A presentation of the Curry-Howard Correspondance.

Chantal Berline
berline@logique.jussieu.fr

1st April 1997 *

Abstract

These notes are extracted from the first version of the paper "From Computation to Foundations: the λ-calculus and its webbed models". They nearly disappeared in the revised version of that paper, and we make them available separately.

0.1 λ-calculus as a foundation for Programming Theory.

λ-calculus came back to the front of the scene in the sixties with the development of Computer Science, under the impulse of Landin [21] and Backus (cf. [1]) and generated the family of *functional languages* (Lisp [McCarthy ~1960], Haskell, Miranda, ML, Caml, ...). In functional languages functions and functionals may be passed as arguments to a program as easily as concrete datas, which is not the case with imperative languages (Fortran, Pascal, C) (the other conceptual differences between imperative programming (founded by Von Neumann) and functional programming are clearly explained e.g. in the first pages of [1]). See barendregt's survey [2].

The other main conceptual contribution of λ-calculus to Programming Theory comes from the fact that its terms can be viewed as proofs as well as programs : λ-calculus allows the mathematical expression of the *Curry Howard "Isomorphism"*, which is a very deep correspondence between *proof theory*, i.e. the mathematical analysis of mathematical reasoning, and concrete programming theory. The *programming with proofs paradigm* is a topic in active progress, and the present developments of the Curry Howard Correspondence generate beautiful and deep exchanges of intuitions and problems between Computer Science, Proof Theory and Mathematics. Because of its significance, we devote the next subsection to this correspondence.

*but for a very slight revision (January 15th, 1999) including the addition of reference [2].

0.2 The Curry-Howard Correspondence.

0.2.1 Typed λ-calculi.

What is a typed-calculus ? Well, just the fragment of λ-calculus which encodes the set of proofs in some given logical system ; in this logical viewpoint *type* is just a synonym for "formula", and " A is a type for the term t " if t encodes a proof of A in the given logic.

Typed λ-calculi are "enriched restrictions" of λ-calculus. "*Enriched*" because types are systematically assigned to λ-terms. "*Restrictions*" because, given a logic, the associated typed λ-calculus contains only those λ-terms which encode proofs in this logic. In such systems one may thus represent only a subclass of the computable functions, and also not all algorithms for computing them. The first point is not a drawback if the logic is powerful enough : then all the functions one will ever wish to compute are indeed representable (Girard [7],[8] ; cf. [15, p.161] or [6]), and the second limitation, which is even present in the full λ-calculus, can often be locally overcome [25].

The *expressive power* of a typed λ-calculus is thus limited by its logic, but the positive counterpart is that the computation can be controlled by the logic. This shows up globally : the restricted set of terms may consist only of normalizable terms (that is terms whose computation ends), and locally : the type of a term can become a very precise specification of what the term computes. Changing to a more expressive logic weakens the first kind of control and strengthens the second (they may also not vary).

Functional languages are not necessarily typed (Lisp is untyped), or types are used only for insuring some compatibility checkings. This corresponds to the use of weak but decidable typing systems. *By contrast the "full" programming with proofs paradigm is based on very expressive logics*, which enjoy the most interesting mathematical and computational properties (on the theoretical side of computer science).

From the programming viewpoint a type A is a *specification* for the program t, that is : a partial description, or comment, of what t actually does. A typed calculus can then be considered as a high-level (i.e. user-friendly) programming language (which is more or less high-level according to the expressive power of its underlying logic). In the case of (untyped) λ-calculus there are no comments at all. As such, λ-calculus is a very low level language and its terms are unreadable programs ; but this doesn't matter for the computer, which anyhow throws all comments out before execution. Finally, λ-calculus can be directly viewed and implemented as a machine language : it is indeed possible to give a simple hardware meaning to all of its very few symbols.

0.2.2 The programming with proofs paradigm.

The *Curry-Howard Correspondence* is a fundamental symetry between two worlds which at first sight may seem unrelated, namely the world of programs and that of mathematical proofs. The Curry-Howard Correspondence considers, and expresses, specifications of programs as mathematical theorems, and extracts programs (written as λ-terms) from the proofs of these theorems.

The importance of the *programming with proofs paradigm* is that, provided that the underlying logic is sufficiently powerful, say second-order predicate logic(s), it necessarily produces programs which do what you expect, and for mathematical reasons [15, p.154].

0.2.3 Programming with intuitionistic proofs.

The Curry-Howard correspondence issues from Heyting's semantics for Intuitionistic Logic [13] and, till recently, concerned only intuitionistic proofs, that is the proofs which do not use the Double Negation Law (DN, which allows to infer X from $\neg\neg X$ (i.e. $non(non(X))$) or, equivalently, the Excluded Middle Law (EM, which states that $X \vee \neg X$ is always true).

