Types as Parameters[1]

Giuseppe Longo

LIENS (CNRS) et DMI, Ecole Normale Supérieure

45, Rue d'Ulm, 75005 Paris; longo@dmi.ens.fr

Abstract. This note is a brief survey and a discussion of recent ideas and open problems in the understanding of an important aspect of Type Theory: how terms may depend on types. This problem is at the core of the distinction between "ad hoc" and proper polymorphism and inspired the large amount of work on "parametricity".

1 Types

In Mathematics, functions are typed. One always talks of functions from reals to reals, or from integers to reals, or from a given vector space to another, or of homomorphisms of groups and so on so forth.... In a sense, functions are always viewed as arrows with specific source and target, in the intended category. The objects of the category are the types. The formalization of mathematics in the frame of a type-free Set Theory was an oversimplification of Frege. A fruitful one, though, as it clarified matters and stimulated the design of a rigorous type-theoretic approach, after a simple inconsistency was pointed out by Russell.

The "mistake" was iterated by Curry and Church, in their functional approach to foundation. Again, this gave rise to a paradox, Curry's paradox, which is similar to Russell's when xx, self application, stands for $x \in x$. In this case though, the solution was twofold: one could add types, in Russell's style [Church41] or, by dropping negation, one could obtain a computationally very expressive type-free system, where Curry's paradoxical combinator was turned into the key tool for computing all partial recursive functions. And this was the type-free λ-calculus. Its many relevant properties are

[1] The presentation is directly endebted to joint work and many stimulating discussions with Giuseppe Castagna, Giorgio Ghelli, Roberto DiCosmo, Simone Martini, Kathleen Milsted, Sergei Soloviev. With Simone, in particular, I discussed of the dinatural interpretation (§.4), while Sergei and Roberto pointed out to me the connection to the isomorphisms of types in §.5.

discussed in [Bare84].

It is worth pointing out that this branching of the functional approach to foundation, which had no impact on the practice of mathematics, originated the relevant areas of type-free and typed functional programming languages and that, in computing, these two directions nicely interact. Ordinary typed languages, both in the case of type checking and of type assignement, deal with types at compile time, while computations are meant to be type free. As well known, one can reconstruct types for (typable) type-free terms and, conversely, erasing type information preserves the computational power of terms, an issue to be discussed later.

2 Parametricity

More richness has been embedded in programming by borrowing from higher order logic: just allow type variables, not just term variables, similarly as one considers set-variables or variables ranging over the objects of a category. In this case, terms may have several types, namely, all the instances of their type schema obtained by the use of type variables, and, hence, programs are (implicitly) *polymorphic* [Mil78]. When the analogy with higher order systems is more fully pursued, one considers quantified type variables and obtains *explicit* polymorphism (Girard's system F; [Gir71], [Rey74]).

Clearly, when type variables, $X, Y, Z\ldots$, are allowed, then they naturally arise both in types and typed terms: $\lambda x{:}X.M$ is a term of type $X \to \rho$, where ρ is the type of M. But, then, terms, as functions, may depend on types. This is perfectly clear in explicit polymorphism, where one may abstract w.r.t. to type variables and terms may be applied to types: $P \equiv \lambda X.N$ of type $\forall X.\sigma$ and $P\tau$ of type $[\tau/X]\sigma$ are well formed. In particular, $\lambda X.N$ is an (explicitly) polymorphic functions from types to terms. Note the peculiar "dimension" (or type) of polymorphic functions: if constant types are objects in a category and terms are morphisms, then they are maps going from the objects of a category to the morphisms.

Several questions then come to the mind. Write $M[X]$ and $\sigma[X]$ in order to stress that X may occur in M and σ. Then the rule

$$\frac{M[X] : \sigma[X]}{\lambda X.M[X] : \forall X.\sigma[X]}$$

gives a uniform definition of the family of maps $\{M[X]\}_X$ with components $M[X]$. Note that this definining rule does not depend on the parameter X, or it is "uniform" in it.

Does the syntactic uniformity of the definition have some relevance in the computational meaning of M? Or, also, how do polymorphic terms, as functions from types to terms, actually behave?

What does it mean that types can be erased at compile time?

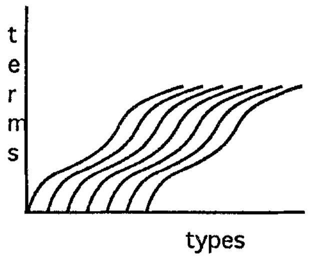
These questions are clearly related; the first two raise the issue of *parametricity*.

