Parametric and Type-Dependent Polymorphism

Giuseppe LONGO
LIENS (CNRS) et DMI, Ecole Normale Supérieure
45, Rue d'Ulm, 75005 Paris; longo@dmi.ens.fr

Content:

1. Polymorphic λ-calculus. Introduction.

In typed languages one can formally describe functions at any finite type, namely functions that take, as input, functions, functionals and so on. In general, higher *types* are obtained by type constructors, such as "$\rightarrow$", "$\times$" or "+". Higher *order* systems allow *quantification* over type variables, thus a higher order language must include variables both for individuals and types. The use of type variables increases in an essential way the expressiveness of the language: this is clear from logic and from computations (see [GLT89]). We focus here on computations, in particular on the behaviour of terms inhabiting the higher order types; indeed, the analysis of parametricity is the investigation on how "polymorphic" terms, which may take types as inputs, actually compute. As the core problem of parametricity and type-dependence resides in Girard's second order system F, we recall first its basic notions.

The expressions of the language, i.e. terms and types (or formulae), may be defined by using the expression Tp, the intended collection of all types. We write capital letters to stress that an expression is a type. A : Tp is short for "A is a type", a : A summarizes that a has type A and that A is a type. The base is given by a collection of variables x, y, z.... X, Y, Z... and of atomic types or predicates.

We maintain the usual conventions on variable binders (λ and ∀), free variables and substitution. A definition, by combined induction, of term and type expressions is first given.

Term expressions: a := var | a(a) | a(A) | λ(var:A)a | λ(Var:Tp)a
Type expressions: A := Var | Atomic | A → A | ∀(Var:Tp)A

Equality "=" is a congruence relation on terms, whose basic axioms may be given jointly with the following rules for assigning types to "legal" terms.

The rules below are given in an intended environment, named E, E'..., when explicitely mentioned, which assigns types to term variables.

Type and Term Formation and Assignment Rules. Axioms:

$$(\to I)\qquad \frac{[x:A]\ \vdash\ b:B}{\lambda(x{:}A)b : A\to B}$$

$$(\to E)\qquad \frac{f:A\to B \qquad a:\ A}{f(a):B}$$

$$(\forall I)\qquad \frac{[X:Tp]\ \vdash\ b:B}{\lambda(X:Tp)b : \forall(X:Tp)B}\qquad \text{for } X \text{ not free in the type of a free term variable}$$

$$(\forall E)\qquad \frac{f:\forall(X:Tp)B \qquad A:Tp}{f(A):[A/X]B}$$

$(\to\beta)$ $(\lambda(x{:}A)b)(a) = [a/x]b$; $\qquad$ $(\to\eta)$ $\lambda(x:A)f(x) = f$ $\quad$ x not free in f

$(\forall\beta)$ $(\lambda(X:Tp)b)(A) = [A/X]b$; $\qquad$ $(\forall\eta)$ $\lambda(X:Tp)f(X) = f$ $\quad$ X not free in f

We may omit Tp in $\lambda(X:Tp)b$ and in $\forall(X:Tp)B$. The usual rules (ρ), (τ), (μ), (ν), (ξ) describe "=" as a congruence relation over well typed terms:

$(\rho)\qquad a = a$

$$(\tau)\qquad \frac{a=b \qquad b=c}{a=c}\qquad\qquad (\mu\nu)\qquad \frac{a=b \qquad a'=b'}{a\,a' = b\,b'}$$

$$(\xi)\qquad \frac{a=b}{\lambda(x{:}A)a = \lambda(x{:}A)b}\qquad\qquad (\xi_\forall)\qquad \frac{a=b}{\lambda(X)a = \lambda(X)b}$$

The fundamental difference between typed λ-calculi and logical systems of propositions is that proofs are coded by terms. These terms are the programs of core functional languages. Moreover, as will be pointed out later, by the work of Abadi, Cardelli, Curien, Hasegawa and Plotkin (and others, see later for references), we also know that the logical expressiveness of types is affected by the existence of proof-terms.

In the higher order case, beginning with second order, the situation is both rich, in expressiveness, and challenging, as type variables may appear in terms. In particular, terms may be fed with input types, as expressed by the rules $(\forall I)$ and $(\forall E)$.

Upon first examination, functions which take types as inputs should behave rather freely, subject to the "obvious" restrictions of definability within our paradigmatic language, system F. This system may seem to deal with input terms and types in the same way (just compare the "$\rightarrow\beta,\eta$" and "$\forall\beta,\eta$" rules). However, this is not so: second order functions cannot act on types "as if they were ordinary inputs" and the restrictions are far from obvious. These restrictions have been collected under the, a priori, rather meaningless name of "parametricity".

The interest in parametricity has been growing recently; however, we must acknowledge that both Girard [Gir71] and Reynolds [Rey83] set the early grounds for the investigation of how terms may depend on types. The directions proposed by these two authors were essentially different, as we will try to explain, and the need for a better understanding of the relations between the various approaches, possibly by a unified categorical environment, is the main motivation of this paper. With the aim of a conceptual clarification, we will classify parametricity in three main classes: relational (§.3), effective (§.4) and uniform (§.5).

Yet another direction, which goes beyond "parametricity" and motivated by Object Oriented Programming, has been suggested in [CGL93]. The required expressiveness, though, takes us away from "pure" λ–calculi, as it will be mentioned in §.6.

In all the approaches mentioned above, the analogy "types as propositions" goes together with the understanding of "types as objects" of categories. Indeed, this "many-ways" connection, Type Theory - λ-calculus - Category Theory, is at the core of most recent advances in the broad area of the Theory of Functional Programming Languages, which largely uses tools from each of these disciplines. Its applications range from the design of Edinburgh ML and its dialects, to the current trends in explicitly polymorphic languages and in functional approaches to Object Oriented Programming. We will survey the current treatments of parametricity, as a relevant problem in the area, and focus on the many issues it raises, in a categorical, but elementary, perspective. As already mentioned, in this theoretical area, Proof Theory and Category Theory are strictly linked: proof-theoretic investigations of languages correspond to categorical developments in their semantics. Thus, the syntactic presentation in each of sections 3, 4, 5 and 6, will end with some open problems for the category-theory oriented reader.

2. Types as invariants and types as sets.

The seminal paper [Rey83] begins with a fable. Professors Descartes and Bessel present, in different but parallel classes, the Theory of Complex Numbers by two different systems of notation: the cartesian and the polar one. The fable continues by stressing that, in spite of the notational difference, the two courses could be entirely interchanged. The moral is that what really matters in Mathematics, namely notions and theorems, do not depend on coding and notations. Indeed, we all know that the relevant results of Number Theory, say, do not depend on the "decimal" notation... similarly for the Theory of Complex Numbers. The point is that our abstract understanding of Mathematics is *invariant* w.r.t. specific notations and structures, even though it must be necessarily given in a specific notation or in an (intended) structure. In general, invariants are, technically and methodologically, at the core of Mathematics and Physics.

In Computer Science, this has been mostly developed within the algebraic approach to data types, where abstract data types have been widely studied: the type "Complex" denotes an abstraction that can be realized by a variety of sets or represented by several denotations. The current algebraic theory of Abstract Data Types, though, as Reynolds stresses, is not perfectly suited for higher type or higher order systems and, thus, he proposes a "relational" treatment of invariance: computations do not depend on types in the sense that they are "invariant" w.r.t. arbitrary relations on types and between types. Reynolds's approach set the basis for most of the current work on parametricity, as we will review below (§.3).

Some twelve years earlier, Girard had given just a simple hint towards another understanding of the properties of "computing with types". In [Gir71], it is shown, as a side remark, that, given a type A, if one defines a term J_A such that, for any type B, $J_A B$ reduces to 1, if $A = B$, and reduces to 0, if $A \neq B$, then $F + J_A$ does not normalize. In particular, then, J_A is not definable in F. This remark on how terms may depend on types is inspired by a view of types which is quite different from Reynolds's. System F was born as the theory of proofs of second order intuitionistic propositional calculus. Its main logical application has been the normalization theorem, that is, the extension of cut-elimination to second order intuitionistic logic. As we all know, the "semantic convention" of second order systems is that second order variables must be interpreted as "sets" of the intended interpretation. More precisely, suppose that a model is given where first order variables are interpreted as "individuals" of a specific (structured) set or algebraic domain. Then, second order variables are intended to range over *subsets* or *substructures* of the given domain. In system F, second order variables are type variables: the types-as-propositions analogy suggests that second order (type) variables have the intended meaning of (possibly) infinite sets or structures, which may come, as in Set Theory, with their own coding and structure or "implementation". Girard's understanding (and ours, by the Genericity Theorem in [LMS93], see §.4 below) is that intuitionistically sound functions should not be able to compute with (possibly) infinite inputs, as types have the intended meaning of possibly infinite collections of individuals. Thus, the non definability of J_A formalizes an issue which is, a priori, unrelated to invariance properties as suggested by abstract data types. Indeed, the present view of types corresponds to the meaning given to logical or computational theories by Scott's denotational semantics: types are "predicates or other entities denoting specific subsets of some universe of values", as acknowledged in [Rey83]. In particular, then, a type denotes a specific (sub-)structure of the intended domain of interpretation, in contrast to the abstract or invariance based understanding of types followed by Reynolds. Again, this more closely follows the main stream logical treatment of types as propositions. Of course, one may violate the second order semantic convention (in [CGL93] it is done so and for good reasons, see §.6), but then the connection to logic or the logical significance of the system is lost.

