

Deliverables: a categorical approach  
to program development  
in type theory

James Hugh McKinna

Doctor of Philosophy  
University of Edinburgh  
1992

*le hasard ne favorise que les esprits préparés*

*Pasteur*

# Abstract

This thesis considers the problem of program correctness within a rich theory of dependent types, the Extended Calculus of Constructions (ECC). This system contains a powerful programming language of higher-order primitive recursion and higher-order intuitionistic logic. It is supported by Pollack's versatile LEGO implementation, which I use extensively to develop the mathematical constructions studied here.

I systematically investigate Burstall's notion of *deliverable*, that is, a program paired with a proof of correctness. This approach separates the concerns of programming and logic, since I want a simple program extraction mechanism. The  $\Sigma$ -types of the calculus enable us to achieve this. There are many similarities with the subset interpretation of Martin-Löf type theory.

I show that deliverables have a rich categorical structure, so that correctness proofs may be decomposed in a principled way. The categorical combinators which I define in the system package up much logical book-keeping, allowing one to concentrate on the essential structure of algorithms.

I demonstrate our methodology with a number of small examples, culminating in a machine-checked proof of the Chinese remainder theorem, showing the utility of the deliverables idea. Some drawbacks are also encountered.

I consider also semantic aspects of deliverables, examining the definitions in an abstract setting, again firmly based on category theory. The aim is to overcome the clumsiness of the language of categorical combinators, using

dependent type theories and their interpretation in fibrations. I elaborate a concrete instance based on the category of sets, which generalises to an arbitrary topos. In the process, I uncover a subsystem of ECC within which one may speak of deliverables defined over the topos. In the presence of enough extra structure, the interpretation extends to the whole of ECC. The wheel turns full circle.

# Acknowledgments

My heartfelt thanks:

- firstly, to my supervisor, Rod Burstall, for his sense of humour, as well as all his encouragement and sage advice; I have enjoyed a very creative partnership with him throughout my studies in Edinburgh;
- to the members of the LEGO club, especially Randy Pollack and Zhaohui Luo, for all the many ways they have supported this work, which would not have been conceivable without Randy's LEGO implementation of Luo's Extended Calculus of Constructions;
- to Eugenio Moggi, Bart Jacobs, Herman Geuvers, for illuminating discussions of fibrations and type theory;
- to Roy Dyckhoff, and Fer-Jan de Vries, who are still waiting, and the other folk of the forty-sixth Peripatetic Seminar on Sheaves and Logic, for their forbearance during a hasty rehearsal of Chapter 5;
- to Wes Phoa, unwittingly responsible for my work on the Chinese remainder theorem, and for being convinced by the thought experiment of Chapter 3; I have enjoyed many fruitful discussions with him in the latter stages of this work;
- to the Science and Engineering Research Council, for funding this research, and providing the spur to write up promptly; and to Harold Simmons,

for all the many meetings and opportunities sponsored by the LogFIT Initiative under his aegis;

- to the readers of various drafts of this work, Stuart Anderson, Philippa Gardner, Healf Goguen, Stephen Gilmore, Jane Hillston, Claire Jones, Wes Phoa and Alan Turland; their thoughtful criticism and attention are more than I either deserve or could have expected;
- to Claire Jones, for her typesetting expertise;
- to the secretaries and computing support staff, for making research possible in the Computer Science department here at Edinburgh;
- for emotional support, laughter, and tears, to Lesley Parker, lately of SACS, and the Salisbury Centre Men's group;
- to my long-suffering flat-mates, Jennifer, Ruth, and latterly Jonathan;
- to all the friends, here and elsewhere, who have kept me going, even when I have neglected our friendship;
- to Philippa, for finishing before me;
- to my parents, to whom I owe special thanks for their unstinting kindness and support throughout my years as a student;
- and to Gill, for her love and companionship.

# Declaration

I declare that this thesis was composed by myself, and the work contained in it is my own except where otherwise stated. A draft paper, outlining the ideas of Chapter 3, already appeared as [13].

James Hugh McKinna

# Table of Contents

Abstract . . . . .	ii
Acknowledgments . . . . .	iv
Declaration . . . . .	v
Contents . . . . .	vi
<b>1. Introduction</b>	<b>1</b>
1.1 Formal methods . . . . .	1
1.2 Foundations: type theory and category theory . . . . .	4
1.3 Overview of the thesis . . . . .	6
1.4 Related work . . . . .	10
1.5 Prerequisites . . . . .	14
<b>2. Type theoretic preliminaries: ECC and LEGO</b>	<b>15</b>
2.1 Review of ECC . . . . .	15
2.1.1 The term calculus . . . . .	16
2.1.2 The derivable judgments of ECC . . . . .	17
2.1.3 Proof-theoretic properties of the calculus . . . . .	21
2.1.4 Equality and conversion . . . . .	22

2.2	Review of LEGO . . . . .	23
2.2.1	Syntax . . . . .	23
2.2.2	The theorem prover . . . . .	24
2.2.3	Additional features of the LEGO system . . . . .	25
2.2.4	Equality and Conversion <i>bis</i> . . . . .	28
2.2.5	A sample derivation . . . . .	28
<b>3.</b>	<b>First-order and second-order deliverables</b>	<b>35</b>
3.1	Definition and properties of first-order deliverables . . . . .	36
3.1.1	Specifications . . . . .	36
3.1.2	First-order deliverables . . . . .	37
3.1.3	Equality of deliverables . . . . .	40
3.1.4	Semi-structure in categories . . . . .	43
3.1.5	Identities and composition . . . . .	46
3.1.6	Semi-Terminal object . . . . .	48
3.1.7	Binary semi-products . . . . .	50
3.1.8	Binary semi-coproducts . . . . .	51
3.1.9	Semi-exponentials . . . . .	53
3.1.10	Semi-pullbacks and the internalisation of equality . . . . .	54
3.1.11	A factorisation system on $\mathbf{del}_1$ . . . . .	56
3.1.12	A consequence rule . . . . .	57
3.1.13	Pointwise construction . . . . .	57
3.1.14	Inductively defined types . . . . .	57

3.2	Second-order deliverables . . . . .	65
3.2.1	A thought experiment . . . . .	65
3.2.2	Basic definitions . . . . .	67
3.2.3	Each $\mathbf{del}_2\mathcal{S}$ is a semi-ccc . . . . .	70
3.2.4	$\mathbf{del}_2$ : an indexed category over $\mathbf{del}_1$ . . . . .	74
3.2.5	Pullback functors . . . . .	74
3.2.6	$\mathbf{del}_2$ has $\mathbf{del}_1$ -indexed sums and products . . . . .	77
3.2.7	Second-order deliverables for natural numbers and lists . . . . .	78
3.2.8	Lists . . . . .	80
<b>4.</b>	<b>Examples</b> . . . . .	<b>85</b>
4.1	Division by two . . . . .	86
4.1.1	The mathematical specification . . . . .	86
4.1.2	The correctness proof of our intended algorithm . . . . .	87
4.1.3	The development in terms of deliverables . . . . .	88
4.2	Finding the minimum of a list . . . . .	93
4.2.1	The mathematical specification . . . . .	94
4.2.2	The correctness proof of our intended algorithm . . . . .	95
4.2.3	The development in terms of deliverables . . . . .	95
4.3	Insert sort . . . . .	100
4.3.1	The mathematical specification . . . . .	100
4.3.2	A correctness proof for insert sort . . . . .	102
4.3.3	The proof recast in terms of deliverables . . . . .	105

4.3.4	The completed development in terms of deliverables . . .	107
4.4	The Chinese remainder theorem . . . . .	110
4.4.1	The mathematical specification . . . . .	110
4.4.2	A development in terms of deliverables . . . . .	114
<b>5.</b>	<b>Abstract deliverables</b>	<b>117</b>
5.1	Construction of first order abstract deliverables . . . . .	118
5.1.1	Hyperdoctrines . . . . .	118
5.1.2	A programming language with assertions . . . . .	121
5.2	A topos-theoretic model . . . . .	122
5.2.1	Introduction . . . . .	122
5.2.2	Modelling dependent types (following Hyland, Pitts) . . .	125
5.2.3	Categories with fibrations . . . . .	125
5.2.4	Interpreting types and terms . . . . .	126
5.2.5	Dependent products in a category with fibrations . . . . .	129
5.2.6	Propositions and types in $\mathcal{C}\mathcal{C}^+$ . . . . .	130
5.3	The model . . . . .	130
5.4	Definition of $\mathcal{B}$ . . . . .	131
5.4.1	Properties of $\mathcal{B}$ . . . . .	131
5.4.2	Definition of the category with fibration $\mathcal{F}_{type}$ over $\mathcal{B}$ . . .	133
5.4.3	Substitution . . . . .	134
5.4.4	Dependent products . . . . .	136
5.4.5	Definition of the category with fibrations $\mathcal{F}_{prop}$ over $\mathcal{B}$ . . .	136

5.4.6	$\mathcal{F}_{prop}$ is included in $\mathcal{F}_{type}$ . . . . .	137
5.4.7	Propositions yield types . . . . .	137
5.4.8	Dependent products . . . . .	138
5.5	Consistency of the model . . . . .	138
5.6	Proof-irrelevance . . . . .	139
5.6.1	Extensionality . . . . .	139
5.7	A model of Luo's ECC . . . . .	140
5.7.1	Sums . . . . .	140
5.7.2	Type universes . . . . .	141
<b>6.</b>	<b>Further work and conclusions</b>	<b>142</b>
6.1	Partial equivalence relations and observational equivalence . . .	142
6.2	Data abstraction . . . . .	144
6.3	Parametricity and second-order $\lambda$ -calculus . . . . .	145
6.4	Extraction and realisability . . . . .	146
6.5	Partial functions in type theory . . . . .	147
6.6	Pragmatics . . . . .	148
6.7	Conclusions . . . . .	149
<b>A.</b>	<b>Fibrations</b>	<b>150</b>
A.1	Basic definitions . . . . .	150
A.2	Naturality and the Beck-Chevalley condition in categories with fibrations . . . . .	153

<b>B. LEGO code relevant to this thesis</b>	<b>157</b>
B.1 Basics . . . . .	157
B.1.1 Basic logic . . . . .	157
B.1.2 Basic datatypes: unit, booleans, and naturals . . . . .	160
B.1.3 The calculus of relations . . . . .	163
B.1.4 Polymorphic lists . . . . .	164
B.1.5 On permutation . . . . .	166
B.1.6 Sorting lemmas . . . . .	176
B.2 Deliverables . . . . .	180
B.2.1 First-order deliverables . . . . .	180
B.2.2 First-order deliverables for sums, natural numbers and polymorphic lists . . . . .	186
B.2.3 Second-order deliverables . . . . .	189
B.2.4 Second-order deliverables for natural numbers and lists .	195
B.3 Examples . . . . .	197
B.3.1 A simple example . . . . .	197
B.3.2 Division by two . . . . .	198
B.3.3 Minimum finding in a list . . . . .	199
B.3.4 Insert sort . . . . .	208
B.3.5 The Chinese remainder theorem . . . . .	212
<b>Bibliography</b>	<b>229</b>

# Chapter 1

## Introduction

This thesis concerns a number of insights and techniques obtained from category theory, applied to the study of programs and their correctness proofs in constructive type theory. As such, it provides a modest exploration of the interaction of two disciplines at the heart of much recent research in theoretical computer science. It is largely based on the unpublished ideas of Rod Burstall [12].

### 1.1 Formal methods

For thirty years, researchers and industrialists of all persuasions have pursued the Grail of verified programming. No longer may we be content to regard programming as a craft, erecting the medieval cathedrals we now strive so strenuously and expensively to preserve [23,24].

This thesis represents an experiment in this field in the context of an integrated environment for the development of mathematical proofs, which also contains a powerful programming language. As such, the environment furnished by Pollack's LEGO implementation of the Extended Calculus of Constructions,

represents a good experimental tool for investigation of new methods, since the features of computation and logic are combined in one system.

One of the strengths of formal reasoning on a machine is that we can relegate much logically dull activity — e.g. the computing of substitutions — to the proof tool, using unification and other devices, without the user having to manage the tedious book-keeping. Set against that, however, the activity of formal proof, as has been observed many times by various authors and research workers, carries with it a huge burden of information typically suppressed in what mathematicians regard as acceptable proofs. One of the aims, therefore, of this thesis will be to develop languages of sufficient expressiveness to enable us to construct proven programs in a well-structured way, mirroring the construction of programs themselves.

There are two contrasting approaches to the formal development of correct programs. Classically, we typically write a program and produce a separate proof of its correctness. This approach seems unsatisfactory in that the proof is separate from the program. It is more acceptable in a formalism such as Floyd-Hoare assertions [40], in which local correctness statements are attached to program phrases, or Dijkstra's use of predicate transformers [22,25], where we may derive appropriate guards by calculating weakest preconditions.

A constructive, or type-theoretic, approach is to prove a mathematical theorem in some constructive logic, and extract from it a program. This method is attractive at first sight but there is considerable difficulty in separating the algorithmic part of the proof from the correctness part. We must somehow annotate the proof in such a way as to mark those parts which are computationally relevant, as opposed to those which serve to demonstrate the correctness of our extracted program.

If we look for a constructive proof of a statement of the form

$$\forall x. I(x) \implies \exists y. O(x, y)$$

then what emerges is a function which, given an individual  $x$  and a proof of  $I(x)$ , produces a pair consisting of an individual  $y$  and a proof that it satisfies  $O(x, y)$ . In other words we cannot get the result  $y$  without providing the proof of  $I(x)$ . It seems that if we try to develop a substantial program this way we need to handle proofs all the time as we attempt to calculate values: the proof and the computation are inextricably mixed.

Burstall's idea is to consider instead pairs  $(f, p)$ , where  $f$  is a function from individuals to individuals and  $p$  is a proof that

$$\forall x. I(x) \implies O(x, f(x))$$

In this thesis, we consider the systematic development of proven programs within type theory using this idea of a *deliverable*.

In this approach, there is a clear separation of the computation from the proof, but we have made no gain if we have to construct  $f$  and  $p$  separately. We should examine the possibility of a *compositional* development of such pairs. But here we face an obstacle, namely the asymmetry of  $I$  and  $O$  in such a definition.

A first approximation is simply to consider unary predicates, rather than relations, that is we examine pairs  $(f, p)$ , where  $f$  is a function and  $p$  is a proof that

$$\forall x. I(x) \implies O(f(x)).$$

Burstall observed that these deliverables form a category with additional structure. This enables us to construct deliverables in the style of categorical combinators for the  $\lambda$ -calculus [21].

In a subsequent paper [13], Burstall and the author sketched how to extend this framework to allow us to speak once more of relations between the input and output, while preserving the categorical character of the construction. These ideas are elaborated in Section 3.2 of this thesis.

The methodology we propose, however, cannot be regarded as a panacea for the problems of verified programming. It does not eliminate the essential difficulty in furnishing proofs, although it does allow the proofs to be structured in a rather less *ad hoc* way. So in some sense, we have localised the difficulty, by insisting that the structure of an intended (if only partially elaborated) algorithm underlie the correctness proof. Our experience with the Chinese remainder theorem in Chapter 4 seems to bear this out. A major drawback seems to be the crudeness of the language of categorical combinators within which we work. A significant advance would be to develop a type-theoretic language for deliverables. The system we present in Chapter 5, derived from semantic considerations of deliverables, should be regarded as a preliminary step in this direction.

## 1.2 Foundations: type theory and category theory

The last twenty years or so has seen a revitalisation of intuitionistic mathematics with the emergence of various type theories, focusing on the notion of construction, and its relationship with proof theory and the  $\lambda$ -calculus. Starting with Howard's account of "propositions as types" [42], a constructive reading of the logical connectives and the semantics of proofs has revealed close connections between intuitionistic mathematics and functional programming. These ideas were elaborated by Scott [98], de Bruijn and his co-workers in the AUTOMATH project [11], and most notably by Martin-Löf, in a succession of predicative systems which bear his name [69,70,71,79]. Central to these systems is the idea of *dependent* type or family of types. In a parallel development [32], Girard considered impredicative systems related to simple type theory [14]. Coquand and Huet [17,18] were able to unite dependent types and impredicativity in their Calculus of Constructions,  $\mathcal{CC}$ . Finally Luo synthesised Martin-Löf's idea

of a predicative hierarchy of type universes with the impredicative Calculus of Constructions in his Extended Calculus of Constructions, ECC [58,59].