3 Genericity

The main motivation of this note is the need to relate the recent result in [LMS92] to (some of) the formal descriptions of how types act as parameters in functional languages. The result can be very easily stated. In a moderate extension of system F, the explicitly polymorphic calculus, types do not affect computations, in the sense that they are "generic" or "act like variables". More formally, call Fc the extension of F by "Axiom C" of the next section, then the following holds:

Theorem (Genericity) Let $M, N : \forall X.\sigma$. If, for some type τ, $M\tau =_{Fc} N\tau$, then $M =_{Fc} N$.

The proof relies on a non trivial technique of generalizing the two terms M and N, to obtain a (well-typed) common term from which the two terms can be equated. 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. Note that the terms $M\tau$, $M\rho$,.... in general may live in different types, namely $[\tau/X]\sigma$, $[\rho/X]\sigma$.... A suggestive pictorial representation of functions from types to terms, as parametric in types, may be the following:



Indeed, the Genericity Theorem tells us that different functions "never cross"; Reynolds' Abstraction Theorem (see §.6) tells us that polymorphic functions "preserve relations between inputs", or that they are "regular", in some precise sense to be discussed later.

The next two sections are meant to understand and justify the intended extension Fc of system F, by relating it to the semantics of F and the syntax of ML.

4 Axiom C and Dinatural Transformations

The Genericity Theorem is valid in a simple extension of system F, namely, Fc = F + Ax.C below. The idea is to impose that a polymorphic term, which outputs all values in the same type, is a constant function:

Ax.C If $M : \forall X.\sigma$, with $X \notin FV(\sigma)$, then $M\rho = M\tau : \sigma$, for any type ρ and τ.

This axiom is clearly compatible with F: the PER model, Girard's model over coherent domains and stable functions as well as other models realize it (see [LMS92]). More than this, Ax.C is truly in the spirit of system F: its negation, where $MY \neq MZ$ is interpreted by "they have a different erasure", yields a non normalizing system (see the remark in [Gir71] quoted in [LMS92]). We simply observe here that Ax.C is realized in all models where terms are understood as "dinatural transformations", defined below.

In §.2 we observed that the intended meaning of a polymorphic term $\lambda X.M$ is that of a function from the collection Ob_C of objects of a category C to the collection of morphisms. However, types may contain type variables. Thus, they are maps from objects to objects, more than just objects. Functors do have this dimension, hence *natural transformations* between functors may seem to provide the right meaning to polymorphic terms. Natural transformations are collections of morphisms, indexed over objects: $v = \{v_A\}_{A \in Ob}$ is a natural transformation, between functors F and G, if for any morphism $f : A \to B$, one has:

$$\begin{tikzcd}
FA \arrow[r, "v_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "v_B"'] & GB
\end{tikzcd}$$

Thus, if $[\![\sigma]\!]$ were the meaning of a type σ as object, the map v, with $v([\![\sigma]\!]) = v_{[\![\sigma]\!]}$, could interpret the polymorphic application of a term $\lambda X.M$ to a type σ. Unfortunately though, types are not functors, but just maps from Ob to Ob. The rub is that any type, in which the same variable occurs both at the right and the left of an "$\to$", should be at the same time a covariant and a contravariant functor. This is impossible, in general.

An interesting partial solution to this problem has been provided by [BFSS90] and further pursued by [GSS91], [FRR92]. The idea is that the interpretation of types, viewed as maps over the objects of a category, may be extended to functors, by distinguishing between the covariant and contravariant occurences of variables. Thus, an n-variable map is turned into a multivariant functor of nxn arguments. Over these kind functors, $F,G : C^n \times C^n \to C$, say, one may define *dinatural transformation* each family of morphisms $u = \{u_{\underline{A}} : F\underline{A}\underline{A} \to G\underline{A}\underline{A} \mid \underline{A} \in C^n\}$ such that, for any vector of morphisms $f : \underline{A} \to \underline{B}$, one has:

$$\begin{tikzcd}
 & F\underline{A}\underline{A} \arrow[rr, "u_{\underline{A}}"] & & G\underline{A}\underline{A} \arrow[dr, "G\underline{A}f"] & \\
F\underline{B}\underline{A} \arrow[ur, "Ff\underline{A}"] \arrow[dr, "F\underline{B}f"'] & & & & G\underline{A}\underline{B} \\
 & F\underline{B}\underline{B} \arrow[rr, "u_{\underline{B}}"'] & & G\underline{B}\underline{B} \arrow[ur, "Gf\underline{B}"'] &
\end{tikzcd}$$

