In *Proc. Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387-401, Springer LNCS 456, 1990

# Compiling Logic Programs with Equality

Michael Hanus
Fachbereich Informatik, Universität Dortmund
D-4600 Dortmund 50, W. Germany
e-mail: michael@ls5.informatik.uni-dortmund.de

Horn clause logic with equality is an amalgamation of functional and logic programming languages. A sound and complete operational semantics for logic programs with equality is based on resolution to solve literals, and rewriting and narrowing to evaluate functional expressions. This paper proposes a technique for compiling programs with these inference rules into programs of a low-level abstract machine which can be efficiently executed on conventional architectures. The presented approach is based on an extension of the Warren abstract machine (WAM). In our approach pure logic programs without function definitions are compiled in the same way as in the WAM-approach, and for logic programs with function definitions particular instructions are generated for occurrences of functions inside clause bodies. In order to obtain an efficient implementation of functional computations, a stack of occurrences of function symbols in goals is managed by the abstract machine. The compiler generates the necessary instructions for the efficient manipulation of the occurrence stack from the given equational logic program.

## 1 Introduction

During recent years, various attempts have been made to amalgamate functional and logic programming languages (see [DL86] for a collection of proposals). A lot of these integrations are based on Horn clause logic with equality which consists of predicates and Horn clauses for logic programming and functions and equations for functional programming. An operational semantics for logic programs with equality is based on the resolution rule for solving literals where the axiom `X = X` is added to solve equations, and narrowing [Fay79] [Hul80] for evaluating functional expressions. This operational semantics is sound and complete if the equational logic program satisfies the Church-Rosser property [GM86] [Pad88]. Since this general operational semantics is inefficient and leads to many infinite branches in the computation tree (because the narrowing rule can be applied to an arbitrary subterm of the goal), several authors have tried to improve the narrowing procedure. Hullot [Hul80] has shown completeness of *basic narrowing* for canonical term rewriting systems where narrowing is only applied at *basic occurrences*, i.e., occurrences which have not been introduced by substitutions. Fribourg [Fri85] has shown that narrowing can be restricted to exactly one *innermost* position in a narrowing step if all functions are totally defined. Hölldobler [Höl88] has generalized these results: He has shown completeness of *innermost basic narrowing* for canonical conditional term rewriting systems if the innermost reflection rule is added which is needed for incompletely defined functions. Furthermore, he has shown that this calculus remains complete if the goals are simplified by rewriting at basic occurrences. Rewriting between narrowing steps may cut an infinite search space to a finite one. Therefore we want to apply rewriting steps whenever it is possible. As a consequence, flattening of clauses and executing the flattened program by SLD-resolution [BGM87] [BCM89] is not useful. Thus we need a direct implementation of basic innermost narrowing.

Narrowing for functions defined by conditional equations can be combined with resolution for predicates defined by Horn clauses (see [Pad88] and, for a more general result, [Han88b]). Therefore we admit functions as well as predicates in our programs, i.e., predicates need not be represented as Boolean functions. Our operational semantics is based on resolution for predicates, basic innermost narrowing and rewriting for functions, and innermost reflection for incompletely defined functions. Hence our language is a proper superset of pure Prolog and a first-order functional language.

This paper proposes a technique for compiling programs of this language into programs of a low-level abstract machine which can be efficiently executed on conventional architectures. The presented approach is based on an extension of the Warren abstract machine (WAM) [War83]. In our approach logic programs without function definitions are compiled as in the WAM-approach, and for logic programs with function definitions particular commands are generated for occurrences of functions inside clause bodies. In order to have an efficient implementation of functional computations, a stack of occurrences of function symbols in goals is managed by the abstract machine. The compiler generates the necessary instructions for the efficient manipulation of the occurrence stack from the given equational logic program.

This paper is organized as follows. In the next section we introduce our source language which is equipped with a module concept and a type system. The operational semantics of the language is presented in section 3. In the main section 4 the necessary extensions to the WAM are shown. Section 5 outlines important operational properties of our extended abstract machine. We assume familiarity with the basic concepts of the WAM.

## 2 The source language ALF

The source language ALF ("**A**lgebraic **L**ogic **F**unctional language") is based on Horn clause logic with equality. Hence it is possible to use functions inside goals and predicates in conditions of functions. Since we want to have a practical language, ALF has a (simple) module concept and a many-sorted type system. An ALF-program is a set of modules where one main module exists. Goals are proved w.r.t. this main module. A module consists of an interface part (export/import declarations), a declaration part containing the declarations of sorts, constructors, functions and predicates defined by this module, and a body consisting of all program clauses (*relational clauses* defining predicates and *conditional equations* defining functions). A module may be parameterized by sorts which allows the definition of generic modules, e.g., modules for lists, trees etc. Modules are imported from other modules by a *use*-declaration in the interface part. In case of parameterized modules, actual sorts must be supplied for the parameter sorts. The module and type concept is purely syntactic, i.e., a preprocessor translates each ALF-program into an equivalent flat-ALF-program which consists of a list of program clauses, i.e., the intermediate language flat-ALF is the language of single-sorted Horn clause logic with equality. The preprocessor checks the type consistence of the ALF-program and adds the module name to function and predicate symbols whenever it is necessary to resolve name clashes between different modules. Hence the semantics of our language is the same as Horn clause logic with equality, and in the next section we only describe the compilation of single-sorted equational logic programs. However it may be interesting to admit richer type structures (order-sorted [GM86] or polymorphic [Han90]) which influence the operational semantics and must be considered in the compilation process [HV87], but this is out of the scope of this paper.