There is no methodological priority of either view. Abstract data types are commonly used in Computer Science, but the logical paradigm of "types-as-propositions" has also been quite successful. Fortunately, we already know of several relevant connections: from the logical representability of abstract data types by second order existential quantification, [MP86,Hase93], to the representation of initial algebras, [BB85, ACC93, Hase93]. However, the two different views above suggested different technical approaches and results, as well as many open problems. [Hase93], [ACC93], [PA93] and [LMS93] are some of the most recent advances in the two directions.

3. Relational (or invariance) Parametricity.

We owe to Strachey the distinction between "parametric" and "ad hoc" polymorphism, according to how polymorphic functions depend on their type parameters. A seminal, enlightening classification, but labelled by unfortunate names: unfortunate because all functions depend on their parameters and, thus, the "uniformity" of this dependence, in the case of second order parameters, is not expressed by the name "parametric". Moreover, "ad hoc" polymorphism may be dealt with by non "ad hoc" treatments of relevant constructs (see [CGL93] and §.7). However, the names of these two classes of polymorphism are now part of folklore and can hardly be changed.

Following Reynolds's approach, the core of parametricity is the formalization of the notion of invariance mentioned above by means of "relations". The idea is that the independence of computations w.r.t. (the internal structure of) types is expressed by their invariance w.r.t. arbitrary relations on types. The pivotal result is the "abstraction theorem", whose main consequence is that a polymorphic function takes related input types to related output values. This approach has been given a categorical presentation in [MaRey92] and further developed, semantically, in [Hase93]. We briefly survey the syntactic presentation in [ACC93].

The idea is that one has relations between types and relations between terms; thus, a theory R of relations is proposed. Relations may be given between terms of different types and they may be extended at higher types and higher order. The environments and judgements of R extend those of F. Environments E set the type of free variables and assert that certain variables are types, as usual; moreover, they include assumptions about relations between types or between terms. Thus, environments contain two sorts of assumptions from F and two additional ones:

X	X is a type variable
x:A	x is a variable of type A
$\begin{matrix} X \\ \mathcal{W} \\ Y \end{matrix}$	W is a relation variable between type variables X (domain) and Y (codomain)
$\begin{matrix} x : A \\ \mathcal{R} \\ y : B \end{matrix}$	the variables x and y have types A and B, respectively, and are related by R

The judgments generalize those of system F:

$\vdash$ E	E is a legal environment

$$E\ ''\ \begin{array}{c} A \\ \mathcal{R} \\ B \end{array}$$ R is a relation between types A and B in E

$$E\ ''\ \begin{array}{c} a\ :\ A \\ \mathcal{R} \\ b\ :\ B \end{array}$$ R relates a of type A and b of type B in E.

Note that equality, as a relation between terms in a type, is a definable relation in this approach. Indeed, a type A can be identified with the diagonal relation in $A\times A$, and one may consider

$$(=)\quad E\ ''\ \begin{array}{c} b\ :\ A \\ A \\ c\ :\ A \end{array} \qquad \text{corresponding to the F judgment} \quad E\ ''\ b = c\ :\ A\,.$$

The **introduction** and **elimination** rules for $\to$ are, respectively:

$$(\to\mathrm{IR})\quad \frac{E,\ \begin{array}{c} x\ :\ A \\ \mathcal{R} \\ x'\ :\ A' \end{array}\ ''\ \begin{array}{c} b\ :\ B \\ \mathcal{S} \\ b'\ :\ B' \end{array} \qquad E\ ''\ \begin{array}{c} B \\ \mathcal{S} \\ B' \end{array} \qquad \begin{array}{cc} x & b' \\ x' & b \end{array}}{E\ ''\ \begin{array}{c} \lambda(x:A)b\ :\ A\to B \\ \mathcal{R}\to\mathcal{S} \\ \lambda(x':A')b'\ :\ A'\to B' \end{array}}$$

$$(\to\mathrm{ER})\quad \frac{E\ ''\ \begin{array}{c} b\ :\ A\to B \\ \mathcal{R}\to\mathcal{S} \\ b'\ :\ A'\to B' \end{array} \qquad E\ ''\ \begin{array}{c} a\ :\ A \\ \mathcal{R} \\ a'\ :\ A' \end{array}}{E\ ''\ \begin{array}{c} b(a)\ :\ B \\ \mathcal{S} \\ b'(a')\ :\ B' \end{array}}$$

The **introduction** and **elimination** rules for $\forall$ are:

$$(\forall\mathrm{IR})\quad \frac{E,\ \begin{array}{c} X \\ \mathcal{W} \\ X' \end{array}\ ''\ \begin{array}{c} b\ :\ B \\ \mathcal{S} \\ b'\ :\ B' \end{array} \qquad \begin{array}{cc} X & b', B', \mathcal{S} \\ X' & b, B, \mathcal{S} \end{array}}{E\ ''\ \begin{array}{c} \lambda(X)b\ :\ \ (X)B \\ (\mathcal{W})\mathcal{S} \\ \lambda(X')b'\ :\ \ (X')B' \end{array}} \qquad (\forall\mathrm{ER})\quad \frac{E\ ''\ \begin{array}{c} b\ :\ \ (X)B \\ (\mathcal{W})\mathcal{S} \\ b'\ :\ \ (X')B' \end{array} \qquad E\ ''\ \begin{array}{c} C \\ \mathcal{T} \\ C' \end{array}}{E\ ''\ \begin{array}{c} b(C)\ :\ B\{X\leftarrow C\} \\ \mathcal{S}\{\mathcal{W}\leftarrow\mathcal{T}\} \\ b'(C')\ :\ B'\{X'\leftarrow C'\} \end{array}}$$

Where ($\forall$IR) applies provided that the type and relation variables X, W, X' are not free in the type of any free variable.

These rules generalize the corresponding rules of F: one has to take identity relations in order to recover the rules in §.1. The connecting assumption is given by the "identity extension", which implies that, if a : A and a' : A', then

$$a = a' \text{ and } A = A' \text{ iff } a \text{ and } a' \text{ are related by } A$$

(and A is indeed the identity relation on itself); moreover, variable substitutions preserve identities. This is a very delicate point, in particular when A may contain type or relation variables and it has been clarified by the evolution of the system in [ACC93]. In short, and in the system of [ACC93], "identity extension" amounts to say that each provable judgement E |- a : A in F, yields a provable (=) judgement, for b = c = a.

The elimination rule $(\forall\mathsf{ER})$ says that if b and b' are related functions then they take related input types to related outputs. Note that the relation $\mathsf{S}\{\mathsf{W} \leftarrow \mathsf{T}\}$, between terms b(C) and b'(C'), is constructed out of the relations, $\forall(\mathsf{W})\mathsf{S}$ between b and b', and T between types C and C', by instantiating relational variables by a relation. By the "identity extension", if b = b' and B = B', then b and b' are related by $\forall(X).B$, the identity relation on itself, and, consequently, b(C) = b'(C) in $B\{X \leftarrow C\}$. Clearly, this is exactly the usual $(\forall\mathsf{E})$ rule of system F. The elimination rule $(\forall\mathsf{ER})$ may be read as the syntactical counterpart of Reynolds's binary relational parametricity, i.e. the Abstraction Theorem. More precisely, it corresponds to the validity of the Abstraction Theorem over all definable elements of all models, while $(\forall\mathsf{ER})$ rule, jointly to the "identity extension", yields the Abstraction Theorem over all elements, not just the definable ones (see Open Problems below). As already mentioned, the Abstraction Theorem says that related polymorphic terms take related types to related output terms, by preserving identities.

Finally, functions themselves can be turned into relations; in a sense, a map b in $A \to B$ relates A and B (call <b> this relation):

$$\frac{E'' \; b : A \to B \qquad E'' \; a : A}{E'' \; \begin{array}{c} a : A \\ b \\ b(a) : B \end{array}} \qquad\qquad \frac{E'' \; \begin{array}{c} a : A \\ b \\ c : B \end{array} \qquad E'' \; b : A \to B}{E'' \; \begin{array}{c} b(a) : B \\ B \\ c : B \end{array}}$$

A consequence of the elimination rule for $\forall$, $(\forall\mathsf{ER})$, and of the identity extension is the following fact:

3.1 Fact. *If $b{:}\forall(X)B$, with $X \notin FV(B)$ (i.e. X not free in B), then $b(C) = b(C') : B$, for all types C and C'.*

The assumption corresponds to taking $B = B' = \mathsf{S}$ in $(\forall\mathsf{ER})$ and, thus, b = b', by (=). The thesis follows by (=) again, as B does not contain X free. Note that 3.1 says that any polymorphic term b, which outputs all values in the same type (since X is not free in B), is a constant function.

