Chapter 6

Further work and conclusions

6.1 Partial equivalence relations and observational equivalence

The basic definitions of Chapter 3 contain a lot of redundancy, as far as proofs are concerned. Having chosen a notion of equality sufficient to give a smooth theory of the categorical structure of $\mathbf{del}_1$, we distinguish different proofs that a given program meets a given specification, whereas in practical terms, we are only interested in the *existence* of a proof. The proof-irrelevance of topos models of higher-order logic, and the possibility of developing a theory of deliverables in the abstract setting of a topos, suggests that we modify the basic definition of deliverable.

In fact, a more extreme modification seems indicated, once we are prepared to abandon the decidable (because reducible to type-checking in ECC) property of being a deliverable in the sense of Definition 3.1.2, in favour of this semi-decidable type-inhabitation problem. Namely, why should we restrict ourselves by distinguishing functions according to their intensional character, when they are equivalent in respect of meeting a certain input-output specification? This suggests that we should advance a fully-fledged theory of specifications based on observational equivalence of the functions under consideration, where our observations are the specifications. This is where we are led to introduce partial equivalence relations (PERs).

Consider the following definition, in context Γ.

Definition 6.1.1 *specification*
A *specification* is given by a pair s, S, where $\Gamma \vdash s : \mathit{Type}$ and $\Gamma \vdash S : s \longrightarrow s \longrightarrow \mathit{Prop}$, such that there are proofs $\Gamma \vdash symS : \forall x, y\colon s.\ Sxy \Longrightarrow Syx$ and $\Gamma \vdash transS : \forall x, y, z\colon s.\ Sxy \Longrightarrow Syz \Longrightarrow Sxz$.

That is to say, we consider types together with a partial equivalence relation defined over them. There is already an issue here, as to whether the terms $symS,\ transS$ are part of the data, or whether we simply require such terms to be derivable. As with the proof terms of Chapter 3, this seems to be a question of book-keeping. A specification in this sense gives rise to one in the sense of Definition 3.1.2, by passing to the diagonal: $S \mapsto \lambda x{:}s.\ Sxx$. A specification *à la* Definition 3.1.2 gives rise to one in this new sense, using Leibniz' equality EQ: $S \mapsto \lambda x, y{:}s.\ Sx \wedge EQ\ x\ y \wedge Sy$. This is easily shown to be a PER on s, since EQ is an equivalence relation: the failure of reflexivity arises from the possible non-totality of the predicate S.

On the ground types, of course, we typically consider equivalence relations — partiality is forced on us when we pass to higher types, with the notion of exponential familiar from logical relations:

$$S^T =_{\text{def}} \lambda f, g{:}s \longrightarrow t.\ \forall x, y{:}s.\ Sxy \Longrightarrow T(fx)(gy).$$

In this framework, the correct notion of deliverable is now that of "PER-respecting function", and again, we can internalise the notion of hom-object, by considering the collection of functions modulo the above PER, to obtain an appropriate notion of cartesian closure. Indeed, it seems possible to develop all of the categorical structure of Chapter 3 on the basis of these new definitions, modulo a principled account of the proof-irrelevance. In general, this seems quite hard, so we have contented ourselves to examining a fully explicit system, via ECC, and the same ideas in the context of the well-studied proof-irrelevant interpretation of higher-order logic in toposes.

Further work remains to be done in exploring the relationships between explicit systems and those which abstract away the details of computationally irrelevant proofs and non-observable behaviour. I have undertaken a preliminary investigation in LEGO, using PERs as specifications, but the work is not yet complete. There seem to be a number of outstanding technical details relating this representation to the categorical perspective of the last chapter.

6.2 Data abstraction

In his paper [61], Luo considers a framework for the specification of abstract data-types and operations defined over them, together with a notion of *refinement*, which is exactly the definition of deliverable, except that refinement maps are taken as going in the opposite direction. He introduces a number of operations on specifications, corresponding to the categorical structure of Chapter 3, but does not consider these in a categorical framework: for example, he does not consider the closed structure corresponding to the idea of hypothetical specification. Also, the analysis is not restricted to simple types at the level of refinement maps, and is thus able to define dependent families of specifications rather more straightforwardly than our account via second-order deliverables. A natural extension of the present work would be to bring together the experience of data abstraction via deliverables, with the obvious influence of ideas from "programming in the large" [97,68, for example], and the "programming in the small" experience reported here. Also, it should perhaps be evident that ideas from the "PER" view of deliverables, outlined above, may clarify issues of behavioural abstraction between different implementations of abstract datatypes.

6.3 Parametricity and second-order λ-calculus

The Calculus of Constructions, and hence ECC, builds on original work of Girard on higher-order extensions of the Curry-Howard correspondence, with applications to proof normalisation for higher-order logic [32]. Reynolds independently rediscovered Girard's second-order λ-calculus in the study of programming languages with polymorphism. In a number of subsequent papers [90,108,91, among others], Reynolds and others have attempted to describe *parametricity* in models of this calculus. Rather than attempt to describe the aims of this work, we merely observe here a number of comparisons which may be made between the ideas underlying this thesis, and work on parametricity.

In particular, parametricity has been used, in Wadler's [108], as an approach to proving properties of programs in second-order λ-calculus. Reynolds' idea was to give an interpretation of the types of the system as *relations*, in the style of logical relations. This clearly has links to both Martin-Löf's subset interpretation, and the type-theoretic description of deliverables. However, since we have analysed a predicative account of computational types, rather than the impredicative style of programming in second-order λ-calculus, these relationships need to be elaborated in some future work.

6.4 Extraction and realisability

Other authors, notably Paulin-Mohring and her collaborators in the Formel project [81,82], Hayashi [39], and the NuPrl group under Constable [15] have studied proofs in constructive mathematics with a view to extracting programs, via realisability translations. These were originally conceived by Kleene as giving a strong constructive reading of the logical connectives, in order to validate certain intuitionistic principles. In contemporary treatments, a proof in the formal system, such as the Calculus of Constructions, is annotated in such a way as to mark those proofs which are deemed computationally relevant, and then a syntactic map applied to the proof term, to yield a term (=program) in some related functional system. In general, this will not remove all the computationally irrelevant information, nor does it necessarily yield familiar algorithms. In Paulin's work, programs are obtained in F_ω, so we are left, as in the work on parametricity above, with a discrepancy between the target programming languages to account for. Nonetheless, one of our reasons for considering a simply-typed programming language, though at the predicative type level in ECC, is that we may throughout replace the primitive recursions with their counterparts in F_ω, considered as a subsystem of the impredicative level of ECC. This opens the way to comparisons with at least Paulin's work on extraction. As a starting point, we may conjecture that the function component of our deliverable constructors for natural number and list recursions are the extracts of their proof component. These relationships remain to be made precise, not least since we do not believe we have a definitive account of recursion within our system, but one which has proved satisfactory for the small examples we have considered.

6.5 Partial functions in type theory

The strong normalisation theorem for ECC[1] implies we may only represent total recursive functions. Since partiality arises naturally in any theory of computation, this limitation should be addressed. Various authors have considered "partial objects" in logic and type theory [16,4,75,99, for example], but it is not immediately evident how to adapt their methods to the framework of deliverables. Category theory defines a partial map from s to t as given by a monomorphism $s' \longrightarrow s$, and a morphism $s' \longrightarrow t$. To adapt this definition, which requires an explicit criterion for definability, namely the *domain* $s' \longrightarrow s$, we employ our intuition that the domain should be represented, logically, as a predicate on s. That is to say, a partial map with domain S, is a function from $\Sigma x{:}s.\ Sx$. Such a function will use the proof of Sx in an essential way. For the purposes of extending the work presented here, it seems a natural definition, with a highly constructive flavour. This seems to be a stronger constructive notion of partial map that those defined in logics with an existence predicate [28,99], which are rather more flexible in how one obtains proofs that a given term denotes. Pragmatically, and theoretically, one would hope to do rather better in reconciling partiality and constructive type theory.

[1]Strong normalisation for the calculus extended with inductive types, and $\beta\delta$-reduction is still an open problem.

6.6 Pragmatics

The examples we have exhibited show that it is far from trivial to formalise the simple arguments used in paper-and-pen verifications of small pieces of code. Clearly, this is in part due to the extra overhead in fully formalised proofs, but equally clearly, this is not the only limitation.

Nonetheless, we regard the experience with the Chinese remainder theorem as good support for our approach: a proof, fully formalised "by hand", *i.e.* considered as a *pointwise* construction, is greatly reduced in length and complexity by the recursion/induction principle for second-order deliverables over lists. But we have yet to formalise the second and third stages of the algorithm using deliverables, so our analysis must be regarded as provisional.

Also, we certainly do not consider the rules for recursion in the case of second-order deliverables to be a definitive account, although in the examples we have considered, they appear to be adequate.

These considerations widen in scope when we come to consider the nature of large programs. We have not considered the kinds of modular development of programs discussed, for example, in the literature on Extended ML [96,97]. The examples of minimum finding and insert sort are parametrised by the the underlying type and its boolean-valued ordering relation, which gives a certain modularity to the constructions. It remains an open question, which can only satisfactorily be answered in the light of greater experience, whether the methodology we propose is more appropriate to the verification of small-scale pieces of code, or the kinds of signature matching conditions encountered in using the modules system of ML (possibly augmented with axioms) to do large-scale developments[2].

The outstanding deficit in any proposal for using deliverables as a programming methodology must be the lack of a type-theoretic language for describing them. Chapter 5 represents a good start in this direction, but many details remain to be elaborated.

6.7 Conclusions

We have shown that it possible to give a principled account of a general notion of functions which respect specifications, our so-called deliverables. Both syntactically, and semantically in the particular setting of a model of higher-order logic, we are able to lift the structure of functions to that of deliverables. The theory seems quite well supported by the small number of examples we have considered. Various limitations have been observed in our approach. We expect that further work in the development of suitable type theories for describing deliverables, in the style of Chapter 5, may extend the utility of the methodology.

[2]Indeed, it was already in this light, that Luo and others used the idea of deliverable, in the guise of *theory morphism*, to describe structuring proof development in mathematical theories [64,59].

