A Calculus for Overloaded Functions with Subtyping

(extended abstract)

Giuseppe Castagna
Dipartimento d'Informatica - Pisa
LIENS(CNRS)-DMI
e-mail: castagna@dmi.ens.fr

Giorgio Ghelli
Dipartimento d'Informatica
Corso Italia 40, Pisa, ITALY
e-mail: ghelli@di.unipi.it

Giuseppe Longo
LIENS(CNRS)-DMI
45 rue d'Ulm, Paris, FRANCE
e-mail: longo@dmi.ens.fr

April 8, 1992

Abstract

We present a simple extension of typed λ-calculus where functions can be *overloaded* by adding different "pieces of code". In short, the code of an overloaded function is formed by several branches of code; the branch to execute is chosen, when the function is applied, according to a particular selection rule which depends on the type of the argument. The crucial feature of the present approach is that a subtyping relation is defined among types, such that the type of a term generally decreases during computation, and this fact induces a distinction between the "compile-time" type and the "run-time" type of a term. We study the case of overloaded functions where the branch selection depends on the run-time type of the argument, so that overloading cannot be eliminated by a static analysis of code, but is an essential feature to be dealt with during computation. We obtain in this way a type-dependent calculus, which differs from the various λ−calculi where types do not play, essentially, any rôle during computation. We prove Confluence and Strong Normalization for this calculus as well as a generalized Subject-Reduction theorem (but proofs are omitted in this abstract, see [CGL92]).

The definition of this calculus is driven by the understanding of object-oriented features and the connections between our calculus and object-orientedness are extensively stressed. We show that this calculus provides a foundation for typed object-oriented languages which solves some of the problems of the standard record-based approach. It also provides a type-discipline for a relevant fragment of the "core framework" (see [Kee89]) of CLOS.

1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0182...$1.50

1 Introduction.

An important distinction has been extensively used in language theory since a couple of decades, between parametric (or universal) polymorphism and "ad hoc" polymorphism (see [CW85]). Both the Proof Theory and the semantics of the first kind of polymorphism have been widely and deeply investigated by many authors (with some contribution also by two of the authors of the present paper), on the grounds of early work of Hindley, Girard, Reynolds and Milner and developed into robust programming practice. The second kind, with the notable exceptions of [WB89], [MOM90] and [Rou90], has deserved little theoretical attention and, consequently, its very wide use has been little affected by any comparable influence as the one exerced by implicit and explicit polymorphism in programming. Probably, the name itself, "ad hoc", has a discouraging connotation for any mathematical investigation. We believe though that time is mature for a theoretical analysis, and thus a "uniform and general" one, also of this programming feature. It turns out that the challenges it poses are non trivial: indeed, this paper is just a preliminary step towards a theoretical universe still to be discovered and which, we claim, may also affect language design. We present here a formalism where functions can be *overloaded* by adding a different "piece 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, according to a particular selection rule. In general this rule depends on the type of the argument the function is applied to. We do not present a general treatment for overloaded functions (this can be partly found in [Ghe91]), but we develop a purely functional approach that better suits the characteristics of object-oriented programming. In fact, our main goal is the definition of a kernel functional language for the study of some features of object-orientedness, such as subtyping, inheritance and message-passing. Since the approach is entirely novel, we first felt the need, by this

preliminary, proof-theoretic analysis, to develop the non trivial investigation of key functional properties. We focus thus on normalization, Church-Rosser property and "subject-reduction" (i.e. termination, consistency and "how types evolve during computation"), in the setting of a truly type dependent calculus. Indeed, "type dependency" (the fact that terms and values may depend on types) is the key property of the calculus. The various (higher order) calculi, such as Girard's System F or its extensions, allow type variables, abstraction w.r.t. type variables and the application of terms to types, but the "value" of an expression does not depend on the types which appear in it or, if preferred, they are *compiled* in the same machine-code. Indeed, this "type-erasure" property plays a crucial role in the basic proof-theoretic property of these calculi: the normalization (cut-elimination) theorem. In the semantic interpretations, this essential type independence of computations is understood by the fact that the meaning of polymorphic functions is given by essentially constant functions (see [AL91]). It is clear, instead, that overloaded functions express computations which depend on types, as different codes may be applied on the basis of input types. This is so in various imperative as well as functional languages; our motivation, though, comes from considering overloading as a key feature of object-oriented programming, when methods are viewed as "global" functions. Let's try to be more specific. In object-oriented languages the computation evolves on objects. Objects are programming items grouped in *classes* and possessing 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. Thus objects are implemented as pairs `(internal state , class_name)`. Now, there are two possible ways to implement message-passing: the first is to consider classes as arrays that associate to each message a method. Therefore when a message `m` is passed to an object `obj` then the method associated to `m` in the class of `obj` is looked for. A conceptual simplification can be made by substituting the field `class_name` in the implementation of an object by the array of the corresponding class: in this approach, an object has the form shown in Figure 1:

object	
internal_state	
`message_1`	method_1
⋮	⋮
`message_n`	method_n

Figure 1: Objects as records

This implementation has been extensively studied and corresponds to the "objects as records" analogy of [Car88] (see also [CL90]). The second way to implement message-passing, as shown in Figure 2, is to consider messages as names of overloaded functions: according to the class (or more generally, the type) of the object the message is passed to, a different method is chosen (this is the approach used in CLOS: see [DG87]). By this, in a sense, we in-

`message_i`	
`class_name_1`	method_1
⋮	⋮
`class_name_n`	method_n

Figure 2: Messages as overloaded functions

verse the previous situation: instead of passing messages to objects we now pass objects to messages. Objects are still implemented by pairs `(internal state , class_name)` and become in this way arguments of overloaded functions. This different approach seems to have, at the first sight, some advantages w.r.t. the "objects as records" paradigm, at least in a proof-theoretical study of the typed case. Indeed in the first approach objects carry methods with them; thus the types of the objects contain also the functionality of the value. This causes some problems and requires an excessive use of recursion. On the contrary, in the overloading approach, the type of an object is no longer blurred by functional types. The functionality is fully expressed by methods as global, overloaded functions. Of course other problems arise, especially in the modeling of the encapsulation of the state, though they do not seem overwhelming. On the other hand, the full expressiveness of records is recovered, as record types and values are derivable notions in our approach.