Since Heyting it is known that all proofs in intuitionistic logic(s) do have a constructive content : in particular any intuitionistic proof of a theorem which states the existence of an object of a certain kind gives in fact a way to build one ; this was Brouwer's motivation for inventing intuitionism, and Heyting's for inventing a formalisation [12].

On the other hand DN and EM were thought to be definitely alien to computation, and to programming theory.

The simplest intuitionistic logic where the Curry-Howard correspondence makes sense is Curry's simple type system [4], namely propositional logic over implication only. The most fruitful ones are the intuitionistic restrictions of Second order propositional logic (Girard's System F ([7], [8],[9]), rediscovered by Reynolds [30] in a computer science context), and Second order predicate logic (Leivant [22], Krivine's System FA_2 [15, p.141]) and extensions (Parigot [27] and Raffali [29]).

0.2.4 Programming with classical proofs.

At the beginning of the 90's it was realized by two computer scientists, Griffin [11], followed by Murthy ([23],[24]), that, surprisingly, classical logic could also be given a programming meaning. Griffin indeed noticed that the type $\neg\neg X \to X$ could be given to a generic "control operator" of Felleisen [5]. *Control operators* are very concrete imperative features (instructions) which are in fact needed for efficient implementations of functional languages (like Scheme Lisp) since they allow the treatment of *exceptions* and of *errors* (one of the simplest examples being the instruction "exit").

As a consequence the Curry Howard Correspondence can be extended to classical logic(s) and functional-imperative programming, by means of extensions of λ-calculus, like the λC-calculus of Felleisen [5] and the variant of λC used by Krivine [17], or the $\lambda\mu$-calculus of Parigot [26]). These calculi allow in turn representations of a lot of other control operators, and hence significant increases in the efficiency of functional programming. In other words they allow us to conceive of languages which combine (at the conceptual level) the advantages of both functional and imperative programming.

Like λ-calculus, the λC-calculi and the $\lambda\mu$-calculus are untyped but have typed restrictions.

Things are rapidly evolving in the λ-calculus world and the extension of the Curry-Howard Correspondence is a very active and enthralling topic. For mathematicians and logicians, this gives rise to a lot of interesting theoretical problems and is an excellent example of how Computer Science can inspire deep mathematics (and logic). A simple and amusing illustration of the present revolution is the realization that it is enlightening to view "false" as the type of "executable programs" (cf. [19]).

One of the intriguing questions arising concerns the operational meaning of a given classical mathematical theorem (that is the common operational behavior of all its formal proofs), where classical has here both meanings of "usual" and "provable within classical logic". The answer, if any, has to be discovered case by case. Very few have been explored as yet, and the first deep result in this direction is Krivine [19] where it is shown that the computational meaning of the Completeness Theorem for Classical Predicate Logic is that of an interactive disassembler, equipped with a device securing system calls ! (a kind of protection which systematically implemented in this kind of computer stuff).

A different, though not unrelated, problem is the search for the possible computational content of a given classical proof of a given classical theorem (for example Van der Waerden's Theorem on arithmetic progressions). The extraction of a program has been carried out for some specific classical proofs, by Berger, Schwichtenberg, Coquand, and others (cf. [3]).

0.3 Further readings.

This introduction can be complemented with the references below.

Intended for a general audience are :

Barendregt's survey on the impact of λ-calculus in logic and computer science [2]. Krivine's brief survey [16], which gives a clear idea of the present interest of the programming with proofs paradigm, and Backus [1], which is a 20 years old call for functional programming.

Girard's Proofs and Types [10] is a *proof theoretic* approach to typed calculi, based on a short post-graduate course.

The Curry-Howard correspondence for intuitionistic second order predicative logic is well explained in [20]. For the classical case see [18]. For the expressiveness of intuitionistic 2nd order typed-calculi, for their interest from mathematical, computational and metamathematical points of view, and for the connections with existing programming languages (at that moment) see [6]. For a clear exposition of the notion of "control operators" we refer to the first pages

of [5].
Finally, [15] gives the mathematical justifications of the (intuitionistic) programming with proofs paradigm (with no comments).

References

[1] J. Backus, Can programming be liberated from the von Neumann style ? A functional Style and its algebra of programs, Communication of the ACM (Association of Computing Machinery), vol.21, n.8, August 1978.

[2] H.P. Barendregt, The impact of the λ-calculus in logic and computer science, Bull. of Symb. Logic, Vol. 3, n°2, p.181-215.

[3] T. Coquand, Computational content of classical logic, in A.Pitts ed., Semantics and logics of Computation, Cambridge University Press, Cambridge, p.470-517, 1997.