In this context, the notion of deliverable, considered in the simple case of a function respecting unary predicates, does not appear to be new. Already in the proof of normalisation for his 1973 theory [69], Martin-Löf gave an interpretation (due to Hancock) of the judgments of the theory in a model of closed terms, where types were interpreted as sets of closed terms, subject to some predicate being satisfied<sup>1</sup>. The interpretation is given by a grand simultaneous inductive definition on the derivable judgments of the theory. The particular clause for the function space embodies the idea of deliverables. Indeed, this remark applies to any normalisation proof based on the reducibility method. The reducibility predicate  $\phi_A$ , defined by induction on the type  $A$ , with the function clause essentially  $\phi_{A \rightarrow B}(f) = \forall a:A. \phi(a) \Rightarrow \phi(fa)$ . Martin-Löf's interpretation has re-emerged in recent years in giving an account of subset types within Martin-Löf type theory, of which we shall have more to say below.

A parallel development over a similar period has been the emergence of internal languages for categories, which at their most crude are simply extensional type theories with unary function symbols. The presence of structure (images, cartesian products, closedness) yields further strengthening of the type-formation and derivation rules in the internal logic. Much effort was expended in the 'Seventies in the systematic exploration of the interaction between logical properties and categorical structure, culminating in the elaboration of the internal logic of toposes [28,49,10,53,5]. The most significant development in this regard must be Lawvere and Tierney's observation that the *representability* of the notion of subobject (via the so-called subobject classifier) is what makes set theory possible. Significant for later authors in the development of

---

<sup>1</sup>I am grateful to Furio Honsell for pointing out this connection to me.

the internal language is the realisation, just as Russell and Church had noted in rather different terms, that all the impredicative definitions of higher-order logic — in an extensional, intuitionistic setting — are obtainable from the small complete internal poset  $\Omega$ , whose object of objects is the subobject classifier [49, Ch.3].

Subsequent research in categorical logic has considered the richer type systems above, with fibrations emerging as the central unifying concept [102,6,45, 26,105,84,48]. We shall return to these issues in Chapter 5 below.

Type-theoretically, the notion of “subset”, or subset type has been much less clearly defined. Probably the most closely argued and theoretically satisfying has been the work of the Göteborg group [79]. One of the troubling (and desirable!) features is the role that subsets play in suppressing propositional information in programs. Within a topos, the use of subobjects to give a semantics to judgments of the logic ensures that propositions have at most one proof, but in a  $\lambda$ -calculus with propositions as types, we have possibly many, yielding a potentially huge proliferation in the dependence of functions on proofs. Our treatment using deliverables follows the path of systematically developing proofs of correctness, in which computational information is kept distinct from propositional information. The presence of  $\Sigma$ -types allows us to do this.

### 1.3 Overview of the thesis

Chapter 2 sets out the type-theoretic preliminaries we require in the study of this approach to program correctness. Luo’s Extended Calculus of Constructions, ECC [58,59], is sketched as the basic framework of types and terms. The calculus is intended to provide a unified account of dependent type theory which brings together a theory of predicative datatypes and impredicative higher-order logic.

The separation of datatypes from propositions is at the heart of the deliverables idea. The presence of  $\Sigma$ -types in the calculus allows us to maintain these distinctions in a structured way. We discuss a number of features of the calculus, including a simple lemma on equality, which is used repeatedly in verifying basic properties of deliverables. A number of derivations are given, illustrating the expressive power of the calculus. We discuss a small example, concerning the even numbers, in the context of our general view of program specification.

We also discuss Pollack's LEGO implementation of the Extended Calculus of Constructions [85,64]. This is a very versatile system, which allows us to develop proofs and constructions by refinement. The system also achieves great utility through a powerful mechanism for extending the basic calculus with definitions. We describe this, and a number of other features of the system, which was used extensively in this research. In particular, *most of the mathematical proofs in this thesis have been checked in the system*, with the exception of those concerning the categorical constructions in Chapter 5. Often, the proof term exhibiting a mathematical construction in the text is illustrated by the corresponding term in the LEGO system. We work through a sample derivation of the even number example, to give the reader an introduction to reasoning in the system. More elaborate examples are discussed in Chapter 4.

The system has recently been extended to allow for inductive definitions of datatypes in the style of Martin-Löf type theory. This allows direct access to a powerful programming language of higher-order primitive recursion. We illustrate the use of such a language with the examples of natural numbers and lists. We impose a restriction on our programs that they be *simply* typed (though we allow polymorphism at the *Type* level), rather than using the full power of dependent types. There are two reasons for this. The first is pragmatic: we would like our programming language to resemble a language such as Standard ML [35], which does not have dependent type constructors, and which we use at various points as a convenient prototype notation for the algorithms we

represent in LEGO. In practical terms, of course, this restriction helps to keep things simple, so that we might understand what is going on. A more ambitious task would be to develop the theory for the full language of dependent types. We feel this thesis represents a substantial step in this direction.

The second reason arises from the particular approach we adopted to developing the structure of the various categories of deliverables. We wished in part to allow the methodology to be applied to the Church representations of the datatypes at the impredicative level of the calculus. This would hopefully have permitted some comparison between the work reported here and the work of Paulin-Mohring and collaborators on program extraction in Coquand's original calculus of constructions [81,82]. Unfortunately, these connections remain to be made precise.

Chapter 3 introduces the central definitions and concepts in this thesis. We define a simple notion of specification, consisting of a type together with some predicate defined over it. We show how to define *first-order deliverables*, program-proof pairs which respect this notion of specification, in the type theory ECC. We demonstrate various constructions we may make with such gadgets, which provide both practical examples of the methodology we propose, and some of the meta-theory. We show that first-order deliverables form a *semi-cartesian closed category* in the sense of [37]. This is Theorem 3.1.1. We may also equip the category of first-order deliverables with a weak notion of natural numbers object and lists. We show the correctness proof of a simple doubling function in this framework.

We then discuss how to refine these basic notions, so that we are able to express the statement that a function respects some *relation* between its input and output. This leads to the notions of *relativised specification* and *second-order deliverables*, which are our principal objects of study in the examples we have considered. It turns out that the structure of second-order deliverables is very similar to that of first-order deliverables. Stated precisely, we obtain

Theorem 3.2.1:

**Theorem** *For each specification  $S$ , the category of second-order deliverables over  $S$  has the structure of a semi-ccc;*

and Theorem 3.2.2:

**Theorem** *Second-order deliverables form an indexed category [50,6] over the category of first-order deliverables, whose fibres are semi-cccs, with semi-cc structure strictly preserved by reindexing along first-order deliverables.*

We also show some rules for recursive specifications over natural numbers and lists, which we employ in our examples.

Chapter 4 discusses a number of examples of the development of small programs using the deliverables methodology. The last example discusses a proof of the Chinese remainder theorem, illustrating both the use of second-order deliverables in separating the rôles of parameters from that of dependent variables in the proof, and the economy of the deliverables style as against a separate correctness proof of the underlying algorithm.

In Chapter 5, we return to category theory. We explore how the idea of deliverables may be elaborated in the abstract setting of a category, intended to model some typed functional language, together with a system of abstract predicates indexed over it. We give a detailed account of a particular instance of the construction, based on a topos. The categorical understanding of the semantics of type theories allows us to describe this model in type-theoretic terms. We uncover a subsystem,  $\mathcal{CC}^+$ , of ECC which describes these “abstract deliverables”.  $\mathcal{CC}^+$  was introduced by Luo to overcome the non-conservativity of  $\mathcal{CC}$  over Church’s simple theory of types [60]. It enables us to represent the type of individuals in Church’s system as a constant at the *Type* level, rather than the *Prop* level. If the set theory embodied in the topos is sufficiently rich, as it is for example in the case of Zermelo-Frankel set theory with inaccessible cardinals, then we may interpret the whole of ECC. In any case, we arrive at the conclusion that a fragment of the LEGO system itself may be used as a logic

of correct programs in the higher-order logic of a topos. We believe this may have very useful consequences for the pragmatics of program development, set against existing formal design methodologies in higher-order logic.

The thesis ends with a number of avenues for further exploration of these ideas.

## 1.4 Related work

There are basically two approaches to a logical account of formal program development. 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 [40, 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. Moreover, the proofs we obtain are not those in some encoded logic of programs. For the case of Hoare's logic in the Edinburgh Logical Framework [34], it proved quite difficult to formulate a satisfactory notion of encoding [73,30].

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 [15,79] and the Calculus of Constructions [17,81,82] have been used in this style. This thesis 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 [79, 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 [81,7]:

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

Consequently,  $\mathcal{CC}$  can represent, in the standard representation of functions on inductive types due variously to Church, Girard, Leivant and others [32,56,9], no more functions than  $F_\omega$ . 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 [79] 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”  $\{x \in A \mid B(x)\}$ , 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 [92,93,79].

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 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  $A''(x) \mathbf{prop} [x \in A']$ , again in the basic theory.

This corresponds to our definition of specification in Section 3.1.

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  $A''(a) \mathbf{true}$  in the basic theory. This captures the essential idea, that the judgments  $A \mathbf{set}$  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  $\Pi$ -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  $\Sigma$ -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 [15], 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 [92,93]. 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  $\Pi, \Sigma$ , 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 [39]. 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 [83], 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 [45], 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" [83, p.8].

## 1.5 Prerequisites

We assume the reader is familiar with at least a basic account of category theory, including the definition of adjunction and cartesian closed category, as for example in Mac Lane [67] or Lambek and Scott [53].

We moreover assume that the reader has a rudimentary understanding of type theory. An excellent introduction to the systems of Martin-Löf is contained in the book by Nordström, Petersson and Smith [79].

## Chapter 2

# Type theoretic preliminaries: ECC and LEGO

### 2.1 Review of ECC

Luo's Extended Calculus of Constructions, ECC, [58,59] is a rich type theory containing Coquand and Huet's Calculus of Constructions [17,19] as a subsystem, together with strong  $\Sigma$ -types and a cumulative hierarchy of predicative universes, much as in the systems considered by Martin-Löf and his collaborators [69,70,79]. All these systems are based on the "propositions as types" paradigm, due to Curry and Howard [42], though the ideas go back to a constructive reading of the logical connectives due to Heyting and Kolmogorov. 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, together with a formal system for deriving judgments which define the well-typed terms.

### 2.1.1 The term calculus

The collection of terms is given by the following grammar

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

where  $V$  ranges over some infinite collection of variables, and  $\kappa$  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 [17], 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 [79,70]). 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:

$$(\beta) (\lambda x:A.M)N \triangleright M[N/x], \text{ and}$$

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

This is then extended to a *cumulativity* relation  $\preceq$ , the least relation on the terms such that

- $\preceq$  is a pre-order (in fact Luo proves that  $\preceq$  is a partial order with respect to conversion  $\simeq_\beta$ )
- on kinds,  $Prop \preceq Type_i \preceq Type_j (i \leq j)$ , and

- $\preceq$  respects the formation of  $\Pi$ -types<sup>1</sup> and  $\Sigma$ -types.

### 2.1.2 The derivable judgments of ECC

A *context* is a finite sequence of declarations of the form  $x:M$ , where  $M$  is a term. We denote the empty context by  $\circ$ . A *judgment* is a relation of the form  $\Gamma \vdash M : A$ , where  $\Gamma$  is a context and  $M, A$  are terms. If  $\Gamma$  is the empty context, we often simply write  $\vdash M : A$  in place of  $\circ \vdash M : A$ . The rules of ECC are given in Table 2–1 below.

**Remark** When  $x \notin FV(B)$  we typically write  $A \Rightarrow B$  for  $\Pi x:A.B$ , in case  $A, B$  are both well-typed of type *Prop* in some context, and  $A \rightarrow B$  otherwise. This is simply a device to mark the distinction between logical implication at the *Prop* level, and the function type constructor at predicative levels  $Type_i$  of the type hierarchy. In the same vein, we typically write  $\forall$  in place of  $\Pi$  in instances of rule (II1) in Table 2–1, to emphasise the distinction between the logical quantifier and the dependent type constructor. For  $\Sigma$ -types, if  $x \notin FV(B)$  then we write  $A \times B$  for  $\Sigma x:A.B$ .

As examples of the derivable judgments, and by way of illustration of some of the features of ECC, we have the following:

**Impredicative definition** The rule (II1) is very strong. It allows us to make the usual higher-order definitions of the logical constants and connectives for intuitionistic logic, for example,

$$\perp =_{\text{def}} \forall \chi : Prop. \chi,$$

---

<sup>1</sup>More precisely, if  $A \simeq_{\beta} A', B \preceq B'$  then  $\Pi x:A.B \preceq \Pi x:A'.B'$ . See [59] for a detailed discussion of this point.

(axiom)	$\frac{}{\circ \vdash Prop : Type_0}$
(wcon)	$\frac{\Gamma \vdash A : \kappa}{\Gamma, x:A \vdash Prop : Type_0} \quad (x \notin FV(\Gamma), \kappa \text{ a kind})$
(type)	$\frac{\Gamma \vdash Prop : Type_0}{\Gamma \vdash Type_i : Type_{i+1}} \quad (i \in \omega)$
(var)	$\frac{\Gamma, x:A, \Delta \vdash Prop : Type_0}{\Gamma, x:A, \Delta \vdash x : A}$
(Π1)	$\frac{\Gamma, x:A \vdash P : Prop}{\Gamma \vdash \Pi x:A. P : Prop}$
(Π2)	$\frac{\Gamma \vdash A : Type_i \quad \Gamma, x:A \vdash B : Type_i}{\Gamma \vdash \Pi x:A. B : Type_i} \quad (i \in \omega)$
(λ)	$\frac{\Gamma, x:A \vdash M : B}{\Gamma \vdash \lambda x:A. M : \Pi x:A. B}$
(app)	$\frac{\Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}$
(Σ)	$\frac{\Gamma \vdash A : Type_i \quad \Gamma, x:A \vdash B : Type_i}{\Gamma \vdash \Sigma x:A. B : Type_i} \quad (i \in \omega)$
(pair)	$\frac{\Gamma \vdash M : A \quad \Gamma, x:A \vdash N : [M/x]B \quad \Gamma, x:A \vdash B : Type_i}{\Gamma \vdash \mathbf{pair}_{\Sigma x:A. B}(M, N) : \Sigma x:A. B}$
(π1)	$\frac{\Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \pi_1(M) : A}$
(π2)	$\frac{\Gamma \vdash M : \Sigma x:A. B}{\Gamma \vdash \pi_2(M) : [\pi_1(M)/x]B}$
(≲)	$\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : Type_i}{\Gamma \vdash M : B} \quad (A \preceq B)$

Table 2–1: The rules of ECC for deriving judgments

$$\exists x:A. \phi(x) =_{\text{def}} \forall \chi:Prop. (\forall x:A. \phi(x) \Rightarrow \chi) \Rightarrow \chi.$$

This interpretation is consistent, in that we may show that there exists no term  $M$  such that  $\circ \vdash M : \perp$  [59].

Moreover, impredicativity allows us to make generalised inductive definitions (see for example the exposition in [1,27]). As a simple example, we may derive in context  $\Gamma_{nat} = nat:Type_0, 0:nat, S:nat \rightarrow nat,$

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

which represents 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

$$\Gamma_{nat} \vdash \lambda \Phi:nat \rightarrow Prop.$$

$$\lambda z:\Phi(0).$$

$$\lambda s:\forall k:nat. \Phi(k) \Rightarrow \Phi(S(Sk)).$$

$z$

$$: \forall \Phi:nat \rightarrow Prop. \Phi(0) \Rightarrow (\forall k:nat. \Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(0)$$

is a proof that 0 is even. Such definitions arise in much the same way as Church’s representation of the datatypes in second-order  $\lambda$ -calculus [32, 56,9];

**Universes and cumulativity** As simple examples of the use of universes, we may derive, for each  $i, j \in \omega$  with  $i < j$ ,

$$\vdash Type_i \rightarrow Type_i : Type_j$$

and

$$\vdash \lambda \tau:Type_j. \lambda x:\tau. x : Type_i \rightarrow Type_i,$$

defining a polymorphic identity function, one for each type universe  $Type_i$ .

A less trivial use of cumulativity is the reflection of propositions, that is to say terms of type  $Prop$ , into the predicative levels using rule ( $\preceq$ ). This allows us to form  $\Sigma$ -types whose second component is a proposition. This gives a strong constructive notion of subset type or *species* [8,52]. In this way, we may account for the use of (some, if not all) impredicative instances of set-theoretic comprehension typical in mathematics.