Appendix A

Fibrations

In this Appendix, we discuss some of the general categorical framework underlying the constructions of Chapter 5. For the further details on fibrations, the interested reader is referred to the papers [6,84,46,48].

A.1 Basic definitions

The idea of a fibration captures two notions at the heart of any theory of dependent types: that types and terms are relative to given context, and that substitution and the rules for valid contexts regulate the passage between data relative to different contexts. Our data arises in two ways: as valid contexts and well-typed substitutions between them, and as types-in-context and well-typed terms-in-context. As in the case of simple types, we may organise the former as category $\mathcal{B}$. How should we organise the latter? We might naïvely hope to do this with another category $\mathcal{F}$ of "judgments-in-context" which is "related to" $\mathcal{B}$, in this case by a functor $p : \mathcal{F} \longrightarrow \mathcal{B}$. What should this relationship p be? We would perhaps ask that p takes a judgment and returns the context in which it has been derived. $\mathcal{F}$ should reflect the derivations of judgments, in particular derivations which use the rule of substitution. Since $\mathcal{B}$ is intended to model well-typed substitutions, this structure should be reflected in $\mathcal{F}$.

Definition A.1.1 *cartesian arrow*

Let $p : \mathcal{F} \longrightarrow \mathcal{B}$ be a functor. Then an arrow $f : B \longrightarrow A$ in $\mathcal{F}$ is p-*cartesian* (over pf) if it has the following "terminal lift" property: given any $g : C \longrightarrow A$ in F, and $h' : pC \longrightarrow pB$ such that

$$\begin{tikzcd} pC \arrow[d, "h'"'] \arrow[dr, "pg"] & \\ pB \arrow[r, "pf"'] & pA \end{tikzcd}$$

commutes, then there is a unique lift h of h' (i.e. $ph = h'$) such that

$$\begin{tikzcd} C \arrow[d, "\exists! h"'] \arrow[dr, "g"] & \\ B \arrow[r, "f"'] & A \end{tikzcd}$$

commutes.

Definition A.1.2 *fibration*

$p : \mathcal{F} \longrightarrow \mathcal{B}$ is a *fibration* if every $\phi : D \longrightarrow pA$ has a p-cartesian lift f.

As a consequence of the very definition of cartesian arrow, the property of being a fibration is equivalent to the requirement that every $\phi : D \longrightarrow pA$ has a lift to $\mathcal{F}$, and that every arrow $g : C \longrightarrow A$ in $\mathcal{F}$ factors as hf, where $ph = id_{pC}$ and f is p-cartesian. We then speak of h being a "vertical" arrow, while f is "horizontal". The collection of vertical arrows over C forms a subcategory of $\mathcal{F}$, called the *fibre* over C. Metaphorically, the cartesian arrows over an f in the base $\mathcal{B}$ are "translations" between the fibres, "parallel" to f: they are fixed uniquely[1], given a choice of f and an object A in the fibre over the codomain of f. The above definition of cartesian arrow is no more than a formalisation of this metaphor. It is the cartesian arrows which reflect the structure of the base $\mathcal{B}$ in $\mathcal{F}$, in the sense hinted at in the introduction to this section. A fibration is then no more than a collection of local data, the fibres, knitted together by these translations (or relativisations; even substitutions), the cartesian arrows, in this essentially unique way.

A central example to the treatment of dependent types is the functor $\mathcal{B}^2 \xrightarrow{cod} \mathcal{B}$, where $\mathcal{B}^2$ has objects the arrows $d : W \longrightarrow X$ in $\mathcal{B}$, and morphisms those (g, f) which yield commutative squares, with cod given by the codomain map:

$$\begin{tikzcd}
Z \arrow[r, "g"] \arrow[d, "d'"'] & W \arrow[d, "d"] \\
Y \arrow[r, "f"] & X \\
Y \arrow[r, "f"'] & X
\end{tikzcd}$$

$$\downarrow cod$$

Lemma A.1.1 *The cod-cartesian arrows in $\mathcal{B}^2$ are precisely the pullback squares.*

Lemma A.1.2 *cod is a fibration precisely when $\mathcal{B}$ has all pullbacks.*

In the study of dependent types, we are principally interested in subfibrations of *cod*. Given a category with fibrations $(\mathcal{B}, Fib, ^\star, \bullet)$, we may form the category $\mathcal{F}(\mathcal{B})$, whose objects are the display arrows, and whose morphisms

[1] This is Euclid's fifth postulate!

(g, f) are the commutative squares:

$$\begin{tikzcd} Y \bullet B \arrow[r, "g"] \arrow[d, "d_B"'] & X \bullet A \arrow[d, "d_A"] \\ Y \arrow[r, "f"'] & X \end{tikzcd}$$

There is an obvious inclusion $i : \mathcal{F}(\mathcal{B}) \rightarrow \mathcal{B}^2$.

Theorem A.1.1 *$i; cod{:}\mathcal{F}(\mathcal{B}) \longrightarrow \mathcal{B}$ is a fibration.*

Proof Immediate from the data in $\mathcal{B}$ and our earlier discussion of cod. The cartesian arrows are given precisely by the pairs $(f, f\bullet A)$ with $f : Y \longrightarrow X$ in $\mathcal{B}$ and $A \in Fib(X)$. ■

A.2 Naturality and the Beck-Chevalley condition in categories with fibrations

The construction for a dependent product varies along three parameters

- the context morphism $f : Y \longrightarrow X$
- the target type B
- the type A over which we abstract

The naturality of the Π construction, which concerns variation along the first two parameters, is expressed as follows: given $g : Z \longrightarrow X$ in $\mathcal{B}$, $A \in Fib(X)$, $B, C \in Fib(X\bullet A)$, $\alpha : Y\bullet f^{\star}A \longrightarrow (X\bullet A)\bullet B$, $h : (X\bullet A)\bullet B \longrightarrow (X\bullet A)\bullet C$ the two parallel arrows below should be equal

$$
\begin{tikzcd}
(X\bullet A)\bullet C & Z\bullet g;f^{\star}A \arrow[l, dotted, "g';\alpha;h"'] \arrow[r] \arrow[d, "\exists! g'"'] \arrow[dr, phantom, "\lrcorner"] & Z \arrow[r, dotted, shift left, "\lambda(g';\alpha;h)"] \arrow[r, shift right, "g;\lambda(\alpha);\Pi_A(h)"'] \arrow[d, "g"] & X\bullet\Pi AC \\
(X\bullet A)\bullet B \arrow[u, "h"] \arrow[dr, "d_B"] & Y\bullet f^{\star}A \arrow[l, "\alpha"'] \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner"] & Y \arrow[r, "\lambda(\alpha)"] \arrow[d, "f"] & X\bullet\Pi AB \arrow[u, "\Pi_A(h)"'] \arrow[dl, "d_{\Pi AB}"] \\
& X\bullet A \arrow[r, "d_A"'] & X &
\end{tikzcd}
$$

where

$$\Pi_A(h) = \lambda(\rho(id_{X\bullet\Pi AB});h) : X\bullet\Pi AB \longrightarrow X\bullet\Pi AC$$

In addition, we require that the construction of Π respects substitution — the so-called Beck-Chevalley condition [45,46,49,48] — which is no more than the variation of the third of the above parameters, namely the variation of the abstraction type A along the substitution f. It may be formulated as follows: for all $f : Y \longrightarrow X$ in $\mathcal{B}$, $A \in Fib(X)$, $B \in Fib(X\bullet A)$, the induced map

$$\eta : Y\bullet\Pi(f^{\star}A)((f\bullet A)^{\star}B) \longrightarrow Y\bullet f^{\star}\Pi AB$$

should be an isomorphism, where η is obtained from a chase round the following diagram:

$$
\begin{tikzcd}
 & Y \bullet \Pi(f^\star A)((f\bullet A)^\star B)\bullet d^\star(f^\star A) \arrow[dl, "\rho(id)"'] \arrow[d, "d"'] \arrow[r, "d"] \arrow[dr, phantom, "\lrcorner", very near start] & Y \bullet \Pi(f^\star A)((f\bullet A)^\star B) \arrow[d, "d"] \\
Y \bullet f^\star A \bullet (f\bullet A)^\star B \arrow[r, "d"] \arrow[d, "(f\bullet A)\bullet B"'] \arrow[dr, phantom, "\lrcorner", very near start] & Y \bullet f^\star A \arrow[r, "d"] \arrow[d, "f\bullet A"'] \arrow[dr, phantom, "\lrcorner", very near start] & Y \arrow[d, "f"] \\
X \bullet A \bullet B \arrow[r, "d"'] & X \bullet A \arrow[r, "d"'] & X \\
 & Y \bullet \Pi(f^\star A)((f\bullet A)^\star B)\bullet (d;f)^\star A \arrow[ul, "\alpha;\rho(id);f\bullet A\bullet B"] \arrow[u, "d"] \arrow[r, "d"'] \arrow[ur, phantom, "\urcorner", very near start] & Y \bullet \Pi(f^\star A)((f\bullet A)^\star B) \arrow[u, "d"']
\end{tikzcd}
$$

Both $Y \bullet \Pi(f^\star A)((f\bullet A)^\star B)\bullet (d;f)^\star A$ and $Y \bullet \Pi(f^\star A)((f\bullet A)^\star B)\bullet d^\star(f^\star A)$ are vertices of pullbacks of $d;f$ against d, hence there exists a unique isomorphism

$$\alpha : Y \bullet \Pi(f^\star A)((f\bullet A)^\star B)\bullet (d;f)^\star A \longrightarrow Y \bullet \Pi(f^\star A)((f\bullet A)^\star B)\bullet d^\star(f^\star A)$$

Thus we obtain an arrow

$$\phi = \alpha;\rho(id); f\bullet A\bullet B : Y \bullet \Pi(f^\star A)((f\bullet A)^\star B)\bullet (d;f)^\star A \longrightarrow X \bullet A \bullet B$$

as indicated, whose transpose across the bijection yields a commuting square

$$
\begin{tikzcd}
Y \bullet \Pi(f^\star A)((f\bullet A)^\star B) \arrow[r, "\lambda(\phi)"] \arrow[d, "d"'] & X \bullet \Pi AB \arrow[d, "d"] \\
Y \arrow[r, "f"'] & X
\end{tikzcd}
$$

Since

$$\begin{tikzcd} Y \bullet f^{\star}\Pi AB \arrow[r] \arrow[d, "d"'] \arrow[dr, phantom, "\lrcorner", very near start] & X \bullet \Pi AB \arrow[d, "d"] \\ Y \arrow[r, "f"'] & X \end{tikzcd}$$

is a pullback, we obtain the unique mediating arrow

$$Y \bullet \Pi(f^{\star}A)((f \bullet A)^{\star}B) \xrightarrow{\eta} Y \bullet f^{\star}\Pi AB$$

as required. We may therefore rephrase the Beck-Chevalley condition as requiring that

$$\begin{tikzcd} Y \bullet \Pi(f^{\star}A)((f \bullet A)^{\star}B) \arrow[r, "\lambda(\alpha;\, \rho(id);\, f \bullet A \bullet B)"] \arrow[d, "d"'] & X \bullet \Pi AB \arrow[d, "d"] \\ Y \arrow[r, "f"'] & X \end{tikzcd}$$

be a pullback in $\mathcal{B}$.

