A Calculus for Overload Functions with Subtyping

G. CASTAGNA G. GHELLI
G. LONGO

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

LIENS - 92 - 4

February 1992

A Calculus for Overloaded Functions with Subtyping *

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

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

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

August 3, 1993

Abstract

We present a simple extension of typed λ-calculus where functions can be *overloaded* by putting different "branches of code" together. When the function is applied, the branch to execute is chosen according to a particular selection rule which depends on the type of the argument. The crucial feature of the present approach is that the branch selection depends on the "run-time type" of the argument, which may differ from its compile-time type, because of the existence of a subtyping relation among types. Hence overloading cannot be eliminated by a static analysis of code, but is an essential feature to be dealt with during computation. We obtain in this way a type-dependent calculus, which differs from the various λ-calculi where types do not play any rôle during computation. We prove Confluence and a generalized Subject-Reduction theorem for this calculus. We prove Strong Normalization for a "stratified" subcalculus. The definition of this calculus is guided by the understanding of object-oriented features and the connections between our calculus and object-orientedness are extensively stressed. We show that this calculus provides a foundation for typed object-oriented languages which solves some of the problems of the standard record-based approach.

1 Introduction

An important distinction has been extensively used in language theory for the last two decades, between parametric (or universal) polymorphism and "ad hoc" polymorphism

*An extended abstract of this work appeared in the proceedings of the ACM Conference on LISP and Functional Programming, San Francisco, June 1992. To appear in INFORMATION and COMPUTATION

†This autor has been supported by a grant of the BRA-Esprit project no. 3020 and by the grant no. 203.01.56 of the Consiglio Nazionale delle Ricerche – Comitato Nazionale delle Scienze Matematiche, Italy

‡This author has been partially supported by E.E.C., Esprit Basic Research Action 6309 FIDE2, by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council under grant No.91.00877.PF69 and by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica".

§Part of this author's work has been supported by a collaboration at DEC- PRL (Digital, Paris Research Laboratory) within the "Paradise" project.

(see [CW85]). Parametric polymorphism allows one to write a function whose code can work on different types, while using "ad hoc" polymorphism it is possible to write a function which executes different code for each type. Both the Proof Theory and the semantics of the first kind of polymorphism have been widely investigated by many authors (including two of the authors of the present paper), on the grounds of early work of Hindley, Girard, Milner and Reynolds, and developed into robust programming practice. The second kind, usually known as "overloading", has had little theoretical attention, with the notable exception of [WB89], [MOM90] and [Rou90]; consequently, its wide use has been little affected by any influence comparable to the one exerted by implicit and explicit polymorphism in programming.

This is due, probably, to the fact that the traditional languages offer a very limited form of overloading: in most of them only predefined functions (essentially arithmetic operators defined on integers and reals and input/output operators) are overloaded, while in the relatively few languages where the programmer can define overloaded functions their actual meaning is always decided at compile time. This form of overloading can be easily understood as a form of syntactic abbreviation which does not significantly affect the underlying language.

We believe though that the ability to define new overloaded functions, when combined with subtyping and with late-binding (as defined below), allows a high level of code reusability, and is the main point which distinguishes object-oriented programming from programming with abstract data types.

In this paper we begin a theoretical analysis, and thus a "uniform and general" one, of this richer kind of overloading. It appears that the challenges it poses are non trivial: indeed, this paper is just a preliminary step towards a theoretical universe still to be discovered and which, we claim, may also affect language design.

We design here a formalism where functions can be *overloaded* by adding a different "piece of code". Thus the code of an overloaded function is formed by several branches of code. The branch to execute is chosen when the function is applied, according to a particular selection rule, which depends on the type of the argument. A crucial feature of the present approach is that the branch selection depends on the "run-time type" of the argument, which may differ from its compile-time type (late binding). Hence, branch selection cannot be performed at compile-time, as happens in imperative languages (early binding), but has to be performed during computation, each time the overloaded application is evaluated.

For example, suppose that *Circle* and *Square* are subtypes of *Picture*, and *draw* is an overloaded function 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 (here denoted $\bullet$) like the following one

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

is always executed using the *draw* code for *pictures*; with late binding, each time the whole function is applied, the code for *draw* is chosen only when the x parameter has been bound and evaluated. Thus the appropriate code for *draw* is used on the basis of

the run-time type of x and according to whether x is bound to a circle or to a square.

We do not present a general treatment for overloaded functions, but we develop a purely functional approach focussed on the study of some features of object-orientedness, namely message-passing, inheritance and subtyping. Since the approach is entirely novel we first felt the need, via this preliminary, proof-theoretic analysis, to develop the non trivial investigation of key functional properties, such as normalization, confluence and "subject-reduction" (i.e. termination and consistency and "how types evolve during computation"), in the setting of a truly type dependent calculus.

Indeed, "type dependency" (the fact that terms and values may depend on types) and the role played by the distinction between run-time and compile-time types are peculiar properties of the calculus. The various (higher order) calculi, such as Girard's System F and its extensions, allow abstraction w.r.t. type variables and the application of terms to types, but the "value" of this application does not truly depend on the argument type, and more generally the semantics of an expression does not depend on the types which appear in it. Indeed, this "parametricity" or "type-erasure" property plays a crucial role in the basic proof-theoretic property of these calculi: the normalization (cut-elimination) theorem. In the semantic interpretations, this essential type independence of computations is understood by the fact that the meaning of polymorphic functions is given by essentially constant functions (we will say more about this in the last section of this paper). On the other hand, it is clear that overloaded functions express computations which truly depend on types, as different branches of code (i.e. possibly unrelated terms) may be applied on the basis of input types.

Our motivation comes from considering overloading as a way to interpret message-passing in object-oriented programming, when methods are viewed as "global" functions. Let us be more specific. In object-oriented languages the computation evolves on objects. Objects are programming items grouped in *classes* and possessing an internal state that may be accessed and modified by sending messages to the object. When an object receives a message it invokes the method (i.e. the code) associated with that message. The association between methods and messages is described by the class the object belongs to.

There are two possible ways to see message-passing: the first is to consider objects as arrays that associate a method with each message. Therefore when a message `m` is passed to an object `obj` then the method associated with `m` in the object `obj` is looked for. In this approach, an object has the form shown in Figure 1:

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

Figure 1.
Objects as records.

message_i	
`class_name_1`	method_1
⋮	⋮
`class_name_n`	method_n

Figure 2.
Messages as overloaded functions.

This first point of view has been extensively studied and corresponds to the "objects as records" analogy [Car88]. The second approach to message-passing, as shown in Figure 2, is to consider messages as names of overloaded functions: depending on the class (or more generally, the type) of the object the message is passed to, a different method is chosen (this is the approach of the CLOS language [DG87]; this approach was introduced, in the context of typed languages, in [Ghe91b]). In this way, in a sense, we reverse the previous situation: instead of passing messages to objects we now pass objects to messages.

This different approach seems to have some advantages w.r.t. the "objects as records" paradigm, at least in a proof-theoretical study of the typed case.

For example, in the record based approach an object of class A containing a method *binary* of type $A \times A \to B$ is modeled as a record belonging to the following recursive type

$$\mu A.\langle\langle \ldots \text{binary}: A \to B \ldots \rangle\rangle;$$

since the first argument of the function is the record itself and, thus, it is an internal or *hidden* argument of the method, referred by the keyword *self*. In the same way a recursive type is needed to type an object of class A with a *copy* method of type $A \to A$:

$$\mu A.\langle\langle \ldots \text{copy}: A \ldots \rangle\rangle.$$

In the model based on overloading there is no need to use recursion to define the type of the *binary* and *copy* methods, and this seems more reasonable, since, intuitively, binary or copying operations generally have nothing to do with fixpoints and do not need the expressive power of recursion.

A similar observation can be made for the first "hidden" parameter *self* of methods, referring to the object receiving the message. In the record model this parameter must be accessed by recursion, while in our model it is just a parameter with the additional feature that it is used for code selection. For other problems regarding inheritance in the record based model see [CHC90, CM91, Mit90, Bru91, Ghe91a, Ghe91b].

Of course other problems arise when overloaded functions are used to define methods, especially to model the dynamic definition of new classes. On the other hand, the full expressiveness of records is recovered, as record types and values are derivable notions in our approach.

In summary, this paper develops a simple extension of the typed λ-calculus meant to formalize the behaviour of overloaded functions with late binding in a type discipline with subtyping. The basic idea is that an overloaded function consists of a finite collection of ordinary functions that are stuck together to form the different branches. Its type will be the set of the types of its branches. Therefore we add, to ordinary λ-terms, new terms such as $(M_1 \& M_2 \& \ldots \& M_n)$, that represent the overloaded function consisting of the n branches M_i.[1] Likewise, we add an operation of overloaded application $M \bullet N$ to the ordinary functional application $M \cdot N$. The types of the overloaded functions are finite lists of arrow types $\{V_1' \to V_1'', \ldots, V_n' \to V_n''\}$ (sometimes denoted

[1] Hereafter we will call a term of the form $(M \& N)$ an "&-term".

by $\{V_i' \to V_i''\}_{i \in I}$), where every arrow corresponds to the type of a branch. Overloaded types must satisfy relevant consistency conditions; among others things, they account for the longstanding debate concerning the use of covariance or contravariance of the arrow type in its left argument. More precisely, the general arrow types will be given by contravariant "$\to$" in the first argument: this is an essential feature of (typed) functional programs, where type assignment (type-checking) helps to avoid run-time errors. On the other hand, the types of overloaded functions are covariant families of arrow types, as explained in detail below.

We stress that the subtyping relation introduced is an essential feature of the calculus: it allows multiple choices, as a type may be a subtype of several types and subtyping is used to choose branches of overloaded terms. The blend of &-terms and subtyping makes our calculus an expressive and original mathematical formalism which shows that "ad hoc" polymorphism may have also theoretical relevance.

In Section 2 we describe the combination of overloading and subtyping and the consequences of their interaction. Section 3 presents the syntax of the system as well as the reduction rules. In Section 4 we show how some other types can be encoded in the primitive system. The next Sections are devoted to the basic properties of the calculus: Section 5 to Subject-Reduction, Section 6 to confluence and Section 7 to the normalization theorem. In Section 8 we give some more intuition on how our calculus fits object-oriented programming, hinting how to implement subtyping and message-passing by the constructs of our calculus. A conclusion suggests further work, in particular the challenging extension to higher order systems.

2 Overloading, Subtyping and "Run-Time Types"

In the introduction we said that overloading is interesting because of its connection with object-oriented languages. However, we think that overloading is worth studying also for its own sake, in spite of the lack of formal studies on this mechanism. This lack may depend on the fact that overloading by itself (i.e. without subtyping), does not increase the expressiveness of the language: an overloaded function can be substituted by the appropriate code, at compile-time; in this case, overloading seems more a notational trick than a programming construct. If combined, though, with subtyping and late-binding, it becomes a flexible and powerful tool.

The idea is that if we have an overloaded function whose n branches have respectively type $U_i \to V_i$ ($i = 1..n$) and we pass it an argument of type U, the chosen branch j is the one that "best approximates" U, i.e. such that $U_j = \min_{i \in I}\{U_i | U \leq U_i\}$. However, it is known that, when a subtyping relation is defined, the type of a term is no longer the same during computation, but it may decrease (see [CG93]). This "shrinking of the run-time type" corresponds to the increase of information that characterizes the evolving of computation[2]. The fact that during computation types may get more

[2]We think that the increase in information is part of the essence of computation, and is due to the approaching of the result, the normal form, which is the most informative form of the term (informative in the sense of understandability, not in the sense of the theory of information where the computation

informative implies that, if one makes the choice of the branch at compile-time (early binding), this choice would be based on rather incomplete information about the "real" type of the argument (see the example with *Picture* in the previous section). Moreover late binding allows, in object-oriented languages, us to write a function whose behaviour may be "extended" by adding new classes. Consider for example a compiler written in object-oriented style, with a class for every syntactic class in the language. Every class implements methods *Parse*, *Type-Check*, *Pretty-Print*, and *Code-Generate*. New syntactic classes may be dynamically added by creating the corresponding class, and every general purpose function which sends, e.g., a *Pretty-Print* message to an object will be able to correctly operate on objects of these new classes too. For this reason, late binding plays an essential role in the high code reusability which can be attained in object-oriented programming, making it possible to write "graphical editor shells" or "compiler shells" where the general purpose code is written before writing the actual definition of the classes which specify the specific behaviour of the application.

