Inductively defined types

Thierry Coquand and Christine Paulin

Preliminary version

Introduction

The aim of this paper is to present an extension of Church's simple type theory (see [5, 20]) where we add inductively defined types. This will introduce the notion of "concrete data types" (like natural numbers, lists, trees,...) in higher-order logic. One main point is to use the "proposition-as-types" paradigm in order to derive uniformly recursion operators and induction principles. Our treatment differs from the one of [14, 8, 19] in that we don't use the notion of subtypes. P. Dybjer has came out independently to a quite close type system, but in a predicative setting (see [9]), and he pointed out to us that R. Backhouse had been suggesting similar ideas [2]. We want to thank him for many interesting discussions. Some discussions with P. Martin-Löf have also been crucial for a lot of results (in particular the "isomorphism theorem" and the importance of the notion of definitional equality).

We restrict mainly our paper to the case of inductively defined *types* (by opposition to inductively defined predicates or relations). After giving some motivations, we present a possible way to implement inductive types in a system like the calculus of constructions. We present then an alternative formulation, that we get by adding a disjoint sum and an "initial algebra" forming operation for strictly positive operators in the system ECC of [16]. We show that we can translate inductively defined types in this system, and explain in what way this coding is satisfactory (it gives the right "definitional equalities").

This alternative framework is used to illustrate the basic features of our extension, in particular one result that we call the isomorphism theorem, which is not satisfied by the impredicative coding of inductive types (see [21]). We show how to give a set-theoretic semantics of the predicative part of this extension, and sketch how to prove normalisation for proofs. We show also that we cannot extend this treatment to an arbitrary positive operators in an impredicative framework, but we conjecture that such an extension to any positive operator is possible in a predicative framework. At the end of the paper, one example tried on a computer is commented.

This paper is rather incomplete and not really homogeneous. In particular, there is still a huge gap between the theoretical part and what is implemented, since the theoretical part treats only the case of inductive types. It is hoped that further piecemeal reports, putting right these defects, will appear elsewhere.

1 An implementation of inductively defined types

The system we use is the version of the calculus of construction extended with a cumulative hierarchy of universes (see [7, 16]). We will use the symbol U to denote one of the universe U_i or the type Prop of propositions (which can be seen as the smallest universe).

1.1 Generalized inductive types

Definitions. Let Φ be such that $\Phi(X) : U \ \ [X : U]$. We say that Φ is *strictly positive* if X does not occur in $\Phi(X)$ or if $\Phi(X) = X$ or if there exists a type K in which X does not occur and a strictly positive operator Φ_0 such that $\Phi(X) = (x : K)\Phi_0(X)$.

Given Φ such that X occurs in $\Phi(X)$ and given $X : U$ and $P : (X)U$, we define the *derivative* $\Phi'(X, P) : (\Phi(X))U$ as follows: if $\Phi(X) = X$ then $\Phi'(X, P) = P$, and if $\Phi(X) = (K)\Phi_0(X)$, then $\Phi'(X, P)(f) = (x : K)\Phi'_0(X, P)(f(x))$.

It is then standard that Φ extends functorially on terms. If $f : (X)Y$, we can, by induction on Φ, define $\Phi(f) : (\Phi(X))\Phi(Y)$. We have furthermore a "dependent functoriality": if $P : (X)U$ and $f : (x : X)P(x)$, we can define the morphism $\Phi(f) : (z : \Phi(X))\Phi'(X, P, z)$. This is also defined directly by induction on Φ.

Let Θ be such that $\Theta(X) : U \ \ [X : U]$. We say that Θ is a *constructor* if $\Theta(X) = X$ or if there exists a constructor Θ_0 such that $\Theta(X) = (x : \Phi(X))\Theta_0(X)$ with Φ a strictly positive operator.

We associate to a constructor Θ, an *arity* that is a list of pairs of variables and strictly positive operators and a recursive arity that is a list of strictly positive operators. If $\Theta(X) = X$ then its arity and recursive arity are equal to the empty list. If $\Theta(X) = (x : \Phi(X))\Theta_0(X)$ and if the arity of Θ_0 is l and its recursive arity is m then the arity of Θ is $(x, \Phi) :: l$ and its recursive arity is $\Phi :: m$ if X occurs in $\Phi(X)$ and m if not.

A *generalized inductive type* will be specified by a list of constructors $[\Theta_1, \ldots, \Theta_n]$. It will be denoted by $\mu[\Theta_1, \ldots, \Theta_n]$.

Let $A = \mu[\Theta_1, \ldots, \Theta_n]$ be a generalized inductive type. We extend the theory with a rule of formation :

$$\mu[\Theta_1, \ldots, \Theta_n] : U$$

and n introduction rules, for $1 \leq i \leq n$:

$$\mathsf{intro}_i : \Theta_i(A)$$

To build the elimination rule, we need a new definition : Let Θ be a constructor of arity and

recursive arity respectively :

$$[(x_1, \Phi_1), \ldots, (x_k, \Phi_k)] \quad \text{and} \quad [\Phi_{i_1}, \ldots, \Phi_{i_p}]$$

Let X be of type U, P be of type $(X)U$, u be of type $\Theta(X)$ and f be of type $(x : X)P(x)$. We define a new type $\Theta'(X, P, u)$, which is defined as

$$(x_1 : \Phi_1(X)) \ldots (x_k : \Phi_k(X))(\Phi'_{i_1}(X, P, x_{i_1})) \ldots (\Phi'_{i_p}(X, P, x_{i_p}))P(u(x_1, \ldots, x_k))$$

