In *Proc. Fourth International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–23, Springer LNCS 631, 1992

Improving Control of Logic Programs by Using Functional Logic Languages

Michael Hanus

Max-Planck-Institut für Informatik
Im Stadtwald
W-6600 Saarbrücken, Germany
e-mail: michael@mpi-sb.mpg.de

Abstract. This paper shows the advantages of amalgamating functional and logic programming languages. In comparison with pure functional languages, an amalgamated functional logic language has more expressive power. In comparison with pure logic languages, functional logic languages have a better control behaviour. The latter will be shown by presenting methods to translate logic programs into a functional logic language with a narrowing/rewriting semantics. The translated programs produce the same set of answers and have at least the same efficiency as the original programs. But in many cases the control behaviour of the translated programs is improved. This requires the addition of further knowledge to the programs. We discuss methods for this and show the gain in efficiency by means of several examples.

1 Introduction

Many proposals have been made to integrate functional and logic programming languages during the last years (see [3, 11] for surveys). Recently, these proposals became relevant for practical applications because efficient implementations have been developed [5, 8, 19, 33, 35, 48]. This raises the natural question for the advantages of such amalgamated languages. In comparison with pure functional languages, functional logic languages have more expressive power due to the availability of features like function inversion, partial data structures and logic variables [42]. In comparison with pure logic languages, functional logic languages allow to specify functional dependencies and to use nested functional expressions. Although this improves the readability of logic programs, it is not clear whether this is only a minor syntactic improvement (which can be added to logic languages by a simple preprocessor [37]) or there is a genuine advantage of functional logic languages compared to pure logic languages. In this paper we show that the latter is true: functional logic languages have a better operational behaviour than logic languages. We show this by presenting methods to translate logic programs into a functional logic language. These methods ensure that the translated programs produce the same set of answers and have at least the same efficiency as the original programs. But in many cases the translation improves the control behaviour of logic programs which will be demonstrated by several examples.

```
sort(L,M) :- perm(L,M), ord(M).

perm([],[]).
perm([E|L],[F|M]) :- del(F,[E|L],N), perm(N,M).

del(E,[E|L],L).
del(E,[F|L],[F|M]) :- del(E,L,M).

ord([]).
ord([E]).
ord([E,F|L]) :- le(E,F), ord([F|L]).

le(0,E).
le(s(E),s(F)) :- le(E,F).
```

Figure 1. Permutation sort (natural numbers are represented by `s`-terms)

Logic programming allows the specification of problems at an abstract level and permits the execution of the specifications. However, these specifications are often very slowly executed because a lot of search is performed under the standard Prolog computation rule. For instance, Figure 1 specifies the notion of a sorted list (cf. [44], p. 55): a list `M` is a sorted version of a list `L` if `M` is a permutation of `L` and all elements of `M` are in ascending order. We can use this Prolog program to sort the list `[4,3,2,1]` by solving the query `?- sort([4,3,2,1],S)`. But this runs very inefficiently under the standard computation rule because all permutations must be enumerated and tested in order to solve this goal.

Therefore several proposals have been made in order to improve the control of Prolog programs. Naish [36] has extended the standard computation model of Prolog by a coroutining mechanism. He allows the addition of "wait" declarations to predicates. Such declarations have the effect that the resolution of a literal is delayed until the arguments are sufficiently instantiated. If a variable of a delayed literal is bound to a non-variable term, this literal is woken and executed in the next step if it is now sufficiently instantiated. In the permutation sort example, the programmer can add a wait declaration to the predicate `ord` and change the ordering in the first clause into

```
sort(L,M) :- ord(M), perm(L,M).
```

Now the goal `?- sort([3,2,1],S)` is executed in the following way: After the application of the first clause to this goal the literal `ord(S)` is delayed and the literal `perm([3,2,1],S)` will be executed. If `S` is bound to the first part of a permutation of `[3,2,1]` (i.e., a list with two elements and a variable at the tail), then `ord(S)` is activated. If the first two elements of `S` are in the wrong order, then the computation fails and another permutation is tried, otherwise `ord` is delayed again until the next part of the permutation is generated. Thus with this modification not *all* permutations are completely computed and therefore the execution time is better than in the naive approach. Naish has also presented an algorithm which generates the wait declarations from a given program and transforms the program by reordering the goals in a clause. Although this approach seems to be attractive, it has some problems. For instance, the generation of wait declarations is based on heuristics and therefore it is unclear whether these heuristics are generally successful. Moreover, it is possible that the annotated program flounders, i.e., all goals are delayed which is considered as a run-time error. Hence completeness of SLD-resolution can be lost when transforming a logic program into a program with wait declarations (see example at the end of Section 3.3 or the `goodpath` example in [46]).

Another approach to improve control has been developed by Bruynooghe's group [7]. They try to avoid the overhead of coroutining execution by transforming a logic program with coroutining into a logic program with an equivalent behaviour executed under the standard computation rule. The transformation is done in several steps. In the first step a symbolic trace tree of a goal is created where the user has to decide which literal is selected and whether a literal is completely executed or only a single resolution step is made, i.e., the user must supply the system with a *good computation rule*. If a goal in the trace tree is a renaming of a goal in an ancestor node, an arc from this goal to the ancestor node is inserted. This results in a symbolic trace graph which is then reduced and in the last step translated into a logic program simulating the symbolic trace under the standard computation rule. The crucial point in this approach is to find a *good* computation rule for the program with respect to the initial goal. In a recent paper [46] a method for the automated generation of an efficient computation rule is presented. The method is based on a global analysis of the program by abstract interpretation techniques in order to derive the necessary information. Since the arguments for choosing a "good" computation rule are heuristics, it is unclear whether the transformed programs are in any case more efficient than the original ones. Another problem is due to the fact that their method uses a given call pattern for the initial goal. Therefore different versions of the program are generated for different call modes of the goal.