[4] H.B. Curry, Functionality in Combinatory Logic, in Proc. Nat. Acad. Sci., USA 20, 584-590, 1934.

[5] M. Felleisen, D.P. Friedman, E. Kohlbecker, and B.F. Duba, A syntactic theory of sequential control, Theoretical Computer Science 52 (3) , 205-237, 1987.

[6] S. Fortune, D. Leivant, M. O'Donnell, The expressiveness of simple and second order type structures, J. of the ACM, p.151-185, 1983.

[7] J.Y. Girard, Une extension de l'interprétation de Gödel à l'analyse, in Proceedings 2nd Scandinavian Logic Symposium, ed. J. Fenstadt, North Holland, p.63-92, 1971.

[8] J.Y. Girard, Interprétation fonctionnelle et élimination des coupures de l'Arithmétique d'ordre supérieure, Thèse, Université Paris 7, 1972.

[9] J.Y. Girard, The system F of variable types, fifteen years later, Theoretical Computer Science, 45, p.159-192, 1986.

[10] J.Y. Girard, Y. Lafont, and P. Taylor, Proofs and Types, Cambridge tracts in Computer Science 7 , Cambridge Univ. Press, 1989..

[11] T.G. Griffin, A formulae-as-type notion of control, In Conference record of the seventeenth annual ACM Symposium on Principles of programming languages (POPL'90), p.47-58, 1990.

[12] A. Heyting, Die formalen Regeln der intuitionistischen Logic (resp. Mathematik), Sitzungsberichte der Preussischen Academie der Wissenschaften, Physikalisch-mathematishe Klasse, p.42-56 (resp. 57-71 and 158-169), 1930.

[13] A. HEYTING, Mathematische Grundlagenforschung, Intuitionismus, Beweistheorie, Ergebnisse der Mathematik und ihre Grenzgebiete, vol.3(4), Springer, Berlin, p.iv+73, 1934.

[14] J.L. KRIVINE, Lambda-calcul, Types et modèles, Masson, Paris, 1990.

[15] J.L. KRIVINE, Lambda-calculus, types and models, Ellis-Horwood, 1993. (augmented english translation of the above).

[16] J.L. KRIVINE, Fonctions, Programmes et Démonstrations, Gaz. Math. Soc. Math. Fr. 60, p.63-73, 1994.

[17] J.L. KRIVINE, Classical logic, storage operators and second-order lambda-calculus, Annals of Pure and Applied Logic 68, p.53-78, 1994.

[18] J.L. KRIVINE, About classical logic and imperative programming, Annals of mathematics and Artificial Intelligence 16, p.405-414, 1996.

[19] J.L. KRIVINE, Une preuve formelle et intuitioniste du théorème de complétude de la logique classique, to appear in the Bulletin of Symbolic Logic, 1997.

[20] J.L. KRIVINE AND M. PARIGOT, Programming with proofs, J. Inf. Process. Cybern. EIK 26 , 3, p. 149-167, 1990.

[21] P.J. LANDIN, The mechanical evaluation of expressions, Computer Journal 6, pp.308-320, 1964.

[22] D. LEIVANT, Reasoning about functional programs and complexity classes associated with type discipline, 24th Annual Symp. on Foundations of Computer Science, p.460-469, 1983.

[23] C.R. MURTHY, Extracting Constructive contents from classical proofs, Phd Thesis, Cornell University, Department of Computer Science, 1990.

[24] C.R. MURTHY, Control operators, hierarchies, and pseudo-classical type systems : A translation at work, in Proceedings of the CW/ACM SIGPLAN workshop on Continuations'92, ed. O. Danvy and Talcott, Technical report, Standford University, Department of Computer science, STAN-CS-92-142, 1992.

[25] M. PARIGOT, On the representations of data in lambda-calculus, in Proc. CSL'89 (Kaiserlautern), LNCS vol.440, pp.309-321, 1989.

[26] M. PARIGOT, $\lambda\mu$-calculus : an algorithmic interpretation of classical natural deduction, in A. Voronkov ed., Proceedings of the international Conference on Logic Programming and Automated Reasoning (St Petersburg), LNCS 624 (LNAI), p.190-201, 1992.

[27] M. PARIGOT, Recursive programming with proofs, Theoretical Computer Science 94, p.335-356, 1992.

[28] M. Parigot, Classical proofs as programs, in G. Gottlod, A. Letsch and D. Mundici eds., KGS'93, LNCS 713, p.263-276, 1993.

[29] C. Raffali, L'arithmétique fonctionnelle avec points fixes, PhD Thesis, Université de Paris 7, 1994.

[30] J. Reynolds, Toward a theory of type structures, Colloque sur la programmation, SLNCS 19, p.408-425, 1974.