INFORMATION AND COMPUTATION 99, 231–264 (1992)

The Extended Calculus of Constructions (ECC) with Inductive Types

CHRISTIAN-EMIL ORE

Department of Computer Science, University of Oslo, Norway

Luo's Extended Calculus of Constructions (ECC) is a higher order functional calculus based on Coquand's and Huet's Calculus of Constructions, but has in addition strong sums and a predicative cumulative type hierarchy. In this paper I introduce inductive types on the predicative type levels of ECC. I also show how the ω-**Set** model for ECC can be extended to a model for this augmented calculus.

Contents

1. THE EXTENDED CALCULUS OF CONSTRUCTIONS, ECC

The extended calculus of constructions (ECC) is based in Coquand and Huet's theory of constructions, a higher order types functional calculus (Coquand, 1986a; Coquand and Huet, 1988). The ECC adds to the theory of constructions an infinite, fully cumulative type hierarchy and also so-called strong sums ($\sum$-types).

The lowest level, Prop, of the type hierarchy is impredicative and the second order λ-calculus (Girand, 1986) can be embedded into this level. All propositions on this level can be lifted as types of the higher, predicative type levels.

The strong sums ($\sum$-types) are defined only on the predicative type level. The $\sum$-types can also be introduced on the impredicative level. However, if this level is kept closed under such types, it is possible to derive Girard's paradox (Coquand, 1986(b)) and the calculus becomes inconsistent as a

0890-5401/92 $5.00

logic. The $\sum$-types can be seen as generalized products and are as described in (Luo, 1989a), a nice tool for modularization of proofs.

ECC can also be seen as a higher-order functional programming language. If the impredicative level is removed ECC corresponds to a higher order version of the language ML though without any kind of recursion. The $\sum$-type constitutes a strong basis for a modularization mechanism and object oriented features. With $\sum$-types it seems possible to construct a stronger and more flexible class concept than in the higher-order object oriented language Quest (Cardelli, 1989). The introduction of inductive types, that is, data types as known from ML, is a step toward the construction of such a higher order programming language.

The syntax of the original ECC is given in Table 1, and the type rules are given in Table 2.

ECC has many good properties. It is strongly normalizable, and the type checking is decidable. Although the cumulativity of the type hierarchy implies that the type of a term is not unique, it is a fact that each well typed term has a least type, the principal type of the term.

DEFINITION 1.1. A type A is the principal type of a term M in a context Γ iff

1. $\Gamma \vdash M:A$
2. for any term A', $\Gamma \vdash M:A'$ iff $A \leqslant A'$ and $\vdash A':K$ for some kind K.

The principal type is denoted $T_\Gamma(M)$.

TABLE I

The Term Calculus of ECC

Syntax:

1. The constants *Prop* and $Type_i$, $i \in \omega$, (called kinds), are terms.
2. Variables $x, y, z, \ldots$ are terms.
3. If A, B, M, and N are terms, so are $\prod x:A.B$, $\lambda x:A.M$, MN, $\sum x:A.B$, $\mathbf{pair}_{\sum x:A.B}(M, N)$, $\pi_1(M)$ and $\pi_2(M)$.

Reductions:

1. $(\lambda x:A.M)N \rhd [N/x]M$
2. $\pi_i(\mathbf{pair}_{\sum x:A.B}(M_1, M_2)) \rhd M_i$ $(i = 1, 2)$

Conversion equivalence:

The term calculus has the Church-Rosser property and $M_1 \simeq M_2$ iff $\exists M.M_1 \rhd M \wedge M_2 \rhd M$

TABLE 2

The Type System of ECC

Type cumulativity:

The relation $\leqslant$ is the smallest partial order over terms w.r.t. conversion $\simeq$ such that

1. $Prop \leqslant Type_0 \leqslant Type_1 \leqslant \ldots$
2. if $A \simeq A'$ and $B \leqslant B'$, then $\prod \pi : A.B \leqslant \prod x : A'.B'$.
3. if $A \leqslant A'$ and $B \leqslant B'$, then $\sum x : A.B \leqslant \sum x : A'.B'$.

Subsidiary relation:

$A \prec B$ iff $A \leqslant B$ and $A \not\simeq B$.

Typing:

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

$$(C) \qquad \frac{\Gamma \vdash A : K}{\Gamma, x : A \vdash Prop : Type_0} \qquad (x \notin FV(\Gamma))$$

$$(T) \qquad \frac{\Gamma \vdash Prop : Type_0}{\Gamma \vdash Type_i : Type_{i+1}}$$

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

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

$$(\prod 2) \qquad \frac{\Gamma \vdash A : K \quad \Gamma, x : A \vdash B : Type_i}{\Gamma \vdash \prod x : A.B : Type_k} \qquad (k = max(\mathscr{L}(K), i))$$

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

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

$$(\Sigma) \qquad \frac{\Gamma \vdash A : K \quad \Gamma, x : A \vdash B : K'}{\Gamma \vdash \sum x : A.B : Type_k} \qquad (k = max(0, \mathscr{L}(K), \mathscr{L}(K')))$$

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

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

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

$$(conv) \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : K}{\Gamma \vdash M : A'} \qquad (A \simeq A')$$

$$(cum) \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash A' : K}{\Gamma \vdash M : A'} \qquad (A \prec A')$$

THEOREM 1.1 (Luo, 1989b). *Every well typed term M in a context Γ has a principal type. It is the minimum type of M with respect to the cumulativity relation $\leqslant$.*

In Table 2 the letter $\mathscr{L}$ stands for a syntactic function returning the type level of a kind K. That is, if the principal type of K is $Type_i$, then $\mathscr{L}(K) = i$.

ECC extended with disjoint sums

To be able to express interesting inductive (data) types it is necessary to be able to construct disjoint unions of types. In ECC it is possible to encode such an operator on the impredicative level, that is, *Prop*. On a predicative level, say $Type_i$, it is not possible to express a disjoint sum operator such that $Type_i$ is closed under this operator. The theme for this paper is to extend ECC with inductive types on the predicative levels. Hence it is necessary to extend the calculus with an explicit disjoint sum operator. The extensions of the syntax and of the type rules are shown in Table 3. The introduction rule (*case*) for the case operator is analogous to the corresponding rule in Martin-Löf's type theory (Nordström, Petersson, and Smith, 1990). The disjoint sum operator is not introduced on the impredicative level. This could, however, have be done in the same way as for the predicative level, but is not necessary since the inductive types will be defined only on the predicative levels.

To construct many interesting inductive (data) types, e.g., the natural numbers, it is in addition necessary to extend the calculus with a one element type "**1**." There are three rules for the type **1**. The rules $F_{\mathbf{1}}$ and $I_{\perp}$ state the existence of the type **1** and the term $\perp$ of this type. The rule $\mathbf{1}_{\perp}$ is perhaps a little uncommon, but is justified by the ideal that $\perp$ is the only closed normal-form term of type **1**. The rule seems to be necessary in inductive proofs. It is not necessary in the construction of terms.

2. INDUCTIVE TYPES IN A FIRST ORDER CALCULUS

Domain theory can be used to give a model for first order typed lambda calculus with recursive types. Each type is interpreted as a domain. The recursive types are modeled as domains satisfying domain equations. E.g., the type of lists over a type A is modeled by a domain satisfying the equation $L = A + A \times L$. Such a domain is constructed as the least fixed point of a corresponding continuous domain operator.

The domain theory can be (usually is) expressed in a categorical setting. The domains (types) are objects in a closed Cartesian category C with sums and an initial object. The latter corresponds to $\perp$ in Scott models.

The continuous domain operators are colimit-preserving functors from C into C. The least fixed point of a domain operator (functor) F is the colimit of the diagram created by the iterative application of F to the initial object. It is of course necessary that such colimits exists in the category C.

The above description is a brief recapitulation of ordinary categorical domain theory. What is more interesting for our purpose, is the following well known fact: If F is a covariant functor from a category C into C, then the initial (least) fixed point of F (if it exists) is isomorphic to the initial

TABLE 3

The necessary Extensions for ECC to Disjoint Sum and the Single Element Type **1**

Syntax:

1. $\bot$ and **1** are terms
2. If A, B, L, M and N are terms, so are $A + B$, $\mathbf{inl}_{A+B}N$, $\mathbf{inr}_{A+B}N$, and $\mathbf{case}(L, M, N)$

The cumulative type relation:

If $A \leqslant A'$ and $B \leqslant B'$ then $A + B \leqslant A' + B'$.

Reductions:

1. $\mathbf{case}(\mathbf{inl}_{A+B}L, M, N) \rhd ML$
2. $\mathbf{case}(\mathbf{inr}_{A+B}L, M, N) \rhd NL$

Typing:

$$(F_1)\qquad \frac{\Gamma \vdash Prop : Type_0}{\Gamma \vdash \mathbf{1} : Type_0}$$

$$(I_\bot)\qquad \frac{\Gamma \vdash Prop : Type_0}{\Gamma \vdash \bot : \mathbf{1}}$$

$$(\mathbf{1}\lambda)\qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash x : \mathbf{1}}{\Gamma \vdash \lambda y : \mathbf{1}.[y/x]M : \prod y : \mathbf{1}.[y/x]A}$$

where y does not occur free in M or A and x is either a variable or $\bot$.

$$(sum)\qquad \frac{\Gamma \vdash A : Type_i \quad \Gamma \vdash B : Type_i}{\Gamma \vdash A + B : Type_i}$$

$$(inl)\qquad \frac{\Gamma \vdash L : A \quad \Gamma \vdash A, B : Type_i}{\Gamma \vdash \mathbf{inl}_{A+B}L : A + B}$$

$$(inr)\qquad \frac{\Gamma \vdash L : B \quad \Gamma \vdash A, B : Type_i}{\Gamma \vdash \mathbf{inr}_{A+B}L : A + B}$$

$$(case)\qquad \frac{\begin{array}{c}\Gamma, x : A + B \vdash C : Type_i \\ \Gamma, x : A + B \vdash M : \prod y : A.[\mathbf{inl}\ y/x]C\end{array} \quad \Gamma, x : A + B \vdash N : \prod z : B.[\mathbf{inr}\ z/x]C}{\Gamma, x : A + B \vdash \mathbf{case}(x, M, N) : C}$$

F-algebra. This simply means that the recursive types can be seen as algebraically defined types. As will be shown below this fact can be exploited to construct introduction rules and computation rules for certain inductive types in ECC.

Before I go into details it may be appropriate to refresh some categorical knowledge:

- Let T be an endofunctor on a category C, that is, a functor from C into C. A T-algebra is a C-object A together with a C-morphisms $\phi: T(A) \to A$.
- The category of T-algebras is defined as follows:
  - — Objects: T-algebras (A, ϕ)
  - — Morphisms: $f: (A, \phi) \to (A', \phi')$, where f is a C-morphism from A to A' such that $f \circ \phi = \phi' \circ T(f)$, that is,

$$\begin{array}{ccc} T(A) & \xrightarrow{\phi} & A \\ {\scriptstyle T(f)}\downarrow & & \downarrow{\scriptstyle f} \\ T(A') & \xrightarrow[\phi']{} & A' \end{array}$$

- The initial T-algebra (A_{init}, ϕ_{init}) is an initial object in the category of T-algebras. That is, for every T-algebra (A, ϕ) there exists a unique $f: A_{init} \to A$ such that the following diagram commutes:

$$\begin{array}{ccc} T(A_{init}) & \xrightarrow{\phi_{init}} & A_{init} \\ {\scriptstyle T(f)}\downarrow & & \downarrow{\scriptstyle f} \\ T(A) & \xrightarrow[\phi]{} & A \end{array}$$

Remark. Usually it is required that T be a monad. This weak definition is taken from T. Hagino (1987) and is sufficient for our purpose.