The reader may wonder whether the complex formalization of Reynolds's understanding of parametricity as invariance w.r.t. relations is really worth pursuing, since it takes us apparently far from the core intuition of "types as propositions" as more closely developed by the alternative, more "effective" view hinted at by Girard (see also §.4 and 5). However, the results below fully justify the relevance of relational invariance as a tool for the investigation of terms and types.

It is an old remark in Logic, going back to Russell and Gentzen, that it is possible to define, in terms of second order universal quantifiers, some other crucial constructs of Logic. In particular, one may define weak forms of conjunction (product), disjunction (coproduct), the absurdum (initial object), generalized singletons (terminal objects) and existential quantification. The weakness of the definition is due to the fact that, in categorical terms, products and coproducts are not universal, as required by the interpretation in categorical logic, as they miss the unicity of the pairing or co-pairing functions (thus they are only "weakly" universal). The same can be said of initial and terminal objects, which are weakly universal too. However, in the system R of [ACC93] the following properties are provable:

(1) $\forall(X)(B \to X) \to (B' \to X) \to X$ is a coproduct of B and B',

(2) $\forall(X)(B \to B' \to X) \to X$ is a product of B and B',

(3) $\forall(X)X$ is initial,

(4) $\forall(X)X \to X$ is terminal.

In summary, if one considers, in the "types as propositions" analogy, both proofs coded by terms *and* the properties of relational parametricity between types and terms, then these definable constructs of second order Logic possess the required universal properties. This fact adds logical (and categorical) significance to λ-calculus and to relational (or invariance) Parametricity (early work on this matter may be found in [Wad89] and [BFSS90]).

Even more is shown in [Hase93], where a model theoretic approach to parametricity is proposed. In short, Hasegawa calls "parametric" those models of system F that realize a weak version of $(\forall ER)$ for all morphisms, objects and relations in the model. Also, in that categorical frame, the definable existential quantifier

(5) $\exists(X).B \equiv \forall(Y)(\forall(X)(B \to Y) \to Y)$

satisfies the intended universal property required in categorical logic; namely, it is the left adjoint of the diagonal functor, and thus, it symmetrically corresponds to $\forall$ quantification, in all models, exactly as required in Proof Theory. Moreover, as a converse to the results in [ACC93], Hasegawa proves that the universality of each of these definable constructs *implies* the parametricity of their interpretations, as objects, in the model.

Firther relevant results should be mentioned about the universality of free algebraic types (see [ACC93] and [Hase93]). These very relevant facts encourage further investigation into the understanding of the various approaches to relational parametricity.

Open problems. 1 - Any model of the system in [ACC93] should be a model in the sense of Hasegawa; a proof though requires a precise categorical understanding of the relational system. Note, for example, that in Hasegawa's models the definable existential quantifier is universal; is it also so in the formal system in [ACC93] ? It does not need to be, as, for example, the "abstraction theorem" (rule $(\forall ER)$ in [ACC93]), say, is given for *definable* relations only. More generally, by a suitable variant of the rules, if needed, is there a two way correspondence between models of [ACC93] and Hasegawa's ? A positive answer would allow to study the equivalence between parametricity and universality of definable types at an abstract proof-theoretic level.

2 - In §.5 we will discuss **PER** models and the interpretation of $\forall$ in them. This is given by the right adjoint to the (internal) diagonal functor of **PER**, in a topos theoretic frame. It is not yet known how that interpretation is related to the meaning of $\forall$ in the parametric models above, when constructed over **PER**.

4. Effective Parametricity and the Genericity Theorem.

Girard's remark about the term J_A in §.2 suggests the extension of system F by the following simple axiom (called Axiom C, where C stands for constant). In view of the undefinability of J_A, system F cannot effectively discriminate between different types, i.e. it cannot output "essentially" different values of the same type, such as 0 and 1, according to the input types. We may assume then that a polymorphic term, which outputs all values in the same type, is a constant function:

Axiom C *If* $a : \forall(X)A$, *with* $X \notin FV(A)$, *then* $a(B) = a(C): A$, *for all types* B *and* C.

We already observed that Axiom C is a *theorem* in the relational or invariance approach to parametricity (Fact 3.1). It holds then in all models of system R of §.3 as well as in all parametric models, in the sense of Hasegawa. Indeed, Axiom C is a very weak property, which may be directly assumed, independently of the theory of relations above, and just on the grounds of our effective, more than invariant, understanding of second order computations, as discussed in §.2. Thus, Axiom C is a common property both of invariance and effective parametricity (see §.5 for its validity in **PER** models).

Call Fc the extension of system F with Axiom C. With no other assumption, in [LMS93] the following is proved:

4.1 Theorem (Genericity)

Let $a, b : \forall(X)A$. *If, for some type* C, $a(C) =_{Fc} b(C)$, *then* $a =_{Fc} b$.

Note that the terms a(C), a(B),.... in general may live in different types, namely [C/X]A, [B/X]A..., as no restriction is made on A, in particular X may occur in it. The meaning (and the strength) of the theorem should be clear: if two polymorphic functions of the same type agree on *one* input, then they agree on all inputs. In other words, any type is generic. In our understanding, it says that we cannot use the possibly infinite information carried by a type, as predicate or (structured) set. Computations deal with types as "black holes": if two terms act the same on a given black hole, they will act accordingly on all other black holes. This suggests the "effective nature" of this approach to parametricity.

There is an analogy between the Genericity Theorem and a lemma, also called "genericity" in Barendregt's book [Bar84]. That result deals with type-free λ-calculus, thus its statement and the proof (a rather simple one) are formally unrelated to ours, but its similar "spirit" may help to understand the Genericity theorem. Lemma 14.3.24 in [Bar84] says that, given an unsolvable term Ω, if a(Ω) has a normal form, then a(c) = a(Ω) for all terms c. The reason is that one cannot effectively "look inside" the infinite information contained in a diverging, unsolvable, term. An immediate consequence is that, if a(Ω) = b(Ω) and possess a normal form, then a(x) = b(x) = b(Ω), and thus a = b, by extensionality. In other words, unsolvable terms, such as Ω, are generic. Theorem 4.1 above says that, in second order polymorphic λ-calculus, any type is generic.

The proof of the Genericity theorem is far from trivial (see [LMS93]) and suggests some general considerations (and these considerations may encourage the scholar to read the proof in

[LMS93] closely). A classification of (extra) inference rules in formal systems can be given as follows. Let G be a formal system and R and S be well-formed formulae in the language of G. Then, for the derivation R |- S, three basic cases are possible:

- R |- S is **compatible**, that is, G + (R |- S) is consistent;
- R |- S is **admissible**, that is, if G |- R, then G |- S;
- R |- S is **derivable**, that is, any model of G + R is also a model of G + S.

Compatibility should be clear: just one model of G is needed, which realizes S when R is realized. Admissibility means that from the *proof* of R in G, one can construct a proof of S in G. Derivability, in proof-theoretic terms, corresponds to the provability of the *implication* R $\Rightarrow$ S, within G; that is, to the derivability of S in G just under assumption R (formally, (G + R) |- S, with no need of a proof of R in G). Clearly, derivability implies admissibility, which implies compatibility.

Before getting into the categorical issues that are raised here, let's better understand this distinction by an example from the type-free λ-calculus. Consider Curry's Combinatory Logic (CL, see [Bar84; HS86]) and the rule (ζ_β) of "functional extensionality":

(a(x) = b(x)) |- (a = b), for a and b "functional",

that is, of the following shape: S, K, Sc, Kc, Scd. Church's type-free λ-calculus ($\lambda\beta\eta$) is a syntactic model of CL + (ζ_β), by the obvious translation of S and K into λ-terms (for the basic notions and facts in this example, see [HS86, ch.8-9]). Thus, (ζ_β) is compatible with CL. It is not admissible, though, as five extra axioms are required in order to derive a = b from a *derivation* of a(x) = b(x) (these equational axioms are due to Curry and each fill a line with S's and K's, [HS86, ch.9]). In other words, call CLβ the extension of CL by Curry's five axioms, then one can prove CLβ |- (a = b) from a derivation CLβ |- (a(x) = b(x)). Thus (ζ_β) is admissible for CLβ. However, it is not derivable in CLβ, since there exist models of CLβ which may realize the assumption a(x) = b(x), but not the consequence a = b (see [HL80], where these distinctions were first made, from the point of view of models: the experienced reader may take as a model the interpretations of closed terms in Scott's Pω model). Finally, as said above, $\lambda\beta\eta$ is an "extension" of CL where (ζ_β) is just an instance of the rule (ζ) of extensionality. In summary:

- (ζ_β) is compatible with CL, but not admissible;
- (ζ_β) is admissible for CLβ, but not derivable;
- (ζ_β) is derivable in $\lambda\beta\eta$.