Theorem A.2.1 *The model of Chapter 5 satisfies this form of the Beck-Chevalley condition.*

Proof Straightforward. The condition holds on the carrier sets of the dependent products, since it is inherited from **Sets**. That the predicates are preserved follows by elementary logic and a chase round the diagram. ■

Appendix B

LEGO code relevant to this thesis

We present here the proof scripts for all the constructions we have considered, except the categorical model of Chapter 5. We have suppressed a good number of lemmas which we required in the course of studying the examples, especially that of Theorem 4.4.1. We hope at some future date to make this library of theorems and lemmas available by anonymous ftp. We offer only minimal commentary, often in the form of comments in the code, signified by the `(* ... *)` notation.

B.1 Basics

Here we collect most of the mathematical knowledge which we employed in the examples. In order to suppress unnecessary information, we simply present the names of terms from the LEGO context which we developed. The reader may certainly omit this material on a first reading of the examples below.

B.1.1 Basic logic

These definitions are a slight modification of those which appear in [64].

```
Init XCC;
[A,B,C,D|Prop][a:A][b:B][c:C][d:D][T,S,U|Type];
[cut = [a:A][h:A->B]h a:A->(A->B)->B];
[I [t:T] = t:T]
[compose [f:S->U][g:T->S] = [x:T]f (g x):T->U]
[permute [f:T->S->U] = [s:S][t:T]f t s:S->T->U];
DischargeKeep A;

(* Conjunction, Disjunction and Negation *)
[and [A,B:Prop] = {C:Prop}(A->B->C)->C:Prop]
[or  [A,B:Prop] = {C:Prop}(A->C)->(B->C)->C:Prop]
[pair = [C:Prop][h:A->B->C](h a b):and A B]
[inl = [C:Prop][h:A->C][_:B->C]h a:or A B]
[inr = [C:Prop][_:A->C][h:B->C]h b:or A B]
[fst [h:and A B] = h A [g:A][_:B]g:A]
[snd [h:and A B] = h B [_:A][g:B]g:B]

(* Constants *)
[false = {A:Prop}A];
[not [A:Prop] = A->false];
[true = {A:Prop}A->A];
[top = [A:Prop][a:A]a:true];

(* Quantification *)
(* a uniform Pi *)
[All [P:T->Prop] = {x:T}P x:Prop]
(* Existential quantifier *)
[Ex [P:T->Prop] = {B|Prop}({t:T}(P t)->B)->B:Prop]
[ExIntro [P|T->Prop][witness:T][prf:P wit]
 = [B|Prop][gen:{t:T}(P t)->B](gen witness prf):Ex P]

(* tuples *)
[and3 [A,B,C:Prop] = {X:Prop}(A->B->C->X)->X:Prop]
[pair3 = [X:Prop][h:A->B->C->X](h a b c):and3 A B C]
[and3_out1 [p:and3 A B C] = p A [a:A][_:B][_:C]a:A]
[and3_out2 [p:and3 A B C] = p B [_:A][b:B][_:C]b:B]
[and3_out3 [p:and3 A B C] = p C [_:A][_:B][c:C]c:C]
[and4 [A,B,C,D:Prop] = {chi|Prop}{p:A->B->C->D->chi}chi];
```

```
[pair4 = [chi|Prop][p:A->B->C->D->chi](p a b c d):and4 A B C D];

(* Predicates, Relations: prerequisites for deliverables *)

[Pred = [s:Type]s->Prop];
[Rel = [s,t:Type]s->t->Prop];

[R:Rel T T];
[refl = {t:T}R t t:Prop]
[sym = {t,u|T}(R t u)->(R u t):Prop]
[trans = {t,u,v|T}(R t u)->(R u v)->(R t v):Prop];
[preorder = and refl trans];
[per = and sym trans];
[equiv = and refl per];

Goal per -> {x:T}(Ex [y:T]R x y) -> (R x x);
Intros _;andE H;Intros __;exE H3;Intros y _;
Refine H2;Refine +2 H1;Immed;
Save perlemma;

Discharge R;

(* families of relations *)
[preserves [f:T->S][R:Rel T T][Q:Rel S S]
  = {t,u|T}(R t u)->(Q (f t) (f u)):Prop];
[respect [f:T->S][R:{X|Type}Rel X X]
  = preserves f (R|T) (R|S):Prop];
DischargeKeep A;

(* Equality *)
[EQ =  [x,y:T]{P:Pred T}(P x)->(P y):Rel T T];
[reflEQ = [t:T][P:Pred T][h:P t]h:refl EQ]
[symEQ = [t,u|T][g:EQ t u]g ([x:T]EQ x t) (reflEQ t):sym EQ]
[transEQ:trans EQ
  = [t,u,v|T][p:EQ t u][q:EQ u v][P:Pred T]compose (q P) (p P)];
DischargeKeep A;
(* application respects equality; a substitution property *)
[respEQ [f:T->S]:respect f EQ
```

```
  = [t,u|T][h:EQ t u]h ([z:T]EQ (f t) (f z)) (reflEQ (f t))];

[pi1 = [p:S#T]p.1];(* for want of anywhere better *)
[pi2 = [p:S#T]p.2];

Discharge A;
```

B.1.2 Basic datatypes: unit, booleans, and naturals

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

Goal {u:unit}EQ void u;
Intros ___;Refine unitrecd;Immed;
Save voidunique;

[T:Type];
Goal (unit -> T) -> T;
Intros phi;Refine phi void;
Save elemR;
Goal T -> (unit -> T);
Intros __;Immed;
Save elemL;
Discharge T;

(* Bool *)
[bool:Type(0)];
[tt:bool];
[ff:bool];
[boolrecd:{C:bool->Type}{d:C tt}{e:C ff}{b:bool}C b];
[[C:bool->Type][d:C tt][e:C ff]
    boolrecd C d e tt ==> d
 || boolrecd C d e ff ==> e];
```

```
[boolind [phi:bool->Prop][phi_tt:phi tt][phi_ff:phi ff]
        = boolrecd phi phi_tt phi_ff];
[boolrec [T|Type][t,f:T][b:bool] = boolrecd ([_:bool]T) t f b];

[if [a:bool][D|Type][d,e:D] = boolrecd ([_:bool]D) d e a];
[andb [a,b:bool] = if a b ff];
[orb  [a,b:bool] = if a tt (if b tt ff)];
[impb [a,b:bool] = if b tt (if a ff tt)];
[notb [b:bool]   = if b ff tt];

Goal {b:bool}or (EQ tt b) (EQ ff b);
Refine boolind [b:bool]or (EQ tt b) (EQ ff b);
Refine inl;Refine reflEQ;Refine inr;Refine reflEQ;
Save boolIsInductive;

Goal not(EQ tt ff);
Intros eq;
Refine eq (boolrec true false);
Intros;Immed;
Save peano4bool;

(* a new nat *)

[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 ==> z
|| natrecd C z s (succ n) ==> s n (natrecd C z s n)];

[natiter [C|Type][z:C][s:C->C]
        = natrecd ([_:nat]C) z ([_:nat][c:C]s c)];
[natrec [C|Type][z:C][s:nat->C->C] = natrecd ([_:nat]C) z s];
[natind [phi:nat->Prop]
```

```
        [phi_zero:phi zero]
        [phi_succ:{k:nat}{ih:phi k}phi (succ k)]
       =  natrecd phi phi_zero phi_succ];

[one = succ zero][two = succ one];
[three = succ two][four = succ three];
[five = succ four][six = succ five];
[seven = succ six][eight = succ seven];
[plus [n,m:nat] : nat = natiter m succ n];
[mult [n,m:nat] : nat = natiter zero (plus m) n];
[exp  [m,n:nat] : nat = natiter one (mult m) n];
[pred [n:nat] : nat = natrec zero ([x,_:nat]x) n];
[minus [m,n:nat] : nat = natiter m pred n];
[maxNat [m,n:nat] = plus m (minus n m)];
[minNat [m,n:nat] = minus n (minus n m)];
[leNat = [m,n:nat] Ex [k:nat]EQ n (plus (succ k) m):Rel nat nat];
[leqNat = [m,n:nat] Ex [k:nat]EQ n (plus k m):Rel nat nat];

Goal {m,n|nat}(EQ (succ m) (succ n)) -> EQ m n;
Intros;Refine H ([k:nat]P(pred k));Immed;
Save peano3Nat;
Goal not (EQ zero one);
Intros;Refine H (natiter true ([_:Prop]false));Intros;Immed;
Save peano4Nat;
Goal {n|nat}not (EQ zero (succ n));
Intros;Refine H (natiter true ([_:Prop]false));Intros;Immed;
Save peano4Nat';
Goal {n|nat}EQ n (plus n zero);
Refine natind [k:nat]EQ k (plus k zero);
Refine reflEQ;intros;Refine respEQ succ;Immed;
Save pluslemma0;
Goal {n,m|nat}EQ (succ (plus m n)) (plus m (succ n));
intros;
Refine natind [m:nat]EQ (succ (plus m n)) (plus m (succ n));
Refine reflEQ;intros;Refine respEQ succ;Immed;
Save pluslemmaS;
Goal {m,n|nat}EQ (plus m n) (plus n m);
Refine natind [m:nat]{n:nat}EQ (plus m n) (plus n m);
```

```
intros;Refine pluslemma0;
intros;Refine pluslemmaS;Refine respEQ succ;Refine ih;
Save pluscommutes;
```

B.1.3 The calculus of relations

This is an edited version of a much more extensive section, based on a systematic study of the representation of the calculus of relations in LEGO. It formed part of a general study of iterated inductive definitions, with applications to the theory of permutation.

