# WAM Based Efficient Implementation of Logic Programming Languages—Final Report

Research team:
Adriana Jergová, Luděk Matyska, Petr Mejzlík,
Václav Račanský, David Toman, Jiří Zlatuška

September, 1, 1992

## 1 Introduction

Logic programming languages, and especially Prolog, gained wide acceptance as useful tools for the rapid prototyping in AI and related areas. Much effort has been spent for the development of their efficient implementations. New paradigmas emerged from the original concept of logic programming, most notable the Constraint Logic Programming scheme. In the same time formal description of these languages has been searched for. The dynamic algebras approach enabled to describe the logical operational semantics of the full Prolog (i.e., Prolog with all meta-logical builtin predicates), and this description serves as the basis for Prolog standardization.

This report collects the work done in the IBM Academic Initiative research project devoted to the efficient implementation of logic programming languages. In the first part the presentation of main research results related to the optimal register allocation in the Warren Abstract Machine (WAM) is provided. The second part is devoted to the presentation of a new solver for linear arithmetic constraints, specifically developed for the CLP($\mathcal{R}$) system. In the third part, results related to the operational semantics of full Prolog and WAM are presented.

## 2 Register Allocation in WAM

The Warren Abstract Machine (WAM) is widely recognized as an excellent implementation schema for PROLOG on traditional architectures. Nevertheless many improvements have been suggested for the original WAM. One of the important parts for optimization is the register allocation in process of unification responsible for data assignment and parameter passing.

Most improvements in the register allocation mechanism [12, 17, 18] take advantage of the folloving property of unification. When two structures are to be unified, the order of unifying their arguments doesn't affect the result (i.e., success or failure), but can sufficiently affect the effort expended to prepare the arguments for next predicate call. The number of instructions and registers needed and also the speed of detecting failure depends on the order in which the arguments are unified. That's why the core of register allocation optimization is firmly rooted in finding a proper order.

Investigation of compound term argument ordering (previous work dealed mostly with variable arguments) has brought a new theoretical result. The problem of finding the optimal order of argument processing (optimal register allocation) is at least NP-Complete (cf. [6, 4]). In spite of this discouraging result, appropriate analysis of a clause and sophisticated treatment of its compound arguments enable to achieve the optimal solution in many cases and a usable solution in other cases. Fixing any explicit order (e.g., left-to-right) for parameter passing may cause unnecessary conflicts. Therefore, proper treatment calls for adaptable order—made for a concrete argument placement. This approach necessitates a vigorous analysis of clause before translating it to WAM instruction.

### 2.1 Outline of register allocation algorithm

New strategy introduced by Matyska, Jergová, Toman [4, 6] advances the work of Janssens et al [17], in which optimal order for constant and single variable arguments is proposed, while compound terms are treated in left-to-right order, to better treatment of compound terms. This is achieved by thorough analysis of relation between given and target placements of all compound term components. Obtained information serves scheduling for order of compound term processing that can be interspersed with in-line call processing.

In brief, new proposed strategy consists of following steps

1. processing of head arguments
   (a) known terms (i.e., constants and ground terms);
   (b) multiple top-level occurrences of variables in head;
   (c) non-blocked compound terms;
   (d) in-line calls with prepared arguments;
   (e) processing of half-blocked and blocked arguments;
2. processing of the first chunk body
   (a) not-yet-processed in-line calls;
   (b) non-blocked compound terms;
   (c) half-blocked and blocked terms;
   (d) non-placed top-level variables;
   (e) known terms;
3. sequential processing of remaining chunks
   (a) in-line calls;
   (b) compound terms;
   (c) variables;
   (d) constants.

New strategy enables to produce optimal code in case of nine solvable conflict types (cf. [6]) in comparison with previous works [12, 17, 18] which solve only three types.

# 3 CLP($\mathcal{R}$) Solver Implementation

The expressive power of the Prolog-like languages is not always followed by the same power of the code efficiency. While in the previous section we dealt with the means how to enhance the efficiency of the Prolog implementation, here we focus our attention on modification of the whole Logic Programming paradigm.