In [LM90] a categorical classification is given of the models of CL, CLβ, $\lambda\beta\eta$ (see also [AL91]). However, in spite of the clear categorical notions that characterize the three classes of models, a surprising "historical" remark can still be made: an example of a "proper" mathematical model of Combinatory Logic, CL, has been given only recently [DiGiHon93], much later than the models of $\lambda\beta(\eta)$. That is, a syntax-independent model, different from the term model (and from the interpretations of closed terms), which interprets CL and which is not a model of $\lambda\beta$. In conclusion, even in type-free λ-calculi, the proof-theoretic and categorical status of rules is a complex one.

We are in a similar, but surely more complex situation here. Consider Genericity as the "rule" ((exists C, a(C) = b(C)) |- (a = b)), where a, b : $\forall$(X)A. Call (**Gen**) this rule. Then the following holds:

4.2 Corollary.

- (Gen) is compatible with F, but not admissible;

• (Gen) is admissible for Fc, but not derivable.

To prove this, recall that there are plenty of models of Fc (see also §.5), thus Fc is thus consistent. By the Genericity theorem (4.1), the term model of Fc proves the compatibility of this rule with F, since the term model of Fc is a model of F such that, when $a(C) = b(C)$ is true (i.e., it is derivable), then $a = b$ is true (derivable). (However, not any model of Fc needs to interpret Fc + (Gen) ! See below.) (gen) is not admissible in F by trivial counterexample: take $a \equiv \lambda(X)x(X)$ and $b \equiv \lambda(X)x(C)$, for $x : \forall(X)B$ with X not free in B, then $a(C) = b(C)$ in F, but $a = b$ is not derivable in F.

Admissibility for Fc is also given by theorem 4.1. As a matter of fact, its proof is based on a proof-theoretic analysis of the derivation Fc |- (exists C, $a(C) = b(C)$), which shows that, indeed, Fc |- ($a = b$). As for non-derivability, models of Fc may be given that realize $a(C) = b(C)$ but not $a = b$. Instead of giving a specific counterexample, by the Fact below we make a more general remark: we observe that no model of Genericity, as an implication, can be a model of system R. Thus, any model of system R is a model of Fc, but not of Fc + (Gen). Therefore relational and "effective" parametricity are two "orthogonal" approaches to parametricity.

As mentioned in the previous section, all categorical models of the relational system R or any parametric model in the sense of Hasegawa, contain initial and terminal objects. As a matter of fact, $\forall(X)X$ and $\forall(X)(X\to X)$ denote these objects, respectively. Consider now $K \equiv \lambda(X)\lambda(x:X)\lambda(y:X)x$ and $O \equiv \lambda(X)\lambda(x:X)\lambda(y:X)y$ of type $\forall(X)(X\to(X\to X))$. Then

$K(\forall(X)(X\to X)), O(\forall(X)(X\to X)) : (\forall(X)(X\to X)) \to ((\forall(X)(X\to X)) \to (\forall(X)(X\to X)))$.

which is isomorphic to $(\forall(X)(X\to X))\times(\forall(X)(X\to X) \to (\forall(X)(X\to X))$. Since $\forall(X)(X\to X)$ is terminal, then the interpretations of $K(\forall(X)(X\to X))$ and of $O(\forall(X)(X\to X))$ coincide. However, K and O, the polymorphic first and second projections (they project any $X\times X$ to X, for any X), differ in the model. The same could be shown by using type $\forall(X)X$. In conclusion, since any model of R realizes Axiom C, one has:

4.3 Fact. *Any parametric model is a model of Fc but is not a model of*

Fc + ((exists C, a(C) = b(C)) ⇒ (a = b)).

This fact proves that the understanding of parametricity as "effective" parametricity is incompatible with parametricity as invariance, even though they both yield Axiom C.

More should be said about models. We have already tried to convince the reader that the "types as propositions", "types as sets" analogies strongly suggest that Genericity is a crucial effectivity property, in the spirit of Intuitionistic Logic. Thus, in the perspective of the Genericity Theorem, we claim that there is not yet a fully satisfactory categorical understanding of the *proof-theoretic* constructivity expressed by system F. In other words, system F is the theory of terms as proofs of the second order Intuitionistic Propositional Calculus, and, in spite of the many, very relevant connections we know of between Intuitionistic Proof Theory and Categories, a close analysis of its deductive power leads to a result which has no categorical counterpart, up to now. Indeed, not only is it not the case that any model of Fc + (exists C, $a(C) = b(C)$) is a model of Fc + ($a = b$), but we have no categorical model of Fc + ((exists C, $a(C) = b(C)$) ⇒ ($a = b$)), except, of course, the term model of Fc (and uninteresting models obtained by erasing all type information.)

In a sense, this shows that all known models have "too many morphisms" in the objects which interpret universally quantified types. Thus, even in the apparently truly constructive frames provided by the **HEO** or **PER** model interpretation (see the Effective Topos, in the next section) or in Coherent Domains, the interpretation of second order quantification seems to contain some morphisms which act on types in some "ad hoc" way and violate the intended impossibility to work with infinite information, that is, a type, as input. However, this explanation is still unsatisfactory: as a matter of fact, the counterexamples to genericity are given by the peculiarity of initial and terminal objects. Their existence doesn't need to be essential in the categorical interpretation of constructivity, as just relational parametricity forces them in (recall that, a priori, models of system F only need to contain weakly initial and terminal objects). An interesting project would be the construction of relevant categories that model the effectiveness of Intuitionistic Logic, up to a sound interpretation of Genericity and with no forced interpretation of invariance parametricity, a concept which is orthogonal to the "types as propositions" analogy, as we tried to point out.

Open problems: 1 - Give a categorical interpretation of impredicative Type Theory, which suggests a "non-syntactic" model and/or a general categorical meaning for

$$Fc + ((\text{exists } C,\ a(C) = b(C)) \Rightarrow (a = b)).$$

2 - Construct, at least, some (categorical) models that contain a collection of "generic" types (objects). That is, a model $\mathcal{M}$ where for a collection $\mathcal{A}$ of objects (types), one has, for $C \in \mathcal{A}$ $\mathcal{M} \models (a(C) = b(C)) \Rightarrow \mathcal{M} \models (a = b)$. If our intuition about constructivity is correct, infinite objects in categories of (effective) sets should satisfy this property.

5. Parametricity and the Uniformity Principle in the Effective Topos.

The model of the Partial Equivalence Relations has already been mentioned in the previous sections. It was invented by Girard and Troelstra, on the grounds of Kreisel's work on the **HEO** (see [LM84], for references and a connecting theory of higher type functionals). The model had an essentially syntactic flavour. Via more recent ideas of Hyland and Moggi, though, it has acquired an independent interest as a "small complete" category, within Hyland's Effective Topos ([Hyl82,Hyl87]). We will use this model here to discuss a fundamental aspect of parametricity: the so-called "uniformity" of the definition of polymorphic terms.

The objects of the category **PER** below are equivalence relations on subsets of the natural numbers or **partial equivalence relations** (p.e.r.'s). Morphisms are defined by Kleene's application: $n \cdot p$ is the result of the application of the n-th partial recursive function to the number p. By $n \cdot p \in A$ we always mean that $n \cdot p$ is defined. $<\,,\,>$ with inverses pr_1, pr_2 is any (effective) and bijective coding of pairs.

(***Notation:*** Let A be a symmetric and transitive relation on ω. Set then:

n A m iff n is related to m by A , **dom(A)** = {n | n A n },

$\mathbf{\{n\}_A}$ = {m | m A n} the equivalence class of n w.r.t. A, **Q(A)** = $\{\{n\}_A \mid n \in \mathrm{dom}(A)\}$)

5.1 Definition: *The category **PER** has as*

objects: *$A \in$ **PER** iff A is a symmetric and transitive relation on ω,*
morphisms : *$f \in$ **PER** $[A,B]$ iff*
$$f: Q(A) \to Q(B) \text{ and } \exists n \ \forall p \ (pAp \Rightarrow f(\{p\}_A) = \{n \cdot p\}_B).$$

Morphisms in **PER** are "computable" in the sense that they are fully described by partial recursive functions which are total on the domain of the source relation.