The meaning of terms like "run-time type" and "compile-time type" is reasonably clear in the context of a traditional, eagerly evaluated programming language: in that case, a single term, such as an occurrence of a formal parameter x of a function, is "evaluated" many times, once each time the function is called. Each time x is bound to a value, the run-time type of that value becomes the "run-time type" of x, while in the source code that occurrence of x has a unique compile-time type, the one written by the programmer. However, the "compile-time type" of a term and the "run-time types" of its values are not unrelated: the property holds that all the run-time types of the values will be subtypes of the unique compile-time type of the term.

This distinction may not be intuitive in the context of a rewriting system, such as λ-calculus, where a more formal definition is needed. To follow the different "evaluations" of an occurrence of a term, we may use the notion of *residual* of an occurrence of a term (see [Bar84] where this definition is used only when the term is a redex). Intuitively, a *residual* is what the term has become after a reduction. As happens in traditional languages, in a rewriting system an occurrence of a term has many different residuals with possibly many different types, which are only guaranteed to be subtypes of the original one.

We will adopt the following definition: when a term is closed and normal, we then say that it is "a value",[3] and we mean by this that it cannot evolve anymore. We similarly say that its type is "a run-time type", which means that no more information can be specified about the type of that term. The type of a *value* which is the residual of a given term is a *run-time type* for that term.

Thus the relation between a compile-time type and a run-time type is the same as the relation between a term and a value: a value for a term is any closed normal form obtained by performing reductions and substitutions over that term; a run-time type

corresponds to a loss of information). At type level this corresponds to the decrease in the type: the smaller the type, the more informative it is. For instance, think of the real and rational numbers ($\mathbf{Q} \leq \mathbf{R}$): if we know that a number is rational then we know that it possesses all the properties of real numbers AND some more (for example, it can be represented as the quotient of two integers)

[3] For example $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ are the only two values of type $Bool \equiv \alpha \to \alpha \to \alpha$.

for that term is the type of any of its values. Note that an open term is bound, during a computation, to many different values, and so it gets many different run-time types. Note also that we did not formally define the notion of "evolution of a term", thought it would be possible. We are now just trying to convey the intuition behind the idea of run-time types, while the formal definition of the reduction rules is given in the next section.

Although the selection of the branches of overloaded functions is based on the run-time types, the static typing of a term is enough to assure that the computation will be type-error free. This is a corollary of Theorem 5.2, which guarantees that types can only decrease during computation (so that the run-time type of any residual of a term is always a subtype of its compile-time type) and thus that well-typed terms rewrite to well typed terms. To guarantee this property a "consistency" condition must be imposed on overloaded types. In short, an overloaded type $\{U_i \to V_i\}_{i\in I}$ is well-formed if and only if for all $i, j \in I$ it satisfies the following conditions, where $U_i \Downarrow U_j$ means that U_i and U_j are downward compatible, i.e. they have a common lower bound:

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

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

Condition (1) is the consistency condition at issue, which assures that during computation the type of a term may only decrease. Informally, in view of our analogy "type–amount of information", it says that, if the input information given to an overloaded function increases, so does the information in the output. More specifically, in our approach, if we have a two-branched overloaded function M of type $\{U_1 \to V_1, U_2 \to V_2\}$ with $U_2 \leq U_1$ and we pass it a term N which has the compile-time type U_1 then the compile-time type of $M \bullet N$ will be (smaller than or equal to) 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 $V_2 \leq V_1$ must hold. The second condition concerns the selection of the correct branch: we said before that if we apply an overloaded function of type $\{U_i \to V_i\}_{i\in I}$ to a term of type U then the selected branch has type $U_j \to V_j$ such that $U_j = \min_{i\in I}\{U_i | U \leq U_i\}$; condition (2) assures the existence and uniqueness of this branch.

At first sight these restrictions may seem excessively complicated, and may discourage the reader with no experience in object-oriented languages. However, these restrictions are more obvious than they appear; especially with respect to the connection with object-oriented programming where they have a very natural interpretation (see Section 8).

3 The $\lambda\&$-calculus

In this section we define the extension of the typed lambda calculus we study in the rest of the paper. We use the following conventions: A, B denote Atomic Types, $S, T, U, V, W \ldots$ denote (Pre)Types, M, N, P, Q, denote Terms, H, I, J, K denote sets of indexes and h, i, j, k, n indexes. We first define a set of *Pretypes* and then from them we select those that satisfy the conditions above and that constitute the types.

$$\textbf{PreTypes} \qquad V \quad ::= \quad A \mid V \to V \mid \{V_1' \to V_1'', \ldots, V_n' \to V_n''\}$$

For technical reasons we consider overloaded types as lists, i.e. possessing an order; the list may also be empty: in this case the type is denoted by $\{\}$.

3.1 Subtyping rules.

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

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

$$\frac{\text{for all } i \in I, \text{ there exists } j \in J \text{ such that } U_i'' \leq U_j' \text{ and } V_j' \leq V_i''}{\{U_j' \to V_j'\}_{j \in J} \leq \{U_i'' \to V_i''\}_{i \in I}}$$

Intuitively, if we consider two overloaded types U and V as sets of functional types then the last rule states that $U \leq V$ if and only if for every type in V there is one in U smaller than it. In contrast to the usual partial order on record types, the cardinality of I and J are unrelated. Note that this is just a preorder, and not a partial order, as $U \leq V$ and $V \leq U$ do not imply $U = V$.

3.2 Types

Our system is an extended strongly typed λ-calculus. Arrow types and overloaded types are defined inductively from atomic types. As mentioned in the introduction, the overloaded types have a well formation rule that allows a consistent application of the reduction rules.

1. $A \in \textbf{Types}$
2. if $V_1, V_2 \in \textbf{Types}$ then $V_1 \to V_2 \in \textbf{Types}$
3. if for all $i, j \in I$
   - **(a)** $(U_i, V_i \in \textbf{Types})$ and
   - **(b)** $(U_i \leq U_j \Rightarrow V_i \leq V_j)$ and
   - **(c)** $(U_i \Downarrow U_j \Rightarrow$ there is a unique $h \in I$ such that $U_h = \inf\{U_i, U_j\})$[4]

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

[4] This notation is not very precise; since $\leq$ is just a preorder, a set generally has many equivalent g.l.b.'s; we should then write $U_h \in \inf\{U_i, U_j\}$, or $\{U_h\} = \inf\{U_i, U_j\}$.

In a system with subtyping, if $f: U \to V$, this means that when f is applied to a term a with a run-time type $U' \leq U$, the run-time type of the result will be a type $V' \leq V$. Intuitively, an overloaded type $\{U_i \to V_i\}_{i \in I}$ is inhabited by functions, made out of different pieces of code, such that when they are applied to a term whose run-time type U' is the subtype of some U_i, the run-time type of the result will be a subtype V' of the corresponding V_i. This is assured by condition **(b)** above.

To ensure the existence of an *inf* for any pair of downward compatible types, we require that $\leq$ yields a "partial lattice" on Atomic Types. In accordance with the rules given in the previous section, the whole **Types** inherits this structure.[5] In object-oriented languages this is not always the case. We can distinguish object-oriented languages where Atomic Types have a tree structure (the so called "single inheritance") and object-oriented languages where Atomic Types have a free order relation and where additional structure is used to solve the problems caused by compatible types without an *inf*. The same kind of technique can be used to extend our approach to this situation, since the partial lattice property is not essential, but is useful for getting a simple branch selection rule, as described in the section on reduction. Likewise, while condition **(b)** above is an essential feature of our approach, condition **(c)** is linked to the branch selection rule, and could easily be modified (see [Ghe91b]). Furthermore, we suppose that the subtyping relation is decidable on atomic types, which implies that it is decidable on **Types** as well. Note that this poses no problem in the current (simple) approach, as we have fixed atomic types; more work would be needed in order to allow programmer's definable base types.

Henceforth we only deal with **Types** and completely forget **PreTypes**; thus we will intend that all the pretypes which appear in the rest of the paper satisfy the conditions above.

3.3 Terms

Roughly speaking, terms correspond to terms of the classical lambda calculus plus operations to build and apply overloaded functions. Overloaded functions are built as customary with lists, by starting from an empty overloaded function and adding branches with the & operator. We distinguish the usual application $M \cdot M$ of lambda-calculus from the application of an overloaded function $M \bullet M$ since they constitute two completely different mechanisms: indeed a notion of variable substitution is associated with the former, while in the latter there is the notion of selection of a branch. This is also stressed by the proof-theoretical viewpoint where these constructors correspond to two different elimination rules. Finally, a further difference, specified in the reduction rules, is that overloaded application is associated with call by value, which is not needed by the ordinary application. For the same reason we must distinguish between the type $U \to V$ and the overloaded function type with just one branch $\{U \to V\}$.

However, in some cases it will be useful to have only one notation to deal with both kinds of application; for this aim the simple juxtaposition will be used.

[5] More precisely, since $\leq$ is not an order, it is **Types** modulo $\sim$ which inherits the partial lattice structure, where $T \sim U$ when $T \leq U$ and $U \leq T$.

Variables are indexed by their type, to avoid the use of type environments in the type-checking rules.

$$\textbf{Terms} \qquad M \quad ::= \quad x^V \mid c \mid \lambda x^V.M \mid M \cdot M \mid \varepsilon \mid M \&^V M \mid M \bullet M$$

The type which indexes the & is a technical trick to allow the reduction inside overloaded function, as explained later on. c represents generic constants while ε is a distinct constant for the empty overloaded function.

Hereafter, we may use the notation $\lambda x{:}V.M$ instead of $\lambda x^V.M$, and we may omit the type indexing of &, when not needed. Also the ε at the beginning of &-terms may be omitted, in examples.

3.4 Type checking

We define here the typing relation ":", a proper subset of **Terms**×**Types**. Therefore, as already pointed out, in the rules below we omit the condition $V \in$ **Types**. This means that, all the *PreTypes* that appear in the following rules are to be considered as well-formed types. Anyway we observe that an algorithm implementing the following type-checking rules should check that the types appearing in the conclusions of the rules [Taut], [$\to$ Intro] and [{}Intro] are well formed.

We use the notation $\vdash M{:}V \le U$ as a shorthand for the conjunction "$\vdash M{:}V$ and $V \le U$".

[Taut] $$\vdash x^V{:}V$$

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

[$\to$ Elim$_{(\le)}$] $$\frac{\vdash M{:}U \to V \quad \vdash N{:}W \le U}{\vdash M \cdot N{:}V}$$

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

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

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

In the last rule the premise on U_j as well as the type constraints are indeed *meta-premises*, i.e. they are conditions to the application of the rules but they do not belong to the tree-structure of the deduction. The empty term ε and the empty type {} are

used to start the formation of overloaded terms and types. We read $M\&N\&P$ as $(M\&N)\&P$.

As the careful reader will have noted, we do not use the *subsumption* rule (see below) in type-checking. We utilized a slightly different type discipline, where the use of subsumption is distributed where needed. The resulting system is equivalent, in the sense explained below, to the subsumption discipline, but every term possesses a unique type, which simplifies the definition of the operational semantics and some proofs.

Consider the functional core of our system, i.e. only the first three typing rules at the beginning of this section and let denote this system by $\vdash_{\leq}$. The subsumption system (denoted by $\vdash_{sub}$) is obtained from this one by replacing $\vdash N{:}W \leq U$ with $\vdash N{:}U$ in $[\rightarrow \textsc{Elim}_{(\leq)}]$ and by adding the subsumption rule:

$$[\rightarrow\textsc{Elim}] \quad \frac{\vdash_{sub} M{:}U \rightarrow V \quad \vdash_{sub} N{:}U}{\vdash_{sub} MN{:}V} \qquad [\textsc{Subsumption}] \quad \frac{\vdash_{sub} M{:}U \quad U \leq V}{\vdash_{sub} M{:}V}$$

Now, we can prove the following theorem.

