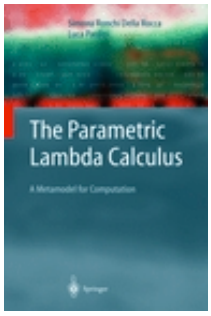
ERRATA CORRIGE | ORDER INFORMATION | TABLE of CONTENTS



THE PARAMETRIC LAMBDA CALCULUS
A Metamodel for Computation

by **Simona Ronchi Della Rocca** and **Luca Paolini**

Series : *Texts in Theoretical Computer Science*. An EATCS Series
Springer-Verlag Berlin,Hidelberg,New York,Hong Kong,London,Milan,Paris,Tokyo
2004, XIII, 252 p., Hardcover ISBN: 3-540-20032-0

AN ABRIDGED PREFACE

The lambda-calculus was invented by Church in the 1930s with the purpose of supplying a logical foundation for logic and mathematics.
Its use by Kleene as a coding for computable functions makes it the first programming language, in an abstract sense, exactly as the Turing machine can be considered the first computer machine. The lambda-calculus has quite a simple syntax (with just three formation rules for terms) and a simple operational semantics (with just one operation, substitution), and so it is a very basic setting for studying computation properties.

The first contact between lambda-calculus and real programming languages was in the years 1956-1960, when McCarthy developed the LISP programming language, inspired from lambda-calculus, which is the first ``functional'' programming language, i.e., where functions are first-class citizens. But the use of lambda-calculus as an abstract paradigm for programming languages started later as the work of three important scientists: Strachey, Landin and Bøhm.
Strachey used the lambda-notation as a descriptive tool to represent functional features in programming when he posed the basis for a formal semantics of programming languages.
Landin formalized the idea that the semantics of a programming language can be given by translating it into a simpler language that is easier to understand. He identified such a target language in lambda-calculus and experimented with this idea by giving a complete translation of ALGOL60 into lambda-calculus.
Moreover, he declared that a programming language is nothing more than lambda-calculus plus some ``syntactic sugar''.

Bøhm was the first to use lambda-calculus as an effective programming language, defining, with Gross, the CUCH language, which is a mixture of lambda-calculus and the Curry combinators language, and showing how to represent in it the most common data structures.

But, until the end of the 1960s, lambda-calculus suffered from the lack of a formal semantics. In fact, while it was possible to codify in it all the computable functions, the meaning of a generic lambda-term not related to this coding was unclear. The attempt to interpret lambda-terms as set-theoretic functions failed, since it would have been necessary to interpret it into a set D isomorphic to the set of functions from D to D, which is impossible since the two spaces always have different cardinality. Scott solved the problem by interpreting lambda-calculus in a lattice isomorphic to the space of its continuous functions, thus giving it a clear mathematical interpretation. So the technique of interpretation by translation, first developed by Landin, became a standard tool to study the denotational semantics of programming languages; almost all textbooks in denotational semantics follow this approach.

But there was a gap between lambda-calculus and the real functional programming languages. The majority of real functional languages have a ``call-by-value'' parameter passing policy, i.e., parameters are evaluated before being passed to a function, while the reduction rule of lambda-calculus reflects a ``call-by-name'' policy, i.e., a policy where parameters are passed without being evaluated. In the folklore there was the idea that a call-by-value behaviour could be mimicked in lambda-calculus just by defining a suitable reduction strategy. Plotkin proved that this intuition was wrong and that lambda-calculus is intrinsically call-by-name. So, in order to describe the call-by-value evaluation, he proposed a different calculus, which has the same syntax as lambda-calculus, but a different reduction rule.

The aim of this book is to introduce both the call-by-name and the call-by-value lambda-calculi and to study their syntactical and semantical properties, on which their status of paradigmatic programming languages is based. In order to study them in a uniform way we present a new calculus, the lambda-Delta-calculus, whose reduction rule is parametric with respect to a subset Delta of terms (called the set of input values) that enjoy some suitable conditions.Different choices of Delta allow us to define different languages, in particular the two lambda-calculus variants we are speaking about. The most interesting feature of lambda-Delta-calculus is that it is possible to prove important properties (like confluence) for a large class of languages in just one step. We think that lambda-Delta-calculus can be seen as the foundation of functional programming.

ORGANIZATION of the BOOK

The book is divided into four parts, each one composed of different chapters.
The first part is devoted to the study of the syntax of lambda-Delta-calculus. Some syntactical properties, like confluence and standardization, can be studied for the whole Delta class. Other

properties, like solvability and separability, cannot be treated in a uniform way, and they are therefore introduced separately for different instances of Delta.

In the second part the operational semantics of lambda-Delta-calculus is studied.
The notion of operational semantics can be given in a parametric way, by supplying not only a set of input values but also a set of output
values Theta, enjoying some very natural properties. A universal reduction machine is defined, parametric into both Delta and
Theta, enjoying a sort of correctness property in the sense that, if a term can be reduced to an output value, then the machine stops,
returning a term operationally equivalent to it. Then four particular reduction machines are presented,
three for the call-by-name lambda-calculus and one for the call-by-value lambda-calculus, thereby presenting four operational behaviours that are particularly interesting for modeling programming languages. Moreover, the notion of extensionality is revised, giving a new parametric definition that depends on the operational semantics we want to consider.

The third part is devoted to denotational semantics.
The general notion of a model of lambda-Delta-calculus is defined, and then the more restrictive and useful notion of a filter model, based on intersection types, is given. Then four particular filter models are presented, each one correct with respect to one of the operational semantics studied in the previous part. For two of them completeness is also proved. The other two models are incomplete: we prove that there are no filter models enjoying the completeness property with respect to given operational semantics,
and we build two complete models by using a technique based on intersection types. Moreover, the relation between the filter models and Scott's models is given.

The fourth part deals with the computational power of lambda-Delta-calculus. It is well known that lambda-calculus is Turing complete, in both its call-by-name and call-by-value variants, i.e. it has the power of the computable functions. Here we prove something more,
namely that each one of the reduction machines we present in the third part of this book can be used for computing all the computable functions.

USE of the BOOK

This book is dedicated to researchers, and it can be used as a textbook for master's or PhD courses in Foundations of Computer Science. Moreover, we wish to advise the reader that its aim is not to cover all possible topics concerning lambda-calculus, but just those syntactical and semantics properties which can be used as tools for the foundation of programming languages.