Briefly, in this paper we develop a simple extension of the typed λ−calculus meant to formalize the behavior of overloaded functions in a type discipline that uses also subtyping. The basic idea is that an overloaded function consists of a finite collection of ordinary functions that are stuck together to form the different branches. Its type will be the set of the types of its branches. Therefore we add to ordinary λ−terms, new terms such as $(M\&N)$ that represents the overloaded function composed by two branches M and N (more branches can be added by iterating the & as for $(M_1\&M_2\&\ldots\&M_n)$). In the following we will call a term of the form $(M\&N)$ an "&-term". The syntax of types will be enriched by finite sets of arrow types $\{V'_1 \to V''_1, \ldots, V'_n \to V''_n\}$ (sometimes denoted by $\{V'_i \to V''_i\}_{i\in I}$), where every arrow designates the type of a different branch. Overloaded types, though, must satisfy relevant consistency conditions, which, among others, take care, in our view, of the long-

standing debate concerning the use of covariance or contravariance of the arrow type in its left argument. 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.

In section 2 we describe the combination of overloading and subtyping and stress the advantages of their interaction. Section 3 presents the syntax of the system as well as the reduction rules. Sections 4 deals with the crucial (and difficult) normalization theorem and other syntactic properties. In section 5 we give some more intuition on how our calculus fits object-oriented programming, hinting how to implement subtyping and message-passing by the constructs of our calculus. In this abstract, proofs have been entirely omitted.

2 Overloading and Subtyping

Overloading by itself (i.e. without subtyping), does not increase the expressiveness of the language: an overloaded function can be substituted by the appropriate code, at compile-time; in this case, overloading seems more a notational trick than a programming construct. If combined, though, with subtyping, it becomes a very flexible and powerful tool. The idea is that if we have an overloaded function whose n branches have respectively type $U_i \to V_i$ $(i = 1..n)$ and we pass it an argument of type U, the chosen branch j is the one that "best approximates" U, that is such that $U_j = \min_{i \in I}\{U_i | U \leq U_i\}$. Now it is known that, in presence of a subtyping relation, the type of a term is no longer the same during computation, but it may decrease, see [CG92]. This "shrinking of the run-time type" corresponds to the increase of information (better, of "understandability") that characterizes the evolving of computation. Thus the choice of the branch can no longer be performed at compile time since the type of the argument may change during the computation and thus the designated branch as well (and indeed we want the choice to be performed on the most informative type of the argument, that is the one of its normal form). For this purpose, we distinguish for a term two kinds of types: the static type of a term i.e. the one which can be deduced looking at the term before its computation, and its run-time types i.e. these which it will possess looking at the term at different phases of its computation. The run-time types will be used in the selection of the branches of overloaded functions, while the static typing of a term is enough to assure that the computation will be type-error free as stated by theorem 4.1 (note though that the static type doesn't suffice to figure out how the computation will evolve). To satisfy this condition not every overloaded type should be accepted: indeed, we must require a "consistency" condition and avoid ambiguity, in case multiple choices are possible. In short, an overloaded type $\{U_i \to V_i\}_{i \in I}$ will be well-formed if and only if for all $i, j \in I$ it satisfies the following conditions:

$$U_i \leq U_j \quad \Rightarrow \quad V_i \leq V_j \tag{1}$$

$$U_i \Downarrow U_j \quad \Rightarrow \quad \text{there exists a unique } h \in I \text{ such that } U_h = \inf\{U_i, U_j\} \tag{2}$$

where $U_i \Downarrow U_j$ means that U_i and U_j are downward compatible, i.e. they have a common lower bound.

Condition (1) is a consistency condition, which assures that during computation the type of a term may only decrease. In a sense, this takes care of the common need for some sort of covariance of the arrow in the practice of programming. More specifically if we have a two-branched overloaded function M of type $\{U_1 \to V_1, U_2 \to V_2\}$ with $U_2 < U_1$ and we pass it a term N which has the compile-time type U_1 then the compile-time type of MN will be V_1; but if the normal form of N has type U_2 then the run-time type of MN will be V_2 and therefore $V_2 < V_1$ must hold. The second condition concerns the selection of the correct branch: we said before that if we apply an overloaded function of type $\{U_i \to V_i\}_{i \in I}$ to a term of type U then the selected branch has type $U_j \to V_j$ such that $U_j = \min_{i \in I}\{U_i | U \leq U_i\}$; condition (2) assures the existence and uniqueness of this branch.

These restrictions have a surprisingly natural interpretation when we consider the connection with object-oriented programming (see section 5).

3 The $\lambda\&-$calculus

In this section we define the extension of the typed lambda calculus we will study in the rest of the paper. We first define a set of *Pretypes* and then among them we will select those that satisfy the conditions above and that will constitute the types.