Theorem 3.1 $\vdash_{\leq} M{:}V$ *iff* $V = \min\{U|\ \vdash_{sub} M{:}U\}$ *(which implies that the set* $\{U|\ \vdash_{sub} M{:}U\}$ *is not empty).*

Proof. ($\Rightarrow$) By induction on the proof of $\vdash_{\leq} M{:}V$ and by cases on the last applied rule.

($\Leftarrow$) By induction on the smallest proof that $\vdash_{sub} M{:}V$ and by cases on the last applied rule. □

Corollary 3.2 *Every well-typed* $\lambda\&$*-term possesses a unique type*

In conclusion, the theorem states that $\vdash_{\leq}$ is equivalent to $\vdash_{sub}$ in the sense that it always returns the smallest (i.e. most precise) type returned by the subsumption system. This theorem suggests that it is possible to define a subsumption based version $\vdash_{sub}$ for the full system too. We must add subsumption, substitute all judgements $\vdash N : T \leq U$ in the rules with $\vdash N : U$ and finally, in the $[\{\}\textsc{Elim}]$ rule, substitute $\vdash N : U$ with "U is the minimum type such that $\vdash_{sub} N : U$".

We can then extend theorem 3.1 to our entire calculus.

Theorem 3.3 (Subsumption Elimination) *For the whole* $\lambda\&$, $\vdash M{:}V$ *iff* $V = \min\{U|\ \vdash_{sub} M{:}U\}$

Since we have chosen the subsumption-free presentation, every term possesses a unique type, because there is a unique derivation for the type of a term.

In our presentation of the subtype rules we implicitly defined a transitivity rule. We can easily prove that this rule is not really needed.

Theorem 3.4 (Transitivity Elimination) $\vdash T \leq U$ *iff* $\vdash^{-} T \leq U$ *where* $\vdash^{-}$ *is defined by the rules of* $\vdash$ *minus transitivity.*

Proof. We first prove that $\vdash^- T \leq U$ and $\vdash^- U \leq V$ implies $\vdash^- T \leq V$. Observe that if $\vdash^- T \leq U$ then either T and U are both atomic types, or they are both arrow types or they are both overloaded types. We prove that $\vdash^- T \leq U$ and $\vdash^- U \leq V$ implies $\vdash^- T \leq V$ by induction on the size of T, U, V. If they are all atomic types the thesis is immediate. If $T = \{T'_i \to T''_i\}_{i \in I}$, $U = \{U'_j \to U''_j\}_{j \in J}$ and $V = \{V'_l \to V''_l\}_{l \in L}$, then for all $l \in L$ exists $j \in J$ such that $\vdash^- U'_j \to U''_j \leq V'_l \to V''_l$ and for all $j \in J$ exists $i \in I$ such that $\vdash^- T'_i \to T''_i \leq U'_j \to U''_j$. By induction, for all $l \in L$ exists $i \in I$ such that $\vdash^- T'_i \to T''_i \leq V'_l \to V''_l$, hence $\vdash^- \{T'_i \to T''_i\}_{i \in I} \leq \{V'_l \to V''_l\}_{l \in L}$, q.e.d.. The arrow case is similar and simpler. Now the theorem follows by induction on the proof of $\vdash T \leq U$ and by cases on the last applied rule, where the only interesting case is transitivity. □

Note that the lack of type variables makes the proof of subsumption elimination and transitivity elimination much easier for this calculus than for $F_{\leq}$ (see [CG92]).

3.5 Reduction Rules

In order to simplify the definition of the system, we consider the types of overloaded functions as ordered sets, where the order corresponds, more or less, to the order in which branches are added when an overloaded function is built. However the reader may note that this order is completely irrelevant in subtyping and typing rules, with the only exception of [{}Intro], where we want to be able to distinguish the only arrow type associated with the right hand side of the & from the set of the other ones. Exactly the same information is all that is needed by the reduction rules.

As we mentioned before, the run-time types are used during computation to perform branch selection. Thus, we have to define what the run-time type of a term is. We propose here a simple solution: the deduction system that infers the run-time type of a term is the same as the one used for type-checking. What distinguishes run-time types and compile-time types is thus the time when the deduction is made. In fact, during the computation the type of a term may change since reduction and substitution may decrease the type of a term (as shown in Theorems 5.1 and 5.2).

> We say that the type of a term is its run-time type when that term is a "value", i.e. when it is normal and closed; a run-time type of a residual of a term is also a run-time type of the term. We allow a reduction of the application of an overloaded function only when the argument is a value, i.e. when it is typed by a run-time type.

This is a crucial point. If we allowed selecting the branch of an overloaded function on the basis of the type of an argument whose type could still be decreased (by reduction or by substitution) then the selection would give different results depending on the time when it is applied, and the system would be no longer confluent.

As a matter of fact, this call-by-value constraint is not a limitation if our aim is to model object-oriented languages. In these languages message passing evaluation always

requires that the receiving object has been fully evaluated.[6]

We start by defining, in a standard way, substitutions on the terms of our system:

Definition 3.5 (Substitution) *We define the term $M[x^T := N]$ by induction on the structure of M:*

1. $x^T[x^T := N] \equiv N$
2. $y^S[x^T := N] \equiv y^S$ *if* $y^S \not\equiv x^T$
3. $\varepsilon[x^T := N] \equiv \varepsilon$
4. $(\lambda y^S.P)[x^T := N] \equiv (\lambda y^S.(P[x^T := N]))$ *where y is not free in N*
5. $(P \&^V Q)[x^T := N] \equiv ((P[x^T := N]) \&^V (Q[x^T := N]))$
6. $(PQ)[x^T := N] \equiv (P[x^T := N])(Q[x^T := N])$

□

Of course, this definition only makes sense when the type of N is a subtype of T. Note that in *5*, even if the types of the subterms change, the type of the whole term is always the same, since it is frozen in the index of the &; thus the selection of the branch does not depend on the *grade* of reduction of the &-term. This is a decisive point in our approach, which makes the system type-safe though reductions within an overloaded term are allowed.

We define the one-step reduction relation $\triangleright$ which is a proper subset of **Terms** $\times$ **Terms**. We denote by $\triangleright^*$ its reflexive and transitive closure, under the usual conditions (in β) to avoid free variables being captured:

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

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

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

context) If $M_1 \triangleright M_2$ then

$$(M_1 N) \triangleright (M_2 N) \quad (3)$$
$$(N M_1) \triangleright (N M_2) \quad (4)$$
$$(\lambda x^U.M_1) \triangleright (\lambda x^U.M_2) \quad (5)$$
$$(M_1 \& N) \triangleright (M_2 \& N) \quad (6)$$
$$(N \& M_1) \triangleright (N \& M_2) \quad (7)$$

[6] This happens not only for the essential reason we pointed out (the run-time type is generally only known for fully evaluated terms) but also since object oriented languages heavily rely on state and state-updating operations, and programs using updates are much more readable if eagerly evaluated.

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

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

However, in general, the number of branches of the overloaded function may be different from the number of arrows in the overloaded types, both since an overloaded function could begin with an application or with a variable, accounting for an initial segment of the overloaded type (they are just required to possess an overloaded type), and because of the subtyping relation used in the rule of $[\{\}\textsc{Intro}]$.

If we allowed $(\beta_\&)$ reductions with open or non normal arguments the system would not be confluent, since the type of an open or non normal argument can be different in different phases of the computation. For example, consider a term

$$(\lambda x^V.((P \&^{\{V \to V, U \to U\}} M) \bullet x^V)) \cdot (N^U)$$

with $U \leq V$ (we superscript terms with their types, like in N^U, to increase readability of examples). If the inner $\beta_\&$ reduction were performed with the x argument (which is not closed), the first P branch would be chosen, while if the outer β reduction is performed first then the term becomes:

$$(P \&^{\{V \to V, U \to U\}} M) \bullet N^U$$

and the second branch M is (correctly) chosen. In short, the argument of an overloaded application must be closed and normal to perform the evaluation, since this is the only case where its type cannot decrease anymore, and describes the value as accurately as possible.

Complementary to the idea of freezing the argument of an overloaded application to its normal form, is the use of the type which indexes the &s to freeze the type of &-terms. We outline two short examples to show the problems that arise with reduction and substitution inside &-terms without this index.

We suppose that $U' \leq U$ and $V' \leq V$; consider the term

$$F_1 \equiv (\varepsilon \& M^{U \to V} \& ((\lambda x^{U' \to V'}.x) \cdot N^{U \to V'}))$$

this can be intuitively typed as follows:

$$F_1 \colon \{U \to V, U' \to V'\}$$

Though, if we reduce the right branch, without freezing the index, we are no longer able to recover a type for the contractum, namely for $(\varepsilon \& M^{U \to V} \& N^{U \to V'})$ since both terms possess the same input type. Consider next the term

$$F_2 \equiv \lambda y^{U' \to V'}.(\varepsilon \& (\lambda x^U.M^V) \& (y^{U' \to V'}))$$

again we can intuitively type it as follows.

$$F_2 \colon (U' \to V') \to \{U \to V, U' \to V'\}$$

but if we apply F_2 to $N^{U\to V'}$ and β-reduce, then we are in the same case as above.

Note, finally, that our calculus is truly type-dependent (that is, the type erasure of a term is not enough to forecast its evolution or meaning, see the conclusion) for two different reasons:

- $\beta_\&$ reduction depends on the type of the argument
- $\beta_\&$ reduction depends on the index T of the & in the overloaded term

More specifically, if $\{U_i \to V_i\}_{i=1..n}$ is the index of &, $\beta_\&$ reduction depends on the list $[U_i]_{i=1..n}$ of the *input types* of the overloaded function. For example, if $U' \leq U$, both terms

$$(\varepsilon \& M^{U\to V} \&^{\{U\to V, U'\to V\}} M'^{U\to V})$$

and

$$(\varepsilon \& M^{U\to V} \&^{\{U'\to V, U\to V\}} M'^{U\to V})$$

are well typed, but they behave differently if applied to a normal closed term N^U.

Note that we are here in a different and more flexible situation than in object-oriented languages, since in those languages every branch of an overloaded function (every method) must be understood as a λ-abstraction (when viewing methods as global, overloaded functions in our sense.) In this language, on the other hand, any expression with a functional type (in particular an application) can be concatenated by using &. Thus, when following the object-oriented style, the left hand side U of the type $U \to V$ of an expression $\lambda x^U.M$ does not change when reductions and substitutions are performed inside $\lambda x^U.M$. In our approach, when reducing inside an &, one may obtain a smaller type for the reductum, in particular a larger U in a type $U \to V$. To allow this possibility of "inside" reductions and preserve determinism, we label the &'s with types .

4 Deriving records

In various approaches to object-oriented programming records play an important rôle. In particular, current functional treatments of object-oriented features formalize objects directly as records. Moreover, if records are not included in a calculus, the subtyping relation may be quite trivial. In our system, records can be encoded in a straightforward way.

Let $L_1, L_2, \ldots$ be an infinite list of atomic types. Assume that they are *isolated* (i.e., for any type V, if $L_i \leq V$ or $V \leq L_i$, then $L_i = V$), and introduce for each L_i a *constant* $\ell_i{:}L_i$. It is now possible to encode record types, record values and record selection, respectively, as follows:

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

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

Subtyping-rule

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

The type-checking rules are similarly derivable:

Type-checking

$$\text{[RECORD]} \quad \frac{\vdash M_1 : V_1 \ldots \vdash M_n : V_n}{\vdash \langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle : \langle\!\langle \ell_1 : V_1; \ldots; \ell_n : V_n \rangle\!\rangle}$$

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

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

$\rho)$ $\langle \ell_1 = M_1; \ldots; \ell_n = M_n \rangle.\ell_i \rhd M_i \quad (0 \leq i \leq n)$

recd) $M \rhd M' \Rightarrow M.\ell \rhd M'.\ell$ and $\langle \ldots \ell = M \ldots \rangle \rhd \langle \ldots \ell = M' \ldots \rangle$

5 The Generalized Subject Reduction Theorem

The Subject Reduction Theorem in classical λ-calculus proves that the type of a term does not change when the term is reduced. In this section, we generalize this theorem for our calculus, since we prove that if a term is typable in our system, then it can only be reduced to typable terms and that these terms have a type smaller than or equal to the type of the redex.