The problem here is that, in general, dinatural transformations do not compose. Thus, a category with all required properties to yield a model, cannot always interpret terms as dinaturals, simply because terms can be composed. However, in [BFSS90] and [GSS91], some interesting models are given, where dinatural transformations do compose. These models then yield the dinatural interpretation of terms.

It is easy now, to point out that Ax.C is realized in any model for dinatural transformations. Indeed, equivalently rewrite Ax.C as

Ax.C* If $\Gamma \vdash N : \sigma$, with $X \notin FV(\Gamma) \cup FV(\sigma)$, then $[\rho/X]N = [\tau/X]N : \sigma$

where the context Γ is explicitly mentioned (it was irrelevant before; the condition $X \notin FV(\Gamma)$ is required now in order to satisfy the side condition in the $(\forall I)$ rule of system F and obtain the equivalence with Ax.C). Then the intepretation of N is a dinatural transformation which goes from the interpretation of Γ, as a product functor, to the interpretation of σ. The point is that both Γ and σ do not contain X free, thus the exagon above collapses to a pair of parallele arrows. Or, the models of terms as dinaturals realize Ax.C*. (Note that Ax.C* can be stated also in the frame of implicit polymorphism, where type variables are allowed but no explicit quantification).

5 Axiom C and the Isomorphisms of Types

As already recalled, ML style languages allow type schemata. Namely, types may contain type variables and a term possesses as types all instances of its type schema. Indeed, an algorithm assigns to each typable term its most general type, from which all of its types may be derived, by instantiation. If preferred, explicit universal quantifications may be used, but, in the spirit of types as schemata, they can be only external.

It may seem obvious then that ML is strictly "weaker" than system F: only certain kinds of quantifications are allowed. In particular, provable isomorphisms of types in ML should be provable also in F (as for the relevance of isomorphisms of types, see [DiCoLo89] and [DiCo93]). This is not so and the difference is expressed by Ax.C.

To see this fact more closely, consider the extension $F_\times$ of F obtained by adding cartesian products (that is add product types, projections and surjective pairings axioms, since usual descriptions of ML include them). In [DiCo93] it is shown that the isomorphism

(Split) $\forall X.\sigma\times\tau \cong \forall X\forall Y.\sigma\times[Y/X]\tau$, for $Y \notin FV(\sigma)$,

can be derived in ML, but not in $F_\times$. This is so by the peculiar use of variable substitution as given by the "let... in..." variant of (β) reduction: one may allow different types for the same term, while performing the substitution. In particular, the type-free terms $\lambda x.x$ and $\lambda x.<p_1x,p_2x>$, where the p_i are projections, yield the isomorphism in the implicitly polymorphic system, while their explicitly polymorphic versions, in system F, do not. Indeed, $\lambda z{:}(\forall X\forall Y.\sigma\times[Y/X]\tau).\lambda X.<p_1(zXX),p_2(zXX)>$ and $\lambda z{:}(\forall X.\sigma\times\tau).\lambda X.\lambda Y.<p_1(zX),p_2(zY)>$ only yield a retraction, in F, from $\forall X\forall Y.\sigma\times[Y/X]\tau$ into $\forall X.\sigma\times\tau$. Note that (Split) is the only isomorphism in the "difference" between ML and F (but, of course, $F_\times$ proves many more; see [DiCo93] for details).

The point here is that Ax.C fills the difference, as (Split) is provable in $F_\times c$. Observe first that, in Fc one has