PreTypes

$$V ::= A \mid V \to V \mid \{V'_1 \to V''_1, \ldots, V'_n \to V''_n\}$$

where by A we denote an atomic type.

3.1 Subtyping rules.

We define a *subtyping* relation on the set of Pretypes. This relation is used to defines the types. The idea is that one may start from a partial order predefined on atomic (pre)types and extend it to all Pretypes: the relation is obtained by adding the rules of transitive and reflexive closure to the following ones:

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{U_1 \to V_1 \leq U_2 \to V_2}$$

$$\frac{\forall i \in I, \exists j \in J \; U_i'' \leq U_j' \text{ and } V_j' \leq V_i''}{\{U_j' \to V_j'\}_{j \in J} \leq \{U_i'' \to V_i''\}_{i \in I}}$$

Intuitively if we consider two overloaded types U and V as a set of functional types then the second rule states that $U \leq V$ if and only if for every type in V there is one in U smaller than it.

3.2 Types

On the base of the previous definition we select those Pretypes which satisfy the conditions of section 2:

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. $(U_i \Downarrow U_j \Rightarrow$ there is a unique $h \in I$ such that $U_h = \inf\{U_i, U_j\})$

   then $\{U_i \to V_i\}_{i \in I} \in$**Types**

The intuition on which overloaded types are based is the following. An overloaded type is inhabited by functions made out of different pieces of code. When an overloaded function is applied to an argument, a choice is made of the code that will be actually used in the computation. The choice is based on the type of the argument and the condition of 3.c assures its unicity.

3.3 Terms

Roughly speaking terms correspond to terms of the classical lambda calculus plus an operation which concatenates two different branches and forms an overloaded term. Since we want the branches of an overloaded function to be ordered, then we construct them as customary with lists, i.e. we start by an empty overloaded function and add branches, concatenated by ampersands. We also distinguish the usual application $M \cdot M$ of lambda-calculus from the application of an overloaded function $M \bullet M$ since they constitute two completely different mechanisms: indeed to the former is associated a notion of variable substitution while in the latter there is the notion of selection of a branch. This is stressed also by the proof-theoretical viewpoint where these constructors correspond to two different elimination rules. Finally, a further difference, specified in the reduction rules, is that overloaded application is associated to call by value, which is not needed by the ordinary application. For the same reason we must distinguish between the type $U \to V$ and the overloaded function type with just one branch $\{U \to V\}$.

Terms $M ::= x^V \mid \lambda x^V.M \mid M \cdot M \mid \varepsilon \mid M \&^V M \mid M \bullet M$

The type which indexes the & is a technical trick to allow the reduction inside overloaded function, as explained in [CGL92].

3.4 Type checking

The rule to type-check the calculus are shown in Table 1 in the next page.

As the reader will have noticed, we do not use the *subsumption rule* in our presentation of the type rules. However our system enjoys the subsumption *property*, i.e. for any $U \leq V$ and for any context $C[\,]$ and terms $M{:}U$and $N{:}V$, if $C[M]$ is well-typed then $C[N]$ is well-typed too. This means that our system could be presented using the subsumption rule. Notice, though, that with the subsumption rule the run-time type of a term (used only in the reduction rules, to perform branch selection), should be defined as the *minimum* type of a closed normal term,as terms would possess many types. In our system instead one has:

Theorem 3.1 *Every well-typed $\lambda\&$−term possesses a unique type*

3.5 Reduction

In order to simplify the definition of the reduction, we consider the types of overloaded functions as ordered sets. The order corresponds to the order in which branches appear, i.e. in which they are "constructed" according to the rules. Also, we will allow a reduction of the application of an overloaded function only when its argument is in normal form. This is a crucial point. If the argument of an overloaded function is reduced, its type may change (indeed, decrease by theorem 4.1). Therefore a different branch of the overloaded function might be chosen. As a matter of fact, in object oriented languages one can send messages only to objects in normal form. For example, in Smalltalk the expression

$object\ message_1\ message_2\ \ldots\ message_n$

is evaluated as

$(\ldots((object\ message_1)\ message_2)\ \ldots\ message_n)$

since the first message is passed to the object `object` the second to the object which is the result of the previous message-passing (in SmallTalk the result of every message is another object) an so on, but the message is sent *after* that the previous object has been calculated

In short, we define the reduction relation $\triangleright$:

β) $(\lambda x^S.M)N \triangleright M[x^S := N]$

$\beta\&$) If $N{:}U$ is closed and in normal form and $U_j = min\{U_i | U \leq U_i\}$ then

$$((M_1 \&^{\{U_i \to V_i\}_{i=1..n}} M_2) \bullet N) \triangleright \begin{cases} M_1 \bullet N & \text{for } j < n \\ M_2 \cdot N & \text{for } j = n \end{cases}$$

[TAUT] $$\vdash x^V : V$$

[$\rightarrow$ INTRO] $$\frac{\vdash M : V}{\vdash \lambda x^U.M : U \rightarrow V}$$

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

[TAUT$_\varepsilon$] $$\vdash \varepsilon : \{\}$$

[{}INTRO] $$\frac{\vdash M : W_1 \leq \{U_i \rightarrow V_i\}_{i \leq (n-1)} \quad \vdash N : W_2 \leq U_n \rightarrow V_n}{\vdash (M \&^{\{U_i \rightarrow V_i\}_{i \leq n}} N) : \{U_i \rightarrow V_i\}_{i \leq n}}$$

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

Table 1: Type checking rules for the λ&-calculus

context) If $M_1 \rhd M_2$ then

$$\begin{array}{rcl} (M_1 N) & \rhd & (M_2 N) \\ (N M_1) & \rhd & (N M_2) \\ (\lambda x^U.M_1) & \rhd & (\lambda x^U.M_2) \\ (M_1 \& N) & \rhd & (M_2 \& N) \\ (N \& M_1) & \rhd & (N \& M_2) \end{array}$$

The reasons of the two restrictions in the $(\beta_\&)$ is that, in our approach, two operations may change the type of a term: namely, reduction and substitution. Since we want the type of the argument of an overloaded function to be fixed, we require that it is in normal form in order to avoid reductions and that it is closed in order to avoid substitutions.

