Higher-Order Logic Programming as Constraint Logic Programming

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

Frank Pfenning
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891, U.S.A.
fp@cs.cmu.edu

Abstract

Higher-order logic programming (HOLP) languages are particularly useful for various kinds of meta-programming and theorem proving tasks because of the logical support for variable binding via λ-abstraction. They have been used for a wide range of applications including theorem proving, programming language interpretation, type inference, compilation, and natural language parsing. Despite their utility, current language implementations have acquired a well-deserved reputation for being inefficient. In this paper we argue that HOLP languages can reasonably be viewed as Constraint Logic Programming (CLP) languages, and show how this can be expected to lead to more practical implementations by applying the known principles for the design and implementation of practical CLP systems.

1 Introduction

Higher-order logic programming (HOLP) languages [17] typically use a typed λ-calculus as their domain of computation. In the case of λProlog [18] it is the simply-typed λ-calculus, while in the case of Elf [22] it is a dependently typed λ-calculus. These languages are particularly useful for various kinds of meta-programming and theorem proving tasks because of the logical support for variable binding via λ-abstraction. They have been used for a wide range of applications including theorem proving [3], programming language interpretation [5, 13], type inference [21], compilation [6], and natural language parsing [20]. Despite their utility, current language implementations have acquired a well-deserved reputation for being inefficient. In this paper we argue that HOLP languages can reasonably be viewed as Constraint Logic Programming (CLP) languages [8]. Measurements with an instrumented Elf interpreter confirm that such a view can produce practical benefits, as the known principles for the design and implementation of practical CLP systems [9, 12] are directly applicable to making implementations of HOLP languages more efficient.

The core domain of the languages we consider is the set of typed λ-expressions, where abstraction and application are the only interpreted operations and equality is the only relation (interpreted as $\beta\eta\alpha$-convertibility). There are other features of these languages that distinguish them from the CLP language scheme as defined in [8], for example, higher-order predicates, dependent or polymorphic types, modules, embedded implication and universal quantification. Many of these features have been addressed in a satisfactory way in ongoing implementation projects at Duke and IRISA/INRIA in Rennes. Surveys and further

references to the design of these implementations can be found in [11] and [1]. In this paper we will concentrate on the issues related to the view of higher-order logic programming as constraint logic programming which, we believe, has the most fundamental impact on expected execution speed.

2 Solving Equations Between Typed λ-Expressions

Full unification in higher-order languages is clearly impractical, due to the non-existence of minimal complete sets of most-general unifiers [7]. Therefore, work on λProlog has used Huet's algorithm for *pre-unification* [7], where so-called flex-flex pairs (which are always unifiable) are maintained as constraints, rather than being incorporated in an explicit parametric form. 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 λProlog programs that attempt 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 compared to first-order unification. These problems have led to a search for natural, decidable subcases of higher-order unification where most general unifiers exist. Miller [15] has suggested a syntactic restriction (L_λ) to λProlog, easily extensible to related languages [23], 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 [16].

However, our empirical analysis [14] 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. As we will describe below, it forms the basis for our approach to the implementation of HOLP languages.

3 A Practical Approach to Constraint Logic Programming

The generality of the CLP scheme allows languages to be defined that raise two important implementation problems:

- *High overhead for frequently-occurring simple constraints.*
  It has been observed [9, 12] that the constraints that occur most frequently in the execution of programs in many CLP systems are relatively simple. However, the generality needed to solve the more complicated, but rarely occurring constraints tends to introduce overheads for solving all constraints. Ideally, it should be possible to solve the simple constraints without incurring this overhead.

- *Some constraints may be too hard to solve or solve efficiently.*
  For many seemingly desirable domains, the required decision algorithms simply do not exist. For others, the decision problem may be open. For many more domains, either the decision problem is known to be intractable, or the best known algorithms are impractical. Even when a reasonably efficient decision procedure exists, it may be incompatible with the CLP operational model. In particular, a CLP implementation requires incremental satisfiability testing to be efficient. That is, it must be possible to determine efficiently whether a satisfiable set of constraints, augmented with a new constraint, is still satisfiable. Efficiency here loosely means that the time should be proportional more to the size of the added constraint than that of the previous, satisfiable, set. Furthermore, because of backtracking, it must be possible to undo such augmentations of the constraint set efficiently.

Solving the first problem requires that the system be implemented with a bias towards frequently occurring constraints. A data structure that is most appropriate for certain special cases but cumbersome and inefficient for the general case can often result in dramatically improved overall performance.

