbf{del}_1$, whose fibres are semi-cccs, with semi-cc structure strictly preserved by reindexing along arrows in $\mathbf{del}_1$.*

4.5 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, which we give below, 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 4.4 *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 $\mathcal{Q} =_{\text{def}} (u, Q)$ with respect to $\mathcal{T}$, let

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

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

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

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

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

Lemma 4.1 *$(k, K)^{\star}\mathcal{Q}$ is a relativised specification with respect to $\mathcal{S}$. Moreover, $(k, K)^{\star}(f, \phi)$ is a second-order deliverable from $(k, K)^{\star}\mathcal{Q}$ to $(k, K)^{\star}\mathcal{R}$.*

Lemma 4.2 *$(k, K)^{\star}$ preserves identities and composition.*

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 $\mathcal{H}^\star; \mathcal{K}^\star \cong (\mathcal{K}; \mathcal{H})^\star$. In fact, more is true. We have the following:

Lemma 4.3 *The reindexing is* strict, *in the sense that*

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

We now turn to the remainder of Theorem 4.2, 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 state only the case of exponentials, the cases of products and terminal object being exactly similar, and rather easier.

Lemma 4.4 *In the notation of Theorem 4.1 above, with* $\mathcal{V} \xrightarrow{\mathcal{K}} \mathcal{S}$, *we have*

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

Proposition 4.1 *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 $\Lambda(\mathcal{K}^\star\mathcal{F}) \equiv \mathcal{K}^\star\Lambda(\mathcal{F})$.

4.6 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 we have considered 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 Section 3.9. 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[12]:

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

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

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

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.

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

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 captures 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

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

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

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

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.

4.7 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

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

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 a list.

We can obtain forms of this rule in which the input list is further constrained. We may, for example, consider the predicate $Listof\ A$, 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:

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

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 $Sorted$, where we take $\Phi\ x\ l$

to be the relation that x is less than each element of the list l. This crops up in the example of an insert sort in the next section.

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 $\mathit{true} \Longrightarrow R\ \mathit{nil}\ n$, which reduces logically to $R\ \mathit{nil}\ n$. 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

$$\begin{aligned}\forall x{:}a.\, \forall l{:}\mathit{list}\, a.\ ((\mathit{depListof}\ \ \Phi\ l) \Rightarrow R\ l\ (\mathit{listrec}\ n\ c\ l)) \Longrightarrow& \\ ((\mathit{depListof}\ \ \Phi\ (x :: l)) \Rightarrow R\ (x :: l)\ (c\ x\ l\ (\mathit{listrec}\ n\ c\ l))).&\end{aligned}$$

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

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

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 $\mathit{listrec}$ from a dependent family of second-order deliverables:

$$\frac{\mathcal{F}{:}\Pi x{:}a.\ R\ \rightharpoonup\ (\mathit{cons}\ x)^{\star}R\quad [(\mathit{cons}\ x)^{\star}\mathit{depListof}\ \ \Phi]}{\mathit{depListrec}_2\ \mathcal{F}{:}\mathit{nil}^{\star}R\ \rightharpoonup\ R\quad [\mathit{depListof}\ \ \Phi]}$$

5 Examples

In his thesis [25], the second author considered a number of example developments using the methodology outlined above, culminating in a machine-checked proof of the Chinese Remainder Theorem. This last example is too long to include here, although it perhaps best exhibits some of the strengths of our approach. We hope to discuss it in a forthcoming paper.

To illustrate the discussion of the previous sections, we give two examples of the use of deliverables in small-scale program development. The use of deliverables may seem heavy-handed and even counter-productive for these smaller examples. The discussion is taken from [25], where detailed accounts in LEGO of the derivations may be found.

5.1 Finding the minimum of a list

An example which is treated several times in the literature [30, 36]. In our case, it involves a somewhat unnatural reperesentation, which makes non-trivial use of the semi-cartesian closed structure in the fibres $\mathbf{del}_2$.