```
[relations:Prop];

[s,t,u,v|Type];

[SubPred = [F,G:Pred s]{x|s}{hyp:F x}G x];
[reflSubPred = [F:Pred s][x|s][hyp:F x]hyp:refl SubPred];
[transSubPred = [F,G,H|Pred s]
                [FsubG:SubPred F G][GsubH:SubPred G H]
                [x|s][hyp:F x]GsubH (FsubG hyp):trans SubPred];

[SubRel = [Q,R:Rel s t]{x|s}{y|t}{hyp:Q x y}R x y];
[reflSubRel = [P:Rel s t][x|s][y|t][hyp:P x y]hyp:refl SubRel];
[transSubRel = [P,Q,R|Rel s t][PsubQ:SubRel P Q][QsubR:SubRel Q R]
              [x|s][y|t][hyp:P x y]QsubR (PsubQ hyp):trans SubRel];

[andPred [F,G:Pred s] = [x:s]and (F x) (G x):Pred s];
[orPred [F,G:Pred s] = [x:s]or (F x) (G x):Pred s];
[impliesPred [G,E:Pred s] = [x:s]{hypG:G x}E x:Pred s];
[notPred [F:Pred s] = [x:s]not(F x):Pred s];

[op [P:Rel s t] = [y:t][x:s]P x y:Rel t s];
[notRel [P:Rel s t] = [x:s][y:t]not(P x y):Rel s t];
[andRel [P,Q:Rel s t] = [x:s][y:t]and (P x y) (Q x y):Rel s t];
[orRel [P,Q:Rel s t] = [x:s][y:t]or (P x y) (Q x y):Rel s t];
[composeRel [R:Rel s t][S:Rel t u] = [x:s][z:u]
```

```
   {phi:Prop}{ex_y:{y:t}{hypR:R x y}{hypS:S y z}phi}phi:Rel s u];
[impliesRel [R:Rel s t][T:Rel s u]
          = [y:t][z:u]{x:s}{hypR:R x y}T x z:Rel t u];
[coimpliesRel [S:Rel t u][T:Rel s u]
            = [x:s][y:t]{z:u}{hypS:S y z}T x z:Rel s t];

[KPred [P:Prop] = [x:s]P:Pred s];
[KRelL [F:Pred s] = [x:s][y:t]F x:Rel s t];
[KRelR [G:Pred t] = [x:s][y:t]G y:Rel s t];

[univPred = KPred true];
[univRel = KRelL univPred];
[emptyPred = KPred false];
[emptyRel = KRelL emptyPred];
[univPredI = [s|Type][F|Pred s][x|s][_:F x]top
          : {s|Type}{F|Pred s}SubPred F (univPred)];
[monotonePred [phi:(Pred s) -> Pred t]
                = preserves phi (SubPred|s) (SubPred|t)];
[monotoneRel [phi:(Rel s t) -> Rel u v]
                = preserves phi (SubRel|s|t)(SubRel|u|v)];

Discharge s;
```

B.1.4 Polymorphic lists

```
[list:([T:Type]T->T)Type]     (* thanks healf *)
[nil:{A:Type}list A]
[cons:{A|Type}A->(list A)->(list A)]
[listrecd:{A|Type}{C:(list A)->Type}
         {Lbase:C (nil A)}
         {Lstep:{b:A}{k:list A}{ih:C k}C (cons b k)}
         {l:list A}C l];
[[A:Type]
[a:A][l:list A]
[C:(list A) -> Type]
[d:C (nil A)]
[e:{b:A}{k:list A}(C k) -> (C (cons b k))]
```

```
      listrecd C d e (nil A) ==> d
|| listrecd C d e (cons a l) ==> e a l (listrecd C d e l)];

[listind [A|Type][phi:(list A) -> Prop]
         [phi_nil:phi (nil A)]
         [phi_cons:{b:A}{k:list A}{ih:phi k}phi (cons b k)]
        = [l:list A]listrecd phi phi_nil phi_cons l];
[listiter [A|Type][C|Type][d:C][e:A->C->C]
         = listrecd ([_:list A]C) d ([a:A][_:list A][c:C]e a c)];
[listrec [A|Type][C|Type][d:C][e:A->(list A)->C->C]
        = listrecd ([_:list A]C) d e];

[A|Type];

[atom [a:A] = cons a (nil A)];
[head [l:list A][a:A] = listiter a ([b,_:A]b) l];
[tail [l:list A] = listrec  (nil A) ([_:A][k,_:list A]k) l];
[append [k,l:list A] = listiter l (cons|A) k];
[listlength = [l:list A]listiter zero ([_:A][n:nat]succ n) l];

[a,b|A][l,m,n|list A];

[headlemma = ... :(EQ (cons a m) l) -> (EQ (cons b n) l) -> EQ a b];
[taillemma = ... :(EQ (cons a m) l) -> (EQ (cons b n) l) -> EQ m n];
[appnil = ... :EQ (append l (nil A)) l];
[appassoc = ... :EQ (append l (append m n))(append (append l m) n)];
[eqappnil = ... :(EQ (append l m) (nil A))->
                                 and (EQ (nil A) l) (EQ (nil A) m)];
[lengthnil = ... :{eq:EQ zero (listlength l)}EQ (nil A) l];
[lengthcons = ... :{n|nat}{eq:EQ (succ n) (listlength l)}
                                    Ex [a:A]EQ l (cons a (tail l))];
[lengthappend = ...
  :EQ (plus (listlength l) (listlength m)) (listlength (append l m))];

Discharge l;

[NonNil = listiter false [_:A][_:Prop]true:Pred (list A)];
[insert [a:A][h,k:list A] = append h (cons a k)];
```

```
[h,k|list A];
[peano4list = ... :{eq:EQ (cons a h) (nil A)}false;
[insertnil = ... :{eq:EQ (insert a h k) (nil A)}false;
DischargeKeep a;

[insertatom = ... :(EQ (insert a h k) (cons b (nil A))) ->
                   and3 (EQ (nil A) h) (EQ a b) (EQ (nil A) k);
Discharge A;
```

This concludes a selection of the large number of lemmas about lists which we found it necessary to prove in the course of this research.

B.1.5 On permutation

We now turn to the theory of permutation of lists, based on the impredicative definition we gave in Chapter 3. This proved to be a major investigation in itself. We first give the definition, and then it is trivial to establish the constructor properties of this relation, namely that this intersection of equivalence relations is indeed an equivalence relation, that it does identify lists up to the commutativity of the append function, and that it is closed under *cons* (and hence by induction, closed under append).

```
(* new permutations, with resolution of the heredity problem *)
[perm:Prop];
[A|Type];
[swap [S:Rel (list A) (list A)]
     = {l,m:list A}S (append l m) (append m l)];
[conscl [a:A][S:Rel (list A) (list A)]
       = [m,n:list A]S (cons a m) (cons a n)];
[consClosed [S:Rel (list A) (list A)]
          = {a:A}(SubRel S (conscl a S))];

[Perm : Rel (list A) (list A) = [l,m:list A]
     {R:Rel (list A) (list A)}
     {reflR:refl R}{symR:sym R}{transR:trans R}
```

```
     {swapR:swap R}{consclR:consClosed R}
      R l m];

[reflPerm = ... : refl Perm];
[symPerm = ... :  sym Perm];
[transPerm = ... : trans Perm];
[swap Perm = ... : swap Perm];
[consclPerm = : consClosed Perm];
```

This next is the crucial elimination rule for this inductive relation. We give the proof, which is the analogue of the derivation of the primitive recursor from the iterator in Church representations of datatypes.

```
Goal {l,m:list A}{perm_hyp:Perm l m}
     {R:Rel (list A) (list A)}
     {reflR:{l:list A}R l l}
     {symR:{l,m:list A}
           {sym_prem:Perm l m}{sym_ih:R l m}
            R m l}
     {transR:{l,m,n:list A}
             {lt_prem:Perm l m}{lt_ih:R l m}
             {rt_prem:Perm m n}{rt_ih:R m n}
              R l n}
     {swapR:{l,m:list A}R (append l m) (append m l)}
     {consclR:{a:A}{m,n:list A}{cons_prem:Perm m n}{cons_ih:R m n}
               R (cons a m) (cons a n)}
     R l m;

Intros;Refine perm_hyp (andRel Perm R);andI;
andI;Intros _;andI;Refine reflPerm;Refine reflR;
andI;Intros ___;andE H;andI;Refine symPerm;Refine +1 symR;Immed;
     Intros _____;andE H;andE H1;andI;
                  Refine transPerm;Refine +3 transR;Immed;
Refine andRelI;
Refine closureInc;Intros l m base_hyp;
Refine base_hyp (andRel Perm R);
andI;
```

```
Intros __;andI;Refine swapPerm;Refine swapR;
Intros a l m cons_hyp;andE cons_hyp;andI;
Refine consclPerm;Refine +1 consclR;Immed;
intros;Immed;
intros;Immed;

Save recPerm;

Goal {R:Rel (list A) (list A)}
     {reflR:{l,m|list A}{eq_prem:EQ l m} R l m}
     {symR:{l,m|list A}
            {sym_prem:Perm l m}{sym_ih:R l m}
             R m l}
     {transR:{l,m,n|list A}
              {lt_prem:Perm l m}{lt_ih:R l m}
              {rt_prem:Perm m n}{rt_ih:R m n}
               R l n}
     {swapR:{l,m|list A}R (append l m) (append m l)}
     {consclR:{a|A}{m,n|list A}{cons_prem:Perm m n}{cons_ih:R m n}
                R (cons a m) (cons a n)}
     SubRel Perm R;