In this paper we propose a much simpler method to improve control of logic programs. This method ensures that the new programs have at least the same efficiency as the original ones. But for a large class of programs ("generate-and-test" programs like permutation sort) we obtain a better efficiency similar to other approaches to improve control. The basic idea is to use a functional logic language and to translate logic programs into functional programs (without considering the initial goal). The motivation for the integration of functional and logic programming languages is to combine the advantages of both programming paradigms in one language: the possibility of *solving* predicates and equations between terms together with the efficient *reduction* paradigm of functional languages. A lot of the proposed amalgamations of functional and logic languages are based on Horn clause logic with equality [40] where the user can define predicates by Horn clauses and functions by (conditional) equations. Predicates are often omitted because they can be represented as Boolean functions. A complete operational semantics is based on the narrowing rule [14, 29, 30]: *narrowing* combines unification of logic languages with rewriting of functional languages, i.e., a narrowing step consists of the unification of a subterm of the goal with the left-hand side of an equation, replacing this subterm by the right-hand side of the equation and applying the unifier to the whole goal. Since we have to take into account *all* subterms of a goal in the next narrowing step, this naive strategy produces a large search space and is less efficient than SLD-resolution (SLD stands for selecting *one* literal in the next resolution step). Also the advantage of functional languages, namely the deterministic reduction principle, is lost by this naive approach.

Therefore a lot of research has been done to improve the narrowing strategy without loosing completeness. Hullot [29] has shown that the restriction to *basic* subterms, i.e., subterms which are not created during unification, is complete. Fribourg [15] has proved that the restriction to subterms at innermost positions is also complete provided that all functions are reducible on all ground terms. Finally, Hölldobler [28] has proved completeness of the combination of basic and innermost narrowing where a so-called innermost reflection rule must be added for partially defined functions. But innermost basic narrowing is not better than SLD-resolution since it has been shown that innermost basic narrowing corresponds to SLD-resolution if a functional program is translated into a logic program by flattening [6]. On the other hand, we can also translate a logic program into a functional one without loosing efficiency if we use the innermost basic narrowing strategy. But now we are able to improve the execution by simplifying the goal by deterministic rewriting before a narrowing step is applied (rewriting is similar to reduction in functional languages with the difference that rewriting is also applied to terms containing variables). The simplification phase cuts down the search space without loosing completeness [28, 39].

We will see in the next sections that the operational behaviour of innermost basic narrowing combined with simplification is similar to SLD-resolution with a particular dynamic control rule. Hence we get an improvement in the execution comparable to previous approaches [7, 36] but with the following advantages:

- The translation technique from logic programs into functional logic programs is simple.
- It is ensured that the translated programs have at least the same efficiency as the original ones. For many programs the efficiency is much better.
- It is ensured that we do not loose completeness: there exists an answer w.r.t. the translated program iff there exists an answer w.r.t. the original program.

The last remark is only true if we use a fair computation strategy. If we use a backtracking implementation of SLD-resolution as in Prolog, the completeness may be lost because of infinite computations. However, infinite paths in the search tree can be cut by the simplification process [15], i.e., it is also possible that we obtain an answer from the functional logic program where the original logic program does not terminate.

These theoretical considerations are only relevant if there is an implementation of the functional logic language which has the same efficiency as current Prolog implementations. Fortunately, this is the case. In [19, 21, 24] it has been shown that it is possible to implement a functional logic language very efficiently by extending the currently known Prolog implementation techniques [47]. The language

ALF ("**A**lgebraic **L**ogic **F**unctional language") is based on the operational semantics sketched above. Innermost basic narrowing and simplification is implemented without overhead in comparison to Prolog's computation strategy, i.e., functional programs are executed with the same efficiency as their relational equivalents by SLD-resolution (see [21] for benchmarks). Therefore it is justified to improve the control of logic programs by translation into a functional logic language.

In the next section we give a precise description of ALF's operational semantics and in Section 3 we present our approach to improve control of logic programs in more detail.

2 Operational semantics of ALF

As mentioned in the previous section, we want to improve the control behaviour of logic programs by translating them into a functional logic language. We have also mentioned that in order to compete with SLD-resolution we have to use a functional logic language with a refined operational semantics, namely innermost basic narrowing and simplification. Hence the target language of the translation process is the language ALF [19, 21] which is based on this semantics. ALF has more features than actually used in this paper, e.g., a module system with parameterization, a type system based on many-sorted logic, predicates which are resolved by resolution etc. (see [25] for details). In the following we outline the operational semantics of ALF in order to understand the translation scheme presented in the next sections.

ALF is a constructor-based language, i.e., the user must specify for each symbol whether it is a constructor or a defined function. Constructors must not be the outermost symbol of the left-hand side of a defining equation, i.e., constructor terms are always irreducible. Hence constructors are used to build data types, and defined functions are operations on these data types (similarly to functional languages like ML [27] or Miranda [45]). The distinction between constructors and defined function symbols is necessary to define the notion of an *innermost position* [15].

An ALF program consists of a set of (conditional) equations which are used in two ways. In a narrowing step an equation is applied to *compute a solution* of a goal (i.e., variables in the goal may be bound to terms), whereas in a rewrite step an equation is applied to *simplify* a goal (i.e., without binding goal variables). Therefore we distinguish between *narrowing rules* (equations applied in narrowing steps) and *rewrite rules* (equations applied in rewrite steps). Usually, all conditional equations of an ALF program are used as narrowing and rewrite rules, but it is also possible to specify rules which are only used for rewriting. Typically, these rules are inductive axioms or CWA-valid axioms (see below). The application of such rules for simplification can reduce the search space and is justified if we are interested in ground-valid answers [15, 39] (i.e., answers which are valid for each ground substitution applied to it).