The elimination rule is :

$$\frac{P : (A)U \quad f_1 : \Theta'_1(A, P, intro_1) \quad \ldots \quad f_n : \Theta'_n(A, P, intro_n)}{\mathsf{rec}(f_1, \ldots, f_n) : (z : A)P(z)}$$

Finally we add n rules of conversion. For $1 \leq i \leq n$ if Θ_i is a constructor of arity $[(x_1, \Phi_1), \ldots, (x_k, \Phi_k)]$ and of recursive arity $[\Phi_{i_1}, \ldots, \Phi_{i_p}]$ the rule of conversion will be with $R = \mathsf{rec}(f_1, \ldots, f_n)$:

$$R(\mathsf{intro}_i(t_1, \ldots, t_k)) = f_i(t_1, \ldots, t_k, \Phi_{i_1}(R, t_{i_1}), \ldots, \Phi_{i_p}(R, t_{i_p}))$$

Remark 1. In order for the theory to be consistent (and in order to get the normalisation property), we have to put some restriction on the universes. We choose some restriction in such a way that we can extend the p.e.r. interpretion (cf. for instance [16]) to inductive types. We will ask that $\mu[\Theta_1, \ldots, \Theta_n] : U_i$ if $\Theta_k(X) : U_i$ for $X : U_i$. If we want $\mu[\Theta_1, \ldots, \Theta_n] : \mathsf{Prop}$ we ask that, if we write $\Theta_k(X) = (\Phi^k_1(X)) \ldots (\Phi^k_{n_k}(X))X$, for each $1 \leq i \leq n_k$, we have $\Phi^k_i(X) : \mathsf{Prop}$ for $X : \mathsf{Prop}$.

Examples. We can recover the usual type constructors of Type Theory

$$\begin{array}{lcl} A \times B & = & \mu[\lambda X.(A)(B)X] \\ (\Sigma x : A)B(x) & = & \mu[\lambda X.(x : A)(B(x))X] \\ A + B & = & \mu[\lambda X.(A)X, \lambda X.(B)X] \\ \mathsf{Unit} & = & \mu[\lambda X.X] \\ \mu(\Phi) & = & \mu[\lambda X.(\Phi(X))X] \end{array}$$

Furthermore we get directly the right introduction and elimination rules for integers by defining :

$$\mathsf{Nat} = \mu[\lambda X.X, \lambda X.(X)X]$$

Indeed, over the type $\mathsf{Nat} = \mu[\lambda X.X, \lambda X.(X)X]$, we have two constructors $O : \mathsf{Nat}$ and $S : (\mathsf{Nat})\mathsf{Nat}$, and the elimination rule is the usual

$$\frac{P : (\mathsf{Nat})U \quad a : P(O) \quad f : (x : \mathsf{Nat})(P(x))P(S(x))}{\mathsf{rec}(a, f) : (x : \mathsf{Nat})P(x)}$$

whereas the computation rules are $\mathsf{rec}(a, f, O) = a$ and $\mathsf{rec}(a, f, S(n)) = f(n, \mathsf{rec}(a, f, n))$. Notice that the elimination rule subsumes both the possibility of defining inductively objects over natural numbers, and of doing proofs by induction (with an explicit notation for the proofs).

Counter-Example. If $A : U_1$ and $B : (A)\mathsf{Prop}$, we can consider $\Theta(X) : (x : A)(B(x))X$, and we have that $\Theta(X) : \mathsf{Prop}$ for $X : \mathsf{Prop}$, but we cannot ask $\mu[\Theta] : \mathsf{Prop}$ since then we get then an inconsistency from the elimination and computation rules (since this is equivalent to adding a strong impredicative sum, cf. for instance [7]).

Remark 2. If X occurs strictly in $\Phi(X)$ then Φ is strictly positive. It is possible to define what it means for a variable Y to be strictly positive in a generalized inductive type. it is just that each operator Φ in the arity of a constructor is such that $\Phi(X) = \Psi(Y)$ with Ψ a strictly positive operator (this implies that either X or Y does not occur in $\Phi(X) = \Psi(Y)$). This possibility allows for example to define the type $\mu(\lambda X.\mathsf{List}(X))$. But it complicates the definition of $\Phi'(P)$ and is not very useful in practice. The actual implementation does not allow such recursive definition.

1.2 Impredicative inductive types

It is well-known that inductive types like the ones presented before are definable in second-order λ-calculus (see for instance [4]). The nice property of the impredicative coding is that it is an internal definition and that we do not need new constants or reduction rules.

The main drawback of the impredicative approach is the impossibility of proving the induction principle. What we get easily is a proof of the non-dependent version of the induction principle obtained with $P = \lambda x.C$ of type $(A)U$ with x not occuring in C.

This gives for the disjoint sum or the integers the following rules :

$$\frac{C : U \quad f : (A)C \quad g : (B)C}{\mathsf{case}(f, g) : (A + B)C}$$

$$\frac{C : U \quad x : C \quad f : (\mathsf{Nat})(C)C}{\mathsf{rec}(x, f) : (\mathsf{Nat})C}$$

This is sufficient at the "programming level" and the dependent elimination rule may be just used for doing proofs and then has only to be proved consistent with the theory.

