A Semantics for Stratified $\lambda\&$-*early*

revised version of Liens-93-13

G. CASTAGNA G. GHELLI
Giuseppe LONGO

Laboratoire d'Informatique, URA 1327 du CNRS
Département de Mathématiques et d'Informatique
Ecole Normale Supérieure

LIENS - 94 - 22

November 1994

A semantics for a stratified $\lambda\&$-*early*

(revised version of LIENS-93-13)

Giuseppe Castagna
LIENS(CNRS)-DMI
Ecole Normale Supérieure
45 rue d'Ulm, Paris, FRANCE
e-mail: castagna@dmi.ens.fr

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

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

November 14, 1994

Abstract

The role of $\lambda-$calculus as core functional language is due to its nature as "pure" theory of functions. In the present approach we use the functional expressiveness of typed λ-calculus and extend it with our understanding of some relevant features of a broadly used programming style: Object-Oriented Programming (OOP). The core notion we focus on, yields a form of "dependency on input types" (or "on the types of the inputs") and formalizes "overloading" as implicitly used in OOP.

The basis of this work has been laid in [CGL92], where the main syntactic properties of this extension have been shown. In this paper, we investigate an *elementary* approach to its mathematical meaning. The approach is elementary, as we tried to follow the most immediate semantic intuition which underlies our system; yet it gives a rigorous mathematical model. Indeed, our semantics provides an understanding of a slightly modified version of the normalizing system in [CGL92]: we had, so far, to restrict our attention only to "early binding".

In order to motivate our extended λ-calculus, we first survey the key features in OOP which inspired our work. Then we summarize the system presented in [CGL92] and introduce the variant with "early binding" and "stratified" subtyping. Finally we present the model.

An extended abstract of this paper was presented at the International Conference on Typed Lambda Calculi and Applications, Utrecht, The Netherlands; March 1993

1 Introduction

The role of λ-calculus as core functional language is due to its nature as "pure" theory of functions: just application, MN, and functional abstraction, $\lambda x.M$, define it. In spite of the "minimality" of these notions, full computational expressiveness is reached, in the type-free case. In the typed case, expressiveness is replaced by the safety of type-checking. Yet, the powerful feature of implicit and explicit polymorphism may be added. With polymorphism, one may have type variables, which apparently behave like term variables: they are meant to vary over the intended domain of types, they can be the argument of an application and one may λ-abstract w.r.t. them. These functions depending on type variables, though, have a very limited behavior. A clear understanding of this is provided by a simple remark in [Gir72], where second order λ-calculus was first proposed: no term taking types as inputs can "discriminate" between different types. More precisely, if one extends system F by a term M such that, given different input types U and V, returns 0 on input type V and 1 on U, then normalization is lost. Second order terms, then, are "essentially" constant, or "parametric". Indeed, the notion of parametricity has been the object of a deep investigation, since [Rey84] (see also [ACC93] and [LMS93] for recent investigations). In the present approach we use the functional expressiveness of λ-calculus and extend it by some relevant features of a broadly used programming style: Object-Oriented Programming. Indeed, the core notion we want to focus on, yields a form of "dependency on input types" (or "on the types of the inputs").

The basis of this work has been laid in [CGL92], were the main syntactical properties of the system below have been shown. In this paper, we investigate an *elementary* approach to its mathematical meaning. A more general (categorical) understanding of what we mean by "dependency on input types" should be a matter of further investigation, possibly on the grounds of the concrete construction below. Indeed, our model provides an understanding of a slightly modified version of the system in [CGL92], as we focus on "early binding" (see the discussion below).

In order to motivate our extended λ-calculus, we first survey the key features in Object-Oriented programming which inspired our work, then, in section 2, we summarize the system presented in [CGL92] and develop some further syntactic properties, instrumental to our semantic approach. Section 3, introduces the variant with "early binding". Section 4 presents the model.

1.1 Abstract view of object-oriented paradigms

Object-oriented programs are built around *objects*. An object is a programming unit that associates data with the operations that can use or affect these data. These operations are called *methods*; the data they affect are the *instance variables* of the object. In short an object is a programming unit formed by a data structure and a group of procedures that affect it. The instance variables of an object are private to the object itself; they can be accessed only through the methods of the object. An object can only respond to *messages* which are *sent* or *passed* to it. A *message* is simply the name of a method that was designed for that object.

Message passing is the key feature of object-oriented programming. As a matter of facts, every object-oriented program consists in a set of objects which interact by exchanging messages. Since objects are units that associate data and procedures, they are very similar to instances of abstract data types and, thus, message passing may be viewed as the application of a function defined on the abstract data type. However, message passing is a peculiar and specific feature as we show in the next section.

1.1.1 Message passing

Every language has its own syntax for messages. We use the following one:

$$message \bullet receiver$$

The *receiver* is an object (or more generally an expression returning an object); when it receives a message, the run-time system selects among the methods defined for that object the one whose name corresponds to passed message; the existence of such a method should be statically checked (i.e. verified at compile time) by a type checking algorithm.

There are two ways to understand message passing.

The first way roughly consists in considering an object as a record whose fields contain the methods defined for the object, and whose labels are the messages of the corresponding methods[1]. This interpretation is the basis of the modeling of the objects as proposed in [Car88] and known as the "objects as records" analogy. This modeling has been adopted by a large number of authors (see [Bru91, BL90, CL91, CW85, CCH+89, CHC90, CMMS91, Ghe91a, Mey88, Pie93, PT93, Rém89, Wan87, Wan91]).

The second way to intepret message passing is the one used in the language CLOS (see [Kee89]), introduced in the context of typed functional languages in [Ghe91b], where message passing is functional application in which the message is (the identifier of) the function and the receiver is its argument. We have based our modeling of object-oriented programming on this intuition, and for this reason our work is alternative and somewhat detached from the type-driven research done so far in object-oriented programming. However, in order to formalize this approach, ordinary functions (e.g. lambda-abstraction and application) do not suffice. Indeed the fact that a method *belongs* to a specific object implies that the meaning of message passing is different from the one of the custom function application. The main characteristics that distinguish methods from functions are the following:

- *Overloading*: Two objects can respond differently to the same message. For instance, the code executed when sending a message `inverse` to an object representing a matrix will be different from the one executed when the same message is sent to an object representing a real number. Though the same message behaves uniformly on objects of the same kind (e.g. on all objects of *class* `matrix`). This feature is known as *overloading* since we overload the same operator (in this case `inverse`) by different operations; the actual operation depends on the type of the operands. Thus *messages are identifiers of overloaded functions*, in message passing the *receiver* is the first argument of an overloaded function, and the selection of the code to be executed depends on the type of that argument. Each method constitutes a branch of the overloaded function referred by the message which the method is associated with.

- *Late Binding*: The second crucial difference between function applications and message passing is that a function is bound to its meaning at compile time while the meaning of a method can be decided only at run-time when the receiving object is known. This feature, called *late-binding*, is one of the most powerful characteristics of object-oriented programming (see the excursus below). In our approach, it will show up in the the combination between overloading and subtyping. Indeed we define on types a partial order which concerns the utilization of values: a value of a certain type can be used wherever a value of a supertype is required. In this case the exact type of the receiver cannot be decided at compile time since it may change (notably decrease) during computation. For example, suppose that a graphic editor is coded using an object-oriented style, defining *Line* and *Square* types as subtypes of *Picture* with a method *draw* defined on all of them, and suppose that x is a formal parameter of a function, with type *Picture*. If the compile time type of the argument is used for branch selection (early binding) an overloaded function application like the following one

  $$\lambda x^{Picture}.(\ldots(draw \bullet x)\ldots)$$

  is always executed using the *draw* code for *pictures*. Using late binding, each time the whole function is applied, the code for *draw* is chosen only when the x parameter has been bound and

[1] This is just a way to understand message passing, and not a way to implement it; as a matter of fact, in real implementations, methods are not searched in the object but in its *class*

evaluated, on the basis of the run-time type of x, i.e. according to whether x is bound to a line or to a square.

Therefore in our model overloading with late binding is the basic mechanism.

Excursus (late vs. dynamic binding) *Overloaded operators can be associated with a specific operation using either "early binding" or "late binding". This distinction applies to languages where the type which is associated at compile time with an expression can be different (less informative) from the type of the corresponding value, at run time. The example above with* Line *and* Picture *should be clarifying enough. Note though that what here we call* late *binding, in object-oriented languages is usually referred as* dynamic *binding (see for example [Mey88, NeX91]). Late and dynamic binding (or "dynamic scoping") are yet two distinct notions. Early vs. late binding has to do with overloading resolution, while static vs. dynamic binding means that a name is connected to its meaning using a static or a dynamic scope. However this mismatch is only apparent, and it is due to the change of perspective between our approach and the one of the languages cited above: in [Mey88] and [NeX91], for example, the suggested understanding is that a message identifies a method, and the method (i.e. the meaning of the message) is* dynamically *connected to the message; in our approach a message identifies an overloaded function (thus a set of methods) and it will always identify this function (thus it is statically bounded) but the selection of the branch is performed by late binding.*

The situation is actually more complex. As a matter of fact, messages obey to an intermediate scoping rule: they have a "dynamically extensible" meaning. If the type Picture *is defined with the method* draw*, then the meaning of the* draw *method is fixed for any object of type picture, like what it happens with static binding. However, if later a new type* Circle *is added to the graphic editor, the set of possible meanings for the* draw *message is dynamically extended by the method for* Circle *and the function in the previous example will use the correct method for* Circle*, even if circles did not exist when the function was defined. This combination of late binding and dynamic extensibility is one of the keys of the high reusability of object-oriented languages. Essentially, these languages allow one to extend an application by simply* adding *a subclass of an existing class, while in traditional languages one usually also needs* modifying *the old code, which is a costlier operation.*

The use of overloading with late-binding automatically introduces a further distinction between message passing and ordinary functions. As a matter of fact, overloading with late-binding requires a restriction in the evaluation technique of arguments: while ordinary function application can be dealt with by either call-by-value or call-by-name, overloaded application with late binding can be evaluated only when the run-time type of the argument is known, i.e. when the argument is fully evaluated (closed and in normal form). In view of our analogy "messages as overloaded functions" this corresponds to say that message passing (i.e. overloaded application) acts by call-by-value or, more generally, only closed and normal terms respond to messages.

Thus to start a formal study on the base of this intuition, we have defined in [CGL92] an extension of the typed lambda calculus that could model these features. We will not give here a detailed description of the calculus, since the reader can find it in the papers cited above. We just recall the underlying intuition and the formal definitions of this calculus.