Figure 2 shows an ALF module to sort a list of naturals. Naturals are represented by the constructors `0` and `s`, `true` and `false` are the constructors of the data type

```
module isort.

  datatype bool = { true ; false }.
  datatype nat  = { 0 ; s(nat) }.
  datatype list = { '.'(nat,list) ; [] }.

  func isort  : list      -> list;
       insert : nat, list -> list;
       le     : nat, nat  -> bool.
rules.
  isort([])    = [].
  isort([E|L]) = insert(E,isort(L)).

  insert(E,[])     = [E].
  insert(E,[F|L])  = [E,F|L]          :- le(E,F) = true.
  insert(E,[F|L])  = [F|insert(E,L)]  :- le(E,F) = false.

  le(0,N)        = true.
  le(s(N),0)     = false.
  le(s(M),s(N)) = le(M,N).

end isort.
```

Figure 2. ALF program for insertion sort

`bool` and lists are defined as in Prolog. The defined functions of this module are `isort` to sort a list of naturals, `insert` to insert an element in an ordered list, and `le` to compare two naturals.

The *declarative semantics* of ALF is the well-known Horn clause logic with equality as to be found in [40]. The *operational semantics* of ALF is based on innermost basic narrowing and rewriting.[1] Before a narrowing step is applied, the goal is simplified to normal form by applying rewrite rules. We will distinguish two kinds of nondeterminism by the keywords "don't know" and "don't care": *don't know* indicates a branching point in the computation where all alternatives must be explored (in parallel or by a backtracking strategy in a concrete implementation); *don't care* indicates a branching point where it is sufficient to select (nondeterministically) one alternative and disregard all other possibilities.

In order to give a precise definition of the operational semantics, we represent a goal (a list of equations to be solved) by a skeleton and an environment part [28, 39]: the *skeleton* is a list of equations composed of terms occurring in the original program, and the *environment* is a substitution which has to be applied to the equations in order to obtain the actual goal. The initial goal G is represented by the pair $\langle G; id\rangle$ where id is the identity substitution. The following scheme describes the operational semantics (if π is a position in a term t, then $t|_\pi$ denotes the subterm of t at position π and $t[s]_\pi$ denotes the term obtained by replacing the subterm $t|_\pi$ by s in t [12]; π is called an *innermost position* of t if the subterm $t|_\pi$ has a defined function

[1] Similarly to EQLOG [18], ALF allows also the definition of predicates which are solved by resolution, but we omit this aspect in the current paper.

symbol at the top and all argument terms consist of variables and constructors). Let $\langle E_1, \ldots, E_n ~;~ \sigma \rangle$ be a given goal ($E_1, \ldots, E_n$ are the skeleton equations and σ is the environment):

1. Select *don't care* a non-variable position π in E_1 and a new variant $l = r \leftarrow C$ of a rewrite rule such that σ' is a substitution with $\sigma(E_1|_\pi) = \sigma'(l)$ and the goal $\langle C ~;~ \sigma' \rangle$ can be derived to the empty goal without instantiating any variables from $\sigma(E_1)$. Then
$$\langle E_1[\sigma'(r)]_\pi, E_2, \ldots, E_n ~;~ \sigma \rangle$$
is the next goal derived by **rewriting**; go to 1.[2] Otherwise go to 2.
2. If the two sides of equation E_1 have different constructors at the same outer position (a position not belonging to arguments of functions), then the whole goal is **rejected**, i.e., the proof fails. Otherwise go to 3.
3. Let π be the leftmost-innermost position in E_1 (if there exists no such position in E_1, go to 4). Select *don't know* (a) or (b):
   (a) Select *don't know* a new variant $l = r \leftarrow C$ of a narrowing rule such that $\sigma(E_1|_\pi)$ and l are unifiable with mgu σ'. Then
   $$\langle C, E_1[r]_\pi, E_2, \ldots, E_n ~;~ \sigma' \circ \sigma \rangle$$
   is the next goal derived by **innermost basic narrowing**; go to 1. Otherwise: fail.
   (b) Let x be a new variable and σ' be the substitution $\{x \mapsto \sigma(E_1|_\pi)\}$. Then
   $$\langle E_1[x]_\pi, E_2, \ldots, E_n ~;~ \sigma' \circ \sigma \rangle$$
   is the next goal derived by **innermost reflection**; go to 3 (this corresponds to the elimination of an innermost redex [28] and is called "null narrowing step" in [6]).
4. If E_1 is the equation $s = t$ and there is a mgu σ' for $\sigma(s)$ and $\sigma(t)$, then
$$\langle E_2, \ldots, E_n ~;~ \sigma' \circ \sigma \rangle$$
is the next goal derived by **reflection**; go to 1. Otherwise: fail.

The attribute *basic* of a narrowing step emphasizes that a narrowing step is only applied at a position of the original program and not at positions introduced by substitutions [29]. The innermost reflection rule need not be applied to completely defined functions, i.e., functions which are reducible on all ground terms of appropriate sorts [15, 28]. Therefore the innermost reflection rule can be avoided by using types and checking whether each function is sufficiently defined for all constructors of their argument types. Since ALF is a typed language and allows such tests, we implicitly assume in this paper that the sufficiently definedness tests are performed

[2] Rewriting is only applied to the first literal, but this is no restriction since a conjunction like E_1, E_2, E_3 can also be written as an equation $and(E_1, and(E_2, E_3)) = true$. This technique will be used in the following sections.

at compile time in order to avoid unnecessary applications of the innermost reflection rule at run time.

