Note

Unifying overloading and λ-abstraction: $\lambda^{\{\}}$

Giuseppe Castagna[*]

Abstract

In this short note we present a minimal system to deal with overloaded functions with late binding, in which λ-abstractions are seen as a special case of overloaded functions with just one code. We prove some relevant properties that this system enjoys, and show its connection with the λ&-calculus. We end by showing the practical interest of this system, in particular in modeling object-oriented languages.

1 Introduction

In [6] we defined, together with Giorgio Ghelli and Giuseppe Longo, the λ&-calculus, an extension of the simply typed λ-calculus with overloaded functions, subtyping and late binding. We used it to study the foundation of object-oriented languages.

In λ& overloaded functions are built by concatenating lambda abstractions. The application of an overloaded function is syntactically distinguished from the one of ordinary functions (λ-abstractions).

In this note we present a minimal systems, the $\lambda^{\{\}}$-calculus, in which there is a unique operation of abstraction, as well as a unique application. The idea is to have only overloaded functions and to consider an ordinary function as an overloaded function for which only one code has been defined. The problem in doing that is that, in λ&, to implement late binding, i.e. the selection of the code of an overloaded function by using the most precise information on the type of the argument, overloaded functions have to use the call-by-value strategy. If we want to faithfully reproduce β-reduction, no particular evaluation strategy must be imposed. In $\lambda^{\{\}}$ this is obtained by the definition of a particular notion of reduction, we call ζ.

Although we initially defined $\lambda^{\{\}}$ mainly for theoretical purposes, namely the unification of overloaded and ordinary functions, it recently turned out that it has also a practical interest concerning object-oriented languages.

This note is organized as follows. In Section 2 we briefly recall the definition of the λ&-calculus. In Section 3 we present the $\lambda^{\{\}}$-calculus. Section 4 is devoted to the properties of the $\lambda^{\{\}}$-calculus: we show how $\lambda^{\{\}}$ is related to λ&, we prove the soundness of $\lambda^{\{\}}$ and of its type system. We end our exposition by showing, in Section 5, the practical interest of this system.

2 The λ&-calculus

In this section we briefly survey the λ&-calculus. The definition we present here is slightly different from the one in [6]. This variant has been studied in [3] and definitely adopted in [7]. For a more detailed discussion the reader may refer to [6, 5, 7].

An overloaded function is formed by a set of ordinary functions (i.e. lambda-abstractions), each one constituting a different *branch* of code. Overloaded functions are built as lists, starting by an *empty* overloaded function denoted by ε, and concatenating new branches by means of &; therefore an overloaded function with n branches M_i is written as $((\ldots((\varepsilon\&M_1)\&M_2)\ldots)\&M_n)$. The type of an

[*]CNRS. Département de Mathématiques et d'Informatique. École Normale Supérieure. 45 rue d'Ulm. 75230 Paris Cedex 05.

overloaded function is the set of the types of its branches. Thus if $M_i{:}S_i \to T_i$ then the overloaded function above has type $\{S_1 \to T_1, S_2 \to T_2, \ldots, S_n \to T_n\}$ (we also use the notation $\{S_i \to T_i\}_{i \leq n}$.) The application of an overloaded function is denoted by "$\bullet$". If we apply the function above to an argument N of type S then we select the branch whose S_i "best approximates" the type of the argument; i.e. we select the branch j such that $S_j = \min_{i=1..n}\{S_i | S \leq S_i\}$. And thus

$$(\varepsilon \& M_1 \& \ldots \& M_n) \bullet N \rhd^{+} M_j \cdot N \qquad (*)$$

where $\rhd^{+}$ means "reduces in one or more steps to".