2 The $\lambda\&$-calculus

2.1 Overloaded types and terms

An overloaded function is constituted by a set of ordinary functions (i.e. lambda-abstractions), each one forming a different branch. To glue together these functions in an overloaded one we have chosen

the symbol &; thus we have added to the simply typed lambda calculus the term

$$(M\&N)$$

which intuitively denotes an overloaded function of two branches, M and N, that will be selected according to the type of the argument. We must distinguish ordinary application from the application of an overloaded function since, as we tried to explain in the previous section, they constitute different mechanism. Thus we use "$\bullet$" to denote the overloaded application and $\cdot$ for the usual one. Overloaded functions are built as it is customary with lists, by starting with an *empty* overloaded function that we denote by ε, and by concatenating new branches by means of &. Thus in the term above M is an overloaded function while N is a regular function, which we call a "branch" of the resulting overloaded function. Therefore an overloaded function with n branches $M_1, M_2, \ldots M_n$ can be written as

$$((\ldots((\varepsilon\&M_1)\&M_2)\ldots)\&M_n)$$

The type of an overloaded function is the ordered set of the types of its branches.[2] Thus if $M_i\colon U_i \rightarrow V_i$ then the overloaded function above has type

$$\{U_1 \rightarrow V_1, U_2 \rightarrow V_2, \ldots, U_n \rightarrow V_n\}$$

and if we pass to this function an argument N of type U_j then the selected branch will be M_j. That is:

$$(\varepsilon\&M_1\&\ldots\&M_n)\bullet N \ \triangleright^* \ M_j \cdot N \tag{1}$$

We have also a subtyping relation on types. Its intuitive meaning is that $U \leq V$ iff any expression of U can be safely used in the place of an expression of V. An overloaded function can be used in the place of another when for each branch of the latter there is one branch in the former that can substitute it; thus, an overloaded type U is smaller than another overloaded type V iff for any arrow type in V there is at least one smaller arrow type in U.

Due to subtyping, the type of N in the expression above may not match any of the U_i but it may be a subtype of some of them. In this case we choose the branch whose U_i "best approximates" the type, say, U of N; i.e. we select the branch z s.t. $U_z = \min\{U_i | U \leq U_i\}$.

It is important to notice that, because of subtyping, in this system types evolve during computation. This reflects the fact that, in languages with subtypes, the run-time types of the values of an expression are not necessarily equal to its compile time type, but are always subtypes of that compile time type. In the same way, in this system, the types of all the reducts of an expression are always smaller than or equal to the type of the expression itself.

In our system, not every set of arrow types can be considered an overloaded type. A set of arrow types is an overloaded type iff it satisfies these two conditions:

$$U_i \leq U_j \quad \Rightarrow \quad V_i \leq V_j \tag{2}$$

$$U_i \Downarrow U_j \quad \Rightarrow \quad \text{there exists a unique } z \in I \text{ such that } U_z = \inf\{U_i, U_j\} \tag{3}$$

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 (2) 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 $\{U_1 \rightarrow V_1, U_2 \rightarrow V_2\}$ with $U_2 < U_1$ and we pass it a term N which at compile-time has type U_1 then the compile-time type of $M\bullet N$ will be V_1; but if the normal form of N has type U_2 then the run-time type of $M\bullet N$ 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

[2] This is just a first approximation; see later for the exact meaning of overloaded types.

of type $\{U_i \rightarrow V_i\}_{i\in I}$ to a term of type U then the selected branch has type $U_j \rightarrow V_j$ such that $U_j = \min_{i\in I}\{U_i | U \leq U_i\}$; condition (3) assures the existence and uniqueness of this branch.[3] While condition (2) is essential to avoid run-time type errors, condition (3) is just the simplest way to deal with the multiple inheritance problem, that is the problem of selecting one branch when many possible choices have been defined; adopting a different solution for this problem would not change the essence of the approach (see [Ghe91b], where different choices are discussed).

Finally we have to include *call-by-value* and *late-binding*. This can simply be done by requiring that a reduction as (1) can be performed only if N is a closed normal form, and that the chosen branch depends on the type of the reduced term. This is late-binding since the branch choice cannot be performed before evaluating the argument, and this choice does not depend on the compile-time type of the expression which generated the value, but on the run-time type of the value itself. The formal description of the calculus is given by the following definitions:

Pretypes

$$V ::= A \,|\, V \rightarrow V \,|\, \{V'_1 \rightarrow V''_1, \ldots, V'_n \rightarrow V''_n\}$$

Subtyping

The subtyping preorder relation is predefined as a partial lattice[4] on atomic types, and it is extended to higher pretypes in the following way:

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{U_1 \rightarrow V_1 \leq U_2 \rightarrow V_2} \qquad \frac{\forall i \in I, \exists j \in J \qquad U'_j \rightarrow V'_j \leq U''_i \rightarrow V''_i}{\{U'_j \rightarrow V'_j\}_{j\in J} \leq \{U''_i \rightarrow V''_i\}_{i\in I}}$$

Since the set (A,$\leq$) of atomic types is a partial lattice, the set of all types can be easily shown to be a partial lattice too.

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)** $(U_i \Downarrow U_j \Rightarrow \exists! h \in I \qquad U_h \in^5 \inf\{U_i, U_j\})$

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

Terms (where V is a type)

$$M ::= x^V \mid \lambda x^V . M \mid M{:}M \mid \varepsilon \mid M \&^V M \mid M \bullet M$$

Type-checking Rules

[Taut] $\qquad\qquad x^V : V$

[3] By the way note how these conditions are very related to the regularity condition discussed in [GM89], in the quite different framework of order-sorted algebras and order-sorted rewriting systems

[4] i.e. it must satisfy the following constraints: $A \Downarrow A' \Rightarrow \inf\{A, A'\} \neq \emptyset$ and $A \Uparrow A' \Rightarrow \sup\{A, A'\} \neq \emptyset$

[5] Since $\leq$ is not irreflexive, hence the g.l.b. of two types is generally not a unique type but a set of types; see Section 3.1.

$[\to \textsc{Intro}]$ $$\frac{M:V}{\lambda x^U.M:U\to V}$$

$[\to \textsc{Elim}_{(\leq)}]$ $$\frac{M:U\to V \quad N:W\leq U}{MN:V}$$

$[\textsc{Taut}_\varepsilon]$ $$\varepsilon:\{\}$$

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

$[\{\}\textsc{Elim}]$ $$\frac{M:\{U_i\to V_i\}_{i\leq n} \quad N:U \quad U_j=\min_{i\leq n}\{U_i|U\leq U_i\}}{M\bullet N:V_j}$$

In this paper $\{U_i\to V_i\}_{i\leq n}$ is a meta notation for $\{U_1\to V_1\ldots U_n\to V_n\}$.

Reduction

The reduction $\triangleright$ is the compatible closure of the following notion of reduction:

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

$\beta_\&$) If $N:U$ is closed and in normal form, $U_j=min_{i=1..n}\{U_i|U\leq U_i\}$ and $(M_1\&M_2):\{U_i\to V_i\}_{i=1..n}$ then

$$(M_1\&M_2)\bullet N \triangleright \begin{cases} M_1\bullet N & \text{for } j<n \\ M_2\cdot N & \text{for } j=n \end{cases}$$

Main Theorems

For this calculus we have proved the following fundamental theorems (see [CGL92]):

- *Subsumption Elimination*: The language admits both a presentation with the subsumption rule ($M:W$ and $W\leq U \Rightarrow M:U$) and an equivalent subsumption-free presentation (the one chosen here).
- *Transitivity Elimination*: Adding transitivity does not modify the subtyping relation.
- *Type Uniqueness*: Every well-typed term possesses a unique type.
- *Generalized Subject Reduction*: Let $M:U$. If $M \triangleright^* N$ then $N:U'$, where $U'\leq U$.
- *Church-Rosser*: Equal terms possess a common reductum.

2.2 The stratified system

The $\lambda\&$-calculus many interesting properties, some were listed at the end of the previous section. However it does not normalize. The reasons are explained in details in [CGL92]; we only observe here that its expressive power allows (a restricted form of) self application. Namely, a suitably overloaded variable may be applied to itself, by overloaded application. This may eventually increase the computational power of this λ-calculus, but it has no clear object-oriented meaning (that is why in [CGL92] a normalizing variant was proposed and investigated). More precisely, the point is that subtyping does not respect the size of the types: in most systems is two types are in subtyping relation then their syntax tree are "similar" (for example, for record types one just erases some branches at some nodes) . In $\lambda\&$ this "stratification" of the subtyping relation is not respected. In particular the syntax tree of a given type may be a subtree of the syntax tree of a subtype of that type. Consider, say, the empty overloaded type: $\{\}$. Then, for any T, $\{\ \{\} \to T\ \} \leq \{\}$ and $\{\}$ is a syntactic occurrence of $\{\ \{\} \to T\ \}$. A function of type $\{\ \{\} \to T\ \}$ accepts arguments of type $\{\}$, but also arguments of any type smaller than $\{\}$ and thus also arguments of its own type; whence the self-application.

Thus no "stratification" is possible, as in the typed λ-calculus, where all well-typed terms normalize. The idea then is to define a suitable notion of rank for types and allow application only in presence of decreasing ranks. Moreover we want that types are preserved (or, at most, reduced[6]) during computation: namely, we are interested in systems that satisfy (a generalized form of) the Subject Reduction Theorem (such as $\lambda\&$ in [CGL92]). Formally:

Definition 2.1 A subsystem $\lambda\&^-$ of $\lambda\&$ closed by Subject Reduction and with associated *rank* function from $\lambda\&^-$ types to integers (write $(\lambda\&^-, rank)$), is *stratified* if:

1. If U (syntactically) occurs in T, then $rank(U) \leq rank(T)$;
2. If $M^T N^U$ is well typed, then $rank(U) < rank(T)$.

□

Theorem 2.2 *[CGL92] Let* $(\lambda\&^-, \mathrm{rank})$ *be a stratified subsystem of* $\lambda\&$*. Then* $\lambda\&^-$ *is Strongly Normalizing.*

Remark 2.3 *We observe here that the proof in [CGL92] shows a more general fact than stated. Indeed, the argument works for any stratified system* $(\lambda\&^-, \mathrm{rank})$*, not necessarily a subsystem of* $\lambda\&$*, such that:*

- $\lambda\&^-$ *has the same terms, pretypes and type formation rules as* $\lambda\&$*;*
- *type formation and subtyping yield the Subject Reduction property (in the general sense above).*

A variant of $\lambda\&$ *with the required properties as a stratified* $(\lambda\&^-, \mathrm{rank})$ *may be obtained by the following simple modification of the subtyping rules.*

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{U_1 \to V_1 \leq U_2 \to V_2} \qquad rank(U_1 \to V_1) \leq rank(U_2 \to V_2)$$

$$\frac{\forall i \in I, \exists j \in J \qquad U'_j \to V'_j \leq U''_i \to V''_i}{\{U'_j \to V'_j\}_{j\in J} \leq \{U''_i \to V''_i\}_{i\in I}} \; rank(\{U'_j \to V'_j\}_{j\in J}) \leq rank(\{U''_i \to V''_i\}_{i\in I})$$

In any well typed overloaded application $M^{\{T_i \to U_i\}_{i\in I}} \bullet N^{T'}$*, the rank of* T' *is then smaller than the rank of some* T_i*, hence it is strictly smaller than the rank of* $\{T_i \to U_i\}_{i\in I}$*; similarly for functional application. The examples below discuss in more details two cases where these rules are admissible or even derivable.*

[6] According to the subtyping order

As a matter of example in [CGL92] the following two calculi were considered.

Example 2.4 In the examples below the properties of system $\lambda\&^-$ in 2.1 are obtained either by restricting the set of types ($\lambda\&^-_{\mathrm{T}}$), or by imposing a stricter subtyping relation ($\lambda\&^-_{\leq}$). In either case, the *rank* function is defined as follows:

$$\begin{array}{lcl} rank(\{\}) & = & 0 \\ rank(A) & = & 0 \\ rank(T \to U) & = & \max\{rank(T)+1, rank(U)\} \\ rank(\{T_i \to U_i\}_{i\in I}) & = & \max_{i\in I}\{rank(T_i \to U_i)\} \end{array}$$

As it should be clear by now, the idea is that the types of a function and of its arguments are "stratified" in a way that the rank of the functional type is strictly greater than the rank of the input type, as required by definition 2.1.