This operational semantics is sound and complete if the term rewriting relation generated by the conditional equations is *canonical* (i.e., confluent and terminating [12]) and the condition and the right-hand sides of the conditional equations do not contain *extra-variables* [28]. Moreover, the conditional equations must be *reductive*, i.e., the conditions must be smaller than the left-hand side w.r.t. some termination ordering (otherwise basic conditional narrowing may be incomplete as Middeldorp and Hamoen [34] have pointed out).[3] If a program has conditional equations with extra-variables, there may be other criteria to ensure completeness (e.g., level-confluence [17] or decreasing rules [13]) or it may be possible to transform the program into an equivalent program for which this operational semantics is complete (e.g., Bertling and Ganzinger [4] have proposed such a method). Therefore we allow extra-variables in conditional equations which is the reason for the instantiation condition in the rewrite step.

Rewriting in ALF is applied from innermost to outermost positions, i.e., rewriting corresponds to eager evaluation in functional languages. Similarly to Prolog, ALF uses a backtracking strategy to implement the choices of different clauses in a narrowing step. Hence the theoretical completeness will be lost due to infinite computations, but for finite search trees the operational semantics is complete. Due to the requirement for a canonical and reductive set of equations, the normal form of a term uniquely exists and can be computed by rewriting with an arbitrary matching equation in a rewrite step. Therefore the creation of choice points is only necessary in narrowing steps.

We have mentioned in the introduction that it is also possible to translate functional programs into logic programs by flattening and to execute these programs by SLD-resolution [6]. ALF's operational semantics has the following advantages in comparison to that and other techniques:

- Since rewriting is a deterministic process (or it can be also seen as "don't care" nondeterminism) and rewriting is done before narrowing, deterministic computations are performed whenever it is possible. This avoids superfluous creation of choice points. Nondeterministic computations are only performed if it is necessary, i.e., if a solution (binding of a goal variable) must be guessed by an application of a narrowing rule.
- A similar behaviour can be achieved in Prolog by inserting delays [36, 37]. But this has the disadvantage that the program with delays may flounder which corresponds to incompleteness. This cannot be the case in ALF because of ALF's complete operational semantics.
- The residuation principle of Le Fun [1] is also related to ALF's operational semantics: If a Le Fun function is applied to a variable argument, the application is delayed until the variable becomes bound to a non-variable term. But this

[3] The requirement for reductive conditional equations is not a real restriction since tools for checking canonicity of conditional equations usually have this requirement [16].

semantics is also incomplete in some cases. For instance, if `append` is a function that concatenates two lists, we can extract the last element `E` of a given list `L` by solving the equation

```
append(_,[E]) = L
```

Residuation will delay this computation (since the first argument is always unbound) and we obtain no result for `E`. But ALF will solve this goal by narrowing and rewriting and delivers the unique solution for `E`. Moreover, the residuation principle of Le Fun may produce an infinite search space for examples where ALF's or Prolog's operational semantics has a finite search space [23].

- Similarly to ALF, the Andorra computation model [26] prefers deterministic computations before nondeterministic ones. However, the rewriting mechanism of ALF yields deterministic computations also when more than one clause matches (see `max` example in section 3.3) 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. The same is true for the relation of ALF and Prolog with Simplification [9]: ALF's rewriting mechanism is more general than simplification because unifiable (but confluent) equations, equations with deleting left-hand side variables and conditional equations are admissible rewrite rules in ALF.
- It is also important to note that ALF's operational semantics can be implemented with the same efficiency as current Prolog implementations [21]. The overhead of searching the next innermost subterm can be avoided by using a stack of references to subterms in the goal (see [19] and [21] for details).

These arguments gives us the feeling that the computation principle of ALF is more efficient than Prolog's SLD-resolution. In the next section we will show how logic programs can be translated into ALF programs and what we gain from such a translation.

3 Translating logic programs into functional programs

There are two principle ways to translate a logic program into a functional one:

1. We consider each predicate as a Boolean function and translate the Horn clauses of each predicate into a functional expression over the Booleans.
2. We try to find out functional dependencies between the arguments of a predicate. If there is such a dependency, we transform the predicate into function from input to output arguments, otherwise we transform the predicate into a Boolean function.

The second method is clearly an extension of the first one. The first method is very simple and always applicable, but we will also show techniques for the second translation method.

Example: The predicates `member` and `append` are defined by the following logic program:

```
member(E,[E|L]).
member(E,[F|L]) :- member(E,L).
append([],L,L).
append([E|R],L,[E|RL]) :- append(R,L,RL).
```

We can translate this program into a functional program by the first method:

func `member`: *term* , *term* → *bool*
```
member(E,[E|L]) = true.
member(E,[F|L]) = true :- member(E,L) = true.
```
func `append`: *term* , *term* , *term* → *bool*
```
append([],L,L) = true.
append([E|R],L,[E|RL]) = true :- append(R,L,RL) = true.
```

But we can also perceive that the first and the second argument of `append` determine the value of the third argument, i.e., there is a functional dependency between the arguments of `append`. Therefore it is possible to translate `append` into the following function definition:

func `append`: *term* , *term* → *term*
```
append([],L)     = L.
append([E|R],L) = [E|append(R,L)].
```

In the following we will discuss both methods in more detail.

3.1 Translating all predicates into Boolean functions

In this section we discuss the simple approach where each n-ary predicate is translated into an n-ary Boolean function. We define the translation of logic programs into functional programs by the following rules:

Facts:	L .	$\Rightarrow$	L `= true.`
Clauses:	L `:-` $L_1, \ldots, L_n$.	$\Rightarrow$	L `= true :-` (L_1 `and` $\cdots$ `and` L_n) `= true.`
Goals:	`?-` $L_1, \ldots, L_n$.	$\Rightarrow$	`?-` (L_1 `and` $\cdots$ `and` L_n) `= true.`

The Boolean values together with the function `and` are defined in Figure 3.[4] Since the right-hand side of each equation in the translated program is the constant `true`, we get immediately the following property of the translated programs:[5]

[4] The declaration "`infixright 650`" defines the symbol "`and`" as a right-associative infix operator with priority `650`. This has the similar effect as the declaration `op(650,xfy,and)` in Prolog.