The intuitive operational meaning of $(\beta_\&)$ is easily understood when looking at the simple case, i.e. when there are as many branches as arrows in the overloaded type. In this case, under the assumptions (and the typing) in the rule, one has

$$(M_1 \& \ldots \& M_n)(N) \rhd^* M_j N$$

The nested formalization above of $(\beta_\&)$ is needed as M_1 may be an application $P_1 Q_1$, i.e. the "external operation" in M_1 is application, instead of an &.

3.6 Deriving records

In various approaches to object-oriented programming records play a very important rôle. In particular, current functional treatments of object-oriented features formalize objects directly as records. Moreover, if records are not included in a calculus, the subtyping relation may turn out to be quite trivial. In our system, records can be encoded in a straightforward way. Let $L_1, L_2, \ldots$ be an infinite list of atomic types. Assume that they are *isolated* (i.e., for any type V, if $L_i \leq V$ or $V \leq L_i$, then $L_i = V$), and introduce for each L_i a *constant* $\ell_i : L_i$. It is now possible to encode record types, record values and record selection, respectively, as follows:

$$\begin{array}{ll} \langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle & \equiv \{L_1 \rightarrow V_1, \ldots, L_n \rightarrow V_n\} \\ \langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle & \equiv (\varepsilon \,\&\, \lambda x^{L_1}.M_1 \,\&\ldots\&\, \lambda x^{L_n}.M_n) \\ & \qquad (x^{L_i} \notin FV(M_i)) \\ M.\ell & \equiv M \bullet \ell \end{array}$$

Since $L_1 \ldots L_n$ are isolated, then the subtyping rule for records is a special case of the corresponding rule for overloaded types:

$$\frac{V_1 \leq U_1 \quad \ldots V_k \leq U_k}{\langle\!\langle \ell_1 : V_1; \ldots; \ell_k : V_k; \ldots; \ell_{k+j} : V_{k+j} \rangle\!\rangle \leq \langle\!\langle \ell_1 : U_1; \ldots; \ell_k : U_k \rangle\!\rangle}$$

The type-checking rules are similarly derivable:

$$\frac{\vdash M_1 : V_1 \ldots \vdash M_n : V_n}{\vdash \langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle : \langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle}$$

$$\frac{\vdash M : \langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle}{\vdash M.\ell_i : V_i}$$

Finally, the rewriting rules (ρ) and (recd) below are just special cases of $(\beta_\&)$ and (context) respectively.

$$\begin{array}{ll} \rho) & \langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle.\ell_i \rhd M_i \quad (0 \leq i \leq n) \\ recd) & M \rhd M' \Rightarrow \\ & \quad M.\ell \rhd M'.\ell \\ & \quad \langle \ldots \ell = M \ldots \rangle \rhd \langle \ldots \ell = M' \ldots \rangle \end{array}$$

4 Main Theorems

In this section we present the main theoretical results for our calculus. We omit the proofs of the theorems even if they would deserve a wider treatment in view of the insight they can give in the understanding of the system; in fact the crucial properties of the overloaded types are heavily used in these proofs which actually suggested some of the latest features we added to our calculus. Though, to this further insight of the system correspond many technical difficulties whose treatment requires at least twice the space at our disposal here. The interested reader can refer to [CGL92] where, besides the proofs, she/he can also find an extensive discussion on the key features of our approach such as the distinction between run-time types and compile-time types, the dualism covariance vs. contravariance and some proof-theoretical aspects with hints to the semantics of type dependencies.

We start with a generalization of the subject reduction theorem which states that if a term is typeable then it can reduce only to typeable terms and these terms have a type lesser than or equal to the type of the redex.

Theorem 4.1 (Generalized Subject Reduction) *Let $M{:}U$. If $M \rhd^* N$ then $N{:}U'$, where $U' \leq U$.*

This theorem is important because it states that the computation well-behaves w.r.t. to types.

Next we have the Strong Normalization therorem. As well known, strong normalization cannot be proved by induction on terms, since β-reduction potentially increases the size of the reduced term. For this reason we introduce, along the lines of [Mit86], a different notion of induction on typed terms, called *typed induction*, proving that every typed-inductive property is satisfied by any typed term. This notion is shaped over reduction, so that some reduction related properties, like strong normalization or confluence, can be easily proved to be typed-inductive. We entirely omit the proof and just note that the main lemma for it, which proves that every typed-inductive property is satisfied by any typed term, is related to the normalization proofs due to Tait, Girard, Mitchell and others. We had to avoid, though, the notions of saturated set and of logical relation, which do not seem to generalize easily to our setting. This new methodology required some original technical insight.