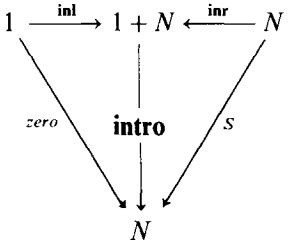
A typed first order λ-calculus with sum types corresponds to a Cartesian closed category (CCC) with coproducts. A λ-calculus with inductive types as well corresponds to a CCC with coproducts where the fixed points of endofunctors corresponding to the inductive types exist.

Consider a first order typed lambda calculus with product and sum types and inductive types. In the sequel we switch from this calculus to the corresponding CCC in an informal way.

The natural numbers can be represented as terms of the type $N \cong 1 + N$, where 1 is the unity type and $+$ represents sum of types. That is, N can be seen as the least fixed point of a functor $T_N(X) = 1 + X$ (due to

Lawvere). As mentioned earlier N will correspond to the initial T_N-algebra, $(N, \textbf{intro})$, where **intro**: $1 + N \to N$ is the isomorphism between $T_N(N)$ and N.

In a category theoretic setting zero and the successor function can be described by the following commutative diagram:



In the λ-calculus the natural number terms can be defined as

$$\underline{0} \equiv \textbf{intro}(\textbf{inl}\ \perp)$$

$$\underline{1} \equiv \textbf{intro}(\textbf{inr}\ \underline{0})$$

$$\vdots$$

The successor function is the following term:

$$\mathbf{S} \equiv \lambda x{:}N.\textbf{intro}(\textbf{inr}\ x)).$$

Now, consider a simple recursive function $f\colon N \to A$ defined as

$$f0 = a$$

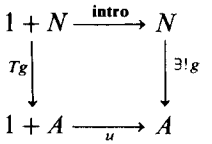
$$f(\mathbf{S}x) = h(fx),$$

where $h\colon A \to A$.

By using the fact that $(N, \textbf{intro})$ is the initial T_N algebra the function f can be expressed in the calculus. First the right hand sides of the definition of f are combined into a function $u\colon 1 + A \to A$ as follows:

$$u = \lambda x{:}\, 1 + A.\textbf{case}(x, \lambda y : 1.a, \lambda z : A.hz).$$

Since $(N, \textbf{intro})$ is the initial T_N algebra, there exists a unique $g\colon N \to A$ such that the following diagram commutes:



That is, $g(\mathbf{intro}\, x) = (u \circ T_N(g))x$. The expression $T_N(g)$ is not a term in the calculus, but for a given covariant functor T it is possible in a mechanical way to construct the term corresponding to the morphism $T(f)$ in the category; in the above case

$$T_N(g) \equiv \lambda x{:}1 + N.\mathbf{case}(x, \lambda y{:}1.\mathbf{inl}(y), \lambda x{:}N.\mathbf{inr}(g(x))).$$

By reducing the right hand side of the equation induced by the above diagram we get

$$g(\mathbf{intro}\, x) = \mathbf{case}(x, \lambda y{:}1\,.\, a, \lambda z.h(gz))$$

which is the definition of f expressed by the use of the **case**-operator.

The general picture is only slightly more complicated than the simplified situation above. Assume that $h\colon N \times A \to A$. Let $f\colon N \to A$ be defined as

$$f0 = a$$
$$f(\mathbf{S}x) = h(x, fx).$$

Here the function $u\colon T_N(N \times A) \to A$ will be

$$u = \lambda x{:}\, 1 + N \times A.\mathbf{case}(x, \lambda y{:}1.a, \lambda z{:}N \times A.h(\pi_1 z, \pi_2 z))$$

We get the following initial algebra diagram:

$$\begin{array}{ccc} T_N(N) & \xrightarrow{\quad\mathbf{intro}\quad} & N \\ {\scriptstyle T_N(\langle id, \mathbf{rec}(u)\rangle)}\downarrow \uparrow {\scriptstyle T_N(\pi_1)} & & \downarrow{\scriptstyle \exists!\langle id, \mathbf{rec}(u)\rangle} \\ T_N(N \times A) & \xrightarrow[\langle id \circ \mathbf{intro} \circ (T_N(\pi_1)), u\rangle]{} & N \times A \end{array}$$

In the above diagram the expression $\langle id, \mathbf{rec}(u)\rangle$ is the categorical notation for a function constructed as a pair of functions and corresponds to the term

$$\lambda x{:}N.\mathbf{pair}_{N \times A}(x, \mathbf{rec}(u)x)$$

in the calculus. This is the unique function which makes the above initial T_N-algebra diagram commute. It is evident that the first component must be the identity. The second component, the term $\mathbf{rec}(u)$ is used to denote the unique function, which makes the diagram commute for the given u.

The equality induced by the diagram is

$$\mathbf{rec}(u)(\mathbf{intro}\, x) = u(T_N(\langle id, \mathbf{rec}(u)\rangle)x).$$

Let in general $\mu X.T$ denote the inductive type defined by a functor T, covariant in X. For a function $u: T(\mu X.T \times A) \to A$ the right diagram is obtained by replacing N by $\mu X.T$ and T_N by T in the above diagram. The induced equality gives the following reduction (computation) rule by replacing equality by reducibility:

$$\mathbf{rec}(u)(\mathbf{intro}\ x) \rhd u(T(\langle id, \mathbf{rec}(u)\rangle)x).$$

The result of computing $T(\langle id, \mathbf{rec}(u)\rangle)$ is a morphism in the category, and in the above rule this expression stands for the term in the calculus describing this morphism (see Table 4 below).

3. ECC WITH INDUCTIVE TYPES

In (Coquand, 1989) T. Coquand suggested that the initial algebra scheme could be used for the calculus of constructions with $\sum$-types. The basic idea is similar to what was explained for the first order calculus in the previous section. In (Coquand and Poulin, 1989), T. Coquand and C. Paulin show that a type constructor on the predicative type level not only must be monotonic (covariant), but also must satisfy a *strictly positive occurrence* condition (see Definition 3.1) to be meaningful.

However, it is not necessary to restrict (as in Coquand, 1989, and Coquand and Poulin, 1989) the inductive types to be defined only in terms of the first order constructors "$\to$," "$\times$," and "$+$," and we can define the constructors in terms of $\sum$ and $\prod$ as well.

The definition of a strictly positive occurrence of a variable, given below in Definition 3.1, is more restrictive than in (Constable and Mendler, 1985; Mendler, 1987). In the definition below the variable y cannot occur in the subterm A of a type term of the form $\prod x{:}A.B$. For type terms of the form $\sum x{:}A.B$ y can occur in A only if B does not depend on $x{:}A$ (see also Section 4.2 for a comment). In (Coquand and Mendler, 1985) we do find the former restriction but not the latter. In (Mendler, 1987) there are no dependent products, and $\sum$-types are not treated. In that paper the first restriction is relaxed to the requirement that y most occur "to the left of two arrows," the so called positive occurrence requirement. This is necessary to keep the type operator monotonic. However, the inductive types in both (Coquand and Mendler, 1985; Mendler, 1987) correspond to types on the lowest level, *Prop*, in the calculus of constructions (and ECC). In (Coquand and Paulin, 1989) it is shown that on the *Type*-level the "positive-occurrence" requirement is too weak; it is necessary to require that the induction variable occur *strictly* positively in the type terms constructed by the "$\to$" constructor, that is, no occurrences to the left of any number of arrows.

TABLE 4

The Terms Corresponding to the Morphism Mapping Part of the Functor T_M

Assume that $\Gamma, y : Type_i \vdash M : Type_i$ and that y occurs strictly positively in M. The term Φ_M is of type

$$\Phi_M : \prod A, B : Type_i.(A \to B) \to ([A/y]M \to [B/y]M)$$

and is defined recursively according to the syntactic structure of the term M:

1. If y does not occur in M, then

$$\Phi_M \equiv \lambda A, B : Type_i.\lambda f : A \to B.$$
$$\lambda x : M.x$$

2. If $M \equiv y$, then

$$\Phi_M \equiv \lambda A, B : Type_i.\lambda f : A \to B.f$$

3. If $M \equiv L + N$, then

$$\Phi_M \equiv \lambda A, B : Type_i.\lambda f : A \to B.$$
$$\lambda x : L' + N'.\mathbf{case}(x, \lambda z : L'.\mathbf{inl}_{L'' + N''}((\Phi_L ABf)z),$$
$$\lambda z : N.\mathbf{inr}_{L'' + N''}((\Phi_N ABf)z)),$$

where

$$L' \equiv [A/y]L \qquad L'' \equiv [B/y]L$$
$$N' \equiv [A/y]N \qquad N'' \equiv [B/y]N$$

4. If $M \equiv \prod x : L.N$, then

$$\Phi_M \equiv \lambda A, B : Type_i.\lambda f : A \to B.$$
$$\lambda g : \prod x : L.[A/y]N.(\lambda z : L.((\Phi_{[z/x]N} ABf)z))$$

5. If $M \equiv \sum x : L.N$ then

$$\Phi_M \equiv \lambda A, B : Type_i.\lambda f : A \to B.$$
$$\lambda z : \sum x : [A/y]\,L.[A/y]N.\mathbf{pair}_E((\Phi_L ABf)(\pi_1 z), (\Phi_{[\pi_1 z/x]N} ABf)(\pi_2 z)),$$

where

$$E \equiv \sum x : [B/y]\,L.[B/y]N$$

6. If $M \equiv \mu x.N$ then

$$\Phi_M \equiv \lambda A, B : Type_k.\lambda f : A \to B.\mathbf{rec}(u),$$

where

$$u \equiv \mathbf{intro} \circ (\Phi_{[B/y]N}(E \times F)F\pi_{2_{E \times F}}) \circ (\Phi_{[(E \times F)/x]N} ABf)$$
$$E \equiv \mu x.[A/y]N$$
$$F \equiv \mu x.[B/y]N$$

The calculi of constructions correspond to categories, e.g., ω-**Set** for ECC (see Section 4.1 for details). A type constructor will correspond to an endofunctor on this category. The inductive types will be the initial fixed points of functors constructed according to the definition of the types. The fixed point does not exist for every functor, that is, every constructor, but every functor corresponding to a type constructor satisfying the strictly positive occurrence requirement will have fixed points.

There is also a minor difference between the usual definitions and the definition below. This definition ensures that if y occurs strictly positively in a term M, then there is at least on occurrence of y in M. This condition is introduced to avoid the possibility to construct "inductive" types which simply are copies of existing types, e.g., $\mu y.Type_i$ and $\mu x.\mu y.Type_i$. From a mathematical point of view such types are feasible, but they have no function. It seems reasonable to avoid such a redundancy in the system.

That y is strictly positive in M implies that M is built up from general subterms without occurrences of y, from variables, and from constants, solely by the use of $\prod$, $\sum$, μ, and $+$. This seems not to be a serious restriction since, at least on the meta-theoretical level, the remaining term constructors are transparent with respect to the μ-operator.

DEFINITION 3.1 (strictly positive occurrence). A variable y is said to occur strictly positively in a term M if and only if

1. $M \equiv y$.

2. $M \equiv \prod x{:}A.B$ and y does not occur in A and y occurs strictly positively in B.

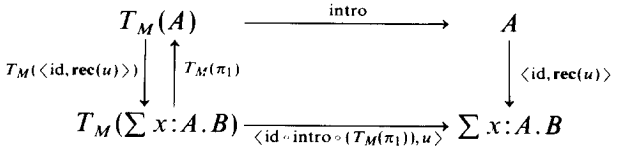
3. $M \equiv \sum x{:}A.B$ where B depends on x and y does not occur in A and y occurs strictly positively in B.