All terms in Prolog programs came from the Herbrand universe, i.e., their interpretation is not known to the inference engine. The unfortunate consequence is the unability of the inference engine to use specific knowledge of the domain under investigation. Moreover, the unification, which is the only data-manipulation procedure used, requires *explicit* representation of results, restricting thus the domain to finite objects only.

To overcome both of these restrictions, a new paradigm was proposed—the Constraint Logic Programming, CLP—where the domain is not restricted to the Herbrand universe only and where the unification is replaced by the more general *constraint satisfiability* [15]. The most known instance of the family of possible CLP languages is the CLP($\mathcal{R}$)—Constraint Logic Programming over the Real domain with interpreted ordinary arithmetic operations (e.g., `+`, `-`, `*`, `/`) and functions (e.g., `sin`, `cos`, ...) [16]. Any CLP system can be conceptionally decomposed into two parts—an inference engine and the solver. The CLP($\mathcal{R}$) system uses the modified inference engine of Prolog, while the core of the system is its arithmetic solver.

Generally, the inference engine sends a new constraint to the solver and the solver replies by a one-bit message expressing the solvability of the system of constraints extended by this new one. The internal state of the solver is extended by this new constraint in case when a solvability is assured; when the last constraint added is inconsistent with preceding constraints accumulated by the solver, its internal state remains unchanged. Moreover, the solver must be able to restore any previous state when the inference engine backtracks for reasons outside the solver.

The solver also submits values of variables computed during the solvability testing. In the opposite direction, the inference engine sends values of numerical variables determined without the use of the solver (e.g., by simple assignments).

The solver deals with the following canonical forms of constraints ($x_i, y$ are variables, $a, b, c, d$ are constants):

1. Linear equations and inequations $\vec{a}\vec{x} = c$, $\vec{a}\vec{x} \leq c$, $\vec{a}\vec{x} \geq c$

2. Equations of the type $t = x_1 x_2 \dots x_k$, $t = x/y$, $0 \neq t$

3. Equations of the type $t = \tau(x)$, where $\tau$ is an unary real function

4. Binomial disjunctions of the type $(\vec{a}\vec{x} \leq c \wedge \vec{b}\vec{x} \leq d) \vee (\vec{a}\vec{x} \geq c \wedge \vec{b}\vec{x} \geq d)$

*Linear constraints* of the type 1) are processed by a modified Simplex method separately from constraint of other types. A *nonlinear constraints* is only delayed until the value of all (but one) of its variables will become known.

An algorithm for conversion of complex constraints encountered in CLP($\mathcal{R}$) system to the canonical form presented above (and required by the solver) was developed together with a solver for linear constraints. This solver is based on an *incremental Simplex methods*. The linear solver developed also includes means for correcting rounding error—this is very essential as in theory CLP($\mathcal{R}$) system deals with numbers of infinite precision [5].

# 4 Operational Semantics of Prolog

Since first implementations of Prolog, there has always been problems with giving an exact semantics to the whole language. The semantics of the Horn clause-based programs is natural. But giving the semantics to the built-in predicates—and these predicates turn an academic toy into a real programming language—was always a bottleneck.

The presented approach aspires for combining the advantages of the exact mathematical description with not too a complex machinery usable directly in a Prolog implementation.

## 4.1 Pure Prolog

The basic idea of giving an operational semantics to a programming language is based on constructing an abstract machine performing "instructions" of the language. We will use the notion of *Evolving Algebra* for description of a finite system (an "abstract machine"), evolving in time according to given rules [14].

The main idea, underlying the definition of the Pure Prolog subset of the whole language is based on the depth-first and leftmost-goal-first rules applied to the SLD resolution principle. The depth-first traversal through the particular SLD-tree is then simulated, using a list-algebra for keeping track of the alternatives while each element of this list represents itself a list of goals yet to be solved to prove the initial goal, together with the substitution, computed so far. This universe together with the 'database' universe (which is yet another list: list of clauses in the Prolog program) form the needed data structures. The 'computation' is then defined by a single rule (which indeed just paraphrases the Resolution rule) allowing transition between individual states of the evolving algebra.