Theorem 4.2 *Terms strongly normalize.*

Finally we can also prove the (syntactical) consistency of the calculus

Theorem 4.3 (Church-Rosser) *If $M \rhd P$ and $M \rhd Q$ then there exists N such that $P \rhd^* N$ $Q \rhd^* N$.*

The proof of this theorem is technically the easiest of the three since it is not difficult to show that the calculus is weakly Church-Rosser. Then, by the Newmann's lemma, one directly derives the Church-Rosser property for it. This theorem is important since assures that no matter how the calculus is implemented it always returns the same result. Thus it behaves in a deterministic way.

5 Overloading and Object-Oriented Programming

We already explained in the introduction the relation between object-oriented languages and our investigation of overloading. We discuss here some more this relation: by now, it should be clear that we represent class-names as types, and methods as overloaded functions that, according to the type (class-name) of their argument (the object the message is sent to), execute a certain code.

There exist many techniques to represent the internal state of objects in this overloading-based approach to object oriented programming. Since this is not the main concern of this research, we follow a rather primitive technique: we suppose that a program ($\lambda\&$-term) may be preceded by a declaration of *class types*: a *class type* is an atomic type, which is associated to a unique *representation type*, which is a record type. Two class types are in subtyping relation if this relation has been explicitly declared and it is *feasible*, in the sense that the respective representation types are in subtyping relation too. In other words class types play the role of the atomic types from which we start up, but in addition we can select fields from a value in a class type as if it belonged to its representation record type, and we have an operation $\_^{classType}$ to transform a record value $r{:}R$ into a class type value $r^{classType}$ of type *classType*, provided that the representation type of *classType* is R. Class types can be represented in our system generalizing the technique used to represent record types, but we will not show this fact in detail. We use *italics* to distinguish class types from the usual types, and $\doteq$ to declare a class type and to give it a name; we will use $\equiv$ to associate a name to a value (e.g. to a function). Thus for example we can declare the following class types:

$$\mathit{2DPoint} \doteq \langle\!\langle x : \mathrm{Int}; y : \mathrm{Int} \rangle\!\rangle$$
$$\mathit{3DPoint} \doteq \langle\!\langle x : \mathrm{Int}; y : \mathrm{Int}; z : \mathrm{Int} \rangle\!\rangle$$

and impose that on the types *3DPoint* and *2DPoint* we have the following relation $\mathit{3DPoint} \leq \mathit{2DPoint}$ (which is feasible since it respects the ordering of the record types these class types are associated to). A simple example of a method for these class types is *Norm*.

This will be implemented by the following overloaded function:

$$Norm \equiv (\ \lambda self^{2DPoint}.\sqrt{self.x^2 + self.y^2} \\ \&\ \lambda self^{3DPoint}.\sqrt{self.x^2 + self.y^2 + self.z^2} \\ )$$

whose type is $\{2DPoint \rightarrow \text{Real}, 3DPoint \rightarrow \text{Real}\}$.

Indeed, this is how we implement methods, as branches of global overloaded functions. Let us now carry on with our example and add some more methods to have a look at what the restrictions in the formation of the types (see section 2), become in this context.

The first condition, i.e. covariance inside overloaded types, expresses the fact that a version of a method which receives a more informative input returns a more informative output. Consider for example a method that updates the internal state of an object, like the method *Erase* which sets to zero the x component of a point:

$$Erase \equiv \\ (\ \lambda self^{2DPoint}.\langle x = 0; y = self.y\rangle^{2DPoint} \\ \&\ \lambda self^{3DPoint}.\langle x = 0; y = self.y; z = self.z\rangle^{3DPoint} \\ )$$

whose type is $\{2DPoint \rightarrow 2DPoint, 3DPoint \rightarrow 3DPoint\}$. Here covariance arises quite naturally.

Of course in the example the notation we used is quite cumbersome and redundant since we do not possess, in this core system, powerful operations on records (here, the update of a field) as for example in [CM91] or [Wan89].

As for the second restriction it simply says that in case of multiple inheritance the methods which are in common to ancestors not related by $\leq$, must be explicitly redefined. For example suppose to have also these definitions

$$Color \doteq \langle\langle c : \text{String}\rangle\rangle$$
$$2DColPoint \doteq \langle\langle x : \text{Int}; y : \text{Int}; c : \text{String}\rangle\rangle$$

and that we extend the ordering on the newly defined atomic types in the following (feasible) way: $2DColPoint \leq Color$ and $2DColPoint \leq 2DPoint$. Then the following function is not legal, as formation rule 3.c in section 3.2 is violated:

$$Erase \equiv \\ (\ \lambda self^{2DPoint}.\langle x = 0; y = self.y\rangle^{2DPoint} \\ \&\ \lambda self^{3DPoint}.\langle x = 0; y = self.y; z = self.z\rangle^{3DPoint} \\ \&\ \lambda self^{Color}.\langle c = \text{“white”}\rangle^{Color} \\ )$$

In object oriented terms, this happens since *2DColPoint*, as a subtype of both *2DPoint* and *Color*, heirs the *Erase* method from both classes. Since there is no reason to choose one of the two methods and no general way of defining a notion of "merging" for inherited methods, we ask that this multiply inherited method is explicitly redefined for *2DColPoint*. Notice that some object oriented languages do not force this redefinition, but use some different criterion to choose among inherited methods, usually related to the order in which class definitions appear in the source code. As discussed in [Ghe91], our rule 3.c in section 3.2 can be easily substituted to model these different approaches to the problem of choosing between inherited methods, allowing a formalization and a comparison of these approches in a unique framework. The approach we have chosen in this foundational study is just the cleaner and the simpler one in a context where the set of atomic types is fixed.