4. $M \equiv \sum x{:}A.B$ where B does not depend on x and y occurs strictly positively in both A and B, or y does not occur in A and y occurs strictly positively in B or y does not occur in B and y occurs strictly in A.

5. if $M \equiv A + B$ and y occurs strictly positively in both A and B, or y does not occur in A and y occurs strictly positively in B or y does not occur in B and y occurs strictly positively in A.

6. if $M \equiv \mu x.N$ and y occurs strictly positively in N.

Suppose we have $\Gamma, y{:}Type_i \vdash M{:}Type_i$, where M is constructed only by the use of μ (see below,) $\sum$, $\prod$, and $+$ from variables and from subterms with no occurrences of y, and y occurs strictly positively in M. Then we will as earlier let $\mu y.M$ denote the inductive type based on M. A recursive function $\mathbf{rec}(u) : \prod x{:}\mu y{:}M.B$ based on a function u will correspond to the unique morphism satisfying the initial algebra diagram



In the above diagram $A \equiv \mu y.M$ and T_M is the functor corresponding to M in the context Γ (see Section 4.2).

As mentioned in the previous section and in the beginning of this section the term M in a definition $\mu y.M$ corresponds to a covariant functor. The term $\lambda y{:}Type_i.M$ corresponds to the object mapping part of this functor. From the diagram above and the diagrams in the previous section we see that also the morphism mapping part is used, e.g., in the reduction rule at the end of Section 2. The morphism mapping part of the functors cannot be expressed in terms of M. However, it is possible to construct mechanically from M a term Φ_M corresponding to the morphism part of the functor. In Table 4 the construction this term is defined by recursion on the syntactical structure of the type operator.

A term T_M defined according to Table 4 is restricted to monomorphic functions. If $\Gamma, y{:}Type_i \vdash M{:}Type_i$, we see from Table 4

$$\Phi_M{:}\prod A, B{:}Type_i.(A \to B) \to ([A/y]M \to [B/y]M).$$

There are two reasons for this restriction. If we inspect the diagram above, we see that the morphism mapping part of T_M is applied to the morphism $\langle \mathrm{id}, \mathbf{rec}(u)\rangle{:}A \to \sum x{:}A.B$ and to the first projection $\pi_1{:}(\sum x{:}A.B) \to A$, both of which are monomorphic. Hence it is sufficient for our purposes to define Φ_M only for monomorphic functions. It is possible to define T_M for polymorphic functions, but some problems will arise. For example, consider a type constructor $M[y] \equiv L + y$ and a function $f{:}\prod x{:}D.E$, for some types D, E, and L. If T_M is extended to polymorphic functions, what is the type of $T_M f$? By analogy to Table 4 we may try $\prod x{:}(L+D).(L+E)$, but this is an illdefined type. The solution seems to be to define the type of $T_M f$ to be

$$\prod z{:}(L+D).(L + \mathbf{case}(z, \lambda v{:}L.\mathbf{1}, \lambda v{:}D.[v/x]E)).$$

The above construction breaks the type uniformity of the definition in Table 4, but can of course be used. This snag is not mentioned in (Coquand and Paulin, 1989), although it will occur if the more sketchy parts in the beginning of the paper are worked out in detail.

The inductive types can be introduced both at the impredicative type level, that is, *Prop*, or at the predicative type levels, that is, $Type_i$. The main goal of this paper is to show how the inductive types can be

introduced at the predicative levels of ECC. If the calculus is to be used as a programming logic, the impredicative level corresponds to the specifications, and the predicative part corresponds to a higher order functional programming language. Inductive types on the predicative levels correspond to data types in such a language.

The introduction of inductive types at the impredicative level of ECC would closely correspond to what was done by Constable and Mendler (1985) and is not included in this paper. It will, however, be mentioned that inductive types on the impredicative level are defined analogously to the inductive types on the predicative levels. The main difference is that at this level the sum can be encoded in terms of the $\prod$-operator, and that the strictly positive occurrence requirement can be relaxed to a positive occurrence requirement.

The necessary extensions of the term calculus of ECC are given in Table 5. The new reduction rule is rather different from the previous ones. The form of the right hand side depends heavily on the type of the term on

TABLE 5

The Extensions of the Term Calculus and the Type Rules to Inductive Types

Syntax:

If M and N are terms and y is a variable then $\mu y.M$, $\mathbf{rec}(N)$, and $\mathbf{intro}_{\mu y.M} N$ are terms

Binding rule:

All free occurrences of y in M become in $\mu y.M$

The cumulative type relation:

If $M \leqslant M'$ then $\mu y.M \leqslant \mu y.M'$

Reduction rule:

If y occurs strictly positive in M then

$$\mathbf{rec}(u)(\mathbf{intro}_{\mu y.M} N) \rhd u((\Phi_M(\lambda z : A.\ \mathbf{pair}_{\sum x:A.B}(z, \mathbf{rec}(u)z)))N),$$

where Φ_M is the term defined in Table 4, and $A \equiv \mu y.M$.

Typing:

$$(I\mu)\quad \frac{\Gamma, y : Type_i \vdash M : Type_i}{\Gamma \vdash \mu y.M : Type_i} \qquad y \text{ occurs strictly positively in } M$$

$$(intro)\quad \frac{\Gamma \vdash \mu y.M : Type_i \quad \Gamma \vdash N : [\mu y.M/y]M}{\Gamma \vdash \mathbf{intro}_{\mu y.M} N : \mu y.M}$$

$$(rec)\quad \frac{\Gamma \vdash u : \prod z : [(\sum x : A.B)/y]M.\,[\mathbf{intro}((\Phi_M \pi_1)z)/x]B}{\Gamma \vdash \mathbf{rec}(u) : \prod x : A.B} \qquad \text{where } A \equiv \mu y.M, \text{ and the term } \Phi_M \text{ is defined as shown in Table 4}$$

left hand side. This was not the case in the reduction rules for application, pair and the case operator. These rules depend only on the syntactical form of the terms and not on the type of the terms.

The new rules given in Table 5 represent a non-trivial extension of ECC. Does the augmented calculus enjoy the strong normalization property? I do not give any normalization proof in this paper. However, the strictly positive occurrence requirement for the inductive types and the fact that all terms in ECC are strongly normalizable make it rather obvious that every recursive function must terminate with a unique result. As mentioned above, the reduction rule for recursive functions does involve the types of the terms. This may make the normalization proof complex and complicated.

Examples—Induction

The introduction rule for recursive functions, rule (*rec*) in Table 5, can be constructed from the induction axiom for natural numbers by translating this axiom into ECC extended with inductive types and then generalizing the parts depending specifically upon the type N of natumral numbers.

The induction rule for natural numbers in (on natural deduction form)

$$\frac{P(0) \quad \forall x(P(x) \to P(Sx))}{\forall P(x)}. \tag{1}$$

How should this axiom be translated (by the propositions-as-types principle) into ECC extended with inductive types? A unary predicate on the natural numbers is expressed as a term of type $N \to \mathit{Prop}$. The definition of zero and the successor function S gives the following rule

$$\frac{a{:}P(\mathbf{intro}(\mathbf{inl}\ \bot)) \quad h{:}\prod x{:}N.(P(x) \to P(\mathbf{intro}(\mathbf{inr}\ x)))}{\mathit{ind}(a, h){:}\prod x{:}N.P(x)} \tag{2}$$

where $P{:}\, N \to \mathit{Prop}$. This rule could be postulated as the induction rule for the recursive type N in ECC. We will, however, show that the above rule can be seen as an instance of the introduction rule for (*rec*) in Table 5.

The two premises in (2) can be amalgamated into one by using the disjoint sum operator as follows: The first premise can be replaced by

$$(\lambda y.a){:}\prod y{:}\mathbf{1}.P(\mathbf{intro}(\mathbf{inl}\ y)) \tag{3}$$

since the theory is extended with the rule $\mathbf{1}\lambda$. (For the sake of readability the explicit types of the arguments are dropped in the above and in the

following equations.) The second premise is in fact a function in two arguments and can by using the type equivalence

$$\prod x{:}A.\prod y{:}Bx.Cxy \cong \prod z{:}\left(\sum x{:}A.Bx\right).C(\pi_1 z)(\pi_2 z)$$

be replaced by

$$(\lambda x.h(\pi_1 x, \pi_2 x)){:}\prod y{:}\left(\sum x{:}N.P(x)\right).P(\mathbf{intro}(\mathbf{inr}(\pi_1 y))). \tag{4}$$

Let

$$u = \lambda y.\mathbf{case}(y, \lambda z.a, (\lambda x.h(\pi_1 x, \pi_2 x))),$$

By using (3), (4), the *case*-operator, and the disjoint sum, the rule (2) can be replaced by the equivalent rule

$$\frac{u{:}\prod y{:}(1+\sum x{:}N.P(x)).P(\mathbf{intro}(\mathbf{case}(y, \lambda z.\mathbf{inl}\, z, \lambda v.\mathbf{inr}(\pi_1 v))))}{\mathbf{rec}(u){:}\prod x{:}N.P(x)}. \tag{5}$$

The type $1+\sum x{:}N.P(x)$ is intuitively equal to $T_N(\sum x{:}N.P(x))$.

The operator Φ_N corresponding to the function mapping part of T_N (see Table 4) makes it possible to write

$$\mathbf{case}(y, \lambda z.\mathbf{inl}\, z, \lambda v.\mathbf{inr}(\pi_1 v))$$

as $(\Phi_N \pi_1)\, y$. Hence (2) can be written as

$$\frac{u{:}\prod y{:}\Phi_N(\sum x{:}N.P(x)).P(\mathbf{intro}((\Phi_N \pi_1)\, y))}{\mathbf{rec}(u){:}\prod x{:}N.P(x)}, \tag{6}$$

which is an instance of the introduction rule for recursive functions, rule (*rec*) in Table 5. The transformation of (2) into (5) is a deduction in the metatheory and gives a scheme for induction proofs done by refinement.

What we have done is simply to translate the standard induction rule for natural numbers into the theory. The general rule (*rec*) is obtained by isolating the parts depending on the type N. The correspondence between the intuitive induction rule for any other inductive type and the general introduction rule can be shown analogously.

EXAMPLE 3.1. Let the type of natural numbers be defined as $N \equiv \mu y.\mathbf{1} + y$. We want to define the function $double(n) = 2n$ as a recursive function of type $N \to N$. The standard way of defining *double* as a recursive function is

$$double\ 0 = 0$$

$$double\ Sx = SS(double\ x).$$

In ECC extended with inductive types, we first combine the parts of the above definition not depending on *double* into one function u, that is,

$$u_{double} = \lambda x{:}\, 1 + N \times N.\mathbf{case}(x, \lambda y{:}\mathbf{1}.\underline{0}, \lambda\langle m, n\rangle{:}\, N \times N.SSn).$$

In the above definition a kind of pattern matching is used to simplify the expression, that is, $\langle m, n\rangle$ is used instead of a single variable together with the projection function.

The desired function is defined as

$$double = \mathbf{rec}(u_{double}).$$

The reduction (computation) of *double* applied to one will be as follows:

$$\begin{aligned} & double(S\underline{0}) \\ \rhd\ & \mathbf{rec}(u_{double})(\mathbf{intro}(\mathbf{inr}\ \underline{0})) \rhd u_{double}(\mathbf{inr}\langle \underline{0}, double\ \underline{0}\rangle) \\ \rhd\ & SS(double\ \underline{0}) \\ \rhd\ & SS(\mathbf{rec}(u_{double})(\mathbf{intro}(\mathbf{inl}\ \perp))) \rhd (u_{double}(\mathbf{inl}\ \perp)) \\ \rhd\ & SS\underline{0}. \end{aligned}$$

EXAMPLE 3.2. Assume that $A{:}\,Type$. The type of lists with elements of type A can be defined as

$$list = = \mu X.\mathbf{1} + A \times X.$$

The constructor function and the empty list are defined as

$$\begin{aligned} cons = & = \lambda x{:}A.\lambda l{:}list.\mathbf{intro}(\mathbf{inr}\langle a, l\rangle) \\ nil = & = \mathbf{intro}(\mathbf{inl}\ \perp). \end{aligned}$$

The type indices are dropped for the sake of convenience.