[5] In this paper we do not deal with the problem of proving termination of the narrowing/rewrite rules since ALF's operational semantics does also work for nonterminating programs. Moreover, the correspondence of narrowing and resolution derivations [6] is also valid for nonterminating programs. But note that the operational semantics may be incomplete for some nonterminating programs and therefore we implicitly assume that the rewrite relation is terminating and all conditional rules are reductive.

```
module bool.

  datatype bool = { true ; false }.

  func and : bool, bool -> bool infixright 650.
rules.
  false and B     = false.
  true  and B     = B.
  B     and false = false.
  B     and true  = B.

end bool.
```

Figure 3. Module for Boolean values

Proposition 1. *If R is the set of conditional equations obtained by translating a logic program with the above translation scheme, then R is confluent.*

Hence we can use the translated equations as narrowing rules and solve the translated goals by innermost basic narrowing. But what is the relation between narrowing derivations of the functional program and resolution derivations of the original logic programs? Bosco et al. [6] have shown that there is a strong relationship between these derivations, i.e., every innermost basic narrowing derivation of a functional program corresponds to an SLD-resolution derivation with the leftmost selection rule if the functional program is appropriately flattened into a logic program. Applying their result to our framework we obtain the following proposition (actually, they have proved the correspondence for unconditional equations but it is not difficult to extend it to the conditional case):

Proposition 2. *Let P be a logic program and R be the set of conditional equations obtained by translating P. For each goal G and each SLD-resolution with the leftmost selection rule there is a corresponding innermost basic narrowing sequence for the translated goal G' where each resolution step corresponds to an innermost basic narrowing step together with at most one application of the equation "`true and B = B`".*

Hence the logic program and its functional version have the same efficiency (if we neglect the simple application of the equation "`true and B = B`") and produce the same set of answers. But the efficiency of the functional version can be improved by adding rewrite rules. We know from Section 2 that we can add the narrowing rules also as rewrite rules and perform rewriting between narrowing steps without loosing completeness. Rewriting can be done in a deterministic way, i.e., it is not necessary to generate choice points during rewriting and therefore rewriting may reduce the search space. For instance, if the functional program contains the equations

```
member(E,[E|L]) = true.
member(E,[F|L]) = true :- member(E,L) = true.
```

both as narrowing rules and rewrite rules, the goal

```
?- member(2,[1,2,3]) = true.
```

is proved by rewriting without generating any choice point. Note that two choice points are generated during the corresponding SLD-resolution (using standard implementation techniques [47]).

Since rewriting cannot bind any goal variable (a rewrite rule is applicable if the left-hand side of the equation *matches* the current subterm), it can only be applied as a *test* and then it avoids the search for alternative proofs of this test. This is a slight improvement and does not justify the translation from the well-known Prolog framework into the new functional logic framework. For instance, if we translate the permutation sort program in Figure 1, the functional version is executed in the same slow way as the relational version. The improvement of the control behaviour in the framework of Naish [36] or Bruynooghe [7] is due to the fact that the failure of a goal is detected early in the computation. Therefore we must add negative information to our functional program. This will be outlined in the next section.

3.2 Adding negative information

For the case that we are interested in valid answers w.r.t. the least Herbrand model, which is a natural assumption in logic programming [32], Fribourg [15] has shown that we can add equations which are valid w.r.t. the so-called "Closed World Assumption" (*CWA-valid*) as rewrite rules to our program. The operational semantics is still sound w.r.t. ground-valid answers, i.e., answers which are valid for each ground substitution applied to it. A conditional equation

L `= false :-` L_1 `and` $\cdots$ `and` L_n `= true.`

is called *CWA-valid* w.r.t. a set of conditional equations R if for any ground constructor substitution σ

$R \models \sigma(L)$ `= true :-` $\sigma(L_1)$ `and` $\cdots$ `and` $\sigma(L_n)$ `= true`

does not hold (later we will also allow equations of the form L`=false` in the condition part; CWA-validity of such clauses is similarly defined). If we rewrite a literal L`=true` to the equation `false=true` by CWA-valid rewrite rules, we can immediately reject the whole goal (compare the "rejection" rule in Section 2). This technique does not affect the completeness of the operational semantics but can be an essential improvement. For instance, consider the following clauses [15] (`a`, `b` and `c` are constructors):

```
on(a,b) = true.
on(b,c) = true.
above(X,Y) = true :- on(X,Y) = true.
above(X,Y) = true :- above(X,Z) and on(Z,Y) = true.
```

The execution of the goal `?- above(a,a) = true` leads to an infinite loop. If the CWA-valid equation `above(X,X) = false` is inserted into the set of rewrite rules, the goal `?- above(a,a) = true` is first rewritten into `?- false = true` and then it fails by the rejection rule.