**$\Sigma$ -types** In context  $\Gamma_{nat}, \Phi: nat \rightarrow Prop$ , we may use  $\Sigma$ -types to form a type representing those functions which preserve  $P$ :

$$\Gamma_{nat}, P: nat \rightarrow Prop \vdash \Sigma f: nat \rightarrow nat . \forall n: nat . P(n) \Rightarrow P(fn) : Type_0$$

This type may be seen as a *specification* of the  $P$ -preserving functions. This makes essential use of the cumulativity, as indicated in the previous section. For the case of the predicate defined above representing evenness,

$$Even =_{\text{def}} \lambda n: nat . \forall \Phi: nat \rightarrow Prop . \Phi(0) \Rightarrow (\forall k: nat . \Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(n)$$

we may derive the judgment in Figure 2–1. This term formally represents the knowledge that the function  $x \mapsto x + 2$  preserves the even numbers. We may read this judgment as “ $x \mapsto x + 2$  meets the specification of being an evens-preserving map”. The use of  $\Sigma$ -types allows us to speak of the function, *together with* its proof of correctness, the idea around which all the subsequent chapters of this thesis will revolve.

These examples illustrate the unwieldiness of the  $\lambda$ -expressions defining even the simplest proofs. We shall not often have occasion to describe such proofs explicitly. One of the principle virtues of the LEGO system is that such proofs may be developed by refinement, with the proof term being computed by the system. The finished scripts of such refinement proofs are usually more perspicuous than the  $\lambda$ -expressions to which they give rise. Where appropriate,

$$\Gamma_{nat} \vdash \mathbf{pair}_{\Sigma f:nat \rightarrow nat. \forall n:nat. Even(n) \Rightarrow Even(fn)}(plustwo, proof)$$

$$: \Sigma f:nat \rightarrow nat. \forall n:nat. Even(n) \Rightarrow Even(fn)$$

where

$$plustwo = \lambda n:nat. S(Sn)$$

$$proof = \lambda n:nat. \lambda p:P(n).$$

$$\lambda \Phi:nat \rightarrow Prop.$$

$$\lambda z:\Phi(0). \lambda s:\forall k:nat. \Phi(k) \Rightarrow \Phi(S(Sk)).$$

$$s n (p \Phi z s)$$

**Figure 2–1:** A proof that  $x \mapsto x + 2$  preserves the even numbers

we shall quote from dialogues with the typechecker, to indicate a particular construction whose existence is claimed in the text. Many of the proofs are implemented in LEGO, and we rely on them rather than a laborious explanation in informal mathematics.

### 2.1.3 Proof-theoretic properties of the calculus

The properties of the calculus which are central to the LEGO implementation, and hence this thesis, are as follows:

**Church-Rosser (CR)** The underlying conversion relation is Church-Rosser;

**Strong Normalisation (SN)** If  $\Gamma \vdash M : A$ , then  $M$ ,  $A$  and every type occurring in  $\Gamma$  are strongly normalising;

**Subject Reduction (SR)** If  $\Gamma \vdash M : A$  and  $M \triangleright N$  then  $\Gamma \vdash N : A$ ;

**Strengthening** If  $\Gamma, y:B \vdash M : A$  is derivable, with  $y \notin FV(M, A)$ , then  $\Gamma \vdash M : A$  is derivable;

**Cumulativity**  $\preceq$  has an inductive definition; moreover, for any two terms  $A, B$  it is decidable whether  $A \preceq B$ ;

**Principal types** If  $\Gamma \vdash M : A$ , then there exists a least  $A_p$  in the ordering  $\preceq$  such that  $\Gamma \vdash M : A_p$ .

### 2.1.4 Equality and conversion

The impredicative quantification at the *Prop* level expressed in rule (Π1) in Table 2–1 allows us to define Leibniz' equality (cf. [59, p. 119]):

$$EQ_\tau =_{\text{def}} \lambda x, y:\tau. \Pi P:\tau \rightarrow \text{Prop}. Px \Rightarrow Py.$$

where  $\Gamma \vdash \tau : \kappa$  for some kind  $\kappa$ . That this indeed defines an equivalence relation follows from the impredicativity: see [59, p. 119], or Appendix B for formal proofs of symmetry and transitivity. In his thesis [*ibid.* pp.158–160], Luo proved the following lemma, which establishes the computational adequacy of Leibniz' equality in specifications:

**Lemma 2.1.1** *Suppose  $\circ \vdash a, b : A$ , and  $\circ \vdash M : EQ_A a b$ . Then  $a \simeq_\beta b$ .*

The proof follows from SN, CR and the structure of derivations in ECC, and shows that the normal form of  $M$  must be the proof

$$\text{refl}EQ =_{\text{def}} \lambda x:A. \lambda P:A \rightarrow \text{Prop}. \lambda h:Px.h : \forall x:A. EQ_A x x$$

of the reflexivity of the relation  $EQ$ . The proof rests crucially on the fact that  $a, b, A$  are defined in the *empty* context. But in fact more is true, if we prescribe the shape of  $M$  to match the above term  $\text{refl}EQ$ . An easy induction on the derivation of judgments in the theory, using the inductive character of  $\preceq$ , proves that to

show two terms  $t_1, t_2$  typable in context  $\Gamma$ , are interconvertible, it is necessary and sufficient that

$$\Gamma \vdash \mathit{reflEQ} \ t_1 = \lambda P:\tau \rightarrow \mathit{Prop}.\lambda h:P t_1.h : EQ_\tau \ t_1 \ t_2$$

be derivable, where  $\tau$  is some type of  $t_1, t_2$ .

## 2.2 Review of LEGO

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 [11,17].

### 2.2.1 Syntax

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

$$\begin{aligned} T ::= & \mathit{Prop} \mid \mathit{Type}(i) \ (i \in \omega) \mid V \\ & \{V:T\}T \mid [V:T]T \mid TT \mid \\ & \langle V:T \rangle T \mid (T, T) \mid T.1 \mid T.2 \end{aligned}$$

where the correspondence with the official syntax is given in Table 2–2.

The system does not accept ill-typed terms. LEGO provides extensions to this syntax, some of which are described below, and some of which are due to the detailed operation of the system, which we do not describe here, but details of which may be found in papers of Pollack and others [64,85]. In particular, the presence of cumulativity and  $\Sigma$ -types means that a term may in general be well-typed with (infinitely) many types. Usually the principal type is the most useful, but on occasion we need to *cast* a term  $M$  with a specified (legal) type  $A$ , which is denoted  $M:A$ . This device is already inherent in the official syntax

$\{V:T\}T$	corresponds to	$\Pi x:T. T,$
$[V:T]T$	to	$\lambda x:T. T,$
$\langle V:T \rangle T$	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 2–2:** Comparison between syntax of LEGO and ECC

for pairs  $\mathbf{pair}_{\Sigma x:A.B}(M, N)$ , where the subscript provides a form of casting of the term. Type-casting is also part of the `Goal` mechanism. For a detailed discussion of this point, see [59, p.32, 37-38][85].

## 2.2.2 The theorem prover

The system has two distinct levels, the so-called *lego state* and *proof state*. In the lego state, the user may build and manipulate contexts, type-check terms and initiate proofs. To type-check an expression, the user simply enters it, followed by a semicolon. If the term is well-typed in the current context, which may include local hypotheses and definitions in the course of a proof, the system returns the value and type of the expression. Typically, in illustrating mathematical constructions, we will include such output, indicating the success of the construction, taking seriously Howard’s intention behind “propositions as types” [42]. The construction *is* the proof.

LEGO implements a small command language for developing proofs by refinement. A fuller explanation is in [64,85], but to account for some of the uses of LEGO in the body of this thesis, here is a brief explanation of some of these commands:

**Goal** The command `Goal A`, where  $\Gamma \vdash A : \kappa$  in the current context  $\Gamma$ , initiates a refinement proof of the type  $A$ . The system enters the proof state, which it exits only when no subgoals remain.

**Intros** This command performs introductions as in rules (III), (pair) in Table 2–1 above.

**Refine** The command `Refine term` attempts to unify the current goal with the type of *term*, expanding definitions and performing  $\beta$ -reduction as needed. If successful, it produces as subgoals the types of any bindings not wholly matched, and it fails in the event of no match. Much of the strength of the system lies in the evaluations performed during unification, especially in the presence of  $\delta$ -rules. In the absence of any other subgoals, the proof is successful and the system prints `*** QED ***`. It returns to the `lego` state.

**Save** The command `Save term` adds a new definition to the context, binding the result of a successful refinement to the name *term*, cast with the type of the goal.

Other commands provide context management, and make essential use of the proof-theoretic properties such as strengthening.

### 2.2.3 Additional features of the LEGO system

Pollack’s typechecker for ECC in his LEGO implementation extends the official syntax of ECC in (at least) four significant ways:

**Typical ambiguity** The syntax is extended with an “anonymous” universe symbol *Type*, freeing the user from having to specify universe levels, which are inferred by the system, subject to the constraints of predicativity [36, 86]; for example,

$$[\mathbf{t} : \text{Type}] [\mathbf{x} : \mathbf{t}] \mathbf{x}$$

defines a polymorphic identity function at all levels;

**Definitions** Both local and global definitions are available to the user. They are denoted by  $[x = M]$  uniformly, which acts as a binder for local definitions, and standing alone, as a global definition. Moreover, universe levels are recomputed for each *instance* of (the expansion of) a definition (*ibid.*), allowing a form of universe polymorphism.

**Argument synthesis** The syntax is extended by *implicit* binding operators  $[x | A]$ ,  $\{x | A\}$ , which allow the suppression of arguments in application terms; this is obtained by a translation of the unofficial, implicit syntax into the explicit system, together with an algorithm to synthesise implicit arguments [86]. For example, we may define Leibniz' equality uniformly, without needing to supply the type explicitly:

$$[\text{EQ} = [\mathbf{t} | \text{Type}] [\mathbf{x}, \mathbf{y} : \mathbf{t}] \{P : \mathbf{t} \rightarrow \text{Prop}\} (P \ \mathbf{x}) \rightarrow P \ \mathbf{y}];$$

When  $x, y$  are defined with the same type,  $\text{EQ} \ x \ y$  is then a legal expression; a type is inferred for  $x, y$  and passed as an argument to  $\text{EQ}$ . In the example of the polymorphic identity  $\text{I}$  above, if we define

$$[\text{I} = [\mathbf{t} | \text{Type}] [\mathbf{x} : \mathbf{t}] \mathbf{x}];$$

then the self-application  $\text{I} \ \text{I}$  becomes a legal expression [86].

**Arbitrary reductions** The syntax has recently been extended to provide arbitrary extensions to the conversion relation  $\simeq_\beta$  via new  $\delta$ -reductions for elimination constants at the *Type* level, in accordance with Luo's view that logical types live at the *Prop* level, and datatypes at the *Type* level.

As an example, the natural numbers may be defined in the system as follows:

```

[nat:Type(0)];
[zero:nat];
[succ:nat->nat];
[natrecd:{C:nat->Type}
  {z:C zero}{s:{k:nat}(C k)->C(succ k)}{n:nat}C n];

```

So far, this is no more than a small context of assumptions, which we could extend with axioms defining the behaviour of the primitive recursor `natrecd` when applied to the constructors `zero` and `succ`. However, if these axioms describe  $\delta$ -reductions at the propositional level, using Leibniz' equality `EQ`, this would not allow us to identify *types* dependent on convertible terms. Moreover, the underlying unification algorithm used in LEGO cannot exploit proofs of equality in resolving goals. So we add new reductions, corresponding to their usual  $\delta$ -rules [79]:

```

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

```

This leaves us with the burden of a context of assumptions of datatypes and type constructors, which would have been necessary in an axiomatisation, but gives us the utility of computation by structural recursion. The universe polymorphism, in the type of `natrecd` above, ensures at least as expressive a programming language as Martin-Löf's systems. It is this which motivates our study of program development within ECC. The meta-theory of such an extension for the case of iterated inductive definitions, as in Martin-Löf's various theories, is an active area of research [20,63,33, to name but a few]. Their consistency has been argued model-theoretically [80, etc.], but the preservation (or otherwise) of the strong proof-theoretic properties above is less well understood.

We shall in the course of this thesis use the first three features extensively to simplify the syntactic complexity of our constructions; though the impact of the

first is not directly felt, it clearly has relevance in considering an extension of the theory of deliverables to the case of structuring mathematical theories, as in [59, pp. 147–151], and [62]. We shall exploit the arbitrary reductions feature to describe disjoint sums of types, a unit type, the natural numbers, and lists, with primitive recursion, and define deliverables over these. We hope to give a uniform account of other datatype definitions in future work. See also [33], among others.

### 2.2.4 Equality and Conversion *bis*.

The lemma concerning Leibniz' equality of Section 2.1.4 above becomes a powerful practical tool in LEGO.

**Lemma 2.2.1 (Equality Lemma)** *To establish the interconvertibility of two terms  $t_1, t_2$  in LEGO, all we require is that the following refinement be successful*

```
Goal EQ t1 t2;
Refine reflEQ;
*** QED ***
```

*where*

```
[reflEQ = [t|Type] [x:t] [P:T -> Prop] [h:P x]h];
```

*is the proof of the reflexivity of Leibniz' equality.*

This will considerably simplify much of the routine work in verifying the equality of various categorical constructs in the subsequent chapters.

### 2.2.5 A sample derivation

To give the reader a feel for the LEGO implementation, we present the derivation of Figure 2–1, the proof that  $x \mapsto x + 2$  preserves the even numbers.

Firstly, we make the context  $\Gamma_{nat}$  of assumptions:

```
Lego> [N:Type(0)] [Z:N] [S:N->N];
```

to which the system responds

```
decl N : Type(0)
decl Z : N
decl S : N->N
```

In this context, we define the predicate for evenness:

```
Lego> [Even = [n:N]{Phi:N -> Prop}
        {evenZ:Phi Z}
        {evenSS:{k:N}{ind_hyp:Phi k}Phi(S(S k))}
        Phi n];
```

to which the system responds

```
defn Even = [n:N]{Phi:N->Prop}
            (Phi Z)->({k:N}(Phi k)->Phi (S (S k)))->Phi n
            Even : N->Prop
```

We now initiate a refinement derivation of a term of the relevant  $\Sigma$ -type:

```
Goal <f:N -> N> {n:N}(Even n) -> Even (f n);
```

to which the system responds

```
Goal
  ?0 : <f:N->N>{n:N}(Even n)->Even (f n)
```

The term `?0` represents a term waiting to be instantiated as the proof progresses. Henceforth, we record the response of the system as the user sees it, directly following commands at the `Lego>` prompt.