But even at the programming level, this representation has another drawback. We are expecting for integers the conversion rule :

$$\mathsf{rec}(x, f)(S(n)) = f(n, \mathsf{rec}(x, f)(n))$$

but we really get the following conversion :

$$\mathsf{rec}(x, f)(S(n)) = f(\varphi(n), \mathsf{rec}(x, f)(n))$$

with $\varphi(n)$ a term such that $\varphi(0) = 0$ and $\varphi(S(p)) = S(\varphi(p))$. By induction, it is possible to prove that $\varphi(n)$ and n are intensionally equal, that is that $(n : N)I(N, \varphi(n), n)$. But, in general these terms are not convertible. This feature seems to lead to inefficient programs where expressions like $\varphi^k(n)$ are generated. Then k reductions are needed for reducing $\varphi^k(S(p))$ to $S(\varphi^k(p))$.

An interesting feature of primitive inductive types is the ability to define a proposition by induction. It is then possible to prove $0 \neq 1$ that is not provable in the impredicative system.

For example, it is possible to define a predicate even on natural numbers such that :

$$\mathsf{even}(0) = \mathsf{True} \qquad \mathsf{even}(S(p)) = \neg\mathsf{even}(p)$$

An impredicative definition of the predicate even will be to say that even is the smallest predicate such that even(0) and even(p)$\Rightarrow$even($S(S(p))$) but this is really another kind of definition. It is easy to prove that this second definition implies the first one but the opposite seems hard. Intuitively a proof of even(p) in the second case has a computational meaning, it is isomorphic to the integer $(p/2)$.

1.3 Inductively defined predicates and relations

It is also possible to extend the system we have presented with a "primitive" definition of inductive predicates and relations.

In the same way that this direct representation has better operational properties, it appears that using this notion of inductively defined predicates and relations lead to "better" proofs (see the example at the end).

It is possible to imagine examples of inductive predicates with a computational meaning. For example a predicate listn of type (Nat)U such that listn(n) is the type of lists of length n. This will be specified by :

$$\mathsf{listn} = \mu.[\lambda X.X(0), \lambda X.(p : \mathsf{Nat})(A)(X(p))X(S(p))]$$

The definition of the previous section are not very hard to extend in order to get the rules for predicates or relations.

1.4 Parameters

We are defining :

$$A \times B = \mu[\lambda X.(A)(B)X]$$

as a "schematic definitions" or definition with parameters. If we call pair the introduction constant, the elimination rule will be: if $P(x) : U$ for $x : A \times B$, then we have split(u) : (x :

$A \times B)P(x)$ if $u : (x : A)(y : B)P(\mathsf{pair}(x, y))$, and the computation rule is that

$$\mathsf{split}(u)(\mathsf{pair}(x, y)) = u(x, y)$$

But it would be possible to introduce the following definition of product, as an inductively defined relation :

$$A \times' B = \mu[\lambda X.(A' : U)(B' : U)(A')(B')X(A', B')](A, B)$$

The introduction rule is then $\mathsf{pair}' : (A, B : U)(A)(B)A \times' B$ and the elimination rule is that if $P(A, B, x) : U$ for $A, B : U$ and $x : A \times' B$, then we have $\mathsf{split}'(u) : (A, B : U)(x : A \times' B)P(A, B, x)$ if

$$u : (A, B : U)(a : A)(b : B)P(\mathsf{pair}'(A, B, a, b)).$$

The computation rule is

$$\mathsf{split}'(u)(A, B, \mathsf{pair}'(A, B, a, b)) = u(A, B, a, b).$$

Actually the rule of parameters A and B is not "active" and the two definitions are provably equivalent (see also [21]).

It is clear how to define $\times'$ from $\times$: simply take $\times' = \lambda A, B.A \times B$ and then check that the elimination rule for $\times'$ is derivable.

The other direction seems more problematic. It is natural to try to take $A \times B$ defined as $A \times' B$, but it is not clear that the elimination rule for $\times$ is derivable from the elimination rule for $\times'$, that is, how to define split in terms of split'. The fact that is indeed possible to do so is thus surprising and illustrate the power of the use of quantified inductive hypothesis. The trick is to take for the relation in the elimination rule $P(A, B, x) = (Q : (A \times' B)U)((a : A)(b : B)Q(\mathsf{pair}(A, B, ()a, b)))Q(x)$. Proving $(A, B : U)(x : A \times B)P(A, B, x)$ will show that we can represent split, and, by induction, this reduces to prove, for $A, B : U$ and $a : A, b : B$:

$$(Q : (A \times' B)U)((a' : A)(b' : B)Q(\mathsf{pair}'(A, B, a', b')))Q(\mathsf{pair}'(A, B, a, b))$$

which is directly provable.

This traduction is all right when U is Prop (and is impredicative). When U is one of the predicative universe, we have to use the version of split' at the level $i + 1$ to define the version of split at the level i.

1.5 Problems with eta-conversion

In the pure calculus ECC without inductive types, there was no need of eta-conversion, the arguments being that the user has always to write terms in full eta-expanded form. Experimentally, this requirement was not too strong, i.e. it was always possible, in the examples, to put a priori the terms in a enough η-expanded form so that η-conversion was not needed.