In order to enhance readability, in this and in the following section, we will often omit the turn-style symbol.

Lemma 5.1 (Substitution Lemma) *Let $x:T$, $M:U$, $N:T'$ and $T' \leq T$.*
Then $M[x := N]:U'$, where $U' \leq U$.

Proof. By induction on the structure of M.

$M \equiv \varepsilon$ straightforward

$M \equiv x$ straightforward

$M \equiv y \not\equiv x$ straightforward

$M \equiv \lambda x^V.M'$ straightforward

$M \equiv \lambda y^V.M'$ Then $U = V \to W$ and $M' : W$.

By induction hypothesis $M'[x := N] : W' \leq W$, therefore $M[x := N] \equiv \lambda y^V.M'[x := N] : V \to W' \leq V \to W$

$M \equiv (M_1 \&^T M_2)$ Then $M[x := N] \equiv (M_1[x := N] \&^T M_2[x := N])$; by induction hypothesis and the rule $[\{\}\textsc{Intro}]$ $M[x := N]$ is well typed and its type is the same of the one of M that is T.

$M \equiv M_1 \cdot M_2$ where $M_1 : V \to U$ and $M_2 : W \leq V$. By induction hypothesis:

$M_1[x := N] : V' \to U'$ with $V \leq V'$ and $U' \leq U$

$M_2[x := N] : W'$ with $W' \leq W$

Since $W' \leq W \leq V \leq V'$ we can apply the rule $[\to\textsc{Elim}_{(\leq)}]$ and thus $M[x := N] \equiv (M_1[x := N]) \cdot (M_2[x := N]) : U' \leq U$

$M \equiv M_1 \bullet M_2$ where $M_1 : \{V_i \to W_i\}_{i \in I}$ and $M_2 : V$.

Let $V_h = \min_{i \in I}\{V_i | V \leq V_i\}$. Thus $U = W_h$.

By induction hypothesis:

$M_1[x := N] : \{V'_j \to W'_j\}_{j \in J}$ with $\{V'_j \to W'_j\}_{j \in J} \leq \{V_i \to W_i\}_{i \in I}$

$M_2[x := N] : V'$ with $V' \leq V$

Let $V'_k = \min_{j \in J}\{V'_j | V' \leq V'_j\}$. Thus $M[x := N] : W'_k$. Therefore we have to prove that $W'_k \leq W_h$

As $\{V'_j \to W'_j\}_{j \in J} \leq \{V_i \to W_i\}_{i \in I}$ then for all $i \in I$ there exists $j \in J$ such that $V'_j \to W'_j \leq V_i \to W_i$. Given $i = h$ we chose an $\tilde{h} \in J$ which satisfies this condition: that is,

$$V'_{\tilde{h}} \to W'_{\tilde{h}} \leq V_h \to W_h \tag{8}$$

We now have the following inequalities:

$$V \leq V_h \tag{9}$$

by the definition of V_h, as $V_h = \min_{i \in I}\{V_i | V \leq V_i\}$;

$$V_h \leq V'_{\tilde{h}} \tag{10}$$

which follows from (8);

$$V' \leq V'_{\tilde{h}} \tag{11}$$

which follows from (9), (10) and $V' \leq V$;

$$W'_{\tilde{h}} \leq W_h \tag{12}$$

which follows from (8).

$$V'_k \leq V'_{\tilde{h}} \tag{13}$$

which follows from (11), as $V'_{\tilde{h}}$ belongs to a set with V'_k as least element. Finally,

$$W'_k \leq W'_{\tilde{h}} \tag{14}$$

follows from (13) and from the covariance rule on $\{V'_j \to W'_j\}_{j\in J}$

Thus, by (12) and (14), $W'_k \leq W_h$

□

Theorem 5.2 (Generalized Subject Reduction) *Let $M{:}U$. If $M \rhd^* N$ then $N{:}U'$, where $U' \leq U$.*

Proof. It suffices to prove the theorem for $\rhd$; the thesis follows from a simple induction on the number of steps of the reduction. Thus, we proceed by induction on the structure of M:

$M \equiv x$ x is in normal form and the thesis is straightforwardly satisfied.

$M \equiv \varepsilon$ as in the previous case.

$M \equiv \lambda x^V.P$. The only case of reduction is that $P \rhd P'$ and $N \equiv \lambda x^V.P'$; but from the induction hypothesis it follows that N is well-typed and the type of the codomain of N will be less than or equal to the one of M; since the domains are the same, the thesis thus holds.

$M \equiv (M_1 \&^T M_2)$. Just note that whenever M is reduced it is still well-typed (apply the induction hypothesis) and its type doesn't change.

$M \equiv M_1 \cdot M_2$ where $M_1{:}V \to U$ and $M_2{:}W \leq V$. We have three subcases:

1. $M_1 \rhd M'_1$, then by induction hypothesis $M'_1{:}V' \to U'$ with $V \leq V'$ and $U' \leq U$. Since $W \leq V \leq V'$, then by rule $[\to\text{ELIM}_{(\leq)}]$ we obtain $M'_1 M_2{:}U' \leq U$.
2. $M_2 \rhd M'_2$, then by induction hypothesis $M'_2{:}W'$ with $W' \leq W$. Again, $W' \leq W \leq V$ and, thus, by $[\to\text{ELIM}_{(\leq)}]$ we obtain $M_1 M'_2{:}U$.
3. $M_1 \equiv \lambda x^V.M_3$ and $M \rhd M_3[x := M_2]$, with $M_3{:}U$. Thus, by Lemma 5.1, $M_3[x := M_2]{:}U'$ with $U' \leq U$.

$M \equiv M_1 \bullet M_2$ where $M_1{:}\{V_i \to W_i\}_{i\in I}$ and $M_2{:}V$.

Let $V_h = \min_{i\in I}\{V_i | V \leq V_i\}$. Thus $U = W_h$. Again we have three subcases:

1. $M_1 \rhd M'_1$ then by induction $M'_1{:}\{V'_j \to W'_j\}_{j\in J}$ with $\{V'_j \to W'_j\}_{j\in J} \leq \{V_i \to W_i\}_{i\in I}$. Let $V'_k = \min_{j\in J}\{V'_j | V \leq V'_j\}$. Thus $M'_1 \bullet M_2{:}W'_k$. Therefore we have to prove that $W'_k \leq W_h$
   Since $\{V'_j \to W'_j\}_{j\in J} \leq \{V_i \to W_i\}_{i\in I}$, then for all $i \in I$ there exists $j \in J$ such that $V'_j \to W'_j \leq V_i \to W_i$. For $i = h$ we choose a certain $\tilde{h} \in J$ which satisfies this condition. That is:

$$V'_{\tilde{h}} \to W'_{\tilde{h}} \leq V_h \to W_h \tag{15}$$

We now have the following inequalities:

$$V \leq V_h \tag{16}$$

by hypothesis, since $V_h = \min_{i \in I}\{V_i | V \leq V_i\}$;

$$V_h \leq V'_{\tilde{h}} \tag{17}$$

follows from (15);

$$V \leq V'_{\tilde{h}} \tag{18}$$

follows from (16) and (17);

$$W'_{\tilde{h}} \leq W_h \tag{19}$$

follows from (15);

$$V'_k \leq V'_{\tilde{h}} \tag{20}$$

by (18), since $V'_{\tilde{h}}$ belongs to a set with V'_k as least element;

$$W'_k \leq W'_{\tilde{h}} \tag{21}$$

follows from (20) and the covariance rule on $\{V'_j \to W'_j\}_{j \in J}$
Finally, by (19) and (21), one has that $W'_k \leq W_h$

2. $M_2 \rhd M'_2$ then by induction hypothesis $M'_2 : V'$ with $V' \leq V$. Let $V_k = \min_{i \in I}\{V_i | V' \leq V_i\}$. Thus $M_1 \bullet M'_2 : W_k$. Since $V' \leq V \leq V_h$ then $V_k \leq V_h$; thus, by the covariance rule in $\{V_i \to W_i\}_{i \in I}$, we obtain $W_k \leq W_h$.
3. $M_1 \equiv (N_1 \& N_2)$ and M_2 is normal. Then we have two cases, that is $M \rhd (N_1 \bullet M_2)$ (case $h < n$) or $M \rhd (N_2 \cdot M_2)$ (case $h = n$). In both cases, by $[\{\}\textsc{Elim}]$ or $[\to\textsc{Elim}_{(\leq)}]$, according to the case, it is easy to show that the terms have type less than or equal to W_h.

□

6 Church-Rosser

In this section we prove that this system is Church-Rosser (**CR**). The proof is a simple application of a lemma due to Hindley [Hin64] and Rosen [Ros73]:

Lemma 6.1 (Hindley-Rosen) *Let R_1, R_2 be two notions of reduction. If R_1, R_2 are* **CR** *and $\rhd^*_{R_1}$ commutes with $\rhd^*_{R_2}$ then $R_1 \cup R_2$ is* **CR**.

Set now $R_1 \equiv \beta_\&$ and $R_2 \equiv \beta$; if we prove that these notions of reduction satisfy the hypotheses of the lemma above, we thus obtain **CR** for our system. It is easy to prove that β and $\beta_\&$ are **CR**: indeed, the first one is a well known result while for the other just note that $\beta_\&$ satisfies the diamond property.

Thus it remains to prove that the two notions of reduction commute, for which we need two technical lemmas.

Lemma 6.2 *If* $N \rhd^*_{\beta\&} N'$ *then* $M[x := N] \rhd^*_{\beta\&} M[x := N']$

Proof. The proof is done by induction on the structure of M and consists in a simple diagram chase

M	LHS	RHS	comment
ε	ε	ε	OK
x	N	N'	OK
y	y	y	OK
PQ	$P[\ ]Q[\ ]$	$P[\ ']Q[\ ']$	use the induction hypothesis
$\lambda y.P$	$\lambda y.P[\ ]$	$\lambda y.P[\ ']$	use the induction hypothesis
$(P\&Q)$	$(P[\ ]\&Q[\ ])$	$(P[\ ']\&Q[\ '])$	use the induction hypothesis

□

Lemma 6.3 *If* $M \rhd_{\beta\&} M'$ *then* $M[x := N] \rhd_{\beta\&} M'[x := N]$

Proof. We proceed by induction on the structure of $M \rhd M'$ (we omit the index since there is no ambiguity here); we have the following cases:

CASE 1 $\lambda y.P \rhd \lambda y.P'$ the thesis follows from the induction hypothesis on $P \rhd P'$.

CASE 2 $PQ \rhd P'Q$ the thesis follows from the induction hypothesis on $P \rhd P'$. The same for $QP \rhd QP'$, $P\&Q \rhd P'\&Q$ and $Q\&P \rhd Q\&P'$.