Now the outermost constructor of the term is `pair`, so we use  $\Sigma$ -introduction:

```

Lego> Intros #;
Intros (0) #
  ?1 : N->N
  ?2 : {n:N}(Even n)->Even (?1 n)

```

There are now two subgoals, ?1, ?2, corresponding to the two components of the pair. We work first on ?1. Since the goal has functional type, we use  $\Pi$ -introduction. We then refine by the term  $S$ , twice (since we have an idea of the construction we wish to make).

```

Lego> Intros n;Refine S;Refine S;
Intros (1) n
  n : N
  ?3 : N
Refine by S
  ?4 : N
Refine by S
  ?5 : N

```

We now close this branch of the derivation tree, using the local assumption  $n : N$ . The `Discharge..` indicates the discharge of this local hypothesis in the derivation tree, just as in natural deduction [89].

```

Lego> Refine n;
Refine by n
Discharge.. n
  ?2 : {n:N}(Even n)->Even (([n'3:N]S (S n'3)) n)

```

We now work on subgoal ?2, which has been further instantiated by the solution to ?1. Again, we use  $\Pi$ -introduction, and then expand the definition of the predicate `Even`.

```

Lego> Intros n hyp;Expand Even;
Intros (2) n hyp
  n : N
  hyp : Even n

```

```

?6 : Even (([n:N]S (S n)) n)
Expand Even
?6 : {Phi:N->Prop}(Phi Z)->({k:N}(Phi k)->Phi (S(S k)))->Phi (S(S n))

```

We see that there are a further three introductions to make, this time using a wildcard character, the underscore `_`, as in Standard ML [35].

```

Lego> Intros _ _ _;
Intros (3) _ _ _
  Phi : N->Prop
  evenZ : Phi Z
  evenSS : {k:N}(Phi k)->Phi (S (S k))
  ?7 : Phi (S (S n))

```

At this stage, not all the local context is visible, but we may use the `Prf` command to display the whole local context, and the current subgoals.

```

Lego> Prf;
  n : N
  hyp : Even n
  Phi : N->Prop
  evenZ : Phi Z
  evenSS : {k:N}(Phi k)->Phi (S (S k))
  ?7 : Phi (S (S n))

```

In fact, at this stage of the proof, there is only one choice open to us, as only the hypothesis `evenSS : {k:N}(Phi k)->Phi(S(S k))` unifies with the current subgoal.

```

Lego> Refine evenSS;
Refine by evenSS
  ?9 : Phi n

```

Once more there is only one choice, namely to use the hypothesis `hyp : Even n`.

```

Lego> Refine hyp;
Refine by hyp
  ?11 : Phi Z
  ?12 : {k:N}(Phi k)->Phi (S (S k))

```

Finally, we may use the `Immed` tactic to resolve any outstanding subgoals against the local context.

```

Lego> Immed;
Immediate
Discharge.. evenSS evenZ Phi
Discharge.. hyp n
*** QED ***

```

The alternative at this point would be to give explicit refinement terms

```

Lego> Refine evenZ;Refine evenSS;
Refine by evenZ
  ?12 : {k:N}(Phi k)->Phi (S (S k))
Refine by evenSS
Discharge.. evenSS evenZ Phi
Discharge.. hyp n
*** QED ***

```

In either case, the derivation is now complete, so we may save the proof term.

```

Lego> Save a_simple_example;
a_simple_example saved

```

We may now examine the term, by simply entering its name. We see here the explicit type-casting in LEGO's representation of terms in a  $\Sigma$ -type.

```

Lego> a_simple_example;
value = ([n:N]S (S n),
         [n:N] [hyp:Even n] [Phi:N->Prop]
         [evenZ:Phi Z] [evenSS:{k:N}(Phi k)->Phi (S (S k))]
         evenSS n (hyp Phi evenZ evenSS))

```

```

      : <f:N->N>{n:N}(Even n)->Even (f n))
type  = <f:N->N>{n:N}(Even n)->Even (f n)

```

We may also construct the explicit terms *plustwo* and *proof* of Figure 2–1, by definition.

```

Lego> [plustwo = a_simple_example.1];
defn  plustwo = a_simple_example.1
      plustwo : N->N
Lego> [proof = a_simple_example.2];
defn  proof = a_simple_example.2
      proof : {n:N}(Even n)->Even (a_simple_example.1 n)

```

Lastly, we may use the commands `Hnf VReg`, and `Normal VReg` to display these values in head-normal form, respectively normal form<sup>2</sup>.

```

Lego> plustwo;
value = a_simple_example.1
type  = N->N
Lego> Hnf VReg;
[n:N]S (S n)
Lego> Normal VReg;
[n:N]S (S n)

Lego> proof;
value = a_simple_example.2
type  = [f=a_simple_example.1]{n:N}(Even n)->Even (f n)
Lego> Hnf VReg;
[n:N][hyp:Even n]
[Phi:N->Prop][evenZ:Phi Z][evenSS:{k:N}(Phi k)->Phi (S (S k))]
evenSS n (hyp Phi evenZ evenSS)
Lego> Normal VReg;

```

---

<sup>2</sup>Likewise, the commands `Hnf TReg`, and `Normal TReg` display their types in head-normal form, respectively normal form.

```

[n:N] [hyp:{Phi:N->Prop}(Phi Z)->({k:N}(Phi k)->Phi (S (S k)))->Phi n]
[Phi:N->Prop] [evenZ:Phi Z] [evenSS:{k:N}(Phi k)->Phi (S (S k))]
evenSS n (hyp Phi evenZ evenSS)

```

This concludes the example.

## Chapter 3

# First-order and second-order deliverables

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

## 3.1 Definition and properties of first-order deliverables

We work relative to some well-formed context  $\Gamma$  in ECC.

### 3.1.1 Specifications

Informally, we consider a first approximation<sup>1</sup> to the idea of specification, given by a pair, consisting of

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

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

---

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

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

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

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  $\text{SPEC}_1 =_{\text{def}} \Sigma s : \text{Type}.s \rightarrow \text{Prop}$ , so that we may consider operations which construct specifications entirely within the framework of ECC (cf. the account of specifications and refinements in [61]). We will consider a category whose objects are the specifications, and whose morphisms are defined below. Specifications defined by *logically* equivalent predicates in general define distinct objects.

### 3.1.2 First-order deliverables

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

**Definition 3.1.2 (Burstall [12,59])** *first-order deliverable*

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

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

---

<sup>3</sup>The reader hopefully will forgive this somewhat unfortunate notation, which is chosen to be consistent with that of specifications.

The motivation for such a definition goes right back to Hoare’s original paper on axiomatic semantics [40], and, like the logic of triples which bears his name, expresses in a *formal* system the informal notion of a program, together with a certificate of some specified input/output behaviour. Of course, we are concerned with a functional language, rather than an imperative one, so there is no confusion over program variables and logical variables. In this framework, moreover, the proof and the program are linked as a pair. This definition uses the  $\Sigma$ -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  $\pi_1$  from the  $\Sigma$ -type yields the underlying algorithm  $f$ . Indeed this accounts for the name “deliverables”: they are what a software house should deliver to its customers, a program plus a proof in a box with the specification printed on the cover (the  $\Sigma$ -type). The customer can independently check the proof and then run the program, without the need for a complicated extraction process which may yield an unusual algorithm. Indeed, we even propose this method as a style for developing programs in the first place. We usually have a reasonable idea of the algorithm in advance, as opposed to its proof of termination, or even correctness, and we would like an understanding, which reflects our intuitions, of how to build up these deliverables from smaller ones, for example by refinement and composition, possibly with machine assistance.

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

algorithm. Nonetheless, the categorical combinators which we develop below give us a schematic description of the algorithm within a deliverable.

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

```

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

```

We have exploited the implicit syntax to enable us to suppress the argument types<sup>4</sup> in the predicate `Del1` and the type `del1`.

With an eye to the categorical aspects of this definition, we typically write<sup>5</sup>

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

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

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

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

---

<sup>4</sup>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.

<sup>5</sup>The reader should be careful to distinguish the category arrow  $\longrightarrow$  from the type constructor arrow  $\rightarrow$ .

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

### 3.1.3 Equality of deliverables

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

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

**Definition 3.1.3** *Equality of specifications*

Given specifications  $(s, S), (t, T)$ , we say  $(s, S) = (t, T)$  if

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

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

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

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

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

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

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

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

A proof of the equivalence of these two definitions involves the *construction*, on the one hand, of specific permutations of initial segments  $\{1, \dots, n\}$  of the

natural numbers, and on the other, essentially, derivation trees of the proofs of propositions  $u \sim v$ . Appendix B contains a detailed treatment of the impredicative definition in LEGO. We developed this in the course of investigating proofs of sorting algorithms.

**Definition 3.1.4** *Equality of deliverables*

Given

$$(s, S) \xrightarrow[\begin{smallmatrix} (f, F) \\ (g, G) \end{smallmatrix}]{=} (t, T) \in \mathbf{del}_1,$$

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

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

and

$$\lambda x:s. \lambda h:S x. Fxh \simeq_{\beta\delta} \lambda x:s. \lambda h:S x. Gxh.$$

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

```

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

```

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

### 3.1.4 Semi-structure in categories

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

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

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

---

<sup>6</sup>I am grateful to Bart Jacobs for introducing me to this concept.

and

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

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

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

$$\mathbf{C} \begin{array}{c} \xrightarrow{F} \\ \xleftarrow{G} \end{array} \mathbf{D},$$

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

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

the following diagram

$$\begin{array}{ccc} \mathbf{D}(FC, D) & \begin{array}{c} \xrightarrow{\sigma_{C,D}} \\ \xleftarrow{\tau_{C,D}} \end{array} & \mathbf{C}(C, GD) \\ \begin{array}{c} \downarrow Ff; (-); g \\ \downarrow \end{array} & & \begin{array}{c} \downarrow f; (-); Gg \\ \downarrow \end{array} \\ \mathbf{D}(FC', D') & \begin{array}{c} \xrightarrow{\sigma_{C',D'}} \\ \xleftarrow{\tau_{C',D'}} \end{array} & \mathbf{C}(C', GD') \end{array}$$

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

The beauty of this definition lies in the following lemma.

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

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

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

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

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

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

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

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

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

$$W \xrightarrow{h} X \xrightarrow{\langle f, g \rangle} Y \times Z = W \xrightarrow{\langle hf, hg \rangle} Y \times Z.$$

(ii)  $X \times Y \xrightarrow{\pi_l} X$  and  $X \times Y \xrightarrow{\pi_r} Y$ , with  $\langle f, g \rangle \pi_l = f$  and  $\langle f, g \rangle \pi_r = g$

---

<sup>7</sup>However, by contrast with adjunctions, structure defined by a semi-adjunction is not in general unique.

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

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

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

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

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

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

(vi)

$$Z^Y \times Y \xrightarrow{ev} Z = Z^Y \times Y \xrightarrow{\langle \pi_l, \pi_r \rangle} Z^Y \times Y \xrightarrow{ev} Z.$$

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

**Theorem 3.1.1** *del<sub>1</sub> is semi-cartesian closed.*

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

### 3.1.5 Identities and composition

By considering our slight modification of the underlying conversion on the terms of ECC, to remedy the failure of surjective pairing and  $\eta$ -conversion in

the calculus, we fixed a notion of equality on arrows. With this definition, it is now trivial to establish that specifications, together with first-order deliverables as morphisms, form a category, denoted by  $\mathbf{del}_1$ .

The identities are given simply by

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

in LEGO, we have

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

Composition is given by

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

in LEGO, we have

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

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

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

## Proof

---

<sup>8</sup>Indeed, we only need the modified equality to prove the identity laws. The conversion relation is sufficient to establish associativity of composition.

```

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

```

■

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

### 3.1.6 Semi-Terminal object

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

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

---

<sup>9</sup>Here,  $()$  is the unique term of type  $unit$ .

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

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

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

This satisfies Hayashi's condition (v) above.

### Proof

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

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

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



### 3.1.7 Binary semi-products

To obtain semi-products, we use the non-dependent  $\Sigma$ -type as the underlying type, with conjunction at the predicate level<sup>10</sup>:

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

In LEGO, we have the following:

```

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

```

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

---

<sup>10</sup>This is hardly surprising, given the Curry-Howard correspondence.

```

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

```

### 3.1.8 Binary semi-coproducts

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

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

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

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

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

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

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

**Remark** In the above definition, we wrote

```
[sum_type: ([tau:Type] tau->tau->tau) Type]
```

for the sum type-constructor. This device, due to H.Goguen, using an  $\eta$ -expansion of the ambiguous expression “Type->Type->Type ” allows us to define a sum type at all type levels, using Pollack’s typical ambiguity translation, in such a way that

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

---

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

```

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

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

```

### 3.1.9 Semi-exponentials

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

```

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

type = {s,t,u|Type}{S|Pred s}{T|Pred t}{U|Pred u}
       (del1 (Product_del1 S T) U)->del1 S (Del1 T U)

```

```

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

type = {t,u|Type}{T:Pred t}{U:Pred u}
       del1 (Product_del1 (Del1 T U) T) U

```

```

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

```

```

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

```

```

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

```

### 3.1.10 Semi-pullbacks and the internalisation of equality

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

the vertex of a pullback square in  $\mathbf{del}_1$ , viz.

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

given a cone of the form

$$\begin{array}{ccc} & (t, T) & \\ & \downarrow (g, G) & \\ (s, S) & \xrightarrow{(f, F)} & (u, U) \end{array}$$

together with projection maps (given essentially by  $\pi_1, \pi_2$ ), and even show that the mediating arrow

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

from a commuting square

$$\begin{array}{ccc} (v, V) & \xrightarrow{(q, Q)} & (t, T) \\ (p, P) \downarrow & & \downarrow (g, G) \\ (s, S) & \xrightarrow{(f, F)} & (u, U) \end{array}$$

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

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

---

<sup>12</sup>Here,  $EQ$  is Leibniz' equality defined in Chapter 2.

<sup>13</sup> $conj$  is the term which, given proofs  $p_1, p_2, p_3$  of three propositions  $\phi, \psi, \chi$ , returns the proof of the conjunction  $\phi \wedge \psi \wedge \chi$ .

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

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

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

$$f : s \perp \rightarrow t, P : t \perp \rightarrow Prop$$

we obtain

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

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

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

**Proof** Obvious.

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

■

### 3.1.12 A consequence rule

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

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

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

### 3.1.13 Pointwise construction

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

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

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

### 3.1.14 Inductively defined types

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

---

<sup>14</sup>This just corresponds to Howard’s observation, emphasised by Martin-Löf, that given a strong interpretation of the existential quantifier as  $\Sigma$ -type, the axiom of choice becomes constructively valid [42,70,71].

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

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

### Natural numbers

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

$$nat: Type, 0: nat, S: nat \perp \rightarrow nat$$

We typically abbreviate  $S$  to “+1” in informal mathematical language. We extend this context with a dependent elimination constant  $natrecd$ <sup>16</sup>,

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

---

<sup>15</sup>Strong, that is, because we can eliminate over *types*, and not merely propositions.

<sup>16</sup>“d” for dependent.

together with reduction rules defining the  $\delta$ -redices<sup>17</sup> (in some context where  $C, z, s, n$  have the appropriate types):

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

This is precisely expressed in LEGO as follows<sup>18</sup> :

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

This yields a derived iterator and primitive recursor

$$\text{natiter}:\Pi\alpha:\text{Type}. \alpha \perp \rightarrow (\alpha \perp \rightarrow \alpha) \perp \rightarrow \alpha$$

and

$$\text{natrec}:\Pi\alpha:\text{Type}. \alpha \perp \rightarrow (\text{nat} \perp \rightarrow \alpha \perp \rightarrow \alpha) \perp \rightarrow \alpha$$

where<sup>19</sup>

$$\text{natiter } z \ s \ 0 \ =_{\text{def}} \ z$$

---

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

<sup>18</sup>cf. the discussion in Chapter 2 of adding arbitrary reductions in LEGO.

<sup>19</sup>The type  $\alpha$  is, of course, inferred by the typechecker.

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

and an induction principle

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

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  $\text{natiter } z \ s$  or  $\text{natrec } z \ s$ , together with proven propositions about them. For the iterator, we obtain

$$\frac{\Phi(z) \ \forall y : \alpha. \Phi(y) \implies \Phi(s \ y)}{\forall n : \text{nat}. \Phi(\text{natiter } z \ s \ n)} [z : \alpha, s : \alpha \perp \rightarrow \alpha]$$

and for the recursor

$$\frac{\Phi(z) \ \forall k : \text{nat}. \forall y : \alpha. \Phi(k) \implies \Phi(s \ k \ y)}{\forall n : \text{nat}. \Phi(\text{natrec } z \ s \ n)} [z : \alpha, s : \text{nat} \perp \rightarrow \alpha \perp \rightarrow \alpha].$$

Since we are interested in building new deliverables from less complex ones, we could of course use the dependent eliminator  $\text{natrecd}$  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  $\text{Nat} =_{\text{def}} \lambda n : \text{nat}. \text{true}$  on the natural numbers. For each constructor of the type, we obtain a corresponding deliverable:

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

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

For the iterator, we obtain

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

and for the recursor

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

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

These terms are easily obtained by refinement in LEGO:

```

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

```

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

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

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

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

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

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

We thus obtain a non-trivial recursive first-order deliverable<sup>20</sup>.

## Lists

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

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

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

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

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

---

<sup>20</sup>This example is originally due to Burstall.

defines  $Listof A$  by primitive recursion.

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

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

and

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

with function component,

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

and proof component

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

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

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

and a recursor

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

In LEGO we have the following:

```

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

```

```

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

```

## 3.2 Second-order deliverables

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

### 3.2.1 A thought experiment

Suppose we are given some  $\Pi_2^0$  specification  $\forall x:s. \exists z:u. R(x, z)$ , and we wish to find some function  $f:s \dashv\rightarrow u$  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

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

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

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

and  $h$  solves

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

This last is logically equivalent to

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

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

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

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

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

The composition of two such gadgets

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

is definable as<sup>21</sup>

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

---

<sup>21</sup>We employ an informal alphabetical convention:  $x$  is of type  $s$ ,  $y$  of type  $t$ ,  $z$  of type  $u$ , and hence  $w$  is of type  $r$ .

We have now established a definition which respects the symmetry of source and target in our previous analysis of the decomposition of the  $\Pi_2^0$  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  $r =_{\text{def}} \text{unit}$ ,  $P(x, w) =_{\text{def}} \text{true}$ , and  $f_{\text{old}} =_{\text{def}} \lambda x:s. f_{\text{new}} x ()$ .

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

### 3.2.2 Basic definitions

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

**Definition 3.2.2** *relativised specification*

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

---

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

- $\Gamma \vdash t : \text{Type}$ , and
- $\Gamma \vdash R : s \perp \rightarrow t \perp \rightarrow \text{Prop}$ .