Intros;Refine recPerm;intros +1;Refine reflR;Refine reflEQ;Immed;
Save PermE;

[B|Type];

Goal {R:Rel B (list A)}
     {swapR:{l,m:list A}{h:B}(R h (append l m)) -> R h (append m l)}
     {consclR:{b:A}{l,m:list A}
               {cons_prem:Perm l m}
               {cons_ih:{h:B}(R h l) -> R h m}
     {k:B}(R k (cons b l)) -> R k (cons b m)}
SubRel Perm (impliesRel R R);

intros;
Refine transSubRel;
[kerR = andRel (impliesRel R R) (coimpliesRel (op R) (op R))];
```

```
Refine +1 PermE kerR;
intros;andI;
Intros;Refine eq_prem;Immed;
Intros;Refine symEQ eq_prem;Immed;
intros;andE sym_ih;andI;
Intros;Refine H1;Immed;
Intros;Refine H;Immed;
intros;andE lt_ih;andE rt_ih;andI;
Intros;Refine H2;Refine H;Immed;
Intros;Refine H1;Refine H3;Immed;
intros;andI;Refine swapR;Refine swapR;
intros;andE cons_ih;andI;
Intros;Refine consclR;Immed;
Intros;Refine consclR;Immed;Refine symPerm;Immed;
Refine andRelE1;Refine +1 reflSubRel;
Save PermRrespR;

Discharge B;
```

We now specialise the elimination rule to consider predicates. As an application, we show that the only permutation of the *nil* list is *nil* itself, and the only permutation between singletons occurs when they are in fact equal.

```
Goal {P:Pred (list A)}
     {swapP:{l,m:list A}(P (append l m)) -> P (append m l)}
     {consclP:{b:A}{l,m:list A}
              {cons_prem:Perm l m}
              {cons_ih:iff (P l) (P m)}
              (P (cons b l)) -> P (cons b m)}
     {l,m:list A}{perm_hyp:Perm l m}(P l) -> P m;

intros;Refine PermE [l,m:list A]iff (P l) (P m);Immed;
intros;andI;
intros;Refine eq_prem;Immed;
intros;Refine symEQ eq_prem;Immed;
intros;andE sym_ih;andI;Immed;
intros;andE lt_ih;andE rt_ih;andI;
```

```
intros;Refine H3;Refine H1;Immed;
intros;Refine H2;Refine H4;Immed;
intros;andI;Refine swapP;Refine swapP;
intros;andE cons_ih;andI;
Refine consclP;Immed;
Refine consclP;Refine symPerm;andI +1;Immed;
intros;Refine H1;Immed;
Save PermPredE;

Goal {l,m|list A}{perm_hyp:Perm l m}(EQ (nil A) l)-> EQ (nil A) m;
intros;Refine PermPredE (EQ (nil A));Immed;
intros;Refine eqappnil;Refine +3 symEQ;Immed;
intros;
Refine H2 [l1:list A]EQ (nil A) (append m1 l1);
Refine H3  [m1:list A]EQ (nil A) (append m1 (nil A));
Refine reflEQ;
intros;Refine insertnil|A|b|(nil A);Refine +1 symEQ;Immed;
Save nilPermlemma;

Goal {l:list A}{perm_nil:Perm (nil A) l}EQ (nil A) l;
intros;Refine nilPermlemma;Immed;Refine reflEQ;
Save nilPerm;

[atomPermlemma = ... : {l,m:list A}{perm_hyp:Perm l m}
     {a:A}(EQ (atom a) l) -> Ex [b:A]and (EQ a b) (EQ (atom b) m)];

Goal {a,b:A}(Perm (atom a) (atom b)) -> EQ a b;
intros;Refine atomPermlemma;Refine +2 H;Refine +1 reflEQ;
intros;andE H1;Refine transEQ;Immed;
Refine headlemma;Immed;Refine +1 reflEQ;
Save atomPerm;

Goal {l,m|list A}{Perm_hyp:Perm l m}EQ (listlength l) (listlength m);
Refine PermE [l,m:list A]EQ (listlength l) (listlength m);
intros;Refine eq_prem [m:list A]EQ ? (listlength m);Refine reflEQ;
intros;Refine symEQ;Refine sym_ih;
intros;Refine transEQ;Refine +1 lt_ih;Refine rt_ih;
intros;Refine lengthisahomomorphism;Refine symEQ;
```

```
       Refine lengthisahomomorphism;Refine pluscommutes;
intros;Refine respEQ succ;Refine cons_ih;
Save PermRespLength;
```

We now turn to the rather tricky proof that $\sim$ is a hereditary property. There is one stage of the proof of the insert sort where we require it, in case (iii) of the proof of Lemma 4.3.11.

```
[a:A];
[PermResidue = [l,m:list A]{phi:Prop}
               {ex_hk:{h,k:list A}{perm_lhk:Perm l (append h k)}
                      {ins_hkm:EQ (insert a h k) m}phi}
                            phi : Rel (list A) (list A)];

Goal {h,k,l:list A}{ins_hkl:EQ (insert a h k) l}
       PermResidue (append h k) l;
Intros;Refine ex_hk;Immed;Refine reflPerm;
Save insResidue;

Goal {l,m,n:list A}{res_nlm:PermResidue n (append l m)}
         PermResidue n (append m l);
intros;Refine res_nlm;
intros;Refine insertapp;Immed;
intros;Refine H;
Intros;Refine ex_hk;
Refine +3 H1 [l:list A]EQ|(list A) ? (append m l);
Refine +3 symEQ(appassoc|?|?|?|?);
Refine transPerm;Refine +1 perm_lhk;
Refine H2 [k:list A]Perm (append h k) ?;
Refine appassoc;
Refine symPerm;Refine symEQ(appassoc|?|?|?|?);
Refine swapPerm;
intros;Refine H;
Intros;Refine ex_hk;
Refine +3 H2 [m:list A]EQ|(list A) ? (append m l);
Refine +3 appassoc;Refine +3 reflEQ;
Refine transPerm;Refine +1 perm_lhk;
```

```
Refine H1 [h:list A]Perm (append h k) ?;
Refine symEQ(appassoc|?|?|?|?);
Refine symPerm;Refine appassoc;
Refine swapPerm;
Save swapResidue;

Goal {b:A}{l,m:list A}
     {cons_prem:Perm l m}
     {cons_ih:{h:list A}{res_hl:PermResidue h l}PermResidue h m}
     {k:list A}{res_kbl:PermResidue k (cons b l)}
      PermResidue k (cons b m);
intros;Refine res_kbl;
intros;Refine insertapp (cons b (nil A)) l;Immed;
intros;Refine H;
intros;Refine insertatom;Immed;
Intros nileqh aeqb nileqn __;Refine ex_hk;
Refine +3 aeqb [b:A]EQ ? (cons b m);
Refine nil A;Refine +2 reflEQ;
Refine transPerm;Refine +1 perm_lhk;
Refine H2 [k1:list A]Perm (append ? k1) ?;
Refine nileqh [h:list A]Perm (append h ?) ?;
Refine nileqn [n:list A]Perm (append n ?) ?;
Refine cons_prem;
intros;Refine H;
intros;Refine cons_ih (append n k1);Refine insResidue;Immed;
Intros;Refine ex_hk;
Refine +3 ins_hkm1 [m:list A]EQ ? (cons b m);
Refine +3 reflEQ (insert a (cons b h1) k2);
Refine transPerm;Refine +1 perm_lhk;
Refine H1 [h:list A]Perm (append h k1) ?;
Refine consclPerm;
Refine perm_lhk1;
Save consclResidue;

Goal {h,k,l:list A}
     {res_hl:PermResidue h l}
     {res_kl:PermResidue k l}Perm h k;
```

```
intros;Refine res_hl;
intros;Refine res_kl;
intros;Refine inseqins;Refine +6 transEQ;Refine +8 symEQ;
Refine +7 ins_hkm;Refine +2 ins_hkm1;
intros;Refine H;intros;
Refine transPerm;Refine +1 perm_lhk;
Refine H2 [k1:list A]Perm (append ? k1) ?;
Refine symPerm;
Refine transPerm;Refine +1 perm_lhk1;
Refine H1 [h2:list A]Perm (append h2 ?) ?;
Refine transPerm;Refine +1 transPerm;
Refine +2 symPerm;Refine +2 appassoc;Refine +2 swapPerm;
Refine +2 symEQ(appassoc|?|?|?|?);Refine +2 swapPerm;
Refine consclPerm;
Refine transPerm;Refine +2 swapPerm;
Refine appassoc;Refine swapPerm;
intros;Refine H;
intros;Refine H1;intros h2eqh1 k1eqk2;
Refine transPerm;Refine +1 perm_lhk;
Refine h2eqh1 [h1:list A]Perm (append h1 ?) ?;
Refine symPerm;
Refine transPerm;Refine +1 perm_lhk1;
Refine k1eqk2 [k2:list A]Perm (append ? k2) ?;
Refine reflPerm;
intros;Refine H1;intros;
Refine transPerm;Refine +1 perm_lhk;
Refine H2 [h1:list A]Perm (append h1 ?) ?;
Refine symPerm;
Refine transPerm;Refine +1 perm_lhk1;
Refine H3 [k2:list A]Perm (append ? k2) ?;
Refine transPerm;Refine +1 transPerm;
Refine +2 symPerm;Refine +2 symEQ(appassoc|?|?|?|?);
Refine +2 swapPerm;
Refine +2 appassoc;Refine +2 swapPerm;
Refine consclPerm;
Refine transPerm;Refine +2 appassoc; Refine +2 swapPerm;
Refine appassoc;Refine reflPerm;
```

```
Save funopResidue;

Goal {l,m,h|list A}{perm_hyp:Perm l m}
     {eq_ahl:EQ (cons a h) l}PermResidue h m;
intros;Refine PermRrespR;Immed;
intros;Refine swapResidue;Immed;
intros;Refine consclResidue;Immed;
Refine insResidue (nil A);Immed;
Save transResiduelemma;