A set of arrow types $\{S_h \to T_h\}_{h \in H}$ is an overloaded type if and only if, for all $i, j \in H$ the following conditions are satisfied:

$$\text{if } S_i \leq S_j \text{ then } T_i \leq T_j \qquad (1)$$

$$\text{for every } S \text{ maximal in } LB(S_i, S_j) \text{ there exists a unique } h \in H \text{ such that } S_h = S \qquad (2)$$

The first condition assures the type safety of the system while the latter is a necessary and sufficient condition to the existence of a "best approximating" branch in every application of an overloaded function.

The features above model overloading. It remains to include *late binding*. This can be done simply by requiring that a reduction as (*) can be performed only if N is a closed normal form.

The formal description of the calculus is given by the following definitions:

PreTypes $\qquad T \quad ::= \quad A \mid T \to T \mid \{T \to T, \ldots, T \to T\}$

Subtyping
We define a partial order on the pretypes starting from a given order for the atomic types and we extend it to higher pretypes in the following way:

$$\frac{S_2 \leq S_1 \quad T_1 \leq T_2}{S_1 \to T_1 \leq S_2 \to T_2} \qquad \frac{\forall i \in I, \exists j \in J \quad S_j \to T_j \leq U_i \to V_i}{\{S_j \to T_j\}_{j \in J} \leq \{U_i \to V_i\}_{i \in I}}$$

Types
A pretype is a type if all the overloaded pretypes that occur in it satisfy the conditions (1) and (2). We denote by **Types** the set of types.

Terms

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

where $T \in$ **Types**. The type indexing the & is used for the selection of the branch in overloaded application and to type check overloaded functions.

Typing algorithm
For the sake of conciseness, we describe only the algorithmic typing rules. These rules return the minimum of the types inferred for a term by the system that use the subsumption rule, and whose definition is given in [7].

[TAUT] $\qquad x^T : T \qquad$ [TAUT$_\varepsilon$] $\qquad \varepsilon : \{\}$

[$\to$INTRO] $\qquad \dfrac{M : T}{\lambda x^S.M : S \to T} \qquad$ [$\{\}$INTRO] $\qquad \dfrac{M : W_1 \leq \{S_i \to T_i\}_{i<n} \quad N : W_2 \leq S_n \to T_n}{(M \&^{\{S_i \to T_i\}_{i \leq n}} N) : \{S_i \to T_i\}_{i \leq n}}$

[$\to$ELIM$_\leq$] $\qquad \dfrac{M : S \to T \quad N : W \leq S}{M \cdot N : T} \qquad$ [$\{\}$ELIM] $\qquad \dfrac{M : \{S_i \to T_i\}_{i \in I} \quad N : S \quad S_j = \min_{i \in I}\{S_i | S \leq S_i\}}{M \bullet N : T_j}$

The condition that every pretype appearing in a term must be a type, assures that the typing relation ":" defined by the rules above is a subset of **Terms**$\times$**Types** (i.e. terms are typed by types).

Reduction

The reduction $\rhd$ is the *compatible closure* of the following *notion of reduction* (for definitions see [1]):

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

($\beta_\&$) If $N{:}S$ is closed and in normal form, and $S_j = min_{i\leq n}\{S_i \mid S \leq S_i\}$ then

$$(M_1\&^{\{S_i\to T_i\}_{i\leq n}}M_2)\bullet N \rhd \begin{cases} M_1\bullet N & \text{for } j < n \\ M_2\cdot N & \text{for } j = n \end{cases}$$

For the $\lambda\&$-calculus and we proved, in [6], that it satisfy some fundamental properties like confluence, subject reduction, and strong normalization of some relevant sub-calculi.