5.1.1 The mathematical specification

Suppose R is a decidable total order on some type α, 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* $\lambda a, b{:}\alpha.\ R\,a\,b = tt$, denoted $\leq_R$. Then we may specify the minimum of a list as follows:

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

We abbreviate the second conjunct to $m \preceq_R l$. We may easily express a solution to this specification in SML as follows (:: adds an element to a list):

```
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 [36][13]

```
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:

$$\forall l{:}list\,\alpha.\ \exists f{:}\alpha \to \alpha.\ \forall a{:}\alpha.\ f a \in a :: l \wedge f a \preceq_R a :: l;$$

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

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

5.1.2 The correctness proof

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

Base case

$$\exists f{:}\alpha \to \alpha.\ \forall a{:}\alpha.\ fa \in a :: nil \wedge fa \preceq_R a :: nil.$$

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

Step case Suppose

$$\exists f{:}\alpha \to \alpha.\ \forall a{:}\alpha.\ fa \in a :: k \wedge fa \preceq_R a :: k.$$

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

5.1.3 The development in terms of deliverables

We give an outline of the proof in the style of a derivation tree, indicating the combinators used to resolve significant subgoals.

Let

$$MinAuxSpec =_{\text{def}} \lambda l{:}list\alpha.\ \lambda f{:}\alpha \longrightarrow \alpha.\ \forall a{:}\alpha.\ fa \in a :: l \wedge fa \preceq_R a :: l.$$

Abbreviating the trivial specifications $\lambda l{:}list\alpha.\ true$ and $\lambda l{:}list\alpha.\ \lambda u{:}unit.\ true$ to 1 (since they are terminal objects, respectively in $\mathbf{del}_1$, and $\mathbf{del}_2$ $(list\alpha,\ 1)$), we seek

$$1 \xrightarrow{minelemaux} MinAuxSpec \quad [1]$$

We use composition to enable us to exploit our recursion combinator for second-order deliverables over lists of Section 4.6, to package up the proof by induction:

$$\frac{1 \xrightarrow{base} ?1 \quad [1] \quad ?1 \xrightarrow{step} MinAuxSpec \quad [1]}{1 \xrightarrow{minelemaux} MinAuxSpec \quad [1]}(compose)$$

and

$$\frac{\Pi a{:}\alpha.\ MinAuxSpec \xrightarrow{\mathcal{F}_{step}} (cons\ a)^{\star}MinAuxSpec \quad [1_{list\,\alpha}]}{?1 \xrightarrow{step} MinAuxSpec \quad [1]}(Listrec_2)$$

which instantiates subgoal ?1 with the specification $nil^{\star}MinAuxSpec$.

We resolve the base case of our induction using a pointwise construction, and thereafter exactly the informal reasoning above. This is sufficient to allow us to completely close this branch of the derivation.

$$\frac{\begin{array}{l} value = \lambda a{:}\alpha.\ a \\ proof : a \in [a] \wedge a \preceq_R [a] \end{array}}{1 \xrightarrow{base} nil^{\star}MinAuxSpec \quad [1]}(pointwise)$$

This leaves the step case. We extend the context with a free variable b of type α, and then conclude with another pointwise construction. The proof follows the informal argument above.

$$\frac{\begin{array}{l} value = \lambda a{:}\alpha.\ min\ a\ (f\ b) \\ proof : \ldots \end{array}}{MinAuxSpec \xrightarrow{\mathcal{F}_{step}\ b} (cons\ b)^{\star}MinAuxSpec \quad [1]}(pointwise)$$

5.2 Insert sort

We consider 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.

5.2.1 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 an impredicative definition of the permutation relation $\equiv$, which we do not discuss here. The predicate *Sorted* was 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*.

5.2.2 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 explain recursive deliverables:

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

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

a consequence of which is the following induction principle for sorted lists, which is just the proof component of an instance of the $depListrec_2$ combinator we considered above:

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

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

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) $\mathit{Sorted}(\mathit{nil}),\ \mathit{nil} \sim \mathit{nil}$

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

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

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

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

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

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

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

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

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

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

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

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

Lemma 5.1 *Suppose $a, b, c \in \alpha$, and $m, n \in \mathit{list}\ \alpha$ such that:*

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

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

5.2.3 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 some additional propositional reasoning.

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

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

to

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

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. Rather than attempt a detailed exposition here, we merely remark on the rather unsatisfactory nature of this step in our derivation. The details are in [25].