PER models will give a clear understanding of the following "uniformity" of polymorphic terms. Write a[X] and A[X] in order to stress that X may occur in a and A. Then the rule

$$(5.2) \qquad \frac{a[X] : A[X]}{\lambda(X)a[X] : \forall(X)A[X]}$$

gives a *uniform definition* of the family of maps $\{a[X]\}_X$ with components a[X], as this definining rule is "uniform" in the parameter X. In other words, the *computation* a[X] "does not depend" on the type X, or, more precisely, depends uniformly on it. This will be explained by reference to the validity, in **PER** models, of the so called "Uniformity Principle" (the contrapositive of König's Lemma).

Instead of getting into the complex definition of Hyland's topos **Eff**, we will develop our remarks in a category which sits "half way" between **Eff** and **PER**, the ω–**Set**, following [LM91]. ω–**Set** is a full subcategory of **Eff**, with sufficient closure properties to provide an expressive frame category. We will recall that **PER** is a subcategory and an internal category of ω–**Set**. Notice also that these constructions, **Eff**, ω–**Set** and **PER**, can be built out of any (partial) combinatory algebra or (partial) model of Combinatory Logic or λ-calculus, even a very concrete one, with no reference to the syntax of λ-calculus, such as Scott's Pω model, say, or any reflexive object in a Cartesian Closed Category, [AL91].

5.3 Definition: *The category **ω– Set** has as*
objects: *$(\mathcal{A}, \Vdash) \in \omega$– **Set** iff*
$\mathcal{A}$ is a set and $\Vdash \subseteq \omega\times \mathcal{A}$, i.e. $\Vdash$ is a relation in $\omega\times \mathcal{A}$, s.t. $\forall A \in \mathcal{A} \ \exists n \ (n,A)\in \Vdash$
(write $n \Vdash A$).
morphisms : *$f \in \omega$– **Set** $[\mathcal{A}, \mathcal{B}]$ iff*
$f : \mathcal{A} \to \mathcal{B}$ and $\exists n \ \forall A \in \mathcal{A} \ \forall p \Vdash_{\mathcal{A}} A, \ n\cdot p \Vdash_{\mathcal{B}} f(A)$
$n \Vdash_{\mathcal{A} \to \mathcal{B}} f$).

Similarly as in **PER**, each morphism in ω–**Set** is "computed" by a partial recursive function, which is total on $\{ p \mid p \Vdash_{\mathcal{A}} A \}$, for each $A\in \mathcal{A}$
(**Notation**: we say that "p realizes A" iff $p \Vdash_{\mathcal{A}} A$ in, $(\mathcal{A})$; in $\Vdash_{\mathcal{A}}$ we may omit $\mathcal{A}$ if there is no ambiguity).

The category **PER** is isomorphic to a full subcategory of **ω-Set**. In fact, for every partial equivalence relation (p.e.r.) A, we can define an ω-set $In(A) = (Q(A), \in_A)$, where Q(A) are the equivalence classes of A, as disjoint subsets of ω, and $\in_A$ is the usual membership relation restricted to ω×Q(A). Clearly, this defines a realizability relation in the sense of 5.3

and the embedding is full. Call $\mathcal{M}$ the image of **PER** in ω-**Set** via *In*.

The relevant point though is that **PER** is also an *object* of ω–**Set**. Just set

$$\mathbf{M_0} = (\mathbf{PER}, \Vdash_{M}) \in \omega\text{–}\mathbf{Set} \quad \text{where} \quad \Vdash_{M} = \omega \times \mathbf{PER}.$$

(If there is no ambiguity, we call **PER** also the set of objects of the category **PER**).

Indeed, **PER** is an *internal category* $\mathbf{M'} = (\mathbf{M_0},\mathbf{M_1})$ of ω-**Set**, whose "object of objects" is $\mathbf{M_0}$ (for the object of morphisms $\mathbf{M_1}$ and further details see [LM91, AL91]). As a matter of fact, ω-**Set** has all finite limits and equalizers. It has then enough structure as to allow a relevant theory of internal categories. The product below, or the intersection of p.e.r.'s, is the object component of an internal product functor (see [AL91], also for references to the many authors who developed this matter). Notice that the embedding of the set **PER** as $(\mathbf{PER},\Vdash_{M})$ into ω–**Set** is just the canonical embedding of **Set** into **Eff**, in [Hyl82]. For our purposes, this embedding corresponds to the intuition that nothing can be said "effectively" when taking as inputs possibly infinite p.e.r.'s; indeed, since $\Vdash_{M} = \omega \times \mathbf{PER}$, realizers cannot help in distinguishing among p.e.r.'s.

Without going into internal adjunctions, the product in ω-**Set**, indexed over any ω-set, can be elementarily given as follows:

5.4 Definition: *Let* $(\mathcal{A}, \Vdash_{\mathcal{A}}) \in \omega\text{–}\mathbf{Set}$ *and* $g: \mathcal{A} \to \omega\text{-}\mathbf{Set}$. *Define then ω–set*
$([\Pi_{A \in \mathcal{A}}\ g(A)], \Vdash_{\Pi,g})$ *by*
1) for $f \in \Pi_{A \in \mathcal{A}}\ g(A)$, $n \Vdash_{\Pi,g} f$ *iff* $\forall A \in \mathcal{A}$ $\forall p \Vdash_{\mathcal{A}} A$, $g(A)\, f(A)$;
2) $f \in [\Pi_{A \in \mathcal{A}}\ g(A)]$ *iff* $f \in \Pi_{A \in \mathcal{A}}\ g(A)$ *and* $\exists n,\ n \Vdash_{\Pi,g} f$.

Clearly, $([\Pi_{A\in \mathcal{A}}g(A)], \Vdash)$ in 5.4 is a well defined object of ω-**Set**. In categorical terms, $([\Pi_{A\in \mathcal{A}}g(A)], \Vdash)$ is an indexed product in ω-**Set** "indexed over" itself. It is very easy to prove that, when the range of g is restricted to $\mathcal{M}$, the product lives in $\mathcal{M}$.

5.5 Theorem. *Let* $(\mathcal{A}, \Vdash_{\mathcal{A}}) \in \omega\text{–}\mathbf{Set}$ *and* $g: \mathcal{A} \to \mathcal{M}$.
Then $([\Pi_{A \in \mathcal{A}}\ g(A)], \Vdash_{\Pi,g}) \in \mathcal{M}$.

This is the semantic core of impredicativity. Suppose that $\mathcal{A} = \mathbf{M_0} = (\mathbf{PER},\Vdash_{M})$ and that $g : \mathbf{M_0} \to \mathcal{M}$ interprets type B, possibly depending on X. Then the interpretation of $(\forall(X{:}Tp)B) : Tp$, as the product in 5.5, gives mathematical meaning to the apparent circularity of impredicative second order types: for the object part of the product one has $([\Pi_{A\in \mathbf{M_0}}g(A)], \Vdash_{\Pi,g}) \in \mathbf{M_0}$. In other words, this meaning is simply obtained by proving the closure property in 5.5 for a category, **PER** or $\mathcal{M}$, built out of any applicative structures with "enough structure" (indeed, the natural numbers with Kleene's application suffice). Of course, one needs to check that the internal product is indeed the right adjoint to the internal diagonal functor, as summarized in [AL91] (this is the "small-completeness" of **PER**).

Note now that, when g is constantly equal to an ω-set $(\mathcal{B},\Vdash)$, say, in particular when g interprets a type B with no variable X freely occuring in it, then $([\Pi_{A\in \mathcal{A}}g(A)], \Vdash_{\Pi,g}) = \omega\text{-}\mathbf{Set}[\mathcal{A},\mathcal{B}]$. Clearly, $(\mathcal{B},\Vdash)$ lives in $\mathcal{M}$, i.e. it is a p.e.r., if B contains no variables at all or when an environment has been fixed. Recall now that the realizability relation in $\mathbf{M_0}$ is "full": that is $\Vdash_{M} = \omega \times \mathbf{PER}$ or any object is realized by any number. Therefore, any map that takes the ω-set $\mathbf{M_0}$ of all p.e.r.'s (which has just one equivalence class) to a single p.e.r.

is constant, as it must take equivalence classes (just one) to equivalence classes. By this, Axiom C is valid in the model, in the strongest sense, i.e. for all morphisms and types, not just for definable ones. In summary, for any $B \in \mathbf{PER} \approx \mathbf{M_o}$, one has $([\Pi_{A\in \mathbf{PER}} B], \Vdash_{\Pi,g}) = \boldsymbol{\omega}\text{-}\mathbf{Set}[\mathbf{M_o},B]$ and that $\boldsymbol{\omega}\text{-}\mathbf{Set}[\mathbf{M_o},B]$ is the set of the constant functions from **PER** to B. That is:

5.6 Fact. *Axiom C is valid in* ***PER*** *models.*

Let's see now in which way a precise form of "structural uniformity" suggests an understanding of the syntactic uniformity in 5.2. In [Gi71] and [Troe73], $\forall(X{:}Tp)B$ is interpreted by

(5.7) $\quad n\,[\forall(X{:}Tp)B]_\xi\, m \quad$ iff $\quad$ for all $C \in \mathbf{PER}$ $\; n\,[B]_{\xi[C/X]} m$.