The way the length function $length{:}list \to N$ is defined is rather analogously to the above definition of *double*. We first define the function u_{length} describing the basis of the recursion and the induction step and then the recursive function:

$$\begin{aligned} u_{length} = & = \lambda x{:}1 + A \times list \times N.\mathbf{case}(x, \lambda y{:}\mathbf{1}.0, \lambda\langle a, l, n\rangle{:}A \times list \times N.Sn) \\ length = & = \mathbf{rec}(u_{length}). \end{aligned}$$

In the two above examples the full complexity of the introduction rule is not needed. The type of the range of the functions, N, is not a dependent type. Second, both functions, as all the most feaquently used functions, can

be defined to a simplified recursion scheme of the form (for primitive recursion)

$$rec(a,h)\underline{0} = a$$

$$rec(a,h)\,Sx = h(rec(a,h)x).$$

That is, the recursion step does not depend upon the predecessor(s), but only upon the value of the recursive function to the predecessor(s). For such simple functions the introduction rule could have been of the form

$$\frac{\Gamma \vdash u : \prod y{:}\Phi(P).P}{\Gamma \vdash \mathbf{rec}(u) : \prod x{:}A.P}.$$

However, if the induction aspect of the introduction rule, (*rec*) in Table 5, is used, then the dependent product is fully needed. The example below illustrates this.

EXAMPLE 3.3. We will show how reflexivity of equality on natural numbers can be deduced from the axioms $\vdash 0 = 0$ and $\vdash x = y \to Sx = sy$ by the use of the rule (*rec*) in Table 5.

Equality and axioms expressed in ECC:

$$\begin{aligned} &eq: && N \to N \to Prop \\ &eqzero: && eq\underline{0}\underline{0} \\ &eqsucc: && \prod x, y{:}N.(eq\,x\,y) \to eq(Sx)(Sy). \end{aligned}$$

Our proposition can be written as

$$\prod x{:}N.P(x),$$

where

$$P = = \lambda x{:}N.eq\,x\,x.$$

The proof of the proposition is given below. The proof is a refinement proof in the style of LEGO (Pollack, 1988). The comments to the right indicate the (refinement) rule used in each step in a thought refinement proof in this system:

$$\dfrac{\dfrac{\dfrac{\textit{Lemma 1} \qquad \textit{Lemma 2}}{y{:}\mathbf{1} + \sum x{:}N.P(x) \vdash P(\mathbf{intro}((\Phi_N \pi_1)\ y))}\ \textit{case}}{\vdash \prod y{:}\mathbf{1} + \sum x{:}N.P(x).P(\mathbf{int}((\Phi_N \pi_1)\ y))}\ \textit{intro}}{\prod x{:}N.P(x)}\ \textit{ind}$$

Let $\Gamma = y{:}\mathbf{1} + \sum x{:}N.P(x)$. The two lemmas represent the induction basis and the induction step.

LEMMA 1 (Induction Basis).

$$\Gamma \vdash \prod z{:}\mathbf{1}.P(\mathbf{intro}((\Phi_N \pi_1)\,\mathbf{inl}\,z)).$$

It is not immediate that the above expression constitutes the induction basis. However, the term $\mathbf{intro}((\Phi_N \pi_1)\,\mathbf{inl}\,z)$ reduces to $\mathbf{intro}(\mathbf{inl}\,z)$, that is, zero. Hence Lemma 1 is $P(0)$.

LEMMA 2 (Induction Step).

$$\Gamma \vdash \prod v{:}\sum x{:}N.P(x).P(\mathbf{intro}((\Phi_N \pi_1)\,\mathbf{inr}\,v)).$$

In Lemma 2 the term $\mathbf{intro}((\Phi_N \pi_1)\,\mathbf{inr}\,v)$ is reduced to $\mathbf{intro}(\mathbf{inr}(\pi_1 v))$, that is, $S(\pi_1 v)$. Every $v{:}\sum x{:}N.P(x)$ is on the form $\langle n, p\rangle$. The term $n{:}N$ represents a natural number. Under the propositions-as-types principle the term $p{:}P(n)$ represents the (intuitionistic) proof of the proposition $P(n)$. Hence Lemma 2 expresses the first order formula $\forall x{:}N.P(x) \to P(Sx)v$ in the type theory. In the proof of the induction step $P(n)$ is the induction hypothesis. The refinement proofs for the lemmas are given below.

Proof of Lemma 1.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash eq(\mathbf{intro}(\mathbf{inl}\,\bot))(\mathbf{intro}(\mathbf{inl}\,\bot))}}{\Gamma \vdash \prod z{:}\mathbf{1}.eq(\mathbf{intro}(\mathbf{inl}\,z))(\mathbf{intro}(\mathbf{inl}\,z))}}{\Gamma \vdash \prod z{:}\mathbf{1}.P(\mathbf{intro}((\Phi_N \pi_1)\,\mathbf{inl}\,z))}$$

refine by eqzero

refine by $\mathbf{1}\lambda$

normalization

The proof of Lemma 1 is straightforward, but illustrates the use of the rule $\mathbf{1}\lambda$. This rule postulates that $\bot$ is the single closed normal-form term of type $\mathbf{1}$.

Proof of Lemma 2.

$$\dfrac{\dfrac{\dfrac{}{\Gamma, v{:}\sum x{:}N.eq\,x\,x \vdash eq(\pi_1 v)(\pi_1 v)}}{\Gamma, v{:}\sum x{:}N.eq\,x\,x \vdash eq(\mathbf{intro}(\mathbf{inr}(\pi_1 v))(\mathbf{intro}(\mathbf{inr}(\pi_1 v))))}}{\Gamma \vdash \prod v{:}\sum x{:}N.P(x).P(\mathbf{intro}((\Phi_N \pi_1)\,\mathbf{inr}\,v))}$$

refine by $(\pi_2 v){:}eq(\pi_1 v)(\pi_1 v)$

refine by eqsuc

norm., intro

In the above proof the induction hypothesis is the second component of the pair $v{:}\sum x{:}N.eq\,x\,x$ and is applied in the uppermost line of the proof.

4. An ω-**Set** Model for ECC with Inductive Types

We will now see how the inductive types can be interpreted in the framework of the ω-**Set** based model of ECC given in (Luo, 1989b). Luo's model is an extension and adaptation of the ω-**Set** model for second order λ-calculus (see Girard, 1972, 1986; Reynolds, 1974) developed by Moggi, Hyland, and others (Longo and Moggi, 1988; Hyland, 1987; Hyland and Pitts, 1987).

The ω-**Set** model for ECC is not widely known, and it seems appropriate to start this section with a summary of Luo's model.

4.1. *The* ω-**Set** *Model for ECC*

The ω-**Set** model is basically an (intuitionistic) set theoretical model. As pointed out in (Luo, 1989b) such a model for ECC has to satisfy the following requirements:

1. $[\![Prop]\!] \in [\![Type_0]\!] \in [\![Type_1]\!] \in \ldots$
2. $[\![Prop]\!] \subseteq [\![Type_0]\!] \subseteq [\![Type_1]\!] \subseteq \ldots$
3. For all $i \in \omega$ the interpretation of $Type_i$ must be closed under the interpretations of $\prod$ and $\sum$.
4. The interpretation of *Prop* must be closed under $\prod$.

These requirements make it impossible to construct a naïve set theoretical model for the calculus.

However, it is possible to extend the ω-**Set** model for the second order λ-calculus to a model for the ECC. The ω-**Set** model is based on the idea of interpreting types as partial equivalence relations (see Longo and Moggi, 1988).

The model can roughly be outlined as follows: The types are interpreted as ω-sets, that is, sets equipped with a realizability relation. The ω-sets form a locally Cartesian closed category in which the morphisms are set functions preserving the corresponding realizability relations. The impredicative level, *Prop*, is interpreted as the full subcategory **PROP** isomorphic to the category of partial equivalence relations over ω. The predicative hierarchy (of the $Type_i$'s) is interpreted by the use of the "set universes," V_{κ_i}, where V_{κ_i} is the κ_ith level in the cumulative set hierarchy and κ_i is the ith inaccessible cardinal. That is, each $Type_i$ is the full subcategory where the objects are restricted to sets in $V_{\kappa_i k}$. The hierarchy of these subcategories of the category of ω-set satisfies the above requirements.

Definition 4.1. The category ω-**Set** is defined as follows:

Objects: An ω-set $A = (|A|, \Vdash_A)$ consists of a (carrier) set $|A|$ and a (realizability) relation $\Vdash_A \subseteq \omega \times |A|$ such that

$$\forall a \in |A| . \exists n \in \omega . n \Vdash_A a.$$

Morphisms. A morphism between two ω-sets A and B is an ordinary set function $f: |A| \to |B|$ such that $\exists n \in \omega . n \Vdash_{A \to B} f$ where (with $n.m$ denoting Kleene application of n to

$$n \Vdash_{A \to B} \text{iff } \forall a \in |A| . \forall m \in \omega . m \Vdash_A a \to n.m \Vdash_B f(a).$$

4.1.1. *The Predicative Hierarchy*

As mentioned above inaccessible cardinals are used to interpret the predicative hierarchy (a cardinal number κ is (strongly) inaccessible iff it is uncountable and $\forall \lambda < \kappa . 2^{\lambda} < \kappa$ (Devlin, 1979)). Consider ZFC with inaccessible cardinals $\kappa_0 < \kappa_1 < \ldots$. For each inacessible cardinal κ_i let V_{κ_i} be the corresponding level in the cumulative set hierarchy.

DEFINITION 4.2. ω-**Set**(i) is the full subcategory of ω-**Set** such that the carrier sets of the objects are in V_{κ_i}.

Let Δ be the following inclusion functor from the category of sets into ω-**Set**:

$$\Delta(X) = (X, \omega \times X) \qquad \text{for sets}$$

$$\Delta(f) = f \qquad \text{for set functions.}$$

As mentioned above, each level $Type_i$ will be interpreted as the full subcategory of ω-**Set** where the carrier sets are in V_{κ_i}. We have that for all ordinals α and β if $\alpha < \beta$ then $V_\alpha \subseteq V_\beta$ and $V_\alpha \in V_\beta$. Hence the first requirement is satisfied for the predicative hierarchy. The second is satisfied by viewing each ω-**Set**(i) as an ω-set through the inclusion functor Δ, that is, $\Delta(\text{Obj}(\omega\text{-}\mathbf{Set}(i))) \in \text{Obj}(\omega\text{-}\mathbf{Set}(i+1))$.

4.1.2. *The Impredicative Level*

The impredicative level *Prop* is interpreted as the small category **PROP** of partial equivalence relations over ω. However, this category is not closed under the interpretation of $\prod$. Hence it is necessary to use a trick and first introduce the following category-isomorphic non-small category.

DEFINITION 4.3. A *modest set* is an ω-set A such that

$$\forall n \in \omega . \forall a, b \in |A| . n \Vdash_A a \wedge n \Vdash_b b \Rightarrow a = b.$$

The category of modest sets, **M**, is the full subcategory of ω-**Set** with the modest sets as its objects.

We then introduce the category of partial equivalence relations. Strictly speaking the category below is the embedding of this category into **M**.