CASE 3 $(P_1\&P_2)Q \rhd P_iQ$ then

$$\begin{array}{rcll} M[x := N] & \equiv & (P_1[x := N]\&P_2[x := N])Q[x := N] & \\ & \equiv & (P_1[x := N]\&P_2[x := N])Q & \text{since } Q \text{ is closed} \end{array}$$

Since substitutions do not change the type in $(P_1\&P_2)$ (just recall that the type is fixed on the & and does not change during computation) then the selected branch will be the same for both $(P_1\&P_2)Q$ and $(P_1[x := N]\&P_2[x := N])Q$, thus:

$$\begin{array}{rcll} & \rhd & P_i[x := N]Q & \\ & \equiv & P_i[x := N]Q[x := N] & \text{since } Q \text{ is closed} \\ & \equiv & M'[x := N] & \end{array}$$

□

The next lemma shows that reductions are not context-sensitive: given a context $C[\ ]$, i.e. a lambda term with a hole, a reduction inside the hole is not affected by the context. This lemma will allow us to reduce the number of the cases in the next theorem:

Lemma 6.4 *Let* R *denote either* β *or* $\beta_\&$*; then for all contexts* $C[\ ]$ *if* $M \rhd^*_R N$ *then* $C[M] \rhd^*_R C[N]$

Proof. The proof is a simple induction on the context $C[\ ]$ □

Theorem 6.5 (Weak commutativity) *If* $M \rhd_\beta N_1$ *and* $M \rhd_{\beta\&} N_2$ *then there exists* N_3 *such that* $N_1 \rhd^*_{\beta\&} N_3$ *and* $N_2 \rhd^*_\beta N_3$

Proof. We proceed by induction on the structure of M. Since M is not in normal form, then $M \not\equiv x$ and $M \not\equiv \varepsilon$. In every induction step we will omit the (sub)cases which are a straightforward consequence of lemma 6.4:

1. $M \equiv \lambda x.P$. This case follows from lemma 6.4 and induction.
2. $M \equiv (M_1 \& M_2)$ then the only subcase which is not resolved by straightforward use of lemma 6.4 is $N_1 \equiv (M_1 \& M_2')$ and $N_2 \equiv (M_1' \& M_2)$ or symmetrically. But then $N_3 \equiv (M_1' \& M_2')$.
3. $M \equiv M_1 \bullet M_2$
   Subcase 1: $N_1 \equiv M_1 \bullet M_2'$ and $N_2 \equiv M_1' \bullet M_2$ or symmetrically. Thus $N_3 \equiv M_1' \bullet M_2'$
   The remaining cases are when $M_1 \equiv (P \& Q)$ and M_2 is closed and in normal form. Then we can have:
   Subcase 2: $N_1 \equiv (P' \& Q) M_2$ and $N_2 \equiv P M_2$ but then $N_3 \equiv P' M_2$
   Subcase 3: $N_1 \equiv (P \& Q') M_2$ and $N_2 \equiv Q M_2$ but then $N_3 \equiv Q' M_2$
   Subcase 4: $N_1 \equiv (P \& Q') M_2$ and $N_2 \equiv P M_2$ but then $N_3 \equiv N_2$
   Subcase 5: $N_1 \equiv (P' \& Q) M_2$ and $N_2 \equiv Q M_2$ but then $N_3 \equiv N_2$
   Note that in the last four cases we have used the property that the type of an &-term doesn't change when we reduce inside it and therefore the selected branch will be the same for the same argument.
4. $M \equiv M_1 \cdot M_2$ then as in the previous case we have:
   Subcase 1: $N_1 \equiv M_1 M_2'$ and $N_2 \equiv M_1' M_2$ or symmetrically. Thus $N_3 \equiv M_1' M_2'$
   The other cases are when M_1 is of the form $\lambda x.P$. Then we can have:
   Subcase 2: $N_1 \equiv P[x := M_2]$ and $N_2 \equiv (\lambda x.P) M_2'$ But $N_1 \rhd^*_{\beta\&} P[x := M_2']$ (by lemma 6.2) and $N_2 \rhd_\beta P[x := M_2']$. Thus $N_3 \equiv P[x := M_2']$.
   Subcase 3: $N_1 \equiv P[x := M_2]$ and $N_2 \equiv (\lambda x.P') M_2$ But $N_1 \rhd^*_{\beta\&} P'[x := M_2]$ (by lemma 6.3) and $N_2 \rhd_\beta P'[x := M_2]$. Thus $N_3 \equiv P'[x := M_2]$

□

Corollary 6.6 $\rhd^*_{\beta\&}$ *commutes with* $\rhd^*_\beta$

Proof. By lemma 3.3.6 in [Bar84]. □

Finally, by applying the HINDLEY-ROSEN lemma, we obtain that the calculus is **CR**.

7 Strong Normalization

7.1 The full calculus is not normalizing

The $\lambda\&$ calculus is not normalizing. Consider the following term, where $\emptyset$ is used instead of $\{\}$ to reduce the parenthesis nesting level, and where M stands for any term

of type $\{\emptyset \to \emptyset\}$, e.g. $M = (\varepsilon \,\&^{\{\emptyset\to\emptyset\}}\, \lambda x^{\emptyset}.\varepsilon)$:

$$\begin{array}{lcl} \mathit{Double} & = & (M \,\&^{\{\emptyset\to\emptyset,\{\emptyset\to\emptyset\}\to\emptyset\}}\, (\lambda x^{\{\emptyset\to\emptyset\}}.x{\bullet}x)) : \mathit{DType} \\ \mathit{DType} & = & \{\emptyset \to \emptyset, \{\emptyset \to \emptyset\} \to \emptyset\} \end{array}$$

Double is a $\lambda\&$ version of the untyped λ-term $\lambda x.xx$, coerced to a type *DType* such that it is possible to apply *Double* to itself. *Double* is well typed; in particular, $x{\bullet}x$ is well typed and has type $\emptyset$ as proved below:

$$[\{\}\textsc{Elim}] \qquad \frac{\vdash x : \{\emptyset \to \emptyset\} \quad \vdash x : \{\emptyset \to \emptyset\} \quad \emptyset = \min_{U \in \{\emptyset\}}\{U \mid \{\emptyset \to \emptyset\} \le U\}}{\vdash x{\bullet}x : \emptyset}$$

It may seem that the possibility to perform self-application is due to the existence of an empty overloaded type which is a maximum element in the set of all the overloaded types. This is not the case; actually, in the following proof of well-typing of $\mathit{Double}{\bullet}\mathit{Double}$, we may substitute $\emptyset$ with any other overloaded type.

$$[\{\}\textsc{Elim}] \qquad \frac{\begin{array}{l}\vdash \mathit{Double} : \mathit{DType} \quad \vdash \mathit{Double} : \mathit{DType} \\ \{\emptyset \to \emptyset\} = \min_{U \in \{\emptyset, \{\emptyset\to\emptyset\}\}}\{U \mid \{\emptyset \to \emptyset, \{\emptyset \to \emptyset\} \to \emptyset\} \le U\}\end{array}}{\vdash \mathit{Double}{\bullet}\mathit{Double} : \emptyset}$$

Now we can show that $\mathit{Double}{\bullet}\mathit{Double}$ has not a normal form as it reduces to itself:

$$\mathit{Double}{\bullet}\mathit{Double} \quad \triangleright_{\beta\&} \quad (\lambda x^{\{\emptyset\to\emptyset\}}.x{\bullet}x){\cdot}\mathit{Double} \quad \triangleright_{\beta} \quad \mathit{Double}{\bullet}\mathit{Double}$$

Simply typed lambda calculus prevents looping, essentially, by imposing a stratification between a function of type $T \to U$ and its argument, whose type T is "simpler" than the whole type $T \to U$; the same thing happens, in a subtler way, with system F.

When we add subtyping, the type T' of the argument of a function with type $T \to U$ is just a subtype of T, and may be, syntactically, much bigger than the whole $T \to U$: consider the case when T' is a record type with more fields that T. However, the *rank* of T' is still strictly smaller than that of $T \to U$, where the rank of an arrow type is at least the rank of its domain part plus one (for a correct definition see below). This happens, in short, since in $\lambda_{\le}$ and in $F_{\le}$ two types can be related by subtyping only when they have the same rank. Hence, $\lambda_{\le}$ and $F_{\le}$ are still strongly normalizing [Ghe90].

$\lambda\&$ typing does not prevent looping, essentially, since it allows to compare types with a different rank. In our example, we pass a parameter of type $\{\emptyset \to \emptyset, \{\emptyset \to \emptyset\} \to \emptyset\}$ (rank 2) to a function with domain type $\{\emptyset \to \emptyset\}$ (rank 1), and in the $x{\bullet}x$ case we pass a parameter of type $\{\emptyset \to \emptyset\}$ (rank 1) to a function with domain type $\{\}$ (rank 0). Hence, $\lambda\&$ typing does not prevent looping since it does not stratify functions w.r.t. their arguments.

However, when $\lambda\&$ is used to model object-oriented programming, it is always used in a stratified way. It is then interesting to define a stratified subsystem of $\lambda\&$ which is both strongly normalizing and expressive enough to model object-oriented programming. To this aim, we will prove the following theorem.

Theorem 7.1 *Let $\lambda\&^-$ be any subsystem of $\lambda\&$ closed by reduction and let* rank *be any function associating integers with $\lambda\&^-$ types. Assume also that, if T (syntactically) occurs in U, then* $\mathrm{rank}(T) \leq \mathrm{rank}(U)$. *If in $\lambda\&^-$, for any well typed application $M^T N^U$ one has* $\mathrm{rank}(U) < \mathrm{rank}(T)$, *then $\lambda\&^-$ is Strongly Normalizing.*

Example 7.2 We may obtain a subsystem of $\lambda\&$ with the properties of $\lambda\&^-$ in 7.1 either by restricting the set of types, or by imposing a stricter subtyping relation. We propose here two significant examples based on these restrictions: $(\lambda\&^-_{\mathrm{T}})$ and $(\lambda\&^-_{\leq})$, respectively. In either case, the *rank* function is defined as follows:

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

The idea is that, by restricting the set of types or the subtyping relation as described below, the types of a function and of its arguments are "stratified", namely the rank of the functional type is strictly greater than the rank of the input type, as required by theorem 7.1.

- $\lambda\&^-_{\leq}$ is defined by substituting $\leq$ in all $\lambda\&$ rules with a stricter subtyping relation $\leq^-$ defined by adding to any subtyping rule which proves $T \leq U$ the further condition $\mathit{rank}(T) \leq \mathit{rank}(U)$. In any well typed $\lambda\&^-_{\leq}$ 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 is strictly smaller than the rank of $\{T_i \to U_i\}_{i\in I}$; similarly for functional application. The subject reduction proof for $\lambda\&$ works for $\lambda\&^-_{\leq}$ too, thanks to the transitivity of the $\leq^-$ 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 $\mathit{rank}(T) = \mathit{rank}(U)$, 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 $\lambda\&^-_{\mathrm{T}}$. □

[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 only used only 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 $\varepsilon_{i\in\omega}$.

Theorem 7.1 and the examples show that there exist subsystems of $\lambda\&$ which are strongly normalizing and expressive enough for our purposes[8]. However we preferred to adopt the whole $\lambda\&$ as our target system, since it is easier to establish results such as Subject Reduction and Confluence on the wider system and apply them in subsystems rather than trying to extend restricted versions to more general cases.

In the following subsections we prove Theorem 7.1.

7.2 Typed-inductive properties

As is well known, strong normalization cannot be proved by induction on terms, since β reduction potentially increases the size of the reduced term. For this reason we introduce, along the lines of [Mit86], a different notion of induction on typed terms, called *typed induction*, proving that every typed-inductive property is satisfied by any typed $\lambda\&^-$ term. This notion is shaped over reduction, so that some reduction related properties, such as strong normalization or confluence, can be easily proved to be typed-inductive. Theorem 7.9, which proves that every typed-inductive property is satisfied by any typed $\lambda\&^-$ term, is the kernel of our proof and is related to the normalization proofs due to Tait, Girard, Mitchell and others. We had to avoid, though, the notions of saturated set and of logical relation, which do not seem to generalize easily to our setting. In this section we define a notion of "typed-inductive property" for $\lambda\&^-$ terms and show that every typed-inductive property is satisfied by any (well-typed) $\lambda\&^-$ term. Although many of the results and definitions in this section hold or make sense for $\lambda\&$ too, the reader should remember that all the terms, types and judgments in this section refer to a $\lambda\&^-$ system satisfying the conditions of Theorem 7.1.

Notation 7.3 *$M \circ N$ will denote $M \cdot N$ if $M : T \to U$ and $M \bullet N$ if $M : \{M_i \to N_i\}_{i=1\ldots n}$.*

Notation 7.4 *$\vec{M}$ denotes a list $[M_i]_{i=1,\ldots,n}$ of terms, possibly empty, and $N \cdot \vec{M}$ means $N \cdot M_1 \circ \ldots \circ M_n$; the same for $N \bullet \vec{M}$; if $\vec{M}$ is empty, $N \circ \vec{M}$ is just N.*

"$\vec{M}$ is well typed" means "each $M_i \in \vec{M}$ is well typed"; similarly for other predicates on terms.