However, this appears more problematic here. For instance, as soon as we define a type with a functional constructor, like the W type, which is defined for $A : U, B : (A)U$, with one constructor $s : (x : A)(f : (B(x))W)W$, has for elimination rule

$$\frac{u : (x : A)(f : (B(x))W)((y : B(x))P(f(y)))P(s(x, f))}{\mathsf{rec}(u) : (z : W)P(z)}$$

but we have to put $P(s(x, (\lambda u : A)f(u)))$ instead of $P(s(x, f))$ if we don't have η-conversion. Thus, the forms of our rules depend on having η-conversion.

Since η-conversion is satisfied in the p.e.r. model for instance, this rule may be semantically motivated. For the implementation this raises the problem of finding a complete algorithm for testing η-conversion, which is yet an open problem. We conjecture that the following algorithm is complete, even when we add the computation rule of the inductively defined types: given two terms for which we know that they have the same type, we compute the weak head-normal form (that is, we do normal reduction until we get a product, an abstraction or an application of a variable to some terms), and in the case where one of the weak head-normal form is $(\lambda x : A)M$ and the other a term N which is not a abstraction, we compare recursively M and $N(x)$.

2 An alternative formulation

We know present another approach, which was motivated by the formulation in [8]. This approach may be used as an intermediary step for giving a semantics of our previous system: we will show how to translate this previous system in this alternative system, and a semantics of the alternative semantics appears to be easier (we will sktech a set-theoretic semantics). Also, this system is "logically motivated", since its basic primitives correspond to universal and existential quantification, conjonction and disjonction.

2.1 The basic system

We work in the system ECC described in [17]. This is essentially the calculus of constructions extended with a predicative hierarchy of universes, and a strong Σ operator. We extend it

with η-conversion and surjective pairing. We add the new type forming operation $A + B$, which is in U_i if $A, B : U_i$ and in Prop if A, B : Prop. The elimination rule is

$$\frac{P : (A+B)U \quad z : A+B \quad f : (x : A)P(i(x)) \quad g : (y : B)P(j(y))}{\mathsf{case}(z, f, g) : P(z)}$$

where U may be any universe or Prop, and the conversion rules are

- $\mathsf{case}(i(x), f, g) = f(x)$,
- $\mathsf{case}(j(y), f, g) = g(y)$,
- $z = \mathsf{case}(z, \lambda x.i(x), \lambda y.j(y))\ [z : A + B]$.

We will write π_1 and π_2 for the two projections of the strong sums Σ. All these laws are valid in the realisability model considered in [17], and we conjecture that all the metamathematical properties of ECC (strong normalisation, Church-Rosser,...) are still valid for this extension.

2.2 Strictly positive operator

We will consider an operator $\Phi(X) : U\ [X : U]$ built syntactically from$\rightarrow$, $\times$, $+$ and Unit. We suppose that this operator is strictly positive that is X never occurs at the left of an arrow.

The operator Φ is described by induction. It is either a constant, or $\Phi(X) = X$, or $\Phi(X) = (K)\Phi_1(X)$, or $\Phi(X) = \Phi_1(X) + \Phi_2(X)$ or $\Phi(X) = \Phi_1(X) \times \Phi_2(X)$, where $\Phi_1(X)$. $\Phi_2(X)$ are described analogously.

We can now extend Φ functorially on terms as above. Since we have η-conversion, this is furthermore "functorial", i.e. we have $\Phi(\lambda x.x) = \lambda x.x$ and $\Phi(\lambda x.g(f(x)) = \lambda x.\Phi(g)(\Phi(f)(x))$, where the equality is definitional.

We can now present the extension of simple type theory with recursive types. Let Φ be a positive operator $\Phi : (U)U$. We extend our type theory with a type $A = \mu(\Phi)$ ("initial Φ-algebra"), an introduction rule intro : $(\Phi(A))A$, and an elimination rule

$$\frac{u : (z : \Phi(\Sigma(A, P)))P(\mathsf{intro}(\Phi(\pi_1)(z)))}{\mathsf{rec}(u) : (x : A)P(x)} \qquad [P : (A)U]$$

Furthermore, we have the following conversion rule

$$\mathsf{rec}(u)(\mathsf{intro}(y)) = u(\Phi(\lambda x.(x, \mathsf{rec}(u)(x)))(y)) : P(\mathsf{intro}(y)),$$

for $y : \Phi(A)$. These rules can be formulated as well at the level of Prop, and then express how to do proofs by induction over A.

As an example, we can define the morphism match : $(A)\Phi(A)$, by induction match = $\mathsf{rec}(\Phi(\pi_1))$. We then need one important definition.

Definition. We write $I(A,x,y)$ for $(P : (A)\mathsf{Prop})(P(y))P(x)$ (Leibniz' equality); we say that $f : (A)B$ is an *intensional* injection if, and only if, $(x,y : A)(I(B,f(x),f(y))I(A,x,y)$ is provable, and that a pair (f,g), with $f : (A)B$ and $g : (B)A$, is a intensional isomorphism if, and only if, $(x : A)I(A,x,g(f(x)))$ and $(y : B)I(B,y,f(g(y)))$ are provable.

Proposition. ("isomorphism theorem") The pair $(\mathsf{intro},\mathsf{match})$ is an intensional isomorphism between $\Phi(A)$ and A.