That is, by

$$n\,[\forall(X{:}Tp)B]_\xi\, m \quad \text{iff} \quad n\,(\cap_{C\in \mathbf{PER}}[B]_{\xi[C/X]})\, m.$$

The interesting theorem, for our purposes, is that the intersection interpretation of second order types is isomorphic to their interpretation as products.

5.8 Theorem. *Let* $g : \mathbf{PER} \to \mathbf{M}$ *. Then*

$$In(\cap_{A \in \mathbf{PER}}\ g(A)) \cong [\ \Pi_{A \in \mathbf{PER}}\ g(A)],\ \text{ in } \mathbf{M}.$$

(See [LM91] for the proof).

Theorem 5.8 is the core step in the proof that the intersection interpretation in 5.7 satisfies the categorical adjunction mentioned above, up to isomorphisms, (indeed, some more work is needed, see [AL91]). Thus, it gives a logical meaning to the interpretation of "for all types" as intersections. Moreover, it provides the model theoretic understanding of the uniformity in 5.2, that we want to stress next.

In **PER**, terms are interpreted by equivalence classes of the interpretation of types as p.e.r. (indeed, the equivalence classes are the elements of a quotient). Set, for short, $S \equiv \cap_{A\in \mathbf{PER}} g(A)$ (we work up to the isomorphism *In*). The isomorphism $G : S \to \Pi_{A\in \mathbf{PER}} g(A)$ in 5.8 is defined by setting

$$G(\{n\}_S)(B) = \{n\}_{g(B)},\ \text{ for all } B \in \mathbf{PER}.$$

When $G(\{n\}_S)$ or $\{n\}_S$ interprets the term $\lambda(X)a[X] : \forall(X)C[X]$, for $g(X)$ interpreting $C[X]$, theorem 5.8 gives the semantic uniformity of the family of maps $\{a[X]\}_X$, by modeling the behaviour of the second order map $\lambda(X)a[X]$ as follows.

Note first that the application of $\lambda(X)a[X]$ to a type B is interpreted as the "projection" from $\{a[X]\}_X$ to $a[B] : C[B]$. Indeed, $G(\ \_\ )(B) : \cap_{A\in \mathbf{PER}}\ g(A) \to g(B)$ *is* a projection, since it is defined by $G(\{n\}_S)(B) = \{n\}_{g(B)}$ and, thus, projects $\cap_{A\in \mathbf{PER}}\ g(A) \cong \Pi_{A\in \mathbf{PER}}\ g(A)$ into the component $g(B)$ of the intersection or product. Moreover, the functional application of $G(\ \_\ )(B)$ to $\{n\}_S$ depends on $\Pi_{A\in \mathbf{PER}}\ g(A)$, similarly as a cartesian projection $fst_{A\times C}$ depends on $A\times C$, not only on A: this is why from an intersection we can reconstruct one of its components. This projection though is very peculiar, since $g(B) \supseteq \cap_{A\in \mathbf{PER}}\ g(A)$ and, thus, it simply coerces $\{n\}_S$ to the larger equivalence class of n itself in $g(B)$, namely to $\{n\}_{g(B)}$. In conclusion:

5.9 Remark. *In the* $\boldsymbol{\omega}$ ***-Set / PER*** *models, the uniformity of the syntactic definition of the family* $\{a[X]\}_X$ *is understood by the fact that the same* n *may be taken as a* *representative of all the semantic equivalence classes which interpret the components a[B] : C[B], for all types B.*

This construction is deeply rooted in the logical structure of these categorical models, as part of the topos theoretic frame provided by **Eff**. Indeed, a crucial property of **Eff** is the validity in it of the Uniformity Principle (UP). That is, if Φ is a formula of IZF (Intuitionistic Zermelo Fraenkel set-theory), the following holds:

(UP) $\forall A \in \mathbf{PER}\ \exists n \in \omega\ \ \Phi(n,A) \quad \Rightarrow \quad \exists n \in \omega\ \ \forall A \in \mathbf{PER}\ \ \Phi(n,A)$.

The proof of the isomorphism in 5.8 uses (UP) (see [LM91], for details, or [Roso86] for a general topos theoretic discussion on constructivity in **Eff**). We just observe here that (UP) corresponds to the contrapositive of König's lemma: "in a brown finitely branching infinite tree, if *for any* branch *there exists* a node where the branch switches to green, then *there exists* a (uniform) level such that *any* branch is green". Thus, (UP) is classically (not intuitionistically) equivalent to König's lemma, a well established principle. In our case, it allows to go from n, a priori depending on A in $\{n\}_{g(A)}$, to the same, uniform, n for all A; this provides an understanding of the uniformity in 5.2, by a "constructive" model.

Open problems. 1 - We pointed out that **PER** is a model of Fc. Since $\forall(X)X$ and $\forall(X)(X\to X)$ denote initial and terminal objects, **PER** is not a model of (Gen), i.e. of Genericity as an implication. Is there a class $\mathcal{A}$ of generic objects, in the sense of problems 2 at the end of §.4 ? Can we construct an interesting subcategory, with no trivial objects (initial, terminal), which is still a model of Fc and realizes (Gen) ?

2 -There is no space here to hint at two more, very relevant understandings of parametricity. First, the meaning of (polymorphic) terms as "dinatural transformations", proposed in [BFSS90] and further developed by [GSS91], among others. Second, the "logic for parametricity" in [PA93], a stimulating blend of Logical Frames and Logical Relations.

Just note then that the interpretation of second order terms as dinatural transformations may be actually given over **PER** models. However, the interpretation of $\forall$ is obtained on a slightly different (sub-)category. In particular our explanation above of the intended uniformity is lost, as the object corresponding to $\forall$ types does not seem to be related to the intersection. We wonder whether it may be seen as a subtype, in the sense of [BL90]. Then, since subtype does not mean subobject, in the categorical sense, is there any other tidy categorical (universal) meaning to this possible inclusion ? Does this preserve the understanding of parametricity via (UP) ? Can one interpret the logic in [PA93] in either of these models ?

An answer would relate the different ways to describe parametricity: by logical relations, by uniformity, by dinaturality.

6. Overloading as Type Dependent Polymorphism.

By the two main approaches to parametricity mentioned above we understand, in very different ways, that second order (polymorphic) functions cannot "compute" with input types, i.e. that their output "value" cannot depend on input types. In a sense, this justifies the practice of erasing type information at run-time (in ML, Quest...): only the type, not the result of a computation may depend on type parameters.

However, in actual programming languages, types may be coded. After all, type symbols are countably many and programmers are not always very concerned by the intended interpretation of second order variables. Thus computations depending on types do exist in the practice of programming. Usually, though, true type dependency is resolved at compile-time. For example, the familiar overloaded functions of many imperative languages (or of imperative features of functional languages) are given different values, according to type information, before computing. Typical examples are the "+" or "print" functions in most executable languages, where their overloaded meaning is decided when checking the type of the inputs, at compile-time. Usually these constructions are as untidy as low level code writing. Moreover, the early resolution of overloading has little expressiveness and little mathematical relevance. However, this should not mislead us from this further expressiveness of programming; as already mentioned, codes for types can be manipulated. Thus, in an even more constructive approach to reality, i.e., in actual programming, one may have functions whose output values depend on input types. As a matter of fact, "ad hoc" polymorphism is a powerful and useful feature and a further mathematical challenge. Too bad that it has been given a name with a negative connotation by the founding fathers of programming language theory; this name and their influential role may have diverted or delayed investigation from an important aspect of computing.

The point is to embed "ad hoc" polymorphism into a sound mathematical frame and turn it into a general, non ad hoc, programming tool.

We summarize here the proposal for the investigation of a true type dependency, viewed as overloading, made in [CGL93]. In that paper, a robust use of overloading is proposed in order to investigate some aspects of Object Oriented Programming in a functional frame. We directly borrow from [CGL93] a brief introduction to this typically "ad hoc" polymorphism. Note that this section is not meant to be a survey nor a discussion on overloading (and subtying), as we tried to do for parametricity in the previous section. The author should consult for this [Rey80], [Rey88] and [Ten89], for example (and [CC90] for a relevant foundation of subtying). Our aim is to justify here a "non parametric" calculus, after such a lengthy analysis of the parametric ones. Thus, reference to overloading is just because this commun programming feature is a simple form of "type-dependency".

The motivation in [CGL93] comes from considering overloading as a way to interpret message-passing in object-oriented programming, where methods are viewed as "global" functions: they are named "outside" the objects and their (operational) value is specified as soon as the name of each global function is associated to an object. This value may change entirely according to the given object: overloading is not parametric in the sense of system F.