Definition 7.5 *Let $\{\mathcal{S}^T\}_T$ be a family of sets of $\lambda\&^-$ terms, indexed over $\lambda\&^-$ types, such that:*

$$M \in \mathcal{S}^T \Rightarrow \vdash M : T.$$

$\mathcal{S}$ is typed-inductive *if it satisfies the following conditions*[9] *(where $M \in \mathcal{S}^{if}$ means "$M \in \mathcal{S}$ if M is well typed"):*

(x/c) $\forall x, \vec{N} \in \mathcal{S}.\ x \circ \vec{N} \in \mathcal{S}^{if}$
and similarly for constants and for ε.

[8] Strictly speaking these are not subsystems: we have excluded some types, thus two types that possessed a common lower bound in the full system may no longer possess it here. Therefore the condition **(c)** may be more easily satisfied and types that were not well formed may now satisfy all the condition of good formation

[9] We use $\mathcal{S}$ for $\{\mathcal{S}^T\}_T$. Furthermore, since any term M has a unique type T, we will write without ambiguity $M \in \mathcal{S}$ to mean $M \in \mathcal{S}^T$.

($\&_1$) $\forall M_1{\in}\mathcal{S}, M_2{\in}\mathcal{S}, N{\in}\mathcal{S}, \vec{N}{\in}\mathcal{S}.$
$M_1{\bullet}N{\circ}\vec{N} \in \mathcal{S}^{if} \wedge M_2{\cdot}N{\circ}\vec{N} \in \mathcal{S}^{if} \Rightarrow (M_1\&M_2){\bullet}N{\circ}\vec{N} \in \mathcal{S}^{if}$

(λ_1) $\forall M{\in}\mathcal{S}, N{\in}\mathcal{S}, \vec{N}{\in}\mathcal{S}.\ M[x:=N]{\circ}\vec{N} \in \mathcal{S}^{if} \Rightarrow (\lambda x{:}T.M){\cdot}N{\circ}\vec{N} \in \mathcal{S}^{if}$

($\&_2$) $\forall M_1{\in}\mathcal{S}, M_2{\in}\mathcal{S}.\ M_1\&M_2 \in \mathcal{S}^{if}$

(λ_2) $\forall M{\in}\mathcal{S}.\ \lambda x^T.M \in \mathcal{S}^{if}$

The $\mathcal{S}^{if}$ notation means that all the "$\in\mathcal{S}$" predicates in the above implications must only be satisfied only by typed preterms. This is difficult only in case $\&_1$: depending on whether $M_1{\bullet}\ldots$ is well-typed, $M_2{\cdot}\ldots$ is well-typed or both are well-typed, the first, the second or both are required to be in $\mathcal{S}$; indeed we want to take into account all the branches that could be selected not only the one that will be actually executed. For this reason we used in $\&_1$ a "$\wedge$" rather than a "$\vee$".

We aim to prove, by induction on terms, that every well-typed $\lambda\&^-$ term N belongs to $\mathcal{S}$. The conditions on typed induction allow an inductive proof of this fact for terms like $\lambda x^T.M$ and $M\&N$, but we have no direct proof that $(M{\in}\mathcal{S} \wedge N{\in}\mathcal{S}){\Rightarrow}(M{\circ}N{\in}\mathcal{S})$. For this reason we derive from $\mathcal{S}$ a stronger predicate $\mathcal{S}^*$ which allows term induction through application. We will then prove that $\mathcal{S}^*$ is not actually stronger than $\mathcal{S}$, since for any typed-inductive property $\mathcal{S}$:

$$M{\in}\mathcal{S}^{*\,T} \Leftrightarrow M{\in}\mathcal{S}^T \Leftrightarrow \vdash M{:}T.$$

The definition of $\mathcal{S}^*$ is the only part of the proof where we need the stratification by the rank function.

Notation 7.6 ($\widehat{[T_i]}_{i\in I}$) *For any list of types $[T_i]_{i\in I}$, $T'{\in}\widehat{[T_i]}_{i\in I} \Leftrightarrow \exists i{\in}I.T'{\leq}T_i$. Note that if $\vdash M{:}\{T_i \to U_i\}_{i\in I}$ and $\vdash N{:}T'$ then $M{\bullet}N$ is well typed iff $T'{\in}\widehat{[T_i]}_{i\in I}$.*

Definition 7.7 *For any typed-inductive property $\{\mathcal{S}^T\}_T$ its application closure on $\lambda\&^-$ terms $\{\mathcal{S}^{*\,T}\}_T$ is defined, by lexicographic induction on the rank and then on the size of T, as follows:*

(*atomic*) $M{\in}\mathcal{S}^{*A} \Leftrightarrow M{\in}\mathcal{S}^A$

($\to$) $M{\in}\mathcal{S}^{*\,T\to U} \Leftrightarrow M{\in}\mathcal{S}^{T\to U} \wedge \forall T'{\leq}T.\forall N{\in}\mathcal{S}^{*\,T'}.\ M{\cdot}N{\in}\mathcal{S}^{*\,U}$

($\{\}$) $M{\in}\mathcal{S}^{*\{T_i\to U_i\}_{i=1\ldots n}}$
$\Leftrightarrow M{\in}\mathcal{S}^{\{T_i\to U_i\}_{i=1\ldots n}} \wedge \forall T'{\in}\widehat{[T_i]}_{i=1\ldots n}.\forall N{\in}\mathcal{S}^{*\,T'}.\exists i{\in}[1..n].M{\bullet}N{\in}\mathcal{S}^{*\,U_i}$

In short:

$$M{\in}\mathcal{S}^* \Leftrightarrow M{\in}\mathcal{S} \wedge \forall N{\in}\mathcal{S}^*.M{\circ}N{\in}\mathcal{S}^{*if}$$

In the definition of $\mathcal{S}^*$, we say that M belongs to $\mathcal{S}^*$ by giving for granted the definition of $\mathcal{S}^*$ over the types of the N's such that $M{\circ}N$ is well typed and over the type of $M{\circ}N$ itself. This is consistent with the inductive hypothesis since:

1. The rank of the type of N is strictly smaller than the rank of the type of M in view of the conditions in Theorem 7.1.
2. Since the type U of $M \circ N$ strictly occurs in the type W of M, then the rank of U is not greater than the rank of W (by the conditions in Theorem 7.1). Hence the definition is well formed either by induction on the rank or, if the ranks of U and W are equal, by secondary induction on the size.

The next lemma shows, informally, that in the condition $M \in \mathcal{S}^* \Leftrightarrow \forall N \in \mathcal{S}^*.M \circ N \in \mathcal{S}^{*if}$ we can trade an $*$ for an $\vec{}$, since $\forall N \in \mathcal{S}^*.M \circ N \in \mathcal{S}^{*if} \Leftrightarrow \forall \vec{N} \in \mathcal{S}^*.M \circ \vec{N} \in \mathcal{S}^{if}$.

Lemma 7.8 *$M \in \mathcal{S}^* \Leftrightarrow M$ is well typed $\wedge \forall \vec{N} \in \mathcal{S}^*.M \circ \vec{N} \in \mathcal{S}^{if}$*

Proof.

($\Rightarrow$) "M is well typed" is immediate since $M \in \mathcal{S}^{*T} \Rightarrow M \in \mathcal{S}^T \Rightarrow \vdash M : T$.

$\forall \vec{N} \in \mathcal{S}^*.M \circ \vec{N} \in \mathcal{S}^{if}$ is proved by proving the stronger property $\forall \vec{N} \in \mathcal{S}^*.M \circ \vec{N} \in \mathcal{S}^{*if}$ by induction on the length of $\vec{N}$. If $\vec{N}$ is empty, the thesis is immediate. If $\vec{N} = N_1 \cup \vec{N}'$ then $M \circ N_1 \in \mathcal{S}^{*if}$ by definition of $\mathcal{S}^*$, and $(M \circ N_1) \circ \vec{N}' \in \mathcal{S}^{*if}$ by induction.

($\Leftarrow$) By definition, $M \in \mathcal{S}^* \Leftrightarrow M \in \mathcal{S} \wedge \forall N \in \mathcal{S}^*.M \circ N \in \mathcal{S}^{*if}$. $\forall \vec{N} \in \mathcal{S}^*.M \bullet \vec{N} \in \mathcal{S}^{if}$ implies immediately $M \in \mathcal{S}$: just take an empty $\vec{N}$. $M \circ N \in \mathcal{S}^{*if}$ is proved by induction on the type of M.

(atomic) $\vdash M : A$: $M \circ N$ is never well typed; $M \in \mathcal{S}^A$ is enough to conclude $M \in \mathcal{S}^{*A}$.

($\{\}$) $\vdash M : \{\}$: as above.

($\rightarrow$) $\vdash M : T \rightarrow U$: we have to prove that $\forall N \in \mathcal{S}^{*T'}, T' \leq T.M \cdot N \in \mathcal{S}^{*U}$. By hypothesis:

$$\forall \vec{N} \in \mathcal{S}^*.M \cdot N \circ \vec{N} \in \mathcal{S}^{if}$$

applying induction to $M \cdot N$, whose type U is smaller than the one of $T \rightarrow U$, we have that $M \cdot N \in \mathcal{S}^{*U}$.

($\{T_i \rightarrow U_i\}$) $\vdash M : \{T_i \rightarrow U_i\}_{i=1 \ldots n+1}$: as in the previous case.

□

Theorem 7.9 *If $\mathcal{S}$ is typed-inductive, then every term $\vdash N : T$ is in $\mathcal{S}^{*T}$.*

Proof. We prove the following stronger property: if N is well-typed and $\sigma \equiv [\vec{x^T} := \vec{N}]$ is a well-typed $\mathcal{S}^*$-substitution (i.e. for $i \in [1..n]$. $N_i \in \mathcal{S}^{*T_i'}$ and $T_i' \leq T_i$), then $N\sigma \in \mathcal{S}^*$; $\vec{x^T}$ is called the domain of $\sigma \equiv [\vec{x^T} := \vec{N}]$, and is denoted as $dom(\sigma)$.

It is proved by induction on the size of N. In any induction step, we prove $\forall \sigma.N\sigma \in \mathcal{S}^*$, supposing that, for any N' smaller than N, $\forall \sigma'.N'\sigma' \in \mathcal{S}^*$ (which implies $N'\sigma' \in \mathcal{S}$ and $N' \in \mathcal{S}$).

(c) $c\sigma \equiv c$. We apply lemma 7.8, and prove that $\forall \vec{N} \in \mathcal{S}^*.c \circ \vec{N} \in \mathcal{S}^{if}$. Since $\vec{N} \in \mathcal{S}^* \Rightarrow \vec{N} \in \mathcal{S}$ then $c \circ \vec{N} \in \mathcal{S}^{if}$ follows immediately from property (c) of $\mathcal{S}$.

(x) If $x \in dom(\sigma)$ then $x\sigma \in \mathcal{S}^*$ since σ is an $\mathcal{S}^*$-substitution. Otherwise, reason as in case (c).

($M_1 \& M_2$) By applying lemma 7.8 we prove that $\forall \sigma. \forall \vec{N} \in \mathcal{S}^*.(M_1 \& M_2)\sigma \circ \vec{N} \in \mathcal{S}^{if}$.

We have two cases. If $\vec{N}$ is not empty then $\vec{N} \equiv N_1 \cup \vec{N}'$. For any σ, $M_1\sigma \bullet N_1 \circ \vec{N}' \in \mathcal{S}^{if}$ and $M_2\sigma \cdot N_1 \circ \vec{N}' \in \mathcal{S}^{if}$ by induction (M_1 and M_2 are smaller than $M_1 \& M_2$). Then $(M_1 \& M_2)\sigma \bullet N_1 \circ \vec{N}' \in \mathcal{S}^{if}$ by property ($\&_1$) of $\mathcal{S}$.

If $\vec{N}$ is empty then $(M_1 \& M_2)\sigma \in \mathcal{S}$ follows, by property ($\&_2$) of $\mathcal{S}$, from the inductive hypothesis $M_1\sigma \in \mathcal{S}$ and $M_2\sigma \in \mathcal{S}$.