We outline the significant refinement steps in the derivation.

We seek, as a top-level goal

$$1 \xrightarrow{insertsort} SortSpec \quad [1]$$

where $SortSpec\ l\ m =_{\mathrm{def}} Sorted(m) \wedge\ l \sim m$.

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

$$\frac{1 \xrightarrow{base} ?1 \quad [1] \quad ?1 \xrightarrow{step} SortSpec \quad [1]}{1 \xrightarrow{insertsort} SortSpec \quad [1]}(compose)$$

$$\frac{\Pi a{:}\alpha.\ SortSpec \xrightarrow{\mathcal{F}_{step}} (cons\ a)^{\star}SortSpec \quad [1_{list\,\alpha}]}{?1 \xrightarrow{step} SortSpec \quad [1]}(Listrec_2)$$

instantiates subgoal ?1 with the specification $nil^{\star}SortSpec$.

The base case of this recursion/induction is resolved by a pointwise construction. Moreover, we can prove that any permutation of the *nil* list must be equal to *nil*, and the *nil* list is trivially sorted. Hence we obtain the correct instantiation

$$\frac{\begin{array}{l} value = nil \\ proof\ : Sorted(nil) \wedge nil \sim nil \end{array}}{1 \xrightarrow{base} nil^{\star} SortSpec \quad [1]} (pointwise)$$

The step case is a little more complicated. We first introduce the parameter $a : \alpha$ 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 Section 3. We then fold this function into a relativised specification, with which we pursue the inner recursion/induction:

$$\frac{SortSpec \rightarrow SortSpec' \quad [1] \qquad \dfrac{function = \lambda p{:}\alpha \times list\alpha.\ \pi_1(p) :: \pi_2(p)}{SortSpec' \rightarrow (cons\ a)^{\star} SortSpec \quad [1]} (functional)}{SortSpec \xrightarrow{\mathcal{F}_{step}\ a} cons\ a^{\star} SortSpec \quad [1]} (compose)$$

where $SortSpec' =_{\text{def}} (\lambda p{:}\alpha \times list\alpha.\ \pi_1(p) :: \pi_2(p))^{\star}(cons\ a)^{\star} SortSpec$

Eliding the details, we massage the goal $SortSpec \rightarrow SortSpec' \quad [1]$ into a form in which we may exploit the principle of Sorted List Induction. It turns out that the target specification remains as $SortSpec'$:

$$\frac{\dfrac{1 \rightarrow SortSpec' \quad [Sorted]}{\cdots}}{SortSpec \rightarrow SortSpec' \quad [1]}$$

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.

$$\frac{1 \xrightarrow{base'} ? \quad [Sorted] \quad ? \xrightarrow{step'} SortSpec \quad [Sorted]}{1 \rightarrow SortSpec \quad [Sorted]} (compose)$$

$$\frac{\Pi b{:}\alpha.\ SortSpec' \xrightarrow{\mathcal{G}_{step'}} (cons\ b)^{\star} SortSpec' \quad [(cons\ b)^{\star} Sorted]}{? \rightarrow SortSpec' \quad [Sorted]} (depListrec_2)$$

The proof concludes by considering a pointwise construction in the base case of the induction,

$$\frac{\begin{array}{l} value = (a, nil) \\ proof : Sorted([a]) \wedge [a] \sim [a] \end{array}}{1 \xrightarrow{base'} nil^{\star}(SortSpec') \quad [Sorted]}(pointwise)$$

and then an account of the verification of the `swapcons` function in the *step*$'$ case. We omit the details, which exploit the informal argument of Lemma 5.1.

6 A set-theoretic model

In the foregoing discussion, our emphasis lay on using and constructing deliverables in the context of a particular type theory and its machine implementation. However, the idea of deliverables need not be restricted to this particular seeting. We observed that one could model deliverables in the framework of elementary set theory. The thesis [25] gives a detailed categorical account of this model, showing how it supports the interpretation of a subtheory of ECC. Indeed, the model is parametric in the choice of an underlying *topos*(categorical model of set theory). We sketch the main ideas. We refer the reader to [25] for a complete account.