(Iso.C) $\forall Y.\sigma \cong \sigma$, if $Y \notin FV(\sigma)$

where $\lambda x{:}(\forall Y.\sigma).xX$ and $\lambda z{:}\sigma.\lambda Y.z$ yield the isomorphism, by an explicit use of Ax.C. (Conversely, F extended by Iso.C, as given by the terms above, is equivalent to Fc.) Compute then

$$\begin{aligned}
\forall X\forall Y.\sigma\times[Y/X]\tau &\cong \forall X.(\forall Y.\sigma)\times(\forall Y.[Y/X]\tau) \text{ since } \forall Y.\sigma\times\rho \cong (\forall Y.\sigma)\times(\forall Y.\rho), \text{ in } F_X,\\
&\cong \forall X.\sigma\times(\forall Y.[Y/X]\tau) \text{ by Iso.C, as } Y\notin FV(\sigma),\\
&\cong (\forall X.\sigma)\times(\forall X\forall Y.[Y/X]\tau) \text{ as above}\\
&\cong (\forall X.\sigma)\times(\forall Y.[Y/X]\tau) \text{ by Iso.C,}\\
&\cong (\forall X.\sigma)\times(\forall X.\tau) \text{ by renaming}\\
&\cong \forall X.\sigma\times\tau .
\end{aligned}$$

In conclusion, we observed, following DiCosmo, that F_X is not an extension of ML, in spite of the popular belief (clearly, an isomorphism of types corresponds to the equations of two terms to the identity). We noticed though that Ax.C allows to overcome the difficulty and establishes, by this, a novel connection between the two systems. The observation is one more step towards justifying the usual practice in higher order functional programming, where types are not used at run-time. Indeed, in ML, types are only used at compile time. Programs are written as type free terms and an automatic type assignement system performs a partial correcteness check by assigning type schemas. Finally, programs run with no type information. This is perfectly fair in a programmming style where types are only used as a metalinguistic "dimensional control", at compile time. Why should running programs use no type information also in explicitly higher order systems such as Cardelli's Quest (see [CL91])? In those languages, types are first class citiziens of the language and they may be explicitly manipulated; why should the type erasures preserve the computational meaning of the intended computations, in these cases as well?

This is the core question discussed by the work on parametricity. Before getting into this issue, we note that the implicit use of Ax.C, by better establishing the theoretical connection to ML, already justifies the analogy at run time. As a matter of fact, in a recent investigation on the theoretical core of Quest, $F_{\leq}$ of [CMS91], a strong version of Ax.C is assumed. By the present remark, $F_{\leq}$ more closely relates to ML.

6 Types as Parameters

In the last two sections we introduced a preliminary understanding of the peculiar way in which terms depend on types in polymorphic functional languages and, at the same time, we justified the extension of system F by Ax.C. In particular, the meaning of terms as dinatural transformations sets an elegant connection to relevant areas of mathematics, where (di)naturality expresses a key categorical uniformity between functorial transformations. The validity of Ax.C in crucial models, see [LMS92], and in the dinatural intepretation, as well as its significance, shed some preliminary light on the "uniformity" of the dependence of terms from types.

A further insight into parametricity is given by a blending of two results, one of which has already been mentioned, the Genericity Theorem. The other is a syntactic understanding of Reynolds' work. In [Rey83] and [MaRey91] some abstract conditions are given such that, if a model satisfies them, then the intended meaning expresses a strong uniformity of terms w.r.t. types as inputs. In [ACC93] a syntactic treatment of Reynolds' approach is proposed. The advantage is given by a simpler presentation and by some relevant applications in the description of properties of programs (on the lines of [Wad89]). The key idea is the introduction of a (strong) extension of system F, system $\mathbb{R}$, which deals with terms, types and *relations* between types. The main result, besides the applications to properties of programs, may be stated as follows (and it may be considered as a reunderstanding of Reynolds' "Abstraction Theorem"). In $\mathbb{R}$, a type variable may be instantiated also by a relation R between types ρ and τ; this substitution, in a type σ, say, yields a relation $\sigma[R/X]$ between types $\sigma[\rho/X]$ and $\sigma[\tau/X]$.

Theorem (Abstraction) Let $M : \forall X.\sigma$. If a relation R is given between types ρ and τ, then $M\rho : \sigma[\rho/X]$ is related to $M\tau : \sigma[\tau/X]$ by $\sigma[R/X]$.

In other words, a polymorphic term takes related input types to related term values, in their output types (the result may be stated in a more general fashion by taking two related terms M and M' instead of one). The first point to be noted, now, is that $\mathbb{R}$ realizes Ax.C. Indeed, under the further condition $X \notin FV(\sigma)$, the relation $\sigma[R/X] \equiv \sigma$ collapses to the identity over $\sigma[\rho/X] \equiv \sigma \equiv \sigma[\tau/X]$. Thus, for any two types ρ and τ, whatever is the relation R between them, by the Abstraction Theorem, one has $M\rho = M\tau : \sigma$.

