MISSING 4 PAGES of TEXT

Appeared in IEEE Symposium on Logic Programming 1987

Le Fun:

Logic, equations, and Functions

Hassan Aït-Kaci
Patrick Lincoln
Roger Nasr

October, 1986

Artificial Intelligence Program

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, TX 78759

Abstract†

We introduce a new paradigm for the integration of functional and logic programming. Unlike most current research, our approach is not based on extending unification to general-purpose equation solving. Rather, we propose a computation delaying mechanism called *residuation*. This allows a clear distinction between functional *evaluation* and logical *deduction*. The former is based on the λ-calculus, and the latter on Horn clause resolution. In clear contrast with equation-solving approaches, our model supports higher-order function evaluation and efficient compilation of both functional and logic programming expressions, without being plagued by non-deterministic term-rewriting. In addition, residuation lends itself naturally to process synchronization and constrained search. Besides unification (equations), other residuations may be any ground-decidable goal, such as mutual exclusion (inequations), and comparisons (inequalities). We describe an implementation of the residuation paradigm as a prototype language called Le Fun—**L**ogic, **e**quations, and **Fun**ctions.

Keywords: Logic Programming, Functional Programming, Equational Programming, Symbolic Computation, Symbolic Deduction, Delayed Evaluation.

[0]†This document is extracted from [5], with the addition of a simpler Le Fun unification algorithm based on (serial) combinator reduction. It is to appear in the proceedings of the Symposium on Logic Programming, San Francisco, September 1987.

1 Introduction

A surge of research activity has been devoted lately to "integrating logic and functional programming." As usual, arguments ranging from matters of taste, pragmatic performance, to deep theoretical concerns have been put forth, some quietly, some forcefully. We, the authors, do not wish to contribute to the debate. Rather than telling the rest of the world how this ought to be done, or even why it ought to be done at all, we shall abide by a more peaceable mode of describing what we do, why we do it, and how. By no means, however, do we wish to appear "holier than thou!" Indeed, we think that some other proposals have definite elegance, are of practical use, or even achieve high performance. Rather, our answer came to us naturally when we tried to define precisely what *we* wanted, and realized that none of the proposals known to us would answer *all* and *only* our needs. Thus, we shall attempt to motivate our work by first laying out our *desiderata*. Length limitations prevent us from giving a full account of current prominent proposals, and a detailed survey can be found in [5].

Section 2 introduces our specific motivation. In Section 3 we illustrate some details, operational points of our idea, by means of Le Fun example programs. The implementation realizing Le Fun's *modus operandi* is detailed in Section 4. For the reader avid to know the heart of the matter, Section **??** explains Le Fun's unification algorithm, which accounts for dynamic function evaluation. Section **??** gives a modification of Le Fun unification based on combinator reduction which simplifies it drastically. Finally, a discussion putting our work in context closes the main body of this document.

2 Motivation and Background

2.1 Desideratum

So what do we wish? Well, to start with, let us define what we mean by functional programming and logic programming. A better qualifier would be "functional and relational" in the following sense.

- By functional, we understand a (1) *directional*, (2) *deterministic*, and (3) *convergent* flow of information.

- By relational, we understand a (1) *multidirectional*, (2) *non-deterministic*, and (3) *not necessarily convergent* flow of information.

That is, (1) functions expect input and return output, whereas relations do not, (2) functions do not fail or backtrack, whereas relations do, and (3) functions must terminate on all legal input, whereas relations may enumerate infinitely many alternative instances of their arguments.

Now that we have defined our terminology, it seems that functional programming is subsumed by relational programming. In a pragmatic sense, this is untrue since the specificity of functional programming allows the elimination of rather heavy computational overhead. Importantly, we shall view functional programming as computation in *evaluation* mode (no information guessing is allowed) and relational programming as computation in a *deduction* mode (information guessing is allowed.) This is an essential delineation since it explains why functional programming can easily handle higher-order objects whose guessing needed for relational programming is, if not impossible, at best computationally very difficult and expensive.

Although it is often the case that application problems can be solved entirely in evaluation or deduction mode, these do not constitute all programming applications. From our programming experience using a logic or functional programming language, we have repeatedly found ourselves in frustrating situations where parts of the problem we had at hand were of a functional nature, and others of a relational nature. Of course, we could always fit those parts into the language, but at the cost of some unnatural and often non-trivial thinking.