Goal {l,m|list A}(Perm (cons a l) (cons a m)) -> Perm l m;

intros;
[forallPerm : Rel (list A) (list A)
            = [l,m:list A]{h,k:list A}
              {eq1:EQ (cons a h) l}{eq2:EQ (cons a k) m}Perm h k];
Refine PermE forallPerm;Immed;
intros;Expand forallPerm;intros;
Refine reflPerm';Refine taillemma;Refine +3 eq_prem;Immed;
intros;Expand forallPerm;intros;
Refine symPerm;Refine sym_ih;Immed;
intros;Expand forallPerm;intros;
Refine funopResidue;
Refine +1 transResiduelemma lt_prem eq1;
Refine transResiduelemma (symPerm rt_prem) eq2;
Refine listind [l:list A]
          {m:list A}forallPerm (append l m) (append m l);
intros;Expand forallPerm;intros;
Refine reflPerm';
Refine taillemma;Refine +4 appnil;Immed;
intros;
Refine listind [m:list A]forallPerm (append ? m) (append m ?);
Intros h n eq1 eq2;
[keqn = symEQ (taillemma eq2 (reflEQ ?)):EQ k n];
[heqk = taillemma (transEQ eq1 (appnil|?|?)) (reflEQ ?):EQ h k];
Refine keqn;Refine heqk;Refine reflPerm;
Intros c n _ h j eq1 eq2;
[aeqb = headlemma eq1 (reflEQ ?) : EQ a b];
```

```
[aeqc = headlemma eq2 (reflEQ ?) : EQ a c];
[kcneqh : EQ (insert c k n) h = symEQ (taillemma eq1 (reflEQ ?))];
[nbkeqj : EQ (insert b n k) j = symEQ (taillemma eq2 (reflEQ ?))];
Refine nbkeqj;Refine symPerm;Refine kcneqh;
Refine aeqb [b:A]Perm (insert b ? ?) ?;
Refine aeqc [c:A]Perm ? (insert c ? ?);
Refine transPerm;Refine +2 swapPerm;
Refine transPerm;Refine +1 swapPerm;
Refine consclPerm;Refine swapPerm;
Intros b h j cons_prem _ k n eq1 eq2;
Refine (symEQ (taillemma eq1 (reflEQ ?)))[k:list A]Perm k n;
Refine (symEQ (taillemma eq2 (reflEQ ?)));
Refine cons_prem;
Refine reflEQ;Refine reflEQ;

Save heredPermlemma;

Discharge a;

[hereditary [S:Rel (list A) (list A)] =
          {l:list A}SubRel (appcl l S) S];

Goal hereditary Perm;
Refine listind [l:list A]SubRel (appcl l Perm) Perm;
Intros ___;Immed;
intros;Intros ___;Refine ih;Refine heredPermlemma;Immed;
Save heredPerm;

Goal {a,b|A}{l,m,n|list A}
    Perm (insert a l (insert b m n)) (insert b l (insert a m n));
intros;
Refine appclPerm;
Refine transPerm;
Refine +2 swapPerm (cons a n) (cons b m);
Refine consclPerm;
Refine transPerm;
Refine +1 swapPerm;
Refine transPerm;
```

```
Refine +2 swapPerm;
Refine consclPerm;
Refine swapPerm;
Save transposePerm;

Discharge A;
```

B.1.6 Sorting lemmas

We now turn to the theory of the predicate $Sorted$, for use in the example of insert sort we considered.

```
(* ``modules specification'' for sorting *)

[sorted:Prop];

[A|Type];
[Le:Rel A A]
[reflLe:refl Le]
[transLe:trans Le]
[antisymLe:{a,b|A}((Le a b)->(Le b a)-> (EQ a b))];

(* here's the specification *)

[Lelist [a:A] = listiter true ([b:A][P:Prop]and (Le a b) P)
              : Pred (list A)];

[Sorted = listrec true ([a:A][l:list A][P:Prop]and (Lelist a l) P)
        : Pred (list A)];

Goal {B|Type}{phi:Rel (list A) B}{n|B}{c|A -> (list A) -> B -> B}
               (phi (nil A) n) ->
               ({a|A}{l|list A}{b|B}(Sorted(cons a l))  ->
                        (phi l b) -> phi (cons a l) (c a l b))->
       {l:list A}(Sorted l) -> (phi l (listrec  n c l));
intros;
Refine listind [l:list A](Sorted l) -> phi l (listrec n c l);Immed;
```

```
intros;Immed;intros;Refine H3;
intros;Refine H1;Immed;Refine ih;Immed;
Save sortedlistinduction;
```

We now prove some lemmas about the specification, including Lemmas 4.3.1, 4.3.2, 4.3.3, 4.3.5, 4.3.6, 4.3.8, 4.3.9, 4.3.10 of Chapter 4.

```
Goal Sorted (nil A);
Intros;Immed;
Save nilSorted;

[b,c|A][m,n|list A];

Goal Lelist c (nil A);
Intros;Immed;
Save nilLelist;
Goal (Sorted (cons b n)) -> Sorted n ;
intros;Refine H;intros;Immed;
Save heredSortedlemma;
Goal (Sorted (cons b n)) -> Lelist b n;
intros;Refine H;intros;Immed;
Save SortedImpliesLelist;
Goal (Lelist c (cons b n)) -> (Lelist c n);
intros;Refine H;intros;Immed;
Save heredLelistlemma;

DischargeKeep b;

Goal (Lelist c (append m n)) -> (Lelist c n);
Refine listind [m|list A](Lelist c (append m n))->Lelist c n;
intros;Immed;
intros;Refine ih;Refine heredLelistlemma;Immed;
Save heredLelist1;
Goal (Lelist c (append m n)) -> (Lelist c m);
Refine listind [m|list A](Lelist c (append m n))->Lelist c m;
intros;Refine nilLelist;
intros;Refine H;
intros;Refine pair;Immed;Refine ih;Immed;
```

```
Save heredLelist2;
Goal (Lelist c m) -> (Lelist c n)  -> Lelist c (append m n);
intros;
Refine listind [m:list A](Lelist c m)->Lelist c (append m n);Immed;
intros;Immed;
intros;Refine H2;intros;
Refine pair;Refine +1 ih;Immed;
Save appclLelist;

DischargeKeep b;

Goal (Sorted (append m n)) -> (and (Sorted m) (Sorted n));
Refine listind [m|list A](Sorted (append m n))->
      and (Sorted m) (Sorted n);
intros;Refine pair;Refine nilSorted;Immed;
intros;Refine H;
intros;Refine ih;Immed;
intros;Refine pair;Refine pair;Immed;
Refine heredLelist2;Immed;
Save heredSorted;

Goal (Le c b) -> (Lelist b m) -> Lelist c m;
Refine listind [m|list A](Le c b)->(Lelist b m)->Lelist c m;
intros;Refine nilLelist;
intros;Refine H1;
intros;Refine pair;Immed;
Refine transLe;Immed;
Refine ih;Immed;
Save LelistIsMonotone;

Goal (Perm m n) -> (Lelist c m) -> (Lelist c n);

Refine PermPredE (Lelist c);
intros;Refine appclLelist;
Refine heredLelist1;Refine +2 heredLelist2;Immed;
intros;Refine cons_ih;Refine H;
intros;Refine pair;Refine +1 H3;Immed;
Save PermPreservesLelist;
```

```
DischargeKeep b;

Goal (Sorted (cons b m)) ->(Sorted (cons c n)) ->
                   (Perm (cons b m)(cons c n)) -> (Le b c);
intros;Refine fst;Refine Lelist b n;
Equiv Lelist b (cons c n);
Refine PermPreservesLelist;Immed;
Refine pair;Refine reflLe;
Refine SortedImpliesLelist;Immed;
Save SortedPermsHaveLeHeads;

DischargeKeep b;

Goal (Sorted (cons b m)) ->(Sorted (cons c n)) ->
                   (Perm (cons b m)(cons c n)) -> (EQ b c);
intros;Refine antisymLe;
Refine SortedPermsHaveLeHeads;Immed;
Refine SortedPermsHaveLeHeads;Immed;
Refine symPerm;Immed;
Save SortedPermsHaveEqualHeads;

Discharge b;

Goal {l,m|list A}{sortedl:Sorted l}{sortedm:Sorted m}
     {permlm:Perm l m}EQ l m;
Refine listind [l|list A]{m|list A}
   {sortedl:Sorted l}{sortedm:Sorted m}{permlm:Perm l m}EQ l m;
intros;Refine nilPerm;Immed;
intros;Refine lengthisahomomorphismcons;
       Refine +3 PermRespLength;Immed;Refine +1 reflEQ;
intros c eq;Refine (eq Sorted sortedm);Refine sortedl;
intros blek sortedk cletailm sortedtailm;
Refine transEQ;Refine +2 symEQ;
Refine +1 ih ? ? ? [m:list A]EQ ? (cons ? m);Immed;
Refine heredPermlemma;
Refine +1 symPerm;Refine +1 ?+2;Refine symPerm(eq ? permlm);
[beqc = SortedPermsHaveEqualHeads
```

```
         sortedl (eq ? sortedm) (eq ? permlm) : EQ b c];
Refine beqc [c:A]EQ ? (cons c ?);Refine reflEQ;
Save SortedPermsAreEqual;

Discharge A;
```

This concludes the basic lemmas. We have not included a substantial number of trivial lemmas in arithmetic, which we employed in the proof of the Chinese remainder theorem. They may safely be left as a substantial exercise to the patient reader.

B.2 Deliverables

We now present the LEGO code for the constructions in Chapter 3. By contrast with the foregoing, this is rather easier to comment. Indeed, a fanatical formalist might argue that Chapter 3 is merely a verbose and inaccurate account of the fully formal treatment here.

B.2.1 First-order deliverables

```
(* --- predicates and first-order deliverables  --- *)
[newdel1:Prop];

[SPEC_1 = <s:Type>s -> Prop];

(* a local context within which to work *)
[s,t,u,v,w | Type];
[S|Pred s][T|Pred t][U|Pred u][V|Pred v][W|Pred w];