One approach to the second problem is to syntactically restrict the kinds of constraints on the given domain that can be expressed. Such syntactic restrictions determine what expressions can be constructed using the operators, and what expressions the various relation symbols can be applied to. For example, arithmetic expressions could be restricted to be linear. It turns out that syntactic restrictions on constraints often rule out useful and natural programs. Such programs contain syntactically complex expressions that are typically simplified by the time they are selected at runtime. For example, a non-linear expression could become linear after instantiation of some variables.

We advocate another approach, by which constraints that cannot be decided (or decided efficiently) at the time they arise are delayed with the expectation that the problem will be simplified under additional constraints. This approach can be justified under two diametrically opposed philosophies underlying constraint logic programming.

In the first, perhaps more traditional view of constraint programming, it is important that the programmer should not have to be concerned with when information becomes available but just needs to provide *enough* constraints for it *eventually* to become available. Under this philosophy, the delaying approach achieves some amount of independence of the order in which constraints arrive at the solver without unduly restricting programs. This philosophy and the justification of delay is described in considerable detail by Jaffar *et al.* in [10], where it is argued further that delay does not require the programmer to think substantially more algorithmically.

The second view advances that a constraint logic programming language is a logic with a completely specified operational semantics, which programmers should know in order to predict runtime behavior and evaluate the efficiency of their programs. Under this view, the delaying semantics is simply a design decision in the specification of the language permitting a larger range of algorithms to be expressed concisely and naturally.

It is shown in [10] that delay can be implemented with overhead proportional to the number of delayed constraints whose state is affected by each additional constraint, rather than the total number of delayed constraints. The issue of how to restrict a CLP language appropriately has often arisen and been addressed in different ways in real systems. In CLP($\mathcal{R}$) [9], the selection rule is modified to delay nonlinear constraints until they become linear. A similar approach to nonlinear arithmetic has been adopted in recent commercial versions of Prolog III. Furthermore, the same approach is now used in Prolog III to deal with the intractability of word unification: word equations are delayed until the length of the value of initial variables is known.

We have studied a selection of 12 representative and non-trivial Elf programs with a total of about 3500 lines of code [14]. We analyzed these programs from a static and dynamic perspective. Our study demonstrates that the above observations and strategies for dealing with the problems of CLP languages in general are directly applicable to HOLP languages, and that a considerable performance improvement can be expected. Conversely, the HOLP languages provide further evidence of the general applicability of this approach.

4 Special Cases of HOLP Constraints

Terms in Elf and λProlog that contain no abstraction or functional variables correspond directly to first-order terms (as in Prolog). Our empirical study showed that most unification was either simple assignment or first-order (Herbrand) unification: around 95%, averaged over all examples. When λ-abstractions are present, substitution of a term for a bound variable (β-reduction) is a common operation. Most of these (about 95%) substitute a parameter[1] for a variable. Because first-order unification and parameter substitution dominate, the representation should be designed to handle these cases particularly efficiently.

The obvious representation for terms is that corresponding to first-order abstract syntax: a DAG with special nodes for application, abstraction etc. This is problematic because the frequently occurring first-order unification cases rely heavily on finding the principal functor, which in this representation is at the head

[1] A parameter (sometimes called *eigenvariable*) acts like a constant in unification, but has proper scope. It arises from solving universally quantified goals.