3 The $\lambda^{\{\}}$-calculus

In this section we define a minimal system implementing overloading with late binding. The goal is to use as few operators as possible. Therefore we renounce to having “extensible” overloaded functions (i.e. functions to which one can add new branches by the & operator). Terms are are built from variables by an operator of abstraction and one of application. Types are built from a set of basic types by the constructor for overloaded types. The key idea is to consider standard functions (λ-abstractions) as overloaded functions with just one branch. We use a special reduction rule (ζ) in order to avoid the use of call-by-value when the function at issue is formed by a unique branch (i.e. when we perform β-reductions).

$$\begin{array}{lcll} T & ::= & A \mid \{T_1 \to T_1, \cdots, T_n \to T_n\} & n \geq 1 \\ M & ::= & x \mid \lambda x(M_1{:}T_1 \Rightarrow T_1, \cdots, M_n{:}T_n \Rightarrow T_n) \mid MM & n \geq 1 \end{array}$$

Since there is only one type constructor, there is also only one subtyping rule:

$$\text{(subtype)} \qquad \frac{\forall i \in I, \exists j \in J \quad U_i \leq S_j \text{ and } T_j \leq V_i}{\{S_j \to T_j\}_{j\in J} \leq \{U_i \to V_i\}_{i\in I}}$$

3.0.1 Types

As usual we have the rules of type good formation. Every atomic type belongs to **Types**. If for all $i, j \in I$, (a) ($S_i, T_i \in$ **Types**) (b) ($S_i \leq S_j \Rightarrow T_i \leq T_j$) (c) (for every S maximal in $LB(S_i, S_j)$ there exists a unique $h \in I$ such that $S_h = S$), then $\{S_i \to T_i\}_{i\in I} \in$ **Types**

Note that variables are no longer indexed by their type. This because in the term $\lambda x(M_1{:}S_1 \Rightarrow T_1, \cdots, M_n{:}S_n \Rightarrow T_n)$ the variable x should be indexed in each branch by a different type (i.e. the corresponding S_i). Thus we prefer to avoid indexing and introduce in the typing rules typing contexts (denoted by Γ). We suppose to work modulo α-conversion so that the order in Γ is not meaningful:

3.0.2 Type system

$$[\text{TAUT}] \qquad \Gamma \vdash_{sub} x{:}\Gamma(x)$$

$$[\text{INTRO}^*] \qquad \frac{\forall i \in I \quad \Gamma, (x{:}S_i) \vdash_{sub} M_i{:}T_i}{\Gamma \vdash_{sub} \lambda x(M_i : S_i \Rightarrow T_i)_{i\in I}{:}\{S_i \to T_i\}_{i\in I}}$$

[ELIM*] $$\frac{\Gamma \vdash_{sub} M\colon \{S_i \to T_i\}_{i\in I} \quad \Gamma \vdash_{sub} N\colon S_j}{\Gamma \vdash_{sub} MN\colon T_j}$$

[SUBSUMPTION] $$\frac{\Gamma \vdash_{sub} M\colon S \quad S \leq T}{\Gamma \vdash_{sub} M\colon T}$$

3.0.3 Typing algorithm

The typing algorithm for $\vdash_{sub}$ is given by the following rules

[TAUT] $$\Gamma \vdash x\colon \Gamma(x)$$

[INTRO] $$\frac{\forall i \in I \quad \Gamma,(x\colon S_i) \vdash M_i\colon U_i \leq T_i}{\Gamma \vdash \lambda x(M_i : S_i \Rightarrow T_i)_{i\in I}\colon \{S_i \to T_i\}_{i\in I}}$$

[ELIM] $$\frac{\Gamma \vdash M\colon \{S_i \to T_i\}_{i\in I} \quad \Gamma \vdash N\colon S}{\Gamma \vdash MN\colon T_j} \qquad S_j = \min_{i\in I}\{S_i | S \leq S_i\}$$

This algorithm is sound and complete with respect to $\vdash_{sub}$, in the sense that a term is typable by subsumption if and only if it is typable by the algorithm (the algorithm returns the minimum of types inferred by the system with subsumption).