- $\lambda\&^-_{\leq}$ is defined by substituting $\leq$ in all $\lambda\&$ rules with a stricter subtyping relation $\leq_m$ defined by adding to any subtyping rule which proves $T \leq U$ the further condition $rank(T) \leq rank(U)$, see remark 2.3. The subject reduction proof for $\lambda\&$ works for $\lambda\&^-_{\leq}$ too, thanks to the transitivity of the $\leq_m$ relation.[7]
- $\lambda\&^-_{\mathrm{T}}$ is defined by imposing, on overloaded types $\{T_i \to U_i\}_{i\in I}$, the restriction that the ranks of all the branch types $T_i \to U_i$ are equal, and by stipulating that $\{\}$ is not a supertype of any non-empty overloaded type (see the previous footnote). Then we can prove inductively that, whenever $T \leq U$, then $rank(T) = rank(U)$, and that $\lambda\&^-_{\mathrm{T}}$ is a subsystem of $\lambda\&^-_{\leq}$. To prove the closure under reduction (i.e., that $\lambda\&^-_{\mathrm{T}}$ terms reduce to $\lambda\&^-_{\mathrm{T}}$ terms), observe first that a $\lambda\&$ term is also a $\lambda\&^-_{\mathrm{T}}$ term iff all the overloaded types appearing in the indexes of variables and of &'s are $\lambda\&^-_{\mathrm{T}}$ overloaded types (this is easily shown by induction on typing rules). The closure by reduction follows immediately, since variables and &'s indexes are never created by a reduction step.

Note that $\lambda\&^-_{\mathrm{T}}$ is already expressive enough to model object-oriented programming, where all methods always have the same rank (rank 1), and that $\lambda\&^-_{\leq}$ is even more expressive than $\lambda\&^-_{\mathrm{T}}$. □

Theorem 2.2 and the examples show that there exist (sub)systems of $\lambda\&$ which are strongly normalizing and expressive enough for the purposes of modeling object-oriented programming. In [CGL92] and section 2 we preferred to adopt the whole $\lambda\&$ as target system, since it is easier to establish results such as Subject Reduction and Confluence on the general system and apply them in subsystems rather than trying to extend restricted versions to more general cases.

3 Early Binding

We have presented in our introduction a limited overview of object-oriented languages. These languages are characterized by an interplay of many features, namely encapsulation, overloading, late-binding, dynamic-binding, subtyping and inheritance. We have selected three of them—overloading, late binding and subtyping—that, in our opinion, suffice to model the relevant features of a class-based object-oriented language. Not that these features are exclusive to this approach: overloading existed long before object-oriented languages (FORTRAN already used it) while subtyping, even if it was first suggest by object-oriented paradigms, has been included in other different paradigms (e.g.

[7] Note that, in this system, $\{\}$ is not a supertype of any non-empty overloaded type; this is not a problem, since the empty overloaded type is only used to type ε, which is needed to start overloaded function construction. However, we may alternatively define a family of empty types $\{\}_{i\in\omega}$, each being the maximum overloaded type of the corresponding rank, and a correspondent family of empty functions $\varepsilon_{i\in\omega}$.

EQLOG [GM85], LIFE [AKP91] or Quest [CL91]). But their combination is peculiar to object-oriented programming. And exactly the interplay of all these features makes the object-oriented approach so useful in the large-scale software production.

In our system we do not deal with encapsulation and dynamic-binding, but we think they can be obtained using existential and reference types (even if, admittedly, there is still much to be done about the combination of existential types and object-oriented programming), while our calculus accounts for many other features although we had not the occasion to show it in this paper (see [Cas92]).

At the semantic level, our system presents three main technical challenges. The first is the true dependence of overloaded functions from types. The second is the fact that subtyping is not an order relation. The third is the distinction between run-time types and compile-time types. In the present paper we just concentrate on the first two aspects, which already requires some technical efforts, while we will avoid the third problem by taking into consideration only a subsystem of full $\lambda\&$ where the type of the arguments of overloaded functions is "frozen", i.e. is the same at compile-time and at run-time. This is just a first step in the direction of defining a semantics for the full system.

The resulting system is someway intermediate between late-binding and early-binding overloading. It features early-binding, since for any application of an overloaded function the type which will be used to perform branch selection is already known at compile-time, as happens for example with arithmetic operators in imperative languages. It has still a form of late-binding since, as overloaded functions are first class values which can be the result of expression evaluation, it is not possible to get rid of branch selection at run time. For example, if a function applies a formal parameter x, which has the type of an overloaded function, to an argument whose type is U, even if we know that branch selection will be based on U, branch selection cannot be statically performed since the function associated with x is unknown.

If overloaded functions were not first-class (i.e. if no variable were allowed to possess an overloaded type) this restricted calculus would correspond to the "classical" (i.e. with early binding) implementation of overloading in imperative languages: the standard example is the operator + which is defined both on *reals* and *integers*, though a different code is used according to the type of the argument[8]. What happens in these languages is that a *preprocessor* scans *at compile time* the text of a program looking for all occurrences of + and it substitutes them by a call to the appropriated code, depending on whether they are applied to reals or integers[9]. As far as we know, all languages that use in an explicit way overloading (and not implicitly as it is done in object-oriented programming via the method definitions) base the selection of the code on the type possessed by the arguments at compile time.

We obtain this "half-way early-binding" restriction of our system simply by adding explicit coercions and imposing that every argument of an overloaded function is coerced. A coercion c_V is just an function which, informally, does nothing, but which cannot be reduced, so that the type of all the residuals of a term $c_V(M)$ is always V. Thus to model overloading with early binding we require that, for each overloaded application, a coercion freezes the type of the argument up to branch selection. We change the system in the following way: Pretypes, Types and Subtyping rules are as before. Terms are now:

$$M ::= x^V \mid \lambda x^V.M \mid MM \mid c_V(M) \mid \varepsilon \mid M\&M \mid M \bullet c_V(M)$$

We add to the rules of type-checking the one for coercions:

[8] This example is sometimes misleading because of the fact that the codes for the to branches must give the same results when applied to integers (being integer numbers a subset of real numbers); this extra property is proper to this example and it has nothing to do with overloading: in general the branches do not need to be related on the values they return

[9] The same is true for paradigms which have a cleverer use of overloading: for example, when the programmer can define its own overloaded operators. This is possible in Haskell; in this language the implementation of overloading is based on strong theoretical grounds as shown in [WB89]; in that paper it is also shown how the selection in overloading can be solved at compile time by the use of a preprocessor

[Coercion] $$\frac{M : U \leq V}{c_V(M) : V}$$

Finally we define the reduction on the coercion; the minimal modification required is the addition of this rule to our notion of reduction:

$$(coerce) \qquad c_V(M) \circ N \ \triangleright M \circ N$$

where $\circ$ denotes either $\cdot$ or $\bullet$. This rule is needed since otherwise coercions could prevent some β or $\beta_\&$ reductions. This rule does not interfere with our use of coercions, since it only allows us to reduce the left hand side, but not the right hand side, of an application. Another rule that could be added to the calculus is

$$c_V(c_U(M)) \ \triangleright c_V(M)$$

However it does not bring any interesting modification to the system, so that we prefer not to include it.

3.1 The completion of overloaded types

This section presents some general, syntactic, properties of (overloaded) types, which may be viewed as some sort of "preprocessing" on the syntactic structures and which provide by this an interface towards our semantic constructions.

Subtyping in our system (and in its variants in 2.2) is transitive but is not antisymmetric (it is only a *preorder* relation). Consider, for example, $\{U \to V\}$ and $\{U \to V, U' \to V'\}$. Clearly, $\{U \to V, U' \to V'\} \leq \{U \to V\}$. But the opposite inclusion is also possible here, in contrast to record types; namely, if $U \to V \leq U' \to V'$, one also has $\{U \to V, U' \to V'\} \geq \{U \to V\}$: "$\leq$" is not antisymmetric.

This preorder relation makes perfectly sense w.r.t. type-checking. Suppose that $M : \{U_i \to V_i\}_{i \in J} \leq \{U'_i \to V'_i\}_{i \in I}$. The intended meaning of this subtyping relation is that M can be fed with any input N which would be acceptable for a term M' in $\{U'_i \to V'_i\}_{i \in I}$, and that the output can be used in any context where $M' \bullet N$ would be accepted. Indeed, let $N : U'_i$ and $C[\,]$ be a context where a value of type V'_i can be put. Then, for some $j \in J$, $U_j \to V_j \leq U'_i \to V'_i$, so that $U'_i \leq U_j$ and $V_j \leq V'_i$, hence the application $M \bullet N$ type-checks and can be used in the context $C[M \bullet N]$. "$\leq$" in $\lambda\&$ is the least (or less fine) preorder that one can define with this property.

However, since we want to interpret "$\leq$" by an order relation among semantic types, in this section we look for a mechanism to get rid of irrelevant differences between equivalent types. The construction below may be given in $\lambda\&$ or in any stratified (sub-)system.

Definition 3.1 Given types U and V, set $U \sim V$ if $U \leq V$ and $V \leq U$. □

Remark 3.2 *If $\{U \to V\} \sim \{U \to V, U' \to V'\}$ then $U' \leq U$ and $V' \sim V$. Indeed, one must have $U \to V \leq U' \to V'$, so that $U' \leq U$ and $V \leq V'$, while $V' \leq V$ follows from $U' \leq U$ by covariance. This gives the intuitive meaning of the equivalence: a type $U' \to V'$ can be freely added or removed from an overloaded type if there is another type $U \to V$ which "subsumes" it, i.e. which is able to produce the same output type on a wider input type ($V \sim V'$ but $U \geq U'$).*

We now extend the usual definitions of g.l.b., l.u.b., etc., to a preorder relation. For any partial preorder $\leq$ defined on a set Y and for any $X \subseteq Y$ we define:

$$\begin{aligned} \min X &=_{def} \{U \in X | \forall V \in X. U \leq V\} \\ \max X &=_{def} \{U \in X | \forall V \in X. V \leq U\} \\ \inf X &=_{def} \max\{U \in Y | \forall V \in X. U \leq V\} \\ \sup X &=_{def} \min\{U \in Y | \forall V \in X. V \leq U\} \end{aligned}$$

Note that the four functions above denote a subset of Y, which in the first two cases, if not empty, is an element of $X/\sim$, and in the last two cases an element of $Y/\sim$.

Our next step is now the definition of the "completion" of overloaded types; intuitively, the completion of an overloaded type is formed by adding all the "subsumed types" (in the sense of remark 3.2), so that two equivalent overloaded types should be transformed, by completion, in *essentially* the same completed type. For this purpose and for the purpose of their semantics, we now adopt a different notation for overloaded types. Write $\Downarrow H$, if the collection H of types has a lower bound.

Definition 3.3 [g.o.t.] A *general overloaded type* (g.o.t.) is a pair (K, out) where K is a set of types, i.e. a subset of Type, and out is a function from K to Type such that:

1. $H \subseteq K$ and $\Downarrow H$ imply that there is $V \in K$ such that $V \in \inf H$.
2. out is monotone w.r.t. the subtype preorder.

We may write $\{U \rightarrow out(U)\}_{U \in K}$ for a general overloaded type. □

Notice that V at the point 1 of the definition is not required to be unique, and also that K is not required to be finite. Thus a g.o.t. is *not* a type. But any overloaded type can be seen as a g.o.t. (K, out), with a finite K.