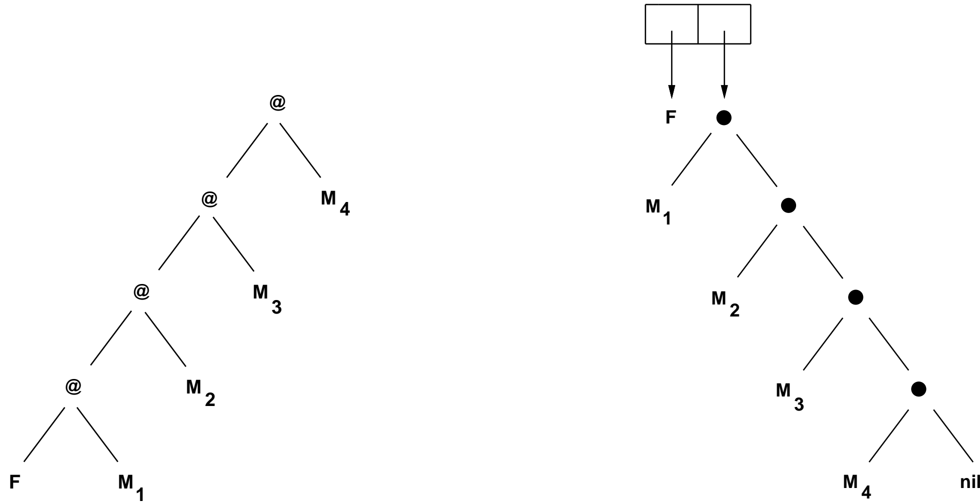

Figure 1: Conventional and Functor/Arguments term representations for Elf

of the spine of applications. Consider, for example, the representation of the term $FM_1M_2M_3M_4$ shown on the left in Figure 1. The major inefficiency results when a disagreement pair must be classified. The classifications are mostly based on the nature of the head and in particular on whether it is a constant or not. Furthermore, in the frequently occurring Rigid-Rigid case (where both heads are parameters or constants), it is necessary to know *which* constant: the pair is only decomposed into argument pairs if the heads are identical; otherwise unification fails. In the left-associative representation, obtaining information about the head is too expensive. This suggests a representation as a pair of a functor and a list of arguments. This representation, for the same example, is shown on the right in Figure 1. Notice that this representation also makes it easier to make use of clause indexing on rigid term heads.

This Prolog-like functor/argument representation can be problematic when the head is a variable, since it may be bound to an expression which requires some normalization, producing a new head, and the old one needs to be stored for backtracking in some way. These complications are outbalanced by the efficiency improvement for the simple cases, since the overwhelming majority of λProlog or Elf equality constraints can be solved by Prolog unification. We conclude that functor/argument representation is an essential optimization for a λProlog or Elf implementation.[2]

In principle, an important part of term comparison in these languages is the test for α-convertibility. The Duke representation proposal [19] suggests a de Bruijn representation [2] of terms for this reason. While that suggestion may well be appropriate, the empirical study showed the comparison of two abstractions to be a rare occurrence, and so this consideration alone should probably not be allowed to determine the choice of term representation.

Similarly, the implementation of substitution is an important issue. The dominant kinds of substitutions are those that replace bound variables by parameters—the representation should be optimized towards handling this case efficiently. This is a much more difficult issue than the representation of application, but our observations suggest that an explicit way to shift variable references in a term (through a special form of environment) might solve the efficiency problem associated with parameter substitutions.

[2]Such a representation was in fact used pervasively in Nadathur's original implementation of λProlog up to LP2.7. In the current Elf implementation, a functor/argument representation is used as an intermediate form in the constraint solver for the reasons cited above.

5 Hard Constraints in HOLP

Even a brief examination of Elf and λProlog programs shows that syntactic restriction to L_λ would affect a significant proportion of programs. While these programs can be rewritten to conform to the L_λ restriction, doing so makes them harder to reason about and, with present implementation technology, significantly less efficient. Furthermore, most programs dynamically conform to the L_λ restriction even without delay, and we are aware of only one useful program that does not run properly when hard constraints are delayed [4]. On the other hand, there are programs that run significantly more efficiently when hard constraints are delayed (for example, type inference in the polymorphic λ-calculus [21]).

The operational semantics of Elf, in contrast to λProlog, 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 delayed. We found that this addresses the problems with higher-order unification without compromising programming methodology. The primary disadvantage of this approach is that one must take care in interpreting the final answer, if it contains delayed constraints, as a conditional: each solution of the remaining constraints yields a proof of the original query. In this section, we state precisely which constraints are deemed to be hard in λProlog and Elf and how they arise, and show how the methodology described in [10] can be used to manage hard constraints in this context.

5.1 Classification of Higher-Order Terms

In Prolog each term can be classified as either a variable, a constant, or a compound term. Solving constraints over higher-order terms requires a finer classification. For example, a term might be a λ-abstraction or a β-redex. The critical cases, however, arise when the head of a term is either a variable or a constant. In our terminology, an *Evar* is an existential variable (logic variable, in Prolog terminology) and a *Uvar* is a parameter (a constant with a well-defined scope introduced when solving a universally quantified goal). An Evar E is said to *depend* on a Uvar x in a goal if E is introduced into the computation within the scope of x. If E depends on x the substitution term for E may contain occurrences of x, otherwise it may not. Terms are then classified as follows.