DEFINITION 4.4. The category **PROP** is the full subcategory of **M** (and hence of ω-**Set**) with the following object set:

$$\mathrm{Obj}(\mathbf{PROP}) = \{(Q(R), \in) \mid R \subseteq \omega \times \omega \text{ is a partial equivalence relation}\},$$

where $Q(R)$ is the quotient set of ω with respect to R, that is, $Q(R) = \{[n]_R \mid (n, n) \in R\}$.

The following lemma states formally the relation between the categories **PROP** and **M**. The proof in (Luo, 1989b) contains important definitions used in the interpretation of the derivable judgements, and is for this reason given here.

LEMMA 4.1. *There is an equivalence of categories* $\mathbf{back}: \mathbf{PROP}$ *such that* $\mathbf{back}(A) \cong A$ *for* $A \in \mathrm{Obj}(M)$ *and* $\mathbf{back}(P) = P$ *for* $P \in \mathrm{Obj}(\mathbf{PROP})$.

Proof. Define the functor $\mathbf{back}: \mathbf{M} \to \mathbf{PROP}$ as follows:

$$A \in \mathrm{Obj}(M) \qquad \mathbf{back}(A) = (Q(R_A), \in)$$

$$f: \mathbf{M}(A, B) \qquad \mathbf{back}(f)([p]_{R_A}) = [n \times p]_{R_B},$$

where $n \Vdash_{A \to B} f$, and where

$$R_A = \{(n, m) \mid \exists a \in A.\, n \Vdash_A a \wedge m \Vdash_A a\}$$

$$R_B = \{(n, m) \mid \exists b \in B.\, n \Vdash_B b \wedge m \Vdash_B b\}.$$

The functor **back** is a category equivalence with the inclusion functor $\mathbf{inc}: \mathbf{PROP} \to \mathbf{M}$ as its inverse. We have the natural transformations

$$id: id_{\mathbf{PROP}} \to \mathbf{back} \circ \mathbf{inc}$$

$$\eta: id_{\mathbf{M}} \to \mathbf{inc} \circ \mathbf{back},$$

where id is the identity and η is defined as follows: for $A \in \mathrm{Obj}(\mathbf{M})$ and $a \in |A|$, $\eta_A(a) = [\eta]_{R_A}$, where $n \Vdash_A a$. Hence, for all $A \in \mathrm{Obj}(\mathbf{M})$, $\mathbf{back}(A) = \mathbf{inc} \circ \mathbf{back}(A) \cong A$. Furthermore, for $P = (Q(R), \in) \in \mathrm{Obj}(\mathbf{PROP})$, it is easy to show that $R_P = R$, and hence $\mathbf{back}(P) = (Q(R_P), \in) = (Q(R), \in) = P$.

4.1.3. *The Interpretation of the Valid Contexts and the Derivable Judgments*

As shown earlier both *Prop* and the predicative type universes $Type_i$ are interpreted as subcategories of the category ω-**Set**. Each type A will be interpreted as an object of the category corresponding to its kind. That is,

a type $A: Type_i$ will be interpreted as an object of the category ω-**Set**(i). A term of type A is interpreted as an element in the carrier set of the interpretation of A.

The above picture is not completely correct, it is of course only true for closed types and terms. Types and terms with free variables depend on the context, and are interpreted as functions in their free variables. The picture is also somewhat complicated by the double nature of the types. A type is also a term of its kind. Hence it must be possible to see the interpretation of a type both as an ω-**Set** object and as an element in the carrier set of such an object. Moreover, the interpretations of the $Type_i$'s must be able to play the role as subcategories of ω-**Set**, as ω-**Set** objects, and as elements in a carrier set.

The valid contexts are interpreted as ω-**Set** objects. The empty context is interpreted as the terminal object $(\{*\}, \omega \times \{*\})$. A context $\Gamma, x:A$ is interpreted as $[\![\Gamma, x:A]\!] = \sigma([\![\Gamma]\!], [\![\Gamma \vdash A:K_A]\!])$, where K_A is the kind of A. As seen from the definition of the operator σ in Table 6, the carrier set of $[\![\Gamma, x:A]\!]$ consists of nested pairs (tuples). The first component of such a pair is a tuple γ which is possible value assignment to the variables defined in Γ. The second component is an element in the carrier set of the interpretation of A constructed under the value assignment γ.

TABLE 6

The Operators for the Interpretation of Contexts, $\sum$- and $\prod$-types

Assume that Γ is an ω-set and $A: |\Gamma| \to \omega$-**Set**.

(σ) The ω-set $\sigma(\Gamma, A)$ is defined as follows:

$$|\sigma(\Gamma, A)| = \{(\gamma, a) \mid \gamma \in |\Gamma|, a \in |A(\gamma)|\}$$

$$\langle m, n\rangle \Vdash_{\sigma(\Gamma, A)} (\gamma, a) \text{ if and only if } m \Vdash_{\Gamma} \gamma \wedge n \Vdash_{A(\gamma)} a$$

(σ_Γ) Assume that $B: |\sigma(\Gamma, A)| \to \omega$-**Set**. The function $\sigma_\Gamma(A, B): |\Gamma| \to \omega$-**Set** is defined as follows: Let for each $\gamma \in |\Gamma|$

$$|\sigma_\Gamma(A, B)(\gamma)| = \{(a, b) \mid a \in |A(\gamma)| \wedge b \in |B(\gamma, a)|\}$$

$$\langle m, n\rangle \Vdash_{\sigma_\Gamma(A, B)(\gamma)} (a, b) \text{ if and only if } m \Vdash_{A(\gamma)} a \wedge n \Vdash_{B(\gamma, a)} b$$

(π_Γ) Assume that $B: |\sigma(\Gamma, A)| \to \omega$-**Set**. The function $\pi_\Gamma(A, B): |\Gamma| \to \omega$-**Set** is defined as follows: Let for each $\gamma \in |\Gamma|$

$$|\pi_\Gamma(A, B)(\gamma)| = \left\{ f: |A(\gamma)| \to \bigcup_{a \in |A(\gamma)|} |B(\gamma, a)| \mid \forall a \in |A(\gamma)|.\, f(a) \in |B(\gamma, a)| \right.$$
$$\left. \wedge\ \exists n \in \omega.\, n \Vdash_{\pi_\Gamma(A, B)(\gamma)} f \right\}$$

$$n \Vdash_{\pi_\Gamma(A, B)(\gamma)} f \text{ iff } \forall a \in |A(\gamma)|\ \forall p \in \omega.\, p \Vdash_{A(\gamma)} a \Rightarrow n.p \Vdash_{B(\gamma, a)} f(a)$$

A derivable judgement, $\Gamma \vdash M:A$, is interpreted as a function from the interpretation of the context Γ into the interpretation of the context Γ, $x:A$, that is, as a function mapping a value assignment tuple γ to a new tuple (γ, a), where a is an element in the carrier set of the interpretation of A under γ.

The operators σ_Γ and π_Γ in Table 6 are used to construct the interpretation of $\sum$- and $\prod$-types respectively.

From the definition of σ_Γ in Table 6 and the use of this operator in Table 7 we see that for a given value assignment γ the interpretation of a type $\sum x:A.B$ is an ω-**Set**-object consisting of all pairs (a, b), where a and b are as follows: The first component a is an element in the carrier set of the interpretation of A under γ. The second component b is an element in the carrier set of the interpretation of B under the value assignment (γ, a). If B does not depend on x then the carrier set is the ordinary Cartesian product.

From the same two tables we also see that the interpretation of a type $\prod x:A.B$ under a given value assignment γ is an ω-**Set** object with the following carrier set: The set consists of the ω-**Set** morphisms from the interpretation of A under γ, $[\![A]\!]_\gamma$, to the object constructed as the $a \in |[\![A]\!]_\gamma|$-indexed union of the interpretations of B under (γ, a), such that for each $a \in |[\![A]\!]_\gamma|$ the function value $f(a)$ is in the interpretation of B under (γ, a). If B is independent of $x:A$, then the interpretation of $\prod x:A.B$ is the ordinary function space object in ω-**Set**.

The two following lemmas are necessary to ensure that the model satisfies the requirements in the beginning of this section.

LEMMA 4.2. (Luo, 1989b). *ω-**Set**(i) is closed under the operators σ_Γ and π_Γ. That is, if $A, B:|\Gamma| \to \omega$-**Set**(i) and $C:|\sigma(\Gamma, A)| \to \omega$-**Set**$(i)$ then*

$$\sigma_\Gamma(A, C), \pi_\Gamma(A, C):|\Gamma| \to \omega\text{-}\mathbf{Set}(i).$$

LEMMA 4.3 (Longo and Moggi, 1988). *The operator π_Γ is closed for the modest sets in the sense that, for all $A:|\Gamma| \to \omega$-**Set**, $B:|\sigma(\Gamma, A)| \to$ **M**, we have $\pi_\Gamma(A, B):|\Gamma| \to$ **M**.*

To be able to see the interpretations of the types both as objects in ω-**Set** and as elements in a carrier set of such an object, the interpretations of the derivable judgements must satisfy the property below.

DEFINITION 4.5. Let $\Gamma \in \omega$-**Set** and $A:|\Gamma| \to \omega$-**Set**. A morphism $f: \Gamma \to \sigma(\Gamma, A)$ in ω-**Set** satisfies the first property (FPP), written as

$$f: \Gamma \to_{\text{FPP}} \sigma(\Gamma, A),$$

TABLE 7

The Interpretation of the Valid Contexts and the Derivable Judgments

Valid contexts:

- The empty context "$\langle\ \rangle$" is valid and is interpreted as the terminal object of ω-**Set** :

$$[\![\langle\ \rangle]\!] = \mathbf{1} = (\{*\}, \omega \times \{*\})$$

- A valid context "$\Gamma, x : A$" is interpreted as the ω-**Set** object:

$$[\![\Gamma, x : A]\!] = \sigma([\![\Gamma]\!], [\![\Gamma \vdash A : K]\!]^*)$$

Derivable judgements:

(**AX**)(**C**) $[\![\Gamma \vdash \mathit{Prop} : \mathit{Type}_0]\!](\gamma) = (\gamma, \Delta(\mathrm{Obj}(\mathbf{PROP})))$

(**T**) $[\![\Gamma \vdash \mathit{Type}_i : \mathit{Type}_{i+1}]\!](\gamma) = (\gamma, \Delta(\mathrm{Obj}(\omega\text{-}\mathbf{Set}(j))))$

(**var**) $[\![\Gamma, x : A, \Gamma' \vdash x : A]\!](\gamma, a, \gamma') = ((\gamma, a, \gamma'), a)$

($\prod$1) $[\![\Gamma \vdash \prod x : A.P : \mathit{Prop}]\!] = \mathbf{back} \circ \pi_\Gamma([\![\Gamma \vdash A : T_\Gamma(A)]\!], [\![\Gamma, x : A \vdash P : \mathit{Prop}]\!])$

($\prod$2) $[\![\Gamma \vdash \prod x : A.P : \mathit{Type}_i]\!] = \pi_\Gamma([\![\Gamma \vdash A : T_\Gamma(A)]\!], [\![\Gamma, x : A \vdash P : \mathit{Type}_j]\!])$

(λ) There are two cases:

- If $\Gamma \not\vdash \prod x : A.B : \mathit{Prop}$ we define

$$[\![\Gamma \vdash \lambda x : A.M : \prod x : A.B]\!](\gamma) = (\gamma, g_\gamma)$$

where g_γ is the function such that $g_\gamma(a) = b$ iff $[\![\Gamma, x : A \vdash M : B]\!](\gamma, a) = ((\gamma. a), b)$, that is, $g_\gamma \in |P(\gamma)|$ where $P = \pi_\Gamma([\![\Gamma \vdash A : T_\Gamma(A)]\!], [\![\Gamma, x : A \vdash B : T_{\Gamma, x:A}(B)]\!]^*)$