## 4.2 Built-in Predicates

To define the behavior of the whole language, we have to extend the Pure Prolog algebra by adding sufficiently clear description of the built-in predicates. When observing the impact of those on the overall Prolog's semantics, we are able to split the set of all built-in predicates to three classes:

1. The 'simple' built-in predicates; members of this class behave like the ordinary (read: user defined) predicates: they eventually succeed (and produce a substitution) or fail. A generic schema for introduction such predicates was presented in [7].

2. The predicates, allowing the updates of the Prolog's database. These predicates influence all subsequent selections of clauses. A 'logical' view of database updates was used for definition of these predicates; and again, a generic schema was provided for this definition in [7].

3. The most problematic built-in predicates are those, dealing with the 'control' of the SLD-tree traversal. In the classical implementations of Prolog, these predicates are represented by the *cut* (`!`) predicate. Note, that all built-in predicates, introduced so far haven't interfere with the resolution rule (they just used a variant of it). The definition of *cut* requires modification to the main structure: we have to assign the cut-related information (scope of which is defined by the bodies of the clauses) to each subgoal (although we will use it if and only if the goal is the *cut* or may contain cut: `call(X)`). For exact definition cf. [10, 7].

   Similar situation occurs in the case of introduction of the constraint-based concepts in the Prolog framework. The information needed for maintaining the constraints also influences the traversal of the SLD-tree (and we usually have to extend the description of Pure Prolog by these structures).

The main concern in this part of the work was not directed towards the individual built-in predicates, but rather towards the generic solutions for wide classes of predicates. Surprisingly, the phenomena of 'being backtrackable' or 'not-being backtrackable' is not discussed here: this problem was shown not to be important on this level of abstraction.

## 4.3 Compilation of Prolog

The main emphasis in this work was given to the description (and correctness proofs) of the compilation techniques of Prolog. In particular, the WAM implementation was chosen here as the most representative way of compiling the Prolog language. The approach is based on stepwise refinement of the Prolog definition (including correctness proofs). During this process the redundant data structures are eliminated, and several optimization techniques are proven to be correct (indexing of clauses, TRO, global and local variables, etc.).

The way from SLD-resolution to WAM then looks like follows:

$$
\begin{array}{ccc}
 & \text{Pure Prolog (+built-in Predicates)} & \\
 & \downarrow & \\
 & \text{Separate rules for selection of clauses and for unification} & \\
\downarrow & & \downarrow \\
\text{Selection theorem, procedures} & & \text{Common environment frames} \\
\downarrow & & \downarrow \\
\text{Introduction of Choicepoints} & & \text{Compiled Unification} \\
\downarrow & & \downarrow \\
\text{Indexing of Procedures} & & \text{Variables, Global data storage} \\
\downarrow & & \downarrow \\
 & \text{Full WAM instruction set and how to get it} & \\
 & \downarrow & \\
 & \text{Code sharing and database updates} & \\
 & \downarrow & \\
 & \text{WAM data structures} &
\end{array}
$$

All the necessary transformations are formally described in [7] except the last one: this may be found in [11] (this paper is more wide: it covers the whole transition from abstract Prolog definition to WAM, but it converts all the structures to the WAM implementation level immediately on the beginning.

That is not necessary and in many cases very tricky (e.g., the manipulation with the variables). Our approach tries to keep as high level of abstraction as possible: there are no pointers, stacks, etc., introduced on the whole way towards WAM—we believe, that this may contribute to better understanding of the concepts hidden in the WAM. Moreover, the correct assignment of variables to the registers of WAM is provided here).