- **Gvar**
  $Fx_1x_2\cdots x_n, n \geq 0$ where F is an Evar, the x_i are Uvars, F does not depend on any x_i, and the x_i are all distinct.
- **Flex**
  $FM_1M_2\cdots M_n, n \geq 0$ where F is an Evar and the M_i are terms, and the Gvar conditions above are not satisfied.
- **Rigid**
  $fM_1M_2\cdots M_n, n \geq 0$ where f is a constant or a Uvar and the M_i are arbitrary terms.

Note that, to simplify the discussion, types and other classes of disagreement pairs have been omitted, since they are dealt with by straightforward recursive unifications. We should also emphasize that in this discussion the Flex case does not include the Gvar case, unlike in Huet's usage.

5.2 Classification of Constraints

A HOLP system must solve the nine kinds of equations arising from these three kinds of terms, collapsing to six kinds due to symmetry. The table in Figure 2 shows which pairs are directly solved and which are delayed. Note that the disagreement pairs that are directly solved will either have a most general solution or no solution.

The constraint solver state consists of a set of substitutions of the form $X = M$ where X is an Evar and M is a term, such that X does not occur elsewhere, and a set of Flex-Flex pairs. This is an implicit solved form. In the implementation, the Evar-Term pairs are not represented as pairs, but by pointers from the variable to its instantiation term (as in Prolog). However, Flex-Flex pairs must be represented explicitly. In addition, the constraint solver must handle hard constraints, which arise in two ways:

Core Elf Unification Table			
	Gvar	Flex	Rigid
Gvar	unify		
Flex	delay	delay	
Rigid	unify	delay	unify

Figure 2: Core unification table for Elf

1. When a new disagreement pair is a Flex-Rigid pair (under the current substitution). This corresponds to the typical source of hard constraints in languages like CLP($\mathcal{R}$).
2. When additional substitutions are added to the solver as a result of solving a new disagreement pair, and these can be used to rewrite some Flex-Flex pair already in the solver to a pair that is a hard constraint.

The second situation is unusual for constraint languages, since a conjunction of directly solvable constraints may be simplified into a hard constraint. However, this is not problematic in the context of the methodology described in [10]: it merely requires that Flex-Flex pairs be treated as if they were hard constraints when designing the wakeup system, as described below.

5.3 A Wakeup System

In this section, our aim is to describe the management of hard constraints in Elf in terms of the framework developed by Jaffar *et al.* in [10].

We need five wakeup degrees in addition to *awakened*. These are for Flex-Flex, Flex-Rigid, Rigid-Flex, Flex-Gvar, and Gvar-Flex. We note that it is not desirable to combine symmetric cases, because the transitions of the two sides of the equation depend on the binding of different variables.

The transitions between these three forms of expressions that we need to consider are as follows. Note that we do not consider leading abstractions.

1. *Flex* $\Rightarrow$ *Rigid*
   The head F in $FM_1M_2\cdots M_n$ is bound to

   $$\lambda x_1 \cdots \lambda x_k.\, gN_1 \cdots N_m$$

   where g is a constant or a Uvar and the N_i are arbitrary terms. The resulting Rigid term will be of the form

   $$gP_1 \cdots P_l.$$

2. *Gvar* $\Rightarrow$ *Rigid*
   Same as *Flex* $\Rightarrow$ *Rigid*.
3. *Flex* $\Rightarrow$ *Gvar*
   (a) All of the arguments are bound to universal variables, such that the Gvar criteria now hold. (This is very unlikely, and expensive to check for, so it has not been implemented to date).
   (b) The head F of $FM_1M_2\cdots M_n$ is bound to

   $$\lambda x_1 \cdots \lambda x_k.\, Gy_1 \cdots y_m$$

   where G is an existential variable, each y_j is either a Uvar or one of the x_i, and the resulting term is a Gvar, that is, a term of the form

   $$Gz_1 \cdots z_l$$

   such that the z_i are all distinct Uvars, and G does not depend on any of them.

4. *Gvar* $\Rightarrow$ *Flex*
   The head F of $Fx_1x_2\cdots x_m$ is bound to

$$\lambda x_1 \cdots \lambda x_k.\, GN_1 \cdots N_m$$

   where G is a an existential variable and the N_i are terms, such that the Gvar criteria are now violated. The resulting Flex term will be of the form

$$GP_1 \cdots P_l.$$

Notice that the above transitions admit the possibility of cycles: A Flex term can turn into a Gvar term when more information becomes available, and with still more information may turn back into a Flex term, all without backtracking. This makes the wakeup system cyclic, as shown in Figure 3. The two arcs shown using a thinner line correspond to the case that is expensive, unlikely, and omitted in the current implementation. We describe the generic wakeup conditions in symmetric pairs, to avoid notational clutter, and ignore the term that does not change in each pair.