($\lambda x^T.M$) We will prove that $\forall \sigma. \forall \vec{N} \in \mathcal{S}^*.\ (\lambda x^T.M)\sigma \circ \vec{N} \in \mathcal{S}^{if}$, supposing, w.l.o.g., that x is not in $dom(\sigma)$.

We have two cases. If $\vec{N}$ is not empty and $(\lambda x^T.M)\sigma \circ \vec{N}$ is well typed then $\vec{N} \equiv N_1 \cup \vec{N}'$ and the type of N_1 is a subtype of T. Then for any $\mathcal{S}^*$-substitution σ, $\sigma[x^T := N_1]$ is a well-typed $\mathcal{S}^*$-substitution, since $N_1 \in \mathcal{S}^*$ by hypothesis, and then $M(\sigma[x := N_1]) \circ \vec{N}' \in \mathcal{S}^{if}$ by induction, which implies $(M\sigma)[x := N_1] \circ \vec{N}' \in \mathcal{S}^{if}$. Then $(\lambda x^T.M\sigma) \cdot N_1 \circ \vec{N}' \equiv (\lambda x^T.M)\sigma \circ \vec{N} \in \mathcal{S}^{if}$ by property (λ_1) of $\mathcal{S}$.

If $\vec{N}$ is empty, $(\lambda x^T.M)\sigma \in \mathcal{S}$ follows, by property (λ_2), from the inductive hypothesis $M\sigma \in \mathcal{S}$.

($M \circ N$) By induction $M\sigma \in \mathcal{S}^*$ and $N\sigma \in \mathcal{S}^*$; then $(M \circ N)\sigma \in \mathcal{S}^*$ by definition of $\mathcal{S}^*$.

This property implies the theorem since, as can be argued by case (x) of this proof, the identity substitution is a well-typed $\mathcal{S}^*$-substitution. □

Corollary 7.10 *If $\mathcal{S}$ is a typed-inductive property, every well-typed term satisfies $\mathcal{S}$ and its application closure:*

$$M \in \mathcal{S}^{*T} \quad \Leftrightarrow \quad M \in \mathcal{S}^T \quad \Leftrightarrow \quad \vdash M : T$$

Proof.

$M \in \mathcal{S}^{*T}$	$\Rightarrow$	$M \in \mathcal{S}^T$	by definition of $\mathcal{S}^*$.
$M \in \mathcal{S}^T$	$\Rightarrow$	$\vdash M : T$	by definition of typed induction.
$\vdash M : T$	$\Rightarrow$	$M \in \mathcal{S}^{*T}$	by theorem 7.9.

□

7.3 Strong Normalization is typed-inductive

In this section we prove Strong Normalization of $\lambda\&^-$ by proving that Strong Normalization is a typed-inductive property of $\lambda\&^-$ terms.

Consider the following term rewriting system *unconditional-*$\beta \cup \beta_\&$, which differs from $\beta \cup \beta_\&$ since unconditional-$\beta_\&$ reduction steps are allowed even if N is not normal or not closed, and the selected branch can be any of those whose input types is compatible with the type of the argument:

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

uncond.-$\beta_\&$) If $N : U \leq U_j$ then

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

Instead of proving Strong Normalization for $\lambda\&^-$ reduction, we prove Strong Normalization for unconditional-$\beta \cup \beta_\&$. Since any $\beta \cup \beta_\&$ reduction is also an unconditional-$\beta \cup \beta_\&$ reduction, Strong Normalization of the unconditional system implies Strong Normalization for the original one. Note that the proof of subject reduction is valid also when using *uncond-*$\beta_\&$ (the proof result even simpler) but that, even if the $\beta_\&$ conditions are not necessary to obtain strong termination, they are still needed to obtain confluence.

Notation 7.11 *If M is strongly normalizing, $\nu(M)$ is the length of the longest reduction chain starting from M. $\nu(\vec{M})$ is equal to $\nu(M_1) + \ldots + \nu(M_n)$.*

Theorem 7.12 *$\mathcal{SN}^T$, the property of being strongly normalizing terms of type T (according to the unconditional relation) is typed-inductive.*

Proof.

(x/c) $\forall \vec{N} \in \mathcal{SN}.\ x^U \circ \vec{N} \in \mathcal{SN}^{if}$

By induction on $\nu(\vec{N})$: if $x \circ \vec{N} \rhd P$ then $P = x \circ N_1' \circ \ldots \circ N_n'$ where just one of the primed terms is a one-step reduct of the corresponding non-primed one, while the other ones are equal. So $P \in \mathcal{SN}$ by induction on $\nu(\vec{N})$.

($\&_1$) $\forall M_1 \in \mathcal{SN}, M_2 \in \mathcal{SN}, N \in \mathcal{SN}, \vec{N} \in \mathcal{SN}.$
$M_1 \bullet N \circ \vec{N} \in \mathcal{SN}^{if} \wedge M_2 \cdot N \circ \vec{N} \in \mathcal{SN}^{if} \Rightarrow (M_1 \& M_2) \bullet N \circ \vec{N} \in \mathcal{SN}^{if}$

By induction on $\nu(M_1) + \nu(M_2) + \nu(N) + \nu(\vec{N})$.

If $(M_1 \& M_2) \bullet N \circ \vec{N} \rhd P$ then we have the following cases:

- ($\beta\&_l$) $P = M_1 \bullet N \circ \vec{N}$: since P is well-typed by subject-reduction, then $P \in \mathcal{SN}$ by hypothesis.
- ($\beta\&_r$) $P = M_2 \cdot N \circ \vec{N}$: as above.
- (congr.) $P = (M_1' \& M_2') \bullet N' \circ \vec{N}'$: $P \in \mathcal{SN}$ by induction on ν.

So $(M_1 \& M_2) \bullet N \circ \vec{N} \in \mathcal{SN}$ since it one-step reduces only to strongly normalizing terms.

(λ_1) $\forall M \in \mathcal{SN}, N \in \mathcal{SN}, \vec{N} \in \mathcal{SN}.\ M[x := N] \circ \vec{N} \in \mathcal{SN} \Rightarrow (\lambda x^T.M) \cdot N \circ \vec{N} \in \mathcal{SN}^{if}$

By induction on $\nu(M) + \nu(N) + \nu(\vec{N})$. If $(\lambda x^T.M) \cdot N \circ \vec{N} \rhd P$ we have the following cases:

(β) $P = M[x := N] \circ \vec{N}$: $P \in \mathcal{SN}$ by hypothesis.

(congr.) $P = (\lambda x^T.M') \cdot N' \circ \vec{N'}$ where just one of the primed terms is a one-step reduct of the corresponding one, while the other ones are equal: $P \in \mathcal{SN}$ by induction on ν.

($\&_2$) $\forall M_1 \in \mathcal{SN}, M_2 \in \mathcal{SN}.\ M_1 \& M_2 \in \mathcal{SN}^{if}$

By induction on $\nu(M_1) + \nu(M_2)$. If $M_1 \& M_2 \rhd P$ then $P \equiv M_1' \& M_2'$ where one of the primed terms is a one-step reduct of the corresponding one, while the other one is equal; then $P \in \mathcal{SN}$ by induction.

(λ_2) $\forall M \in \mathcal{SN}.\ \vdash \lambda x^T.M : T \to U \Rightarrow \lambda x^T.M \in \mathcal{SN}$

If $\lambda x^T.M \rhd \lambda x^T.M'$ then, since $\nu(M') < \nu(M)$, $\lambda x^T.M' \in \mathcal{SN}$ by induction on $\nu(M)$. So $\lambda x^T.M \in \mathcal{SN}$.

□

The last proof can be easily extended to show that the reduction system remains strongly normalizing if we add the following extensionality rules:

(η) $\lambda x^T.M \cdot x \rhd M$ if x is not free in M

($\eta_\&$) $M \& (\lambda x^T.M \bullet x) \rhd M$ if x is not free in M

Theorem 7.1 is now a corollary of Theorem 7.12 and of Corollary 7.10.

8 Overloading and Object-Oriented Programming

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

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

fact in detail. We use *italics* to distinguish class types from the usual types, and $\doteq$ to declare a class type and to give it a name; we will use $\equiv$ to associate a name with a value (e.g. with a function). Thus for example we can declare the following class types:

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

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

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

whose type is $\{\mathit{2DPoint} \to \mathrm{Real}, \mathit{3DPoint} \to \mathrm{Real}\}$.

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

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

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

whose type is $\{\mathit{2DPoint} \to \mathit{2DPoint}, \mathit{3DPoint} \to \mathit{3DPoint}\}$. Here covariance arises quite naturally.[10] In object-oriented jargon, covariance says that an overriding method must return a type smaller than the one returned by the overriden one.

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

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

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

[10] In the example the notation we used is quite cumbersome since we did not use field update operations on records like those of [CM91] or [Wan91]. Such operations may be derived in our system, by exploiting the & operator and by a clever use of explicit coercions: see [Cas92].

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

In object-oriented terms, this happens since *2DColPoint*, as a subtype of both *2DPoint* and *Color*, inherits the *Erase* method from both classes. Since there is no reason to choose one of the two methods and no general way of defining a notion of "merging" for inherited methods, we ask that this multiply inherited method is explicitly redefined for *2DColPoint*. Note that some object-oriented languages do not force this redefinition, but use some different criterion to choose from inherited methods, usually related to the order in which class definitions appear in the source code. As discussed in [Ghe91b], our rule 3.c in Section 3.2 can be easily substituted to model these different approaches to the problem of choosing between inherited methods, allowing a formalization and a comparison of these approaches in a unique framework. The approach we have chosen in this foundational study is just the simplest one in a context where the set of atomic types is fixed.

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

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

which has type:

$$
\begin{array}{ll}
\{ & \mathit{2DPoint} \to \mathit{2DPoint}, \\
& \mathit{3DPoint} \to \mathit{3DPoint}, \\
& \mathit{Color} \to \mathit{Color}, \\
& \mathit{2DColPoint} \to \mathit{2DColPoint} \quad \}
\end{array}
$$