2.2 Overview of our Approach

We now introduce a relational and functional programming language called Le Fun where first-order terms are generalized by the inclusion of *applicative expressions* as defined by Landin [**?**] (atoms, abstractions, and applications) augmented with logical variables. The purpose is to allow *interpreted* functional expressions to participate as *bona fide* arguments in logical expressions.

A unification algorithm generalized along these lines must consider unificands for which success or failure cannot be decided in a local context (*e.g.*, function applications may not be ready for evaluation while expression components are still uninstantiated.) We propose to handle such situations by delaying unification until the operands are ready. That is, until further variable instantiations make it possible to reduce unificands containing applicative expressions. In essence, such a unification may be seen as a residual equation which will have to be verified, as opposed to solved, in order to confirm eventual success—whence the name *residuation*. If verified, a residuation is simply discarded; if failing, it triggers chronological backtracking at the latest instantiation point which allowed its evaluation.

Although primarily motivated as an experiment in integrating logic programming (Horn clause resolution) and functional programming (as in the λ-calculus style of functional evaluation), this residuation principle can also be generalized beyond just unification (*i.e.*, syntactic equality) to encompass any syntactical decisions which can be made pending further instantiation. In particular, ground-decidable predicates like arithmetic inequality, or syntactic mutual exclusion can be *implicitly* handled by residuation.

A remarkable corollary of this is that such unclean patches as Prolog's *is* evaluation predicate are no longer needed, yielding a truly more declarative operational semantics. That is, the programmer can describe her problem as a combination of function definitions and Horn clauses where the order in which conjuncts are verified for a given query is truly completely independent of the order in which they are specified. Thus, there is no longer any need nor justification for explicit control annotations [**?**, **?**, **?**].

In fact, this asynchronicity, if taken seriously, gives the implementor the opportunity to exploit implicit large-grained parallelism, on top of the already abundant parallelism which can be automatically detected in Serial Combinator reductions (through the use of graph reduction [**?**]), and Horn clause resolutions (through the use of *and*-parallelism and *or*-parallelism.) This truly asynchronous aspect of Le Fun is being investigated at M.I.T. in a dataflow context [**?**], but is expected to be efficiently realizable in more mundane concurrent computational environments. The key to exploiting this observation is described in Section **??**, where a greatly simplified Le Fun unification algorithm is proposed.

3 Le Fun Examples

Exposing our ideas is better done by illustrating key points of the residuation principle, giving very simple examples focusing attention away from details.

3.1 Unifying Reducible Expressions

SLD-resolution on which pure Prolog is based is not a complete deduction system for Horn logic because its depth-first may diverge although finite solutions exist. In addition, Prolog implementations are also incomplete because of built-in arithmetic. Of course, it is possible to manipulate numbers through a first-order axiomatization of arithmetic. However, performance of a "real-life" programming language forbids this. Thus, arithmetic is built into Prolog as a primitive system. Of course, this is done at the expense of completeness since numbers are thus not synthesized by unification. As a result, a goal literal involving arithmetic variables may not be proven by Prolog, even if those variables were to be provided by proving a subsequent goal. This is why arithmetic expressions cannot be nested in literals other than the *is* predicate, a special one whose operation will force evaluation of such expressions, and whose success depends on its having no uninstantiated variables in its second argument.

We give two simple examples on how this poses no problem to Le Fun.

3.1.1 Simple Case

Consider the set of Horn clauses:

```
q(X,Y,Z)
:- p(X,Y,Z,Z),
   pick(X,Y).

p(X,Y,X+Y,X*Y).
p(X,Y,X+Y,(X*Y)-14).

pick(3,5).
pick(2,2).
pick(4,6).
```

and the following query:

```
?- q(A,B,C).
```

From the resolvent $q(A,B,C)$, one step of resolution yields as next goal to establish $p(A,B,C,C)$. Now, trying to prove the goal using the first of the two p assertions is contingent on solving the equation $A+B = A*B$. Naturally, using Peano's axioms to solve this is out of the question. At this point, Prolog would fail, regardless of the fact that the next goal in the resolvent, $pick(A,B)$ may provide instantiations for its variables which may verify that equation. Our solution is to stay open-minded and proceed with the computation as in the case of success, remembering however that eventual success of proving this resolvent must insist that the equation be verified. As it turns out in this case, the first choice for $pick(A, B)$ does not verify it, since $3 + 5 \neq 3 * 5$. However, the next choice instantiates both A and B to 2, and thus verifies the equation, confirming that success is at hand.