The syntax of ALF is similar to Prolog [CM87]. We do not present the formal syntax but give an example of an ALF-program. The example consists of a module for natural numbers, a parameterized module for stacks and a main module that uses this two modules:

```
module natMod.
   export 0, s, +.
   sort nat.
   cons 0: nat;  s: nat → nat.
   func +: nat, nat → nat  total.

   N + 0    = N          reduction.
   N + s(M) = s(N + M)   reduction.
   0 + N    = N          onlyreduction.
   s(M) + N = s(M + N)   onlyreduction.
end natMod.

module stackMod(elem).
   export empty, push, top, pop, isEmpty, isNotEmpty.
   sort stack.
   cons empty: stack;
        push: elem, stack → stack.
   func pop: stack → stack;
        top: stack → elem.
   pred isEmpty: stack;
        isNotEmpty: stack.
```

```
    pop(push(E,S)) = S     reduction.
    top(push(E,S)) = E     reduction.
    isEmpty(empty).
    isNotEmpty(push(E,S)).
end stackMod.

module main.    % Compute the sum of a stack of natural numbers
    use natMod;
        stackMod(nat) = natStack.
    func sum: stack → nat  total.

    sum(S) = 0 ←  isEmpty(S).
    sum(S) = top(S) + sum(pop(S)) ←  isNotEmpty(S).
end main.
```

In ALF-programs constructor and function symbols are distinguished: A *constructur* must not be the outermost symbol on the left-hand side of a conditional equation, i.e., constructors are non-reducible function symbols. This distinction is necessary for the notion of *innermost* occurrences. Function symbols may be defined as *total* if this function symbol is reducible for all ground terms of appropriate sorts. The innermost reflection rule need not be applied for total functions. Therefore defining functions as total leads to more efficient programs. For instance, the addition on natural numbers is total in contrast to the functions `pop` and `top` on stacks (e.g., `pop(empty)` is not reducible).

A (conditional) equation may be applied in a narrowing step to evaluate an expression. If an equation should also be used in rewriting steps, it must be marked with the keyword `reduction`. If an equation should *only* be used in rewriting steps, it must be marked with `onlyreduction`. The preprocessor generates two groups of conditional equations in a flat-ALF-program: equations for narrowing steps and equations for rewriting steps. These groups need not be disjoint. If the left-hand side of a narrowing equation or the head of a relational clause contains defined function symbols as arguments, the preprocessor replaces these function symbols by new variables and adds corresponding equations for these variables to the condition. For instance, the equation

```
top(pop(push(E,S))) = top(S).
```

will be replaced by

```
top(S1) = top(S) ←  S1 = pop(push(E,S)).
```

This transformation is necessary for the completeness of narrowing [Han88b].

## 3 Operational Semantics

Since a modularized ALF-program will be translated into a single-sorted equational logic program consisting of lists of relational clauses, conditional equations for narrowing and conditional equations for rewriting, it is sufficient to describe the semantics for such flat-ALF-programs. The declarative semantics is the well-known Horn clause logic with equality [Pad88]. The operational semantics is based on resolution for predicates and innermost basic narrowing for functions with some further rules to cut infinite computations. In order to define the operational semantics we represent a goal by a skeleton and an environment part [Höl88]: the *skeleton* is a goal and the *environment* is a substitution which has to be applied to the goal. The initial goal $G$ is represented by the pair $(G; id)$ where $id$ is the identity substitution. Then the following steps define the operational semantics (if $\pi$ is a position in a term $t$, then $t/\pi$ denotes the subterm of $t$ at position $\pi$ and $t[\pi \leftarrow s]$ denotes the term obtained by replacing the subterm $t/\pi$ by $s$ in $t$):

Let $(L_1, \ldots, L_n; \sigma)$ be a goal ($L_1, \ldots, L_n$ are the skeleton literals and $\sigma$ is the environment).

1. If there is a leftmost-innermost position $\pi$ in the first skeleton literal $L_1$, i.e., the subterm $L_1/\pi$ has a defined function symbol at the top and all argument terms consist of variables and constructors (cf. [Fri85]), then:

   (a) If there is a new variant $l = r \leftarrow C$ of a program clause and $\sigma(L_1/\pi)$ and $l$ are unifiable with mgu $\sigma'$, then

   $$(C, L_1[\pi \leftarrow r], L_2, \ldots, L_n; \sigma' \circ \sigma)$$

   is the next goal derived by *innermost basic narrowing*,

(b) otherwise let $x$ be a new variable and $\sigma'$ be the substitution $\{x \leftarrow \sigma(L_1/\pi)\}$, then

$$(L_1[\pi \leftarrow x], L_2, \ldots, L_n; \sigma' \circ \sigma)$$

is the next goal derived by *innermost reflection* (this corresponds to the elimination of an innermost redex [Höl88]).

2. If there is no innermost position in $L_1$, then:

(a) If $L_1$ is an equation $s = t$ and there is a mgu $\sigma'$ for $\sigma(s)$ and $\sigma(t)$, then

$$(L_2, \ldots, L_n; \sigma' \circ \sigma)$$

is the next goal derived by *reflection*.

(b) If $L_1$ is not an equation and there is a new variant $L \leftarrow C$ of a program clause and $\sigma'$ is a mgu for $\sigma(L_1)$ and $L$, then

$$(C, L_2, \ldots, L_n; \sigma' \circ \sigma)$$

is the next goal derived by *resolution*.

The innermost reflection rule need not be applied to functions declared as *total* because a narrowing step is always applicable for such functions. The attribute *basic* of a narrowing step emphasizes that a narrowing step is only applied at an occurrence of the original program and not at occurrences introduced by substitutions. The restriction to basic occurrences is important for an efficient compilation of narrowing (see below).

This operational semantics corresponds to SLD-resolution if all clauses are flattened [BGM87]. In order to be more efficient (cutting infinite search spaces) than SLD-resolution, rewriting steps have to be applied before narrowing steps: Let $(L_1, \ldots, L_n; \sigma)$ be a goal, $\pi$ be a non-variable position in $L_1$, $l = r \leftarrow C$ be a new variant of a rewrite rule and $\sigma'$ be a substitution. Then

$$(L_1[\pi \leftarrow \sigma'(r)], L_2, \ldots, L_n; \sigma)$$

is the next goal derived by *rewriting* if

1. $\sigma(L_1/\pi) = \sigma'(l)$
2. The goal $(C; \sigma')$ can be derived to the empty goal, i.e., there exists at least one solution for this goal.

A further optimization is *rejection*: The rejection rule immediately fails to prove a goal if the first literal is an equation and the outermost symbols of the two sides are different constructors. In many cases the rejection rule avoids infinite narrowing derivations inside arguments if an equation cannot be unified. A discussion of the advantages of rewriting and rejection in combination with narrowing can be found in [Fri85] and [Höl88].