The preorder on g.o.t.'s is the one defined by applying to g.o.t.'s the rules given in the previous section.

We are now ready to define the notion of completion. We complete a g.o.t. by enlarging its domain to its downward closure and by extending the "out" map to the enlarged domain. The extended map $\widehat{out}$ is defined over a type U', essentially, by setting

$$\widehat{out}(U') = out(\min\{U \in K | U' \leq U\}).$$

But recall that min denotes a set of types; thus we have to choose one of them. To this aim, we suppose that a choice function *choose* is defined which chooses a type out of a non-empty set of equivalent types. Then, the extended map can be defined as:

$$\widehat{out}(U') = out(choose(\min\{U \in K | U' \leq U\})).$$

For brevity, we will denote the functional composition of *choose* and min as *a_min*:

$$a\_min(X) =_{def} choose(\min(X))$$

Remark 3.4 *Even if $\leq$ is a preorder on* **Types**, *in the rule* [{ }ELIM] *there is not ambiguity in the selection of the minimum. Indeed, by the definition of good formation of (overloaded) types we required the property*

$$\textbf{(c)} \qquad (U_i \Downarrow U_j \Rightarrow \exists! h \in I \qquad U_h \in \inf\{U_i, U_j\})$$

Thus the rule picks up the unique *U_j with the required property. For the same reason, when the g.o.t. which is the argument of completion (see below) is actually a type, the argument of the choose function is just a singleton.*

Definition 3.5 [completion] Let $\{U \rightarrow out(U)\}_{U \in K}$ be a g.o.t.. Its *completion* $\{U \rightarrow \widehat{out}(U)\}_{U \in \widehat{K}}$ is the g.o.t. given by: $\widehat{K} = \{U' | \exists U \in K \;\; U' \leq U\}$ and $\widehat{out}(U') = out(a\_min\{U \in K | U' \leq U\})$. □

Fact 3.6 *The completion of a g.o.t. $\{U \rightarrow out(U)\}_{U \in K}$ is a well defined g.o.t..*

Proof. Recall first that Type is a partial lattice: this gives 1 in 3.3. As for 2 ($\widehat{out}$ monotonicity), let $U' \leq V'$ be two types such that out is defined on both of them. Both $U'' = a\_min\{U \in K | U \geq U'\}$ and $V'' = a\_min\{U \in K | U \geq V'\}$ are well defined by 1; moreover, $V'' \in K$, $V'' \geq V'$ and $V' \geq U'$ imply that $V'' \in \{U \in K | U \geq U'\}$, and then that $U'' \leq V''$, so that $\widehat{out}(U') = out(U'') \leq out(V'') = \widehat{out}(V')$. □

Remark 3.7 *In the completion of a g.o.t.* $\{U \rightarrow out(U)\}_{U \in K}$, *if* $U \in K$, *then* $\widehat{out}(U) \sim out(U)$, *since* $a\_min\{V \in K | U \leq V\} \sim U$ *and out is monotone.*

Clearly, the completion is an idempotent operation (modulo equivalence). Note also that, even for a singleton $K = \{U\}$, $\widehat{K}$ may be infinite (e.g. U equal to the type of ε i.e. $\{\ \}$).

Fact 3.8 *By completion, one obtains an equivalent g.o.t., that is:*

$$\{U \rightarrow out(U)\}_{U \in K} \sim \{U \rightarrow \widehat{out}(U)\}_{U \in \widehat{K}}$$

Proof. "$\geq$": we have to prove that

$$\forall U \in K. \exists U' \in \widehat{K}. U \rightarrow out(U) \geq U' \rightarrow \widehat{out}(U').$$

Take $U' = U$. By remark 3.7, since $U \in K$, $\widehat{out}(U) \sim out(U)$, hence $U \rightarrow out(U) \sim U \rightarrow \widehat{out}(U)$.

Conversely, we have to prove that

$$\forall U' \in \widehat{K}. \exists U \in K. U \rightarrow out(U) \leq U' \rightarrow \widehat{out}(U').$$

For $U' \in \widehat{K}$, $\exists U \in K. U' \leq U$; thus, let $Z = a\_min\{V \in K | V \geq U'\}$, one has $Z \rightarrow out(Z) \leq U' \rightarrow \widehat{out}(U')$, since by definition of completion $\widehat{out}(U') = out(Z)$. □

The idea is to interpret overloaded types by using their completions, in the model. However, as some preliminary facts may be stated at the syntactic level, we preferred to define syntactic completions and work out their properties. The theorem 3.9 below, is the most important one, since it guarantees the monotonicity of completion. Note that subtyping between overloaded types is contravariant w.r.t. the collections $\widehat{K}$ and $\widehat{H}$. The reader familiar with the semantics of records as indexed products (see [BL90]) may observe analogy with that contravariant understanding of records. Indeed in [CGL92] we showed that record types may be coded as particular overloaded types.

Theorem 3.9 *Let* (K, out) *and* (H, out') *be g.o.t.. Then*

$$\{U \rightarrow out'(U)\}_{U \in H} \leq \{U \rightarrow out(U)\}_{U \in K} \Leftrightarrow \widehat{K} \subseteq \widehat{H} \text{ and } \forall U \in \widehat{K}.\ U \rightarrow \widehat{out'}(U) \leq U \rightarrow \widehat{out}(U)$$

Proof. ($\Rightarrow$) As for $\widehat{K} \subseteq \widehat{H}$, just observe that, $\forall U \in \widehat{K}$, $\exists U' \in K, U \leq U'$; hence $\exists V \in H. U' \leq V$, by the assumption, and, thus, $U \in \widehat{H}$. Let now $U \in \widehat{K}$. There exists then $V \in K$ such that $U \leq V$: take $V = a\_min\{W \in K | U \leq W\}$. As $V \in K$, by the assumption one has:

$$\exists U' \in H. V \leq U' \text{ and } out'(U') \leq out(V) \tag{4}$$

Now,

$$\widehat{out}(U) = out(a\_min\{W \in K | U \leq W\}) = out(V) \tag{5}$$

Thus:

$$\begin{array}{rcll} \widehat{out'}(U) & \leq & \widehat{out'}(U') & \text{since } U \leq V \leq U' \text{ and } U, U' \in \widehat{H} \\ & \sim & out'(U') & \text{by remark 3.7, since } U' \in H \\ & \leq & out(V) & \text{by (4)} \\ & = & \widehat{out}(U) & \text{by (5)} \end{array}$$

In conclusion, $\forall U \in \widehat{K}. U \rightarrow \widehat{out'}(U) \leq U \rightarrow \widehat{out}(U)$.

($\Leftarrow$) We have to prove that $\forall U \in K. \exists V \in H. V \geq U$ and $out'(V) \leq out(U)$. Let $U \in K$. By hyp., $U \in \widehat{H}$. Hence, $V = a\_min\{Z \in H | U \leq Z\}$ is well defined. Thus:

$$\begin{array}{rcll} out'(V) & = & \widehat{out'}(U) & \text{by definition} \\ & \leq & \widehat{out}(U) & \text{hypothesis} \\ & \sim & out(U) & \text{by remark 3.7, since } U \in K \end{array}$$

□

Corollary 3.10 *Let (K, out) and (H, out') be g.o.t.; then:*

$$\{U \to out'(U)\}_{U \in H} \sim \{U \to out(U)\}_{U \in K} \Leftrightarrow \widehat{K} = \widehat{H} \text{ and } \forall U \in \widehat{K}.\widehat{out'}(U) \sim \widehat{out}(U)$$

In conclusion, completions are not exactly canonical representatives of equivalence classes, but at least they push the differences between two overloaded types one level inside the types. In this way in the interpretation of types we will be able to get rid of the differences between equivalent types by iterating completion at all the levels inside the type structure. The fact that a type is equivalent to its completion makes it clear that an overloaded type, seen modulo $\sim$, does not describe the structure of the corresponding functions (e.g. how many different branches they have) but just which are the contexts where they can be inserted. Hence we understand overloaded types as "type-checkers", used to check the "dimension" of programs, similarly as in Physics where, by a "dimensional analysis", one checks that in an equation, say, a force faces a force etc.. This will be the crucial semantic difference between arrow types and overloaded types, since arrow types will keep their usual, more restrictive, meaning as "collection of functions or morphisms identified by the input and output types" (see section 4.1.3 for further discussions).

4 Semantics

4.1 PER as a model

In this section we give the basic structural ideas which will allow us to interpret the syntax of $\lambda\&$-*early*. Namely, we state which geometric or algebraic structures may interpret arrow and overloaded types; terms will be their elements and will be interpreted in full details in section 4.1.5.

A general model theory of $\lambda\&$-*early* may be worth pursuing as an interesting development on the grounds of the concrete model below. Indeed, by some general categorical tools, one may even avoid to start with a model of type-free lambda calculus, but this may require some technicalities from Category Theory (see [AL91]). Thus we use here a model $(\mathcal{D}, \cdot)$ of type-free lambda calculus and a fundamental type structure out of it. We survey first the basic ideas for the construction. Later we specialize the general construction by starting out of a specific type-free model which will yield a semantic for our typed calculus.

4.1.1 PER out of $(\mathcal{D}, \cdot)$

Let $(\mathcal{D}, \cdot)$ be an applicative structure, which yields a model of type-free lambda calculus (see [Bar84]).

Example 4.1 Let $\mathcal{P}\omega$ be the powerset of the natural numbers, ω. $\mathcal{P}\omega$ may be turned into an applicative structure $(\mathcal{P}\omega, \cdot)$, indeed a model of type-free λ-calculus, by setting:

$$a \cdot b = \{k | \exists e_h \subseteq b, \prec h, k \succ \in a\}$$

where a and b are elements of $\mathcal{P}\omega$, $\{e_i\}_{i \in \omega}$ is an enumeration of finite sets of numbers and $\prec \_, \_ \succ$ is a bijective coding of $\omega \times \omega$ into ω (see [Sco76] and also [Lon83] for a general set-theoretic construction). □

We first define the category of types as Partial Equivalence Relations out of $(\mathcal{D}, \cdot)$. When A is a symmetric and transitive relation on $\mathcal{D}$, we set, for $n, m \in \mathcal{D}$:

$$\begin{array}{rcl}
\mathbf{nAm} & \stackrel{def}{\Leftrightarrow} & n \text{ is related to } m \text{ by } A \\
\mathbf{dom(A)} & \stackrel{def}{=} & \{n | nAn\} \\
\lceil \mathbf{n} \rceil_{\mathbf{A}} & \stackrel{def}{=} & \{m | mAn\} \text{ (the equivalence class of } n \text{ with respect to } A) \\
\mathbf{Q(A)} & \stackrel{def}{=} & \{\lceil n \rceil_A | n \in dom(A)\} \text{ (the quotient set of } A).
\end{array}$$

Clearly, if A is a symmetric and transitive relation on $\mathcal{D}$ then A is an equivalence relation on $dom(A)$, as a subset of $\mathcal{D}$. (Note that, even if we will use n, m for arbitrary elements of $\mathcal{D}$, when $\mathcal{D}$ is $\mathcal{P}\omega$ each element n in $\mathcal{D}$ is actually a *set* of numbers).

