in *Proc. International Workshop on Processing Declarative Knowledge*,
pp. 344-365, Springer LNAI 567, 1991

Efficient Implementation of Narrowing and Rewriting

*Michael Hanus**
Technische Fakultät, Universität Bielefeld
W-4800 Bielefeld 1, Germany
e-mail: hanus@techfak.uni-bielefeld.de

Abstract

We present an efficient implementation method for a language that amalgamates functional and logic programming styles. The operational semantics of the language consists of resolution to solve predicates and narrowing and rewriting to evaluate functional expressions. The implementation is based on an extension of the Warren Abstract Machine (WAM). This extension causes no overhead for pure logic programs and allows the execution of functional programs by narrowing and rewriting with the same efficiency as their relational equivalents. Moreover, there are many cases where functional programs are more efficiently executed than their relational equivalents.

1 Introduction

During the last years a lot of approaches have been proposed in order to amalgamate functional and logic programming languages [7] [1]. Such integrations have several advantages:

1. Functional and logic programming styles can be used in one language.
2. It extends logic programming by allowing nested expressions, i.e., it is not necessary to flatten complex expressions as in Prolog.
3. It extends functional programming by solving equations between functional expressions.
4. It allows the programmer to specify functional dependencies between data. This information can be used for a more efficient implementation.
5. Large parts of logic programs are functional computations. In an integrated language these parts are defined as functions which can be more efficiently executed than their relational equivalents.

Point 1 is a matter of taste, and point 2 is no real argument since nested expressions can be flattened by a preprocessor [4]. But the last three arguments show that an integration of functional and logic languages yields a proper extension of each of these language types.

*on leave from Fachbereich Informatik, Universität Dortmund, W-4600 Dortmund 50

For instance, consider the following logic program for the addition of natural numbers where numbers are represented as terms constructed by `0` and `s`:

```
add(0, N, N) ←
add(N, 0, N) ←
add(s(M), N, s(L))  ←  add(M, N, L)
add(N, s(M), s(L))  ←  add(N, M, L)
```

If the literal `add(0,0,Z)` should be proved, then a backtrack point (also called "choice point" in [33]) must be generated since there are two alternative proofs yielding the result {`Z/0`} in both cases. The equivalent functional program is

```
0 + N   =  N
N + 0   =  N
s(M) + N   =  s(M + N)
N + s(M)   =  s(N + M)
```

The equation `0 + 0 = Z` can be solved in a determinstic way by applying one of the first two equations to the left-hand side. A creation of a backtrack point is unnecessary since "`+`" is a function which has a unique result. One could argue that a Prolog compiler can also optimize the code for the predicate `add` but this requires some sort of mode information which is not available if the equation `X + Y = s(0)` should also be solved (where `X` and `Y` are free variables). A genuine integration of functional and logic languages permits such goals and has no fixed modes for the application of functions. In this paper we present such a language together with an implementation which avoids the creation of backtrack points if it is not necessary.

Another advantage of an integrated functional and logic language is the reduction of the search space by functional computations: Fribourg [8] has given examples for terminating functional-logic programs where equivalent Prolog programs do not terminate or need more computation steps. This aspect is also covered by our language and we will discuss this point in more detail in subsequent sections.

A lot of the proposed integrations of functional and logic languages are based on Horn clause logic with equality [31] which offers predicates defined by Horn clauses for logic programming and functions defined by (conditional) equations for functional programming. The declarative semantics is the well-known Horn clause logic [25] with the restriction that the equality predicate is always interpreted as identity. The operational semantics is based on *resolution* for predicates (like in logic languages) and *rewriting* for functions (like in functional languages). Since it is also required to *solve* equations between functional expressions, a new inference rules is added: *narrowing* is a combination of unification and rewriting, i.e., a subterm of the goal is unified with the left-hand side of an equation such that the instantiated subterm can be rewritten with that equation and the unifier is applied to the whole goal. This general strategy has been refined by Hölldobler [19] to the *innermost basic narrowing* strategy where exactly one possible subterm must be narrowed in a computation step. This strategy has the same efficiency as SLD-resolution, but Hölldobler has shown that goals can also be simplified by rewriting before a narrowing step is performed. This loses no solutions and is more efficient than Prolog's computation strategy.

However, the discussion about the better efficiency of functional computations is only relevant if there is a good implementation technique for narrowing and rewriting. Up to now most of the proposed systems are implemented by an interpreter which can not compete with present Prolog implementations based on a compilational approach [33]. Merely [3], [24], [27] and [26] contain approaches to compile (lazy) narrowing rules into code of an abstract machine, but the integration of rewriting is not addressed in these papers. This paper presents an implementation technique for a functional and logic programming language with the following properties:

- The operational semantics of the language is based on resolution, narrowing and rewriting.
- Pure logic programs without functions are compiled in the same way as in Prolog systems based on the Warren Abstract Machine (WAM) [33], i.e., there is no overhead because of the functional part.
- There is a particular technique to deal with occurrences (references to subterms) where the next narrowing or rewrite rule can be applied. Thus functional programs are executed by narrowing and rewriting at least with almost the same efficiency as their relational equivalents by resolution. Moreover, there are large classes of programs where the functional versions are more efficiently executed by narrowing and rewriting than the relational versions by resolution.
- There are no modes for the execution of functions. Similarly to logic programming, functions can be evaluated with ground or non-ground terms at each argument position. However, functions are evaluated by determinstic rewriting if the arguments are ground, and in other cases (non-deterministic) narrowing is applied. This is automatically decided at run time, i.e., user annotations are not necessary to specify where rewriting or narrowing should be applied.

Our implementation is based on an extension of the WAM [33] and therefore we assume familiarity with the basic concepts of this machine. The techniques presented in this paper are based on a previous proposal [12] but have the following basic differences: The current implementation simplifies the goal by rewriting before *each* narrowing step (normalized narrowing) whereas in [12] rewriting is only applied before an entire narrowing derivation is computed. Furthermore, we present new techniques for the management of occurrences which speeds up the execution time up to 30% and saves up to 40% of the heap space because of a delayed copying of function symbols onto the heap.