This operational semantics (innermost basic narrowing, innermost reflection, reflection, resolution, and simplification by rewriting and rejection) is sound and complete if the term rewriting relation generated by the conditional equations is canonical, the condition and the right-hand side of each conditional equation do not contain extra-variables and the set of rewrite rules is equal to the set of narrowing rules [Höl88]. If these restrictions are not satisfied, it may be possible to transform the program into an equivalent program for which this operational semantics is complete. For instance, Bertling and Ganzinger [BG89] have proposed a method to transform conditional equations with extra-variables such that narrowing and reflection will be complete. In the next section we present an efficient implementation of this operational semantics based on an extension of the WAM. Similarly to Prolog, the program clauses in flat-ALF are ordered and the different choices for clauses in a computation step are implemented by a backtracking strategy.

# 4 Compiling flat-ALF-programs

We want to implement the above inference rules for equational logic programs by an extension of the WAM. Therefore we define an abstract machine called **A-WAM** which is designed for the efficient execution of flat-ALF-programs. Since pure Prolog is a subset of flat-ALF, the instruction set of the A-WAM is a superset of the WAM-instructions. Additional data structures and instructions are needed to execute narrowing and rewriting. First we give a short outline of the main modifications in comparison to the WAM. After that we explain more details about the implementation.

## 4.1 Implementing the inference rules for equational logic programs

In the following we show the basic implementation schemes for innermost basic narrowing, innermost reflection, reflection, resolution, rewriting and rejection in the A-WAM.

The **resolution rule** is the same as in Prolog. Therefore the compilation scheme for relational clause heads and predicates in conditions is equal to the WAM.

The **reflection rule** can be implemented as resolution with the axiom `X = X`. An extension to the WAM is not necessary for this rule.

For the **innermost basic narrowing rule** a direct access to the leftmost-innermost subterm of the first literal is necessary. This subterm must be unified with the left-hand side of a conditional equation. The position can be found by a dynamic search through the actual arguments of the literal but such an implementation will be very slow. Fortunately, we can observe that all these positions can be determined by the compiler since we use a *basic* narrowing strategy. For instance, if `p(f(c(g(Y))))` is a literal in the initial goal or in the body of some clause (`f` and `g` are defined functions and `c` is a constructor), then `g(Y)` is an innermost term and `f(c(g(Y)))` will be an innermost term after an application of an innermost reflection step. Therefore the only possible positions for applying the narrowing rule are at the symbols `g` and `f`. It is not necessary to apply narrowing rules inside `Y` if this variable is bound to a complex term while proving the goal because we use a *basic* strategy. Hence the compiler can generate all pointers to the basic narrowing positions in a literal.

Our solution to an efficient implementation of innermost basic narrowing is the following. The A-WAM manages a stack of possible occurrences (positions for narrowing). The compiler generates A-WAM-instructions to push and to pop elements from this occurrence stack. The top element of this stack is always the leftmost-innermost position in the actual literal. The other stack elements are positions in leftmost-innermost order. These occurrences are not innermost terms but they become innermost terms after applying narrowing rules or the innermost reflection rule. For instance, the term `f(c(g(Y)))` is not innermost in the literal `p(f(c(g(Y))))`, but after applying the equation `g(a) = a` to this goal (`a` is a constructor), this (modified) subterm is innermost in the literal `p(f(c(a)))`. Therefore all potentially innermost positions are stored on the occurrence stack. This ensures a fast access to the next innermost position after an application of a narrowing or innermost reflection rule.

The A-WAM compilation scheme for literals in clause bodies is similar to the WAM with the difference that instructions for pushing elements on the occurrence stack are generated. For instance, the literal `p(f(c(g(Y))))` is compiled into the following instruction sequence:

```
put_structure g/1, X1
unify_value Y1          % Y was stored in Y1
put_structure c/1, X2
unify_value X1
put_structure f/1, A1
unify_value X2
set_begin_of_term A1    % store root of narrowing argument
push_occ A1             % store occurrence of f(c(g(Y)))
load_occ X1             % store occurrence of g(Y)
save A1, Y2             % save A1 since arg. registers may be altered in narrowing
narrow 2                % call narrowing, arg. is number of permanent variables
put_value Y2, A1
call p/1, 1
```

For the sake of efficiency the top element of the occurrence stack (the actual leftmost-innermost position) is always stored in a particular register. This element is stored by a `load_occ`-instruction where all other elements are pushed by `push_occ`-instructions. Rewriting can be applied at an arbitrary basic occurrence in the literal. Thus a successful application of a rewrite rule makes the contents of the occurrence stack invalid and a new occurrence stack must be created. For this purpose it is necessary to store the root of the actual narrowing argument in a particular register which is done by the instruction `set_begin_of_term`. Rebuilding the occurrence stack after successful rewriting is started from this register. The `narrow`-instruction loads the arguments of the structure at the actual occurrence into the argument registers `Ai` and tries to apply rewrite rules (see below). Afterwards it jumps to the narrowing code for the function stored at the actual occurrence.

Note that all these narrowing-specific instructions are only generated if a defined function symbol occurs in the argument of the literal. Otherwise the compilation scheme is identical to the WAM-scheme for Prolog, i.e., there is no overhead for narrowing.

A *narrowing rule* of the form $f(t_1, \ldots, t_n) = r \leftarrow C$ is compiled in the following way: First, get-instructions are generated for the arguments $t_1, \ldots, t_n$ (identical to the WAM), followed by the code for the body $C$ and instructions for storing the right-hand side at the actual occurrence. This replacement is implemented by put-instructions with the suffix _occ. For instance, the narrowing rule g(a) = a $\leftarrow$ is translated into the instructions

```
get_constant a, A1     % unify argument with constant a
put_const_occ a        % store constant a at actual occurrence
pop_occ                % pop the next innermost occurrence from the stack
proceed_occ            % proceed with rewriting and narrowing at new occurrence
```

The replacement of the subterm at the actual occurrence by the put..._occ-instructions must be stored in the trail stack in order to restore the original terms in case of backtracking. Thus the A-WAM-trail contains unbound variables *and* terms. This causes no problem in the implementation because only the outermost symbol of the replaced term must be stored.

Hence the compilation scheme for narrowing rules is similar to relational clauses, i.e., the indexing structure for the first argument is identical to the WAM. The additional instructions for the right-hand side at the end of the body is the only difference to the WAM.

The **innermost reflection rule** moves the actual occurrence from the skeleton into the environment part. Therefore we have to distinguish in compound terms the skeleton part from the environment part (this is also necessary for rebuilding the occurrence stack after a succesful application of a rewrite rule). Unfortunately, in the original WAM it is impossible to see whether a compound term originally occurred in a clause body or was created by unification. For instance, consider the goals X=g(Y), p(f(X)) and p(f(g(Y))). If these goals are proved and the predicate p is called, then the argument of p has the same representation on the heap in both cases. But in the skeleton/environment representation the subterm g(Y) belongs to the environment in the first case and to the skeleton in the second case.