In this interpretation, a (closed) type is given by a pair of sets (A, A'), with $A' \subseteq A$. In particular, the type *Prop* of propositions, is given by the pair (Ω, Ω), where Ω is the subobject classifier. A morphism between such objects is given by the obvious abstract notion of first-order deliverable, i.e. an arrow on the underlying sets which respects the distinguished subsets. This data gives us an interpretation of contexts and morphisms between them.

Types in a given context $\Gamma = (X_0, X_1)$ are given by the relativised specifications, i.e. pairs of sets (A, R), where now $R \subseteq X_0 \times A$. Without going into further detail, this structure gives rise to a *fibration*, whose fibres essentially consist of the second-order deliverables described above. Moreover, this fibration has sums, products and a small subfibration and thus supports an interpretation of the "$\mathit{Prop}{:}\mathit{Type}_0{:}\mathit{Type}_1$" fragment of ECC. In the presence of set-theoretic universes, given for example by inaccesible cardinals, we may model the whole of ECC.

7 Related work

Two approaches to a logical account of formal program development are well known. The first relies on annotating programs with logical formulae, and expressing the correctness of a program in terms of logical deductions. This has its origins in the work of Floyd, Hoare and others in the 'Sixties [14, for example].

The idea of a deliverable clearly has echoes of this idea, but brought into the functional setting. It also avoids the defect of the Floyd/Hoare style in having object language and meta-logical variables on the same footing as object variables of our chosen type theory.

The second approach, based on various intuitionistic type theories, has been to develop constructive proofs, and use realisability techniques to extract algorithmic information. Both Martin-Löf type theory [5, 29] and the Calculus of Constructions [6, 30, 31] have been used in this style. Our work uses ideas from both these schools. Most influential has been the theory of subsets in Martin-Löf type theory. This has been given an eloquent treatment in [29, Chapter 18], to which the reader is referred for a detailed discussion. The central problem in using constructive proofs as a programming discipline is that proofs contain redundant information. Dependent types allow us to express logical predicates as types, but do not permit the representation of any more recursive functions. For the Calculus of Constructions, this has a precise statement in the result of Berardi and Mohring [30, 2]:

Theorem *$\mathcal{CC}$ is conservative over F_ω.*

Consequently, $\mathcal{CC}$ can represent, in the standard (Church) representation of functions on inductive types, no more functions than F_ω. The proof is based on a syntactic mapping, the so-called Berardi-Mohring projection. Mohring used this mapping as an extraction function, which, coupled with the associated realisability predicate, allows a powerful and flexible approach to program development from proofs. Under this interpretation, an arbitrary type is interpreted as a type, together with a predicate (the realisability predicate) defined over it.

The approach taken in [29] is to separate computationally relevant proofs from the purely logical, via a translation of the judgments of the basic formal system into multiple judgments. This translation permits the formation of a "subset type" $\{x \in A | B(x)\}$, for which a reasonable elimination rule may be given. An attempt to equip the basic theory with such a type (which may be used to precisely hide the information of the precise nature of a proof that predicate $B(x)$ holds) yields very unsatisfactory results [34, 35, 29].

Given the basic theory of types and terms in Martin-Löf type theory, the first step is to extend the theory with a notion of *proposition* and a judgment P **true** for propositions. This is very straightforward, using propositions as types. A proposition is just a type in the basic theory. A proposition is true if there is some element inhabiting it, again in the basic theory. This seemingly innocent proof-irrelevance gives the subset theory its power.

The subset theory now interprets the basic judgment

$$A \ \textbf{set}$$

of Martin-Löf type theory as two judgments in the underlying theory, prescribing

- a set A' in the basic theory, and

- a family of propositions $A''(x)$ **prop** $[x \in A']$, again in the basic theory.

This corresponds to our definition of specification in Section 3.

Equality of sets A, B is based on equality of the underlying sets A', B', but uses *logical* equivalence of the families A'', B''. We rejected such a choice, in favour of the decidable relation of convertibility in ECC.

The membership judgment $a \in A$ is interpreted in the obvious way: we may derive $a \in A$ if we can derive $a \in A'$ and $A''(a)$ **true** in the basic theory. This captures the essential idea, that the judgments A **set** and $a \in A$ should describe subsets of the terms in A' in the underlying theory. In this way, the proofs of the propositions $A''(a)$ are systematically suppressed. It is then relatively straightforward to see how this allows the interpretation of a subset-type constructor.