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

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

```
class 3DPoint is 2DPoint and
  state      z:Int
  methods    Norm = sqrt(self.x^2+self.y^2+self.z^2);;
  interface  Norm: Real
endclass


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

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

8.1 Inheritance

Inheritance is the ability to define the state, interface and methods of a class "by difference" with respect to another class; inheritance on methods is the most important one. In the record based model, inheritance is realized using the record concatenation operation to add to the record of the methods of a superclass the new methods defined in the subclass. However, the recursive nature of the hidden *self* parameter forces one to distinguish between the "generator" associated with a class definition, which is essentially a version of the methods where *self* is a visible parameter, from the finished method set, obtained by a fix point operation which transforms *self* into a recursive pointer to the object which the methods belong to. This operation is called "generator wrapping". Inheritance may be defined by record concatenation over generators.[11]

To be able to reuse a generator, the type of *self* parameter must not be fixed: it must be a type variable that will assume as value the type for which the generator is reused. A first approach is to consider the type of self as a parameter itself; let us call it *Likeself*. In this case, if this "recursive type" appears in the result type of some method, then, when a generator is wrapped, the same operation must be performed on the type, to bind *Likeself* to the type of the class under definition, hence we need a fix point operator at the type level too. If, furthermore, there is some binary method, then *Likeself* must be linked to the type of the class under definition on the left hand side of arrows too. But, if a generator G has such a binary method, and a generator G' is obtained by extending G, then the type obtained by wrapping G' is not a subtype of the one

[11] To be fair, we must note that the generator based approach may account for the special identifier *super* used in object-oriented languages to refer to a method as it is implemented in a superclass, while we do not have this possibility in our system.

obtained by wrapping G, as explained in more detail below. Hence, subtyping cannot be used to write functions operating on objects corresponding to both G and G', but F-bounded polymorphism must be introduced. F-bounded polymorphism is essentially a way of quantifying over all types obtained by wrapping an extension of a generator F. For an account of this approach see for example [CCH+89, CHC90, Mit90, Bru91].

The feeling is that in the approach outlined above, recursion is too heavily used. An approach close to the previous one but that avoids the use of recursive types has been recently proposed in [PT93]. The idea is to separate the state of an object from its methods and then encapsulate the whole object by existentially quantifying over the type of the state. The type of a method that works on the internal state does not need to refer to the type of the whole object (as in the previous approach) but only to its state part; therefore recursive types are no longer needed. The type of the state is referred by a type variable since it is the abstract type of the existential quantification. The whole existential type is passed to the generator as in the previous case but without any use of recursive types. Finally, the behavior of F-bounded polymorphism is obtained by a clever use of higher order quantification.

Our approach to method inheritance is even simpler since we also separate the state from the methods. In our system, every subtype of a type inherits all the methods of its supertypes, since every overloaded function may be applied to every subtype of the types which the function has been explicitly written for. Moreover, the behavior of an inherited method M appearing as a branch of an overloaded function (i.e. a message) N can be overridden, i.e. defined in a way which is specific for a subtype T, by defining a branch for T inside the overloaded function N. Finally, new methods may be defined for a subtype by defining new overloaded functions. By this, we may say that, in our system, inheritance is given by subtyping plus the branch selection rule. This can be better seen by an example: suppose to have a message for which a method has been defined in the classes $U_1 \ldots U_n$; thus this message denotes an overloaded function of type $\{U_i \to T_i\}_{i=1..n}$ for some T_i's. When this overloaded function is applied to an argument, the *branch selected* is the one defined for the class $\min_{i=1..n}\{U_i | U \leq U_i\}$, where U is the class (type) of the argument. If this minimum is exactly U, this means that the receiver uses the method that has been defined in its class; otherwise, i.e. if this minimum is strictly greater, then the receiver uses the method that its class, U, has *inherited* from this minimum (a superclass); in other terms, the code written for the class which resulted to be the minimum, is *reused* by the objects of the class U.

The reader should note that, although our system has a static nature (the set of atomic types is fixed), it is possible to extend it to a dynamic one, along the lines drawn in [Ghe91b].

8.2 Binary methods and multiple dispatch

Let us now see the problem with binary methods in greater detail. Let us see what happens in the "objects as records" analogy: if we add a method *Equal* to *2DPoint* and *3DPoint* then, in the notation typical of formalisms built around this analogy, we obtain the following recursive record types (we forget the other methods):

$2DEPoint \equiv \langle\langle x : \mathrm{Int}; y : \mathrm{Int}; Equal : 2DEPoint \to \mathrm{Bool}\rangle\rangle$

$3DEPoint \equiv \langle\langle x : \mathrm{Int}; y : \mathrm{Int}; z : \mathrm{Int}; Equal : 3DEPoint \to \mathrm{Bool}\rangle\rangle$.

The two types are not comparable because of the contravariance of the arrow type in *Equal*: since one would expect $2DEPoint$ to be larger, as a record, than $3DEPoint$, the type at the left of the outer arrow in $2DEPoint$ should be larger, which is impossible by contravariance.[12] Note that this should not be considered a flaw in the system but a desirable property, since a subtyping relation between the two types, in the record based approach, could cause a run-time type error (see [CL91] for an example). Hence, there is an apparent incompatibility between the covariant nature of most binary operations and the contravariant subtyping rule of arrow types.

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

$$Equal\colon \{2DPoint \to (2DPoint \to \mathrm{Bool}),\ \ 3DPoint \to (3DPoint \to \mathrm{Bool})\}$$

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

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

the function above has type:

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

which is well formed[13].

In the presence of a subtyping relation, the covariance versus contravariance of the arrow type, w.r.t. the left argument (domain), is a delicate and classical debate. Semantically (categorically) oriented people have no doubt: the hom-functor is contravariant in the first argument. Moreover, this nicely fits with typed models constructed over type-free universes, where types are subsets or subrelations of the type-free structure, and type-free terms model runtime computations. Also the common sense of the type-checking forces contravariance: if we consider one type a subtype of another if and only if all expressions of the former type can be used in the place of expressions of the latter, then a function $g : T \to U$ may be substituted by a function f only if the domain of f is *greater* than T. However, practitioners often have a different attitude. In OOP, in particular, the "overriding" of a method by one, say, with a smaller domain

[12] Recursive types should be considered as denotations for their infinite expansion, and an infinite type is a subtype of another one when all the finite approximations of the first one are subtypes of the corresponding finite approximation of the second one; see [AC90].

[13] This is not surprising as, even if the types of the two versions of equal are componentwise isomorphic, in general isomorphisms of types do not preserve subtyping.

(input type) leads to a smaller codomain (output type), in the spirit of a "preservation of information". Indeed, in our approach, we show that both viewpoints are correct, when adopted in the "right" context.

In fact, our general arrow types (the types of ordinary functions) are contravariant in the first argument, as required by common sense and mathematical meaning. However, the *families* of arrow types which are glued together in overloaded types form covariant collections, by our conditions on the formation of these types (see 3.2). Besides the justification of this at the end of Section 2, consider the practice of overriding. The implementation of a method in a superclass is substituted by a more specific implementation in a subclass; or, more precisely, overriding methods must return smaller or equal types than the overridden one. For example, the "+" operation, on different types, may be given by two different implementations: one implementation of type $Int \times Int \to Int$, the other of type $Real \times Real \to Real$. In our approach, we can glue these implementations together into one global method, precisely because their types satisfy the required covariance condition.

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

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

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

9 Conclusion: intersections, products and their semantics

This work is only the starting point of a new type discipline to be more extensively explored. We believe that we have proposed here a sound solution to the use of contravariant arrow types and of covariant ones in programming: the "purely functional" or external arrows are contravariant, in the first argument, while overloaded functions, as inspired by our understanding of message passing and methods in object-oriented programming, yield covariant families of arrow types. It should also be clear that our language essentially models message passing and inheritance, by the use of overloaded application and subtyping, as described in the previous sections, thus avoiding unnecessary use of recursion.

We have not dealt, though, with abstract data types nor with incremental class definitions: this may be a matter for future extensions. We have tried to present our perspective and motivations in the introduction, by stressing the need to found the so called "ad hoc" polymorphism onto decent mathematical grounds, in particular in view of its role in the understanding of the object-oriented features mentioned above. Reference has been given to the work we are aware of in this subject, all of which has, in fact, a quite different perspective.

As for the Type Theory proposed, one has first to stress that that "terms depending on types" is a concept entirely different from "types depending on terms", (as described in the —first order— types of Martin-Löf type theory or of the Calculus of Constructions). One should also quote possible connections with other type disciplines, in particular, the intersection types, originated in [CDCV81]. Indeed, an overloaded type, in our sense, is strictly related to an intersection of types: recent applications of intersection types in [Pie90] and in the programming language Forsythe support this analogy [Rey88]. However, the two notions are slightly different. An intersection type $T \cap U$ is a type whose elements can play both the role of an element of type T and of an element of type U, and this is the case for our overloaded types too. In the case of intersection, though, types, a coherence condition is imposed too, which means, essentially, that when a value can play different roles, we are free to choose any of these roles, without affecting the result of the computation [Rey91]. In our context, this is not the case; a programmer may define an overloaded function "foo" of type $\{Int \times \mathrm{Int} \to Int, \mathrm{Real} \times Real \to \mathrm{Real}\}$ which sums two integers but multiplies two real numbers, while in a coherent intersection type discipline an overloaded function with that type should behave in a consistent way on integers and reals.

Consider now higher order systems and observe that in Girard's system F (Reynolds' second order λ-calculus, [GLT89]), second order terms may be fed with input types. These terms then may seem to express an explicit type dependency as the one we tried to formalize in this paper. We fully understand now that it is not so. On the model theoretical side, this is made clear by the interpretation of second order product types as intersections (in view of the semantic relation between intersection and dependent products given in [LM91], this is a delicate issue: see [Lon94] for a discussion). From the point of view of proof theory, the low expressiveness of terms which may take types as inputs is explained by early intuitions of Reynolds on parametricity and the work in [MR91], [ACC93] and [LMS93]. Intuitively, Reynolds Abstraction Theorem says that a term, taking as inputs two "related" types, gives "related" terms as outputs and the Genericity Theorem in [LMS93] shows that if two terms coincide on one input type then they coincide on all input types. The moral is that in system F, as a properly second order logical system, one cannot have terms whose output values truly depend on input types.

For all these reasons, we had to design a completely new language: no tools are available from (constructive) logic to express the simple fact of practice that the value of a term may depend on a type as input. We decided, consistently with the practice of object-oriented programming, to allow only a "finite dependency": overloaded terms are finitely branching terms and the essential richness of the discipline is largely due to the

use of subtyping, which fulfills a quantification over an infinity of types. However, the way is open to further strengthening, once we set the safe grounds of a few, but crucial, consistency properties (Church-Rosser, Strong Normalization, Subject Reduction).

In particular, one may think to allow type variables and second order λ-abstraction, also in overloaded types and terms (e.g. allow $\lambda X.(\ldots \&^X \ldots)$) and study how the covariance constraints and the syntactic properties are transformed in this case. Explicit polymorphism would then be entirely revised as type dependency would be as uniform and as effective as ever. The border line, though, between safe systems and inconsistencies would then become narrow: the technical difficulties of the normalization theorem, at our propositional level, may suggest the major obstacles one may encounter in defining "sound" higher order systems.

We plan to explore this direction as well as three less ambitious projects. First the semantics of $\lambda\&$, a non trivial matter even in the propositional case (a preliminary proposal is in [CGL93]). Second, the application of the ideas of this calculus to the implementation of a prototypical object-oriented language. Third, a more detailed investigation of "compile-time vs. run-time" types. In this paper we proposed a simple view of this "dualism", which fits our approach. More should be said, though, in particular regarding subtyping, coercions, etc., i.e. the various ways of dealing with "types evolving during computations".

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

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

References

[AC90] R. Amadio and L. Cardelli. Subtyping recursive types. Technical report, Digital System Research Center, August 1990.

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

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

[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.

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

[CG92] P. L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and the type checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1), 1992.

[CG93] P.-L. Curien and G. Ghelli. Confluence and decidability of $\beta\eta\text{top}_{\leq}$ reduction in $\text{F}_{\leq}$. *Information and Computation*, 1993. To appear.

[CGL93] G. Castagna, G. Ghelli, and G. Longo. A semantics for $\lambda\&$-*early*: a calculus with overloading and early binding. In M. Bezem and J.F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664 in LNCS, pages 107–123, Utrecht, The Netherlands, March 1993. Springer-Verlag. TLCA'93.

[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] G. Castagna and G. Longo. From inheritance to Quest's type theory. In *Ecole Jeunes Chercheurs du GRECO de Programmation*, Sophia-Antipolis (Nice), April 1991. Talk given at the 5th Jumelage on Typed Lambda Calculus - Paris - January 1990.

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

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

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

[Ghe90] G. Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism.* PhD thesis, Dipartimento di Informatica, Università di Pisa, March 1990. Tech. Rep. TD-6/90.

[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.

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

[Hin64] R. Hindley. The Church-Rosser property and a result of combinatory logic. Dissertation, 1964. University of Newcastle-upon-Tyne.

[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. In *8th Annual IEEE Symposium on Logic in Computer Science*, Montreal, June 1993.

[Lon94] G. Longo. Parametric and type-dependent polymorphism. *Fundamenta Informaticae*, 1994. To appear.

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

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

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

[MR91] Q.Y. Ma and J. Reynolds. Types, abstractions and parametric polymorphism, part II. In *MFCS*. LNCS, Springer-Verlag, 1991.

[Pie90] B. Pierce. Intersection and union types. Technical report, Carnegie Mellon University, 1990.

[PT93] B.C. Pierce and D.N. Turner. Object-oriented programming without recursive types. In *10th Ann. ACM Symp. on Principles of Programming Languages*. ACM-Press, 1993. To appear in Journal of Functional Programming.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in LNCS, pages 675–700. Springer-Verlag, September 1991.

[Ros73] B. K. Rosen. Tree manipulation systems and Church-Rosser theorems. *Journal of ACM*, 20:160–187, 1973.

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

[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.