Thus the distinction between the skeleton and the environment part of a compound term must be made explicit in the A-WAM. For this purpose an additional tag field (Boolean value) is added to all terms: All instructions corresponding to terms occurring in program clauses (get-, put-, unify-instructions) mark the terms as belonging to the skeleton part whereas the (implicitly called) unification procedure mark a variable which is bound to another term as belonging to the environment part. With this small modification the implementation of the innermost reflection rule is very simple: The term at the actual occurrence must be marked as "environment" and the A-WAM-instructions pop_occ and proceed_occ have to be executed.

**Rewriting** is executed before a narrowing step. It can be viewed as narrowing with the difference that goal variables are not modified by rewriting (the substitution is only applied to the rewrite rule). Therefore rewrite rules are compiled like narrowing rules with the following modifications:

- The A-WAM contains two additional registers R and HR which point to the local stack and heap, respectively. Initially, these registers point to the bottom of the stacks. Before rewriting is called, R is set to the top of the local stack and HR is set to the top of the heap. The WAM-instruction trail, which is called if a variable is bound to a term in the unification procedure, is extended as follows: If the variable is in the local stack before address R or if the variable is in the heap before address HR, then the instruction fail is executed. Hence binding a goal variable while executing a rewrite rule causes a failure which has the consequence that the next rewrite rule is tried. Therefore each individual rewrite rule is compiled identically to a narrowing rule. The different behaviour (matching instead of unification) is implicitly controlled by the registers R and HR. This has the advantage that no additional instructions for the translation of rewrite rules are needed. It is also possible to compile rewrite rules with particular get- and unify-instructions for matching, but this needs more work for the implementation of the abstract machine.
- Since rewriting does not change the actual goal before the right-hand side of the rule is inserted into the goal, it is not necessary to generate full backtrack points in the indexing scheme for rewrite rules. It is sufficient to store the address of the next alternative rewrite rule. For this purpose the A-WAM contains particular indexing instructions for rewrite rules, but the scheme for generating these instructions is identical to the WAM.
- The body (condition) of a rewrite rule may contain additional variables which do not occur on the rule's left-hand side. Therefore a solution for the body must be found by resolution and narrowing,

i.e., the body is compiled like the body of a program clause but with the following difference: Since we assume confluence, it is sufficient to find *one* solution for the body. Thus a backtrack point is generated before the body of the rewrite rule is proved and it is deleted after the (possibly unsuccessful) proof.

- If a rewrite rule is applicable and the right-hand side of the rule is inserted into the goal, the occurrence stack for narrowing becomes invalid because a rewrite rule can be applied at an arbitrary (basic) occurrence. Therefore the occurrence stack is marked as *invalid* in case of a successful application of a rewrite rule. If the rewriting process is terminated (no more rewrite rules are applicable) and the occurrence stack is invalid, then a new occurrence stack must be computed for the subsequent narrowing process. This computation, started at the root of the actual argument term (this was stored by the instruction `set_begin_of_term`, see above), pushes all occurrences of function symbols in the *skeleton* part of the term onto the occurrence stack. This is the only reason why the skeleton/environment information of terms must be stored at run time (see above).
- If no rewrite rule is applicable for a function symbol at the actual occurrence, rewriting has to be tried at the next (outermost) position. Hence the last alternative in a sequence of rewrite rules for a function symbol is always: Pop an occurrence from the occurrence stack and proceed with rewriting at this new occurrence. Therefore rewriting does not fail if no rule is applicable (in contrast to narrowing). The rewriting process is terminated if the occurrence stack is empty. In this case a new occurrence stack is computed (if a rewrite rule has been applied) or the old occurrence stack is restored (if no rewrite rule could be applied), and then computation proceeds by applying narrowing rules.

**Rejection** is a possibility for cutting infinite unsuccessful narrowing derivations. For instance, let `c1` and `c2` be constructors and

`f(c1(X)) = c1(f(X))` $\leftarrow$

be a narrowing rule. In order to prove the literal `c1(f(Y)) = c2(Z)`, innermost basic narrowing produces an infinite derivation. But this literal cannot be true since the constructors at outermost positions are different. This will be recognized by the rejection rule. The A-WAM-instruction `reject A1,A2` causes a failure if the outermost symbols of the two terms in registers `A1` and `A2` are different constructors. This instruction is generated for an equation $t_1 = t_2$ in a goal in front of the narrowing instructions if the outermost symbols of $t_1$ and $t_2$ are variables or constructors. Hence the equation `c1(f(Y)) = Z` in a condition will be compiled as follows:

```
put_structure f/1, X1
unify_value Y1          % Y was stored in Y1
put_structure c1/1, A1
unify_value X1
put_value Y2, A2        % Z was stored in Y2
reject A1, A2           % apply rejection
set_begin_of_term A1    % store root of narrowing argument
load_occ X1
save A1, Y3             % save A1 in Y3
narrow 3                % call narrowing for function f
put_value Y3, A1
put_value Y2, A2
get_value A1, A2        % equivalent to  call =/2
```

## 4.2 The abstract machine A-WAM

In the last section we have given an outline of the necessary A-WAM-extensions to implement the inference rules for flat-ALF. Since we are dealing with *conditional* narrowing and rewrite rules, the actual implementation of the A-WAM needs more complex data structures:

- If a conditional narrowing rule is applied and defined function symbols occur in the condition, then these functions must be evaluated by narrowing again. Hence the occurrences of these function symbols must be pushed onto the occurrence stack, but these new occurrences must be distinguished from the old occurrences stored on the occurrence stack before proving the condition. Because of this recursive structure of the narrowing process, the occurrence stack must have a recursive structure too. The A-WAM contains a *list of occurrence stacks*. The last element of this list is always the occurrence stack belonging to the actual argument (narrowing instructions are generated for each argument of a literal which contains defined function symbols).

- Since narrowing and rewriting may be called to prove conditions of rewrite rules, there is a recursive structure in rewriting too. Therefore the new A-WAM-registers `R` and `HR` for rewriting must be set before starting rewriting (see above) and restored to the old values after finishing rewriting and before starting narrowing.

