

An Empirical Study of the Runtime Behavior of Higher-Order Logic Programs ¹ (*Preliminary Version*)

Spiro Michaylov
Department of Computer and Information Science
The Ohio State University
228 Bolz Hall
2036 Neil Avenue Mall
Columbus, OH 43210-1277, U.S.A.
spiro@cis.ohio-state.edu

Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890, U.S.A.
fp@cs.cmu.edu

1 Introduction

Implementation technology for higher-order logic programming languages such as λ Prolog [17] and Elf [21] is still in its infancy. There are many features of these languages that do not occur in ordinary Prolog programs, such as types, variable binding constructs for terms, embedded implication and universal quantification, or dependent types and explicit construction of proofs. Some initial work on compiler design for higher-order logic programming languages can be found in [11, 16, 18, 19]². At the same time, the language design process for such languages is far from complete. Extensions [2, 7] as well as restrictions [14] of λ Prolog have been proposed to increase its expressive power or simplify the language theory or its implementation.

Obviously, further language design and implementation efforts must be closely linked. It is easy to design unimplementable languages or implement unusable languages. In order to understand and evaluate the challenges and available choices, we report the results of an empirical study of existing example programs. We chose Elf over λ Prolog for this study for two reasons: (1) accessibility of the large suite of examples, and (2) ease of instrumentation of the Elf interpreter to perform measurements. Many of these examples can be trivially transformed into λ Prolog programs, and essentially the same issues arise regarding their runtime behavior. We will discuss later which

¹This research was sponsored partly by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

²See also the paper by Kwon and Nadathur in this volume

measurements are specific to Elf.

Currently, we have access to about 10,000 lines of Elf code, written mostly by the authors and students in a course on *Computation and Deduction* taught in the Spring of 1992. We selected a sample of 12 representative examples of about 3500 total lines of code to conduct this study. The examples cover a range of applications from logic and the theory of programming languages. They are explained further in Section 3.

We briefly summarize what we consider to be some of the central issues and our conclusion.

Full unification in higher-order languages is clearly impractical, due to the non-existence of minimal complete sets of most-general unifiers [8]. Therefore, work on λ Prolog has used Huet's algorithm for *pre-unification* [8], where so-called flex-flex pairs (which are always unifiable) are postponed as constraints, in effect turning λ Prolog into a constraint logic programming language. Yet, even pre-unifiability is undecidable, and sets of most general pre-unifiers may be infinite. While undecidability has not turned out to be a severe problem, the lack of unique most general unifiers makes it difficult to accurately predict the run-time behavior of a λ Prolog program that attempts to take advantage of full higher-order pre-unification. It can result in thrashing when certain combinations of unification problems have to be solved by extensive backtracking. Moreover, in a straightforward implementation, common cases of unification incur a high overhead. These problems have led to a search for natural, decidable subcase of higher-order unification. Miller [14] has suggested a syntactic restriction (L_λ) to λ Prolog, easily extensible to related languages [22], where most general unifiers are unique modulo $\beta\eta\alpha$ -equivalence.

Miller's restriction has many attractive features. Unification is deterministic and thrashing behavior due to unification is avoided. Higher-order unification in its full power can be implemented if some additional control constructs (**when**) are available [15].

However, our study suggests that this solution is unsatisfactory, since it has a detrimental effect on programming methodology, and potentially introduces a new efficiency problem. Object-level variables are typically represented by meta-level variables, which means that object-level capture-avoiding substitution can be implemented via meta-level β -reduction. The syntactic restriction to L_λ prohibits this implementation technique, and hence a new substitution predicate must be programmed for each object language. Not only does this make programs harder to read and reason about, but a substitution predicate will be less efficient than meta-language substitution.

This is not to diminish the contribution that L_λ has made to our understanding of higher-order logic programming. The operational semantics of Elf, in contrast to λ Prolog, is based on solving all dynamically arising equations that lie within an appropriate extension of L_λ to dependent types. All other equations (solvable or not) are postponed as constraints. We found that this addresses the problems with higher-order unification without compromising programming methodology.

This still leaves open the question whether this constraint satisfaction algorithm can be implemented efficiently. Part of our study was aimed at determining the relative frequency of various forms of equations, in order to guide future design of efficient implementations.

In this paper we study the run-time behavior of a large suite of Elf programs, and demonstrate the following:

- While a large proportion of programs are outside L_λ syntactically, the cases of unification that occur dynamically are almost all deterministic.