- If $\Gamma \vdash \prod x : A.B : \mathit{Prop}$ we define

$$[\![\Gamma \vdash \lambda x : A.M : \prod x : A.B]\!](\gamma) = (\gamma, \eta_{P(\gamma)}(g_\gamma))$$

where P and g_γ are as above and η is the natural transformation defined in the proof of Lemma 4.1

(**app**) There are two cases:

- If $\Gamma, x : A \not\vdash B : \mathit{Prop}$, assume that for each $\gamma \in |[\![\Gamma]\!]|$

$$[\![\Gamma \vdash M : \prod x : A.B]\!](\gamma) = (\gamma, f)$$
$$[\![\Gamma \vdash N : A']\!](\gamma) = (\gamma, a)$$

Then $[\![\Gamma \vdash MN : [N/x]B]\!](\gamma) = (\gamma, f(a))$

- If $\Gamma, x : A \vdash B : \mathit{Prop}$, assume that for each $\gamma \in |[\![\Gamma]\!]|$

$$[\![\Gamma \vdash M : \prod x : A.B]\!](\gamma) = (\gamma, [n]_{R_{[\![\Gamma \vdash \prod x:A.B:\mathit{Prop}]\!](\gamma)}})$$
$$[\![\Gamma \vdash N : A']\!](\gamma) = (\gamma, a)$$
$$p \Vdash_{[\![\Gamma \vdash A : T_\Gamma(A)]\!](\gamma)} a$$

Then $[\![\Gamma \vdash MN : [N/X]B]\!](\gamma) = (\gamma, [n.p]_{R_{[\![\Gamma, x:A \vdash B:\mathit{Prop}]\!](\gamma, a)}})$, where $R_{[\![\Gamma \vdash \prod x:A.B:\mathit{Prop}]\!](\gamma)}$ and $R_{[\![\Gamma, x:A \vdash B:\mathit{Prop}]\!](\gamma, a)}$ are defined as explained in the proof of Lemma 4.1

($\sum$) $[\![\Gamma \vdash \sum x : A.B : \mathit{Type}_i]\!] = \sigma_\Gamma([\![\Gamma \vdash A : K]\!, [\![\Gamma, x : A \vdash B : K']\!])$

(**pair**) Assume for each $\gamma \in |[\![\Gamma]\!]|$ that $[\![\Gamma \vdash M : A']\!](\gamma) = (\gamma, m)$ and $[\![\Gamma \vdash N : B']\!](\gamma) = (\gamma, n)$, then

$$[\![\Gamma \vdash \mathbf{pair}_{\sum x:A.B}(M, N) : \sum x : A.B]\!](\gamma) = (\gamma, (m, n))$$

($\prod$1), ($\prod$2) Assume for each $\gamma \in |[\![\Gamma]\!]|$ that $[\![\Gamma \vdash M : \sum x : A.B]\!](\gamma) = (\gamma, (a, b))$, then

$$[\![\Gamma \vdash \pi_1(M) : A]\!](\gamma) = (\gamma, a)$$
$$[\![\Gamma \vdash \pi_2(M) : [\pi_1(M)/x]B]\!](\gamma) = (\gamma, b)$$

(**conv**) $[\![\Gamma \vdash M : A']\!]$ is defined as $[\![\Gamma \vdash M : A]\!]$

(**cum**) $[\![\Gamma \vdash M : A']\!](\gamma) = [\![\Gamma \vdash M : A]\!](\gamma)$

if and only if $p(\Gamma, A) \circ f = id_\Gamma$, where $p(\Gamma, A): \sigma(\Gamma, A) \to \Gamma$ is the morphism defined by $p(\Gamma, A)(\gamma, a) = \gamma$.

A derivable judgment $\Gamma \vdash M:A$ will be interpreted as a function

$$[\![\Gamma \vdash M:A]\!] : [\![\Gamma]\!] \to_{\mathrm{FPP}} [\![\Gamma, x:A]\!].$$

The judgment $\Gamma, x:A$ is in turn interpreted as an ω-**Set** object by the application of σ to the interpretations of Γ and $\Gamma \vdash A:K$, where K is the type of A, that is, either *Prop* or $Type_i$ for some i (see Table 7). To stop this unfolding, we need as mentioned above to be able to see the interpretations of *Prop* and $Type_i$ both as ω-**Set** objects and as elements in carrier sets of such objects. This can easily be done since the interpretations of *Prop* and $Type_i$ are independent of the context.

DEFINITION 4.6. Assume that $\Gamma \in \mathrm{Obj}(\omega\text{-}\mathbf{Set})$ and that $K: |\Gamma| \to \omega$-**Set** is a constant function such that, for some set X, $K(\gamma) = \Delta(X) = (X, \omega \times X)$ for all $\gamma \in |\Gamma|$.

- For $f: \Gamma \to_{\mathrm{FPP}} \sigma(\Gamma, K)$ the function $f^*: |\Gamma| \to X$ is defined to be $f^*(\gamma) = x$, where $\gamma \in |\Gamma|$ and $f(\gamma) = (\gamma, x)$.
- For $g: |\Gamma| \to X$ the morphism $g^\circ: \Gamma \to_{\mathrm{FPP}} \sigma(\Gamma, K)$ is defined as $g^\circ(\gamma) = (\gamma, x)$, where $\gamma \in |\Gamma|$ and $g(\gamma) = x$.

We have the following correspondence:

LEMMA 4.4 (Luo, 1989b). *Assume that* $\Gamma \in \mathrm{Obj}(\omega\text{-}\mathbf{Set})$ *and that* $K: |\Gamma| \to \omega$-**Set** *is a constant function such that, for some set* X, $K(\gamma) = \Delta(X) = (X, \omega \times X)$ *for all* $\gamma \in |\Gamma|$. *Then there is a* 1–1 *correspondence between the set of morphisms from* Γ *to the object* $\sigma(\Gamma, K)$ *which satisfy the first projection property and the set of functions from* $|\Gamma|$ *to* X.

Proof. Use the operators * and $^\circ$ in Definition 4.6.

The correctness of the interpretation is stated in the theorem below, for a proof see (Luo, 1989b).

THEOREM 4.1. *There is an interpretation* $[\![\_]\!]$ *of the valid contexts and the derivable judgments of* **ECC** *such that*

1. *If* $\Gamma \vdash Prop: Type_0$, *then* $[\![\Gamma]\!] \in \mathrm{Obj}(\omega\text{-}\mathbf{Set})$.
2. *If* $\Gamma \vdash M:A$, *then* $[\![\Gamma \vdash M:A]\!] : [\![\Gamma]\!] \to_{\mathrm{FPP}} [\![\Gamma, x:A]\!]$.
3. *if* $\Gamma \vdash A \leqslant A'$ *then there is an inclusion morphism*

$$\mathbf{inc}_\Gamma(A, A') : [\![\Gamma, x:A]\!] \subsetneq [\![\Gamma, x:A']\!]$$

such that, if $\Gamma \vdash M:A$, $\Gamma \vdash N:A'$ *and* $M \simeq N$, *then*

$$[\![\Gamma \vdash N:A']\!] = \mathbf{inc}_{\Gamma}(A, A') \circ [\![\Gamma \vdash M:A]\!].$$

4.2. *The Extensions of the ω-Set Model to Inductive Types*

It is not necessary to do any alterations to extend the ω-**Set** model to the calculus with disjoint sums and inductive types. The single object type **1** and the disjoint sums are interpreted as shown in Table 8: The type **1** is interpreted as the terminal object, $(\{*\}, \omega \times \{*\})$, in ω-**Set**. The interpretation of the disjoint sum $A + B$ of two types A and B is roughly the disjoint sum of the interpretations of A and B. The operator δ_Γ is analogous to the

TABLE 8

The Interpretation of the Single Element Type and Disjoint Union Type

The operator δ_Γ used in the interpretation of a disjoint union: Assume that $B: |\Gamma| \to \omega$-**Set**. The function $\delta_\Gamma : |\Gamma| \to \omega$-**Set** is defined as follows:

Let for each $\gamma \in |\Gamma|$

$$|\delta_\Gamma(A, B)(\gamma)| = \{(0, a) \mid a \in |A(\gamma)|\} \cup \{(1, b) \mid b \in |B(\gamma)|\}$$

$$\langle i, n\rangle \Vdash_{\delta_\Gamma(A,B)} \text{ if and only if } \begin{cases} i = 0 \wedge n \Vdash_{A(\gamma)} x \\ i = 1 \wedge n \Vdash_{B(\gamma)} x \end{cases}$$

(F_1) $[\![F \vdash 1 : Type_0]\!](\gamma) = (\gamma, (\{*\}, \omega \times \{*\}))$

$(I_\perp)$ $[\![\Gamma \vdash \perp : 1]\!](\gamma) = (\gamma, *)$

(sum) $[\![\Gamma \vdash A + B : Type_i]\!] = \delta_\Gamma([\![\Gamma \vdash A : T_\Gamma(A)]\!], [\![\Gamma \vdash B : T_\Gamma(B)]\!])$

(inl) Assume for $\gamma \in |[\![\Gamma]\!]|$ that $[\![\Gamma \vdash M : A]\!](\gamma) = (\gamma, m)$; then

$$[\![\Gamma \vdash \mathbf{inl}_{A+B} M : A + B]\!](\gamma) = (\gamma, (0, m))$$

(inr) Assume for $\gamma \in |[\![\Gamma]\!]|$ that $[\![\Gamma \vdash N : A]\!](\gamma) = (\gamma, n)$; then

$$[\![\Gamma \vdash \mathbf{inr}_{A+B} N : A + B]\!](\gamma) = (\gamma, (1, n))$$

(case) Assume for $\gamma \in |[\![\Gamma]\!]|$ that $[\![\Gamma, x : A + B \vdash C : Type_i]\!](\gamma, (i, m)) = (\gamma, (i, m), c)$ where $c = f(\gamma, (i, m))$ and f is defined from the structure of C; assume also that

$$[\![\Gamma, x : A + B \vdash M : \textstyle\prod x : A.[(\mathbf{inl}\ y)/x]C]\!](\gamma, (i, m)) = (\gamma, (i, m), g_M)$$
$$[\![\Gamma, x : A + B \vdash N : \textstyle\prod x : B.[(\mathbf{inr}\ y)/x]C]\!](\gamma, (i, n)) = (\gamma, (i, n), h_N),$$

where

$$g_M \in [\pi_{\Gamma, x:A+B}([\![\Gamma, x : A + B \vdash A : T_\Gamma(A)]\!], [\![\Gamma, x : A + B, y : A \vdash [(\mathbf{inl}\ y)/x]C : Type_i]\!])|$$
$$h_N \in |\pi_{\Gamma, x:A+B}([\![\Gamma, x : A + B \vdash B : T_\Gamma(B)]\!], [\![\Gamma, x : A + B, y : B \vdash [(\mathbf{inr}\ y)/x]C : Type_i]\!])|;$$

then

$$[\![\mathbf{case}(x, M, N)]\!](\gamma, (i, m)) = \begin{cases} g_M(m) & \text{if } i = 0 \\ h_N(m) & \text{if } i = 1 \end{cases}$$

operators σ_Γ and π_Γ used in the interpretation of $\sum$- and $\prod$-types. The following lemma corresponds to Lemma 4.2 for the operators σ_Γ and π_Γ. The proof is trivial.