There are more details which have to be considered in the implementation. These will be explained below. In the following we present the registers, data structures and commands of the A-WAM. Since we assume familiarity with the basic concepts of the WAM, we only explain the differences to the WAM. A detailed description of the A-WAM together with a formal specification of the operational semantics of all A-WAM-instructions in the style of [Han88a] can be found in [PIL90].

### 4.2.1 Data areas and registers of the A-WAM

The main data areas of the A-WAM are the following (there is also a system stack or push-down list for the recursive implementation of the unification procedure):

**Code area:** Contains the compiled flat-ALF-program. Note that it must be also stored whether a functor symbol is a constructor or a defined function (necessary for rebuilding the occurrence stack).

**Local stack:** Contains environments and backtrack points.

**Heap:** Contains terms. A term cell has an additional tag (one bit) which shows whether this term belongs to a skeleton or an environment. Skeleton terms are terms occurring in a program clause and environment terms are terms created by unification (or an application of the innermost reflection rule). Hence environment terms belong to substitutions. The `get`-, `put`- and `unify`-instructions set the "skeleton"-tag in created terms whereas the unification procedure and the instruction `reflection` set the "environment"-tag.

**Trail:** Contains references to variables that have been bound during unification and old values and references to function symbols on the heap that have been replaced by an application of a narrowing or rewrite rule. These values must be restored on backtracking. If a subterm in the heap is replaced by a narrowing/rewrite rule, then only the outermost function symbol is replaced by a reference to the new term. Therefore it is sufficient to store the term cell containing this function symbol on the trail. It is not necessary to store the complete subterm on the trail (this may be false if a garbage collector reorganizes the heap!).

**Occurrence stack:** Contains references to subterms in the heap where narrowing or rewriting can be applied. It is organized as a list of stacks of occurrences. Only the last element is manipulated by A-WAM-instructions. The last element of this list (the *actual occurrence stack*) contains a reference to a previous list element and all occurrences of function symbols in the skeleton of the actual argument where narrowing and rewriting should be applied. The occurrences are placed in leftmost-innermost order on this stack where the top element is always stored in the A-WAM-register `AO`, i.e., the stack is empty iff `AO` is undefined. A new actual occurrence stack is created by the instruction `allocate_occ` if the condition of a narrowing or rewrite rule is executed.

The organization of terms is identical to the WAM, i.e., $n$-ary structures are stored on the heap in $n+1$ consecutive cells where the first cell contains the functor and the next $n$ cells are the arguments. Narrowing and rewriting is always executed on heap terms. Hence a 0-ary defined function symbol must also be stored as a term on the heap (in contrast to the WAM where constants are separately treated).

The registers of the A-WAM are summarized in figure 1. The second group of registers are new in comparison to the WAM. The registers `R` and `HR` are used to implement rewriting (see description of rewriting implementation in the previous section). Register `OV` indicates the successful application of a rewrite rule. Register `OP` points to the top of the actual occurrence stack and `OM` points to the bottom of the actual occurrence stack, i.e., the list of occurrence stacks is implemented with backward pointers to previous elements. The actual occurrence, i.e., the top element of the actual occurrence stack, is always stored in register `AO`. Register `TS` contains the reference to the root of the actual argument term before narrowing is called. It will be set by the instruction `set_begin_of_term`.

| Name | Function |
|---|---|
| P | program pointer |
| CP | continuation program pointer |
| E | last environment |
| B | last backtrack point |
| H | top of heap |
| TR | top of trail |
| S | structure pointer |
| RW | read/write mode for unify instructions |
| A1, A2, ... | argument registers |
| X1, X2, ... | temporary variables |
| R | rewrite pointer (to the local stack) |
| HR | heap rewrite pointer (to the heap) |
| OP | top of actual occurrence stack |
| OM | bottom of actual occurrence stack |
| AO | actual occurrence |
| TS | term start (root of the actual argument term) |
| OV | Is the actual occurrence stack valid? Will be set to false in case of rewriting. |
| RFP | rewrite fail pointer (to the code area) |
| TFP | try rewrite fail pointer (to the code area) |

Figure 1: The registers of the A-WAM

As mentioned in the previous section, a backtrack point need not be generated if a chain of rewrite rules for a function symbol is executed because rewriting does not affect the actual goal. It is only necessary to store the address of the next alternative rewrite rule. Therefore the `try/retry/trust`-instructions for the indexing scheme for rewrite rules are prefixed by `r_`. The particular `r_try/r_retry/r_trust`-instructions do not create or manipulate a backtrack point but set and change the values of the registers `RFP` and `TFP`. For instance, `r_try_me_else L` sets register `RFP` to `L` and `r_try L` sets register `TFP` to the address of the next instruction. The instruction `fail`, which will be executed on failure, examines the values of `RFP` and `TFP`: If `TFP` is defined, then the program pointer `P` is set to `TFP`. If `TFP` is undefined and `RFP` is defined, then `P` is set to `RFP`. If both are undefined, the computation state is reset to the last backtrack point.

The values of these registers and the contents of the data areas characterize the computation state of the A-WAM. In order to reset to an old state in case of failure (backtracking), a backtrack point contains the following information: contents of the registers `P`, `CP`, `E`, `B`, `H`, `TR`, `R`, `HR`, `OP`, `OM`, `AO`, `TS`, `OV`, `RFP`, `TFP`, where the component `P` contains the address of the next alternative clause, and copies of the first $n$ argument registers (if it is a backtrack point for an $n$-ary predicate or function) and the actual occurrence stack. It is sufficient to save the actual occurrence stack and not the whole list of occurrence stacks in the backtrack point because the A-WAM-instruction `deallocate_occ` deletes the last list element only if a new backtrack point has not been created (see below; this is due to the same reason why environments need not be saved in backtrack points in the WAM).

### 4.2.2 Instructions of the A-WAM

The instruction set of the A-WAM is a superset of the WAM-instructions. Therefore we only describe the additional A-WAM-instructions.

`load_occ Ai`: Set the actual occurrence register `AO` to `Ai` which must contain a reference to a structure on the heap with a defined function symbol at the top. This instruction may also be used with a temporary variable `Xi` instead of `Ai`.

`push_occ Ai`: Push the reference to the structure `Ai` onto the actual occurrence stack. This instruction may also be used with a temporary variable `Xi` or the actual occurrence register `AO` instead of `Ai`.