- All of the programs behave well if nondeterministic cases of unification are delayed until they are deterministic.
- While most programs at some point use non-trivial cases of higher-order unification, the vast majority of unification instances are extremely simple, in fact, essentially Prolog unification.

This empirical study has been performed by instrumenting an Elf interpreter to count:

- the relative frequency of different cases of unification,
- the relative frequency of various instances of substitution,
- the number of times non-deterministic unification would arise were these cases not delayed.

This leads us to suggest a strategy for efficient implementation of higher-order logic programming languages, which is essentially the strategy described for Constraint Logic Programming languages in [9, 12]. That is:

- The languages should not be restricted syntactically.
- The unification instances corresponding to those of L_λ should be identified as *directly solvable*, and the remainder as *hard*. Hard constraints should be delayed until they become directly solvable as a result of further variable instantiation. The relevant terminology, concepts and implementation methods are described in [10].
- Data structures and algorithms should be designed to favor the simple cases of unification.

2 Properties of Programs

Since our concern in this paper is with efficient implementation (and its interaction with language design), the properties of programs that we most need to study are the dynamic properties: how frequently do various phenomena arise when typical queries are executed? This allows us to tune data structures and algorithms. On the other hand, to evaluate the possibility of syntactic restrictions, we also need to know what occurs syntactically in programs. We begin by discussing these syntactic properties and why they are of interest. Then we go on to discuss the dynamic properties.

2.1 Static Properties

L_λ vs. general variable applications. Because of our interest in the syntactic restriction to L_λ , we need to understand how often and why programs do not fall into this subset. An important use of general variable applications appears in a rule like the following (taken from a natural semantics in [13])

```
eval_app_lam      : eval (app E1 E2) V
                   <- eval E1 (lam E1')
                   <- eval E2 V2
                   <- eval (E1' V2) V.
```

where we see an application of two existential variables (E1' V2) to implement substitution in an object language by meta-level β -reduction.

Even within the L_λ subset, we can observe interesting static properties of programs. For example, many programs structurally recurse through an object language expression, where the object is represented using higher-order abstract syntax. Consider the rule above: the head of this rule requires only first order unification, which could be implemented as simple variable binding.

Type redundancy. Both in λ Prolog and Elf there is a potential for much redundant runtime type computation. In λ Prolog, this is due to polymorphism (see [11]), in Elf it is due to type dependency. Such redundancy can be detected statically. However, the question about the dynamic properties of programs remains: how much type computation remains after all redundant ones have been eliminated.

Level of indexing. This is an Elf-specific property of a program. Briefly, a (simple) type is a level 0 type family. A type family indexed by objects of level 0 type is a level 1 type family. In general, a type family indexed by objects whose type involves level n families is a family of level $n + 1$. For example,

```
o : type.                % propositions, level 0.
pf : o -> type.          % proofs of propositions, level 1.
norm : pf A -> pf A -> type. % proof transformations, level 2.
proper : norm P Q -> type. % proper proof transformations, level 3.
```

This is of interest because the level of indexing determines the amount of potentially redundant type computation. Empirically, it can be observed that programs at level 2 or 3 have in some respects different runtime characteristics than programs at level 1. We have therefore separated out the queries of the higher-level. This also helps to separate out the part of our analysis which is directly relevant to λ Prolog, where all computation happens at levels 0 and 1 (due to the absence of dependency).

2.2 Dynamic Properties

The major dynamic properties studied in this paper are substitution, unification and constraint solving.

Substitution. Substitution can be a significant factor limiting performance. It is thus important to analyze various forms of substitution that arise during execution. When measuring these, our concern is simple: substitutions with anything other than parameters (*uvars*) result from the fragment of the language outside L_λ , so these represent substitutions that would have had to have been performed using Elf code if the L_λ restriction had been applied. Moreover, the relative frequency of parameter substitution suggests that it is crucial for it to be highly efficient, while general substitution is somewhat less critical. A proposal regarding efficient implementation of terms has been made in [18]. For our study we eliminated substitutions which arose due to clause copying and during type reconstruction, since these are residual effects of the interpreter and would most likely be eliminated in any reasonable compiler.