In our approach, a correct redefinition of the *Erase* method would be:

$$Erase \equiv \\ (\ \lambda self^{2DPoint}.\langle x = 0; y = self.y\rangle^{2DPoint} \\ \&\ \lambda self^{3DPoint}.\langle x = 0; y = self.y; z = self.z\rangle^{3DPoint} \\ \&\ \lambda self^{Color}.\langle c = \text{“white”}\rangle^{Color} \\ \&\ \lambda self^{2DColPoint}. \\ \qquad \langle x = 0; y = self.y; c = \text{“white”}\rangle^{2DColPoint} \\ )$$

which has type:

$$\{\ 2DPoint \rightarrow 2DPoint, \\ 3DPoint \rightarrow 3DPoint, \\ Color \rightarrow Color, \\ 2DColPoint \rightarrow 2DColPoint\ \}$$

The way we have written these methods may seem complicated with respect to the simplicity and modularity of object-oriented languages. Indeed the terms above can be regarded as the result of a compilation (or translation) of the following higher-level object-oriented program:

```
class 2DPoint
  state
    x:Int;
    y:Int
  methods
    Norm = sqrt(self.x^2 + self.y^2);;
    Erase = x <- 0;;
  interface
    Norm: Real;
    Erase: Likeself
endclass


class 3DPoint is 2DPoint and
  state
    z:Int
```

```
  methods
    Norm = sqrt(self.x^2+self.y^2+self.z);;
  interface
    Norm: Real
endclass


class Color
  state
    c:String
  methods
    Erase = c <- "white";;
  interface
    Erase: Likeself
endclass


class 2DColPoint is Color, 2DPoint and
  methods
    Erase = x <- 0; c <- "white";;
endclass
```

As for inheritance, note that this crucial feature of object-oriented programming is implemented here by the use of *branch selection* and *code reusing*. Recall now that, in our approach, the code of methods is *centralized* in the overloaded functions, as "global" functions, instead of being duplicated in every object, as the "objects as records" analogy seems to suggest. Then, when a message is sent to a receiver (in our language: an overloaded function is applied to an argument) exactly the code (the branch) defined in the superclass (supertype) of the class (the type) of the input is executed.

Thus, inheritance relies first on the fact that, when an overloaded function is applied to an argument, the *branch selected* is the $\min\{U_i | U \leq U_i\}$, where U is the class (type) of the argument. If this minimum is exactly U, this means that the receiver uses the method that has been defined in its class; otherwise, i.e. if this minimum is strictly greater, then the receiver uses the method that its class, U, has *inherited* from this minimum (a superclass); in other terms, the code written for the class which resulted to be the minimum, is *reused* by the objects of the class U[1]. Consider, say, the type *2DColPoint* and send the message *Norm* to it. Then the branch selected is the one defined for *2DPoint*. Indeed, $Norm$: {*2DPoint* $\rightarrow$ Real, *3DPoint* $\rightarrow$ Real} and the minimum type among those types greater than *2DColorPoint* is *2DPoint*; thus the branch selected is exactly the one which has been defined for *2DPoint*'s.

[1] Of course, real code-sharing depends on the implementation; in the previous example, when defining the class *2DColPoint*, the overloaded function *Erase* does not need to be completely redefined, as the new two branches are appended to the existing code

By this, we may say that inheritance *is* branch selection *and* code-reusing.

It is well known that problems arise when, in examples as the ones above, one tries to define a binary method like *Equal*. Let us see what happens in the "objects as records" analogy: if we add a method *Equal* to *2DPoint* and *3DPoint* then, in the notation typical of formalisms built around this analogy, we obtain the following recursive record types (we forget the other methods):

$2DEPoint \equiv$
$\langle\langle x : \text{Int}; y : \text{Int}; Equal : 2DEPoint \rightarrow \text{Bool}\rangle\rangle$
$3DEPoint \equiv$
$\langle\langle x : \text{Int}; y : \text{Int}; z : \text{Int}; Equal : 3DEPoint \rightarrow \text{Bool}\rangle\rangle$

and in this case the two types are not comparable because of the contravariance of the arrow type in *Equal*: since one would expect *2DEPoint* to be larger, as record, than *3DEPoint*, the type at the left of the outer arrow in *2DEPoint* should be larger, impossible by contravariance.[2] By the way, this is not to be considered a flaw in the system but a desirable property, since a subtyping relation between the two types could cause a run-time type error (see [CL91] for an example). The problem with the "object as record" analogy is that there is no way to write a method as *Equal* and compare, by subtyping, the two classes.

Our system is essentially more flexible, in this case. Indeed if we set *3DPoint*$\leq$*2DPoint* then an equality function,with type:

$$\begin{aligned} Equal: \{&2DPoint \rightarrow (2DPoint \rightarrow \text{Bool}),\\ &3DPoint \rightarrow (3DPoint \rightarrow \text{Bool})\} \end{aligned}$$

would not be well-typed in our system either, since $3DPoint \leq 2DPoint$ while $2DPoint \rightarrow Bool \leq 3DPoint \rightarrow Bool$. This expresses the fact that a comparison function cannot be chosen only on the basis of the type of the first argument. In our system instead we can write an equality function where the code is chosen on the basis of both arguments

$$\begin{aligned} Equal \equiv (\ &\lambda(p,q)^{2DPoint \times 2DPoint}.\ (p.x = q.x)\text{AND} \\ &\qquad\qquad\qquad\qquad\quad (p.y = q.y) \\ &\&\ \lambda(p,q)^{3DPoint \times 3DPoint}.(p.x = q.x)\text{AND} \\ &\qquad\qquad\qquad\qquad\quad (p.y = q.y)\text{AND} \\ &\qquad\qquad\qquad\qquad\quad (p.z = q.z) \\ )& \end{aligned}$$

the function above has type:

[2] Recursive types should be considered as denotations for their infinite expansion, and an infinite type is a subtype of another one when all the finite approximation of the first one are subtypes of the corresponding finite approximation of the second one; see [AC90].
A better notation for the types above would have been $2DEPoint \equiv \mu t\langle\langle x : \text{Int}; y : \text{Int}; Equal : t \rightarrow \text{Bool}\rangle\rangle$ and $3DEPoint \equiv \mu s\langle\langle x : \text{Int}; y : \text{Int}; z : \text{Int}; Equal : s \rightarrow \text{Bool}\rangle\rangle$

$$\{(\mathit{2DPoint} \times \mathit{2DPoint}) \rightarrow \mathrm{Bool},\ (\mathit{3DPoint} \times \mathit{3DPoint}) \rightarrow \mathrm{Bool}\}$$

which is well formed[3].

In presence of a subtyping relation, the covariance versus the contravariance of the arrow type, w.r.t. the left argument (domain), is a delicate and classical debate. Semantically (categorically) oriented people have no doubt: the hom-functor is contravariant in the first argument. Moreover, this nicely fits with typed models constructed over type-free universes, where types are subsets or subrelations of the type-free structure and type-free terms model runtime computations. Also the common sense of the type-checking suggests contravariance: if we consider one type subtype of another if and only if all expressions of the former type can be used in the place of expressions of the latter, then type-error free computations can be obtained only if contravariance is used for arrow types. However, practitioners often have a different attitude. In OOP, in particular, the "overriding" of a method by one, say, with a smaller domain (input type) leads to a smaller codomain (output type), in the spirit of a "preservation of information". Indeed, in our approach, we take care of both view points, as they are both correct, when viewed in the "right" frame.

As a matter of fact, our general arrow types (the types of ordinary functions) are contravariant in the first argument, as required by common sense and mathematical meaning. However, the *families* of arrow types which are glued together in overloaded types form covariant collections, by our conditions on the formation of these types (see 3.2). Besides the justification of this at the end of section 2, consider, say, the practice of overriding. Roughly, the implementation of a method in a superclass is substituted with a more specific implementation in a subclass. For example, the "+" operation, on different types, may be given by two different implementations: one implementation of type $\mathit{Int} \times \mathit{Int} \rightarrow \mathit{Int}$, the other of type $\mathit{Real} \times \mathit{Real} \rightarrow \mathit{Real}$. In our approach, we can glue these implementations together in a unique global method, exactly because their types satisfy the required covariance condition.

We have already noticed that part of the expressive power of our system derives from the ability of choosing one implementation on the basis of the types of many arguments. This ability makes it possible even to decide explicitly how to implement "mixed binary operations". For example, besides implementing "pure" equality between *2DPoint*'s and between *3DPoint*'s, we can also decide how we should compare a *2DPoint* and a *3DPoint*, as below:

$$\begin{aligned}\mathit{Equal} \equiv (\ & \lambda(p,q)^{\mathit{2DPoint}\times\mathit{2DPoint}}.\ \ \ldots \\ \&\ & \lambda(p,q)^{\mathit{3DPoint}\times\mathit{3DPoint}}.\ \ \ldots \\ \&\ & \lambda(p,q)^{\mathit{2DPoint}\times\mathit{3DPoint}}.(p.x = q.x)\mathrm{AND} \\ & \qquad\qquad\qquad\qquad\quad (p.y = q.y) \\ \&\ & \lambda(p,q)^{\mathit{3DPoint}\times\mathit{2DPoint}}.(p.x = q.x)\mathrm{AND} \\ & \qquad\qquad\qquad\qquad\quad (p.y = q.y)\mathrm{AND} \\ & \qquad\qquad\qquad\qquad\quad (p.z = 0) \\ )\end{aligned}$$

The ability to choose a method on the basis of more object parameters is called, in object oriented jargon, *multiple dispatch.*

We conclude this brief excursion in the object oriented world with a hint on the use of coercions. In object oriented programming a mechanism is often offered to choose a specific implementation for a method; the **super** identifier can be used inside method definitions to refer to the implementation of a method in an ancestor. To allow choosing a specific implementation for a method we offer a "run-time coercion" operator which changes the run-time type of an object, so that to get the *Norm* of a three-dimensional point M as if it where a two-dimensional one, the following expression can be used:

$$\mathit{Norm}\bullet(\mathbf{coerce}_{\mathrm{2DPoint}}(M))$$

These run-time coercions are different from the ones used in [BL90, CG92] since they are not used only to change the type of expressions but their run-time behavior too. Note that the use of coercions is more flexible than **super** as modelled in [CHC90, Mit90] since it is possible to remount farther than the direct ancestor, and it can be used everywhere and not only inside method definitions [4]. Furthermore, in a forthcoming paper we will show how, by using coercions, it is possible to encode the powerful calculi on record values presented in [CM91] and [Wan87].

Finally a pointer must be given to CLOS since our system provides a possible type-discipline for a fragment of this language including generic functions formed by primary multi-methods and whose dispatch never uses the Class Precedence List (see [DG87, Kee89]).