`pop_occ`: Pop an element from the actual occurrence stack and store the value in register `AO`. If the actual occurrence stack is empty, set `AO` to "undefined". This instruction is used for a narrowing/rewrite rule which has no occurrences of defined function symbols on the right-hand side. For instance, the narrowing rule `f(a) = b` ← is translated into the instruction sequence

```
get_constant a, A1
put_const_occ b        % store constant b at actual occurrence
pop_occ                % pop the next innermost occurrence from the stack
proceed_occ            % proceed with narrowing at new occurrence
```

`set_begin_of_term Ai`: Set the term start register `TS` to the structure referenced by `Ai`. If the occurrence stack must be rebuilt because of an application of a rewrite rule, the creation of the new occurrence stack starts at term position `TS`. If the outermost symbol of this term is a constructor, it is possible to use this instruction with a temporary variable `Xi`. In this case `Xi` is a reference to a subterm of `Ai` which contains all basic occurrences of defined function symbols in `Ai`.

`proceed_occ`: This instruction terminates a narrowing rule. If register `AO` is undefined (= no more occurrences), program pointer `P` is set to `CP`, otherwise the narrowing rules for the function at occurrence `AO` are executed after loading the argument registers with the components of the structure at occurrence `AO`.

`r_proceed_occ`: This instruction terminates a rewrite rule. Registers `RFP` and `TFP` are set to "fail" (no alternative rewrite rule must be applied because of confluence). If register `AO` is undefined, program pointer `P` is set to `CP`, otherwise the rewrite rules for the function at occurrence `AO` are executed after loading the argument registers with the components of the structure at occurrence `AO`.

`narrow N`: This instruction starts rewriting and narrowing after loading the occurrences of the actual argument term. It is a macro and equivalent to the following sequence of A-WAM-instructions:

```
call_rewriting AO, N
rebuild_occ_stack
call_narrowing AO, N
```

These three instructions are only used for the implementation of `narrow`. `call_rewriting` corresponds to the WAM-instruction `call`, i.e., `N` is the number of permanent variables wich are still in use in the actual environment. `call_rewriting AO,N` creates a copy of the actual occurrence stack, saves the values of registers `R` and `HR` in the environment, loads the components of the structure at position `AO` into the argument registers, sets registers `OV`, `R` and `HR` to true, top of local stack and top of heap, respectively, and calls the rewrite rules for the function at occurrence `AO`, i.e., `CP` is set to the address of the following instruction (`rebuild_occ_stack`) and `P` is set to the address of the rewrite rules.

Note that in the A-WAM rewriting always returns to the instruction `rebuild_occ_stack` because the last alternative in a sequence of rewrite rules is always `r_proceed_occ` and *not* `trust_me_else fail` (see below). `rebuild_occ_stack` deletes the last element from the list of occurrence stacks (that was the new occurrence stack created by `call_rewriting`) and sets `R` and `HR` to their old values stored in the environment. Then it replaces the actual occurrence stack by a new occurrence stack for the term at position `TS`, if `OV` is false (i.e., a rewrite rule has been applied), otherwise nothing is done.

`call_narrowing AO,N` loads the components of the structure at position `AO` into the argument registers and calls the narrowing rules for the function at occurrence `AO`.

`save Ai, Yj`: Since narrowing may recursively call narrowing and resolution for proving conditions, the contents of the argument registers are altered in the narrowing process. Therefore some registers must be saved which can be done by this instruction. It is equivalent to `get_variable Yj, Ai`. Thus this instruction is only added for readability reasons. For instance, the goal literal `p(f(X),g(Y))` (`f` and `g` are defined functions) is compiled into

```
put_structure f/1, A1
unify_value Y1          % X was stored in Y1
set_begin_of_term A1
load_occ A1
save A1, Y3             % save A1 in Y3
narrow 4                % narrow the first argument
put_structure g/1, A2
unify_value Y2          % Y was stored in Y2
set_begin_of_term A2
load_occ A2
```

```
save A2, Y4             % save A2 in Y4
narrow 4                % narrow the second argument
put_value Y3, A1
put_value Y4, A2
call p/2, 2             % call predicate p/2
```

`reject Ai, Aj`: This instruction causes a failure if the outermost symbols of the terms referenced by registers `Ai` and `Aj` are different constructors. Otherwise, no action is taken.

`reflection`: This is the last alternative in a sequence of narrowing rules for a function which is not total. It follows the last `trust_me_else fail` in the indexing scheme for these narrowing rules and implements the innermost reflection rule: The term at the actual occurrence `AO` is marked as "environment" and the A-WAM-instruction sequence

```
pop_occ
proceed_occ
```

is executed.

`put_value_occ Xi`: In order to insert the right-hand side of a narrowing or rewrite rule into a goal, the A-WAM contains a set of `put..._occ`-instructions. For each `put`-instruction of the WAM (except for `put_unsafe_value`) there is a corresponding A-WAM-instruction `put..._occ` which substitutes the argument at the actual occurrence `AO` and stores the old value at occurrence `AO` on the trail. For instance, the narrowing rule `f(X) = g(h(X))` ← is compiled into

```
get_variable X1, A1
put_structure h/1, X2
unify_value X1
put_structure_occ g/1
unify_value X2
push_occ AO
load_occ X2
proceed_occ
```

`put_structure_occ` puts a new structure on the heap and replaces the heap cell at address `AO` by a reference to this new structure.

`invalid_os`: This instruction sets register `OV` to false and indicates the successful application of a rewrite rule. It is executed before the `r_proceed_occ`-instruction in a rewrite rule.

`r_try_me_else L`: The indexing instructions for rewrite rules are prefixed by `r_`. These different indexing instructions are executed since it is not necessary to create a backtrack point (see discussion above). The second difference in the indexing scheme is the last alternative in the sequence of rewrite rules: Instead of `trust_me_else fail` it is always the instruction sequence `pop_occ/r_proceed_occ` (try rewriting at the next innermost occurrence).

This instruction sets register `RFP` to code address `L`.

`r_retry_me_else L`: Identical to `r_try_me_else L`.

`r_try L`: Set register `TFP` to the address of the following instruction and program pointer `P` to code address `L`.

`r_retry L`: Identical to `r_try L`.

`r_trust L`: Set register `TFP` to "fail" and program pointer `P` to code address `L`.