Unification and Constraint Satisfaction. We measure various aspects of unification and constraint satisfaction. Terms involved in equations (disagreement pairs) are classified as *rigid* (con-

stant head), *uvars* (parameters, *i.e.*, temporary constants), *evars* (simple logic variables), *gvars* (generalized variables, *i.e.*, logic variables applied to distinct, dominated parameters [14]), *flexible* (compound terms with a logic variable at the head, but not a gvar), *abst* (a term beginning with a λ -abstraction), or *quant* (a term beginning with a Π -quantification, in Elf only).

One of our goals is to determine how close Elf computations come to Prolog computations in several respects:

- How many pairs, at least at the top level, require essentially Herbrand unification? These are the rigid-rigid and evar-anything cases.
- How many pairs still have unique mgus, that is, gvar-gvar, or admit a unique strategy for constraint simplification, that is, gvar-rigid, abst-anything, or quant-anything?
- How often do the remaining cases arise (which are postponed to avoid branching)?
- How successful is rule indexing (as familiar from Prolog) to avoid calls to unification?

In our opinion, while we have not yet completed the required experiments, it is also very important to determine the following:

- How important is the occurs-check (extended to deal with a dependency check)?
- How much time is spent on type computations as compared to object computations?
- How much time is spent on proof computations, when it is requested by the user or required for further computation?

3 Study of Programs

In this section we report our preliminary findings. We currently have detailed statistics on the kinds of disagreement pairs that arise during unification, and the kind of substitution that is performed during unification and search.

3.1 The Examples

Figures 1 and 2 show the data for basic computation queries and proof manipulation queries respectively, for the range of programs. Thus Figure 1 is especially applicable to the understanding of λ Prolog programs, while Figure 2 measures Elf-specific behavior.

The two tables in each figure give data on five areas of interest, as follows:

- *All Unifications*

The total gives an indication of computational content, while the breakdown indicates the usefulness of first-argument indexing and the amount of deep search.

<i>Unif</i>	Total number of subgoal/head pairs to be unified.
<i>%Ind</i>	Percentage of above total unifications avoided by rule indexing.
<i>%S</i>	Percentage of total unifications that succeeded.
<i>%F</i>	Percentage of total unifications that failed.

- *Dynamic Unifications*

It is also useful to have this information for rules assumed through embedded implication, since indexing of such rules is more complicated, and compilation has a runtime cost.

Dyn Total number of subgoal/head pairs to be unified, where the head is from a rule assumed (dynamically) through embedded implication.

%Ind, %S, %F

Percentages of number of unifications with heads from dynamic rules, as above.

- *Dynamic/Assume*

By knowing how many rules are assumed dynamically, and on average how often they are used, we can see whether it is worthwhile to index and compile such rules or whether they should be interpreted.

Ass Number of rules assumed by implication.

U/Ass Normalized ratio of total unifications with dynamic rules to number of rules assumed by implication.

AU/Ass As above, but using only those rules where the unification was not avoided through indexing.

- *Disagreement Pairs*

We study the kinds of disagreement pairs that arise to determine which kinds of unification dominate.

Tot Total number of disagreement pairs examined throughout the computation.

%E-? Percentage of disagreement pairs that involved a simple evar.

%G-? Percentage of disagreement pairs that involved a gvar which is *not* a simple evar.

%R Percentage of disagreement pairs between two rigid terms.

%A Percentage of disagreement pairs between two abstractions.

- *Substitutions*

Substitutions and abstractions (the inverse of uvar substitutions) are expensive, and the efficiency of one can be improved at the expense of the other. Furthermore, some kinds of substitutions are more costly than others. Thus it is useful to know what kinds of substitutions arise, how often both substitution and abstraction arise, and their relative frequency.

Tot Total number of substitutions for bound variables.

%Uv Percentage of the above where a uvar is substituted.

Abs Number of abstractions over a uvar.

Abs/Uv Normalized ratio of such abstractions to substitutions of uvars.