**Definition 3.2.3 (Burstall)** *second-order deliverable*

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

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

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

$$\lambda f : s \perp \rightarrow t \perp \rightarrow u. \forall x : s. Sx \implies \forall y : t. Q(x, y) \implies R(x, (fxy)).$$

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

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

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

*If*

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

*then*

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

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

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

**Proof** As constructions in LEGO, we obtain

```

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

```

■

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

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

```

Lego> id_del2;

```

```

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

```

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

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

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

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

### 3.2.3 Each $\text{del}_2 S$ is a semi-ccc

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

on  $h$ . The function component  $f$  may not depend on  $h$ , but the proof component  $F$  may do so.

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

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

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

**Semi-terminal object** This is given by the relativised specification

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

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

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

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

Then we may form the relativised specification

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

---

<sup>23</sup>Here,  $\top$  is the term corresponding to *true*-introduction,

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

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

```
Lego> top;
```

```
value = [A|Prop][a:A]a
```

```
type = {A|Prop}A->A
```

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

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

where

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

Projections are similarly straightforward to define:

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

has function component  $\lambda x:s. \lambda yz:t \times u. \pi_1(yz)$ , and proof component <sup>25</sup>

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

---

<sup>24</sup>Here, *pair* is the term corresponding to  $\wedge$ -introduction,

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

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

Lego> pair;

value = [A,B|Prop] [a:A] [b:B] [C:Prop] [h:A->B->C] h a b

type = {A,B|Prop} A->B->and A B

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

<sup>25</sup>As above, *fst* is a term corresponding to  $\wedge$ -elimination,

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

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

Lego> fst;

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

**Semi-exponentials** Just as the predicate  $\text{Del}_1$  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  $\text{Del}_2$  gives rise. More precisely, suppose  $(r, P), (t, Q), (u, R)$  are relativised specifications. We obtain a relativised specification

$$R^Q =_{\text{def}} (t \perp \rightarrow u, \lambda x:s. \lambda f:t \perp \rightarrow u. \forall y:t. Q x y \implies R x (fy)).$$

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

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

we obtain

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

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

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

and

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

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

---

```
value = [A,B|Prop] [p:and A B]p A ([a:A] [b:B] a)
```

```
type = {A,B|Prop} (and A B) -> A
```

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

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

■

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

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

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

### 3.2.5 Pullback functors

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

**Definition 3.2.4** *pullback along a first-order deliverable*

Suppose we are given specifications  $S, T$ , and a first-order deliverable  $S \xrightarrow{(k, K)} T$ .

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  $Q =_{\text{def}} (u, Q)$  with respect to  $T$ , let

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

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

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

we define  $(k, K)^*(f, F)$  to be the pair

$$(\lambda x:s. \lambda z:u. f (kx) z, \lambda x:s. \lambda h:Sx. \lambda z:u. \lambda p:Q (kx) z. F (kx) (Kxh) z p).$$

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

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

```

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

```

■

**Lemma 3.2.3**  $(k, K)^*$  preserves identities and composition.

**Proof** Using the equality Lemma 2.2.1,

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

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

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

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

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

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

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

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

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

**Proof** Straightforward typechecking. ■

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

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

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

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

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

But this is straightforward. ■

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

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

As a consequence of this theorem, we might hope, in the light of [102], to give a semantics for Martin-Löf type theory in terms of deliverables. In particular, we would expect to interpret *dependent* products and sums of specifications. This would seem to be part of the development of the subset theory in [79]. Since all the structure is defined by semi-adjunctions, however, rather than adjunctions as in Seely’s account of an extensional theory, we defer discussing such an idea — and the corollary that we may use a language of dependent types to describe

and manipulate deliverables — to Chapter 5. In particular, we avoid discussing a technical difficulty in the definition of the structure of dependent products, which arises once again from the absence of surjective pairing. Namely, to give the structure of dependent  $\Pi$ , we must define right semi-adjoints to the weakening functors between fibres [48]. In order to be able to do this, we must restrict ourselves, in the absence of surjective pairing, to those relations defined over  $\Sigma$ -types, such that

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

### 3.2.7 Second-order deliverables for natural numbers and lists

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

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

This means, for the case of natural numbers, that we now examine proofs of statements of the form<sup>26</sup>:

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

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

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

---

<sup>26</sup>We only consider  $\mathit{natrec}$ , since  $\mathit{natiter}$  is a degenerate instance of it.

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

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

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

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

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

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

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

where  $(+1)^*R$ , otherwise written  $R[n+1/n]$ , is the relation

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

In like manner, we write  $0^*R$ , or  $R[0/n]$ , for the relation

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

We thus obtain the second-order deliverable constructor for  $\mathit{nat}$  recursions as the following derived rule

$$\frac{R \xrightarrow{(s, S)} (+1)^*R \quad [1_{\mathit{nat}}]}{\mathit{Natrec}_2\ (s, S):0^*R \rightarrow R \quad [1_{\mathit{nat}}]}$$

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

$$\frac{1 \xrightarrow{(z, Z)} 0^*R \quad [1_{nat}] \quad R \xrightarrow{(s, S)} (+1)^*R \quad [1_{nat}]}{Natrec'_2(z, Z) (s, S):1 \rightarrow R \quad [1]}.$$

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

`del12 ?n ?m R`

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

### 3.2.8 Lists

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

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

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

where

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

and

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

The LEGO term we obtain is:

```

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

```

where

```

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

```

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

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

as proof component.

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

This is not to say, however, that we cannot obtain forms of this rule in which the input list is not further constrained. We may, for example, consider the predicate  $Listof A$ , 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  $\Phi$  is some *relation* between values of the variable  $x$  varying over the parameter type  $a$ , and lists over  $a$ . An example is the predicate  $Sorted$ , for which we take  $\Phi x l =_{\text{def}} x \preceq l$ , the relation that  $x$  is less than each element of the list  $l$ . This is discussed in more detail in the examples in the next chapter.

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

$$\forall l: list a. (depListof \Phi l) \implies R l (listrec n c l)$$

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

**base case**  $true \implies R \text{ nil } n$ , which reduces logically to  $R \text{ nil } n$ . As with the rules for  $\text{Natrec}_2$ , 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

$$\begin{aligned} \forall x:a. \forall l:\text{list } a. ((\text{depListof } \Phi \ l) \implies R \ l \ (\text{listrec } n \ c \ l)) \implies \\ ((\text{depListof } \Phi \ (x :: l)) \implies R \ (x :: l) \ (c \ x \ l \ (\text{listrec } n \ c \ l))). \end{aligned}$$

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

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

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

$$\frac{\mathcal{F}:\prod x:a. R \rightarrow (\text{cons } x)^*R \quad [(\text{cons } x)^*\text{depListof } \Phi]}{\text{depListrec}_2 \ \mathcal{F}:\text{nil}^*R \rightarrow R \quad [\text{depListof } \Phi]}$$

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

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

```

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

```

## Chapter 4

# Examples

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

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

## 4.1 Division by two

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

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

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

### 4.1.1 The mathematical specification

Rather than consider the original specification,

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

we started from a slightly weaker one, namely

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

Our intended algorithm below meets the stronger specification, namely

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

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

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

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

### 4.1.2 The correctness proof of our intended algorithm

We consider the prototype algorithm in Standard ML [35]

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

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

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

is easily shown to meet the second specification:

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

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

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

### 4.1.3 The development in terms of deliverables

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

Let

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

We seek

$$\lambda n:\mathbb{N}. \lambda u:\text{unit}. \text{true} \xrightarrow{\text{div}_2} Div2Spec \quad [\lambda n:\mathbb{N}. \text{true}]$$

This gives us our first subgoal

$$1 \xrightarrow{\Gamma_0} Div2Spec \quad [1]$$

```
Lego> [Div2Spec = [n:nat] [p:nat#nat] [q=p.1] [r=p.2]
      and3 (EQ n (plus q r)) (leqNat q r) (leqNat r (succ q))];
defn Div2Spec = [n:nat] [p:nat#nat] [q=p.1] [r=p.2]
      and3 (EQ n (plus q r)) (leqNat q r) (leqNat r (succ q))
Div2Spec : nat->(nat#nat)->Prop
```

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

We then decompose this subgoal, into two others

$$\Gamma_0 \rightsquigarrow 1 \xrightarrow{\Gamma_9} \Gamma_7 \quad [1], \quad \Gamma_7 \xrightarrow{\Gamma_{10}} Div2Spec \quad [1]$$

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

$$\begin{array}{l} \Gamma_9 \rightsquigarrow 1 \xrightarrow{\Gamma_9} \text{Div2Spec}[0/n] \quad [1] \\ \Gamma_{10} \rightsquigarrow \text{Div2Spec} \xrightarrow{\Gamma_{13}} \text{Div2Spec}[n + 1/n] \quad [1] \end{array}$$

```
Lego> Refine compose_del2;
```

```
Refine by compose_del2
```

```
?3 : Type
```

```
?7 : Rel nat ?3
```

```
?9 : del2 (univPred|nat) (univRel|nat|unit) ?7
```

```
?10 : del2 (univPred|nat) ?7 Div2Spec
```

```
Lego> Refine +3 natrec_del2;
```

```
Refine 10 by natrec_del2
```

```
?9 : del2 (univPred|nat) (univRel|nat|unit) (zstarRel Div2Spec)
```

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

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

```
Lego> Refine pointwise_del2;
```

```
Refine by pointwise_del2
```

```
?20 : {a:nat}{b:unit}
```

```
<c:nat#nat>(univPred|nat a)->(univRel|nat|unit a b)->
```

```
zstarRel Div2Spec a c
```

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

```
Lego> intros n u # #;
```

```
intros (2) n u #
```

```
n : nat
```

```

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

```

This branch of the tree is now closed.

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

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

```

?13 : del2 Nat Div2Spec (sstarRel Div2Spec)

Lego> Refine pointwise_del2;
Refine by pointwise_del2

```

```

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

```

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

```

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

```

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

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

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

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

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

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

```

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

```

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

## 4.2 Finding the minimum of a list

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

### 4.2.1 The mathematical specification

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

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

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

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

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

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

We have explicitly curried the function definition of [96]<sup>1</sup>

```
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:

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

---

<sup>1</sup>Sannella in fact treats the *maximum* of the list.

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

## 4.2.2 The correctness proof of our intended algorithm

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

### Base case

$$\exists f:A \rightarrow A. \forall a:A. fa \in a :: nil \wedge fa \preceq_R a :: nil.$$

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

### Step case Suppose

$$\exists f:A \rightarrow A. \forall a:A. fa \in a :: k \wedge fa \preceq_R a :: k.$$

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

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

## 4.2.3 The development in terms of deliverables

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

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

```

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

```

Let

$$\text{MinSpec} =_{\text{def}} \lambda l:\text{list } A. \lambda f:A \perp \rightarrow A. \forall a:A. fa \in a :: l \wedge fa \preceq_R a :: l.$$

We seek an arrow

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

```

Lego> [min [a,b:A] = if (r a b) a b];
Lego> [Inlist [a:A] =
      listrec false ([b:A][l:list A][p:Prop]or (EQ a b) p)];
Lego> [MinSpec [l:list A][f:A->A]
      = {a:A}and (Inlist (f a) (cons a l))
        (Lelist R (f a) (cons a l))];
Lego> Goal del2 (univPred|(list A)) (univRel|(list A)|unit) MinSpec;

```

yielding

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

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

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

yielding

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

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

```
Lego> Refine pointwise_del2;
?21 : {a:list A}{b:unit}<c:A->A>
      (univPred|(list A) a)->(univRel|(list A)|unit a b)->
      nstarRel MinSpec a c

Lego> intros l u #;Intros +1 __ a;
h : univPred|(list A) l
pre : univRel|(list A)|unit l u
a : A
?25 : and (Inlist (?23 a) (cons a (nil A)))
      (Lelist R (?23 a) (cons a (nil A)))
```

```
Lego> Refine pair;
```

This gives us the two conjuncts

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

which we resolve with

```

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

```

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

```

Lego> intros b;Refine pointwise_del2;
  ?48 : {a:list A}{b'2:A->A}<c:A->A>
      (univPred|(list A) a)->
      (MinSpec a b'2)->
      cstarRel b MinSpec a c
Lego> intros l f #;Refine [a:A]min a (f b);
  ?51 : (univPred|(list A) l)->
      (MinSpec l f)->
      cstarRel b MinSpec l ([a:A]min a (f b))

```

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

```

Lego> Intros _ spec a;Refine spec b;
Lego> intros inlist lelist;
Lego> Refine boolIsInductive (r a (f b));
  ?57 : (EQ tt (r a (f b)))->
      and (Inlist (min a (f b)) (cons a (cons b l)))
          (Lelist R (min a (f b)) (cons a (cons b l)))
  ?58 : (EQ ff (r a (f b)))->
      and (Inlist (min a (f b)) (cons a (cons b l)))
          (Lelist R (min a (f b)) (cons a (cons b l)))

```

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

```

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

```

The second case requires the use of an explicit lemma

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

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

```

Lego> intros case2;
Lego> Refine case2 [bb:bool][ga = if bb a (f b)]
           and (Inlist ga (cons a (cons b l)))
           (Lelist R ga (cons a (cons b l)));
Lego> Refine pair;Refine inr;Immed;
Lego> Refine pair;Refine linearRlemma;Immed;
Immediate
Discharge.. case2
Discharge.. lelist inlist
Discharge.. a spec h
Discharge.. f l
Discharge.. b
*** QED ***

```

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

```

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

[l:list A]listrecd ([_:list A]A->A)
  ([t:A]t)
  ([b:A] [_:list A] [f:A->A]
   [a:A]boolrecd ([_:bool]A) a (f b)
                (r a (f b)))
  1

```

### 4.3 Insert sort

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

#### 4.3.1 The mathematical specification

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

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

where

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

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

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

**Lemma 4.3.1**  $\forall a: \alpha. \forall l: \text{list } \alpha. \text{Sorted } (a :: l) \implies \text{Sorted } (l)$

**Lemma 4.3.2**  $\forall a: \alpha. \forall l: \text{list } \alpha. \text{Sorted } (a :: l) \implies a \preceq l$

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

Induction enables us to extend these results to

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

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

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

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

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

$$\frac{l \sim m \quad S l}{S m}$$

Hence, as a corollary,

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

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

From which we obtain

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

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

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

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

**Lemma 4.3.10**

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

### 4.3.2 A correctness proof for insert sort

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

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

and here is its translation into LEGO code:

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

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

[sort = listrec nil scons];
```

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

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

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

$$\frac{}{\forall l:list\ \alpha. \phi\ l\ (listrec\ n\ c\ l)}$$

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

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

$$\left[ \begin{array}{c} Sorted\ (a::l), \ \phi\ l\ b \ [a:\alpha, l:list\ \alpha, b:\beta] \\ \vdots \\ \phi\ nil\ n \quad \phi\ (a::l)\ (c\ a\ l\ b) \end{array} \right]$$

$$\frac{}{\forall l:list\ \alpha. \ Sorted\ l \implies \phi\ l\ (listrec\ n\ c\ l)} \quad \text{Sorted List Induction}$$

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

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

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

(b)  $\forall a : \alpha. \forall l, m : list\ \alpha. (Sorted\ m) \wedge l \sim m \implies$   
 $(Sorted\ (scons\ a\ m) \wedge a :: l \sim (scons\ a\ m)).$

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

$$\forall a : \alpha. \forall m : list\ \alpha. Sorted\ m \implies (Sorted\ (scons\ a\ m) \wedge a :: m \sim (scons\ a\ m))$$

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

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

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

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

(d)  $\forall b : \alpha. \forall m, n : list\ \alpha. Sorted\ (b :: m) \wedge \psi_a\ m\ n \implies \psi_a\ (b :: m)\ (swapcons\ b\ m).$

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

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

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

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

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

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

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

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

- (i)  $b \leq c$ ;
- (ii)  $c \leq b, b \leq a$ ;
- (iii)  $c \leq b, a \leq b$ .

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

- (i)  $b \leq c \implies \max(b, c) = c \preceq n$ , by Lemma 4.3.2
- (ii)  $c \leq b \implies \max(b, c) = b$ . By definition of  $\preceq$ ,  $b \leq a, b \preceq m \implies b \preceq a :: m$ .  
 $c :: n \sim a :: m \implies b \preceq c :: n$ . Hence  $b \preceq n$ .
- (iii)  $a \leq b, b \preceq m \implies \text{Sorted}(a :: m)$ . Hence by Lemma 4.3.10,  $a = c$ . Hence  $m \sim n$ , and so  $b \preceq n$ , by Lemma 4.3.8.

