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 Σ-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

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

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

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

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

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

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

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 λ-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 λ-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[1]. 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 ϕ_A, defined by induction on the type A, with the function clause essentially $\phi_{A \to B}(f) = \forall a{:}A.\phi(a) \Rightarrow \phi(f\,a)$. 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

[1] 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 Ω , 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 λ-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 Σ-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 Σ-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 $\mathcal{S}$, the category of second-order deliverables over $\mathcal{S}$ 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_ω.*

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_ω. The proof is based on a syntactic mapping, the so-called Berardi-Mohring projection. Mohring used this mapping as an extraction function, which, coupled with the associated realisability predicate, allows a powerful and flexible approach to program development from proofs. Under this interpretation, an arbitrary type is interpreted as a type, together with a predicate (the realisability predicate) defined over it.

The approach taken in [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" $\{x \in A|B(x)\}$, 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 $P\ \ \mathbf{true}$ 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

$$A\ \mathbf{set}$$

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

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

This corresponds to our definition of specification in Section 3.1.

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

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

The NuPrl system, developed by Constable and his co-workers [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 Π, Σ, on which most work to date on type theory has been based, in favour of a more set-theoretic style, with union, intersection and singleton types [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 Σ-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

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

where V ranges over some infinite collection of variables, and κ ranges over $Prop$ and $Type_i$ $(i \in \omega)$, the so-called *kinds* of ECC. $Prop$ is an impredicative universe, as in Coquand and Huet's original systems [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:

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

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

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 Π-types[1] and Σ-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 Γ is a context and M, A are terms. If Γ 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 Π in instances of rule $(\Pi 1)$ in Table 2–1, to emphasise the distinction between the logical quantifier and the dependent type constructor. For Σ-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 $(\Pi 1)$ is very strong. It allows us to make the usual higher-order definitions of the logical constants and connectives for intuitionistic logic, for example,

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

---

[1]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.

$$(axiom) \qquad \frac{}{\circ \vdash Prop : Type_0}$$

$$(wcon) \qquad \frac{\Gamma \vdash A : \kappa}{\Gamma, x{:}A \vdash Prop : Type_0} \; (x \notin FV(\Gamma), \kappa \text{ a kind})$$

$$(type) \qquad \frac{\Gamma \vdash Prop : Type_0}{\Gamma \vdash Type_i : Type_{i+1}} \; (i \in \omega)$$

$$(var) \qquad \frac{\Gamma, x{:}A, \Delta \vdash Prop : Type_0}{\Gamma, x{:}A, \Delta \vdash x : A}$$

$$(\Pi 1) \qquad \frac{\Gamma, x{:}A \vdash P : Prop}{\Gamma \vdash \Pi x{:}A.P : Prop}$$

$$(\Pi 2) \qquad \frac{\Gamma \vdash A : Type_i \quad \Gamma, x{:}A \vdash B : Type_i}{\Gamma \vdash \Pi x{:}A.B : Type_i} (i \in \omega)$$

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

$$(app) \qquad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}$$

$$(\Sigma) \qquad \frac{\Gamma \vdash A : Type_i \quad \Gamma, x{:}A \vdash B : Type_i}{\Gamma \vdash \Sigma x{:}A.B : Type_i} (i \in \omega)$$

$$(pair) \qquad \frac{\Gamma \vdash M : A \quad \Gamma, x{:}A \vdash N : [M/x]B \quad \Gamma, x{:}A \vdash B : Type_i}{\Gamma \vdash \mathbf{pair}_{\Sigma x:A.B}(M, N) : \Sigma x{:}A.B}$$

$$(\pi 1) \qquad \frac{\Gamma \vdash M : \Sigma x{:}A.B}{\Gamma \vdash \pi_1(M) : A}$$

$$(\pi 2) \qquad \frac{\Gamma \vdash M : \Sigma x{:}A.B}{\Gamma \vdash \pi_2(M) : [\pi_1(M)/x]B}$$

$$(\preceq) \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : Type_i}{\Gamma \vdash M : B} \; (A \preceq B)$$

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 $\circ \vdash M : \bot$ [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 $\Gamma_{nat} = nat{:}\,Type_0, 0{:}nat, S{:}nat \to nat$,

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

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

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

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 λ-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 $i < j$,

$$\vdash Type_i \to Type_i : Type_j$$

and

$$\vdash \lambda\tau{:}\,Type_i.\lambda x{:}\tau.x : Type_i \to Type_i,$$

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 Σ-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.

Σ-types In context $\Gamma_{nat}, \Phi{:}nat \rightarrow Prop$, we may use Σ-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 Σ-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 λ-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 λ-expressions to which they give rise. Where appropriate,

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

where

$$\begin{array}{rcl} plustwo & = & \lambda n{:}nat\,.\,S(Sn) \\ proof & = & \lambda n{:}nat\,.\,\lambda p{:}P(n)\,. \\ & & \quad \lambda\Phi{:}nat \rightarrow Prop. \\ & & \qquad \lambda z{:}\Phi(0)\,.\,\lambda s{:}\forall k{:}nat.\Phi(k) \Rightarrow \Phi(S(Sk))\,. \\ & & \qquad\quad s\,n\,(p\,\Phi\,z\,s) \end{array}$$

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 Γ are strongly normalising;

Subject Reduction (SR) If $\Gamma \vdash M : A$ and $M \rhd N$ 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 $(\Pi 1)$ in Table 2–1 allows us to define Leibniz' equality (cf. [59, p. 119]):

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

where $\Gamma \vdash \tau : \kappa$ for some kind κ. 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

$$\mathit{reflEQ} =_{\text{def}} \lambda x{:}A.\lambda P{:}A \to Prop.\lambda h{:}Px.h \ : \ \forall x{:}A.EQ_A\ x\ x$$

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 reflEQ. 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 Γ, are interconvertible, it is necessary and sufficient that

$$\Gamma \vdash \mathit{reflEQ}\ t_1 = \lambda P{:}\tau \rightarrow \mathit{Prop}.\lambda h{:}Pt_1.h : EQ_\tau\ t_1\ t_2$$

be derivable, where τ 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

`T` ::= `Prop` | `Type(`i`)` $(i \in \omega)$ | `V`
    `{V:T}T` | `[V:T]T` | `TT` |
    `<V:T>T` | `(T,T)` | `T.1` | `T.2`

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

Table 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 Γ, 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 (ΠI), (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 β-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 δ-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,

```
[t:Type][x:t]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:

```
[EQ = [t|Type][x,y:t]{P:t -> Prop}(P x) -> P y];
```

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

```
[I = [t|Type][x:t]x];
```

then the self-application `I 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 δ-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 δ-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 δ-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* `t1`, `t2` *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 Γ_{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 Σ-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 Σ-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 Π-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 Π-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 Σ-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[2].

```
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;
```

[2]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.