As a further example, consider the following set of rules defining the predicates `even` and `le` (less-or-equal):

```
even(0) = true.
even(s(s(N))) = true :- even(N) = true.
le(0,N) = true.
le(s(M),s(N)) = true :- le(M,N) = true.
```

The execution of the goal `?- even(N) and le(N,s(s(0))) = true` leads to an infinite loop after producing the answers `N=0` and `N=s(s(0))`, because the predicate `even` generates an infinite number of even naturals. In order to avoid this loop, we may add the CWA-valid equation `le(s(N),0) = false`. But this does not solve the problem because there is the following infinite derivation (the narrowed subterms are underlined):

```
?- even(N) and le(N,s(s(0))) = true.
?- even(N1) = true, true and le(s(s(N1)),s(s(0))) = true.
?- even(N2) = true, true = true,
                    true and le(s(s(s(s(N2)))),s(s(0))) = true.
...
```

The reason for this infinite derivation is that only the first literal of a goal is simplified by rewriting (cf. Section 2).[6] But this is no real problem since we can also translate the original logic program for `even` and `le` in the following way:

```
even(0) = true.
even(s(s(N))) = even(N).
le(0,N) = true.
le(s(M),s(N)) = le(M,N).
```

Now we obtain the following derivation with the additional CWA-valid rewrite rule `le(s(N),0) = false`:

```
?- even(N) and le(N,s(s(0))) = true.
        narrowing with the second equation for even
?- even(N1) and le(s(s(N1)),s(s(0))) = true.
        simplifying the goal
?- even(N1) and le(N1,0) = true.
        narrowing with the second equation for even
?- even(N2) and le(s(s(N2)),0) = true.
        simplifying the goal:
?- false = true.
        failure by rejection
```

Hence the search space of this goal is finite in contrast to the original Prolog program. In order to implement the improved proof strategy, we simply modify our translation scheme for clauses:

[6] This is for the sake of an efficient implementation [21] because rewriting the whole goal allows less optimizations during the compilation phase.

```
sort(L,M) = perm(L,M) and ord(M).
perm([],[])       = true.
perm([E|L],[F|M]) = del(F,[E|L],N) and perm(N,M).
del(E,[E|L],L)     = true.
del(E,[F|L],[F|M]) = del(E,L,M).
ord([])      = true.
ord([E])     = true.
ord([E,F|L]) = le(E,F) and ord([F|L]).
le(0,E)       = true.
le(s(E),s(F)) = le(E,F).
```

Figure 4. Functional version of permutation sort

Translation of clauses: Let L `:-` L_1`,`$\ldots$`,`L_n be a clause for which one of the following conditions holds:

1. L is not unifiable with the head of any variant of another clause of the logic program.
2. If there are a variant of another clause L' `:-` L'_1`,`$\ldots$`,`L'_m and a unifier σ for L and L', then the goals `?-` $\sigma(L_1$ `and` $\cdots$ `and` $L_n)$ `= true` and `?-` $\sigma(L'_1$ `and` $\cdots$ `and` $L'_m)$ `= true` can be rewritten to the same goal using the rewrite rules corresponding to the logic program w.r.t. the old translation scheme (*confluence of clauses*).

Then the clause is translated into the equation

L `= (`L_1 `and` $\cdots$ `and` L_n`).`

otherwise it is translated into the conditional equation

L `= true :- (`L_1 `and` $\cdots$ `and` L_n`) = true.`

Note that this modified translation is only necessary because of the restricted rewriting in ALF. If we use another functional logic language which performs rewriting on the whole goal (like SLOG [15]), this modification is superfluous. The conditions guarantee that the translated program is confluent, i.e., Proposition 1 holds also for the modified translation scheme. Figure 4 shows the translation of the logic permutation sort program of Figure 1. Note that this is nearly the same program which Fribourg [15] has presented in a rather ad-hoc manner.

The final problem is the generation of CWA-valid rules for rewriting. For instance, from the given rules

```
le(0,E) = true.
le(s(E),s(F)) = le(E,F).
```

we have to generate the CWA-valid rule

```
le(s(E),0) = false.
```

In this case it can be done by inspecting the constructors of the argument terms of the left-hand side, and then generating `false` rules for all constructor terms on which le is not reducible. Fortunately, there is also a systematic method for doing this in general. *Intensional negation* [2] is a transformation technique which synthesizes clauses for new predicates p'_i from a given logic program for the predicates p_i. The new predicates p'_i describe the finite failure set of the original predicates p_i and hence they are a computable approximation of the CWA-valid literals [32]. E.g., given the clauses

```
even(0).
even(s(s(X))) :- even(X).
```

intensional negation generates the new clauses

```
even'(s(0)).
even'(s(s(X))) :- even'(X).
```

which define the odd numbers. If we translate the predicate `even'(···)` into `even(···) = false`, we obtain the CWA-valid rewrite rule used in our `even` example above.

We do not propose to compute the intensional negation of all defined predicates since this leads to a large number of additional rewrite rules. Moreover, intensional negation does not generate Horn clauses for the negated predicates if the original clauses contain local variables in their bodies (see [2] for details). But in most cases it is possible and sufficient to compute the negation of some *base predicates*. For instance, from the given definition of the less-or-equal predicate `le` in Figure 1 we obtain by intensional negation the CWA-valid rule

```
le(s(X),0) = false.
```

If we add this single rule as a rewrite rule to the narrowing/rewrite rules of Figure 4, the computation is automatically optimized without control instructions: as soon as the variable `M` in the goal `perm([···],M) and ord(M) = true` is bound to a partial list `[a,b|L]` with a greater than b, the goal is simplified by rewriting as follows:

```
perm([···],[a,b|L]) and ord([a,b|L]) = true
⟹  perm([···],[a,b|L]) and le(a,b) and ord([b|L]) = true
⟹  perm([···],[a,b|L]) and false and ord([b|L]) = true
⟹  perm([···],[a,b|L]) and false = true
⟹  false = true
```

Hence not all permutations are enumerated but the computation of a permutation immediately stops if two consecutive elements are in the wrong order. Thus we have obtained the same improved operational behaviour as in related approaches [7, 36] in a simple and declarative way. The following table shows the execution times in seconds to sort the list `[n,...,2,1]` for different values of n:

Length of the list:	5	6	7	8	9	10
Original logic program (Figure 1)	0.10	0.65	4.63	37.92	348.70	3569.50
Translated functional program (Figure 4)	0.10	0.27	0.61	1.43	3.28	7.43

Both the original logic version and the functional version were executed by the ALF system since ALF also allows the definition of predicates which are executed as in

Prolog (pure logic ALF programs are translated into code of an abstract machine as described in [47]).