3.0.4 Reduction

The optimal reduction rule for this system would be the following one

(β^{opt}) $\quad \lambda x(M_i : S_i \Rightarrow U_i)_{i\in I} N \ \rhd_\Gamma \ M_j[x\colon= N]$ $\quad$ S_j least type compatible with the run-time type of N

But clearly this rule is surely intractable if not even undecidable. In general it will be necessary at least to compute a good deal of the program this redex appears in, in order to discover the right S_j. In λ& we adopted the simplest solution choosing to allow the reduction only after that this computation had taken place, that is when the argument had reached its run-time type. This solution was inspired by what happens in object-oriented programming in which a message is bound to a method only when the receiver of the message is a fully evaluated object. Though, some reasonable improvements are possible.

We think that a good trade-off between the tractability of the reduction and its generality is to allow reductions also when we are sure that however the computation evolves the selected branch is always the same. This exactly is what the reduction (ζ) below does. More formally:

The selection of the branch of an overloaded function needs the (algorithmic) type of its argument. Since this argument may be an open term (and variables are no longer indexed by their type) reduction will depend on a typing context Γ. Thus we define a family of reductions, subscripted by typing contexts $\rhd_\Gamma \subseteq$ **Terms** $\times$ **Terms**, such that if $M \rhd_\Gamma N$ then $FV(M) \subseteq dom(\Gamma)$.

We have the following notion of reduction:

(ζ) Let $S_j = \min_{i\in I}\{S_i | U \leq S_i\}$ and $\Gamma \vdash N\colon U$. If N is a closed normal form or $\{S_i \,|\, i \in I, S_i \leq S_j\} = \{S_j\}$ then
$$\lambda x(M_i : S_i \Rightarrow U_i)_{i\in I} N \ \rhd_\Gamma \ M_j[x\colon= N]$$

Then there are the rules for the context closure: the change of the context must be taken into account when reducing inside λ-abstractions:

$$\frac{M \rhd_\Gamma M'}{MN \rhd_\Gamma M'N} \qquad\qquad \frac{N \rhd_\Gamma N'}{MN \rhd_\Gamma MN'}$$

$$\frac{M_i \ \rhd_{\Gamma,(x:S_i)} M_i'}{\lambda x(\cdots M_i{:}S_i \Rightarrow T_i \cdots) \rhd_\Gamma \lambda x(\cdots M_i'{:}S_i \Rightarrow T_i \cdots)}$$

Note that if $M \rhd_\Gamma N$ then $FV(N) \subseteq FV(M)$ thus the transitivity closure of $\rhd_\Gamma$ is well-defined.

It is important to remark that the β-reduction is the special case of the ζ-reduction of a function containing a unique branch.

4 Properties

We cannot relate $\lambda^{\{\}}$ directly to λ& since their respective reduction rules are too different. Therefore, we define $\lambda\&^+$, a conservative extension of λ& obtained by using a reduction rule analogous to (ζ).

4.0.5 The $\lambda\&^+$-calculus

The $\lambda\&^+$-calculus is obtained by replacing in the λ&calculus the rule $(\beta_\&)$ by the following one

$(\beta_\&^+)$ Let $S_j = \min_{i=1..n}\{S_i | S \leq S_i\}$. If $N{:}S$ is closed and in normal form or $\{S_i \,|\, i \leq n, S_i \leq S_j\} = \{S_j\}$ then

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

It is clear that the theory of $\lambda\&^+$ is an extension of the theory of λ& since $M \rhd_{\beta_\&} N$ implies $M \rhd_{\beta_\&^+} N$.

The proof of subject reduction for $\lambda\&^+$ is strictly the same as the one for λ& given in [6] (the only exception is that in the case $M \equiv (N_1 \& N_2) \bullet M_2$ of Theorem 5.2 the argument M_2 may not be in normal form) while the one of confluence requires some slight modifications (two cases must be added in the lemma and in the corresponding theorem given in [6]. See [3].)

4.0.6 Subject reduction

Subject reduction can be directly proved on $\lambda^{\{\}}$. However we prefer to prove it by translating $\lambda^{\{\}}$ into $\lambda\&^+$ in order to better understand how the two systems are related.