A most relevant application of parametricity, as described by the Abstraction Theorem, has been recently given by Hasegawa, [Hase93]. As well known, in intuitionistic second order logic implication and universal quantification, the type constructors of system F, are sufficient to form all other connectives. In particular, the absurdum, $\perp$, the existential quantifier $\exists$, *and* and *or* are all definable. However, all these definable connectives are weak, in the sense that $\perp$ (intuitively, the empty set or unprovable statement) is not initial and the others do not have the required projections or injections to be interpreted in all models as true existential, product and coproduct. The surprising result of Hasegawa is that the definable connectives have the right properties (initiality, projections...) exactly in those models of system F which are parametric, in the sense that they provide the relational frame for the Abstraction Theorem. Moreover, in [Hase93] it is shown that also initiality of free algebraic constructions holds exactly in presence of parametricity. In short, it is known that "free" algebraic types are representable in system F, see [BB85]: given an endofunctor G, $\forall X.((G(X) \to X) \to X)$ defines the algebraic type freely generated from G (for example, $G(X) = 0+X$, for a terminal object 0, generates the natural numbers). So far so good, but, the crucial algebraic property of initiality is usually lost by this weak definition, inside system F. Well, parametricity gives it back and allows to embed algebraic definitions, in their full expressiveness, into all parametric models of system F. If nothing else could be said about the relevance of parametricity, this should be enough to convince the reader of the interest of this uniformity property of system F. Indeed, it tells us why lambda calculus, by uniformely coding proofs by terms, is much more than just the logical systems of their types as propositions.

One final remark. The Abstraction Theorem (or its instance, as presented here) is, in a way, "dual" to the Genericity Theorem. Given *one* term M of type $\forall X.\sigma$, if *two* input types ρ and τ are related, then so are the output values $M\rho$ and $M\tau$. Dually, Genericity says that, given *two* terms M and N of type $\forall X.\sigma$, if they coincide on *one* input type τ, then M and N are equal. Thus, the two results study the consequence of applying one term on two inputs vs two terms on one input. Jointly, they give some robust information on the parametric dependence of terms on types, as we tried to suggest, very informally, by the picture at the end of §.2: polymorphic functions never cross and are all similarly regular (or preserve relations, all in the same way). The difficulty (and the research issue) here, is that the Genericity Theorem does not hold in system $\mathbb{R}$. Indeed, $\mathbb{R}$ realizes Ax.C, but does not need to realize its implicative consequences. Namely, the equational theory of $\mathbb{R}$ is an equational variety and nothing is known, a priori, about the implications between equations that it realizes. A formal understanding of this problem or of the relations between the two theorems, by a unified frame for parametricity, would surely shed further light on this crucial property of lambda calculus.

7 True type dependency or "ad hoc" polymorphism

The results just mentioned stress the faithful correspondence of system F, the core of higher order functional programming, to constructive logical system. The intended meaning of a type, as formalized in intuitionistic type theories, is that of possibly infinite collection of individuals. Effective computations can only be carried on individuals or elements of types and cannot use the infinite amount of information implicit in the notion of type. Thus, a properly constructive second order system, where type or set variables are explicitly allowed, cannot compute with these variables nor with their instantiation by type symbols. In a sense, this justifies the practice of erasing type information at run-time: only the type, not the result of a computation can 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 do exist in the practice of programming. Usually, though, true type dependency are 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 running 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 expressivity and little mathematical relevance. However, this should not mislead us from this further expressivity 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 [CGL92]. 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 [CGL92] a brief introduction to this typically "ad hoc" polymorphism.

The motivation come from considering overloading as a way to interpret message-passing in object-oriented programming, when methods are viewed as "global" functions:

they are named "outside" the objects and their (operational) value is specified as soon as the name of a global function is associated to an object. This value may entirely change 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 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 [CGL92], terms describe overloaded functions by "gluing up 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 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

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

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 [CGL92] 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 id 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 MN by an operation of overloaded application M•N.

The types of the overloaded functions are finite lists of arrow types $\{U_1 \rightarrow V_1, \ldots, U_n \rightarrow V_n\}$ (denoted by $\{U_i \rightarrow V_i\}_{i \in I}$ 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 "$\rightarrow$" in the first argument: this is an essential feature of (typed) functional programs, were type assignment (type-checking) helps avoiding run-time errors. Instead, the types of overloaded functions are covariant families of arrow types, as explained later.