Using our method we can translate arbitrary logic programs into functional programs. An essential speeding up will be obtained for the class of "generate-and-test" programs like the permutation sort above, the classical 8-queens problem or the `goodpath` program of [46].

3.3 A more sophisticated translation scheme

Until now we have simply translated predicates into Boolean functions. But it is often the case that a programmer has a function in mind but must write it down as a predicate in a logic program. Any n-ary function can be expressed as a $(n+1)$-ary relation by adding the result as an additional argument. For instance, the concatenation of two lists is a function from two list arguments into another list. It can be defined in a functional language with pattern-matching by the equations

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

Since Prolog does not allow the definition of functions and nested expressions, a Prolog programmer must express the concatenation as a predicate with three arguments and writes down the following clauses:

```
append([],L,L).
append([E|R],L,[E|RL]) :- append(R,L,RL).
```

Innermost basic narrowing execution of the first program is equivalent to the Prolog execution of the `append` clauses. But the additional simplification mechanism of the functional evaluation can avoid infinite loops which may occur in the relational evaluation. For instance, Naish [36] has noted that the following goal causes an infinite loop under the standard Prolog evaluation rule for any order of literals and clauses:

```
?- append([1|V],W,X), append(X,Y,[2|Z]).
```

But the evaluation of the equivalent `conc` equation causes a fail and does not loop:

```
?- conc(conc([1|V],W),Y) = [2|Z].
```
simplifying the goal by two applications of the second `conc` *rule:*
```
?- [1|conc(conc(V,W),Y)] = [2|Z].
```
failure by rejection since 1 *and* 2 *are different constructor terms*

Note that the failure situation is detected without any additional CWA-valid rule. The only knowledge used here is the fact that constructor terms are irreducible and therefore different constructor terms cannot denote the same object. This knowledge is expressed by the rejection rule (Section 2).

We see from this example that it is desirable to declare predicates with functional dependencies between arguments as functions from input to output arguments and not as Boolean functions. Since we use a functional logic language with a complete operational semantics, this does not restrict the class of evaluable goals.

If a programmer writes down a program, he has the functional dependencies between data in mind. Thus he can directly define the functions if he uses a functional logic language like ALF. But it is also possible to find functional dependencies in a given Prolog program. In general, a functional dependency is an undecidable property of a logic program [38]. However, in particular cases one can find sufficient criteria for that. For instance, Reddy [41] has proposed a technique for transforming logic programs into functional ones. However, his technique is based on modes for the predicates in the logic program which obviously restricts the application of his method (e.g., if a predicate is called in two different modes, two different functions are generated for that predicate). Debray and Warren [10] have proposed a technique to detect functional computations in logic programs. It is also based on modes and tries to find out mutual exclusions between different clauses of a predicate. We do not want to discuss the detection of functional dependencies in more detail but give another sufficient criterion for this property.

Let p be an n-ary predicate. If we suppose that the first $n-1$ arguments determine the value of the last argument (the generalization to other argument combinations is straightforward), we modify our translation scheme of Section 3.1 in the following way. Instead of defining p as an n-ary Boolean function, we define p as an $(n-1)$-ary function and perform the following transformation steps:

1. Every literal $p(t_1, \ldots, t_n)$ in a clause or in the goal is replaced by the equation $p(t_1, \ldots, t_{n-1}) = t_n$.
2. If we have generated an equation $p(t_1, \ldots, t_{n-1}) = X$ in the body of a clause and X is a variable which does not occur in the left-hand side of the clause head, all occurrences of X in the clause are replaced by the term $p(t_1, \ldots, t_{n-1})$ and the equation is deleted
3. If we have generated an equation $p(t_1, \ldots, t_{n-1}) = X$ in the goal and X is a variable, then all occurrences of X in the goal are replaced by the term $p(t_1, \ldots, t_{n-1})$ and the equation is deleted.

It is easy to see that this transformation is the inverse of flattening the clauses (compare [6]). Since Bosco et al. [6] have shown the correspondence of innermost basic narrowing derivations and SLD-resolution derivations w.r.t. the flattened clauses, we immediately obtain the following proposition:[7]

Proposition 3. *If the set of rules after the transformation steps is canonical and reductive, then the functional program has the same set of answers as the original logic program.*[8]

[7] The requirement for canonical and reductive rules is not essential for the correspondence of narrowing and resolution derivations, but it is important for the unique termination of the rewriting process between the narrowing steps.

[8] Actually, the functional program may compute more answers than the original logic program since it can "skip" calls to partially defined functions by the innermost reflection

Hence, if we have a supposition about the functional dependencies of the arguments of the predicates, we apply the above transformation and then check the resulting program for canonicity which can often be done by simple syntactic criteria (e.g., the arguments of the left-hand side are constructor terms and two different left-hand sides are not unifiable) or by special completion procedures for conditional equations [16]. For instance, the logic program of `append` is transformed into the functional `conc` program above which is obviously canonical. As a further example take the following logic program:

```
max(X,Y,Y) :- le(X,Y).
max(X,Y,X) :- ge(X,Y).
le(0,X).
le(s(X),s(Y)) :- le(X,Y).
ge(X,0).
ge(s(X),s(Y)) :- ge(X,Y).
```

The clauses for `max` are not mutually exclusive and therefore the algorithm in [10] does not detect a functionality in these clauses. However, if we suppose that the third argument of predicate `max` is functional dependent on the first and the second argument, we apply our transformation above and obtain the following rules:

```
max(X,Y) = Y :- le(X,Y) = true.
max(X,Y) = X :- ge(X,Y) = true.
le(0,X) = true.
le(s(X),s(Y)) = le(X,Y).
ge(X,0) = true.
ge(s(X),s(Y)) = ge(X,Y).
```

Now we can construct a successful proof of the canonicity of these rules using the completion procedure in [16] (this can be easily done since there is an interface between the ALF system and the completion system). Hence the canonicity criterion is more general than other more syntactically oriented criteria [10, 41].