This paper is organized as follows. In the next section we define the operational semantics implemented by our system. The techniques for the efficient management of occurrences are shown in section 3 and details about our abstract machine are presented in section 4. For the sake of simplicity we introduce the basic implementation techniques only for unconditional equations. The necessary extensions to deal with conditional equations are shown in section 5. Section 6 shows some results of our implementation.

2 The implemented operational semantics

We have mentioned in the introduction that our approach to integrate functional and logic programming languages is based on Horn clause logic with equality (see [31] for details) which extends pure Horn logic by allowing user definitions for the binary predicate "=". Since Horn clause logic with equality interprets this predicate as identity, we can define functions by this feature. For instance, the following clauses define a function `isort` on lists which produces a sorted permutation of the argument list by the insertion sort method (we use the Prolog notation for lists [6] and we assume that the ordering predicates `=<` and `>` are defined elsewhere):

```
isort([])     = []
isort([E|L])  = insert(E,isort(L))

insert(E,[])     = [E]
insert(E,[F|L])  = [E,F|L]            ←  E =< F
insert(E,[F|L])  = [F|insert(E,L)]    ←  E > F
```

Clauses for the predicate "=" are also called **conditional equations**. If the condition is empty, we call it also **unconditional equation**. Note that this program is neither a valid K-LEAF program [3] (since the left-hand side of the two conditional equations are identical) nor a valid BABEL program (since the conditions of the two conditional equations are "propositional satisfiable" [28]). But it is allowed in our language since we only require the confluence of the term rewriting relation generated by the (conditional) equations (the `insert` equation system is confluent since "`E =< F` *and* `E > F`" is unsatisfiable).

Our source language ALF ("**A**lgebraic **L**ogic **F**unctional language") consists of Horn clauses for user-defined predicates and equations for user-defined functions (the left-hand sides can also be non-linear in contrast to K-LEAF and BABEL). Furthermore, ALF has a (parametrized) module system and a many-sorted type structure. Since these features have no influence on the execution of ALF-programs, we omit the details here and refer the interested reader to [12] and [15]. An important aspect of the language is the distinction between **constructors** and **functions**. A constructor must not be the outermost symbol of the left-hand side of a conditional equation, i.e., constructor terms are always irreducible. This distinction is specified by the user [12] and necessary for the notion of *innermost occurrences* [8].

The *declarative semantics* of ALF is the well-known Horn clause logic with equality as to be found in [31]. As mentioned in the introduction, the *operational semantics* of ALF is based on resolution for predicates and rewriting and innermost basic narrowing for functions. In order to give a precise definition of the operational semantics, we represent a goal by a skeleton and an environment part [19]: the *skeleton* is a goal composed of terms and literals occurring in the original program, and the *environment* is a substitution which has to be applied to the goal in order to obtain the actual goal. The initial goal G is represented by the pair $< G; id >$ where id is the identity substitution. We define the following inference rules to derive a new goal from a given one (if π is a position in a term t, then t/π denotes the subterm of t at position π and $t[\pi \leftarrow s]$ denotes the term obtained by replacing the subterm t/π by s in t): Let $< L_1, \ldots, L_n \ ; \ \sigma >$ be a given goal ($L_1, \ldots, L_n$ are the skeleton literals and σ is the environment).

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

$$< L_2, \ldots, L_n \ ; \ \sigma' \circ \sigma >$$

is derived by **reflection**.

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

$$< C, L_2, \ldots, L_n \; ; \; \sigma' \circ \sigma >$$

is derived by **resolution**.

3. Let π be a leftmost-innermost position in the first skeleton literal L_1, i.e., the subterm L_1/π has a defined function symbol at the top and all argument terms consist of variables and constructors (cf. [8]).
   (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 σ', then the goal

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

   is derived by **innermost basic narrowing**.
   (b) If x is a new variable and σ' is the substitution $\{x \leftarrow \sigma(L_1/\pi)\}$, then the goal

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

   is derived by **innermost reflection** (this corresponds to the elimination of an innermost redex [19]).

4. If π is a non-variable position in L_1, $l = r \leftarrow C$ is a new variant of a program clause and σ' is a substitution with $\sigma(L_1/\pi) = \sigma'(l)$ and the goal $< C \; ; \; \sigma' >$ can be derived to the empty goal without instantiating any variables from $\sigma(L_1)$, then the goal

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

is derived by **rewriting** (thus rewriting is only applied to the first literal, but this is no restriction since a conjunction like L_1, L_2, L_3 can also be written as an equation $and(L_1, and(L_2, L_3)) = true$).

5. If L_1 is an equation and the two sides have different constructors at the same outermost position (a position not belonging to arguments of functions), then the whole goal is **rejected**, i.e., the proof fails.

The complete operational semantics of ALF is shown in figure 1. The innermost reflection rule must only be applied to *partial* functions, i.e., functions which are not reducible for all ground terms of appropriate sorts [19]. 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 [21]. The restriction to basic occurrences is important for an efficient implementation of narrowing and rewriting (see below). The rewriting rule has the disadvantage that terms from the environment part can be moved to the skeleton part, but it has been shown that such terms can be safely moved back to the environment part [30]. Therefore environment terms are never moved to the skeleton part in our implementation.

```
Start:  Apply rewriting as long as possible (from innermost to outermost positions).
        If the goal is not rejected then:
Narrow: If possible, apply the innermost basic narrowing rule and go to Start.
        If possible, apply the innermost reflection rule and goto Narrow.
        If the first literal of the goal is an equation
        then:  If possible, apply the reflection rule and go to Start.
        else:  If possible, apply the resolution rule and go to Start.
        Otherwise: fail (and try an alternative proof)
```

Figure 1: Operational semantics of ALF

This operational semantics is sound and complete if the term rewriting relation generated by the conditional equations is canonical and the condition and the right-hand side of each conditional equation do not contain *extra-variables* [19]. 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 [2] have proposed a method to transform conditional equations with extra-variables such that narrowing and reflection will be complete. Therefore we allow extra-variables in conditional equations. For instance, our operational semantics is complete for the following set of equations defining quicksort, which can be proved by the CEC completion system [2] (we omit the definition of `=<` and `>`):

```
conc([],L)    = L
conc([E|R],L) = [E|conc(R,L)]

split(E,[])    = ([],[])
split(E,[F|L]) = ([F|L1],L2)  ←  E >  F, split(E,L) = (L1,L2)
split(E,[F|L]) = (L1,[F|L2])  ←  E =< F, split(E,L) = (L1,L2)

qsort([])    = []
qsort([E|L]) = conc(qsort(L1),[E|qsort(L2)])  ←  split(E,L) = (L1,L2)
```