Definition 4.2 The category **PER** (of Partial Equivalence Relations) is defined as:
- *objects*: $A \in \mathbf{PER}$ iff A is a symmetric and transitive relation on $\mathcal{D}$
- *morphisms*: $f \in \mathbf{PER}[A, B]$ iff $f : Q(A) \to Q(B)$ and $\exists n \in \mathcal{D}. \forall a \in dom(A). f(\lceil a \rceil_A) = \lceil n \cdot a \rceil_B$ □

Note that the morphisms in **PER** are computable, w.r.t. $(\cdot, \mathcal{D})$, in the sense that any $n \in \mathcal{D}$ such that $\forall a \in dom(A). f(\lceil a \rceil_A) = \lceil n \cdot a \rceil_B$ computes or **realizes** $f : Q(A) \to Q(B)$ in the definition (notation: $n \Vdash_{A \to B} f$). Thus **PER** is a category where the identity map, in each type, is computed by (at least) the interpretation of the term $\lambda x.x$, i.e. the identity function on $\mathcal{D}$.

Theorem 4.3 **PER** *is a CCC's.*

Proof.(Hint, the proof is in several papers since [Sco76]; in particular, in [AL91]). The exponent object $A \to B$ is defined by

$$\forall m, n.\ m(A \to B)n \quad \Leftrightarrow \quad \forall p, q(pAq \Rightarrow (m \cdot p)B(n \cdot q))$$

Products are defined by taking a coding of pairs of $\mathcal{D}$ into $\mathcal{D}$, as given for example by the fact that $\mathcal{D}$ is a model of type free lambda-calculus. □

To clarify the construction, let us look more in detail to exponent objects in **PER**. Take say $A \to B$, that is, the representative of $\mathbf{PER}[A, B]$. Then by definition each map $f \in \mathbf{PER}[A, B]$ is uniquely associated with the equivalence class of its realizers, $\lceil n \rceil_{A \to B} \in A \to B$ in the sense above. It should be clear that the notion of realizer, or "type-free computation" computing the typed function, is made possible by the underlying type-free universe, $(\mathcal{D}, \cdot)$. As we will discuss later, this gives mathematical meaning to the intended type-free computations of a typed program after compilation. In this context, it is common to identify, by an abuse of language, each typed function with the equivalence class of its realizers. Of course, the semantic "$\to$" gives meaning to arrow types.

Definition 4.4 The semantics of arrow types is given by $[\![U \to V]\!] = [\![U]\!] \to [\![V]\!]$ □

4.1.2 Subtyping

Before going into the semantics of the other types, we briefly introduce the meaning of "subtypes", in view of the relevance this notion has in our language. The semantics of subtypes over **PER** is given in terms of "subrelations", (see [BL90]).

Definition 4.5 [subtypes] Let $A, B \in \mathbf{PER}$. Define: $A \leq B$ iff $\forall n, m.(nAm \Rightarrow nBm)$ □

The intuition for this approach to subtyping is better understood when looking at "arrow types".

Proposition 4.6 *Let $A, A', B, B' \in \mathbf{PER}$ be such that $A' \leq A$ and $B \leq B'$. Then $A \to B \leq A' \to B'$. In particular, for $n \in dom(A \to B)$, one has $\lceil n \rceil_{A \to B} \subseteq \lceil n \rceil_{A' \to B'}$*

Proof.

$$\begin{array}{lcll} n(A \to B)m & \Leftrightarrow & \forall p, q.(pAq \Rightarrow n \cdot pBm \cdot q) & \\ & \Rightarrow & \forall p, q.(pA'q \Rightarrow n \cdot pB'm \cdot q) & \text{as } pA'q \Rightarrow pAq \Rightarrow n \cdot pBm \cdot q \Rightarrow n \cdot pB'm \cdot q \\ & \Leftrightarrow & n(A' \to B')m & \end{array}$$

The rest is obvious. □

The proposition gives the antimonotonicity of $\rightarrow$ in its first argument, as formalized in the rules and required by subtyping. Moreover, and more related to the specific nature of this interpretation of $\rightarrow$, this gives a nice interplay between the extensional meaning of programs and the intensional nature of the underlying structure, namely between functions and the set of indexes that compute them. Indeed, typed programs are interpreted as extensional functions in their types, as we identify each morphism in **PER** with the equivalence class of its realizers. That is, in the notation of the proposition, let $\lceil n \rceil_{A \rightarrow B} \in A \rightarrow B$ represent $f \in \mathbf{PER}[A, B]$ in the exponent object $A \rightarrow B$. Note then that the intended meaning of subtyping is that one should be able to run any program in $A \rightarrow B$ on terms of type A' also, as A' is included in A. When $n \Vdash_{A \rightarrow B} f$, this is exactly what $\lceil n \rceil_{A \rightarrow B} \subseteq \lceil n \rceil_{A' \rightarrow B'}$ expresses: any computation which realizes f in the underlying type-free universe actually computes f viewed in $A' \rightarrow B'$ also. Of course, there may be more programs for f in $A' \rightarrow B'$, in particular if A' is strictly smaller than A. This elegant interplay between the extensional collapse, which is the key step in the hereditary construction of the types as partial equivalence relations, and the intensional nature of computations is a fundamental feature of these realizability models. Clearly "$\leq$" is a partial order which turns the objects of **PER** into an algebraic complete lattice. The crucial point here is that "$\leq$" defines a refinement relation which goes exactly in the sense we want in order to interpret subtypes. Namely if $A \leq B$ then the equivalence class of A are included in those of B or A is finer than B.

Note finally that, given $n \in \mathbf{dom}(A)$ and $A \leq B$, we may view the passage from $\lceil n \rceil_A$ to $\lceil n \rceil_B$ as an obvious coercion.

Definition 4.7 [semantic coercions] Let $A, B \in \mathbf{PER}$ with $A \leq B$. Define $c_{AB} \in \mathbf{PER}[A, B]$ by $\forall n \in \mathbf{dom}(A)\ c_{AB}(\lceil n \rceil_A) = \lceil n \rceil_B$ $\Box$

Remark 4.8 *By the previous definition, for any $a \in Q(A)$, $c_{AB}(a) \supseteq a$*

For the sake of conciseness if U and V are syntactic types, we denote by c_{UV} the semantic coercion $c_{[\![U]\!][\![V]\!]}$

Note that semantic coercions do not do any work, as type-free computations, but, indeed, change the "type" of the argument, i.e. its equivalence class and the equivalence relation where it lives . Thus they are realized by the indexes of the type free identity map, among others, and they are meaningful maps in the typed structure.

Since terms will be interpreted as equivalence classes in (the meaning as p.e.r.'s of) their types, we need to explain what the application of an equivalence class to another equivalence class may mean, as, so far, we only understand the application "$\cdot$" between elements of the underlying type-free structure $(\mathcal{D}, \cdot)$.

Definition 4.9 [Application] Let A, A' and B be p.e.r.'s, with $A' \leq A$. Define then, for $n(A \rightarrow B)n$ and $mA'm$, $\lceil n \rceil_{A \rightarrow B} \cdot \lceil m \rceil_{A'} = \lceil n \cdot m \rceil_B$. $\Box$

Note that this is well defined, since $mA'm'$ implies mAm' and, thus, $n \cdot mBn' \cdot m'$, when $n(A \rightarrow B)n'$. This is clearly crucial for the interpretation of our "arrow elimination rule". We end this section on subtyping by two technical lemmas that will be heavily used in the next sections.

Lemma 4.10 (Monotonicity of application) *Let a, b, a', b' be equivalence classes such that the applications $a \cdot b$ and $a' \cdot b'$ are well defined (i.e. $a \in Q(A_1 \rightarrow A_2)$ and $b \in Q(B)$ with $B \leq A_1$, and similarly for a' and b'). If $a \subseteq a'$ and $b \subseteq b'$ then $a \cdot b \subseteq a' \cdot b'$*

Proof. $n \in a \cdot b \quad \Leftrightarrow \quad \exists p \in a, q \in b.\ n = p \cdot q \quad \Rightarrow \quad p \in a', q \in b' \quad \Rightarrow \quad n = p \cdot q \in a' \cdot b'$ $\Box$

Lemma 4.11 (Irrelevance of coercions) *Let A, A' and B be p.e.r.'s, with $A' \leq A$. Assume that $n(A \rightarrow B)n$ and $mA'm$. Then*
$\lceil n \rceil_{A \rightarrow B} \cdot c_{A'A}(\lceil m \rceil_{A'}) = \lceil n \rceil_{A \rightarrow B} \cdot \lceil m \rceil_A = \lceil n \cdot m \rceil_B = \lceil n \rceil_{A \rightarrow B} \cdot \lceil m \rceil_{A'} = c_{A \rightarrow BA' \rightarrow B}(\lceil n \rceil_{A \rightarrow B}) \cdot \lceil m \rceil_{A'}$.

Proof. Immediate $\Box$

4.1.3 Overloaded types as Products

The intuitive semantics of overloaded types is quite different from the meaning of arrow types. The essential difference is that types directly affect the computation: the output value of a $\beta_\&$ reduction explicitly depends on the type of the $(M_1 \& M_2)$ term, and on the type of the argument N.[10]

A second difference has to do with the ability of accepting as parameters values of any type which is a subtype of the input types explicitly specified. This fact is managed implicitly in arrow types. For example, let $M : U \to V$, then M will be interpreted as a function from the meaning of U to the meaning of V, as sets (or objects in a category of sets) and $U \to V$ will be interpreted as the collection of such functions. Robust structural properties of the model we propose will allow a function in $U \to V$ to be applied to elements of a subtype of U, as if they were in U. This kind of interpretation is not possible with overloaded types, at least since the set of acceptable input types has not, generally, a maximum.

Thus two crucial properties need to be described explicitly in the semantics of overloaded terms. First, output values depend on types; second, as a type may have infinitely many subtypes and the choice of the branch depends on "$\leq$", overloaded semantic functions explicitly depend on infinitely many types. By this, we will consider overloaded functions as, essentially, functions which take two parameters, one type and one value, and give a result whose type depend on the first parameter. Hence overloaded functions of, e.g., a type $\{U \to V, U' \to V\}$ will be elements of the indexed product (see later)

$$\prod_{W \leq U, U'} (W \to V).$$

In other words, the interpretation of $U \to V$ will be given by the usual set of functions from the interpretation of U to the interpretation of V (in a suitable categorical environment), while the meaning of $\{U \to V, U' \to V\}$ will directly take care of the possibility of applying overloaded functions to all the subtypes of the argument types.

The fact that the index in the product ranges over all subtypes of U, U', not just over $\{U, U'\}$, solves a third problem of overloaded types: the fact that subtyping is just a preorder, while its semantics interpretation is an order relation. By exploiting the notion of completion defined in section 3.1, we will be able to interpret all equivalent types as the same object.

We are now ready to be formal.

In set theory, given a set A and a function $G : A \to \mathbf{Set}$ ($\mathbf{Set}$ is the category of sets and set-theoretical maps), one defines the indexed product:

$$\bigotimes_{a \in A} G(a) = \{f \mid f : A \to \cup_{a \in A} G(a) \text{ and } \forall a \in A.\ f(a) \in G(a)\}$$

If A happens to be a subset of an applicative structure $(\cdot, \mathcal{D})$ and $G : A \to \mathbf{PER}$, then the resulting product may be viewed as a p.e.r. on $\mathcal{D}$, as follows.