■

### 4.3.3 The proof recast in terms of deliverables

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

However, there is a point of considerable delicacy in the course of this proof. There is a stage at which we must shift from a second-order deliverable over 1,

the identically true predicate on lists, defined by the outermost recursion, to the inner recursion on second-order deliverables defined over the predicate *Sorted*. It appears that we must eliminate the *input* parameter  $l$  using the permutation relation, which moves the condition  $\text{Sorted}(m)$  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

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

to

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

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

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

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

**Lemma 4.3.12 (Exchange Lemma)** *Suppose  $(s, S)$  and  $(t, T)$  are specifications, and that  $(u, Q)$  is a relativised specification with respect to  $(s, S)$ , and  $(u, R)$  is a relativised specification with respect to  $(t, T)$ . Suppose further that  $P$  is a relation on  $s$  and  $t$ ,*

such that  $P \otimes Q \subseteq R$ , where  $\otimes$  is relational composition. Then the following rule is derivable:

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

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

**Proof** See Appendix B. ■

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

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

as we shall see below.

### 4.3.4 The completed development in terms of deliverables

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

The specification for sorting is given by

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

where, as in the previous example,  $\text{Le}$  is the associated propositional level ordering, based on a boolean-valued order function  $\text{le}$ :

```
[A|Type];
[le:A -> A -> bool][Le = [a,b:A]EQ tt (le a b)];
[reflLe:refl Le]
[transLe:trans Le]
[antisymLe:{a,b|A}(Le a b)->(Le b a)->EQ a b];
[linearLe:{a,b:A}or (Le a b) (Le b a)];
```

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

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

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

$$1 \xrightarrow{\Gamma_0} \text{InsertSortSpec} \quad [1]$$

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

The outermost recursion uses the `Listrec2` combinator, which we apply after splitting the initial goal with the composition operator.

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

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

The step case is a little more complicated. We first introduce the parameter `a : A` in the `Listrec2` rule. Recall that the algorithm we considered above contains a `let` construct.

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

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

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

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

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

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

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

```
[consc1 = [A|Type][a:A][S:Rel (list A) (list A)]
          [m,n:list A]S (cons a m) (cons a n)];
[consc1Perm = ... : {A|Type}{a:A}SubRel Perm (consc1 a Perm)];
```

As indicated above, we are now in a position to use the *depListrec<sub>2</sub>* combinator. Again, we preface its application with an appeal to composition.

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

The proof concludes by considering a pointwise construction in the base case of the induction, and then an account of the verification of the `swapcons` function. As in Lemma 4.3.11 above, we distinguish three cases, by induction

on the possible boolean values of the expressions (1e b c) and (1e a b). A more refined analysis of recursion/induction might allow us to consider this inductive step in terms of combinators for deliverables. We have not pursued this, however.

## 4.4 The Chinese remainder theorem

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

### 4.4.1 The mathematical specification

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

#### Theorem 4.4.1 (Chinese remainder theorem)

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

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

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

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

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

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

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

**Lemma 4.4.1 (Euclidean algorithm)**

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

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

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

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

---

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

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

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

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

We turn to the proof of Theorem 4.4.1 above.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 4.4.2 A development in terms of deliverables

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

Accordingly, we wish to extract the above steps  $\iota$ ,  $\epsilon$ ,  $\sigma$  as the function components of three second-order deliverables:

**initialisation** the corresponding proof  $I$  we seek is of the proposition

$$\forall \vec{m}r: \text{list } \mathbb{Z}^2. \forall u: \text{unit}. \text{PairwiseCoprime}(\vec{m}) \implies \text{true} \implies \text{Matrix}_i(\vec{m}r, \iota(\vec{m}r, u))$$

where  $\text{Matrix}_i$  is a representation of the relativised specification

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

That is, we wish to show

$$1 \xrightarrow{(\iota, I)} \text{Matrix}_i \quad [\text{PairwiseCoprime}]$$

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

$$\forall \vec{m}r: \text{list } \mathbb{Z}^2. \forall \vec{p}: \text{list } \mathbb{Z}. \text{true} \implies \text{Matrix}_i(\vec{m}r, \vec{p}) \implies \text{Matrix}_\epsilon(\vec{m}r, \epsilon(\vec{m}r, \vec{p})).$$

The relativised specification  $\text{Matrix}_\epsilon$  is a representation of

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

---

<sup>3</sup> $\delta_{ij}$  is the Kronecker  $\delta$  symbol:

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

The second-order deliverable we seek to derive is

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

**summation** Finally, the summation step is

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

where  $Vector_{\sigma}$  represents

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

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

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

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

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

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

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

## Chapter 5

# Abstract deliverables

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

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

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

speak *about* the structure of the composite objects we call deliverables, which we originally defined *within* the basic type theory.

## 5.1 Construction of first order abstract deliverables

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

### 5.1.1 Hyperdoctrines

**Definition 5.1.1 (Lawvere [54,55])** *hyperdoctrine*

A *hyperdoctrine* is an indexed category  $\mathbf{H}$  over a cartesian closed base category  $\mathbf{C}$ , with cartesian closed fibres, called the *attributes* of  $\mathbf{C}$ , together with the following additional structure :

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

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

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

**Definition 5.1.2** *first-order abstract deliverables*

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

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

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

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

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

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

**Proof** The constructions of Section 3.1 give a syntactic description of a putative proof of this theorem. We use the cartesian closed structure of  $\mathbf{C}$  in the first components of objects in  $\mathcal{G}(\mathbf{H})$ , and the cartesian closed structure in the fibres, together with the quantifiers, is used in defining the predicates (these are the second components of objects in  $\mathcal{G}(\mathbf{H})$ ). We exploit the coherence of the

quantifiers and exponentiation with respect to substitution in the definition of hyperdoctrine to reflect exactly the “logical” constructions of Section 3.1. Indeed, the uniqueness of the categorical structure defined by *bona fide* adjunctions smooths over some of the technical niceties we encountered in ECC.

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

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

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

**Definition 5.1.3** *second-order abstract deliverables*

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

- objects are pairs, of an object  $D$  in  $\mathbf{C}$ , and an attribute  $\psi \in \mathbf{H}[C \times D]$ ,

- arrows are also pairs, where  $(D, \psi) \xrightarrow{(f, F)} (E, \chi)$  if

$$- C \times D \xrightarrow{f} E \text{ in } \mathbf{C}, \text{ and}$$

$$- (\pi_C^* \phi) \wedge \psi \xrightarrow{F} \langle \pi_C, f \rangle^* \chi \text{ in } \mathbf{H}[C \times D].$$

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

### 5.1.2 A programming language with assertions

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

## 5.2 A topos-theoretic model

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

### 5.2.1 Introduction

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

---

<sup>1</sup>We follow the Nijmegen notation, by referring to this system as  $\lambda\text{HOL}$ [77,3].

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

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

Contexts

$$\frac{}{\circ \text{ valid}}$$

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

Type formation

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

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

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

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

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

Basic typing

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

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

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

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

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

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

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

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

## Conversion

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

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

Substitution (derivable in the presence of  $\beta$ -conversion and the above  $\Pi$  rules)

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

### 5.2.2 Modelling dependent types (following Hyland, Pitts)

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

### 5.2.3 Categories with fibrations

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

obvious interaction between contexts and the well-formedness of types, as well as the well-typedness of terms.

**Definition 5.2.1 (Pitts)** *category with fibrations*

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

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

$$\begin{array}{ccc}
 Y \bullet f^*A & \xrightarrow{f \bullet A} & X \bullet A \\
 \downarrow d_{f^*A} & \lrcorner & \downarrow d_A \\
 Y & \xrightarrow{f} & X
 \end{array}$$

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

## 5.2.4 Interpreting types and terms

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

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

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

$$\begin{array}{ccc} X & \xrightarrow{s} & X \bullet A \\ & \searrow id_X & \downarrow d_A \\ & & X \end{array}$$

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

In particular, we may give a semantics for the rule

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

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

$$\begin{array}{ccc} [\Gamma] \bullet [A] \bullet [\Delta] & \xrightarrow{s} & [\Gamma] \bullet [A] \bullet [\Delta] \bullet A' \\ & \searrow id & \downarrow d_{A'} \\ & & [\Gamma] \bullet [A] \bullet [\Delta] \end{array}$$

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

$$\begin{array}{ccccc} [\Gamma] \bullet [A] \bullet [\Delta] & \xrightarrow{\exists! s} & [\Gamma] \bullet [A] \bullet [\Delta] \bullet A' & \longrightarrow & [\Gamma] \bullet A \\ & \searrow id & \downarrow d_{A'} & \lrcorner & \downarrow d_A \\ & & [\Gamma] \bullet [A] \bullet [\Delta] & \longrightarrow & [\Gamma] \\ & & & \xrightarrow{f_\Delta; d_A} & \end{array}$$

where  $A' = (f_\Delta; d_A)^*A$  and  $f_\Delta : \llbracket \Gamma \rrbracket \bullet \llbracket A \rrbracket \bullet \llbracket \Delta \rrbracket \perp \rightarrow \llbracket \Gamma \rrbracket \bullet \llbracket A \rrbracket$  is just a composite of the  $d$ , defined by induction on the length of  $\Delta$ .

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

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

requires that

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

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

$$\begin{array}{ccccc} \Gamma & \xrightarrow{\exists! t} & \Gamma \bullet s^*B & \xrightarrow{s \bullet B} & \Gamma \bullet A \bullet B \\ & \searrow id & \downarrow & \lrcorner & \downarrow d_B \\ & & \Gamma & \xrightarrow{s} & \Gamma \bullet A \\ & & & \searrow id & \downarrow d_A \\ & & & & \Gamma \end{array}$$

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

$$u = \llbracket \Gamma, x:A \vdash b : B \rrbracket : \Gamma \bullet A \longrightarrow \Gamma \bullet A \bullet B.$$

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

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

### 5.2.5 Dependent products in a category with fibrations

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

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

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

$$\begin{array}{ccccc}
 X \bullet A \bullet B & \xleftarrow{\alpha = \rho(\beta)} & Y \bullet f^* A & \xrightarrow{d_{f^* A}} & Y & \xrightarrow{\beta = \lambda(\alpha)} & X \bullet \Pi AB \\
 & \searrow^{d_B} & \downarrow^{f \bullet A} & \lrcorner & \downarrow^f & \searrow^{d_{\Pi AB}} & \\
 & & X \bullet A & \xrightarrow{d_A} & X & & 
 \end{array}$$

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

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

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

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

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

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

## 5.3 The model

We now turn to the details of our topos-theoretic model. As above, we present a base category  $\mathcal{B}$ , together with data for two categories with fibrations over  $\mathcal{B}$ . We use the category  $\mathbf{Sets}$ , of ordinary sets and functions, to illustrate the construction. It should be clear, however, that we make no use of the category-theoretic structure of  $\mathbf{Sets}$  other than its finite limit structure, the cartesian closed structure and the fact that the notion of “subset”, on which the construction

clearly rests, is representable. In other words this construction may be applied to any topos  $\mathcal{E}$  [49,51,53] to yield a model of  $\mathcal{CC}^+$ .

## 5.4 Definition of $\mathcal{B}$

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

- The objects of  $\mathcal{B}$  are pairs  $X = (X_0, X_1)$ , where  $X_0$  is a set and  $X_1$  a subset of  $X_0$ . In the sequel, we shall refer to  $X_0$  as the *carrier* of  $(X_0, X_1)$ , and  $X_1$  as its *distinguished subset*, or even its *predicate*.
- The arrows of  $\mathcal{B}$  with domain  $(Y_0, Y_1)$  and codomain  $(X_0, X_1)$  are functions  $f : Y_0 \dashrightarrow X_0$  such that  $fY_1 \subseteq X_1$ .

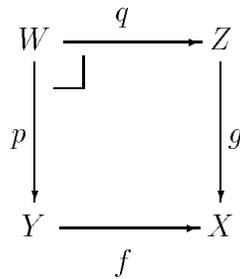
The identity on  $(X_0, X_1)$  is just given by the identity on  $X_0$ , and composition of arrows is just composition of functions: if  $f : Y_0 \dashrightarrow X_0$  and  $g : Z_0 \dashrightarrow Y_0$  satisfy  $fY_1 \subseteq X_1$  and  $gZ_1 \subseteq Y_1$ , then  $(f \circ g)Z_1 = f(gZ_1) \subseteq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

### 5.4.1 Properties of $\mathcal{B}$

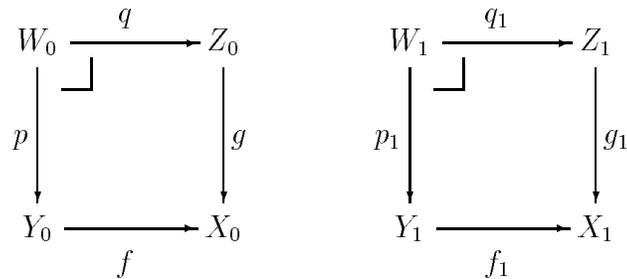
**Lemma 5.4.1**  *$\mathcal{B}$  has finite limits*

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

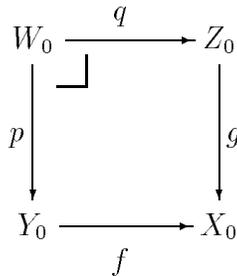
$$W = (W_0, W_1), X = (X_0, X_1), Y = (Y_0, Y_1), Z = (Z_0, Z_1),$$



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



are pullbacks in Sets iff



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

**Remark** We may take

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

with projections

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

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

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

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

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

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

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

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

$$\lambda(f)(z) = \lambda y:Y_0.f(z, y) : Z \perp \rightarrow X^Y$$

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

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

hence

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

■

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

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

We define context extension by

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

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

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

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

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

But this is exactly as required. ■

### 5.4.3 Substitution

**Definition of  $f^*$**  Given  $f : Y \perp \rightarrow X$  in  $\mathbf{B}$ , and  $A$  in  $Fib_T(X)$ , let

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

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

$$f \bullet_T A : Y \bullet_T f^*A \perp \rightarrow X \bullet_T A$$

in  $\mathbf{B}$ , since

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

Moreover,

$$\begin{array}{ccc}
 Y \bullet_T f^* A & \xrightarrow{f \bullet_T A} & X \bullet_T A \\
 \downarrow d_{f^* A} & \lrcorner & \downarrow d_A \\
 Y & \xrightarrow{f} & X
 \end{array}$$

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

$$\begin{array}{ccc}
 Y_0 \times A_0 & \xrightarrow{f \times id_{A_0}} & X_0 \times A_0 \\
 \downarrow \pi_{Y_0} & & \downarrow \pi_{X_0} \\
 Y_0 & \xrightarrow{f} & X_0
 \end{array}$$

is a pullback in **Sets**, and

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

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

$$\begin{array}{ccc}
 & & A \\
 & \swarrow & \searrow \\
 & \blacktriangledown & \\
 & \blacktriangledown & \\
 & \blacktriangledown & \\
 & \blacktriangledown & \\
 A_0 & & X_0
 \end{array}$$

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

### 5.4.4 Dependent products

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

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

The natural isomorphism

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

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

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

### The collection of propositions is a type

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

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

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

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

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

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

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

### 5.4.7 Propositions yield types

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

$$\llbracket T \rrbracket = \llbracket p:Prop \vdash Prf(p) \text{ type} \rrbracket = (1, \{(\phi, u) : \Omega \times 1 \mid \phi = true\}) = (1, \{(\phi, u) : \Omega \times 1 \mid \phi\})$$

Moreover, every proposition is a pullback of  $d_T$  along some map — indeed the classifying map in  $\mathbf{Sets}$  of the subset representing the proposition.

### 5.4.8 Dependent products

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

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

But this is trivial: both sides reduce to

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

## 5.5 Consistency of the model

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

$$\llbracket \circ \vdash \Pi x:Prop.Prf(x) \text{ type} \rrbracket \cong (1, \{u:1 | \forall \phi:\Omega.\phi\}) \cong (1, \emptyset)$$

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

## 5.6 Proof-irrelevance

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

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

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

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

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

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

### 5.6.1 Extensionality

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

$$(\forall x : S. fx = gx) \implies f = g$$

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

$$\textit{Extensionality}: \forall S, T: \textit{Type}. \forall f, g: S \multimap T. (\forall x: S. fx = gx) \implies f = g$$

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

## 5.7 A model of Luo's ECC

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

### 5.7.1 Sums

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

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

Details that this indeed defines a  $\Sigma$ -type are left to the reader.