('`,`' is defined as an infix operator for building pairs of lists). Note that this is not a valid K-LEAF or BABEL program since the extra-variables `L1` and `L2` occur in the right-hand side of the defining equations. In order to avoid the extra-variables one has to replace the last equation by

```
qsort([E|L]) = conc(qsort(split1(E,L)),[E|qsort(split2(E,L))])
```

and redefine the split function. This solution is less efficient (because the list `L` must be processed twice) and simplification orderings fail to prove the termination of the rewrite relation [2]. These drawbacks may be accepted, but there are other examples where the use of extra-variables cannot be avoided with simple transformations. The function `last` computes the last element of a given list. It can be explicitly defined or, if `conc` is defined as above, by the simple conditional equation

```
last(L) = E  ←  conc(L1,[E]) = L
```

In this case `last(L)` is evaluated by searching the right instantiations of `L1` and `E` (note that there is at most one solution if `L` is given). The use of extra-variables gives us the full power of logic programming inside functional programming. Hence ALF allows extra-variables in conditional equations. If such a conditional equation is applied in a rewrite step, only the first solution to the extra-variables is considered. This is sufficient because all equations are required to be confluent.

It is also possible to specify additional equational clauses which are only used for rewriting. For instance, Fribourg [8] has shown that the addition of inductive axioms for rewriting is useful to reduce the search space. In this case the proved goals are valid with respect to the least Herbrand model but may be invalid in the class of all models. Therefore an ALF-program consists of three groups of clauses: relational clauses which define all predicates except "`=`", conditional equations used for narrowing and conditional equations used for rewriting (Fribourg's SLOG language allows only unconditional equations for rewriting). Usually, all conditional equations in an ALF-program are used for narrowing and rewriting, but the programmer can specify that some equations should only be applied for narrowing or rewriting, respectively. For instance, the inductive axiom `rev(rev(L)) = L` can be used for rewriting to reduce the search space (the function `rev` reverses all elements in a list). To use it as a narrowing rule makes no sense since this would expand the search space.

Similarly to Prolog, the program clauses in ALF are ordered and the different choices for clauses in a computation step are implemented by a backtracking strategy. Note that backtracking is only necessary in the resolution and narrowing rule but not in rewriting since simplification by rewriting produces unique terms independently of the chosen clauses (because of the confluence of the term rewriting relation). Therefore rewriting is a *deterministic* process and the simplification of a goal by rewriting before a narrowing step means that in ALF deterministic computations are performed whenever possible and nondeterministic computations (narrowing/resolution) are only used when it is not avoidable. The Andorra computation model [17] is related to ALF's operational semantics. But in contrast to the Andorra model the rewriting mechanism of ALF yields deterministic computations also when more than one clause matches (see `add` example in section 1) and may delete goals with infinite or nondeterministic computations. E.g., if `X*0 = 0` is a defining equation for the function `*`, then a term like t `* 0` will be simplified to `0`, i.e., the entire subterm t will be deleted. This is important if t contains unevaluated functions with variable arguments.

In order to demonstrate the improved efficiency of this operational semantics in comparison to Prolog's computation strategy, consider the following equations for the concatenation function on lists:

```
conc([],L)     = L
conc([E|R],L)  = [E|conc(R,L)]
```

If `a` and `b` are constructors, then the goal

```
conc(conc([a|V],W),Y) = [b|Z]
```

is simplified by rewriting to the goal

```
[a|conc(conc(V,W),Y)] = [b|Z]
```

which is immediately rejected since `a` and `b` are different constructors. The equivalent Prolog goal

```
append([a|V],W,L), append(L,Y,[b|Z])
```

causes an infinite loop for any order of literals and clauses [29]. More details about the advantages of rewriting and rejection in combination with narrowing can be found in [8] and [19].

3 The management of occurrences

In this section we want to show the basic ideas to implement the operational semantics of ALF in an efficient way. Since Prolog's operational semantics is included in our language, we have decided to extend the WAM in order to implement the new aspects of ALF. The resolution and reflection rule can be directly implemented in the WAM since there is no difference to Prolog. The implementation of rejection is also obvious (note the similarity between unification and rejection). Therefore we discuss the implementation of narrowing and rewriting in more detail. For the sake of simplicity we consider only unconditional equations in this section. The necessary extensions to deal with conditional equations are shown in section 5.

The WAM stores terms on the heap. In order to obtain an efficient implementation of narrowing and rewriting, we need a fast access to the subterm where the next narrowing or rewrite rule should be applied. A dynamic search through the argument term of the current literal is too expensive for this purpose. But since we use an innermost basic strategy, all relevant occurrences of subterms can be determined at compile time. For instance, consider the clause

```
fac(s(N))  =  fac(N) * s(N)
```

If this equation is applied to reduce a term of the form `fac(A)`, then we know by the innermost basic strategy that the argument term `A` does not contain any occurrences of functions belonging to the skeleton part. Therefore we replace the term `fac(A)` by the right-hand side `fac(N) * s(N)` (after unifying `A` and `s(N)`) and then we reduce the subterm `fac(N)`. If this subterm is completely reduced to a term `T`, then the term `T * s(N)` is the next term where an equation must be applied.

