Implicit Computational Complexity
ESSLLI 2010 CPH

Ugo Dal Lago
Università di Bologna
dallago@cs.unibo.it

This document includes some basic information about the course "Implicit Computational Complexity" which the author will give at ESSLLI 2010. It includes a description of the topics covered by the course, a delineation of the prerequisites necessary to follow it and a list of bibliographic references. This document should *not* be intended as a set of lecture notes, which are available from the course webpage[1], but are still under active development at the time of writing.

1 Scope and Structure

This course is an introduction to Implicit Computational Complexity (ICC for short), which aims at characterizing complexity classes by logical systems and paradigmatic programming languages. Some characterizations of major complexity classes based on model theory, recursion theory and proof theory have been introduced in the last thirty years. On the other hand, similar results have been obtained using tools from programming language theory, such as type systems or the interpretation method.

The field is sufficiently broad and scattered to prevent us from being exhaustive in this course. We have chosen to present some of the most interesting ICC systems from a programming language perspective. First of all, some paradigmatic programming languages with high expressive power will be presented, then showing how they can be restricted, thus obtaining languages in which only relatively small classes of functions (i.e., complexity classes) can be written. Only at that point, the relations with mathematical logic (in particular with proof theory and recursion theory) become apparent. Arguably, this way of presenting the field does not really follow the historical development, which was mainly driven by mathematical logic. However, we strongly believe that this way of learning ICC makes the task of understanding the basic ideas easier, since less prerequisites are needed before each ICC system can be properly defined and studied.

2 Contents

The course is divided into four parts:

A brief introduction to computability and complexity. Understanding ICC is not possible without knowing the basics of computability and complexity. In this part of the course, we review the notions of an algorithm, of a computable function and of the many complexity measures one can attribute to algorithms and problems. A good reference, in which much more can be found than what we will able to say, is Papadimitriou's textbook [5].

Functional programs and complexity classes. One of the most interesting programming paradigms is the functional one. There, the basic blocks programs are built with are pure functions, namely functions which do not have an internal state nor produces side effects. In this part of the course, a paradigmatic language of functional programs, namely the language of constructor term rewriting systems, is defined and shown to be Turing Complete. Afterwards, restrictions on the same language are presented and proved to exactly capture complexity classes, such as the class of polynomial time computable functions. These restrictions are based on either path orders or the interpretation method. In this part of the course, we will follow Baader and Nipkov's textbook on term rewriting [1], then switching to research papers such as [4].

Higher-order functions. A feature many functional languages offer to the programmer is the possibility of writing higher-order functions, namely functions which can take other functions as arguments and return functions as results. Constructor term rewrite systems cannot express higher-order functions in a direct manner. As a consequence, a more expressive model is needed. λ-calculus, introduced by Church in the 1930's, is the ideal candidate. We here show that the

[1] `http://www.cs.unibo.it/~dallago/ICC2010`

untyped λ-calculus is itself Turing Complete. Then, we show how to characterize complexity classes by restrictions on the lambda calculus, starting with the elementary functions and reaching the polynomial time computable functions. A good reference for λ-calculus is [2], while the way we will restrict it to get characterizations of complexity classes is inspired from [6].

Beyond the functional paradigm. Functional programming languages are definitely not the only programming languages around. Indeed, ICC has produces characterizations of different complexity classes in other paradigms, like the imperative one, to which we focus our attention in the last part of the course. In particular, we will present a characterization of the complexity classes in the Grzegorczyk hierarchy by the μ-measure on loop programs, following [3].

3 Prerequisites

This is an introductory course on the relations between logic and computation. As a consequence, the prerequisites are kept to a minimum and much effort has been put into keeping the course self-contained. Anybody with some familiarity with the language and methods of mathematics should not find problems in following it. Knowing the basics of discrete mathematics, combinatorics, mathematical logic and the theory of algorithms, on the other hand, could be of some help.

References

[1] Baader, F., and Nipkow, T. *Term rewriting and all that.* Cambridge University Press, 1998.

[2] Hindley, J. R., and Seldin, J. P. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, 2008.

[3] Kristiansen, L., and Niggl, K.-H. On the computational complexity of imperative programming languages. *Theoretical Computer Science 318*, 1-2 (2004), 139 – 161.

[4] Marion, J.-Y. Analysing the implicit complexity of programs. *Information and Computation 183*, 1 (2003), 2–18.

[5] Papadimitriou, C. H. *Computational Complexity.* Addison Wesley, 1993.

[6] Terui, K. Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic 46*, 3-4 (2007), 253–280.