To emphasize the fact that such an equation as $A + B = A * B$ is a left-over granule of computation, we call it a *residual equation* or *equational residuation—E-residuation, for short*. We also coin the verb *"to residuate"* to describe the action of leaving some computation for later. We shall soon see that there are other kinds of residuations. Those variables whose instantiation is awaited by some residuations are called *residuation variables* (RV). Thus, an E-residuation may be seen as an *equational closure*—by analogy to a lexical closure—consisting of two functional expressions and a list of RV's.

There is a special type of E-residuation which arises from equations involving an uninstantiated variable on one hand, and a not yet reducible functional expression on the other hand (*e.g.*, $X = Y + 1$). Clearly, these will never cause failure of a proof, since they are equations in solved form. Nevertheless, they may be reduced further pending instantiations of their RV's. Hence, these are called *solved residuations* or *S-residuations*. Unless explicitly specified otherwise, "E-residuation" will mean "equational residuations which are not S-residuations."

Going back to our example, if one were interested in further solutions to the original query, one could force backtracking at this point and thus, computation would go back eventually before the point of residuation. The alternative proof of the goal $p(A, B, C, C)$ would then create another residuation; namely, $A + B = (A * B) - 14$. Again, one can check that this equation will be eventually verified by $A = 4$ and $B = 6$.

One may observe that a possible realization of the residuation principle would be to accumulate all residual equations along a depth-first walk of the *and/or* proof tree until a leaf is reached; then, instantiate all E-residuations with the substitution at hand; and succeed if and only if they are all verified. Clearly, this would be far more expensive than using any relevant instantiations *as they materialize*—especially if partial evaluation is supported [?]. This is very reminiscent of the process of asynchronous backpatching used in one-pass compilers to resolve forward reference.

It is important to remark that, in order to be correct, a sufficiently small grain of asynchronous propagation must be necessarily larger than the unification operation on literals. Namely, it is obviously dangerous as well as a source of inefficiency to propagate instantiations coming from the partial unification of two Le Fun literals. However, such "pipelining" may be possible under very specific considerations. Such is an issue for further study.[1]

As a matter of experiment in our current prototype, we enforce atomicity of the unification operation, and do not support partial evaluation. That is, a verification of a residuation is triggered only between unifications of literals and when all its RV's are instantiated to ground values.

3.1.2 Trickier Case

A consequence of the above remark is that since instantiations of variables may be non-ground, *i.e.*, may contain variables, residuations mutate. To see this, consider the following example:

```
q(Z)
:- p(X,Y,Z),
   X = V-W,
   Y = V+W,
   pick(V,W).

p(A,B,A*B).

pick(9,3).
```

together with the query:

```
?- q(Ans).
```

The goal literal $p(X, Y, Ans)$ creates the S-residuation $Ans = X * Y$. This S-residuation has RV's X and Y. Next, the literal $X = V - W$ instantiates X and creates a new S-residuation. But, since X is an RV to some residuation, rather than proceeding as is, it makes better sense to substitute X into that residuation and eliminate the new S-residuation. This leaves us with the *mutated* residuation $Ans = (V - W) * Y$. This mutation process has thus altered the RV set of the first residuation from $\{X, Y\}$ to $\{V, W, Y\}$.

As computation proceeds, another S-residuation instantiates Y, another RV, and thus triggers another mutation

[1] Thus, we have initiated a collaboration with Rishiyur Nikhil, of the MIT Computer Science Laboratory, for a study of a Dataflow model of residuation [?].

of the original residuation into $Ans = (V - W) * (V + W)$, leaving it with the new RV set $\{V, W\}$.