Hence we introduce a new data structure called **occurrence stack**. An **occurrence** is a reference to a term on the heap. The occurrence stack contains all references to subterms of an argument of the current literal where narrowing and rewrite rules could be applied (in innermost order, i.e., the reference to the innermost term is always the top element). For instance, if `p(f(c(g(X))))` is the current skeleton literal, `f` and `g` are functions and `c` a constructor, then the occurrence stack contains a reference to the subterm `f(c(g(X)))` and a reference to the subterm `g(X)` at the top. Now it is easy to see that the compiler can generate all necessary instructions for the manipulation of the occurrence stack. For instance, the right-hand side of the above equation for `fac` can be translated into

<replace the term at the current occurrence by `fac(N) * s(N)`>
<push a reference to the subterm `fac(N)` onto the occurrence stack>

The right-hand side contains two functions, therefore an additional occurrence must be pushed onto the occurrence stack. If the right-hand side does not contain a function symbol (i.e., only constructors and variables), then an element must be popped from the occurrence stack. For instance, the right-hand side of the clause `fac(0) = s(0)` is translated into

<replace the term at the current occurrence by `s(0)`>
<pop a reference from the occurrence stack>

This has the effect that the computation proceeds at the next innermost occurrence stored on the occurrence stack.

Before a literal is proved by resolution, all arguments must be evaluated by rewriting and narrowing. Therefore the arguments must be stored on the heap and the occurrence stack is initialized with the appropriate references. For instance, the literal `p(f(c(g(X))))` is translated into

<write the term `f(c(g(X)))` onto the heap>
<push reference to the term `f(c(g(X)))` onto the occurrence stack>
<push reference to the term `g(X)` onto the occurrence stack>
<start rewriting and narrowing>

Now a new problem occurs. Rewriting tries to simplify the current argument term by applying rewrite rules from innermost to outermost positions in the term. If a subterm cannot be rewritten, then the next innermost position is tried, i.e., an element is popped from the occurrence stack. This is necessary as the following example shows: If the only equations for `f` and `g` are

```
f(Z) = 0
g(0) = 0
```

then the term `g(X)` cannot be rewritten (only narrowing could be applied), but the term `f(c(g(X)))` can be simplified to `0`.

Hence the rewriting process pops all elements from the occurrence stack and therefore the stack is empty when rewriting is finished and a narrowing rule should be applied. In order to avoid a dynamic search for the appropriate innermost occurrence, we introduce a second stack for storing the deleted occurrences (in [12] all occurrences are stored on one stack and therefore more time is needed to recompute the occurrences in case of successful rewriting). This stack (called **copy occurrence stack**) contains all occurrences if rewriting is finished and the original occurrence stack is empty. Thus the occurrence stack can be reinstalled by a simple block-copy operation. There is only one case where this method cannot be applied (but fortunately this case rarely occurs): If a rewrite rule deletes a subterm because there are variables on the left-hand side which do not occur on the right-hand side (as in the clause `f(Z) = 0`) and the copy occurrence stack is not empty, then some occurrences must be deleted from the copy occurrence stack. Since this is expensive or requires additional information in the data structures, we have implemented a simple solution: In this case the copy occurrence stack is marked as "invalid" which has the consequence that a new occurrence stack for the current argument term is computed before a narrowing rule is applied.

The presented technique for the management of occurrences has the advantage that the next relevant subterm for rewriting or narrowing can be found in constant time and a dynamic search for reducible subterms is not necessary. As a consequence we will see in section 6 that functional programs are executed by rewriting and narrowing with almost the same efficiency as their relational equivalents by resolution.

4 Details of the abstract machine

After discussing the basic ideas of the implementation in the previous section, we can present more details about our abstract machine. The abstract machine for the efficient execution of ALF-programs, called **A-WAM**, is an extension of the WAM. Hence the main data areas of the A-WAM are the *code area* containing the compiled code of the ALF-program, the *local stack* containing environments and backtrack points, the *heap* containing terms constructed at run time, the *trail* containing variables bound during unification, and the *occurrence stack* and the *copy occurrence stack* as described in the last section. In contrast to the WAM, the trail contains also the contents of heap cells which were replaced by an application of a rewrite or narrowing rule, and the terms in the heap have an additional tag indicating whether they belong to the skeleton or environment part of the goal. This is necessary because the basic occurrences must be recomputed in some cases (cf. previous section).

The A-WAM has several additional registers and instructions for the implementation of rewriting and narrowing. A description of these can be found in the appendix. In this section we describe the A-WAM by selected examples.

An equational clause l `=` r is always translated into the following scheme:

<unify or match the left-hand side l with the current subterm>
<replace the current subterm by the right-hand side r>
<update the occurrence stack (delete or add occurrences)>
<proceed with rewriting/narrowing at new innermost occurrence>

The current subterm is referenced by the top element of the occurrence stack. Therefore this top element is always stored in the particular A-WAM-register `AO`, i.e., the occurrence stack is empty iff `AO` is undefined. Similarly to the WAM, the arguments of a n-ary predicate or function are passed through the argument registers `A1`,…,`A`n. Hence the `get`-instructions of the WAM can be used to unify the left-hand side of an equation. If this equation is used as a rewrite rule, then the left-hand side must be *matched* with the current subterm, i.e., variables in the current subterm must not be bound. One possible implementation of this behaviour is the introduction of additional registers `R` and `HR` which point to the local stack and heap, respectively. Before rewriting is called, `R` and `HR` are set to the top of the local stack and the top of the heap, respectively. If a variable is bound to a term in the unification procedure, the WAM-instruction `trail` is called. Now we modify the instruction `trail` such that this instruction causes a `fail` if the variable to be bound is stored in the local stack before address `R` or in the heap before address `HR`. With this small modification we need no additional instructions for matching but can use the given `get`-instructions.

In order to replace the current subterm (pointed by register `AO`) by a new term (the right-hand side of an equation), the A-WAM contains a duplicated set of `put`-instructions with the suffix `_occ` which replace the current subterm in the heap by another term. For instance, the instruction `put_const_occ C` writes the constant `C` on the heap at address `A0` and stores the old value at occurrence `A0` on the trail, and the instruction `put_struct_occ f/n` puts a new structure on the top of the heap, replaces the heap cell at address `A0` by a reference to this new structure and trails the old value at `A0`.

The A-WAM has three instructions for the manipulation of the occurrence stack: `load_occ R` sets register `A0` to the value in register `R`, `push_occ R` pushes the value in `R` onto the occurrence stack, and `pop_occ` pops an element from the occurrence stack and stores its value in register `A0`.