Proof: Indeed, we have first that

$$\mathsf{match}(\mathsf{intro}(z)) = \Phi(\pi_1)(\Phi(\lambda x.(x,\mathsf{rec}(\Phi(\pi_1))(x)))(z)) = \Phi(\lambda x.x)(z) = z$$

where the equality is the definitional equality. Next, we prove $(x : A)I(A,x,\mathsf{intro}(\mathsf{match}(x)))$ by induction on x. Putting $P(x) = I(A,x,\mathsf{intro}(\mathsf{match}(x)))$, we are reduced to the subgoal

$$(z : \Phi(\Sigma(A,P)))P(\mathsf{intro}(\Phi(\pi_1)(z)))$$

but the term $P(\mathsf{intro}(\Phi(\pi_1)(z)))$ is convertible (i.e. definitionaly equal) to

$$I(A,\mathsf{intro}(\Phi(\pi_1)(z)),\mathsf{intro}(\mathsf{match}(\mathsf{intro}(\Phi(\pi_1)(z))))),$$

which is convertible by what is above to $I(A,\mathsf{intro}(\Phi(\pi_1)(z)),\mathsf{intro}(\Phi(\pi_1)(z)))$, hence the result.

We regard the fact that we have the *definitional* equality $\mathsf{match}(\mathsf{intro}(z)) = z$ as one of the basic difference between this direct representation of inductively defined types and the one that we get via an impredicative coding in the system F (see [18]).

2.3 How to represent inductive types

We will analyse only one example. If we want to represent the type of integers, we define first the positive operator $\Phi(X) = 1 + X$ and then we take $N = \mu(\Phi)$. We define $O = \mathsf{intro}(i(*))$ (where $*$ is the only element of type 1) and $S = \lambda x : A.\mathsf{intro}(j(x))$. If $P(x) : U\ [x : N]$ and $a : P(O)$, $g : (x : N)(P(x))P(S(x))$ we build an object in $(x : N)P(x)$ as follows: it will be $\mathsf{rec}(u)$ with $u : (z : \Phi(\Sigma(N,P))P(\mathsf{intro}(\Phi(\pi_1)(z)))$ which is $u = \lambda z.\mathsf{case}(z,\lambda x.a,\lambda y.\mathsf{split}(y,\lambda p.\lambda q.g(p,q)))$. We can then check that we have $u(O) = a$ and $u(S(x)) = g(x,u(x))$, where the equality is definitional. This holds in general.

In this sense, this coding of inductive data types with positive operator is a "good" one: we get the expected definitional equality. This is not the case for the impredicative coding of inductive data types (cf. for instance [18, 21]).

2.4 Set-theoretic semantics

We now present a set-theoretical semantics of recursive types. The construction of this interpretation follows from one key remark. This remark is that Φ is interpreted by a

functor $\Phi : \mathsf{Set} \to \mathsf{Set}$ which is κ-continuous for a certain cardinal κ (by induction on Φ, and we have $\kappa < \aleph_\omega$), hence that has an initial Φ-algebra. This cardinal κ may be bigger than $\aleph_0$, for instance when we want to represent the type of ordinals. We take this initial Φ-algebra to be the interpretation A of $\mu(\Phi)$, and we thus get an interpretation of the map $\mathsf{intro} : \Phi(A) \to A$, such that, for any set X and any map $f : \Phi(X) \to X$, there exists one and only one map $\hat{f} : A \to X$ such that

$$f \circ \Phi(\hat{f}) = \hat{f} \circ \mathsf{intro}.$$

We have to interpret rec, hence we suppose given a map

$$u : (z : \Phi(\Sigma(A, P)))P(\mathsf{intro}(\Phi(\pi_1))(z)).$$

We have then that $f = \lambda z.(\mathsf{intro}(\Phi(\pi_1)(z)), u(z))$ is of type $(\Phi(\Sigma(A, P)))\Sigma(A, P)$, and such that $\pi_1 \circ f = \mathsf{intro} \circ \Phi(\pi_1)$. By using initiality ("existence" part), we get a map $s : (A)\Sigma(A, P)$ such that $s \circ \mathsf{intro} = f \circ \Phi(s)$. We take then $\mathsf{rec}(u) = \pi_2 \circ s$. We deduce that $\mathsf{intro} \circ \Phi(\pi_1) \circ \Phi(s) = \pi_1 \circ f \circ \Phi(s) = \pi_1 \circ s \circ \mathsf{intro}$. Still by initiality ("unicity" part), we deduce that we have $\pi_1 \circ s = id_A$, so that s is a section $s = \lambda x.(x, \mathsf{rec}(u)(x))$ of $\pi_1 : \Sigma(A, P) \to A$. The equation $s \circ \mathsf{intro} = f \circ \Phi(s)$ gives then $\mathsf{rec}(u)(\mathsf{intro}(y)) = u(\Phi(\lambda x.(x, \mathsf{rec}(u)(x)))(y))$, and thus we have an interpretation of the elimination operator with the conversion rules.

An important point is that the interpretation of $\mu(\Phi)$ is naturally stratified: $[\![\mu(\Phi)]\!]$ is the directed colimits of $\Phi^\alpha(\emptyset)$, with $\alpha < \kappa$. We can thus define the rank of an element of $[\![\mu(\Phi)]\!]$ as being the first ordinal α such that this element appears in $\Phi^\alpha(\emptyset)$. This allows us to extend the proof of reducibility as presented in Shoenfield [24].

Once we have a normalisation proof at the level of terms, and a set-theoretic interpretation, there is no problems in extending the proof of normalisation of F_ω, which expresses nicely a cut-elimination theorem for higher-order minimal logic, to a proof of cut-elimination for higher-order minimal logic extended with recursive types. In any case, this shows how to extend the realisability semantics of [17] to our calculus.