We stress that the subtyping relation introduced 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 chose 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 have also theoretical relevance. Here is a short survey of some basic idea in the calculus and its reduction rules.

The subtyping relation is defined as usual on arrow types. On overloaded types, it expresses that a type $T' = \{U'_j \rightarrow V'_j\}_{j\in J}$ is smaller than another $T'' = \{U''_i \rightarrow V''_i\}_{j\in I}$, if the programs in T' type check also when given as input an argument meant for programs in T" (see the rule $[\rightarrow \mathrm{ELIM}_{(\leq)}]$ below):

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

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 family of covariant types to overloaded types:

1. $A\in$ Types
2. if $V_1, V_2\in$ Types, then $V_1 \rightarrow V_2 \in$ 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 $\{U_i \rightarrow V_i\}_{i\in I} \in$ Types

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

$$M ::= x^V \mid c \mid \lambda x^V. M \mid MM \mid M\&^V M \mid M \bullet M$$

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

$$[\rightarrow \mathrm{ELIM}_{(\leq)}] \qquad \frac{\vdash M: U \rightarrow V \quad \vdash N: W \leq U}{\vdash MN: V}$$

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

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

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 [$\to$ ELIM($\le$)] to be applied (this is were subtyping blends with overloading in a crucial way). Indeed, the reduction rule below says that also the value 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 &.

β&) If $N : U$ is closed and in normal form and $U_j = \min \{U_i \mid U \le U_i\}$ then

$((M_1\&^{\{U_i \to V_i\}_{i=1..n}} M_2)\bullet N) \mapsto$ "if $j < n$ then $M_1 \bullet N$, else $M_2 \bullet N$ for $j = n$"

Clearly, the choice performed by the (β&) rule may give essentially different output values, as no restriction is set on the computation expressed by the terms. Iformally, one obtains a reduction $(M_1\&...\&M_n)\bullet N \mapsto M_jN$, for $j \le n$ depending on the type of the input N. The motivations for the conditions on N are discussed in [CGL92]. (β) reductions are defined as usual (but [$\to$ ELIM $_{(\le)}$] 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 [CGL92].

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. Much more is said in [CGL92 and 93]. We only wanted to mention here some motivations and a proposal for true type dependency or computations depending on input types. The approach though is just a preliminary attempt, as the goal would be to reach the smoothness and "uniformity" of higher order λ-calculi. 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 (ongoing work of Castagna and Pierce is exploring this and other directions). Then we would really reach an alternative language to current functional approaches, restricted as they are by the limitations of parametricity.

References

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

[Bare84] H. Barendregt, *The Lambada Calculus, its syntax and semantics*, North-Holland,Amsterdam, revised edition, 1984

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

[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.

[Church41] A. Church, *The Calculi of Lambada Conversion*, Princeton University Press, Princeton

[CGL92] G. Castagna, G. Ghelli and G. Longo, A calculus for overloaded functions with subtyping **ACM Conference on LISP and Functional Programming**, San Francisco, Juillet 1992.

[CGL93] G. Castagna, G. Ghelli and G. Longo, The semantics for Lamda &-early: a calculus with overlaorading and early binding, Report LIENS.

[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.

[DiCo92] R. DiCosmo, Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, To appear in the Special Issue on ML.

[DiCo93] R. DiCosmo, Isomorphisms of Types, PhD Thesis, Universita di Pisa.

[DiCoLo89] R. DiCosmo and G. Longo, Constructively equivalent propositions and isomorphisms of objects (or terms as natural transformations). *Workshop on Logic for Comptuer Science*, Moschovakis (ed), MSRI, Berkeley, November 1989.

[FrS92] P.J. Freyd and A. Scedrov, *Categories, Allegories*. Mathematical 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'interpr›tation de Godel/l'analyse et la th›orie et son application/l'›limination des coupures dans l'analyse et la th›orie des types, 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 Science*.

[LMS92] G. Longo, K. Milsted and S. Soloviev, The genericity theorem and the notion of parametricity in the plymorphic Lamda-calculus, Report LIENS 92-25 (submitted to LICS93).

[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, Proceedings 1991*, 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.

[Rey74] J.C. Reynolds, Towards a theory of type structure, in LNCS, Springer, Berlin, pp.408-425.

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

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