Now we can show the **translation of rewrite rules** (remember that each equation can be used as a rewrite rule as well as a narrowing rule). Consider the two rewrite rules for the function `rev`:

```
rev([])     = []
rev([E|R]) = conc(rev(R),[E])
```

The first rewrite rule is translated into

```
get_nil A1
put_nil_occ
pop_occ
execute_rewriting A0
```

The first instruction matches the current argument stored in `A1` with the constant `[]` representing the empty list. If this is successful, the second instruction replaces the current subterm by the empty list. Now rewriting must proceed at the next innermost occurrence. Therefore an element is popped from the occurrence stack by the third instruction and the last instruction loads the argument registers with the components of the new current subterm and jumps to the code of the appropriate rewrite rules. The second rewrite rule for `rev` is translated into

```
get_list A1              % match A1 with [E|R]
unify_variable X4
unify_variable A1
put_list X3              % write [E] on the heap
unify_value X4
unify_nil
put_struct_occ conc/2    % replace current subterm by conc(_,[E])
unify_variable X2
unify_value X3
push_occ A0              % update occurrence stack
load_occ X2
execute_rewriting rev/1  % jump to the rewrite rules for rev/1
```

Note that the subterm `rev(R)` is not written on the heap because this is the next innermost subterm where a rewrite rule should be applied. Therefore a new unbound variable is stored instead of this subterm and the argument register `A1` is set to the value of `R` (this is different from the implementation presented in [12]). If a rewrite rule can be applied to rev(R), then the variable is overwritten by the right-hand side of the applied rule. Otherwise rewriting must be applied at the next innermost position. Thus the last alternative of the sequence of rewrite rules for rev is always the code sequence

```
put_function_occ rev/1
copy_pop_occ
execute_rewriting A0
```

The first instruction puts the structure rev/1 with the value of argument register A1 onto the heap at address A0 if this heap cell contains an unbound variable. The second instruction pops an element from the occurrence stack and pushes it onto the copy occurrence stack (as described in section 3). The last instruction proceeds with rewriting at the new occurrence.

We have also mentioned in section 3 that the copy occurrence stack may become invalid if the rewrite rule deletes a subterm in an argument. Therefore the instruction invalid_os must be generated if a rewrite rule is applied where the right-hand side does not contain all variables of the left-hand side. For instance, the rewrite rule f(Z) = 0 is translated into

```
put_const_occ 0
pop_occ
invalid_os
execute_rewriting A0
```

The instruction invalid_os marks the copy occurrence stack as invalid if it is not empty. In this case the occurrence stack must be recomputed before a narrowing rule is applied.

The translation of narrowing rules is similarly to rewrite rules. The only difference is that after an application of a narrowing rule we do not proceed with another narrowing rule but must perform rewriting and rejection first. Hence the narrowing rule conc([],L) = L is translated into

```
get_nil A1
put_value_occ A2
pop_occ
call_rewriting A0
rebuild_occ_stack
reject
execute_narrowing A0
```

The instruction call_rewriting A0 sets the registers R and HR and jumps to the rewrite code of the function at occurrence A0. When the whole term is simplified by rewriting, execution continues with the instruction rebuild_occ_stack which moves the copy occurrence stack to the occurrence stack (if it is valid) or recomputes the occurrence stack. reject performs the rejection rule if the current literal is an equation, and execute_narrowing A0 tries to apply a narrowing rule at the occurrence A0.

The **indexing scheme** for narrowing rules is similar to the WAM-translation scheme for predicates, i.e., all narrowing rules for a function are connected with a chain of try_me_else-, retry_me_else- and trust_me_else_fail-instructions. Moreover, instructions for indexing on the first argument are generated. For rewrite rules the same

```
conc/2: r_try_me_else b2
        switch_on_term c1a,c1,c2,fail
c1a:    r_try_me_else c2a       % Clause: conc([],L) = L
c1:     get_nil A1
        put_value_occ A2
        pop_occ
        execute_rewriting A0
c2a:    r_trust_me_else_fail    % Clause: conc([E|R],L) = [E|conc(R,L)]
c2:     get_list A1
        unify_variable X4
        unify_variable A1
        put_list_occ
        unify_value X4
        unify_variable X3
        load_occ X3
        execute_rewriting conc/2
b2:     put_function_occ conc/2  % go to next innermost position
        copy_pop_occ
        execute_rewriting A0
```

Figure 2: A-WAM-code of the rewrite rules for `conc`

scheme is generated, but all indexing instructions are replaced by "rewrite indexing instructions" which are prefixed by `r_`. This is due to the fact that rewriting is a deterministic process and rewrite rules do not change the current literal before the right-hand side is inserted. Therefore the A-WAM contains two registers `RFP1` and `RFP2` which contains the address of an alternative rewrite rule (*two* registers are necessary because there may exist two backtrack points for one clause due to the indexing scheme [33]). These registers are set by the `r_try`...–instructions instead of creating a backtrack point. The instruction `fail`, which is executed on failure, considers the values of `RFP1` and `RFP2`: If one of these registers is defined (not equal to "fail"), `P` is set to the last one, otherwise the computation state is reset to the last backtrack point. The instruction `execute_rewriting`, which is always executed at the end of a rewrite rule, sets `RFP1` and `RFP2` to "fail" which implements the determinstic behaviour of rewriting. The complete translation of the rewrite rules for the function `conc` is shown in figure 2.

If an argument term of a literal in a goal contains function symbols, then this argument term must be evaluated by rewriting and narrowing before the resolution rule is applied to the literal. Therefore instructions for initializing the occurrence stack and rewriting and narrowing instructions must be inserted in such literals. For instance, the literal `p(fac(s(0)))` in a goal is translated into

```
put_structure s/1, X2     % store argument term  fac(s(0))
unify_constant 0
put_structure fac/1, Y2
```

```
unify_value X2
set_begin_of_term Y2      % store root of argument term
load_occ Y2               % initialize occurrence stack
call_rewriting A0, 2
rebuild_occ_stack
call_narrowing A0, 2
put_value Y2, A1          % restore argument term
call p/1, 1
```

The first 4 instructions are identical to the WAM-code with the only difference that the root of the argument term is not stored in register `A1` but in the permanent variable `Y2`. This is necessary since argument registers are altered during rewriting and narrowing. The A-WAM has a register `TS` which contains the root of the argument currently evaluated by rewriting and narrowing. This register is used when the occurrence stack must be recomputed after rewriting if the copy occurrence stack has been marked as invalid. Therefore `TS` is initialized by the instruction `set_begin_of_term` with the appropriate value. The second arguments of `call_rewriting` and `call_narrowing` are the number of permanent variables which are still in use in the current environment (similar to the WAM-instruction `call`).