### 5.7.2 Type universes

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

## 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  $\Gamma$ .

**Definition 6.1.1** *specification*

A *specification* is given by a pair  $s, S$ , where  $\Gamma \vdash s : \text{Type}$  and  $\Gamma \vdash S : s \perp \rightarrow s \perp \rightarrow \text{Prop}$ , such that there are proofs  $\Gamma \vdash \text{sym}S : \forall x, y : s. Sxy \implies Syx$  and  $\Gamma \vdash \text{trans}S : \forall x, y, z : s. Sxy \implies Syz \implies Sxz$ .

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  $\text{sym}S$ ,  $\text{trans}S$  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:

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

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 $\lambda$ -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  $\lambda$ -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  $\lambda$ -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  $\lambda$ -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  $F_\omega$ , 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  $F_\omega$ , 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<sup>1</sup> 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 than 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.

---

<sup>1</sup>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 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<sup>2</sup>.

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.

---

<sup>2</sup>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].

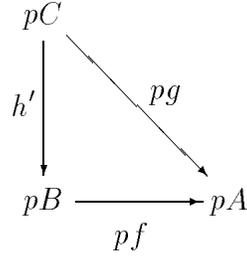
### A.1 Basic definitions

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  $p : \mathcal{F} \rightarrow \mathcal{B}$ . 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

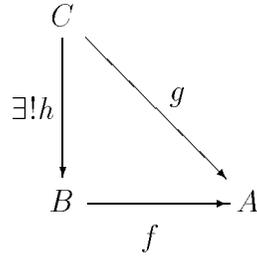
derivations which use the rule of substitution. Since  $\mathcal{B}$  is intended to model well-typed substitutions, this structure should be reflected in  $\mathcal{F}$ .

**Definition A.1.1** *cartesian arrow*

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



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



commutes.

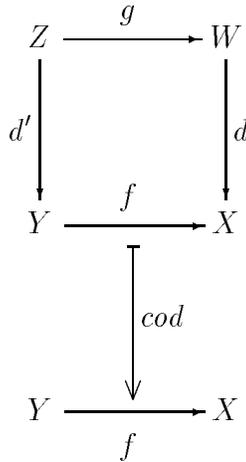
**Definition A.1.2** *fibration*

$p : \mathcal{F} \perp \rightarrow \mathcal{B}$  is a *fibration* if every  $\phi : D \perp \rightarrow pA$  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  $\phi : D \perp \rightarrow pA$  has a lift to  $\mathcal{F}$ , and that every arrow  $g : C \perp \rightarrow A$  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<sup>1</sup>, 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  $d : W \dashrightarrow X$  in  $\mathcal{B}$ , and morphisms those  $(g, f)$  which yield commutative squares, with  $cod$  given by the codomain map:



**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  $(\mathcal{B}, Fib, *, \bullet)$ , we may form the category  $\mathcal{F}(\mathcal{B})$ , whose objects are the display arrows, and whose morphisms

---

<sup>1</sup>This is Euclid's fifth postulate!

$(g, f)$  are the commutative squares:

$$\begin{array}{ccc}
 Y \bullet B & \xrightarrow{g} & X \bullet A \\
 d_B \downarrow & & \downarrow d_A \\
 Y & \xrightarrow{f} & X
 \end{array}$$

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

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

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

## 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  $f : Y \perp \rightarrow X$
- the target type  $B$
- the type  $A$  over which we abstract

The naturality of the  $\Pi$  construction, which concerns variation along the first two parameters, is expressed as follows: given  $g : Z \perp \rightarrow X$  in  $\mathcal{B}$ ,  $A \in \text{Fib}(X)$ ,  $B, C \in \text{Fib}(X \bullet A)$ ,  $\alpha : Y \bullet f^* A \perp \rightarrow (X \bullet A) \bullet B$ ,  $h : (X \bullet A) \bullet B \perp \rightarrow (X \bullet A) \bullet C$  the two

parallel arrows below should be equal

$$\begin{array}{ccccc}
 (X \bullet A) \bullet C & \xleftarrow{g'; \alpha; h} & Z \bullet g; f^*A & \xrightarrow{\quad} & Z & \xrightarrow{\lambda(g'; \alpha; h)} & X \bullet \Pi AC \\
 \uparrow h & & \downarrow \exists!g' & \lrcorner & \downarrow g & & \uparrow \Pi_A(h) \\
 (X \bullet A) \bullet B & \xleftarrow{\alpha} & Y \bullet f^*A & \xrightarrow{\quad} & Y & \xrightarrow{\lambda(\alpha)} & X \bullet \Pi AB \\
 & \searrow d_B & \downarrow & \lrcorner & \downarrow f & & \downarrow d_{\Pi AB} \\
 & & X \bullet A & \xrightarrow{d_A} & X & & 
 \end{array}$$

where

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

In addition, we require that the construction of  $\Pi$  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  $f : Y \perp \rightarrow X$  in  $\mathcal{B}$ ,  $A \in \text{Fib}(X)$ ,  $B \in \text{Fib}(X \bullet A)$ , the induced map

$$\eta : Y \bullet \Pi(f^*A)((f \bullet A)^*B) \perp \rightarrow Y \bullet f^* \Pi AB$$

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

$$\begin{array}{ccccc}
 & & Y \bullet \Pi(f^*A)((f \bullet A)^*B) \bullet d^*(f^*A) & \xrightarrow{d} & Y \bullet \Pi(f^*A)((f \bullet A)^*B) \\
 & & \downarrow d & \lrcorner & \downarrow d \\
 & \rho(id) \swarrow & & & \\
 Y \bullet f^*A \bullet (f \bullet A)^*B & \xrightarrow{d} & Y \bullet f^*A & \xrightarrow{d} & Y \\
 \downarrow (f \bullet A) \bullet B & \lrcorner & \downarrow f \bullet A & \lrcorner & \downarrow f \\
 X \bullet A \bullet B & \xrightarrow{d} & X \bullet A & \xrightarrow{d} & X \\
 \downarrow \alpha; \rho(id); f \bullet A \bullet B & \swarrow d & \downarrow d & \lrcorner & \downarrow d \\
 & & Y \bullet \Pi(f^*A)((f \bullet A)^*B) \bullet (d; f)^*A & \xrightarrow{d} & Y \bullet \Pi(f^*A)((f \bullet A)^*B)
 \end{array}$$

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

$$\alpha : Y \bullet \Pi(f^*A)((f \bullet A)^*B) \bullet (d; f)^*A \xrightarrow{\perp} Y \bullet \Pi(f^*A)((f \bullet A)^*B) \bullet d^*(f^*A)$$

Thus we obtain an arrow

$$\phi = \alpha; \rho(id); f \bullet A \bullet B : Y \bullet \Pi(f^*A)((f \bullet A)^*B) \bullet (d; f)^*A \xrightarrow{\perp} X \bullet A \bullet B$$

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

$$\begin{array}{ccc}
 Y \bullet \Pi(f^*A)((f \bullet A)^*B) & \xrightarrow{\lambda(\phi)} & X \bullet \Pi AB \\
 \downarrow d & & \downarrow d \\
 Y & \xrightarrow{f} & X
 \end{array}$$

Since

$$\begin{array}{ccc}
 Y \bullet f^* \Pi AB & \xrightarrow{\quad} & X \bullet \Pi AB \\
 \downarrow d & \lrcorner & \downarrow d \\
 Y & \xrightarrow{\quad f \quad} & X
 \end{array}$$

is a pullback, we obtain the unique mediating arrow

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

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

$$\begin{array}{ccc}
 Y \bullet \Pi(f^*A)((f \bullet A)^*B) & \xrightarrow{\lambda(\alpha; \rho(id); f \bullet A \bullet B)} & X \bullet \Pi AB \\
 \downarrow d & & \downarrow d \\
 Y & \xrightarrow{\quad f \quad} & X
 \end{array}$$

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];
[tailemma = ... :(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 nilPermma;

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

[atomPermma = ... : {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 atomPermma;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_h1;
intros;Refine res_k1;
intros;Refine inseqns;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;

```



```

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|(fstRel2 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;

```

## B.3 Examples

### B.3.1 A simple example

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

```

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

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

```

### B.3.2 Division by two

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

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

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

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

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

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

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

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

Refine compose_del2;
Refine +3 natrec_del2;

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

```

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

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

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

Save Div2_del2;

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

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

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

```

### B.3.3 Minimum finding in a list

We first give the LEGO script.

```

[minlist:Prop];

[A|Type];

[Inlist [a:A] = listrec false

```

```

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

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

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

intros rba;Immed;
Save linearRlemma;

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

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

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

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

(* step case *)

```

```

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

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

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

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

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

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

```

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

Goal

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

Refine by compose\_del2

```
?3 : Type
```

```
?7 : Rel (list A) ?3
```

```
?9 : del2 (univPred|(list A)) (univRel|(list A)|unit) ?7
```

```
?10 : del2 (univPred|(list A)) ?7 MinSpec
```

Refine 10 by Listrec\_del2

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

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

Refine by pointwise\_del2

```
?21 : {a:list A}{b:unit}<c:A->A>
      (univPred|(list A) a)->
      (univRel|(list A)|unit a b)->
      nstarRel MinSpec a c
```

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

intros (2) l u #

```
l : list A
```

```
u : unit
```

```
?23 : A->A
```

```
?24 : (univPred|(list A) l)->
      (univRel|(list A)|unit l u)->
      nstarRel MinSpec l ?23
```

Intros (3) \_ \_ a

```
h : univPred|(list A) l
```

```
pre : univRel|(list A)|unit l u
```

```
a : A
```

```
?25 : and (Inlist (?23 a) (cons a (nil A)))
          (Lelist R (?23 a) (cons a (nil A)))
```

Refine by pair

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

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

Refine by inl

```
?32 : EQ (?23 a) a
```

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

Refine by reflEQ (I a)

```
?29 : Lelist R (I a) (cons a (nil A))
```

Refine by pair

```

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

```

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

```

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

```

```

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

```

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

```

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

```

```

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

```

```

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

```

Here is the final result

```

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

```

```

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

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

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

```

### B.3.4 Insert sort

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

```
[insertsortdel:Prop];

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

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

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

Discharge P;

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

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

```

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

Discharge s;

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

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

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

Goal del2 ONEL ONELU InsertSortSpec;

Refine compose_del2;
Refine +3 Listrec_del2;
Refine pointwise_del2;

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

(* step case *)

```

```

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

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

Refine exchange_del2;Immed;

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

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

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

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

```

```

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

```

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

```

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

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

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

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

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

Refine pair;Refine pair;Immed;
Refine linearLemma notblec;Refine pair;Immed;

```

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

```

(* ?185 : Lelist Le b k *)

```

```

Refine boolIsInductive (le a b);

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

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

Save InsertSort_del2;

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

```

### B.3.5 The Chinese remainder theorem

#### Technical preliminaries

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

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

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

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

**Definition B.3.1** *Generalised Kronecker  $\delta$*

Suppose given types  $\alpha, \beta$  and relations  $\phi, \psi: \alpha \perp \rightarrow \beta \perp \rightarrow Prop$ . Given two vectors  $\vec{a} =_{\text{def}} \{a_1, \dots, a_n\}$ ,  $(a_i: \alpha)$ ,  $\vec{b} =_{\text{def}} \{b_1, \dots, b_n\}$ ,  $(b_i: \beta)$ , we define the *generalised Kronecker  $\delta$*  to be the matrix of propositions *Kronecker  $\phi \psi (\vec{a}, \vec{b})$* , where

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

We consider the following instances of this general construction:

$Matrix_i$  we take, for  $m, p: nat$ ,

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

$Matrix_e$  we take, for  $m, b: nat$ ,

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

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

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

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

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

**Lemma B.3.1** *zip induction*

Suppose given types  $\alpha, \beta$ , and a three-place relation

$$\Phi : \alpha \perp \rightarrow \beta \perp \rightarrow (\alpha \times \beta) \perp \rightarrow Prop$$

Then the following rule is derivable:

$$\frac{\begin{array}{l} \forall a:\alpha. \forall b:\beta. \forall l:list \alpha. \forall m:list \beta \\ |l| = |m| \implies \Phi l m (zip l m) \implies \\ \Phi nil_\alpha nil_\beta (nil_\alpha, nil_\beta) \quad \Phi (a :: l) (b :: m) ((a, b) :: (zip l m)) \end{array}}{\forall l:list \alpha. \forall m:list \beta. |l| = |m| \implies \Phi l m (zip l m)}$$

Now the Kronecker matrix is definable in LEGO as follows:

```
[Kronecker [A,B|Type][Phi,Psi:Rel A B][as:list A][bs:list B] =
  depListof ([ab:A#B][a = ab.1][b = ab.2][m:list (A#B)]
    and (Phi a b)
```

```

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

(zip as bs)];

```

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

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

```

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

```

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

## The proof

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

```

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

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

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

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

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

Save maplist_multp_preserves_INIT_SPEC;

(* this is the devious initialisation function *)

```

```

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

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

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

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

Goal del1 PCM PairwiseCoprime;
Refine functional_del1;
Save PCM_TO_PC;

Goal del2 PCM ONE_CRT CRT_INIT_SPEC;

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

(* nil case *)
Refine pointwise_del2;

```

```

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

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

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

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

Save recursive_CRT_INIT_del2;

```

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

```

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

```

```

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

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

Intros;Immed;

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

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

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

(* ?31 : ... *)

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

```

```

Refine +1 ih;Immed;

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

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

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

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

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

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

```

```

Refine listwise_divides_multiples;

Refine pair;Refine multDivides';Refine reflDivides;

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

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

Refine maplistPreservesLength;
Refine maplistPreservesLength;
Refine BarPreservesLength;

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

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

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

```

```
Refine init_lnlbars;
```

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

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

```
(* crt_step.1 *)
```

```
(* step case of the CRT *)
```

```
[crt_step:Prop];
```

```
[STEP_SPEC [mrs:list(zed#zed)][xs:list zed]
```

```
  = Kronecker
```

```
    ([mr:zed#zed][x:zed][m = mr.1][r = mr.2]Mod m x r)
```

```
    ([mr:zed#zed][x:zed][m = mr.1]Divides m x)
```

```
    (zip mrs xs)];
```

```
[SUBSCRIPT_CHECK_2 [mrs:list(zed#zed)][xs:list zed]
```

```
  = EQUAL_LENGTH mrs xs];
```

```
[CRT_STEP_SPEC = andRel STEP_SPEC SUBSCRIPT_CHECK_2];
```

```
(* the Euclidean algorithm as a deliverable *)
```

```
[EUCLID:del2 ([mn:zed#zed][m = mn.1][n = mn.2]Coprime m n)
```

```
  (univRel|(zed#zed)|unit)
```

```
  ([mn:zed#zed][ab:zed#zed]
```

```
    [m = mn.1][n = mn.2][a = ab.1][b = ab.2]
```

```
    EQ_zed (plus_zed (mult_zed m a)
```

```
              (mult_zed n b)) one_zed)];
```

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

```
[Euclid [m,n|zed][p = euclid m n]
```

```
  [a = p.1][b = p.2][coprime_mn:Coprime m n] =
```

```
  EUCLID.2 |(m,n) coprime_mn |void top
```

```
  :EQ_zed (plus_zed (mult_zed m a)
```

```
              (mult_zed n b)) one_zed];
```

```

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

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

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

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

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

Save crt_lemma69;

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

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

```

```

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

Save crt_lemma70;

(* with which we resolve the step case *)

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

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

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

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

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

```

```

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

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

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

Refine phi_lmn matrix_lm;

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

Save pointwise_CRT_STEP_del2;

(* crt.1 *)

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

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

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

```

```

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

Refine pointwise_del2;

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

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

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

intros mod_mxr matrix listof_divides_mxs
  listof_divides_msx;Refine pair;

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

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

```

```

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

Save pointwise_CRT_del2;

(* these are the undischarged assumptions on which it depends

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

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

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

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

(* ideal arithmetic *)

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

