# Confluence of Terminating Rewriting Computations

Jean-Pierre Jouannaud

# Chapter 11
# Confluence of Terminating Rewriting Computations


**Jean-Pierre Jouannaud**

**Abstract** Rewriting is an intentional model of computation which is inherently non-deterministic. Defining functions by rewriting requires to prove that the result of a given computation is unique for every input, a property called confluence. This chapter describes confluence criteria for first-order as well as higher-order, terminating rewriting computations described by means of a set $R$ of rewrite rules and a set $E$ of equations such as associativity and commutativity. These criteria aim at reducing the confluence property of $(R, E)$ to the joinability of its critical pairs, obtained by overlapping left-hand sides of rules at subterms, which is the only source of non-trivial non-determinism. Joinability is the property that the normal forms of a pair of expressions resulting from an overlap are equal modulo $E$. Two properties are therefore needed: termination of rewriting modulo $E$, and existence of finitely many most general unifiers modulo $E$ for computing the overlaps. We show that the first-order and higher-order case yield the same results, obtained by following a uniform approach. In the higher-order case, an abstract axiomatization of types is provided so as to capture by a single result typed computations based on different type systems.

## 11.1 Introduction

Functional programs are meant to define functions. There are three main approaches, which are often combined. For the first, only definitions by fixpoints are possible. This approach ensures that only functions can be defined. Early functional languages such as LISP belong to that category. For the second, pattern matching definitions are also permitted, provided the rules are orthogonal, that is, left-linear and non-overlapping. In this second approach, orthogonality ensures that the result of a computation is unique, hence only functions can be defined. Languages of the ML family such as standard ML and CAML belong to that category. For the

J.-P. Jouannaud (✉)
Laboratoire de Méthodes Formelles, Université Paris-Saclay, Gif-sur-Yvette, France



B. Meyer (ed.), *The French School of Programming*,
https://doi.org/10.1007/978-3-031-34518-0_11

third, arbitrary rewrite rules are allowed, in which case the result of a computation may no more be unique. Uniqueness needs then to be proved, resulting from the so-called confluence property stating that diverging computations can always be extended until they eventually join. Languages of the OBJ and MAUDE families such as CAFÉ OBJ and ELAN belong to that category. Most modern functional languages, for example HASKELL, as well as recent versions of dependently typed languages such as AGDA, COQ, DEDUKTI, $\lambda\Pi$ and LEAN, combine these different approaches.

Another issue is typing. LISP was untyped. All modern functional languages are (strongly) typed, ensuring that computations cannot abort. They differ in their language of expressions, and in their languages of types. In OBJ and MAUDE, expressions are first-order terms while types are constants belonging to a finite set equipped with a transitive relation, called *subsort*, semantically interpreted as inclusion. Only pattern matching definitions are allowed. In languages of the ML and HOL families such as HOL-light and Isabelle, types are prenex, closed logical expressions built from propositional constants, variables, implication and universal quantification. In the object-oriented language CAML of this family, an order on types, contravariant with respect to implication, allows one to express inheritance. The language of types becomes richer with typed functional languages such as AGDA, COQ, DEDUKTI, LEAN, MATITA and PVS. In most of these languages, a type is a logical statement, a program of a given type is a proof of that statement, while functional computation is cut elimination, this is called the Curry-Howard principle. This paradigm has driven a large part of the research on functional programming languages and proof assistants during the last two decades, following De Bruijn Automath, Martin-Löf's type theory, and Girard's System $F$. An important milestone in this trend is COQ, the first language whose types are arbitrary higher-order logical formulae.

There are of course other important aspects in functional programming languages, such as structuring, that have been ignored because they are not important to us here.

Because computations are based on rewriting, confluence plays a key rôle in all these languages, not only to ensure functionality of computations, but also to ensure, together with type preservation and strong normalization, that the type theory is consistent from a logical point of view. Our goal in this chapter is to look at these languages from a rewriting perspective, and describe the mathematical tools that allow us to prove that programs in those languages are functional.

Historically, rewriting was not conceived as a language for defining functions, but as the main computational tool to carry out proofs. This tool emerged from various mathematical studies: in algebraic geometry where computing with axioms was needed, and led eventually to various algorithms for computing bases of algebraic varieties, such as Gröbner bases or standard bases; in logic, from Herbrand's work and Church's work for defining functions by quite different means, algebraic computations for the first, and lambda calculus for the second; in automated deduction, from seminal works by Wos, by Lankford, by Knuth and by Huet, whose goal was an automatic treatment of equality in logic.