How does this compare with our approach? The resulting expressions for the various type constructors are very similar, compare for example the definition of exponential for second-order deliverables with the Π-type in the subset interpretation. We consider explicit proofs of the propositional parts of our specifications, whereas we need only know that some proof may be derived in the subset theory. However, this seems to be one of the limitations of their approach, in that we only know that certain *derivations* in the subset theory arise from certain other derivations. Our use of Σ-types, by contrast, means that we can represent the derivations of deliverables as actual *terms* within ECC, using the definable combinators which code up the explicit translation. The price we pay seems to be that we have to work with a rather clumsy language for these terms, as opposed to the conceptual elegance of reusing the basic language of types and terms in the subset theory.

The NuPrl system, developed by Constable and his co-workers [5], is based on early versions of Martin-Löf's type theory. In particular, the underlying term calculus is untyped, and the system has extensional equality types. This has the advantage of suppressing some irrelevant information in proofs. It also overcomes the limitations on the use of an explicit subset-type constructor in a theory with intensional equality, exposed in [34, 35]. The sovereign disadvantage is that the basic judgments of the theory become undecidable, coupled with a proliferation of well-formedness conditions in the application of the rules.

Recently, Hayashi has also proposed a system based on realisability, which abandons the usual type constructors Π, Σ, on which most work to date on type theory has been based, in favour of a more set-theoretic style, with union, intersection and singleton types [13]. The system he considers is, however, ingenious enough to represent dependent products and dependent sums. At the same time, the typing rules for union and intersection hide information. This allows a simple translation or extraction into a programming language with a polymorphic type discipline. Singleton types seem essential in achieving this harmony between the type system and the underlying untyped terms.

Pavlovič, in his thesis [32], elaborates in categorical terms a theory of constructions in which programs do not depend on proofs of logical propositions.

As with the models of Constructions considered by Hyland and Pitts [18], the emphasis is on extensional systems, rather than the intensional system we work with here. Proof-theoretic properties seem to be regarded as something "...an implementation would have to answer" [32, p.8].

8 Conclusions

We have described a proof development method which takes as its basic entity a program plus its correctness proof. If the programs are functional these program proof-pairs admit exactly the same combination operators as do functions. We have studied the mathematical theory of such program proof-pairs and the operations which combine them, and we have considered the case where the pre- and postconditions are contextually linked to express a relationship between input and output. The combination operations have been coded within the Lego proof system; a number of examples have been developed in Lego, and some metatheoretic results about the operations have been proved in Lego.

Acknowledgements

We would like to thank Randy Pollack for the Lego system and Zhaohui Luo for his contribution to its theoretical underpinning. We also thank our colleagues in the Lego Club for friendly comments and stimulation. We thank Per Martin-Löf, Gordon Plotkin and Susumu Hayashi for helpful comments on presentations of this work. Thanks also to Chris Owens for proof-reading the manuscript.

References

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

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

[3] N.G. de Bruijn, *A survey of the project AUTOMATH*, in: [37].

[4] R.M.Burstall and J.H.McKinna, *Deliverables: an approach to program development in Constructions*, in [17], also available as a University of Edinburgh technical report ECS-LFCS-91-133.

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

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

[7] T.Coquand, *Metamathematical Investigations of a Calculus of Constructions*, in: [16].

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

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

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

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

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

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

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

[15] W.A.Howard, *The "formulae-as-types" notion of construction*, in: [37].

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

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

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

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

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

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

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

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

[24] Z.Luo and R.Pollack, *LEGO Proof Development System: User's Manual*, LFCS Technical Report ECS–LFCS–92-211, 1992.

[25] J.H.McKinna, *Deliverables: a categorical approach to program development in type theory*, Ph.D. thesis, University of Edinburgh, 1992.

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

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

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

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

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

[31] C.Paulin-Mohring and B.Werner, *Extracting and Executing Programs developed in the Inductive Constructions System: a Progress Report*, in: [17].

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

[33] R.A.Pollack, *Implicit Syntax*, in: [17].

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

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

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

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

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