The examples used are as follows:

- Extraction — *Constructive theorem proving and program extraction* [1]

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Mini-ML	15333	87	13	0	1532	93	7	0	67	22.87	1.61
Canonical	177	66	28	6	8	50	50	0	3	2.67	1.33
Prop	677	60	30	10	41	44	41	15	9	4.56	2.56
F-O	359	65	28	7	33	18	82	0	17	1.94	0.07
Forsythe	2087	38	23	39	16	25	75	0	10	1.60	1.20
Lam	240	50	40	10	26	80	15	5	4	6.50	1.25
Polylam	982	65	34	1	389	88	12	1	45	8.64	1.00
Records	2459	61	31	8	274	61	39	0	28	9.79	3.79
DeBruijn	451	25	39	36	5	40	60	0	5	1.00	0.60
CLS	278	0	32	68	0	-	-	-	0	-	-

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Mini-ML	8716	47	0	52	0	6411	98	0	0.00
Canonical	427	41	8	56	0	180	96	36	0.21
Prop	1681	54	0	45	1	202	100	8	0.04
F-O	438	40	6	58	0	108	100	58	0.54
Forsythe	5812	43	0	57	0	39	100	0	0.00
Lam	874	41	0	59	0	149	86	0	0.00
Polylam	2085	48	3	50	1	7907	89	81	0.01
Records	3880	46	3	53	0	1347	100	204	0.15
DeBruijn	1554	44	1	56	0	688	97	16	0.02
CLS	2455	36	0	65	0	0	-	0	-

Figure 1: Basic Computation

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Extraction	878	89	11	0	165	82	17	1	54	3.05	0.54
Mini-ML	2415	73	11	16	107	87	13	0	10	10.70	1.40
CPS	162	59	41	0	72	57	43	0	48	1.50	0.65
Prop	4957	67	25	8	509	71	14	15	71	7.17	2.10
F-O	1140	69	27	4	27	0	100	0	13	2.08	2.08
Lam	369	50	44	6	36	75	22	3	12	3.00	0.75
DeBruijn	627	20	44	36	77	51	30	19	24	3.21	1.58
CLS	333	30	42	28	0	-	-	-	0	-	-

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Extraction	1580	22	9	66	6	9016	96	1124	0.01
Mini-ML	5872	17	1	76	6	3644	96	55	0.02
CPS	592	24	34	54	0	1509	100	1029	0.68
Prop	13809	35	3	63	1	12040	99	443	0.04
F-O	6800	21	1	74	5	12716	99	38	0.00
Lam	3464	22	2	74	3	1825	94	83	0.05
DeBruijn	13441	15	1	71	13	14632	99	150	0.01
CLS	5227	23	0	77	0	2	-	0	-

Figure 2: Proof Manipulation

Program	All Unifications				Dynamic Unifications				Dynamic/Assume		
	Unif	%Ind	%S	%F	Dyn	%Ind	%S	%F	Ass	U/Ass	AU/Ass
Comp	5562	90	10	0	1532	92	8	0	67	22.87	1.61
ExpComp	7200	70	10	20	1798	80	8	12	87	10.67	4.30
ExpIndComp	7200	88	10	2	1798	92	8	0	87	10.67	4.30
Trans	2159	70	11	19	107	86	14	0	10	10.70	1.40
ExpTrans	5255	29	10	59	633	15	13	72	67	9.45	8.06
ExpIndTrans	5255	76	11	13	633	84	13	3	67	9.45	8.06

Program	Disagreement Pairs					Substitutions			
	Tot	%E-?	%G-?	%R	%A	Tot	%Uv	Abs	Abs/Uv
Comp	2424	43	0	57	0	445	97	0	-
ExpComp	10765	24	4	56	17	22743	100	778	0.03
ExpIndComp	4251	32	10	52	8	15801	100	778	0.05
Trans	5709	17	1	76	7	3612	96	55	0.02
ExpTrans	27342	20	4	61	16	280679	97	2522	0.01
ExpIndTrans	13482	17	8	65	12	264399	98	2522	0.01

Program	Computation	Transformation
Implicit	1.30	2.48
Explicit	8.48	155.09
Explicit-Indexed	5.80	145.89

Figure 3: Mini-ML comparison

This example involves a large number of level 2 judgments. Indexing is particularly effective here, and assumed rules are used unusually infrequently. Note that these examples do not include any basic computation.

- Mini-ML [13]

An implementation of Mini-ML, including type-checking, evaluation, and the type soundness proof. Because of the large number of cases, indexing has a stronger effect than in all other examples.

- CPS — *Interpretation of propositional logics and CPS conversions* [3, 23]

Various forms of conversion of simply-typed terms to continuation-passing and exception-returning style. Substitutions are all parameter substitutions, and unification involves an unusually large number of gvar-anything cases. The redundant type computations are very significant in this example—all the examples are level 2 judgments.