In short, in object-oriented languages, computations evolve on objects. Objects are programming items grouped in classes and possess an internal state that is modified by sending messages to the object. When an object receives a message it invokes the method (i.e., code or procedure) associated to that message. The association between methods and messages is described by the class the object belongs to. In particular, objects are pairs (internal state , class_name).

The idea then is to consider messages as names of overloaded functions and message passing as overloaded application: according to the class (or more generally, the type) of the object that the message is passed to, a different method is chosen (this is similar to programming in CLOS, for example). Thus, we pass objects to messages, similarly as types are passed as inputs to the polymorphic functions of system F. The crucial difference is that parametricity is lost by allowing a finitely branching choice of the possible code to be applied. And this choice will depend on types as inputs (or, more precisely, on the type of the inputs).

In the formalism designed in [CGL93], terms describe overloaded functions by "gluing together" different "pieces of code". Thus the code of an overloaded function is formed by several branches of code. The branch to execute is chosen when the function is applied to an argument, according to a selection rule which uses the type of the argument.

A key feature of this approach is that the branch selection is not performed on the basis of the type that the argument possesses at compile-time. As already mentioned this is a fundamental limitation of overloading as used in imperative languages (early binding). In the present approach, the selection is performed each time the overloaded application is evaluated during computation. Moreover, the branch selection can be performed only when the argument is fully evaluated, and depends on its "run-time type" (late binding) which may differ from the compile-time type.

For example, suppose that *Real* and *Nat* are subtypes of *Complex* and that *add* is an overloaded function defined on all of them, and suppose that x is a formal parameter of a function, with type *Complex*. Assume also that the compile-time type of the argument is used for branch selection (early binding). Then an overloaded function application (here denoted •), such as the following one

$$\lambda(x : \mathrm{Complex})(\ldots \mathrm{add} \bullet x \ldots),$$

is always executed using the *add* code for complex numbers; with late binding, each time the whole function is applied, the code for *add* is chosen only when the parameter x has been bound and evaluated. Thus the appropriate code for *add* is used on the basis of the run-time type of x and according to whether x is bound to a real or to a natural number.

In summary, in [CGL93] a simple extension of the typed lambda-calculus is designed, which is meant to formalize the behavior of overloaded functions with late binding in a type discipline with subtyping. The first point to add to ordinary λ-terms, new terms such as $(M_1\&\ldots\&M_n)$ that represent the overloaded function composed by the n branches M_i, for $i \leq n$. We extend then the ordinary functional application $M(N)$ by an operation of overloaded application $M\bullet N$.

The types of the overloaded functions are finite lists of arrow types

$$\{U_1\to V_1, \ldots, U_n \to V_n \}$$

(denoted by $\{U_i \to V_i\}_{i\in I}$ for a suitable set I), where every arrow type is the type of a branch. Overloaded types, though, must satisfy relevant consistency conditions, which, among others, take care, in our view, of the longstanding debate concerning the use of covariance or contravariance of the arrow type in its left argument. More precisely, the general arrow types will be given by contravariant "→" in the first argument: this is an essential feature of (typed) functional programs, where type assignment (type-checking) helps avoiding run-time errors, and corresponds to the contravariance of the **Hom** functor in categories. Instead, the types of overloaded functions are *covariant families* of arrow types, as explained later.

The subtyping relation below is a complex, but expressive, feature of the calculus: it allows multiple choices, as a type may be a subtype of several types and subtyping is used to choose branches of overloaded terms. The blend of &-terms and subtyping makes this calculus an expressive and original mathematical formalism which shows, we claim, that "ad hoc" polymorphism may also have theoretical relevance. Here is a short survey of some basic ideas in the calculus and its reduction rules.

Subtyping on arrow types $U \to V$ is defined by contravariance w.r.t. U and covariance w.r.t. V, as usual and as mentioned above. On overloaded types, it expresses that a type $T' = \{ U'_j \to V'_j\}_{j\in J}$ is smaller than another $T'' = \{U''_i \to V''_i \}_{j\in I}$, if the programs in T' also type check when given as input an argument meant for programs in T'' (see the rule [$\to$ ELIM $_{(\leq)}$] below):

$$\frac{\text{for all } i\in I, \text{ there exists } j\in J \text{ such that } U''_i \leq U'_j \text{ and } V'_j \leq V''_i}{\{ U'_j \to V'_j\}_{j\in J} \leq \{U''_i \to V''_i \}_{i\in I}}$$

Well-formed types are defined by using the (pre-)order on them (in case the preorder gives a set instead of a single element, e.g. the greatest lower bound, we choose a canonical one). The definition gives the structure of a family of covariant types to overloaded types (see 3(b)):

1. $A\in$ Types
2. if $V_1, V_2\in$ Types, then $V_1\to V_2\in$ Types
3. if for all $i, j\in I$
   (a) ($U_i, V_i\in$ Types) and
   (b) ($U_i \leq U_j \Rightarrow V_i \leq V_j$) and
   (c) If, when U_i and U_j have a common lower bound, there is a unique (or canonical) $h\in I$ such that $U_h = \inf \{U_i, U_j\}$, then $\{U_i \to V_i\}_{i\in I} \in$ Types

Terms are defined by adding &-terms and overloaded application:

$M ::= x^V \mid c \mid \lambda(x^V)M \mid M(M) \mid M\&^V M \mid M\bullet M$

The crucial type-checking rules are the following. Note the type label over the &, in &-terms.

[$\to$ ELIM $_{(\leq)}$]

$$\frac{\vdash M: U\to V \qquad \vdash N: W \leq U}{\vdash M(N): V}$$

[{}INTRO]

$$\frac{\vdash M: W_1 \leq \{U_i \to V_i\}_{i \leq (n-1)} \qquad \vdash N : W_2 \leq U_n \to V_n}{\vdash (M\&^{\{U_i\to V_i\}_{i \leq n}} N) : \{U_i\to V_i\}_{i \leq n}}$$

$$[\{\}\text{ELIM}] \quad \frac{\vdash M: \{U_i \to V_i\}_{i\in I} \quad \vdash N : U \quad U_j = \min_{i\in I} \{U_i \mid U \leq U_i\}}{\vdash M \bullet N: V_j}$$

The last rule says that the output of an overloaded application lives in a type depending on the type of the input, namely the type V_j corresponding to the least U_i which contains the type of the input In a sense, U_i is the least type which allows the rule $[\to \text{ELIM}(\leq)]$ to be applied (this is were subtyping blends with overloading in a crucial way). Indeed, the reduction rule below says that the value also depends on the type of the input, as the intended M_j is chosen inductively by using, again, the type of the input and the type label on the &.

$\beta\&$) If $N : U$ is closed and in normal form and $U_j = \min \{U_i \mid U \leq U_i\}$ then
$((M_1 \&^{\{U_i \to V_i\}i=1..n} M_2) \bullet N)$ >> "if $j < n$ then $M_1 \bullet N$, else $M_2 \bullet N$ for $j = n$"

Clearly, the choice performed by the ($\beta\&$) rule may give essentially different output values, as no restriction is placed on the computation expressed by the terms. Informally, one obtains a reduction $(M_1 \& ... \& M_n) \bullet N >> M_j(N)$, for $j \leq n$ depending on the type of the input N. The motivations for the conditions on N (call by value) are discussed in [CGL93]. (β) reductions are defined as usual (but $[\to \text{ELIM}_{(\leq)}]$ may let the type decrease during computations).

The non-obvious fact of this calculus is that it satisfies Strong Normalization and the Church-Rosser theorem, see [CGL93].

We believe that this sets on solid "functional" and non "ad hoc" grounds some aspects of Object Oriented Programming, when message passing is described as overloading. More is said in [CGL93], where further motivations for this proposal for type dependency, or computations depending on input types, are given.

Semantics and open problems. The approach above is just a preliminary attempt, as the goal would be to reach the smoothness and "uniformity" of higher order λ-calculi, in formalizing features that cannot be expressed in that calculus. The gluing together of terms given here is rather heavy. It takes care of many aspects, beyond type dependency, namely late binding and flexible subtyping, but it should be turned into an explicitly second order system, if ever possible. One should allow, say, notations such as $\lambda(X)(...\&^X...)$ and still preserve the effectiveness (normalization?) of the present system (Castagna and Pierce are exploring this and other directions, in ongoing work). Then we would really reach an alternative language to current functional approaches, restricted as they are by the limitations of parametricity.

However, there may be crucial difficulties there. The intended meaning of second order variables as subsets of the domain of interpretation is in contrast to the decidability, within a programming language, of type equality (or inclusion). Thus, the restrictions to finitely many choices, as described above, seems unavoidable. Alternatively, one should compare, by inclusion, only atomic or ground types.

Yet another direction, which gives meaning to the system above according to the practice of programming, is proposed in [CGL93s]. The idea is that types are just (countably many) symbols, which can be effectively compared by inclusion or equality. Or types are code, as anything else in programming languages. Then the semantic convention of second order logic is abandoned and types are interpreted in the same universe as terms.