More precisely, in order to prove that $\lambda^{\{\}}$ satisfies the subject reduction property, we define a translation $[\![\ ]\!]_\Gamma$ from $\lambda^{\{\}}$ to $\lambda\&^+$ with the following properties:

1. $\Gamma \vdash M{:}T \quad \Leftrightarrow \quad [\![M]\!]_\Gamma{:}T$
2. $M \rhd_\Gamma N \quad \Rightarrow \quad [\![M]\!]_\Gamma \rhd^* [\![N]\!]_\Gamma$

It is then clear that the subject reduction of $\lambda^{\{\}}$ follows from the subject reduction of $\lambda\&^+$.

Define an arbitrary *total* order $\preceq$ on **Types** with the following property: if $S \leq T$ then $S \preceq T$.[1] Given an overloaded type $\{S_i \to T_i\}_{i=i..n}$ we denote by σ the permutation that orders the S_i's according to $\preceq$.

[1]The relation $\leq$ is a preorder but not an order (see [6]). Therefore, strictly speaking, $\preceq$ is defined on **Types**/$\sim$ where $S \sim T$ iff $S \leq T \leq S$. This however does not affect the substance of what follows.

Thus $S_i \leq S_j$ implies $\sigma(i) \leq \sigma(j)$. This permutation is used to translate $\lambda^{\{\}}$ into $\lambda\&^{+}$.

$$\begin{aligned}
\llbracket x \rrbracket_\Gamma &= x^{\Gamma(x)} \\
\llbracket MN \rrbracket_\Gamma &= \llbracket M \rrbracket_\Gamma \bullet \llbracket N \rrbracket_\Gamma \\
\llbracket \lambda x(M_i : S_i \Rightarrow T_i)_{i=1..n} \rrbracket_\Gamma &= (\cdots((\varepsilon \\
&\quad \&^{\{S_{\sigma(1)} \to T_{\sigma(1)}\}} \lambda x^{S_{\sigma(1)}}.\llbracket M_{\sigma(1)} \rrbracket_{\Gamma,(x:S_{\sigma(1)})}) \\
&\quad \&^{\{S_{\sigma(i)} \to T_{\sigma(i)}\}_{i=1,2}} \lambda x^{S_{\sigma(2)}}.\llbracket M_{\sigma(2)} \rrbracket_{\Gamma,(x:S_{\sigma(2)})}) \\
&\quad \vdots \\
&\quad \&^{\{S_{\sigma(i)} \to T_{\sigma(i)}\}_{i=1..n}} \lambda x^{S_{\sigma(n)}}.\llbracket M_{\sigma(n)} \rrbracket_{\Gamma,(x:S_{\sigma(n)})})
\end{aligned}$$

The proof that this translation satisfies the two properties above is simple, and can be found in [3].

4.0.7 Church-Rosser

The system also satisfy the Church-Rosser property.

Theorem 4.1 *For all Γ the relation $\rhd_\Gamma$ is Church-Rosser*

Proof. This theorem can be proved by using the technique due to W. Tait and P. Martin-Löf (see [1]), according to which it suffices to define a parallel reduction which satisfies the diamond property and whose transitive closure is $\rhd^*_\Gamma$. See [3] for details. $\square$

5 Practical motivations

As we said at the beginning, we use $\lambda\&$ to study the foundation of object-oriented languages. More precisely, messages are seen as overloaded functions and sending a message to an object corresponds to apply the message to the object. Therefore, the selection of a branch for an overloaded function corresponds to the selection of a method for a message. The ζ-reduction (or equivalently the $\beta^+_\&$-reduction), allows to precociously select a branch for a given application (i.e. a method for a given message passing). In particular the system does not have to fully evaluate the argument of an overloaded function (i.e. the receiver of a message) to perform the selection. It just suffices that the computation of the argument gets to a stage such that any further computation would not change the selection.