Finally, as $pick(9, 3)$ instantiates V to 9 and W to 3, the residuation is left with an empty RV set, triggering evaluation, and releasing the residuation, and yielding final solution $Ans = 72$.

3.2 Residuating Ground-Decidable Goals

Equations are not the only computations which may be residuated. As a matter of fact, any goal whose decision is entailed by grounding its arguments is potentially residuatable. In particular, *inequations* ($\neq$) as well as *comparisons* ($<, >$) may as well residuate. These are called *I-residuations*.

Consider, for example,

```
q(X,Y,Z)
:- p(X,Y,Z),
   X < Y,
   Y < Z,
   pick(X,Y).

p(X,Y,X*Y).

pick(3,9).
```

with the query,

```
?- q(A,B,C).
```

Understanding this example is left as exercise.

3.3 Higher-Order Expressions

The last example illustrates how higher-order functional expressions and automatic currying are handled implicitly. Consider,

```
sq(X) = X*X.

twice(F,X) = F(F(X)).

valid_op(twice).

p(1).

pick(lambda(X,X)).

q(Val)
:- G = F(X),
   Val = G(1),
   valid_op(F),
   pick(X),
   p(sq(Val)).
```

with the query,

```
?- q(Ans).
```

The first goal literal $G = F(X)$ creates an S-residuation with the RV set $\{F, X\}$. Note that the "higher-order" variable F poses no problem since no attempt is made to solve. Proceeding, a new S-residuation is obtained as $Ans = F(X)(1)$. One step further, F is instantiated to the *twice* function. Thus, this mutates the previous S-residuation to $Ans = twice(X)(1)$. Next, X becomes the identity function, thus releasing the residuation and instantiating Ans to 1. Finally, the equation $sq(1) = 1$ is immediately verified, yielding success.

4 Le Fun Operational Semantics

4.1 General Principle

The general idea is that residuations can happen at different levels, and that timely and efficient invocation of such residuations can be accomplished through a careful run-time accruement of backchaining information built into a generalized resolution/unification algorithm. Hence, using such run-time information, invoking a residuation should happen automatically when enough information is available for such an invocation to be meaningful (*e.g.*, a residuated functional expression reduction should be invoked as soon as all the free variables in that expression are ground.) One undesired alternative, for obvious reasons, is having to accumulate all residuations in a central repository and check them there periodically for progress potential. The difference between these two alternatives is reminiscent of the difference between *interrupt servicing* and *polling* when a system is dealing with an external signal. The following is a description of the supported residuations and the backchaining information that is deemed necessary for their economical implementation.

At the resolvent level, and as part of a regular goal resolution, a unification can become residuated if a unificand is a function application not ready for reduction. Therefore, internal representation of function applications must remember the unifications pending on them. Also at the resolvent level, the resolution of certain ground-decidable predicates can be residuated if their operands are either function applications not ready for reduction, or uninstantiated variables. Therefore, both function applications and uninstantiated variables should have the capability of remembering the residuated ground-decidable predicates pending on them.

Reduction of function applications should residuate if free variables therein are still uninstantiated. Therefore, uninstantiated variables should have the capability of remembering the residuated functional reductions pending on them. A sufficient condition for the release of a residuation is thus that its RV set become empty. It is however not a necessary condition; *e.g.*, if partial computation is supported or, more generally, if function strictness information is available.

For example, we note that, given a function application, partial progress may be possible in reducing such expressions even if all free variables in the expression are not ground. For example, following Ershov [**?**], partial computation may allow earlier failures in some computations such as the E-residuation:

$$\mathit{append}([0], X) = \mathit{append}([1], Y)$$

However, the computational overhead needed to support such eager evaluation with the potential of backtracking is considerably more severe, since in worst cases trailing of all partial reductions must be kept. Generally, in the case where a function in not strict in one of its arguments (*i.e.*, it does not insist that particular argument denote in order for its application to denote) it is clear that a non-strict RV need not be waited for in order for the computation to proceed.

The above points lead us to the following observations.

- Computation fragments that may need to be delayed and remembered (residuated) are (1) functional applications (S-residuations), (2) ground-decidable predicate invocations (I-residuations), and (3) unification operations (E-residuations.)

- Objects that may need to remember residuated computations are: (1) functional applications, and (2) uninstantiated variables.