In short, the model is constructed out of a **PER** model. This is built over an applicative structure which is also an *injective* topological space, in the category of T_0 topological spaces and continuous maps: namely, any function from a subset of a T_0 space to it can be extended to a representable (continuous) one. Interpret then the collection of types as a topologically discrete subspace, by "coding" them into the domain of interpretation, and extend (the meaning of) any function from types to terms to a morphism in the category. Clearly, the semantics of types squeezes them at the same level as terms, thus terms can compute with types.

Can something better be done, while still preserving the intuition inherited from (run-time) overloading ? Are there categories with the effective flavor of the ones we mentioned in the previous sections, that can distinguish different levels of decidability and allow a constructive type dependency? One should preserve, though, the expressiveness of impredicative definitions and the second order semantic convention. This is a non minor categorical challenge. Impredicative definitions are interpreted by non trivial closure properties of categories (small completeness, in our case). It is not clear whether these may be compatible with a true type dependency.

Aknowledgements. I am greatly endebted to K. Milsted for the many comments on this paper and for the joint work on Genericity: Sergei Soloviev's and her calm enthousiasm and commitment to a careful understanding of a complicated matter meant much for our collaboration. Roberto Bellucci is currently working at the non trivial semantics of the system in [ACC93] and made several helpful comments on §.3. Pierre-Louis Curien and Furio Honsell also suggested several improvements.

References

[ACC93] M.Abadi, L. Cardelli, and P.-L. Curien, Formal parametric polymorphism. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, 1993.

[AL91] A. Asperti, G. Longo *Categories, Types and Structures: an introduction to Category Theory for the working computer scientist.* M.I.T.- Press, 1991

[Bar84] H. Barendregt, *The Lambda Calculus, its syntax and semantics*, North-Holland, Amsterdam, revised edition, 1984

[BB85] C. Berarducci and C. Boehm, Automatic synthesis of typed A-programs on term algebras, *Theoret. Comput. Sci*. 39, pp.135-154, 1985.

[BFSS90] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott,. Functorial Polymorphism. *Theoretical Computer Science*, 70:35-64, 1990. Corresgendum *ibid*., 71:431, 1990.

[BL90] K. Bruce, G. Longo, A Modest Model of Records, Inheritance and Bounded Quantification, *Information and Computation* vol. 87, 1-2, 1990

[CC90] F. Cardone, M. Coppo, Two extensions of Curry's type inference system, In *Logic and Computer Science*, Odifreddi ed., Academic Press, 1990.

[Church41] A. Church, *The Calculi of Lambda Conversion*, Princeton Univ. Press, Princeton, 1941.

[CGL93] G. Castagna, G. Ghelli and G. Longo, A calculus for overloaded functions with subtyping *ACM Conference on LISP and Functional Programming*, San Francisco, July 1992 (*Information and Computation*, to appear).

[CGL93s] G. Castagna, G. Ghelli and G. Longo, The semantics for Lambda &-early: a calculus with overloading and early binding, *TLCA*, LNCS 664, Springer-Verlag, Utrecht, Feb. 1993.

[CL91] L. Cardelli and G. Longo, A semantic basis for Quest. In *Journal of Functional Programming* I(4), October 1991, pp.417-458.

[CMS91] L. Cardelli, J.C. Mitchell, S. Martini, and A. Scedrov, An extension of system F with Subtyping. To appear in *Information and Computation*. Extended abstract in T. Ito and A.R. Meyer (eds.), *Theoretical Aspects of Computer Software*, Springer-Verlag LNCS 526, 1991, pp. 750-770.

[CHS67] H. B. Curry, R. Hindley, J. Seldin *Combinatory Logic*, Vol. II, North-Holland, 1967.

[DiGiHon93] P. Di Gianantonio, F. Honsell, An abstract notion of application, *TLCA*, LNCS 664, Springer-Verlag, Utrecht, Feb. 1993.

[FrS92] P.J. Freyd and A. Scedrov, *Categories*, *Allegories*. Math. Library, North-Holland, 1990.

[FRR92] P.J. Freyd, E.P. Robinson, and G. Rosolini, Functorial parametricity. In *Proc. 7th Annual IEEE Symposium on Logic in Computer Science*, 1992.

[Gir71] J.-Y. Girard, Une extention de l'interpretation de Godel, In *Proceedings of the Second Scandinavian Logic Symposium, Studies in Logic 63*, J.E. Fenstad (ed.), North-Holland, Amsterdam, pp.63-92.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[GSS91] J.-Y. Girard, A. Scedrov, and P.J. Scott, Normal forms and cut-free proofs as natural transformations. In: Y.N. Moschovakis, editor, *Logic from Computer Science, Pro. M.S.R.I. Workshop, Berkeley, 1989*. M.S.R.I. Series Springer-Verlag, 1991.

[Hase93] R. Hasegawa, Categorical data types in parametric polymorphism, To appear in *Mathematical Structure in Computer Scienc*e.

[HL80] R. Hindley, G. Longo, Lambda-calculus models and extensionality, *Zeit. Math. Logik Grund. Math.* n. 2, vol. 26 (289-310), 1980.

[HS86] R. Hindley, J. Seldin, *Introduction to Combinators and Lambda-Calculus*, London Mathematical Society, 1986.

[Hyl82] M. Hyland, The effective Topos, *in The Brouwer Symposium,* (Troelstra, Van Dalen eds.) North-Holland, 1982.

[Hyl87] M. Hyland, A small complete category, *Annals of Pure and Applied Logic* , vol. 40, 1988

[LMS93] G. Longo, K. Milsted and S. Soloviev, The genericity theorem and the notion of parametricity in the polymorphic Lambda-calculus, *Theoretical Computer Science 1993, Special Issue in honour of C. Böhm, to appear* (an abstract in LICS'93, Montreal, June 1993).

[LM84] G. Longo, E. Moggi, The Hereditary Partial Recursive Functionals and Recursion Theory in higher types, *Journal of Symbolic Logic* n. 4, vol. 49, 1984

[LM90] G. Longo, E. Moggi, A category-theoretic characterization of functional completeness, *Theoretical Computer Science*, vol. 70, 2, 1990.

[LM91] G. Longo, E. Moggi, Constructive Natural Deduction and its "Ω-Set" Interpretation, *Mathematical Structures in Computer Science,* vol. 1, n. 2, 1991

[MaRey92] Q. Ma and J.C. Reynolds, Types, abstraction, and parametric polymorphism, Part 2. In S. Brookes *et al.*, editors, *Mathematical Fundations of Programming Semantics*, Springer-Verlag LNCS 598, 1992, pp. 1-40.

[Mai 91] H. Mairson, Outline of a proof theory of parametricity. In *Proc. 5-th Intern. Symp. on Functional*

Programming and Computer Architecture, 1991.

[Mil78] R. Milner, A theory of type polymorphism in programming. In *Journal of Computer and Systytem Science*, 17(3): 348-375, 1978.

[Mit88] J.C. Mitchell, Polymorphic type inference and containment. Information and Computation, 76(2/3): 211-249, 1988. Reprinted in *Logical Fundations of Functional Programming*, ed. G. Huet, Addison-Wesley, 1990, pp.153-194.

[MP86] J.C. Mitchell, G. D. Plotkin, Abstract types have exixtential types, *Proc. Popl 85*, ACM, 1986.

[PA93] G. D. Plotkin, M. Abadi, A logic for parametric polymorphism, *TLCA,* LNCS, Springer, Berlin, 1993.

[Rey74] J.C. Reynolds, Towards a theory of type structure, in *Colloque sur la Programmation*, LNCS 19, Springer, Berlin, pp.408-425, 1974.

[Rey80] J.C. Reynolds, Using Category Theory to design implicit conversion and generic operators, In *Semantics directed compiler generetion, LNCS 94,* Springer-Verlag, Berlin, 1980.

[Rey83] J.C. Reynolds, Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing'83*, pp. 513-523. North-Holland, 1983.

[Rey88] J.C. Reynolds, Preliminary design of the programming language Forsythe, Tech. Rep. CMU-CS-88-159, Carnegie Mellon University, 1988.

[Roso86] P. Rosolini, Continuity and efffectiveness in Topoi, *D. Phil. Thesis,* Oxford Univ., 1986.

[Ten89] R. Tennent, Elementary data structures in Algol-like languages, *Science of Computer Programming*, 13, 1989, pp. 73-110.

[Troe73] A. Troelstra Metamathematical investigation of Intuitionistic Arithmetic and Analysis, *LNM 34*, Springer-Verlag, Berlin, 1973.

[Wad89] P. Wadler, Theorems for free! in *4th internat. Symp. on FP Languages and Computer Architecture, London*, pp.347-359, ACM, 1989.