Now we have shown how ALF-programs (with unconditional equations) can be translated into A-WAM-code. Note that the A-WAM-code for functions is very similar to the WAM-code for the equivalent predicate (e.g., compare the code for the functions `conc` and `rev` with the WAM-code for the naive reverse program). Thus functional programs are executed with the same efficiency as their relational equivalents. Moreover, backtrack points are not generated for rewriting and therefore many functional programs are more efficiently executed. Before we present concrete results of our implementation, we will show how conditional equations are implemented in our framework.

5 Conditional equations

Conditional equations causes a new problem since the condition must be proved before the equation could be applied. To prove the condition rewriting and narrowing may be recursively used. Hence the current occurrence stack must be saved before the condition is proved and restored after the proof of the condition. To implement this recursive structure of the narrowing process, the A-WAM contains not only one occurrence stack but a list (or stack) of occurrence stacks. The last element of this list is always the current occurrence stack belonging to the argument term currently evaluated by narrowing or rewriting. Since rewriting may have a recursive structure too, the copy occurrence stack is also a list of stacks where the last element is the current copy occurrence stack.

The A-WAM has two instructions to manipulate the list of occurrence stacks. The instruction `allocate_occ` adds a new (empty) occurrence stack to the list of occurrence stacks. It is used before a condition in a narrowing or rewrite rule will be proved. At the end of the condition the instruction `deallocate_occ` is executed which deletes the last element from the list of occurrence stacks. If a backtrack point has been created during the proof of the condition, then the last occurrence stack is not deleted since it is needed on backtracking. Hence a backtrack point freezes the current occurrence stack (note the similarity to environments and the `allocate`/`deallocate`-instructions in the WAM).

Consider the conditional equation `f(N) = 0` $\leftarrow$ `odd(g(N))`. It is translated as a narrowing rule into the following code:

```
allocate
get_variable X2, A1
allocate_occ              % create a new occ. stack for the condition
put_structure g/1, Y1     % create argument term g(N)
unify_value X2
set_begin_of_term Y1
load_occ Y1
call_rewriting A0, 1      % rewrite argument term g(N)
rebuild_occ_stack
call_narrowing A0, 1      % narrow argument term g(N)
put_value Y1, A1
call odd/1, 1
deallocate_occ            % delete occurrence stack for the condition
put_const_occ 0
deallocate
pop_occ
call_rewriting A0         % proceed with rewriting at next occurrence
rebuild_occ_stack
reject
execute_narrowing A0      % proceed with narrowing
```

The compilation scheme for conditional rewrite rules is a little bit more complicated because it is sufficient to compute *one* solution for the condition (rewriting is a deterministic process). Thus backtrack points generated during the proof of the condition can be safely deleted. The second problem is that the indexing scheme for rewrite rules (`r_try`...-instructions) does not generate backtrack points. Therefore a backtrack point must be created at the beginning of the condition. Hence a conditional rewrite rule of the form l = $r \leftarrow c$ is translated into

```
    allocate
    <get-instructions for l>
    l_try_me_else L,A,N         % create new backtrack point for condition
    allocate_occ                % create new occurrence stack
    <instructions for condition c>
    deallocate_occ              % delete occurrence stack for condition
    l_trust_me_else fail        % delete backtrack points for condition
    <put..._occ-instructions for r>
    <occurrence-stack-instructions for r>
    deallocate
    invalid_os                  % if necessary
    execute_rewriting A0
L:  l_trust_me_else fail        % delete backtrack points for condition
```

```
deallocate
fail                          % try next rewrite rule
```

The instruction `l_try_me_else L,A,N` creates a backtrack point similarly to `try_me_else L,A` (`A` is the number of argument registers to be saved) and stores the address of the last backtrack point in the environment (usually in the permanent variable `Y1`). The additional argument `N` contains the size of the current environment (the WAM accesses the size of the current environment via the continuation pointer `CP` which is not possible in this context). The instruction `l_trust_me_else fail` deletes all backtrack points generated during the proof of the condition, i.e., the pointer to the last backtrack point (WAM-register `B`) is set to `Y1` (the backtrack point before the condition).

6 Results

The current implementation consists of two parts: a compiler written in Prolog which translates ALF-programs into a compact bytecode representing A-WAM-programs, and a bytecode emulator for the A-WAM written in C. The details of the implementation together with a complete formal specification of the A-WAM in the style of [10] can be found in [16]. In this section we present some results of our implementation.

First of all, let us remark that pure logic programs without equations are compiled identical to the WAM, i.e., there is no overhead because of the functional part of our language (only backtrack points are a little bit bigger because of the additional registers of the A-WAM). Although the current implementation is a first prototype and not very fast[1], it is interesting to see the relation between execution times for functional programs and their relational equivalents, because this shows the relationship between our implementation of narrowing and rewriting and the current techniques for logic programming.

The first example is the classical (but controversial) naive reverse benchmark. The relational version is executed by resolution, the functional version by narrowing and rewriting. The following table shows the time for reversing a list of 30 elements in both directions (all benchmarks were executed on a Sun4):

Naive reverse		
Initial goal:	`rev([···]) = L`	`rev(L) = [···]`
Relational "naive reverse":	18 msec	190 msec
Functional "naive reverse":	19 msec	210 msec

The next example demonstrates one advantage of integrating functions into logic programming languages. In the first section we have shown clauses for defining the predicate `add` and the function `+`. We have stated that the functional computation is more efficient than the relational because no backtrack points must be generated for evaluating the function by rewriting. The following table shows that this is true in our implementation

[1]The performance of our current implementation is approximately 38 KLips on a Sun4 for the naive reverse benchmark; for typical logic programming examples with backtracking, like the permutation sort program (see below), our implementation is approximately 6-7 times slower than a commercial Prolog system (Quintus-Prolog 3.0).