LEMMA 4.5. *ω-**Set**(i) is closed under the operator δ_Γ. That is, if $A, B: |\Gamma| \to \omega$-**Set**$(i)$ and $C: |\sigma(\Gamma, A)| \to \omega$-**Set**$(i)$ then*

$$\delta_\Gamma(A, B): |\Gamma| \to \omega\text{-}\mathbf{Set}(i).$$

An inductive type $\mu y.M$ is interpreted as the least (initial) fixed point of the functor from ω-**Set** to ω-**Set** given by the interpretation of the term M in a given context. That is, if $\Gamma, y: Type_i \vdash M: Type_i$ the interpretation of this judgment will be a function

$$[\![\Gamma, y: Type_i \vdash M: Type_i]\!] : [\![\Gamma, y: Type_i]\!] \to [\![\Gamma, y: Type_i, z: Type_i]\!]$$

$$[\![\Gamma, y: Type_i \vdash M: Type_i]\!](\gamma, t_1) = ((\gamma, t_1), t_2).$$

Hence for $\gamma \in |\Gamma|$ we can define a function $\phi_\gamma: \omega\text{-}\mathbf{Set}(i) \to \omega\text{-}\mathbf{Set}(i)$ such that

$$\phi_\gamma(t_1) = t_2 \text{ iff } [\![\Gamma, y: Type_i \vdash M: Type_i]\!](\gamma, t_1) = ((\gamma, t_1), t_2).$$

This function can be extended to an endofunctor T_M on ω-**Set**(i) such that the function mapping parts act correspondingly to the textual operator Φ_M defined in Section 3 (Table 4). Below it is shown that if y occurs strictly positive in M the function ϕ_γ is a monotonic function, and that the iterated application function is bounded. Hence ϕ_γ has fixed points and $[\![\mu y.M]\!]$ will for a given value assignment γ be the least (initial) fixed point which corresponds to the initial ϕ_γ algebra as explained in the beginning of this paper. See Table 9.

To be able to talk about monotonic functions from ω-**Set** into ω-**Set** there has to be an ordering on ω-sets.

DEFINITION 4.7. Let $A, B \in \mathrm{Obj}(\omega\text{-}\mathbf{Set})$. Then

$$A \sqsubseteq B \text{ if and only if } |A| \subseteq |B| \wedge \Vdash_A \subseteq \Vdash_B.$$

Remark. For each $a \in |A|$ there exist $n \in \omega$ such that $n \Vdash_A a$. Hence an ω-set is uniquely determined by its realizability relation, and the condition $|A| \subseteq |B|$ is strictly speaking unnecessary. The definition of an ω-set as a pair is for the same reason unnecessarily complex. In fact, an ω-set can be seen as the set of pairs defining the realizability relation.

LEMMA 4.6. *Let Γ be an ω-set.*

TABLE 9

The Interpretation of Inductive Types

Let $\Gamma, y : Type_i \vdash Type_i$ be such that y occurs strictly positive in M.

(μI) Let for $\gamma \in |[\![\Gamma]\!]|$

$$\phi_{\Gamma, y M, \gamma} : \omega\text{-}\mathbf{Set}(i) \to \omega\text{-}\mathbf{Set}(i)$$

be defined as $\phi_{\Gamma, y M, \gamma}(t_1) = t_2$ iff $[\![\Gamma, y : Type_i \vdash M : Type_i]\!](\gamma, t_1) = ((\gamma, t_1), t_2)$; then

$$[\![\Gamma \vdash \mu y . M : Type_i]\!](\gamma) = (\gamma, t_{fix}),$$

where t_{fix} is the least fixed point of $\phi_{\Gamma, y M, \gamma}$

(intro) Assume for $\gamma \in |[\![\Gamma]\!]|$ that $[\![\Gamma \vdash N : \Phi_M(\mu y . M)]\!](\gamma) = (\gamma, t)$; then

$$[\![\Gamma \vdash \mathbf{intro}_{\mu y . M} N : \mu y . M]\!](\gamma) = (\gamma, f_M(t))$$

where f_M is the ω-$\mathbf{Set}(i)$ morphism corresponding to the isomorphism from $|\phi_{\Gamma, y M, \gamma}(t_{fix})|$ to $|t_{fix}|$ realized by the number of the identity on ω

(*rec*) Let $A \equiv \mu y . M$; assume for $\gamma \in |[\![\Gamma]\!]|$ that

$$[\![\Gamma \vdash u : \prod z : [(\sum x : A . B)/y] M . [\mathbf{intro}_A((\Phi_M \pi_1)\ y)/x] B]\!](\gamma) = (\gamma, f);$$

then

$$[\![\Gamma \vdash \mathbf{rec}(u) : \prod x : A . B]\!](\gamma) = (\gamma, g_{unique}),$$

where g_{unique} is the unique ω-**Set** morphism given by the universal property of the interpretation of $\mu y . M$ seen as an initial algebra

1. *Assume that* $A : |\Gamma| \to \omega$-**Set**. *Assume also that* B: **Set** $\times |\sigma(\Gamma, A)| \to \omega$-**Set**, *that is, B is a function taking a set and an element in* $|\sigma(\Gamma, A)|$ *as arguments.*

If B is monotonic in its first argument, then the functions

$$\pi'_\Gamma(y, \gamma) = \pi_\Gamma(A, B(y))(\gamma)$$
$$\sigma'_\Gamma(y, \gamma) = \sigma_\Gamma(A, B(y))(\gamma)$$

are monotonic in argument y.

2. *Assume that* A: **Set** $\times |\Gamma| \to \omega$-**Set** *is monotonic in its first argument. Assume also that* B: **Set** $\times |\Gamma| \times$ **Set** $\to \omega$-**Set** *is monotonic in its first argument and that the function value is independent of the third argument. If B is monotonic in its first argument then the function*

$$g''_\Gamma(y, \gamma) = \sigma_\Gamma(A(y), B(y))(\gamma)$$

is monotonic in argument y.

3. *Assume that* $A, B: \mathbf{Set} \times |\Gamma| \to \omega\text{-}\mathbf{Set}$ *are monotonic in their first argument. Then the function*

$$\delta'_\Gamma(y, \gamma) = \delta_\Gamma(A(y), B(y))(\gamma)$$

is monotonic in argument y.

Proof. Consider two sets $Y \subseteq Z$. For $\gamma \in |\Gamma|$ we have

$$|\pi'_\Gamma(Y, \gamma)| = \left\{ f: |A(\gamma)| \to \bigcup_{a \in |A(\gamma)|} |B(Y, \gamma, a)| \mid \forall a \in |A(\gamma)|. \right.$$
$$\left. f(a) \in |B(Y, \gamma, a)| \wedge \exists n \in \omega. n \Vdash_{\pi_\Gamma(A, B(Y))(\gamma)} f \right\}$$

$$|\pi'_\Gamma(Z, \gamma)| = \left\{ f: |A(\gamma)| \to \bigcup_{a \in |A(\gamma)|} |B(Z, \gamma, a)| \mid \forall a \in |A(\gamma)|. \right.$$
$$\left. f(a) \in |B(Z, \gamma, a)| \wedge \exists n \in \omega. n \Vdash_{\pi_\Gamma(A, B(Z))(\gamma)} f \right\}.$$

Since B is monotonic $B(Y, \gamma, a) \sqsubseteq B(Z, \gamma, a)$. Hence the set

$$\left\{ f: |A(\gamma)| \to \bigcup_{a \in |A(\gamma)|} |B(Y, \gamma, a)| \mid |\forall a \in |A(\gamma)|. f(a) \in |B(Y, \gamma, a)| \right\}$$

is a subset of

$$\left\{ f: |A(\gamma)| \to \bigcup_{a \in |A(\gamma)|} |B(Z, \gamma, a)| \mid \forall a \in |A(\gamma)|. f(a) \in |B(Z, \gamma, a)| \right\}.$$

Since $\forall n, p \in \omega\ \forall a \in |A(\gamma)|.\ n.p \Vdash_{B(Y,\gamma,a)} f(a) \Rightarrow n.p \Vdash_{B(Y,\gamma,a)} f(a)$ we have that $n \Vdash_{\pi'(Y,\gamma)} f \Rightarrow n \Vdash_{\pi'(Z,\gamma)} f$. Hence $\pi'_\Gamma(Y, \gamma) \sqsubseteq \pi'_\Gamma(Z, \gamma)$.

The proof for σ'_Γ is analogous.

The second item in the lemma is almost obvious. If we inspect the definition of σ_Γ we see that if B is independent of A, then the ω-set $\sigma_\Gamma(A(y), B(y))(\gamma)$ is the cartesian product of $A(y, \gamma)$ and $B(y, \gamma)$.

The third item is obvious.

To be able to construct the interpretation of a term $\mu y.M$, we have to show that the operator corresponding to M is monotonic and bounded. Then we known that the operator has fixed points. The least fixed point will be interpretation of the inductive type. We start with two simplified lemmas concerning monotonicity and boundedness of operators constructed from terms M without subterms of the form $\mu y.N$, **intro** N, and **rec**$(L)N$.

LEMMA 4.7. *Assume that* $\Gamma,\ y: Type_i \vdash M: Type$, *and that* M *is constructed solely by the use of the constructors* $\sum$, $\prod$ and $+$ *from variable, constants, and terms with no occurrences of* y *and without subterms of the form* $\mu y.N$, **intro** N, *and* **rec**$(L)N$. *For* $\gamma \in |[\![\Gamma]\!]|$ *let*

$$\phi_{\Gamma y M,\gamma}: \omega\text{-}\mathbf{Set}(i) \to \omega\text{-}\mathbf{Set}(i)$$

be defined as

$$\phi_{\Gamma y M,\gamma}(t_1) = t_2 \quad \textit{iff} \quad [\![\Gamma,\ y: Type_i \vdash M: Type_i]\!](\gamma, t_1) = ((\gamma, t_1), t_2).$$

If y *occurs strictly positively in* M, *then* $\phi_{\Gamma y M,\gamma}$ *is monotonic.*

Proof. The lemma is proved by induction on the construction of $[\![\Gamma,\ y: Type_i \vdash M: Type_i]\!]$.

1. If $M \equiv y$ then $\phi_{\Gamma y M,\gamma}$ is clearly monotonic.

2. $M \equiv \prod x{:}A.B$: Since y occurs strictly positively in M, y cannot occur in A and must occur strictly positively in B. Hence by the induction hypothesis $[\![\Gamma,\ y: Type_i, x: A \vdash B]\!](\gamma, t, a)$ is monotonic in t. By Lemma 4.6 the function $[\![\Gamma,\ y: Type_i \vdash \prod x{:}A.B]\!](\gamma, t)$ is monotonic in t.

3. $M \equiv \sum x{:}A.B$: The proofs are analogous to the above case.

4. $M \equiv A + B$: If y does not occur in, say, A then the corresponding function $\Phi_{\Gamma y A,\gamma}$ is a constant function. Hence by the induction hypothesis, Lemma 4.6, and the assumption that y occurs strictly positively in $A + B$, the function $\Phi_{\Gamma y A+B,\gamma}$ must be monotonic.

LEMMA 4.8 (Boundedness). *Assume that* $\Gamma,\ y: Type_i \vdash M: Type_i$ *and that* M *is as in lemma* 4.7.

Let for $\gamma \in [\![\Gamma]\!]$ *the function* $\phi_{\Gamma y M,\gamma}$ *be defined as in Lemma* 4.7. *If* y *occurs strictly positively in* M, *then there exists a cardinal* κ *such that for all* ω*-sets* X *with* $card(|X|) \geqslant \kappa$ *we have*

$$card(|X|) = card(|\phi_{\Gamma y M,\gamma}(X)|).$$

We need the following lemma in the proof of Lemma 4.8:

LEMMA 4.9. *Let* $\varphi(x) = A \to x$ *be a set operator mapping a set* x *to the set of all functions from an infinite set* A *into* x. *For all sets* y *such that* $card(y) \geqslant card(2^{card(A)})$ *we have that* $card(y) = card(\varphi(y))$.