3 Some paradoxes

3.1 Positive inductive definition

We show that the condition of *strict* positivity for Φ is essential. We will limit ourselves to the (positive) example $\Phi(X) = ((X)\mathsf{Prop})\mathsf{Prop}$, and show that, on this example, it is not possible to extend in a satisfactory way the previous treatment of recursive types.

For this we will use only the "computational" part of our previous discussion. We notice first that Φ acts in a functorial way: if $f : (X)Y$, then we define $\Phi(f)$ by $\Phi(f)(a, y) = a(\lambda u.y(f(u)))$, for $a : ((X)\mathsf{Prop})\mathsf{Prop}$ and $y : (Y)\mathsf{Prop}$.

The natural way to express primitive recursion over the inductive type $A = \mu(\Phi)$ defined by Φ is:

$$\frac{u : (z : \Phi(A \times B))P(\mathsf{intro}(\Phi(\pi_1)(z))}{\mathsf{rec}(u) : (A)B}$$

where B is an arbitrary type.

Furthermore, we have the following conversion rule

$$\mathsf{rec}(u)(\mathsf{intro}(z)) = u(\Phi(\lambda x.(x, \mathsf{rec}(u)(x)))(z)).$$

It is then possible to define $\mathsf{match} : (A)((A)\mathsf{Prop})\mathsf{Prop}$, in such a way that $\mathsf{match}(\mathsf{intro}(z)) = z$ is a definitional equality for $z : ((A)\mathsf{Prop})\mathsf{Prop}$. We simply take $\mathsf{match} = \mathsf{rec}(\lambda z.\Phi(\pi_1)(z))$. In particular this implies that intro is an intensional injection from $((A)\mathsf{Prop})\mathsf{Prop}$ to A.

But this is now enough to get a contradiction.

Proposition. For any type A, there cannot be an intensional injection from $((A)\mathsf{Prop})\mathsf{Prop}$ to A.

Proof: For any type X there is a intensional injection from X to $(X)\mathsf{Prop}$ which is $\lambda x : X.\lambda y : X.I(X, x, y)$. Hence, by composition, we get an intensional injection f from $(A)\mathsf{Prop}$ to A. We are now back to the usual Cantor-Russell's paradox. We can define $P_0 : (A)\mathsf{Prop}$ by $P_0(x) = \exists P\,[I(A, f(P), x)\&\neg P(x)]$ (using the standard coding of the existential quantification, the negation and the conjonction in higher-order logic) , and also $x_0 = f(P_0)$. We have $P_0(x_0)$ if, and only if, $\neg P_0(x_0)$, hence a contradiction. □

Notice that we did not even need to express an induction principle over the type A. It was enough to have a primitive recursion operator.

Remark 1. The operator $\Phi(X) = ((X)\mathsf{Prop})\mathsf{Prop}$ we have used to show the inconsistency of the extension is the same as the one used in Reynolds' proof that there is no set-theoretic semantics of polymorphism [22]. It is actually possible to use the ideas presented here in order to derive a purely type-theoretic version of Reynolds' theorem (see [6]).

Remark 2. This has to be contrasted with an operator of the form $\Phi(X) = ((X)B)B$, with $B : \mathsf{Prop}$. Here, it is possible to interpret $\mu(\Phi) : \mathsf{Prop}$, following [14].

3.2 Predicativity

It should be noticed that we have used strongly in the last step of this argument the *impredicativity* of the calculus when we are using the property $\lambda x.\exists P\,[I(A, f(P), x)\&\neg P(x)]$ which has a quantification over properties. A natural question is what happens in a predicative calculus, that is a calculus where the quantification is restricted over types that are purely

functional, i.e. types that don't contain the types of propositions. This question actually goes back to the preface of the second edition of Principia [23], where it is asked if one can derive Cantor's result that 2^A is of cardinality greater than A in a predicative framework. It seems possible to use the model presented in [1] and the technique of [14] in order to show the consistency of the predicative calculus with any positive (and not only strictly positive) operator. This will show indirectly that Cantor's result is unprovable in the formalisation of Principia without the reducibility axiom.

4 An example

Here we show how we can represent some facts about ordinals, and we will follow closely (a tiny part of) the elementary proof of S. Wainer [25] of J.Y. Girard's theorem of comparison of hierarchy. One remark is that, even on this simplified example, the concept of induction over a predicate may clarify a proof (see below). This concept is not emphasized usually, and seems to come from Type Theory, see [21] for a precise formulation.

It is possible to declare the type of "ordinal notations", since it is definable with a (strictly positive) operator. Here are the declaration rules for ordinals of class I (or integers), and ordinals of class II and of class III. Historically, the ordinals of class II are the ones that are countable, but here we work in a constructive framework, and in a sense, everything is "countable", and we work with ordinal *notations* instead of working with set-theoretic ordinals.

```
Inductive N0:U = Z0:N0 | S0:(N0)N0.
Inductive N1:U = Z1:N1 | S1:(N1)N1 | L01:((N0)N1)N1.
Inductive N2:U = Z2:N2 | S2:(N2)N2 | L02:((N0)N2)N2 | L12:((N1)N2)N2.
```