(* the fundamental definitions *)
[Del1 [S:Pred s][T:Pred t][f:s->t] = {x|s}{pre:S x} T(f x)];
[del1 [S:Pred s][T:Pred t]         = <f:s->t>(Del1 S T f)];
```

```
DischargeKeep s;

(* a hack to assist equality of deliverables *)
[ext_del1 [FF:del1 S T] = [f = FF.1][F = FF.2]
          (([x:s]f x,[x:s][pre:S x]F pre):del1 S T)];

(* identities *)
[id_del1 [S:Pred s] = ([x:s]x, [x|s][p:S x] p) : del1 S S];

(* law of composition *)
Goal (del1 S T)->(del1 T U)->(del1 S U);
Intros FF GG #;
[f = FF.1][F = FF.2][g = GG.1][G = GG.2];
Refine [x:s]g(f x);
Intros;Refine G;Refine F;Immed;
Save compose_del1;

DischargeKeep s;

[FF|del1 S T][GG|del1 T U][HH|del1 U V];

Goal EQ (compose_del1 (id_del1 S) FF) (ext_del1 FF);
Refine reflEQ;
Save leftidentity_del1;

Goal EQ (compose_del1 FF (id_del1 T)) (ext_del1 FF);
Refine reflEQ;
Save rightidentity_del1;

Goal EQ (compose_del1 FF (compose_del1 GG HH))
         (compose_del1 (compose_del1 FF GG) HH);
Refine reflEQ;
Save associativity_del1;

(* basic pointwise construction of deliverables
   from underlying term calculus              *)
Goal  ({x:s}<y:t>(S x)->T y) -> del1 S T;
```

```
Intros family # x;Refine (family x).1;
Intros x _;Refine (family x).2;Immed;
Save pointwise_del1;

(* a trivial construction - every function yields a deliverable *)
[f:s->t];
[fstarPred [T:Pred t] = [x:s]T (f x)];

Goal del1 (fstarPred T) T;
Intros #;Refine [x:s]f x;
Intros;Immed;
Save functional_del1;

Discharge f;

(* logical inferences yield deliverables *)
[S'|Pred s][T'|Pred t];

Goal (SubPred S S')->del1 S S';
Intros subS # x;Immed;
Intros;Refine subS;Immed;
Save logical_del1;

Goal (SubPred S' S)->(SubPred T T')->
       (del1 S T)->del1 S' T';
Intros subS subT PHI #;
[phi = PHI.1][Phi = PHI.2];Immed;
Intros;Refine subT;
Refine Phi;Refine subS;Immed;
Save consequence_del1;

Discharge S';

Goal [f = FF.1][F = FF.2]
     EQ (ext_del1 FF)
        (compose_del1 (logical_del1|(fstarPred f T) F)
                      (functional_del1 f));
Refine reflEQ;
```

```
Save factorisation_del1;

Discharge FF;

(* a semi-terminal object *)
[Unit:Pred unit = [u:unit]EQ void u];

Goal {S:Pred s}del1 S Unit;
Intros _ # x;Refine void;
Intros x hyp;Refine reflEQ;
Save shriek_del1;

(* semi-cartesian (binary) products *)
[Product_del1 [S:Pred s][T:Pred t]
             = [xy:s#t]and (S xy.1) (T xy.2)];

DischargeKeep s;

(* the associated pairing function *)
[pair_fun  [f:s->t][g:s->u] = [x:s](((f x), g x) : t#u)];
[pair_del1 [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)
            : del1 S (Product_del1 T U))];

(* the associated projections *)
Goal {T:Pred t}{U:Pred u}del1 (Product_del1 T U) T;
Intros __#;Refine [yz:t#u]yz.1;
Intros;Refine fst pre;
Save pi1_del1;

Goal {T:Pred t}{U:Pred u}del1 (Product_del1 T U) U;
Intros __#;Refine [yz:t#u]yz.2;
Intros;Refine snd pre;
Save pi2_del1;

DischargeKeep s;
```

```
(* semi-exponentials *)
(* Del1 is the underlying predicate of the exponential object *)

Goal (del1 (Product_del1 S T) U) -> (del1 S (Del1 T U));
Intros FF #;[f = FF.1][F = FF.2];Refine [x:s][y:t](f(x,y));
Intros;Refine F;Refine pair;Immed;
Save lambda_del1;

Goal (del1 S (Del1 T U)) -> (del1 (Product_del1 S T) U) ;
Intros GG #;[g = GG.1][G = GG.2];Refine [p:s#t]g p.1 p.2;
Intros __;Refine pre;
intros __;Refine G;Immed;
Save uncurry_del1;

Goal {T:Pred t}{U:Pred u}del1 (Product_del1 (Del1 T U) T) U;
Intros __#;Refine [p:(t->u)#t][h = p.1][y = p.2]h y;
Intros p hyp;[h = p.1][y = p.2];Refine fst hyp;Refine snd hyp;
Save ev_del1;

DischargeKeep s;

(* Hayashi's equational conditions for an algebraic semi-ccc *)

[FF|del1 (Product_del1 S T) U];
[GG|del1 W S][HH|del1 W T][KK|del1 V W];

Goal EQ (compose_del1 KK (pair_del1 GG HH))
        (pair_del1 (compose_del1 KK GG)
                   (compose_del1 KK HH));
Refine reflEQ;
Save hayashi1;

Goal EQ (compose_del1 (pair_del1 GG HH) (pi1_del1 S T)) GG;
Refine reflEQ;
KillRef; (* this doesn't quite work *)

Goal EQ (compose_del1 (pair_del1 GG HH) (pi1_del1 S T))
        (ext_del1 GG);
```

```
Refine reflEQ;
Save hayashi2i;

Goal EQ (compose_del1 (pair_del1 GG HH) (pi2_del1 S T))
        (ext_del1 HH);
Refine reflEQ;
Save hayashi2ii;

Goal EQ (compose_del1
          (pair_del1 (compose_del1 GG (lambda_del1 FF)) HH)
           (ev_del1 T U))
        (compose_del1 (pair_del1 GG HH) FF);
Refine reflEQ;
Save hayashi4i;

Goal  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));
Refine reflEQ;
Save hayashi4ii;

Goal EQ (shriek_del1 V) (compose_del1 KK (shriek_del1 W));
Refine reflEQ;
Save hayashi5;

Goal  EQ (ev_del1 T S)
         (compose_del1
            (pair_del1 (pi1_del1 (Del1 T S) T)
                       (pi2_del1 (Del1 T S) T))
            (ev_del1 T S));
Refine reflEQ;
Save hayashi6;

Discharge s;
```

B.2.2 First-order deliverables for sums, natural numbers and polymorphic lists

```
[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];

[s,t,u|Type][S|Pred s][T|Pred t][U|Pred u];

[Sum_del1 = [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))];

Goal (del1 S U)->(del1 T U)->del1 (Sum_del1 S T) U;
intros FF GG;[f=FF.1][F=FF.2][g=GG.1][G=GG.2];
Intros # c;
Refine case f g c;
Intros;
Refine when [c:sum_type s t](Sum_del1 S T c)->U (case f g c);
Immed;

intros a h;Refine h;
intros e;Refine e;
intros a conj;
Refine fst conj [c:sum_type s t]U (case f g c);
```

```
Refine F;Refine snd conj;
intros e;Refine e;
intros b abs;
Refine (fst abs)
  (when ([_:sum_type s t]Prop) ([x:s]false) ([y:t]true));
Intros;Immed;

intros b h;Refine h;
intros e;Refine e;
intros a abs;
Refine (fst abs)
  (when ([_:sum_type s t]Prop) ([x:s]true) ([y:t]false));
Intros;Immed;
intros e;Refine e;
intros b conj;
Refine fst conj [c:sum_type s t]U (case f g c);
Refine G;Refine snd conj;

Save case_del1;

Discharge s;

(* first-order deliverables for natural numbers *)

[Nat = [n:nat]true];
[NAT = (nat,Nat):SPEC_1];

[t|Type];
[T|Pred t];

Goal {ZZ:del1 Unit T}{SS:del1 T T}del1 Nat T;
intros;[z = ZZ.1 void][Z = ZZ.2][s = SS.1][S = SS.2];
Refine pointwise_del1;
intros n #;Refine natiter z s n;
Refine natind [n:nat](Nat n)->T (natiter z s n);
intros;Refine Z;Refine voidunique;
intros;Refine S;Refine ih;
Intros;Immed;
```

```
Save natiter_del1;

Goal {ZZ:del1 Unit T}{SS:del1 Nat (Del1 T T)}del1 Nat T;
intros;[z = ZZ.1 void][Z = ZZ.2][s = SS.1][S = SS.2];
Refine pointwise_del1;
intros n #;Refine natrec z s n;
Refine natind [n:nat](Nat n)->T (natrec z s n);
intros;Refine Z;Refine voidunique;
intros;Refine S;Refine +1 ih ?+0;
Intros;Immed;
Save natrec_del1;

Discharge t;

(* first-order list deliverables *)

[listdel1:Prop];

[s,t|Type];
[S|Pred s][T|Pred t];

[Listof [S:Pred s] = listiter true ([a:s][p:Prop]and (S a) p)
                   : Pred (list s)];

Goal (del1 Unit  T) ->
      (del1 S (Del1 T T)) ->
       (del1 (Listof S) T);
intros NN CC;
[n = NN.1 void][N = NN.2];
[c = CC.1][C = CC.2];
Intros #;Refine listiter n c;
Refine listind [l:list s]{listofSl:Listof S l}T (listiter n c l);
intros;Refine N;Refine voidunique;
intros;Refine listofSl;
intros;Refine C;Refine +1 ih;Immed;
Save listiter_del1;

Goal (del1 Unit  T) ->
```

```
     (del1 S (Del1 (Listof S) (Del1 T T))) ->
     (del1 (Listof S) T);
intros NN CC;
[n = NN.1 void][N = NN.2];
[c = CC.1][C = CC.2];
Intros #;Refine listrec n c;
Refine listind [l:list s]{listofSl:Listof S l}T (listrec n c l);
intros;Refine N;Refine voidunique;
intros;Refine listofSl;
intros;Refine C;Refine +2 ih;Immed;
Save listrec_del1;