6 Conclusion

Higher-Order Logic Programming languages differ substantially from other Constraint Logic Programming languages. However, our empirical evidence shows that the language design and implementation strategies that have made such a substantial difference to better known CLP languages are applicable here as well.

We believe that the main challenge in the design of an abstract machine and a compiler for HOLP languages that achieves Prolog's efficiency on Prolog-like programs is the design of a representation that permits efficient substitution of parameters for bound variables without incurring an undue overhead for the usual first-order unification computation.

References

[1] Pascal Brisset and Olivier Ridoux. The architecture of an implementation of λProlog: Prolog/mali. In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 195–200, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[2] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[3] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989. Available as Technical Report MS-CIS-87-109.

[4] Juergen Haas and Bharat Jayaraman. Interactive synthesis of definite-clause grammars. In Krzystof Apt, editor, *Proc. Joint International Conference and Symposium on Logic Programming*, pages 541–555, Washington, DC, November 1992. MIT Press.

[5] John Hannan. *Investigating a Proof-Theoretic Meta-Language for Functional Programs*. PhD thesis, University of Pennsylvania, January 1991. Available as technical report MS-CIS-91-09.

[6] 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.

[7] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

`F`	$FM_1M_2\cdots M_n$
`G`	$Fx_1x_2\cdots x_m$
`R`	$fM_1M_2\cdots M_n$

`F -> R`	$(\omega_1,\ \omega_2,\ \omega_{13},\ \omega_{14},\ \omega_{15},\ \omega_{16})$	$F = \lambda x_1 \cdots \lambda x_k.\ gN_1 \cdots N_m$
`F -> G`	$(\omega_5,\ \omega_6)$	$M_1 = x_1 \wedge \cdots \wedge M_n = x_n$
		$\wedge \quad Gvar(Fx_1 \cdots x_n)$
`F -> G`	$(\omega_3,\ \omega_4,\ \omega_{11},\ \omega_{12},\ \omega_{17},\ \omega_{18})$	$F = \lambda x_1 \cdots \lambda x_k.\ Gy_1 \cdots y_m$
		$\wedge \quad (\lambda x_1 \cdots \lambda x_k.\ Gy_1 \cdots y_m)M_1 \cdots M_n \to M'$
		$\wedge \quad Gvar(M')$
`G -> R`	$(\omega_9,\ \omega_{10})$	$F = \lambda x_1 \cdots \lambda x_k.\ gN_1 \cdots N_m$
`G -> F`	$(\omega_7,\ \omega_8)$	$F = \lambda x_1 \cdots \lambda x_k.\ GN_1 \cdots N_m$
		$\wedge \quad (\lambda x_1 \cdots \lambda x_k.\ GN_1 \cdots N_m)M_1 \cdots M_n \to M'$
		$\wedge \quad \neg Gvar(M')$

Figure 3: Wakeup system for Elf

[8] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, pages 111–119. ACM, January 1987.

[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 (TOPLAS)*, 14(3):339–395, July 1992.

[10] Joxan Jaffar, Spiro Michaylov, and Roland Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 306–316, Toronto, Canada, June 1991.

[11] Keehang Kwon and Gopalan Nadathur. An instruction set for higher-order hereditary harrop formulas (extended abstract). In D. Miller, editor, *Proceedings of the Workshop on the λProlog Programming Language*, pages 195–200, Philadelphia, Pennsylvania, July 1992. University of Pennsylvania. Available as Technical Report MS-CIS-92-86.

[12] Spiro Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems*. PhD thesis, School of Computer Science, 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, *Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNCS/LNAI 595.

[14] Spiro Michaylov and Frank Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Proc. Workshop on the λProlog Programming Language*, pages 257–271, Philadelphia, PA, USA, July/August 1992. Appears as University of Pennsylvania technical report MS-CIS-92-86.

[15] 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*, pages 253–281, Tübingen FRG, December 1991. Springer-Verlag LNCS 475.

[16] 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.

[17] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 448–462, London, July 1986. Springer Verlag LNCS 225.

[18] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Seattle, WA, August 1988. MIT Press.

[19] 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, Nice, France, June 1990. ACM Press.

[20] Fernando C. N. Pereira. Semantic interpretation as higher-order deduction. In *Proceedings of the Second European Workshop on Logics and AI*, pages 78–96. Springer-Verlag LNCS 478, September 1990.

[21] 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.

[22] 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.

[23] 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.