It is then possible to define inductively the slow hierarchy and the Hardy hierarchy (see [25]). We give the primitive recursive equations satisfied by these hierarchies.

```
G1:(N1)(N0)N0.
G1(Z1,x) = Z0, G1(S1(z),x) = S0(G1(z,x)), G1(L01(u),x) = G1(u(x),x).

i0:(N0)N1.
i0(Z0) = Z1, i0(S0(x)) = S1(i0(x)).

G2:(N2)(N0)N1.
G2(Z2,x) = Z1, G2(S2(z),x) = S1(G2(z,x)), G2(L02(u),x) = G2(u(x),x),
G2(L12(u),x) = L01([n:N0]G2(u(i0(n)),x)).
```

```
H1:(N1)(N0)N0.
H1(Z1,x) = x, H1(S1(z),x) = H1(z,S0(x)), H1(L01(u),x) = H1(u(x),x).

H2:(N2)(N1)N1.
H2(Z2,x) = x, H2(S2(z),x) = H2(z,S1(x)),
H2(L01(u),x) = L01([n:N0]H2(u(n),x)), H2(L12(u),x) = H2(u(x),x).
```

The two hierarchies we are interested in are G_1 (slow hierarchy) and H_1 (Hardy hierarchy), written also λ and also called the fast hierarchy. To give some ideas, $G_1(\omega)$ is the identity function, $G_1(\omega^\omega)$ the function $n \mapsto n^n$, $H_1(\omega^\omega)$ is of the order of Ackerman function. The goal is to relate these two hierarchies (our treatment differs from the one of Wainer [25], in that Wainer considers Grzegorczyk hierarchy instead of the Hardy hierarchy). That is, given an ordinal notation α, we want to know if there exists another ordinal notation $f(\alpha)$ such that $H_1(\alpha)$ is extentionally equal to $G_1(f(\alpha))$.

Over the type of integers, we can define "intensional equality"

```
Inductive E0:(N0)(N0)Prop = ref0:(x:N0)E0(x,x).
```

Notice that this corresponds precisely to the new intensional equality type introduced by P. Martin-Löf. Another possible equality on the type N_0 is the following provably equivalent equality (that may be interesting with "partial" or non-standard element):

```
Inductive E0:(N0)(N0)Prop =
 refZ:E0(Z0,Z0)
|refS:(x,y:N0)(E0(x,y))E0(S0(x),S0(y)).
```

Over the type of "constructive ordinals of class I", we can define a "weakly extensional" equality (the strong one being the equality of the set-theoretic semantics of the two ordinals)

```
Inductive Ext1:(N1)(N1)Prop =
         ext0:Ext1(Z1,Z1)
        |ext1:(x,y:N1)(Ext1(x,y))Ext1(S1(x),S1(y))
        |ext2:(u,v:(N0)N1)((n:N0)Ext1(u(n),v(n)))Ext1(L01(u),L01(v)).
```

This equality reflects the struture of the type N_1 in the same way that the second equality E_0 reflects the structure of the type N_0.

The reader can check that it is possible to prove by induction on the definition of Ext_1:

```
(x,y:N1)(Ext1(x,y))(n:N0)E0(H1(x,n),H1(y,n))
```

This shows that H_1 is a function in Bishop's sense [3] between the sets (N_1, Ext_1) and (N_0, E_0).

We can now express the (simplified version of the) "collapsing lemma", which is the main lemma in Wainer's proof.

```
(n:N0)(x:N2)(y:N1)E0(G1(H2(x,y),n),H1(G2(x,n),G1(y,n)))
```

See [25], on page 611. This results gives a relation between the slow hierarchy G_1 to the Hardy hierarchy H_1: indeed, we will have $G_1(H_2(x,\omega),n) = H_1(G_2(x,n), G_1(\omega,n))$, and it is easy to check that $G_1(\omega,n) = n$. Hence, we will get $G_1(H_2(x,\omega),n) = H_1(G_2(x,n),n)$. For some ordinal notation $x : N_2$, $G_2(x,n)$ is independent of n, and we get a relation between G_1 and H_1. For instance, if $x = 3$, we get $G_1(\omega+3,n) = H_1(3,n)$. For $x = \omega_1$, we get that $G_1(H_2(\omega_1,\omega),n) = H_1(\omega,n)$, that is $G_1(\omega+\omega,n) = H_1(\omega,n)$.

A natural attempt is to try to prove this result by induction over x, with the parameter n. The following subgoal is then generated.

```
E0(H1(G2(f(y),n),G1(y,n)),H1(G2(f(i0(G1(y,n))),n),G1(y,n)))
====================
y : N1
h1 : (X1:N1)(X2:N1)E0(G1(H2(f(X1),X2),n),H1(G2(f(X1),n),G1(X2,n)))
f : (N1)N2
n : N0
```

By using the previous lemma about Ext_1, this subgoal becomes

```
Ext1(G2(f(y),n),G2(f(i0(G1(y,n))),n))
====================
y : N1
h1 : (X1:N1)(X2:N1)E0(G1(H2(f(X1),X2),n),H1(G2(f(X1),n),G1(X2,n)))
f : (N1)N2
n : N0
```

And then, it seems difficult to go further. But this subgoal motivates the following inductive predicate (with one parameter n of type N_0).

```
Inductive O2:(N2)Prop =
        O2_0:O2(Z0)
       |O2_1:(x:N2)(O2(x))O2(S2(x))
       |O2_2:(u:(N0)N2)((p:N0)O2(u(p)))O2(LO2(u))
       |O2_3:(f:(N1)N2)((x:N1)O2(f(x)))
             ((x:N1)Ext1(G2(f(x),n),G2(f(i0(G1(x,n))),n)))O2(L12(f)).
```

This corresponds to the predicate O_n defined on page 610 of [25]. Notice however one important difference: this is an inductively defined predicate, and *not* a propositional function defined over the type $N2$. One can easily build a proof of $O_2(n, 3)$, or $O_2(n, \omega_1)$.