Definition 4.12 Let $A \subseteq \mathcal{D}$ and $G : A \to \mathbf{PER}$. Define the p.e.r. $\prod_{a \in A} G(a)$ by

$$n(\prod_{a \in A} G(a))m \quad \Leftrightarrow \quad \forall a \in A.\ n \cdot a G(a) m \cdot a$$

□

Remark 4.13 (Empty product) *Notice that, by the definition above, for any G:*

$$\prod_{a \in \emptyset} G(a) = \mathcal{D} \times \mathcal{D}$$

[10] The fact that terms depend on types should not be confused with the different situation of "dependent types" where types depend on terms, e.g. in the Calculus of Constructions

Clearly, $\prod_{a\in A} G(a)$ is a well defined p.e.r. and may be viewed as a collection of computable functions, relatively to $\mathcal{D}$: any element in $\mathbf{dom}(\prod_{a\in A} G(a))$ computes a function in $\bigotimes_{a\in A} G(a)$, and when $n \prod_{a\in A} G(a) m$, then n and m compute the same function. That is, by the usual abuse of language, we may identify functions and equivalence classes and write:

$$f \in \prod_{a\in A} G(a) \text{ iff } f \in \bigotimes_{a\in A} G(a) \text{ and } \exists n \in \mathcal{D}. \forall a \in A. f(a) = \lceil n \cdot a \rceil_{G(a)}. \tag{6}$$

We then say that n **realizes** f.

Our aim is to give meaning to functions "computing with types". The idea is to consider the type symbols as a particular subset of $\mathcal{D}$ and use some strong topological properties of a particular model $(\mathcal{D}, \cdot)$, namely of $(\mathcal{P}\omega, \cdot)$ in 4.1, to interpret these peculiar functions. Thus, from now on, we specialize $(\mathcal{D}, \cdot)$ to $(\mathcal{P}\omega, \cdot)$. Recall that $\mathcal{P}\omega$ may be given a topological structure, the Scott topology, by taking as a basis the empty set plus the sets $\{a \in \mathcal{P}\omega \mid e_n \subseteq a\}$, where $\{e_n\}_{n\in\omega}$ is an enumeration of the finite subsets of $\mathcal{P}\omega$.

Assume then that each type symbol U is associated, in an injective fashion, with an element n in $\mathcal{D}$, the **semantic code** of U in $\mathcal{D}$. Call $[\mathbf{Type}] \subseteq \mathcal{D}$ the collection of semantic codes of types. The choice of the set of codes is irrelevant, provided that

- it is in a bijection with **Type**;
- the induced topology on $[\mathbf{Type}]$ is the discrete topology.

These assumptions may be easily satisfied, in view of the cardinality and the topological structure of the model $\mathcal{D}$ we chose. For example, enumerate the set of type symbols and fix $[\mathbf{Type}]$ to be the collection of singletons $\{\{i\} | i \in \omega\}$ of $\mathcal{P}\omega$ (**Type** is countable as each type has a finite representation). We then write $\mathcal{T}_n$ for the type-symbol associated with code n[11] and, given $K \subseteq \mathbf{Type}$, we set $[K] = \{n | \mathcal{T}_n \in K\}$.

We can now interpret as a p.e.r. any product indexed over a subset $[K]$ of $[\mathbf{Type}]$. Indeed, this will be the semantic tool required to understand the formalization of overloading we proposed: in $\lambda\&$, the value of terms or procedures may depend on types. This is the actual meaning of overloaded terms: they apply a procedure, out a finite set of possible ones, according to the type of the argument. As terms will be functions in the intended types (or equivalence classes of their realizers), our choice functions will go from codes of types to (equivalence classes in) the semantic types.

Remark 4.14 *The reader may observe that there is an implicit higher order construction in this: terms may depend on types. However:*

- *in view of the countable (indeed finite) branching of overloaded terms and types, we do not need higher order models to interpret this dependency;*
- *note though that the intended meaning of an overloaded term is a function which depends on a possibly infinite set of input types, as it accepts terms in any subtype of the U_i types in the $\{U_i \to V_i\}$ types. Whence the use of g.o.t.'s and completions.*
- *known higher order systems (System F, Calculus of Constructions...) would not express our "true" type dependency, where different types of the argument may lead to essentially different computations. This was mentioned in the introduction and it is understood in the* **PER** *model of these calculi by a deep fact: the product indexed over (uncountable) collections of types is isomorphic to an intersection (see [LM91]). A recent syntactic understanding of this phenomenon may be found in [LMS93].*

[11] Remember that, despite the letter n, n is a singleton, not just an integer.

Remark 4.15 *Overloaded functions are similar, in a sense, to records; in the first case the basic operation is selection of a function depending on a type, while in the second case it is selection of a field depending on a label. Consequently, subtyping is strictly related too: theorem 3.9 shows that, working with the completion of types, subtyping is the same in the two cases. However we cannot get rid of overloaded type in λ&-early by encoding them as product types, using the technique developed in [CL91] for record types, since the completion of an overloaded type is an infinite structure, and also because we want to lay foundations which can be used to study the whole late-binding version of λ&.*

4.1.4 The semantics of stratified types and subtyping

We are ready to define the semantics of overloaded types as products. As we want to interpret subtyping, which is a preorder, by an order relation in the model, we will use completion to get rid of "irrelevant differences" between overloaded types. Note now that the semantics below is well defined only in case of stratified variants of λ&, as given in section 2.2. This extra condition, which also lead us to normalizing overloaded and functional applications, is essential in order to avoid a circularity in the definition of meaning for overloaded types: in the notation of definition 4.16, $\mathcal{T}_n \leq U \in [\widehat{K}]$ implies that $\mathcal{T}_n$ is "structurally simpler" than U (has a smaller equal rank than U), when the system is stratified. Thus $\{U \to out(U)\}_{U \in K}$ may be understood in terms of the "composition of meanings" of the $\mathcal{T}_n$'s. Let then $(\lambda\&^{-}, rank)$ be a stratified system

Definition 4.16 The semantics of overloaded types is given by

$$[\![\{U \to out(U)\}_{U \in K}]\!] = \prod_{n \in [\widehat{K}]} [\![\mathcal{T}_n \to \widehat{out}(\mathcal{T}_n)]\!]$$

where $[\widehat{K}] = \left\{ n \,\middle|\, \mathcal{T}_n \in \widehat{K} \right\}$ □

This is a well defined meaning over **PER**, by definition 4.12, where $A = [\widehat{K}]$ and $G\colon [\widehat{K}] \to \mathbf{PER}$ is given by $G(n) = [\![\mathcal{T}_n \to \widehat{out}(\mathcal{T}_n)]\!]$. It clearly extends to g.o.t.'s, as we only need that K is countable, here. Now we are finally in the position to check that the preorder on types is interpreted as the partial order "$\leq$" on **PER**.

Theorem 4.17 *If $U \leq V$ is derivable, then $[\![U]\!] \leq [\![V]\!]$ in* **PER**.

Proof. The proof goes by induction on the structure of types, the only critical case concerns the overloaded types and will be an easy consequence of theorem 3.9.

atomic types

by definition

arrow types

by proposition 4.6

overloaded types

Let (K, out) and (H, out') be g.o.t.. Assume that $\{U \to out'(U)\}_{U \in H} \leq \{U \to out(U)\}_{U \in K}$. We need to show that $\prod_{i \in [\widehat{H}]} [\![\mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i)]\!] \leq \prod_{i \in [\widehat{K}]} [\![\mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)]\!]$ in **PER**.

By 3.9, $[\widehat{K}] \subseteq [\widehat{H}]$ and $\forall i \in [\widehat{K}].\, \mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i) \leq \mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)$. Hence:

$$\begin{array}{lcll} m \prod_{i \in [\widehat{H}]} [\![\mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i)]\!] n & \Leftrightarrow & \forall i \in [\widehat{H}].\, m [\![\mathcal{T}_i \to \widehat{out'}(\mathcal{T}_i)]\!] n & \text{by definition} \\ & \Rightarrow & \forall i \in [\widehat{K}].\, m [\![\mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)]\!] n & \text{by 3.9} \\ & \Rightarrow & m \prod_{i \in [\widehat{K}]} [\![\mathcal{T}_i \to \widehat{out}(\mathcal{T}_i)]\!] n & \end{array}$$

□

It should be clear that we presented here a core extension of the typed λ-calculus with overloading and subtyping. Note though that it suffices to represent records, as shown in [CGL92]. As for recursively defined procedures, the natural extension of our language by a fixed point operator over terms may represent them. A sound model, as a substructure of our semantics, may be derived from the work in [Ama90].

4.1.5 The semantics of terms

We can now give meaning to terms of the $\lambda\&$-calculus, with the use of coercions introduced in the previous section.

Syntactic coercions were denoted by c_V where V was the type the argument was coerced to; the type-checker assured that this type was greater than the type of the argument of c_V. In the semantics we need also to know the type of the argument since the semantic coercions are "typed functions", from a p.e.r. to another: thus, we denote semantic coercions between p.e.r.'s $[\![U]\!]$ and $[\![V]\!]$ by c_{UV}; the double indexation distinguishes them from the syntactic symbol. Also, in the following we index a term by a type as a shorthand to indicate that the term possesses that type.

An environment e for typed variables is a map $e: Var \to \bigcup_{A \in \mathbf{PER}} A$ such that $e(x^U) \in [\![U]\!]$. Thus each typed variable is interpreted as an equivalence class in its semantic type. This will be now extended to the interpretation of terms by an inductive definition, as usual.

In spite of the heavy notation, required by the blend of subtyping and overloading, the intuition in the next definition should be clear. The crucial point 6 gives meaning to an overloaded term by a function which lives in an indexed product (as it will be shown formally below): the product is indexed over (indexes for) types and the output of the function is the (meaning of the) term or computation that one has to apply. Of course, this is presented inductively. Some coercions are required as M_1 and M_2 may live in smaller types than the ones in $\&^{\{V_i \to W_i\}_{i \le n}}$. Then, in point 7, this term is actually applied to the term argument of the overloaded term.

Definition 4.18 [semantics of terms] Let $e: Var \longrightarrow \bigcup_{A \in PER} A$ be an environment. Set then:

1. $[\![\varepsilon]\!]_e = \mathcal{D}$, the only equivalence class in the p.e.r. $\mathcal{D} \times \mathcal{D}$ (see remark 4.13)
2. $[\![x^U]\!]_e = e(x^U)$
3. $[\![\lambda x^U.M^V]\!]_e = \lceil n \rceil_{[\![U \to V]\!]}$ where n is a realizer of f such that $\forall u \in [\![U]\!].f(u) = [\![M^V]\!]_{e[x/u]}$
4. $[\![M^{U \to V} N^W]\!]_e = [\![M^{U \to V}]\!]_e [\![N^W]\!]_e$
5. $[\![c^V(M^U)]\!]_e = c_{UV}([\![M^U]\!]_e)$ (the semantic coercion)
6. Let $(M_1 \&^{\{V_i \to W_i\}_{i \le n}} M_2): \{U \to out(U)\}_{U \in \{V_i\}_{i \le n}}$ with $M_1: T_1 \le \{V_i \to W_i\}_{i<n}$ and $M_2: T_2 \le V_n \to W_n$.

   Set then $[\![M_1 \& M_2]\!]_e = f$ such that, given $j \in [\widehat{\{V_i\}_{i \le n}}]$ and $Z = \min^{12}\{U \in \{V_i\}_{i \le n} | \mathcal{T}_j \le U\}$, one has

$$f(j) = \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e) & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_e)(j) & \text{else} \end{cases}$$

7. $[\![M^{\{U \to out(U)\}_{U \in K}} \bullet c_V(N^W)]\!]_e = [\![M^{\{U \to out(U)\}_{U \in K}}]\!]_e(i)[\![c_V(N^W)]\!]_e$ where $\mathcal{T}_i = V$.