- Canonical — *Canonical forms in the simply-typed lambda-calculus* [21]

Conversion of lambda-terms to canonical form. A small number of non-parameter substitutions arise, but mostly unification is first-order. Here, too, there is much redundant type computation.

- Prop — *Propositional Theorem Proving and Transformation* [5]

This is mostly first-order. In the transformations between various proof formats (natural deduction and Hilbert calculi), a fairly large number of assumptions arise, and are quite heavily used. Unification involves a large number of evar-anything cases.

- F-O — *First-order logic theorem proving and transformation*

This includes a logic programming style theorem prover and transformation of execution trace to natural deductions. There is rather little abstraction.

- Forsythe — *Forsythe type checking*

Forsythe is an Algol-like language with intersection types developed by Reynolds [24]. This example involves very few substitutions, all of which are parameter substitutions. Thus the runtime behavior suggests an almost entirely first-order program, which is not apparent from the code.

- Lam — *Lambda calculus convertibility*

Normalization and equivalence proofs of terms in a typed λ -calculus. A relatively high percentage of the substitutions are non-parameter substitutions.

- Polylam — *Type inference in the polymorphic lambda calculus* [20]

Type inference for the polymorphic λ -calculus involves postponed constraints, but mostly parameter substitutions. Unification can be highly non-deterministic. This is not directly reflected in the given tables, as this is the only one of our examples where any hard constraints

are delayed at run time (and in only 10 instances). In fact, one of these hard constraints remains all the way to the end of the computation. This indicates that the input was not annotated with enough type information (within the polymorphic type discipline, not within the framework).

- Records — *A lambda-calculus with records and polymorphism*

Type checking for a lambda-calculus with records and polymorphism as described in [6]. This involves only parameter substitutions, and assumptions are heavily used.

- *DeBruijn* [4]

A compiler from untyped λ -terms to terms using deBruijn indices, including a call-by-value operational semantics for source and target language. The proof manipulation queries check compiler correctness for concrete programs. Indexing works quite poorly, and an unusually large number of abst-abst cases arise in unification.

- *CLS* [4]

A second compiler from terms in deBruijn representation to the CLS abstract machine. Simple queries execute the CLS machine on given programs, proof manipulation queries check compiler correctness for concrete programs. This is almost completely first-order.

Overall, the figures suggest quite strongly that most unification is either simple assignment or first-order (Herbrand) unification, around 95%, averaged over all examples. Similarly, substitution is the substitution of parameters for λ -bound variables in about 95% of the cases. The remaining 5% are substitution of constants, variables, or compound terms for bound variables. These figures do not count the substitution that may occur when clauses are copied, or unifications or substitutions that arise during type reconstruction.

Finally, we compare the Mini-ML program with a version written using explicit substitution, to evaluate the effects of a syntactic restriction along the lines of L_λ . The computation queries had to be cut down somewhat because of memory restrictions. In Figure 3 we show the same data as above for the computation and transformation queries with and without explicit substitution. We also show a version with explicit substitution with the substitution code rewritten to take better advantage of indexing. Then we compare the CPU times (in seconds) for the two sets of queries for all three versions of the program, using a slightly modified³ Elf version 0.2 in SML/NJ version 0.80 on a DEC station 5000/200 with 64MB of memory and local paging. These results show that there is a clear efficiency disadvantage to the L_λ restriction, given present implementation techniques. Note that the disadvantage is greater for the transformation queries, since a longer proof object is obtained, resulting in a more complicated proof transformation. Explicit substitution increases the size of the relevant code by 30%.⁴ Substitutions dominate the computation time, basically because one meta-level β -reduction has been replaced by many substitutions. These substitutions

³The modification involves building proof objects only when needed for correctness.

⁴Actually, the meta-theory was not completely reduced to L_λ , because type dependencies in the verification code would lead to a very complex verification predicate. We estimate that the code size would increase an additional 5% and the computation time by much more than that.

should all be parameter (uvar) substitutions, which suggests that some (but clearly not all) of the performance degradation could be recovered through efficient uvar substitution. See the previous footnote on why non-parameter substitutions still arise in the proof transformation examples.

3.2 Further Summary Analysis

A few figures were obtained through simple summary profiling and await further detailed analysis. The summary figures suggest that, for examples of average size, omitting the (extended) occurs-check in the current implementation can result in speed improvements of between 40% and 60%. This is therefore an upper bound on the speed-up that could be achieved through smart compilation to avoid the occurs-check.