`allocate_occ`: This instruction saves the occurrences in `AO` and `TS` onto the occurrence stack and adds a new (empty) actual occurrence stack to the list of all occurrence stacks. It is used before a condition in a narrowing or rewrite rule will be proved. For instance, the narrowing rule `f(X) = b ←p(g(X))` is compiled into the following A-WAM-instructions:

```
allocate
get_variable  X1, A1
allocate_occ           % allocate a new occurrence stack to prove the condition
put_structure g/1, A1
unify_local_value X1
set_begin_of_term A1
load_occ A1
save A1, Y1            % save A1 in Y1
```

```
narrow 1               % narrow the term g(X)
put_value Y1, A1
call p/1, 0
deallocate_occ         % deallocate the occurrence stack for the condition
put_const_occ b        % store constant b at actual occurrence
pop_occ                % pop the next innermost occurrence from the stack
deallocate
proceed_occ
```

`deallocate_occ`: This instruction deletes the last element from the list of occurrence stacks and loads registers `AO` and `TS` from the previous occurrence stack. If a backtrack point has been created after the corresponding `allocate_occ`-instruction (i.e., if `B.OM` $\geq$ `OM`), it is not allowed to alter previous elements of the occurrence stack list since only the actual occurrence stack has been saved into the backtrack point. In this case `deallocate_occ` creates a copy of the previous occurrence stack and adds this copy to the list of occurrence stacks.

`l_try_me_else L, N`: The condition of a rewrite rule must be proved by resolution and narrowing but it is sufficient to compute *one* solution for the body (cf. section 4.1). Since we do not generate backtrack points in the indexing scheme for rewrite rules, a backtrack point must be generated for the proof of the condition of a rewrite rule. Hence the translation scheme for conditional rewrite rules of the form $l$ = $r$ $\leftarrow c$ is the following:

```
   allocate
   <get-instructions for l>
   allocate_occ                 % create new occurrence stack
   l_try_me_else L,n            % create new backtrack point for condition
                                % n is number of perm. variables
   <instructions for condition c>
   trust_me_else fail           % delete backtrack point for condition
   deallocate_occ               % delete occurrence stack for condition
   <put..._occ-instructions for r>
   <occurrence-stack-instructions for r>
   deallocate
   invalid_os
   r_proceed_occ
L: trust_me_else fail           % delete backtrack point for condition
   deallocate_occ               % delete occurrence stack for condition
   deallocate
   fail                         % try next rewrite rule
```

The instruction `l_try_me_else L,N` creates a backtrack point similarly to `try_me_else L`. The difference is the additional argument `N` which contains the size of the actual environment. The WAM accesses the size of the actual environment via the continuation pointer `CP` which is not possible in this context.

Now we have described the data structures and additional instructions of the A-WAM. Moreover, we have shown the compilation scheme for narrowing and rewrite rules by several examples. Some difficulties in the compilation scheme are due to the fact that we consider *conditional* rules. If we restrict flat-ALF programs to unconditional rules (as done in other approaches for combining resolution and narrowing [Yam87]), a lot of optimizations are possible in our implementation. For instance, there is no recursive structure in the narrowing process and hence only one occurrence stack is needed. Therefore the data structure for the occurrence stack is simpler and the instructions `allocate_occ` and `deallocate_occ` are superfluous.

### 4.2.3 Treatment of variables

In the WAM variables are classified as temporary or permanent. Permanent variables require space on the local stack where temporary variables can be stored in registers. A variable is permanent if it occurs in two different literals in a goal. This is due to the fact that the contents of registers for temporary variables may be altered during the execution of the WAM-instruction `call`. In the A-WAM this can also be the case *inside* literals if narrowing is applied. Therefore the classification of variables in the A-WAM is different from the WAM. We do not want to go into details but explain the differences by examples. For instance, let

```
p(X) ← q(f(X),g(X))
```

be a program clause. The WAM does not generate an environment during the execution of this clause because the value of `X` can be stored in a temporary register. If `f` and `g` are defined functions, then narrowing must be executed for the two arguments of the literal in the body. Hence the value of `X` cannot be stored in a temporary register and an environment is necessary to store `X`, i.e., `X` is classified as permanent in the A-WAM. The environment is also necessary to store the references to the argument terms `f(X)` and `g(X)`. The above clause is compiled into the instructions

```
allocate
get_variable  Y1, A1    % store X in permanent variable Y1
put_structure f/1, A1
unify_local_value Y1
set_begin_of_term A1
load_occ A1
save A1, Y2             % save A1 in permanent variable Y2
narrow 2                % narrow the term f(X)
put_structure g/1, A2
unify_value Y1
set_begin_of_term A2
load_occ A2
save A2, Y1             % save A2 in permanent variable Y1
narrow 2                % narrow the term g(X)
put_value Y2, A1
put_value Y1, A2
deallocate
execute q/2
```

The variable classification is based on the division of the sequence of A-WAM-instructions into chunks (cf. [Deb86]). A *chunk* is an instruction sequence that does not contain the instructions `call` and `narrow`. A variable is *permanent* if it occurs in two different chunks.

The WAM puts the arguments of a literal from left to right into the registers. It has been observed that better code can be produced if this fixed order is replaced by an adaptable order [JDM88]. This is also the case for narrowing in the A-WAM. In order to generate more efficient code, arguments with occurrences of defined function symbols should be treated *before* other arguments of a literal. For instance, generating code for the literal `p(c(X),f(X))` (`c` is a constructor and `f` is a defined function) in strict left-to-right order yields

```
put_structure c/1, A1
unify_value Y1          % X was stored in Y1
put_structure f/1, A2
unify_value Y1
set_begin_of_term A2
load_occ A2
save A1, Y2             % save A1 in permanent variable Y2
save A2, Y3             % save A2 in permanent variable Y3
narrow 3                % narrow the term f(X)
put_value Y2, A1
put_value Y3, A2
call p/2, 1
```

But the save instruction for the first argument is unnecessary if the order of putting arguments is changed:

```
put_structure f/1, A2
unify_value Y1
set_begin_of_term A2
load_occ A2
save A2, Y2             % save A2 in permanent variable Y2
narrow 2                % narrow the term f(X)
put_value Y2, A2
put_structure c/1, A1
unify_value Y1
call p/2, 1
```

## 5 Properties of the A-WAM

At the moment we cannot present detailed results on the efficiency of our proposed abstract machine since the implementation is not finished yet (a bytecode emulator for the A-WAM written in C is under implementation). But we can discuss the properties of the A-WAM for the execution of particular classes of equational logic programs. This yields some insight into the expected behaviour of the A-WAM.