Discharge s;
```

B.2.3 Second-order deliverables

```
(* Relations and second order deliverables *)

[del2s:Prop];

[s,t,u,v,w|Type];
[S|Pred s][T|Pred t][U|Pred u][V|Pred v][W|Pred w];
[P|Rel s t][Q|Rel s u][R|Rel s v][O|Rel s w];

(* the fundamental definitions *)
[Del2 [S:Pred s][Q:Rel s t][R:Rel s u][f:s->t->u]
     =  {x|s}{h:S x}{y|t}{pre:Q x y} R x (f x y)];
[del2 [S:Pred s][Q:Rel s t][R:Rel s u]
     = <f:s->t->u>(Del2 S Q R f)];

DischargeKeep s;

(* a hack to assist equality of deliverables *)
[ext_del2 [FF:del2 S P Q] = [f = FF.1][F = FF.2]
   (([x:s][y:t]f x y,
     [x|s][h:S x][y|t][pre:P x y]F h pre):del2 S P Q)];
```

```
(* identities *)
[id_del2 [S:Pred s][R:Rel s t] =
         ([x:s][y:t]y,
          [x|s][h:S x][y|t][pre:R x y]pre):del2 S R R];

(* law of composition *)
Goal (del2 S P Q) -> (del2 S Q R) -> (del2 S P R);
Intros FF GG #;
[f = FF.1][F = FF.2][g = GG.1][G = GG.2];
Refine [x:s][y:t]g x (f x y);
Intros;Refine G;Refine +1 F;Immed;
Save compose_del2;

DischargeKeep s;

[FF|del2 S P Q][GG|del2 S Q R][HH|del2 S R O];

Goal EQ (compose_del2 (id_del2 S P) FF) (ext_del2 FF);
Refine reflEQ;
Save leftidentity_del2;

Goal EQ (compose_del2 FF (id_del2 S Q)) (ext_del2 FF);
Refine reflEQ;
Save rightidentity_del2;

Goal EQ (compose_del2 FF (compose_del2 GG HH))
         (compose_del2 (compose_del2 FF GG) HH);
Refine reflEQ;
Save associativity_del2;

(* basic construction of deliverables
   from underlying term calculus      *)
Goal  ({a:s}{b:t}
       <c:u>{h:S a}{pre:P a b}Q a c)->del2 S P Q;
Intros family # a b;Refine (family a b).1;
Intros a _ b _;Refine (family a b).2;Immed;
Save pointwise_del2;
```

```
(* basic tool in lifting structure from del1 to del2 *)
Goal ({x:s}<fx:t->u>{h:S x}Del1 (P x) (Q x) fx) ->
         del2 S P Q;
Intros family # x y;Refine (family x).1;Immed;
Intros;Refine (family x).2;Immed;
Save family_of_del1_to_del2;

Goal (del2 S P Q) ->
       {x:s}<fx:t->u>{h:S x}Del1 (P x) (Q x) fx;
Intros FF x #;
[f = FF.1][F = FF.2];Refine f;Immed;
Save del2_to_family_of_del1;

Goal (del2 S P Q) ->
       {x|s}{h:S x}del1 (P x) (Q x);
Intros FF __#;
[f = FF.1][F = FF.2];Refine f;Refine +1 F;Immed;
Save del2_to_family_of_del1';

(* a trivial construction - every function yields a deliverable *)
[f:s->t->u];
[fstarRel2 [P:Rel s u] = [x:s][y:t]P x (f x y):Rel s t];

Goal del2 S (fstarRel2 Q) Q;
Intros #;Refine f;
Intros;Immed;
Save functional_del2;

Discharge f;

(* logical inferences yield deliverables *)
[S'|Pred s][P'|Rel s t][Q'|Rel s u];

Goal ({x|s}(S x)->(SubPred (P x) (P' x)))
         ->del2 S P P';
Intros subSP # x y;Immed;
Intros;Refine subSP;Immed;
Save logical_del2;
```

```
Goal (SubPred S' S)->(SubRel P' P)->(SubRel Q Q')->
             (del2 S P Q)  ->  del2 S' P' Q';
Intros subS subP subQ PHI #;
[phi = PHI.1][Phi = PHI.2];Refine phi;
Intros;Refine subQ;Refine Phi;
Refine subS;Refine +1 subP;Immed;
Save consequence_del2;

Discharge S';

(* a factorisation theorem *)
Goal [f = FF.1][F = FF.2]
     EQ (ext_del2 FF)
        (compose_del2 (logical_del2|(fstarRel2 f Q) F)
                          (functional_del2 f));
Refine reflEQ;
Save factorisation_del2;

Discharge FF;

(* structure inherited from del1 *)
[Product_del2 [Q:Rel s t][R:Rel s u]
             = [x:s]Product_del1 (Q x)(R x):Rel s t#u];

DischargeKeep s;

Goal (del2 S P Q) -> (del2 S P R) ->
          (del2 S P (Product_del2 Q R));
Intros FF GG #;
[f = FF.1][F = FF.2][g = GG.1][G = GG.2];
Refine [x:s]pair_fun (f x)(g x);
Intros ____;
Refine pair;Refine F;Refine +2 G;Immed;
Save pair_del2;

Goal del2 S (Product_del2 Q R) Q;
Intros # x p;Refine pi1;Immed;
```

```
Intros ____;Refine fst pre;
Save proj1_del2;

Goal del2 S (Product_del2 Q R) R;
Intros #;intros x p;Refine pi2;Immed;
Intros ____;Refine snd pre;
Save proj2_del2;

(* ... *)
(* ... *)
[Exp_del2 [Q:Rel s t][R:Rel s u]
         = [x:s]Del1 (Q x) (R x):Rel s t->u];

DischargeKeep s;

Goal (del2 S (Product_del2 P Q) R) -> del2 S P (Exp_del2 Q R);
intros FF;[f = FF.1][F = FF.2];
Intros # x y z;
Refine f x;Refine (y,z);
Intros ____ z q;
Refine F;Immed;Refine pair;Immed;
Save lambda_del2;

Goal (del2 S (Product_del2 (Exp_del2 Q R) Q) R);
Intros # x p;[f = p.1][y = p.2];Refine f y;
Intros ____;Refine fst pre;Refine snd pre;
Save ev_del2;

(* ... *)
(* ... *)

(* reindexing between the fibres *)
[del1starRel [KK:del1 V S][P:Rel s t]
            = [k = KK.1][a:v][y:t]P (k a) y:Rel v t];

DischargeKeep s;

Goal {KK:del1 V S}(del2 S P Q) ->
```

```
       del2 V (del1starRel KK P) (del1starRel KK Q);
Intros KK FF #;
[k = KK.1][K = KK.2][f = FF.1][F = FF.2];
Refine compose f k;
Intros;Refine F;Refine K;Immed;
Save pullback_del2_along_del1;

DischargeKeep s;

(* preserves all the structure on the nose *)
(* ... *)
(* ... for example ... *)

[KK:del1 V S];
Goal EQ (del1starRel KK (Exp_del2 P Q))
        (Exp_del2 (del1starRel KK P)(del1starRel KK Q));
Refine reflEQ;

(* ... *)
(* ... *)

[FF:del2 S (Product_del2 P Q) R];

(* here we see the use of type casting, so
   that this goal is even well-typed at all *)

Goal EQ (pullback_del2_along_del1 KK (lambda_del2 FF))
        (lambda_del2 (pullback_del2_along_del1 KK FF
          :del2 V (Product_del2 (del1starRel KK P)
                                (del1starRel KK Q))
                  (del1starRel KK R)));
Refine reflEQ;

Discharge s;
```

B.2.4 Second-order deliverables for natural numbers and lists

```
(* second-order deliverables for natural numbers *)
[natdel2:Prop];

[t|Type][R|Rel nat t];

[zstarRel [R:Rel nat t] = [n:nat][y:t]R zero y :Rel nat t];
[sstarRel [R:Rel nat t] = [n:nat][y:t]R (succ n) y :Rel nat t];

Goal (del2 Nat R (sstarRel R)) -> del2 Nat (zstarRel R) R;
Intros SS;[s = SS.1][S = SS.2];
Intros # n z;Refine natrec;Immed;
Intros n _ z zR;
Refine natind [n:nat]R n (natrec z s n);Immed;
intros k _;Refine S;Immed;
Save natrec_del2;

Discharge t;

(* some second-order deliverables for lists *)
[listdel2:Prop];

[s,t|Type];

[nstarRel [R:Rel (list s) t]
         = [l:list s][y:t]R (nil s) y : Rel (list s) t];
[cstarPred [x:s][P:Pred (list s)]
         = [l:list s]P (cons x l) : Pred (list s)];
[cstarRel [x:s][R:Rel (list s) t]
         = [l:list s][y:t]R (cons x l) y : Rel (list s) t];

[R|Rel (list s) t];

Goal ({x:s}del2 (univPred|(list s)) R (cstarRel x R)) ->
(* ---------------------------------------------------- *)
         del2 (univPred|(list s)) (nstarRel R) R;
```

```
intros family;
[c = [x:s](family x).1][C = [x:s](family x).2];
Intros # l n;Refine listrec;Immed;
Intros l _ n Rnil;
Refine listind [l:list s]R l (listrec n c l);Immed;
intros;Refine C b;Immed;

Save Listrec_del2;

[depListof [Phi:Rel s (list s)]
          = listrec true
                    ([x:s][l:list s][phi:Prop]
                        and (Phi x l) phi)
          : Pred (list s)];

[Phi|Rel s (list s)];

Goal ({x:s}del2 (cstarPred x (depListof Phi)) R (cstarRel x R)) ->
(* -------------------------------------------------------------- *)
                  del2 (depListof Phi) (nstarRel R) R;

intros family;
[c = [x:s](family x).1][C = [x:s](family x).2];
Intros # l n;Refine listrec;Immed;
Intros l lhyp n Rnil;
Refine listind [l:list s]{indhyp:depListof Phi l}R l (listrec n c l);
Immed;
(* base case *)
intros;Immed;
(* step case *)
intros;Refine C b;Refine +1 ih;Refine +1 snd indhyp;Immed;

Save depListrec_del2;

Discharge s;
```