# 5 Conclusion

The research project was concerned with the problem of efficient implementation of logic programming languages. The main attention was focused towards the efficient implementation of Prolog based on Warren Abstract Machine, followed by the search of formal description of its semantics using dynamic algebras. However, looking at the efficiency problem from more general point of view, the Constraint Logic Programming scheme was also considered.
The main results of the research may be summarized as follows:

- New scheme of handling registers, which generates optimal code for a larger multitude of instances than previous works was presented. The original ambition—to generate optimal code for every instance of a clause—was not achieved; instead, the problem of register allocation in WAM proved to be at least NP–Complete.
- A description of the Prolog operational semantics was presented. The framework used offers an efficient, understandable, and precise tool for definition of the full Prolog language. Also, the compilation steps, needed for efficient implementation were formalized using this approach.
- The Constraint Logic Programming scheme was studied and a new constraint solver for CLP($\mathcal{R}$) system was developed.

# List of Publications

[1] Zlatuška, J.: Od logiky k logickému programování. *Informačné Systémy*, **6**, 611–668, 1990.

[2] Zlatuška, J.: Od logického programování k paralelismu. *Informačné Systémy*, **4–5**, 325–350, 1991.

[3] Matyska, L., Toman, D.: Implementační techniky Prologu. *Informačné Systémy*, **1**, 21–59, 1991.

[4] Matyska, L., Jergová, A., Toman, D.: Register Allocation in WAM. In *Proc. 8th International Conference on Logic Programming*, 142–156. MIT Press, Cambridge, MA, 1991.

[5] Mejzlík, P.: Efficient Implementation of CLP($\mathcal{R}$) System. In *Proceedings LOP'92, Rusava*, 187–192. MU Brno, January 1992.

[6] Jergová, A.: New Reordering Register Allocation Strategy for Warren Abstract Machine. *M.A. thesis*, Masaryk University Brno, 1992.

[7] Toman, D.: Operational Semantics of Prolog. Semantics of WAM. *M.A. thesis*, Masaryk University Brno, 1992.

[8] Feit, J., Matyska, L.: Prolog Expert System for Liver Biopsies. *(submitted to AI in Medicine)*, 1992.

# References

[9] Börger, E.: A Logical Operational Semantics of Full Prolog, Part I, *Proc. CSL'89*, Springer LNCS 440, 1990.

[10] Börger, E.: A Logical Operational Semantics of Full Prolog, Part II, *Proc. MFCS'90*, Springer LNCS 452, 1990.

[11] Börger, E., Rosenzweig D.: WAM Algebras—A Mathematical Study of Implementation, part II, *CSE-TR-88-91*, Univ. Michigan Dept. of CSEE, 1991.

[12] Debray, S. K.: Register Allocation in a Prolog Machine, *Symposium on Logic Programming*, 267–275, IEEE, 1986.

[13] Even, S.: *Graph Algorithms*, Computer Science Press, Rockville, Maryland, 1979, 223–224.

[14] Gurievich, Y.: Evolving Algebras. A Tutorial Introduction. *Bull. EATCS 2*, 1991.

[15] Jaffar, J., Lassez, J.-L.: Constraint Logic Programming. In *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, 111–119, München, ACM Press, 1987.

[16] Jaffar, J., Michaylov, S., Stuckey, P. J., Yap, R. H. C.: The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, **14(3)**, 339–395, 1992.

[17] Janssens, G., Demoen, B. and Martën, A.: Improving the Register Allocation in WAM by Reordering Unification, *Proc. 5th Int. Conf. Symp. Logic Programming*, 1388–1402, MIT Press, Cambridge, MA, 1988.

[18] Kurosawa, S. A. K. and Kiriyama, K.: A new Optimization Technique for a Prolog Compiler, *IEEE COMPCON'86*, San Francisco, CA, 241–245, Computer Society Press, Rockville, Maryland, 1986.

[19] Warren, D. H. D.: An Abstract Prolog Instruction Set, *Tech. Report 309*, SRI International, AI Center, Palo Alto, 1983.