Proof (Lemma 4.9). *Let* $\kappa = card(A)$. For $card(y) = 2^\kappa$ we have $card(\varphi(y)) = (2^\kappa)^\kappa = 2^{\kappa \cdot \kappa} = 2^\kappa = card(y)$.

The lemma is in general proved by induction on the cardinality of y, by using the fact that for cardinal numbers $\kappa, \lambda: \lambda < cf(\kappa)$ implies that ${}^\lambda\kappa = \bigcup \{{}^\lambda\alpha \mid \alpha < \kappa\}$ (see Levy, 1979).

Proof (Lemma 4.8). The lemma is proved by induction on the construction of $\phi_{\Gamma y M,\gamma}$. In the following L_ϕ denotes the required bound for an operator ϕ.

1. If $M \equiv y$ then $L_{\phi_{\Gamma y M,\gamma}} = \varnothing$.
2. $M = \prod x{:}A.B$: In this case

$$|\phi_{\Gamma y M,\gamma}(Y)| = \Big\{ f{:}\,|A(\gamma)| \to \bigcup_{a \in |A(\gamma)|} |B(Y,\gamma,a)| \;|\forall a \in |A(\gamma)|.\; f(a) \in |B(Y,\gamma,a)| \wedge \exists n \in \omega.n \Vdash_{\pi_{\Gamma(A,B(Y))(\gamma)}} f \Big\}.$$

The induction hypothesis ensures that for each $\phi_{\gamma,a}(Y) = B(Y,\gamma,a)$ there exists a required bound $L_{\phi_{\gamma,a}}$. Let L_{ϕ_B} be an upper bound for the set $\{L_{\phi_{\gamma,a}}\}_{a \in |A(\gamma)|}$ (e.g., the union). Then, for all ω-sets Y such that $L_{\phi_B} \sqsubseteq Y$ we have

$$|\phi_{\Gamma y M,\gamma}(Y)| = \Big\{ f{:}\,|A(\gamma)| \to \bigcup_{a \in |A(\gamma)|} |Y| \;|\forall a \in |A(\gamma)|.\; f(a) \in |Y| \wedge \exists n \in \omega.n \Vdash_{\pi_{\Gamma(A,Y)(\gamma)}} f \Big\}.$$

Hence for every ω-set Y such that $L_{\phi_B} \sqsubseteq Y$ and $card(|X|) \geqslant 2^{card(|A(\gamma)|)}$, we have by Lemma 4.9 that $card(|X|) = card(|\phi_{\Gamma y M,\gamma}(X)|)$.

3. $M \equiv \sum x{:}A.B$: If B depends on x, the proof is analogous to the above case. If B does not depend on x, the interpretation of $\sum x{:}A.B$ corresponds to the cartesian product of the interpretations of A and B. Hence this case is trivial.
4. $M \equiv A + B$: trivial.

In the definition of the interpretation of a term $\mu y.M$ we need the iterated-application operator. It is defined in the standard way:

DEFINITION 4.8. Let $\phi{:}\,\omega\text{-}\mathbf{Set}(i) \to \omega\text{-}\mathbf{Set}(i)$. The iterated-application operator ϕ^α, where α is an ordinal number, is defined as follows:

$$\phi^0 = (\varnothing, \varnothing)$$
$$\phi^{\alpha+1} = \phi(\phi^\alpha)$$
$$\phi^\lambda = \bigcup_{\alpha < \lambda} \phi^\alpha \text{ for } \lambda \text{ limit ordinal.}$$

LEMMA 4.10. *Let* $\Gamma, y{:}Type_i \vdash M{:}Type_i$ *be such that* y *occurs strictly positively in* M *and* M *is as in Lemma* 4.7. *Let*

$$\phi{:}\,\omega\text{-}\mathbf{Set}(i) \to \omega\text{-}\mathbf{Set}(i)$$

be defined on the basis of $[\![\Gamma, y:Type_i \vdash M:Type_i]\!]$ *as in Lemma* 4.7. *Then* ϕ *has a least fixed point.*

Proof. By Lemma 4.7 the function $\phi_{\Gamma y M,\gamma}$ is monotonic. Hence we have the increasing sequence (of inclusions)

$$\phi^0 \sqsubseteq \phi^1 \sqsubseteq \phi^2 \sqsubseteq \cdots,$$

By Lemma 4.8 the operator ϕ is bounded. Hence there must exist a least α such that there is an isomorphism between ϕ^α and $\phi^{\alpha+1}$. By the monotonicity property we have an inclusion $i: \phi^\alpha \to \phi^{\alpha+1}$. Hence $\phi^\alpha = \phi^{\alpha+1}$.

The fixed point in the above lemma will be the interpretation of the type $\mu y.M$. However, we have only seen that a function constructed from a term $[\![\Gamma, y:Type_i \vdash M:Type_i]\!]$ is monotonic and bounded for M without subterms of the form $\mu z.N$. The general case is shown by extending the proofs of Lemmas 4.7 and 4.8 and by using the respective original lemmas as basic cases. This is straightforward, and is left to the reader.

The terms of the form $\mathbf{intro}_{\mu y.M}\, N$ will be interpreted as the result of applying the (iso)morphism between the interpretations of $[\mu y.M/y]M$ and $\mu y.M$ to the interpretation of N. That is, seen as an ω-**Set** morphism $\mathbf{intro}_{\mu y.M}$ is realized by the (number of) the identity on ω.

Remark. Why is the induction variable y not allowed to occur in A in a term $\sum x:A.B$ when B depends on $x:A$?

Consider $\Gamma, y:Type_i \vdash M:Type_i$, where $M \equiv \sum x:A.B$, y occurs in A, and B depends on $x:A$. Let the function $\phi_{\Gamma y M,\gamma}$ be defined analogously to the above definition. We have

$$|\Phi_{\Gamma y M,\gamma}(y)| = \bigcup_{a \in |A(\gamma, y)|} \{a\} \times |B(\gamma, a, y)|.$$

Let ϕ^i denote the ith iteration of $\phi_{\Gamma y M,\gamma}$. According to the above equation we have

$$\phi^{\lambda+1} = \bigcup_{a \in |A(\gamma, \phi^\lambda)|} \{a\} \times |B(\gamma, a, \phi^\lambda)|.$$

The ω-set ϕ^λ consists of the predecessors of the b's in $B(\gamma, a, \phi^\lambda)$. However, if a_1 is new on level λ for some λ, that is, $a_1 \in |A(\gamma, \phi^\lambda)|$ and $a_1 \notin |A(\gamma, \phi^\kappa)|$ for all $k < \lambda$, then the corresponding b's do not have any predecessors. Hence the set $B(\gamma, a_1, \phi^\lambda)$ is to big and does not have any meaning for the calculus. Moreover, the function $\phi_{\Gamma y M,\gamma}$ does not have any fixed points.

It is possible to construct a fixed point for a variant of $\phi_{\Gamma y M,\gamma}$ and ϕ^i by extending the equation above with the following:

$$\phi^{\lambda+1} = \bigcup_{a \in |A(\gamma,\phi_\lambda)|} \begin{cases} \{a\} \times |B(\gamma, a, \varnothing)| & \text{if } a \text{ is new on level } \lambda+1 \\ \{a\} \times |B(\gamma, a, \phi^\lambda)| & \text{otherwise.} \end{cases}$$

But it is an open question whether there exists a meaningful logic corresponding to a calculus with such inductive types (see also Normann, 1989).

5. Conclusion

In this paper I have showed how the Extended Calculus of Construction in a consistent way can be extended with inductively defined types. These types are defined as initial algebras, and in this way they correspond to the data types in a language ML. The recursion, defined in connection with the inductive types, is a purely structural induction. That is, the value of a recursive function must depend directly on the closest predecessor in the hierarchy. This fact imposes some limitations on expressibility. For example, the Euclidean algorithm for the computation of the greatest common divisor can be defined in an elegant way as a recursive function where the value for a given pair does not necessarily depend on the predecessor of one of the components. The leap can be much longer. It is possible to define this function in the formalism described in this paper. But some encoding has to be involved. This is, however, a problem for all type systems defined in this way, e.g., the Guttag systems (Guttag, 1975).

The inductive type system in Per Martin-Løf's type theory (see Normann, 1989) seems not to have deficit and it would be interesting to study a combination of this with ECC.

Acknowledgments

I would thank Rod Burstall, Eugenio Moggi, and especially Zhaohui Luo for many fruitful discussions. I also thank Lill Kristiansen and Dag Normann for their interest and for many interesting discussions.

Received January 19, 1990; Final manuscript received September 18, 1990

References

Cardelli, L. (1989), The Quest language and system, tracking draft.

Constable, R. L., and Mendler, N. P. (1985), "Recursive Definitions in Type Theory," Lecture Notes in Computer Science, Vol. 193, Springer-Verlag, Berlin/New York.

COQUAND, TH., (1986a), "A Calculus of Construction," Paris.

COQUAND, TH. (1986b), An analysis of Girard's paradox, *in* "Proceedings of the First IEEE Symposium on Logic in Computer Science."

COQUAND, TH. (1989), unpublished notes from a seminar, Paris.

COQUAND, TH., AND HUET, G. (1988), The calculus of constructions, *Inform. and Comput.* **76**, 95–120.

COQUAND, TH., AND PAULIN, C. (1989), Inductively defined types, *in* "Proceedings of the Workshop on Programming, Båstad."

DEVLIN, K. (1979), "Fundamentals of Contemporary Set Theory," Springer-Verlag, Berlin/New York.

GIRARD, J.-Y. (1972), "Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur," Thèse, Université Paris VII.

GIRARD, J.-Y. (1986), The system *F* of variable types, fifteen years later, *Theoret. Comput Sci.* **45**.

GUTTAG, J. V. (1975), "The Specification and Application to Programming of Abstract Data Types," Ph.D. Thesis, University of Toronto.

HAGINO, T. (1987), "A Categorical Programming Language," CST-47-87 (Ph.D. Thesis), University of Edinburgh.

HARPER R., MILNER, R., AND TOFTE, M. (1989), "The Definition of Standard ML, version 3," LFCS Report Series, ECS-LFCS-89-81, University of Edinburgh.

HYLAND, M. (1987), A small complete category, to appear in *Ann. Pure Appl. Logic.*

HYLAND, M. AND PITTS, A. (1987), The theory of constructions: Categorical semantics and topos-theoretic models, *in* "Categories in Computer Science and Logic, Boulder."

LEVY, A. (1979), "Basis Set Theory," Springer-Verlag, Berlin/New York.

LONGO, G., AND MOGGI, E. (1988), "Constructuve Natural Deduction and Its "Modest' Interpretation," Report CMU-CS-88-131, Computer Science Department, Carnegie Mellon University.

LUO ZHAOHUI (1989a), ECC, and extended calculus of constructions, *in* "Proceedings of the Fourth IEEE Symposium on Logic in Computer Science."

LUO, ZHAOHUI (1989), A higher-order calculus and theory abstraction, to appear in *Inform. and Comput.*

MENDLER, N. P. (1987), Recursive types and type constraints in Second-order lambda calculus, *in* "Proceedings of the Second IEEE Symposium on Logic in Computer Science."

NORMANN, D. (1989), Inductively and recursively defined types, unpublished manuscript, Department of Mathematics, University of Oslo.

NORDSTRÖM, B., PETERSSON, K., AND SMITH, J. M. (1990), "Programming in Martin-Løf's Type Theory, an Introduction," International Series of Monographs on Computer Science, Vol. 7, Oxford Science Publications.

POLLACK, R. (1988), The theory of LEGO, unpublished, draft, Edinburgh.

REYNOLDS, J. C. (1974), "Towards a Theory of Type Structure," Lecture Notes in Computer Science, Vol. 19, Springer-Verlag, Berlin/New York.