Chapter 10
Tracking Redexes in the Lambda Calculus

Jean-Jacques Lévy

Abstract Residuals of redexes keep track of redexes along reductions in the lambda calculus. Families of redexes keep track of redexes created along these reductions. In this chapter, we review these notions and their relation to a labeled λ-calculus introduced here in a systematic way. These properties may be extended to combinatory logic, term rewriting systems, process calculi and proofnets of linear logic.

10.1 Introduction

The λ-calculus is a basic setting in the foundations of mathematical logic. It gained much importance in proof theory within types and the Curry-Howard correspondence. In the theory of programming languages, the λ-calculus is a key model for functional languages. It has also applications in their type theory and even in imperative languages through the use of continuations. The beauty of the λ-calculus is that all calculations, named reductions, are generated by a single rule, the β-conversion rule, which defines the application of an argument to a function.

We suppose that the reader is familiar with the usual definitions and notations of the λ-calculus. If not the reader is refereed to next section or to Barendregt's book [6]. We will mainly consider β-conversion and β-redexes, although many of the results exposed here also hold in other calculi. A reducible expression (redex) is any term of the form $(\lambda x.M)N$ where argument N is applied to function $\lambda x.M$. Its contraction produces the term $M\{x := N\}$ in which all occurrences of the free variable x in M are replaced by N. We ignore all problems due to the renaming of bound variables (α-conversion) and assume that the binding of bound variables are respected as in standard mathematics.

J.-J. Lévy (✉)
Irif & Inria, Paris, France
e-mail: jean-jacques.levy@inria.fr

B. Meyer (ed.), *The French School of Programming*,
https://doi.org/10.1007/978-3-031-34518-0_10

Although β-conversion is a simple rule, many results of the λ-calculus are not easy to prove. This is due to the ability of computing inside the body of functions at the same time as their arguments are passed to functions. Moreover reductions may be infinite and inductive proofs have to be carefully performed. However in the typed versions of the λ-calculus, there are no infinite reductions and proofs get much simpler. In this chapter, we show how the untyped λ-calculus can be viewed as an infinite limit of labeled calculi, and how proofs in the untyped case can be conducted in these pseudo-typed calculi.

The results in this chapter are already present in many old articles [23, 25], but our presentation is focused here on a systematic way of defining a labeled calculus that we will show related to the notion of family of redexes. This family relation generalizes the notion of residuals which appeared in Church original monograph [11]. We start from the Hyland-Wadsworth calculus in Sect. 10.3 and derive the labeled calculus. In Sect. 10.4, we show its correspondence with permutation equivalence. The history of redexes is related to labeled redexes in Sect. 10.5. We explain our the results are applicable to combinatory logic and orthogonal term rewriting systems in Sect. 10.6. In the conclusion, we cite several related topics such as optimal reductions, reversible calculi, causality and event structures. The acknowledgements section contains a brief history of the beginning of theoretical computer science at Inria-Rocquencourt.

10.2 Preliminaries

The reader familiar with the lambda-calculus may skip this section.

The λ-calculus is a calculus of functions. For instance, take the successor function on integers $\mathsf{succ}(n) = n + 1$. In many programming languages (Lisp, Scheme, Python, Javascript, Java 8, Ocaml, Haskell), we may define succ by $\mathsf{succ} = \lambda n.\, n + 1$ where $\lambda n.\, n + 1$ is the function which for any n returns $n + 1$. This lambda notation is a way of manipulating anonymous functions. For instance, with lists, one may write the expression $\mathsf{map}(\lambda n.\, n + 1)[1; 2; 3]$, which returns the list $[2; 3; 4]$. When a function has several arguments as in $\mathsf{add}(x, y) = x + y$, we can define it by $\mathsf{add} = \lambda x.\lambda y.\, x + y$ where add is now a function from x returning a function from y which returns $x + y$. This transformation of a binary function into unary functions is called the curryfication in the λ-calculus. It is no more than the isomorphism between $\mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$ and $\mathbb{N} \mapsto \mathbb{N} \mapsto \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers. Thus in this chapter, we only consider unary functions.

To apply an argument N to function $\lambda x.M$, we write $(\lambda x.M)(N)$ in standard mathematics. But in order to minimize the use of parentheses, we write $(\lambda x.M)N$ in the λ-calculus. Thus succ could be defined by $\mathsf{succ} = (\lambda x.\lambda y.\, y + x)1$. Notice that the name of a variable bound by a λ is not important. Therefore $\lambda n.\, n + 1$ is the same expression as $\lambda y.\, y + 1$. Bound variables of the lambda-calculus behave like bound variables of summations or integrals in calculi of mathematics.