□

[12] We should write $a\_min\{\ldots\}$, but note that in this case the set $\min\{\ldots\}$ is a singleton.

Remark 4.19 *Notice that this semantics does not interpret reduction as equality. Indeed:*

$$\begin{array}{rcll} [\![(\lambda x^V.Q^U)P^W]\!]_e & = & [\![\lambda x^V.Q^U]\!]_e[\![P^W]\!]_e & \\ & = & \lceil n \rceil_{[\![V \to U]\!]}[\![P^W]\!]_e & \textit{with } n \textit{ as in point 3 of definition 4.18} \\ & = & \lceil n \rceil_{[\![V \to U]\!]}(c_{WV}[\![P^W]\!]_e) & \textit{by 4.11} \\ & = & [\![Q^U]\!]_{e[x/c_{WV}[\![P^W]\!]_e]} & \textit{by point 3 of definition 4.18} \end{array}$$

In general, $[\![Q]\!]_{e[x/c_{WV}[\![P^W]\!]_e]}$ *is different from* $[\![Q[x/P^W]]\!]_e$*. For example, if* $Q = x$*, the two expressions evaluate to* $c_{WV}[\![P^W]\!]_e$ *and to* $[\![P^W]\!]_e$ *respectively. This will be more generally understood in 4.22.*

The soundness of this definition is split into two theorems and is proved right below. We recall first, in a lemma, that $\mathcal{P}\omega$ is an "injective" topological space.

Lemma 4.20 (injectivity) *Let* Y *be a topological space and* $X \subseteq Y$*, a subspace with the induced topology. Then any continuous* $h : X \to \mathcal{P}\omega$ *can be extended to a continuous* $\hbar : Y \to \mathcal{P}\omega$*. Indeed,* $\hbar$ *is given by* $\hbar(y) = \sqcup\{\sqcap\{h(x) | x \in X \cap U\} | y \in U\}$*. (The proof is easy; see [Sco76] for this and more properties of* $\mathcal{P}\omega$*).*

Theorem 4.21 (soundness w.r.t. type-checking) *If* $N{:}U$ *then, for any environment* e*,* $[\![N]\!]_e$ *is well defined and* $[\![N]\!]_e \in [\![U]\!]$*.*

Proof. The proof goes by induction on the structure of N:

1. If $N \equiv x^U$, then $e(x^U) \in [\![U]\!]$ by definition.
2. If $N \equiv \varepsilon$, since $[\![\varepsilon]\!] = \mathcal{D}$, then $[\![\varepsilon]\!] \in [\![\{\}]\!] \equiv \mathcal{D} \times \mathcal{D}$ (see 4.13).
3. If $N \equiv \lambda x^U.M^V$, consider $[\![\lambda x^U.M^V]\!]_e = f$ such that $\forall u \in [\![U]\!].f(u) = [\![M^V]\!]_{e[u/x]}$. We need to show that f lives in the right type and it is realized, or, equivalently, that f, as a set of realizers, is in $[\![U \to V]\!]$. This is a consequence of the proof that **PER** is a CCC, as lambda abstraction is the currying operation (see for instance [AL91]).
4. If $N \equiv M^{U \to V}P^W$, where $W \leq U$. By induction $[\![M^{U \to V}]\!]_e \in [\![U \to V]\!]$ and $[\![P^W]\!]_e \in [\![W]\!] \leq [\![U]\!]$. Then the result follows by 4.9.
5. Assume that $N \equiv (M_1 \&^{\{V_i \to W_i\}_{i \leq n}} M_2){:}\{U \to out(U)\}_{U \in \{V_i\}_{i \leq n}}$ with $M_1{:}T_1 \leq \{V_i \to W_i\}_{i<n}$ and $M_2{:}T_2 \leq V_n \to W_n$. Set $[\![M_1 \& M_2]\!]_e = f$ as in the definition 4.18. We prove this case by induction on n; in each case we have to prove that f lives in the set-theoretic indexed product $\bigotimes_{j \in [\widehat{\{V_i\}_{i \leq n}}]}[\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$ and that it is realized.

   ($n = 1$) . In this case $\{V_i\}_{i \leq n} = \{V_1\}$ By definition 4.18 we have that $\forall j \in [\widehat{\{V_1\}}].f(j) = c_{T_2(\mathcal{T}_j \to W_1)}([\![M_2]\!]_e)$ as there is only one branch to choose. By induction $[\![M_2]\!]_e \in [\![T_2]\!]$ and, since $\mathcal{T}_j \leq V_1$, the coercion application makes sense. The result lives in the correct type since, for each $j \in [\widehat{\{V_1\}}]$, $f(j) = c_{T_2(\mathcal{T}_j \to W_1)}([\![M_2]\!]_e) \in [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$ and, thus, the whole f lives in the correct set-theoretic indexed product. To conclude the proof we have to show that f is realizable.

   Intuitively, f can be realized by an index for a constant function, since for any input type f executes always the same code M_2; this code is coerced to different types $\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)$, but, thanks to the interpretation of subtyping, any realizer for this code in the minimum type T_2 lives in the domain of any of its supertypes $\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)$.

Formally, consider first a map $f' : [\widehat{\{V_1\}}] \to \mathcal{P}\omega$ which chooses, for each $j \in [\widehat{K}]$, always the same element n of the equivalence class $[\![M_2]\!]_e \in [\![T_2]\!]$. Then f' is computed by any index of the chosen constant function. These indexes realize f: since $f'(j)$ lives in $[\![M_2^T]\!]_e$, then

$$\lceil f'(j) \rceil_{[\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]} = c_{T,\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)}[\![M_2^T]\!]_e = f(j).$$

Note that, though f is realized by an index of a constant function, f itself is *not* constant.

($n > 1$) . In this case, we have that $M_1 : T_1 \leq \{V_i \to W_i\}_{i<n} \equiv \{U \to out(U)\}_{U \in \{V_i\}_{i<n}}$. By induction hypothesis, $[\![M_1]\!]_e \in [\![T_1]\!]$ and thus $c_{T_1(\{V_i \to W_i\}_{i<n})}([\![M_1]\!]_e) \in [\![\{V_i \to W_i\}_{i<n}]\!]$. In particular, let g be the map such that for each $j \in [\widehat{\{V_i\}_{i<n}}]. g(j) = c_{T_1(\{V_i \to W_i\}_{i<n})}([\![M_1]\!]_e)(j)$. Then, by the induction, g is in $\prod_{j \in [\widehat{\{V_i\}_{i<n}}]} [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$. Consider now f defined as in 4.18 point 6 from $(M_1 \&^{\{V_i \to W_i\}_{i \leq n}} M_2)$, that is, $f(j) = c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e)$ or $f(j) = g(j)$ according to whether $Z = \min\{U \in \{V_i\}_{i \leq n} | \mathcal{T}_j \leq U\}$ is equal to V_n or to some V_m for $m < n$. Note that f is well defined, as, for each $j \in [\widehat{\{V_i\}_{i \leq n}}]$ there exists (and is unique) the least V_z such that $\mathcal{T}_j \leq V_z$, by the definition of well-formed overloaded types. By this, f is in $\bigotimes_{j \in [\widehat{\{V_i\}_{i \leq n}}]} [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$.

To prove that f is realizable, consider the map f' which chooses for each $j \in [\widehat{\{V_i\}_{i \leq n}}]$ an element $f'(j)$ of the equivalence class $f(j)$. Clearly, f' is (well defined and) continuous, since $[\widehat{\{V_i\}_{i \leq n}}] \subseteq \mathcal{P}\omega$ is endowed with the discrete topology. Then its continuous extension $\underline{f'} : \mathcal{P}\omega \to \mathcal{P}\omega$, given as in the lemma, is computed by some $n \in \mathcal{P}\omega$, i.e. $\underline{f'}(p) = n \cdot p$, see [Sco76]. In conclusion,

$$\exists n \in \mathcal{P}\omega.\ \forall j \in [\widehat{\{V_i\}_{i \leq n}}].\ f(j) = \lceil f'(j) \rceil_{[\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]} = \lceil n \cdot j \rceil_{[\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]}$$

and thus $f \in \prod_{j \in [\widehat{\{V_i\}_{i \leq n}}]} [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!]$. (Equivalently, one could extend by continuity g, defined as above, to f, by the same "injectivity" argument; this argument is needed, anyway, as an extension by a constant value, does not need to be continuous, in general. We preferred to define a new realizer and use the inductive hypothesis just to check the semantic types).

6. If $N \equiv M^{\{U \to out(U)\}_{U \in K}} \bullet c_V(P^W) : out(Z)$ where $Z = \min\{U \in K | V \leq U\}$, then

$$[\![N]\!] = [\![M^{\{U \to out(U)\}_{U \in K}}]\!]_e(j) c_{WV}([\![P^W]\!]) \text{ for } \mathcal{T}_j = V.$$

By the previous point, $[\![M^{\{U \to out(U)\}_{U \in K}}]\!]_e(j) \in [\![\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j)]\!] \equiv [\![V \to out(Z)]\!]$, hence

$$[\![N]\!]_e \in [\![out(Z)]\!]$$

□

We observed that reductions are not interpreted as equalities in the model. Indeed they yield set theoretic inclusions.

Lemma 4.22 (substitution) $[\![Q[x/P]]\!]_e \subseteq [\![Q]\!]_{e[x/p]}$ *where* $p = c_{T'T}[\![P]\!]_e$, $x : T$, $P : T' \leq T$

Proof. The proof goes by induction on the structure of the terms:

1. $Q \equiv y \neq x$ or $Q \equiv \varepsilon$: trivial

2. $Q \equiv x$

   $[\![x[x/P]]\!]_e = [\![P]\!]_e \in [\![T']\!]$ is contained in $[\![x]\!]_{e[x/p]} = p = c_{T'T}[\![P]\!]_e \in [\![T]\!]$, by the definition of semantic coercions.

3. $Q \equiv \lambda y^U.M$

   As usual we identify a function with the set of its realizers, in the intended type. Thus

   $$[\![\lambda y^U.M[x/P]]\!]_e = f \text{ such that } \forall u \in [\![U]\!].f(u) = [\![M[x/P]]\!]_{e[y/u]}$$

   while

   $$[\![\lambda y^U.M]\!]_{e[x/p]} = f' \text{ such that } \forall u \in [\![U]\!].f'(u) = [\![M]\!]_{e[x/p;y/u]} \supseteq [\![M[x/P]]\!]_{e[y/u]}$$

   (the inclusion holds by induction hypothesis), thus each realizer for f is also a realizer for f', that is

   $$[\![\lambda y^U.M[x/P]]\!]_e \subseteq [\![\lambda y^U.M]\!]_{e[x/p]}$$

4. $Q \equiv M \cdot N$

   the proof goes by the usual induction. Just recall 4.11 and the monotonicity of application (Lemma 4.10).