The current implementation avoids building proof objects to some extent (applicable to Elf only), which saves about 50% of total computation time, although the savings are not additive (some of the occurs-check overhead arises in building proofs).

4 Conclusions

We briefly summarize our preliminary conclusions, which are very much in line with the experience gained in other constraint logic programming languages [12].

Language Design. Statically prohibiting difficult cases in unification (by a restriction to L_λ , for example) is not a good idea, since it leads to a proliferation of code and significantly complicates meta-theory as it is typically expressed in Elf. This coincides with experience in other constraint logic programming languages such as CLP(\mathcal{R}) and Prolog-III.

Our recommendation is to delay hard constraints (including flexible-rigid pairs that are not gvar-rigid pairs) and thus avoid branching in unification at runtime.

Language Implementation. Indexing and representation of terms in the functor/arg notation (rather than the curried notation typical for λ -calculi) are crucial for achieving good performance, as they enable quick classification of disagreement pairs and rigid-rigid decomposition. It is rather obvious that runtime type computation must be avoided whenever possible as suggested in [11], and that proof building must be avoided whenever the proof object will not be needed.

We need special efficient mechanisms for direct binding and first-order unification. Furthermore, unification as in L_λ and substitution of parameters for bound variables are very important special cases that merit special attention. Efficiency of substitution of constants or compound terms for bound variables is important in some applications, but not nearly as pervasive and deserves only secondary consideration.

5 Future Work

A study such as this is necessarily restricted and biased by the currently available implementation technology. The most important figures that are currently missing:

- How much type computation can be eliminated, and what would be the effect of eliminating redundant type computation on the remaining figures.

- How often can the occurs-check be avoided.

In longer term work, one would also like to analyse the effect of other standard compilation techniques of logic programming languages in this new setting, but much of this requires an implemented compiler as a basis.

References

- [1] Penny Anderson. *Program Development by Proof Transformation*. PhD thesis, Carnegie Mellon University, 1992. In preparation.
- [2] Scott Dietzen and Frank Pfenning. Higher-order and modal logic as a framework for explanation-based generalization. *Machine Learning*, 9:23–55, 1992.
- [3] Timothy G. Griffin. Logical interpretations as computational simulations. Draft paper. Talk given at the North American Jumelage, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1991.
- [4] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.
- [5] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. Technical Report CMU-CS-92-191, Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1992.
- [6] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 131–142, Orlando, Florida, January 1991.
- [7] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 199?. To appear. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.
- [8] Gérard Huet. A unification algorithm for typed λ -calculus. *Theoretical Computer Science*, 1:27–57, 1975.
- [9] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(\mathcal{R}) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.
- [10] Joxan Jaffar, Spiro Michaylov, and Roland Yap. A methodology for managing hard constraints in CLP systems. In Barbara Ryder, editor, *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 306–316, Toronto, Canada, June 1991.

- [11] Keehang Kwon, Gopalan Nadathur, and Debra Sue Wilson. Implementing logic programming languages with polymorphic typing. Technical Report CS-1991-39, Duke University, Durham, North Carolina, October 1991.
- [12] Spiro Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems*. PhD thesis, Carnegie Mellon University, August 1992. Available as Technical Report CMU-CS-92-168.
- [13] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.
- [14] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming: International Workshop, Tübingen FRG, December 1989*, pages 253–281. Springer-Verlag LNCS 475, 1991.
- [15] Dale Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 255–269. MIT Press, July 1991.
- [16] Gopalan Nadathur and Bharat Jayaraman. Towards a WAM model for lambda Prolog. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 1180–1198. MIT Press, October 1989.
- [17] Gopalan Nadathur and Dale Miller. An overview of λ Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.
- [18] Gopalan Nadathur and Debra Sue Wilson. A representation of lambda terms suitable for operations on their intensions. In *Proceedings of the 1990 Conference on Lisp and Functional Programming*, pages 341–348. ACM Press, June 1990.
- [19] Pascal Brisset Olivier Ridoux, Serge Le Huitouze. Prolog/mali. Available via ftp over the Internet, March 1992. Send mail to pm@irisa.fr for further information.
- [20] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.
- [21] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

- [22] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.
- [23] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 537–551, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.
- [24] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.