(in the implementation natural numbers are represented as terms constructed by `s` and `0`):

Functional vs. relational computations		
Initial goal:	`add(100,100,S)`	`100 + 100 = S`
Time used (msec):	16	8
Heap used (bytes):	2412	2420
Local stack used (bytes):	13352	124
Trail used (bytes):	808	0
Occurrence stack used (bytes):	0	0

This table contains the time and space used for computing the first solution to the initial goal. The time and the local stack space shows the advantage of functional computations.

However, our implementation is not restricted to evaluate functions by rewriting, but also narrowing steps are applied if rewriting fails and some variables of the goal must be instantiated in order to proceed with rewriting. Fribourg [8] has shown that the combination of narrowing and rewriting can reduce the search space in comparison to resolution. At the end of section 2 we have presented an example where rewriting cuts down an infinite search space to a finite one. It is also possible that a finite search space can be dramatically reduced by rewriting. For instance, in the "permutation sort" program a list is sorted by enumerating all permutations and checking whether they are sorted. The relational version of the program ([32], p. 55) enumerates *all* permutations whereas in the functional version not all permutations are enumerated since the generation of a permutation is stopped (by rewriting the goal to "fail") if two consecutive elements `X` and `Y` have the wrong ordering `Y < X` (cf. [8], p. 182). Therefore we yield the following execution times in seconds for different lengths of the input list in our system:

Functional vs. relational computations: permutation sort				
Program:	*Initial goal:*	$n = 6$	$n = 8$	$n = 10$
Relational ([32], p. 55)	`psort([`n`,...,1],L)`	0.65	37.92	3569.50
Functional ([8], p. 182)	`psort([`n`,...,1]) = L`	0.27	1.43	7.43

This is a typical example for the class of "generate-and-test" programs. The rewriting process performs the "test part" of the program: if a portion of the potential solution is generated by narrowing, rewriting immediately tests whether or not this can be a part of the solution. Therefore narrowing and rewriting yield a more efficient control strategy than SLD-resolution for equivalent relational programs. This is achieved in a purely clean and declarative way without any user annotations to control the proof strategy or transformations applied to the source program [5]. A more detailed discussion on this advantage of a functional language based on rewriting and narrowing can be found in [14].

We have also compared our implementation with other implementations of functional languages with pattern matching. The following table contains the results of the naive reverse benchmark for different implementations which we had available.

Naive reverse for a list of 30 elements		
System:	*Machine:*	*Time:*
ALF	Sun4	19 msec
Standard-ML (Edinburgh)	Sun3	54 msec
CAML V 2-6.1	Sun4	28 msec
OBJ3	Sun3	5070 msec
RAP 2.0	Sun4	4800 msec

OBJ3 [23] and RAP [9] are systems for executing equational specifications by rewriting (and narrowing in case of RAP). Since these are based on an interpreter, we can observe the impressive speeding up achieved by our compilational approach. Thus we conjecture that our approach is also more efficient than the implementation technique proposed by Josephson and Dershowitz [22] because they handle unification and control at the interpretive level.

7 Conclusions

We have presented a method to compile a language that amalgamates functional and logic programming styles into code of an abstract machine which can be easily implemented on conventional architectures. The operational semantics of our language is based on resolution for predicates and rewriting and narrowing to evaluate functional expressions. We have shown that narrowing in combination with rewriting is more efficient than resolution for equivalent (flattened) relational programs. This was clear from a theoretical point of view, but our implementation has shown that these advantages can also be used in practical applications.

The integration of functions into logic programming leads to programs which are more readable and easier to understand because functions need not be simulated by predicates and nested functional expressions need not be flattened. Since the programmer can express functional dependencies between data, this information could be used for a better implementation. In our system a functional expression is simplified by rewriting before a narrowing rule is applied. This reduces the search space (without "cuts"!) and avoids the generation of superfluous backtrack points since rewriting is a deterministic process. Thus the non-deterministic narrowing operation is rarely applied.

In some cases the positive effect of rewriting (search space reduction) can also be achieved by analysing a logic program in order to find deterministic computations and inserting "cuts" at appropriate program points. But this analysis may be expensive and do not yield satisfactory results if a predicate is called in different modes: a call with ground terms could have a deterministic computation while a call with non-ground terms may have a non-deterministic computation. Such problems are solved by our implementation in a clean and declarative way: Since rewriting is applied before each narrowing step, a goal is simplified by deterministic rewriting as long as possible depending on the instantiation state of the arguments. A similar behaviour can also be obtained in logic programs by using other control strategies instead of Prolog's fixed left-to-right strategy [29]. But this requires the insertion of control annotations into the program (which may effect completeness because of floundering problems) and the extension of the WAM to deal with such a flexible control strategy. In our declarative solution control annotations are not necessary (see also [8]).

Currently we are working on better methods for code generation which can speed up the rewriting part of the system. At the moment we are using the WAM-instructions for rewriting as shown in this paper, but it is possible to generate particular code for fast pattern matching (see, e.g., [18]). We are also working on the integration of types into the computation process [11] [13] [20] since this allows a further reduction of the search space.

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

References

[1] M. Bellia and G. Levi. The Relation between Logic and Functional Languages: A Survey. *Journal of Logic Programming (3)*, pp. 217–236, 1986.

[2] 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.

[3] 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.

[4] 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.

[5] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling Control. *Journal of Logic Programming (6)*, pp. 135–162, 1989.

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

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

[8] 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.

[9] A. Geser and H. Hussmann. Experiences with the RAP system – a specification interpreter combining term rewriting and resolution. In *Proc. of ESOP 86*, pp. 339–350. Springer LNCS 213, 1986.

[10] 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.

[11] M. Hanus. Polymorphic Higher-Order Programming in Prolog. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 382–397. MIT Press, 1989.

[12] M. Hanus. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 387–401. Springer LNCS 456, 1990.

[13] 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.

[14] M. Hanus. A Declarative Approach to Improve Control in Logic Programming. Univ. Dortmund, 1991.