6 Conclusion: intersections, products and their semantics

This work is just the starting point of a new type discipline to be more extensively explored. We tried to

[3] This is not surprising as, even if the types of the two versions of equal are componentwise isomorphic, in general isomorphisms of types do not preserve subtyping: an iso may be a rather involved operation which has nothing to do with inheritance or related matters

[4] This is not the exact behavior of **super** in object oriented languages, since in oo-languages the branch selected by super works on the object which received the message, while in our case it works on a *coercion* of that object. This topic indeed deserves further work

present our motivations in the introduction, by stressing the need to found also the so called "ad hoc" polymorphism onto decent mathematical grounds, in particular in view of its role in the understanding of some object-oriented features. Reference has been given to the work we are aware of in the subject, which, by the way, has an entirely different perspective. One should also quote possible connections to other type disciplines, in particular the intersection types, originated in [CDCV81]. Indeed, at first glance, an overloaded type, in our sense, may seem an intersection of types: recent applications of intersection types in [Pie90] and in the programming language Forsythe, may suggest this analogy. However, this is not so. A term living in an intersection type loses the type information: semantically, from an intersection of types (sets), one cannot recover the collection of types (sets) which form the intersection nor the value of a term in an intersection may depend on a specific (input) type. This is a crucial point for our approach, where values depend on types (of inputs), and makes its semantics challenging and worth exploring. Note that "terms depending on types" is a novel and entirely different concept from "types depending on terms", as described in the (first order) types of Martin-Löf type theory or of the Calculus of Constructions.

Further work should also lead to a detailed investigation of "compile-time vs. run-time" types. In the complete version of this paper, [CGL92], we propose a simple view of this "dualism", which fits our approach. More should be said, though, in particular in connection with subtyping, coercions etc., i.e. with the various ways of dealing with "types evolving during computations".

As for the use of recursion, surely a very important tool for smooth programming practice, we believe that the theoretic investigation of complex issues, like this, should be made into two steps, if possible. First, analyze type disciplines were some "unshakable grounds" can be set: following the analogy "types as propositions" in λ-calculus, this means consistency proofs, via normalization, say, and related facts, as we tried to do here. Then, if everything works fine, add recursion, when really needed for computations, both for types and terms. This is another "methodological" point which distinguishes our approach to the current theoretical treatments of object-oriented features.

Acknowledgments. G. Castagna would like to thank Maribel Fernández for her comments on an early draft and Roberto Di Cosmo for his help in the work and patience in sharing an office. A very special thank to Franca and Nico, too.

References

[AC90] R. Amadio and L. Cardelli. *Subtyping Recursive Types.* Technical Report, Digital System Research Center, August 1990.

[AL91] A. Asperti and G. Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT-Press, 1991.

[BL90] K.B. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990. A first version can be found in *3rd Ann. Symp. on Logic in Computer Science*, 1988.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. A first version can be found in Semantics of Data Types, LNCS 173, 51-67, Springer-Verlag, 1984.

[CDCV81] M. Coppo, M. Denzani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeit. Math. Logik*, 27:45–58, 1981.

[CG92] P. L. Curien and G. Ghelli. Coherence of subsumption. *Mathematical Structures in Computer Science*, 2(1), 1992.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. *A calculus for overloaded functions with subtyping.* Technical Report 92-4, Laboratoire d'Informatique, Ecole Normale Supérieure - Paris, February 1992.

[CHC90] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance is not subtyping. *17th Ann. ACM Symp. on Principles of Programming Languages*, January 1990.

[CL90] L. Cardelli and G. Longo. *A semantic basis for Quest.* Technical Report, Digital System Research Center, February 1990. LISP and FP, Nice, July 1990; Journal of Functional Programming, 1(4):417-458 (to appear).

[CL91] G. Castagna and G. Longo. From inheritance to Quest's type theory. In *Ecole Jeunes Chercheurs du GRECO de Programmation*, Sophia-Antipolis (Nice), April 1991.

[CM91] L. Cardelli and J.C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[DG87] L.G. DeMichiel and R.P. Gabriel. Common lisp object system overview. In *Proc. of ECOOP '87 European Conference on Object Oriented Programming*, 1987.

[Ghe91] G. Ghelli. A static type system for message passing. In *Proc. of OOPSLA '91*, 1991.

[Kee89] S.K. Keene. *Object Oriented Programming in* COMMON LISP*: A Programming Guide to CLOS*. Addison-Wesley, 1989.

[Mit86] J. C. Mitchell. A type inference approach to reduction properties and semantics of polymorphic expressions. In *ACM Conference on LISP and Functional Programming (LFP)*, pages 308–319, 1986.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. *17th Ann. ACM Symp. on Principles of Programming Languages*, January 1990.

[MOM90] N. Martí-Oliet and J. Meseguer. *Inclusions and Subtypes*. Technical Report, SRI International, Computer Science Laboratory, December 1990.

[Pie90] B. Pierce. *Intersection and Union Types*. Technical Report, Carnegie Mellon University, 1990.

[Rou90] F. Rouaix. *ALCOOL-90, Typage de la surcharge dans un langage fonctionnel*. PhD thesis, Université PARIS VII, December 1990.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In *2nd Ann. Symp. on Logic in Computer Science*, 1987.

[Wan89] Mitchell Wand. Type inference for record concatenation and multiple inheritance. In *4th Ann. Symp. on Logic in Computer Science*, 1989.

[WB89] Philip Wadler and Stephen Blott. How to make "ad-hoc" polymorphism less "ad-hoc". In *16th Ann. ACM Symp. on Principles of Programming Languages*, pages 60–76, 1989.