- The backchaining information is always recorded at unification time, or at the time certain built in predicates are invoked; this is when it is realized whether residuation will be necessary. The unification algorithm will detail the issues related to the nature and placement of that information.

- The backchaining information will be extracted and used at unification time; this is when appropriate instantiations will satisfy the criteria of releasing residuations for more processing. The unification algorithm will detail the actions taken as part of the backchaining operation, and the ensuing release of residuations for more processing. Failure of released residuated computations (unifications or resolution of ground-decidable predicates), simply calls the regular backtracking algorithm modulo a more sophisticated treatment of the trailing of variable instantiations augmented with trailing residuations needed to be undone as part of the backtracking process.

4.2 Informal Syntax for Le Fun

We present here a minimal syntax for Le Fun. The idea is not to give an exhaustive description of a "real-life" syntax with all conveniences and sugaring to accommodate aesthetics, but rather to define just enough to focus the reader's attention on the specific originality of Le Fun's syntax—namely, a generalization of applicative expressions and first-order terms. Thus, the reader is assumed to be familiar with Prolog's syntax as well as with basic sugaring of the λ-calculus. Therefore, many unspecified details (*e.g.*, pattern-directed conditionals for functions, handling of functional recursion, *etc.*) are left to the reader's taste.

Le Fun's terms are a combination of conventional first-order terms and applicative expressions. More precisely, a Le Fun term is one of the following:

1. *Variables*—represented as capitalized identifiers;

2. *Identifiers*—represented starting with a lower case letter;

3. *Abstractions*—of the form $\lambda X.e$, where X is a variable, and e is a Le Fun term;

4. *Applications*—of the form $e(e_1, \ldots, e_n)$, where e is a Le Fun term, and the e_i's are Le Fun terms.

All classical conventions related to left-associativity, infix notation, and currying of applications are assumed. Those special applications of the form $c(e_1, \ldots, e_n)$, where

c is an identifier known to be a *constructor* symbol, and the e_i's are Le Fun terms are called *constructions*.

A Le Fun program consists of a sequence of *equations* and *clauses*. An equation is of the form $f = e$ where f is an identifier called an *interpreted symbol*, and e is a Le Fun term. In the case where e is an abstraction of the form $\lambda X_1. \ldots .\lambda X_n.e'$, we may also write $f(X_1, \ldots, X_n) = e'$. A clause is defined exactly as in Prolog, with the difference that Le Fun terms are expected where first-order terms are in Prolog—*i.e.*, as predicate arguments. Such literals which constitute Le Fun clauses will be called Le Fun literals.

The lexical distinction between constuctor and interpreted symbols is simply that a constructor is any identifier which does not appear in a left-hand side of an equation. For those, fixed arity is assumed. Hence, any construction with root constructor of arity n must have exactly n arguments. If it has more, the term is ill-formed. If it has less, then the term is not a construction, but an abstraction. Indeed, if c is an n-ary contructor, the term $c(e_1, \ldots, e_k)$ for $k < n$ is in reality the term $\lambda X_1. \ldots .X_{n-k}.c(e_1, \ldots, e_k, X_1, \ldots, X_{n-k})$, where the X_j's do not occur free in any of the e_i's.

References

[1] Aït-Kaci, H. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* Ph.D. Thesis. Computer and Information Science, University of Pennsylvania. Philadelphia, 1984.

[2] Aït-Kaci, H., and R. Nasr, "Logic and Inheritance." *Proceedings of POPL'86 Symposium*, St-Petersburg Beach, FL, pp. 219–228. 1986.

[3] Aït-Kaci, H. and R. Nasr, ": A Logic Programming Language with Built-in Inheritance." *Journal of Logic Programming* **3**(3), pp. 187–215. 1986.

[4] Aït-Kaci, H., "An Algebraic Semantics Approach to the Effective Resolution of Type Equations." *Journal of Theoretical Computer Science* **45**, pp. 293–351. 1986.

[5] Aït-Kaci, H. and R. Nasr, *Residuation: A Paradigm for Integrating Logic and Functional Programming.* MCC Technical Report AI-359-86. Microelectronics and Computer Technology Corporation, Austin, TX. October 1986.