[15] M. Hanus and A. Schwab. ALF User's Manual. FB Informatik, Univ. Dortmund, 1991.

[16] M. Hanus and A. Schwab. The Implementation of the Functional-Logic Language ALF. FB Informatik, Univ. Dortmund, 1991.

[17] S. Haridi and P. Brand. Andorra Prolog: An Integration of Prolog and Committed Choice Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 745–754, 1988.

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

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

[20] 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.

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

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

[23] C. Kirchner, H. Kirchner, and J. Meseguer. Operational Semantics of OBJ3 (Extended Abstract). In *Proc. of the 15th ICALP*, pp. 287–301. Springer LNCS 317, 1988.

[24] 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.

[25] J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[26] R. Loogen. From Reduction Machines to Narrowing Machines. In *Proc. of the TAPSOFT '91*, pp. 438–457. Springer LNCS 494, 1991.

[27] J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pp. 298–317. Springer LNCS 463, 1990.

[28] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. Technical Report DIA/89/3, Universidad Complutense, Madrid, 1989.

[29] L. Naish. *Negation and Control in Prolog.* Springer LNCS 238, 1987.

[30] W. Nutt, P. Rety, and G. Smolka. Basic Narrowing Revisited. SEKI Report SR-87-07, FB Informatik, Univ. Kaiserslautern, 1987.

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

[32] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

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

A Registers of the A-WAM

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)
`OM`	top of current occurrence stack
`OR`	top of current copy occurrence stack
`AO`	actual occurrence (reference to the current subterm to be evaluated)
`TS`	term start (root of the current argument term)
`OV`	Is the current copy occ. stack valid? May be set to false during rewriting.
`RFP1`, `RFP2`	rewrite fail pointers (addresses of alternative rewrite rules)

The argument registers and temporary variables are identical to the WAM registers [33].

B New instructions of the A-WAM

In the following we list the new instructions of the A-WAM together with a short explanation in alphabetical order.

`allocate_occ`: This instruction is used before a condition in a narrowing or rewrite rule will be proved. It saves the occurrences in `AO` and `TS` onto the occurrence stack and adds a new (empty) current occurrence stack to the list of all occurrence stacks.

`call_narrowing AO,N`: Load the components of the structure at position `AO` into the argument registers and call the narrowing rules for the function at occurrence `AO`. `N` is the number of permanent variables in the current environment.

`call_rewriting R`: This instruction is used to rewrite the current argument term after a narrowing rule has been applied. It starts rewriting at the innermost occurrence `R` (`f/n` or `AO`) and continues with the next instruction (`rebuild_occ_stack`) if the rewriting process is finished.

`call_rewriting R,N`: This instruction is used to rewrite the current argument term in a literal where `N` is the number of permanent variables in the current environment. It starts rewriting at the innermost occurrence `R` (`f/n` or `AO`) and continues with the next instruction (`rebuild_occ_stack`) if the rewriting process is finished.

`copy_pop_occ`: Push `AO` onto the current copy occurrence stack and execute `pop_occ`.

`deallocate_occ`: Delete the last element from the list of occurrence stacks and load registers `AO` and `TS` from the previous occurrence stack. If a backtrack point has been created after the corresponding `allocate_occ`-instruction, it is not allowed to alter previous elements of the occurrence stack list since only the current 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.

`execute_narrowing AO:` This instruction terminates a narrowing rule. The narrowing rules for the function at occurrence `AO` are executed if `AO` is defined, otherwise program pointer `P` is set to `CP`.

`execute_rewriting R:` This instruction terminates a rewrite rule. Registers `RFP1` and `RFP2` are set to "fail" and the rewrite rules for the function `f/n` are executed if `R`=`f/n`, otherwise (`R`=`AO`) the rewrite rules for the function at occurrence `AO` are executed.

`inner_reflection:` This is the last alternative in a sequence of narrowing rules for a partial function. It implements the innermost reflection rule: The term at the actual occurrence `AO` is marked as "environment" and the A-WAM-instruction sequence "`pop_occ ; execute_narrowing AO`" is executed.

`invalid_os:` Set register `OV` to false if the current copy occurrence stack is not empty.

`load_occ R:` Set the actual occurrence register `AO` to the contents of `R`.

`l_trust_me_else fail:` Delete all backtrack points generated after the corresponding `l_try_me_else`, i.e., the pointer to the last backtrack point (register `B`) is set to `Y1`.

`l_try_me_else L,A,N:` Create a backtrack point and store the address of the last backtrack point in the permanent variable `Y1`. `A` is the number of argument registers to be saved and `N` contains the number of permanent variables in the current environment.

`pop_occ:` Pop an element from the current occurrence stack and store the value in register `AO`. If the current occurrence stack is empty, set `AO` to "undefined".

`push_occ R:` Push the contents of `R` onto the current occurrence stack.

`put_...._occ R:` Substitute the current subterm at address `AO` by `R` and store the old value at `AO` on the trail. Furthermore, `put_struct_occ f/n` puts a new structure `f/n` on the top of the heap and replaces the heap cell at address `AO` by a reference to this new structure.

`put_function_occ f/n:` Put the structure `f/n` with the values of the argument registers `A1`,...,`An` onto the heap at address `AO` if this heap cell contains an unbound variable. It is used in the last alternative of the rewrite rules for `f/n`.

`rebuild_occ_stack:` Replace the current (empty) occurrence stack by the current copy occurrence stack if `OV` is true, otherwise by a new occurrence stack for the term at position `TS` (if the copy occurrence stack is invalid).

`reflection:` This instruction implements the reflection rule. It unifies the two sides of an equation (the current literal) which must be a structure referenced by register `TS`.

`reject:` If the current literal is an equation (referenced by register `TS`), then this instruction causes a failure if both sides have different constructors at the same outermost position (a position not belonging to arguments of functions). Otherwise, no action is taken.

`r_try...:` The indexing instructions for rewrite rules are prefixed by `r_`. In contrast to the indexing instructions of the WAM no backtrack point is generated but the address of the alternative clause is stored in `RFP1` or `RFP2`.

`set_begin_of_term R:` Set the term start register `TS` to the contents of `R`.