- For *logic programs* without occurrences of defined function symbols there is only a small overhead in the A-WAM in comparison to the WAM. The A-WAM-code for such programs is identical to the WAM-code. The only overhead is due to the fact that backtrack points in the A-WAM are bigger than in the WAM because the additional registers (`RFP`, `TFP`, ...) are saved in backtrack points.
- The other extreme is the class of *functional programs* where only ground terms have to be evaluated. Such flat-ALF-program only consists of unconditional rewrite rules. The compiler generates the necessary instructions for loading the occurrences of function symbols onto the occurrence stack, i.e., no run-time search must be made to find the next occurrence in the term where a rewrite rule should be applied. The ground term (functional expression) will be evaluated by one `narrow`-call since the term is reduced after rewriting and does not contain any occurrences of function symbols.
- Pure functional and pure logic programs will be efficiently executed by the A-WAM. We expect that programs containing a mixture of functional and logic parts will also be efficient in execution time. The most interesting class is the set of *programs with unconditional narrowing/rewrite rules* where narrowing and rewriting cannot be recursively called. In this case the list of occurrence stacks contains only one element at run time and the narrowing and rewrite rules are efficiently applied by the A-WAM.
- Another interesting case is the class of programs where the *set of rewrite rules is a superset of the narrowing rules* (this is not required in flat-ALF but usually true, see, e.g., SLOG [Fri85]). Since argument terms are simplified by rewriting before narrowing is applied, function calls with ground arguments are automatically evaluated by rewriting and not by narrowing. This is more efficient because rewriting is a deterministic process (no backtrack points are created). Hence in most practical cases our combined rewriting/narrowing implementation will be more efficient than an implementation of narrowing by flattening terms and applying SLD-resolution [BGM87].

## 6 Conclusions

We have presented an approach to compile logic programs with equality. The operational semantics is based on the rules for resolution, reflection, innermost basic narrowing, innermost reflection, rewriting and rejection. Related work includes [KLMR90] where an implementation of the language BABEL is presented. BABEL is a combination of a higher-order functional and a first-order logic language. The operational semantics uses reflection and innermost narrowing without rewriting. BABEL is implemented by a functional graph reduction machine which is extended to perform unification and backtracking. Josephson and Dershowitz [JD89] have also proposed an implementation technique for narrowing and rewriting, but they handle unification and control at the interpretive level. In our approach equational logic programs are compiled into code for the abstract machine A-WAM which is an extension of the Warren abstract machine. One important extension is the management of a stack of occurrences of function symbols in goals. Since a basic narrowing strategy is used, the compiler can generate particular instructions for the management of the occurrence stack.

We have mentioned several optimizations for the A-WAM, but there are further possibilities for optimizing the A-WAM which will be investigated in the future. For instance, rewrite rules can be more efficiently executed if the compiler generates code for fast pattern matching (see, e.g., [Heu87]). If at least one rewrite rule is applied, then the occurrence stack is marked as "invalid" and a new occurrence stack must be computed by traversing the term before narrowing starts. But there are a lot of cases where the application of a rewrite rule changes the occurrence stack only in a few positions. The compiler may generate particular code for these changes instead of the instruction `invalid_os`. The detailed analysis of such cases and a better management of the occurrence stack is also a topic for future research.

**Acknowledgements:** The author is grateful to Renate Schäfers and the members of the project group "PILS" for many discussions on the design of the A-WAM.

## References


[BCM89] P.G. Bosco, C. Cecchi, and C. Moiso. An extension of WAM for K-LEAF: a WAM-based compilation of conditional narrowing. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 318–333. MIT Press, 1989.

[BG89] H. Bertling and H. Ganzinger. Completion-Time Optimization of Rewrite-Time Goal Solving. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 45–58. Springer LNCS 355, 1989.

[BGM87] P.G. Bosco, E. Giovannetti, and C. Moiso. Refined strategies for semantic unification. In *Proc. of the TAPSOFT '87*, pp. 276–290. Springer LNCS 250, 1987.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer, third rev. and ext. edition, 1987.

[Deb86] S.K. Debray. Register Allocation in a Prolog Machine. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 267–275, Salt Lake City, 1986.

[DL86] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.

[Fay79] M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.

[Fri85] L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.

[GM86] J.A. Goguen and J. Meseguer. Eqlog: Equality, Types, and Generic Modules for Logic Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming, Functions, Relations, and Equations*, pp. 295–363. Prentice Hall, 1986.

[Han88a] M. Hanus. Formal Specification of a Prolog Compiler. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, pp. 273–282, Orléans, 1988. Springer LNCS 348.

[Han88b] M. Hanus. *Horn Clause Specifications with Polymorphic Types.* Dissertation, FB Informatik, Univ. Dortmund, 1988.

[Han90] M. Hanus. A Functional and Logic Language with Polymorphic Types. In *Proc. Int. Symposium on Design and Implementation of Symbolic Computation Systems*, pp. 215–224. Springer LNCS 429, 1990.

[Heu87] T. Heuillard. Compiling conditional rewriting systems. In *Proc. 1st Int. Workshop on Conditional Term Rewriting Systems*, pp. 111–128. Springer LNCS 308, 1987.

[Höl88] S. Hölldobler. From Paramodulation to Narrowing. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 327–342, 1988.

[Hul80] J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.

[HV87] M. Huber and I. Varsek. Extended Prolog with Order-Sorted Resolution. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 34–43, San Francisco, 1987.

[JD89] A. Josephson and N. Dershowitz. An Implementation of Narrowing. *Journal of Logic Programming (6)*, pp. 57–77, 1989.

[JDM88] G. Janssens, B. Demoen, and A. Marien. Improving the Register Allocation in WAM by Reordering Unification. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 1388–1402. MIT Press, 1988.

[KLMR90] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based Implementation of a Functional Logic Language. In *Proc. ESOP 90*, pp. 271–290. Springer LNCS 432, 1990.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1988.

[PIL90] Projektgruppe PILS. Zwischenbericht der Projektgruppe PILS. Univ. Dortmund, 1990.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.

[Yam87] A. Yamamoto. A Theoretical Combination of SLD-Resolution and Narrowing. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 470–487. MIT Press, 1987.