[6] Aït-Kaci, H., Boyer, R., Lincoln, P., and R. Nasr, *The Efficient Implementation of Object Inheritance.* MCC Technical Report AI-102-87, AI/ISA Project. Microelectronics and Computer Technology Corporation, Austin, TX. May, 1987.

[7] Aleliunas, R., and F. Bronsard, *Constructive Mathematics for Logic Programming.* Technical Draft, IBM Canada Laboratory, Toronto. 1987.

[8] Barendregt, H.P., *The Lambda-Calculus. Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics **103**. North-Holland, Amsterdam. (Revised Edition) 1984.

[9] Barwise, J. and S. Feferman (Eds.), *Model-Theoretic Logics.* Springer-Verlag, Berlin. 1985.

[10] de Bruijn, N.G., "A Survey of the Project AUTOMATH." In J.P. Seldin and J.R. Hindley (Eds.), *To H.B. Curry: Essays in Combinatory Logic, Lambda-Calculus, and Formalism*, pp. 589–606. Academic Press, New York. 1980.

[11] Cardelli, L., *A Polymorphic λ-calculus with Type:Type.* Technical Report, Digital Equipment Corporation Systems Research Center. Palo Alto, CA. May, 1986.

[12] Church, A., "A Formulation of the Simple Theory of Types." *Journal of Symbolic Logic* **5**, pp. 56-68. 1940.

[13] Church, A., *The Calculi of Lambda Conversion. Publishers?* 1941.

[14] Curry, H.B., and *?*. Feys, *An Introduction to Combinatory Logic.* North-Holland Elsevier, Amsterdam. 195*?*.

[15] Clément, D., Despeyroux, J., Despeyroux, Th., Hascoët, L., and G. Kahn, *Natural Semantics on the Computer.* Rapport de Recherche N^o **416**, INRIA-Sophia Antipolis. Valbonne, France. June, 1985.

[16] Cohen, E., *Type Theory and Non-Determinism.* Doctoral Thesis Proposal, The University of Texas at Austin (Computer Sciences). Austin, TX. December, 1986.

[17] Constable, R.L., *et al.*, *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, Englewood Cliffs, NJ. 1986.

[18] Coquand, Th., *Une Théorie des Constructions*. Thèse de Doctorat de $3^{\grave{e}me}$ Cycle, Université de Paris-VII (Mathématique, Informatique), Paris. Janvier, 1985.

[19] Coquand, Th., and G. Huet, *Constructions: a Higher Order Proof System for Mechanizing Mathematics*. Rapport de Recherche N^o **401**, INRIA-Rocquencourt. Le Chesnay, France. May, 1985.

[20] Coquand, Th., and G. Huet, "Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions." Presented at the *Colloque de Logique d'Orsay*. Orsay, France. July, 1985.

[21] Coquand, Th., "An Analysis of Girard's Paradox." *Proceedings of the 1st Symposium on Logic In Computer Science*, pp. 227–236. Cambridge, MA. June, 1986.

[22] Cousineau, G., Curien, P.-L., and M. Mauny, "The Categorical Abstract Machine." In J.-P. Jouannaud (Ed.), *Proceedings of the Conference on Functional Programming and Computer Architecture*, Nancy, France, September 1985. LNCS **201**. Springer-Verlag. 1985.

[23] Curien, P.-L., *Categorical Combinators, Sequential Algorithm and Functional Programming*. Research Notes in Theoretical Computer Science. Pittman, London. 1986.

[24] D.T. van Daalen, *The Language Theory of AUTOMATH*. Ph.D. Thesis. Technical University of Eindhoven, The Netherlands. 1980.

[25] Gallier, J.H., *Logic for Computer Science. Foundations of Automatic Theorem Proving*. Harper & Row, New York. 1986.

[26] Girard, J.-Y., *Interprétation Fonctionnelle et Elimination des Coupures de l'Arithmétique d'Ordre Supérieure*. Thèse de Doctorat d'Etat, Université de Paris-VII (Mathématique), Paris. 1972.

[27] Girard, J.-Y., "Quantitative and Qualitative Semantics." Invited Lecture, *1st Symposium on Logic In Computer Science*. Cambridge, MA. June, 1986. (A short abstract is on page 258 of the proceedings.)