At run-time this kind of rule would hardly be used. Indeed, one does not want to early select or partially evaluate methods but rather to apply them to concrete objects. On the contrary at compile time a rule that permits the early resolution of the dispatching is essential for the production of efficient object code. A preliminary study on early implementations of the overloaded-functions based language Dylan, showed that on non optimized code about 30% of the time of computation is spent to perform the method dispatching (source: Dylan group, Eastern Research and Technology, personal communication). It is then clear that a mechanism which makes possible to solve the dispatching (branch selection) at compile time is one of the main tasks in designing a compiler producing code comparable for speed to the code produced by, say, a C++ compiler. The rule ζ (and $\beta^+_\&$) goes in that direction. Such a rule allows a significative amount of resolution at compile time of method dispatching and, thus, the production of efficient object code.

Also the fact that in $\lambda^{\{\}}$ there is a unique notation for application is very important. Indeed, the languages like CLOS, Cecil or Dylan that use lately binded overloaded functions ("generic functions" in CLOS' jargon), never distinguish the application of a regular function from an overloaded one.

The absence of such a distinction is even more important in the single-dispatching object-oriented languages like Simula, Smalltalk or C++. In [4] we have shown that the use of an overloaded function for method definitions constitutes a possible solution to the longstanding problem of binary methods (see [2]). In particular when defining a new class it is possible to covariantly override the definition of a binary method in a type safe way, by using an overloaded function. Now, if the binary method to override is a regular function, then this solution works if and only if overloaded and regular functions' applications have the same syntax . . . as in $\lambda^{\{\}}$. Otherwise, the solution would require to the programmer a look ahead, since he should define also the first binary method as an overloaded function (of just one branch), look ahead that is contrary to the spirit of object-oriented programming.

Of course, all these advantages could have been obtained directly by considering $\lambda\&^{+}$ via the translation we defined in Section 4. For example, to obtain a unique application (or to faithfully translate generic functions) it suffices to translate every ordinary function into a single branched overloaded function, so that every application becomes the application of an overloaded function (this is exactly what we do in [7] to have a unique application). However, this translation seems to suggest that λ& is too powerful and that the same advantages can be obtained by using a much simpler calculus, $\lambda^{\{\}}$ indeed. In this sense, $\lambda^{\{\}}$ constitutes a study toward the definition of a minimal calculus to interpret object-oriented programming. This research is not without interest since a simpler calculus allows the definition of a simpler interpretation of object-oriented languages, which should lead to a better understanding of object-oriented features For example, in [5] most of the technical difficulties were caused by the use of type annotation for &'s and by the ordering of the branches. These problems do not subsist in $\lambda^{\{\}}$ where there are no special type annotations for overloaded functions, and abstractions can be considered equivalent modulo the branch ordering. On the other hand it is true that $\lambda^{\{\}}$ is too minimal. For example, it does not allow to add a new branch or redefine an old one, as λ& does. Thus the next step will be to study how to allow such features. It will be possible, then, to define a metalanguage derived from λ&, as we did in [5], by adding type declarations, coercions, *super*, and so on, and to interpret a generic functions based object-oriented language into it. The hope is that the resulting interpretation will be much simpler then the one defined in [5], and that it will all allow to grasp a better understanding of object-oriented languages. Another possibility for future research is the extension of $\lambda^{\{\}}$ to second order type systems, on the lines of the work done in [3, 7] for λ&.

References

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

[2] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3), 1996.

[3] G. Castagna. *Overloading, subtyping and late binding: functional foundation of object-oriented programming*. PhD thesis, Université Paris 7, January 1994. Appeared as LIENS technical report.

[4] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[5] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, November 1995. Extended abstract in the Proceedings of the *13th Conference on the Foundations of Software Technology and Theoretical Computer Science*; Lecture Notes in Computer Science number 761, December 1993.

[6] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995. A preliminary version was presented at the *1992 ACM Conference on LISP and Functional Programming*, San Francisco, June 1992.

[7] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston. To appear.