```

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

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

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

```
[CRT_iota_del2 = recursive_CRT_INIT_del2];

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

[CRT_epsilon_del2
 = pullback_del2_along_del1 PCM_to_1
   pointwise_CRT_STEP_del2          ];

[CRT_sigma_del2
 = pullback_del2_along_del1 PCM_to_1
   pointwise_CRT_del2              ];

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

# Bibliography

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

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

LEGO is available by anonymous ftp.

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

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

- [1] P.Aczel, *An Introduction to Inductive Definitions*, in: *The Handbook of Mathematical Logic*, ed. J.Barwise, North-Holland, Amsterdam 1977.
- [2] R.Backhouse, P.Chisholm, and G.Malcolm, *Do-it-yourself Type Theory*, notes for the International Summer School on Constructive Methods in Computer Science, Marktobendorf 1988.

- [3] H.Barendregt,  *$\lambda$ -calculi with types*, survey article in: *The Handbook of Logic in Computer Science*, eds. S.Abramsky, D.M.Gabbay, and T.S.Maibaum, Oxford University Press, forthcoming.
- [4] E.Bishop, *Foundations of Constructive Mathematics*, *Ergebnisse der Mathematik und ihrer Grenzgebiete, Series 3*, no. 6, Springer-Verlag, 1985.
- [5] J.Bell, *Toposes and Local set theories*, Oxford University Press, 1990.
- [6] J.Bénabou, *Fibred categories and the foundations of naïve category theory*, *JSL*, 1985.
- [7] S. Berardi, *Type Dependence and Constructive Mathematics*, Ph.D. thesis, Dipartimento di Informatica, Torino, Italy 1990.
- [8] E.Bishop, *Foundations of Constructive Analysis*, McGraw-Hill, 1967.
- [9] C.Böhm and A.Berarducci, *Automatic synthesis of typed  $\lambda$ -programs on term algebras*, in: *Theoretical Computer Science*, Vol. 39, North-Holland, Amsterdam, 1985.
- [10] R.Boileau and A.Joyal, *La logique des topos*, *Annals of Pure and Applied Logic*, North-Holland, 1981.
- [11] N.G. de Bruijn, *A survey of the project AUTOMATH*, in: [103].
- [12] R.M.Burstall, *An approach to Program Specification and Development in Constructions*, talk given at the Workshop on Programming Logic, Båstad, Sweden, May 1989. See also the discussion in [78, pp. 96–99].
- [13] R.M.Burstall and J.H.McKinna, *Deliverables: an approach to program development in Constructions*, in [44], also available as a University of Edinburgh technical report ECS-LFCS-91-133.

- [14] A.Church, *A simple theory of types*, Journal of Symbolic Logic, Vol. 5, 1940.
- [15] R.Constable *et al.*, *Implementing Mathematics with the NuPrl Proof Development System*, Prentice-Hall, New Jersey, 1986.
- [16] R.Constable and S.Fraser Smith, *Partial Objects in Constructive Type Theory*, in: Proceedings of the Second LICS Symposium, IEEE, 1987.
- [17] T.Coquand and G.Huet, *Constructions: a Higher-order Proof system for mechanizing mathematics*, in: Proceedings EUROCAL '85, LNCS 203, Springer-Verlag, 1985.
- [18] T.Coquand and G.Huet, *The Theory of Constructions*, in: Information and Computation, Vol. 76, nos. 2/3, Academic Press, 1988.
- [19] T.Coquand, *Metamathematical Investigations of a Calculus of Constructions*, in: [43].
- [20] T.Coquand and C.Paulin-Mohring, *Inductively defined types*, in: [43].
- [21] P-L.Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman Research Notes in Theoretical Computer Science, Pitman, London, 1986.
- [22] E.W.Dijkstra, *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*, in: Communications of the ACM, Vol. 18, 1975.
- [23] E.W.Dijkstra *A Discipline of Programming*, Prentice-Hall, New Jersey, 1976.
- [24] E.W.Dijkstra, *Selected writings on Computing*, Springer-Verlag, 1982.

- [25] E.W.Dijkstra, C.S.Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.
- [26] T.Ehrard, *A Categorical Semantics of Constructions*, in: Proceedings of the Third LICS Symposium, IEEE, 1988.
- [27] S.Feferman, *Formal Theories for Transfinite Iterations of Generalised Inductive definitions and some subsystems of Analysis*, in: Intuitionism and Proof Theory, ed.s A.Kino, J.Myhill and R.E.Vesley, North-Holland, Amsterdam 1970.
- [28] M.P.Fourman, *The logic of topoi*, in: The Handbook of Mathematical Logic, ed. J.Barwise, North-Holland, 1977.
- [29] P.J.Freyd and A.Scedrov, *Categories and Allegories*, North-Holland, Amsterdam, 1990.
- [30] P.A.Gardner, *Representing Logics in Type Theory*, Ph.D. thesis, University of Edinburgh, 1992.
- [31] H.Geuvers, *The Church-Rosser property for  $\beta\eta$ -reduction in typed  $\lambda$ -calculi*, draft, December 1991.
- [32] J-Y.Girard, *Interpretation fonctionnelle et élimination des coupures dans l'arithmétique de l'ordre supérieure*, thesis, University of Paris VII, 1972.
- [33] H.Goguen and Z.Luo, *Inductive data types: Well-ordering types revisited*, manuscript submitted to Proceedings of the Second Workshop on Logical Frameworks, 1991.
- [34] R.Harper, F.Honsell and G.D.Plotkin, *A Framework for Defining Logics*, Journal of the ACM, to appear.

- [35] R.Harper, R.J.G.Milner and M.Tofte, *The Definition of Standard ML, Version 3*, Technical Report ECS-LFCS-89-91, LFCS, Dept. of Computer Science, University of Edinburgh, 1989.
- [36] R.Harper and R.A.Pollack, *Type checking with universes*, in: Theoretical Computer Science, Vol. 89, North-Holland, Amsterdam, 1991.
- [37] S.Hayashi, *Adjunction of semifunctors: categorical structures in nonextensional lambda calculus*, in Theoretical Computer Science, Vol. 41, North-Holland, Amsterdam, 1985.
- [38] S.Hayashi, *ATT: Optimised Curry-Howard isomorphism for Program Extraction*, talk given at First Annual BRA Workshop on Logical Frameworks, Antibes, May 1990.
- [39] S.Hayashi, *Singleton, Union and Intersection Types for Program Extraction*, in: Proceedings of TACS '91, Sendai, Japan, Springer LNCS 526, Springer-Verlag, 1991.
- [40] C.A.R.Hoare, *An axiomatic basis for computer programming*, in: Communications of the ACM, Vol. 12, 1969.
- [41] C.A.R.Hoare, *Data refinement in a categorical setting*, Draft, Oxford 1987.
- [42] W.A.Howard, *The "formulae-as-types" notion of construction*, in: [103].
- [43] G.Huet, T.Coquand, C.Paulin-Mohring *et al.*, *The Calculus of Constructions, Version 4.10, Documentation and user's manual*, Rapports Techniques no.110, Projet Formel, INRIA-Rocquencourt, Paris, August 1989.

- [44] G.Huet and G.Plotkin, eds. *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks, Antibes, May 1990*, distributed electronically to participating BRA sites, January 1991.
- [45] J.M.E.Hyland and A.M.Pitts, *The Theory of Constructions: Categorical Semantics and Topos-theoretic models*, in: Proceedings of the AMS Conference on Categories in Computer Science, Boulder, Colorado, 1986.
- [46] J.M.E.Hyland, E.P.Robinson and G.Rosolini, *The discrete objects in the effective topos*, Proceedings of the London Mathematical Society, Vol. 60, pp.1–36, Jan 1991.
- [47] K.Ireland and M.Rosen, *A classical introduction to modern number theory*, second edition, Springer Graduate Texts in Mathematics no. 84, Springer-Verlag, 1990.
- [48] B.P.F.Jacobs, *Categorical Type Theory*, proefschrift, University of Nijmegen, 1991.
- [49] P.T.Johnstone, *Topos Theory*, Academic Press, London, 1977.
- [50] P.T.Johnstone and R.Paré, eds., *Indexed Categories and their Applications*, Springer LNM 661, Springer-Verlag, 1978.
- [51] A.Kock and G.Wraith, *Elementary Toposes*, Aarhus Lecture Notes no. 30, Aarhus Universitet, Denmark, 1971.
- [52] G.Kreisel, *Functions, Ordinals, Species*, in: Logic, Philosophy and Methodology of Science III, eds. B.van Rootselaar and J.Staal, North-Holland, Amsterdam, 1982.

- [53] J.Lambek and P.J.Scott, *An Introduction to Higher-Order Categorical Logic*, Cambridge Studies in Advanced Mathematics no. 7, Cambridge University Press, Cambridge, England, 1986.
- [54] F.W.Lawvere, *Adjointness in foundations*, *Dialectica* 23, 1969.
- [55] F.W.Lawvere, *Equality in hyperdoctrines and the comprehension schema as an adjoint functor*, in: Proceedings of the AMS symposium on Applications of Category Theory, AMS, Providence R.I., 1970.
- [56] D.Leivant, *Reasoning about functional programs and complexity classes associated with type disciplines*, in: Proceedings of the 24th IEEE symposium on Foundations of Computer Science, 1983.
- [57] Z.Luo, *An Higher-order Calculus and Theory Abstraction*, Technical report ECS-LFCS-88-57, Department of Computer Science, University of Edinburgh, 1988.
- [58] Z.Luo, *ECC, an Extended Calculus of Constructions*, in: Proceedings of the Fourth IEEE Conference on Logic in Computer Science, Asilomar, California, 1989.
- [59] Z.Luo, *An Extended Calculus of Constructions*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh, June 1990.
- [60] Z.Luo, *A problem of adequacy: conservativity of Calculus of Constructions over higher-order logic*, Technical Report ECS-LFCS-90-121, Department of Computer Science, University of Edinburgh, October 1990.
- [61] Z.Luo, *Program Specification and Data Refinement in Type Theory*, Technical Report ECS-LFCS-90-131, Department of Computer Science, University of Edinburgh, January 1991.

- [62] Z.Luo, *An Higher-order Calculus and Theory Abstraction*, Information and Computation, Vol. 90, No. 1, 1991.
- [63] Z.Luo, *A unifying theory of dependent types I*, Technical Report ECS-LFCS-91-154, Department of Computer Science, University of Edinburgh, 1991.
- [64] Z.Luo, R.A.Pollack and P.Taylor, *How To Use LEGO (A Preliminary User's Manual)*, LFCS Technical Note LFCS-TN-27, October 1989.
- [65] Z.Luo and R.A.Pollack, *LEGO Proof Development System: User's Manual*, Technical Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, 1992.
- [66] J.H.McKinna, *On permutation*, manuscript, Edinburgh, 1991.
- [67] Saunders Mac Lane, *Categories for the Working Mathematician*, Springer Graduate Texts in Mathematics, no. 5, Springer-Verlag, 1971.
- [68] D.MacQueen, *Using dependent types to express modular structure*, in: Proceedings POPL 13 , 1986.
- [69] P.Martin-Löf, *An Intuitionistic Theory of Types: Predicative part*, in: Logic Colloquium 73, North-Holland, Amsterdam, 1975.
- [70] P.Martin-Löf, *Constructive Mathematics and Computer Programming*, in: proceedings of the Conference on Logic, Philosophy and Methodology of Science VI, 1979, North-Holland, Amsterdam, 1982.
- [71] P.Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, Naples, 1984.
- [72] P.Martin-Löf, *On the meaning of the logical constants, and the justification of the logical laws*, Technical Report 2, Scuola di Specializzazione in Logica Matematica, Università di Siena, 1985.

- [73] I.Mason, *Hoare's Logic in the LF*, Technical Report ECS-LFCS-87-32, Department of Computer Science, University of Edinburgh, 1987.
- [74] M.Mendler, *The Logic of Design*, Ph.D. thesis, University of Edinburgh, forthcoming, 1992.
- [75] E.Moggi, *The partial  $\lambda$ -calculus*, Ph.D. thesis, University of Edinburgh, available as LFCS report ECS-LFCS-88-63, 1988.
- [76] E.Moggi, *A category-theoretic account of program modules*, in: *Mathematical Structures in Computer Science*, Vol.1, Cambridge University Press, 1991.
- [77] H.Barendregt, E.Barendsen, W.Dekkers, H.Geuvers, B.P.F.Jacobs, lecture notes for the Summer School in  $\lambda$ -calculus, Nijmegen, 1991.
- [78] P.Dybjer, L.Hallnäs, B.Nordström, K.Petersson, and J.Smith, editors, *Proceedings of the workshop on Programming Logic*, Programming Methodology Group Report no. 54, University of Göteborg and Chalmers University of Technology, May 1989.
- [79] B.Nordström, K.Petersson, and J.Smith, *Programming in Martin-Löf's type theory*, Oxford University Press, 1990.
- [80] C-E.Ore, *The ECC extended with inductive types*, draft, Edinburgh, 1989.
- [81] C.Paulin-Mohring, *Extracting  $F_\omega$ 's programs from proofs in the Calculus of Constructions*, in: *Proceedings POPL89*, ACM, 1989.
- [82] C.Paulin-Mohring and B.Werner, *Extracting and Executing Programs developed in the Inductive Constructions System: a Progress Report*, in: [44].

- [83] D.Pavlovič, *Predicates and Fibrations*, proefschrift, University of Utrecht, 1990.
- [84] A.M.Pitts, *Categorical Semantics of Dependent Types*, talk given at SRI International, Menlo Park, California, June 1989, and at the Logic Colloquium, Berlin, 1990.
- [85] R.A.Pollack, *The theory of LEGO*, manuscript, Edinburgh, 1989.
- [86] R.A.Pollack, *Implicit Syntax*, in: [44].
- [87] R.A.Pollack, *Implicit Syntax*, draft of a paper given at the LogFIT Summer School in Proof Theory, Leeds, UK, July 1990.
- [88] A.J.Power, *An algebraic formulation of data refinement*, in: Proceedings of MFPS '89, Tulane University, Louisiana, Springer LNCS 442, Springer-Verlag, 1990.
- [89] D.Prawitz, *Natural Deduction: a Proof-Theoretic Study*, Almqvist and Wiksell, Stockholm, 1965.
- [90] J.C.Reynolds, *Types, abstraction and parametric polymorphism*, in: Information Processing '83, ed. R.E.A.Mason, North-Holland, 1983.
- [91] J.C.Reynolds and Qing-Ming Ma, *Parametric polymorphism revisited*, paper presented at the LMS Symposium on Category Theory and Computer Science, Durham, 1991.
- [92] A.Salvesen and J.Smith, *On the strength of the subset type in Martin-Löf's type theory*, in: Proceedings of the Third LICS Symposium, IEEE, 1988.

- [93] A.Salvesen, *On Information Discharging and Retrieval in Martin-Löf's type theory*, Ph.D. thesis, Institute of Informatics, University of Oslo, 1989.
- [94] A.Salvesen, *The Church-Rosser Property for the LF with  $\beta\eta$ -reduction*, talk given at the First Annual BRA Workshop on Logical Frameworks, Antibes, May 1990.
- [95] A.Salvesen, *The Church-Rosser Property for Pure Type Systems with  $\beta\eta$ -reduction*, manuscript, November 1991.
- [96] D.Sannella, *Formal specification of ML programs*, LFCS technical report ECS-LFCS-86-15, Dept. of Computer Science, University of Edinburgh, 1986.
- [97] D.Sannella and A.Tarlecki, *Towards formal development of ML programs: foundations and methodology*, LFCS technical report ECS-LFCS-89-71, Dept. of Computer Science, University of Edinburgh, 1989.
- [98] D.S.Scott, *Constructive validity*, in: Proceedings of the Symposium on Automatic Demonstration, IRIA, Rennes, Springer LNM 125, Springer-Verlag, 1970.
- [99] D.S.Scott, *Identity and Existence in Intuitionistic Logic*, in: Proceedings of the LMS Symposium on Applications of Sheaves, Durham 1977, Springer LNM 753, Springer-Verlag, 1977.
- [100] D.S.Scott, *Relating Theories of the  $\lambda$ -calculus*, in: [103].
- [101] R.A.G.Seely, *Hyperdoctrines, Natural Deduction and the Beck Condition*, in: Zeitschrift für Mathematische Logik und Grundlagen, Vol. 29, 1983.

- [102] R.A.G.Seely, *Locally Cartesian Closed Categories and Type Theory*, in: Mathematical Proceedings of the Cambridge Philosophical Society, Vol. 95, 1984.
- [103] J.P.Seldin and J.R.Hindley, eds., *To H.B.Curry, essays in Combinatory Logic,  $\lambda$ -calculus and Formalism*, Academic Press, 1980.
- [104] S.Stenlund, *Combinators,  $\lambda$ -calculus and Proof Theory*, D.Reidel, Dordrecht, 1972.
- [105] T.Streicher, *Correctness and completeness of a categorical semantics of Constructions*, thesis, Passau, 1989.
- [106] P.Taylor, `diagrams3.tex`, available from `pt@doc.ic.ac.uk`.
- [107] A.S.Troelstra and D. van Dalen *Constructivism in Mathematics I+II*, North-Holland, Amsterdam, 1988.
- [108] P.Wadler, *Theorems for Free!*, in: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, London, ACM, 1989.