[28] Harper, R., Honsell, F., and G. Plotkin, "A Framework for Defining Logics." To appear in the *Proceedings of the 2nd Symposium on Logic In Computer Science*. Ithaca, NY. June, 1987.

[29] Huet, G., "Equational Systems for Category Theory and Intuitionistic Logic." Invited Lecture, *1st Conference in Rewriting Techniques and Applications*, Dijon, France. May, 22, 1985. (This lecture was not included in the conference proceedings.)

[30] Huet, G., *Intuitionistic Higher-Order Natural Deduction as a Programming Methodology*. Lecture Notes, AI Program, Microelectronics and Computer Technology Corporation. Austin, TX. December 18, 1985.

[31] Huet, G., *Formal Structures for Symbolic Deduction and Computation*. Lecture Notes, Computer Science Department, Carnegie-Mellon University. Pittsburgh, PA. 1986.

[32] Huet, G., *Cartesian Closed Categories and Lambda-Calculus*. Lecture Notes, Computer Science Department, Carnegie-Mellon University. Pittsburgh, PA. 1986.

[33] Longo, G., "The Lambda-Calculus and its Connections to Proof-Theory, Category Theory, Higher Type Recursion Theory." Preliminary Draft, *Workshop on the Semantics of Computer and Human Languages*. Half-Moon Bay, CA. March 1987. Organized by J. Barwise *et al.*, CSLI, Stanford, CA. (Proceedings are forthcoming.)

[34] Martin-Löf, P. "Constructive Mathematics and Computer Programming." In L.J. Cohen *et al.* (Eds.), *Proceedings of 6th International Congress for Logic, Methodology, and Philosophy of Science*, pp. 153-175. North-Holland, Amsterdam. 1982.

[35] Meyer, A.R., "What Is a Model of the Lambda Calculus?" *Information and Control* **52**, pp. 87-122. 1982.

[36] Meyer, A.R., and M.B. Reinhold, " 'Type' Is Not a Type: Preliminary Report." *Proceedings of POPL'86 Symposium*, St-Petersburg Beach, FL, pp. 287–295. 1986.

[37] Mohring, C., "Algorithm Development in the Calculus of Constructions." *Proceedings of the 1st Symposium on Logic In Computer Science*, pp. 84–91. Cambridge, MA. June, 1986.

[38] Mosses, P.D., "Modularity in Action Semantics." Preliminary Draft, *Workshop on the Semantics of Computer and Human Languages*. Half-Moon Bay, CA. March 1987. Organized by J. Barwise *et al.*, CSLI, Stanford, CA. (Proceedings are forthcoming.)

[39] Nordström, B., "Programming in Constructive Set Theory: Some Examples." *Proceedings of POPL'8? Symposium*, *where?*, pp. 141–153. 198*?*.

[40] Plotkin, G.D., *Lattice-Theoretic Properties of Subsumption*. Technical Memorandum MIP-R-77, Department of Machine Intelligence and Perception. University of Edinburgh, Scotland. 1977.

[41] Prawitz, D., *Natural Deduction. A Proof-Theoretical Study*. Almquist & Wiksell, Stockholm. 1965.

[42] Reynolds, J.C., "Towards a Theory of Type Structure." *Programming Symposium, Paris. LNCS* **19**, pp. 408–425. Springer Verlag, Berlin. 1974.

[43] Scott, D. "Data Types as Lattices." *SIAM Journal of Computing* **5**(3), pp. 522–587. September, 1976.

[44] Seldin, J.P., and J.R. Hindley (Eds.), *To H.B. Curry: Essays in Combinatory Logic, Lambda-Calculus, and Formalism*. Academic Press, New York. 1980.

[45] Statman, R., "On Translating Lambda Terms into Combinators: the Basis Problem." *Proceedings of the 1st Symposium on Logic In Computer Science*, pp 378–382. Cambridge, MA. June, 1986.

[46] Turner, D., *Intuitionistic Mathematics as a Programming Language*. Lecture Notes., *Computer Science Invited Lecture Series*. Department of Computer Sciences, University of Texas at Austin, Austin. Austin, TX. *Fall 1985?*