We can now state, and prove, the correct version of the collapsing lemma

```
(n:N0)(x:N2)(O2(n,x))(y:N1)E0(G1(H2(x,y),n),H1(G2(x,n),G1(y,n)))
```

We will not give the formal proof here, which is a direct induction on the definition of the predicate O_2.

The fact we see O_2 as an inductively defined predicate will allow us to prove this by induction on the definition on O_2, while in [25], the predicate O_2 is defined by induction on $N2$, and the proof is an induction on $x : N2$. The proof, with the use of the inductively defined predicate O_2, appears to easier to formalise completely than the proof given in [25]. In conclusion, this example shows that the concept of induction over predicates, relations,... is important if one is concerned with the idea of proofs as first-class objects (and with the formal elegance of proofs).

Conclusion

One basic point of this paper is that the notion of subtypes is not needed for the introduction of inductive types in a type system (in opposition to [14, 8]). As pointed out to us by P. Aczel, it will be interesting to develop a similar formulation for the dual of indutive types (cf. [14]). One another point is that it seems more elegant to have the notion of inductive types in the core of the formal system, rather to build it as a derived notion (for instance, in Martin-Löf type theory, with the W type, see [8]). However this approach seems (at least in a first attempt) to have a theoretical drawback: it is easier to make a metamathematical study of a fixed system than of one that contains a schema of rules. In this paper, we propose an intermediate system, that we can study metamathematically (and for instance give a set-theoretic semantics), and in which we can translate the implemented system, and we remark in this framework that the condition of strict positivity is essential. We can remark however that the semantics we sketch (in term of T-algebra) is quite close to the intermediate syntactic system and presents actually the same drawbacks: it introduces unatural coding w.r.t. the primitive notion of inductively defined object. It will thus be nice to have a more direct understanding of this notion. We hope that to be able to play mechanically with it will help towards such an understanding.

Acknowledgement

This research was partly supported by the Esprit Basic Research Action Logical Frameworks.

References

[1] P. Aczel. "The strength of Martin-Löf's intuitionistic type theory with one universe." In the Proceedings of the Symposium on Mathematical Logic, Helsinki, 1977.

[2] R. Backhouse. "On the meaning and construction of the rules in Martin-Löf's theory of types." Technical Report CS 8606, University of Groningen, 1986.

[3] E. Bishop. "Foundations of Constructive Analysis." McGraw-Hill, New-York, 1967.

[4] C. Böhm and A. Berarducci. "Automatic synthesis of typed Λ-programs on term algebras." Theoretical Computer Science, 39, 1985.

[5] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic 5, 1940.

[6] Th. Coquand. "An introduction to type theory." Course notes, Albi, 1989.

[7] Th. Coquand. "An Analysis of Girard's Paradox." Proceedings of the first Logic in Computer Science, Boston, 1986.

[8] P. Dybjer. "Inductively Defined Types in Martin-Löf's Set Theory." Unpublished manuscript, 1987.

[9] P. Dybjer. "An inversion principle for Martin-Löf's type theory." To appear in the proceedings of Bastad, 1989.

[10] K. Gödel. "On a hitherto unexploited extension of the finitist viewpoint." Translation by W. Hodge, appeared in Journal of Philosophical Logic 9 (1980).

[11] D. Hilbert. "On the Infinite." Published in Van Heijenoort.

[12] P. Martin-Löf. "Notes on Constructive Mathematics." Almqvist & Wiksell, Stockholm.

[13] P. Martin-Löf. "Intuitionistic Type Theory." Bibliopolis, 1980.

[14] P. F. Mendler. "Inductive Definition in Type Theory." Ph. D. Thesis, Cornell, 1987.

[15] P.J. Landin. "The mechanical evaluation of expressions." Comput. J. 6, 1964.

[16] Z. Luo. "CC and its meta Theory." LFCS report ECS-LFCS-88-57, Dept. of Computer Science, University of Edinburgh.

[17] Z. Luo. "ECC, an Extended calculus of Constructions." Proc. of the Fourth IEEE Symposium on Logics in Computer Science, June 1989, Asilomar, California, U.S.A.

[18] Ch. Paulin-Mohring. "Extraction de programmes dans le Calcul des Constructions." Thèse, Université Paris 7, 1989.

[19] D. Normann. "Inductively and recursively defined types." A seminar report, Department of Mathematics, University of Oslo, 1987.

[20] L. C. Paulson. "A formulation of the Simple Theory of Types (for Isabelle)." Unpublished manuscript, Cambridge, 1989.

[21] F. Pfenning and Ch. Paulin-Mohring. "Inductively Defined Types in the Calculus of Constructions." To appear in the proceedings of MFPLS'89, 1989.

[22] J.C. Reynolds. "Polymorphism is not Set-Theoretic." Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[23] B. Russell and A.N. Whitehead. "Principia Mathematica." Volume 1,2,3 Cambridge University Press, 1912.

[24] J.R. Shoenfield. "Mathematical Logic." Addison-Wesley, 1967.

[25] S.S. Wainer. "Slow Growing Versus Fast Growing." Journal of Symbolic Logic, Volume 54, 1989.