5. $Q \equiv M \bullet c_V(N)$

   $$\begin{array}{lll} [\![M \bullet c_V(N)]\!]_{e[x/p]} = & & \\ = & [\![M]\!]_{e[x/p]}(j)([\![c_V(N)]\!]_{e[x/p]}) & \text{for } \mathcal{T}_j = V \\ \supseteq & [\![M[x/P]]\!]_e(j)([\![c_V(N)[x/P]]\!]_e) & \text{by ind. and by the monotonicity of application in } P\omega \\ = & [\![(M[x/P]) \bullet (c_V(N)[x/P])]\!]_e & \\ = & [\![(M \bullet c_V(N))[x/P]]\!]_e & \end{array}$$

6. $Q \equiv M_1 \& M_2 : \{V_i \to W_i\}_{i<n}$

   $$[\![M_1 \& M_2]\!]_{e[x/p]} = f \text{ such that } f(j) = \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_{e[x/p]}) & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_{e[x/p]})(j) & \text{else} \end{cases}$$

   $$[\![(M_1 \& M_2)[x/P]]\!]_e = f' \text{ such that } f'(j) = \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2[x/P]]\!]_e) & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1[x/P]]\!]_e)(j) & \text{else} \end{cases}$$

   the result follows by the same reasoning as in (3.). More precisely, since in this case $[\![Q[x/P]]\!]_e$ and $[\![Q]\!]_{e[x/p]}$ have the same type, from $[\![Q[x/P]]\!]_e \subseteq [\![Q]\!]_{e[x/p]}$ we can deduce $[\![Q[x/P]]\!]_e = [\![Q]\!]_{e[x/p]}$.

□

The immediate consequence of the work done so far is the construction of a simple model of "reduction" and not, as customary in denotational semantics, of "conversion". This is precisely stated by the following theorem.

Theorem 4.23 (soundness wrt reductions) $M^U \rhd N^V \Rightarrow [\![M^U]\!]_e \supseteq [\![N^V]\!]_e$

Proof. The proof goes by induction on the definition of $\rhd$. The critical cases are:

1. $M \equiv (\lambda x.P)Q^W$ and $N \equiv P[x/Q]$ with $(\lambda x.P): U \to V$ and $Q: W$

$$\begin{array}{rcll} [\![(\lambda x.P)Q]\!]_e & = & [\![(\lambda x.P)]\!]_e [\![Q]\!]_e & \\ & = & [\![(\lambda x.P)]\!]_e c_{WU}([\![Q]\!]_e) & \textit{by } 4.11 \\ & = & [\![P]\!]_{e[x/q]} & \text{where } q = c_{WU}([\![Q]\!]_e) \\ & \supseteq & [\![P[x/Q]]\!]_e & \text{by the substitution lemma} \\ & = & [\![N]\!]_e & \end{array}$$

2. $M \equiv (M_1 \& M_2) \bullet P$ with $(M_1 \&^{\{V_i \to W_i\}_{i \le n}} M_2): \{U \to out(U)\}_{U \in \{V_i\}_{i \le N}}$, $M_1: T_1 \le \{V_i \to W_i\}_{i<n}$, $M_2: T_2 \le V_n \to W_n$ and $P \equiv c_{\mathcal{T}_j}(P'): \mathcal{T}_j$ for some $P' \in$Terms and $\mathcal{T}_j \in$Types. Thus, $N \equiv M_h P$ for $h = 1$ or $h = 2$. More precisely, for $Z = \min\{U \in \{V_i\}_{i \le N} | \mathcal{T}_j \le U\}$,

$$N \equiv \begin{cases} M_2 \cdot P & \text{if } Z = V_n \\ M_1 \bullet P & \text{else} \end{cases}$$

Compute then

$$\begin{array}{rcl} [\![(M_1 \& M_2) \bullet P^{\mathcal{T}_j}]\!]_e & = & [\![(M_1 \& M_2)]\!]_e(j)[\![P]\!]_e \\ & = & \begin{cases} c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e)[\![P]\!]_e & \text{if } Z = V_n \\ (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_e)(j)[\![P]\!]_e & \text{else} \end{cases} \end{array}$$

In the first case,

$$\begin{array}{rcll} [\![M]\!]_e & = & c_{T_2(\mathcal{T}_j \to \widehat{out}(\mathcal{T}_j))}([\![M_2]\!]_e)[\![P]\!]_e & \\ & \supseteq & ([\![M_2]\!]_e)[\![P]\!]_e & \text{by monotonicity and by } a \subseteq c_{UV}(a) \\ & = & [\![N]\!]_e & \end{array}$$

Otherwise

$$\begin{array}{rcll} [\![M]\!]_e & = & (c_{T_1(\{V_i \to W_i\}_{i<n})}[\![M_1]\!]_e)(j)[\![P]\!]_e & \\ & \supseteq & [\![M_1]\!]_e(j)[\![P]\!]_e & \text{by monotonicity and by } a \subseteq c_{UV}(a) \\ & = & [\![M_1 \bullet P]\!]_e & \\ & = & [\![N]\!]_e & \end{array}$$

□

Remark 4.24 *Clearly, theorem 4.23 specializes to the implicative fragment of our calculus, which is simply typed λ-calculus with subtyping. Thus, by a simple observation of the properties of* **PER**, *we spotted a mathematical model of the reduction predicate "$\triangleright$" between terms of λ-calculi, instead of conversion "=". The non-syntactic models so far constructed could only give mathematical meaning to the theory of "=" between λ-terms and β-reduction was interpreted as the "=".*

It is important to notice, however, that the decrease of the size of the equivalence class which is the interpretation of a term is not directly related to the reduction process, but to the fact that types decrease during computation. In fact, if you consider two terms M and $c_V(M)$ and apply the same reduction steps to both of them, while the semantics of M can decrease, any time its type changes, the semantics of $c_V(M)$ remain fixed, even if the same reduction steps are executed.

5 Summary

As already mentioned in the introduction, there is a general understanding that polymorphism, as intended in λ-calculus, is not compatible with "procedures depending on input types". As pointed out

in [Gir72], one cannot extend II order λ-calculus with a term giving different output terms according to different input types. Indeed, in [LMS93], it is shown that terms depending on types use types as "generic", i.e. the value on just one type determines the value everywhere. This is why, in order to express an explicit type dependency, it was not sufficient to extend simply typed λ-calculus by type variables, and we proposed an entirely new feature, based on "finite branching of terms", in order to formalize the dependency we wanted. Moreover, the use of late-binding and subtyping added expressiveness to the system. Indeed, the expressive power of the syntax poses some problems which were partly dealt with by a "stratification" of the subtyping relation.

As for the stepwise construction of the model, the use of a pre-order between types is first handled at a syntactic level, by the notion of completion in section 3.1. But then our finite branching immediately becomes an infinite one: this is indeed what is actually meant in the syntax, by the rules, as we allow terms to work also on inputs inhabiting types *smaller* than the intended one. Thus, the intended function depending on the type of the input, may depend on an infinity of input types, implicitly. This must be made explicit in the semantics.

Finally, we considered types as "coded" in the semantics, by using their indexes also as meaning in the model. Note that this mathematical meaning of types corresponds to the practice of type-dependent computations. In programming, when and if computing depends on types, this is possible as types, after all, are just "code" (in OOP it corresponds to consider classes as *tags*); thus they are handled like any countable (and enumerated) data type. This is impossible in sound mathematical models which respect the logical "second order" convention. Indeed, in this case, types must be (arbitrary) subsets of the (infinite) sets interpreting terms. Observe finally that the implicit polymorphism of our approach shows up in the semantics by the interpretation of overloaded functions as elements of an (infinite) indexed product.

To sum up, in this phase of the work we focused on the problems related to type dependency and to the lack of antisymmetry of the subtyping relation. The result obtained may be the basis, in futere work, for the definition of a denotational semantics for the full language, dealing also with the problem of late-binding.

Acknowledgments
Giuseppe Castagna has been supported by the grant no. 203.01.56 of the Italian National Research Council (CNR) - "Comitato Nazionale delle Scienze Matematiche".

Giorgio Ghelli has been partially supported by C.E.C., Esprit Basic Research Action 6309 FIDE2 and by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of the Italian National Research Council under grant No.92.01561.PF69.

References

[ACC93] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. In Dezani, Ronchi, and Venturini, editors, *Böhm Festschrift*. 1993.

[AKP91] H Aït-Kaci and A. Podelski. Towards a meaning of LIFE. Technical Report 11, Digital, Paris Research Laboratory, June 1991.

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

[Ama90] R. Amadio. Domains in a realizability framework. Technical Report 19, Laboratoire d'Informatique, Ecole Normale Supérieure - Paris, 1990.

[Bar84] H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*. North-Holland, 1984. Revised edition.

[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 preliminary version can be found in *3rd Ann. Symp. on Logic in Computer Science*, 1988.

[Bru91] K.B. Bruce. The equivalence of two semantic definitions of inheritance in object-oriented languages. In *Proceedings of the 6th International Conference on Mathematical Foundation of Programming Semantics*, 1991. To appear.

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

[Cas92] G. Castagna. Strong typing in object-oriented paradigms. Technical Report 92-11, Laboratoire d'Informatique, Ecole Normale Supérieure - Paris, June 1992.

[CCH+89] P.S. Canning, W.R. Cook, W.L. Hill, J. Mitchell, and W.G. Orthoff. F-bounded quantification for object-oriented programming. In *ACM Conference on Functional Programming and Computer Architecture*, September 1989.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping, 1992. To appear in *Information and Computation*. An extended abstract has appeared in the proceedings of the *ACM Conference on LISP and Functional Programming*, pp.182-192; San Francisco, June 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.

[CL91] L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[CMMS91] L. Cardelli, S. Martini, J.C. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 750–771. Springer-Verlag, September 1991. LNCS 526 (preliminary version). To appear in Information and Computation.

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

[Ghe91a] G. Ghelli. Modelling features of object-oriented languages in second order functional languages with subtypes. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in LNCS, pages 311–340, Berlin, 1991. Springer-Verlag.

[Ghe91b] G. Ghelli. A static type system for message passing. In *Proc. of OOPSLA '91*, 1991.

[Gir72] J-Y. Girard. Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Thèse de doctorat d'état, 1972. Université Paris VII.

[GM85] Joseph Goguen and José Meseguer. EQLOG: equality, types and generic modules for logic programming. In deGroot and Lindstrom, editors, *Functional and Logic Programming.* Prentice-Hall, 1985.

[GM89] J.A. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical Report SRI-CSL-89-10, Computer Science Laboratory, SRI International, July 1989.

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

[LM91] G. Longo and E. Moggi. Constructive natural deduction and its ω-set interpretation. *Mathematical Structures in Computer Science*, 1(2):215–253, 1991.

[LMS93] G Longo, K. Milsted, and S. Soloviev. The genericity theorem and parametricity in functional languages. *Theoretical Computer Science*, 1993. Special issue in honour of Corrado Böhm, to appear. An extended abstract has been presented at the 8th Annual IEEE Symposium on Logic in Computer Science, Montreal, June 1993.

[Lon83] G. Longo. Set-theoretical models of lambda-calculus: Theories, expansions, isomorphisms. *Annals of Pure and Applied Logic*, 24:153–188, 1983.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall International Series, 1988.

[NeX91] NeXT Computer Inc. *NeXTstep-concepts. Chapter 3: Object-Oriented Programming and Objective-C*, 2.0 edition, 1991.

[Pie93] B.C. Pierce. Bounded quantification is undecidable. In *20th Ann. ACM Symp. on Principles of Programming Languages*. ACM-Press, 1993.

[PT93] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 1993. To appear; a preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Rém89] D. Rémy. Typechecking records and variants in a natural extension of ML. In *16th Ann. ACM Symp. on Principles of Programming Languages*, 1989.

[Rey84] J.C. Reynolds. Polymorphism is not set-theoretic. *LNCS*, 173, 1984.

[Sco76] D. Scott. Data-types as lattices. *S. I. A. M. J. Comp.*, 5:522–587, 1976.

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

[Wan91] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1):1–15, 1991.

[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.