The transformation of predicates into functions has at least two advantages. Firstly, the search space can be reduced because more terms can be evaluated by rewriting (e.g., the term `conc([1],[2])` is evaluable by rewriting where `append([1],[2],L)` must be evaluated by narrowing/resolution) and thus the rejection rule is applicable in more cases (see the above example for `conc` and `append`). Secondly, the execution is more efficient because less nondeterminism must be implemented. For instance, the execution of the goal `add(s(s(s(0))),s(s(s(0)))) = L` w.r.t. the functional program

```
add(0,N) = N.          add(s(M),N) = s(add(M,N))
add(N,0) = N.          add(N,s(M)) = s(add(N,M))
```

rule. An innermost reflection step for the subterm $p(t_1,\dots,t_k)$ corresponds to resolution with the unit clause $p(X_1,\dots,X_k,p(X_1,\dots,X_k))$ in the logic program. To state the exact equivalence of the functional and the logic program, these facts must be added to the logic program for functions which are not completely defined.

does not create any choice point since the goal is fully evaluated by rewriting and not by nondeterministic narrowing, whereas the execution of the goal `add(s(s(s(0))),s(s(s(0))),L)` w.r.t. the Prolog program

```
add(0,N,N).          add(s(M),N,s(L)) :- add(M,N,L).
add(N,0,N).          add(N,s(M),s(L)) :- add(N,M,L).
```

creates at least three choice points. The concrete effect of this behaviour on the execution time and memory usage can be found in [21].

Our final example demonstrates the advantage of our approach in comparison to other proposals to improve control. In this example we combine the advanced translation scheme with the addition of negative information. Consider the following Prolog program for the definition of mobiles (a *mobile* is a *fish* with a fixed positive weight, or a *bridge* of weight 1 (=`s(0)`) where two mobiles of the same weight hang at the left and right end):

```
mobile(fish(_)).
mobile(bridge(M1,M2)) :-
      mobile(M1), mobile(M2),
      weight(M1,W1), weight(M2,W2), equal(W1,W2).
weight(fish(s(W)),s(W)).     % a fish has a positive weight
weight(bridge(M1,M2),s(W)) :-
                  weight(M1,W1), weight(M2,W2), add(W1,W2,W).
add(N,0,N).
add(0,N,N).
add(N,s(M),s(Z)) :- add(N,M,Z).
add(s(M),N,s(Z)) :- add(M,N,Z).
equal(0,0).
equal(s(M),s(N)) :- equal(M,N).
```

If we want to know whether a given `fish`/`bridge`-structure is a mobile, we prove the goal

```
?- mobile(bridge(fish(s(s(s(0)))),bridge(fish(s(0)),fish(s(0))))).
```

which yields the answer `yes`. If we want to get all mobiles of weight 3, we prove

```
?- mobile(M), weight(M,s(s(s(0)))).
```

This query goes into an infinite loop after enumerating all solutions because it generates bigger and bigger mobiles which are not of weight 3. If we want to avoid this under the standard computation rule, we have to restructure the whole program.[9] Hence we need another program for another mode of predicate `mobile` which is

[9] Note that Naish's algorithm for generating wait declarations [36] does not help because it generates waits for the first arguments of `mobile` and `weight`; hence the goal immediately flounders. The method of [46] depends on a given call pattern of the initial goal, i.e., it would generate two programs for the two modes of `mobile`. Generally, if the modes of the initial goal are not known in advance, it is necessary to generate a program for *each* possible mode of the goal.

clearly unsatisfactory from a logical point of view. This problem can be avoided using our translation scheme. It is easy to see that `weight` and `add` are functions where the last argument depends on the other arguments. Hence we obtain the following functional program using our translation method:

```
mobile(fish(_)) = true.
mobile(bridge(M1,M2)) =
      mobile(M1) and mobile(M2) and equal(weight(M1),weight(M2)).
weight(fish(s(W)))    = s(W).
weight(bridge(M1,M2)) = s(add(weight(M1),weight(M2))).
add(N,0) = N.
add(0,N) = N.
add(N,s(M)) = s(add(N,M)).
add(s(M),N) = s(add(M,N)).
equal(0,0) = true.
equal(s(M),s(N)) = equal(M,N).
```

This program is canonical which can be easily checked by standard completion procedures for equational specifications. In order to avoid the infinite loop, we simply add negative information about unequal numbers. Intensional negation generates the following rules (among others):

```
equal(0,s(M)) = false.
equal(s(M),0) = false.
```

After adding these equations as rewrite rules, `mobile` has a finite search tree for all modes, i.e., the following queries terminate after enumerating all solutions:

```
?- mobile(bridge(fish(s(s(s(0)))),bridge(fish(s(0)),fish(s(0)))))=B.
?- mobile(M) and equal(weight(M),s(s(s(0)))) = true.
```

The termination of the last goal is due to the fact that the generation of mobiles `M` with weight greater than 3 is prevented by rewriting

```
equal(weight(M),s(s(s(0))))
```

to `false`.

4 Conclusions

We have presented a technique to translate logic programs into programs of the functional logic language ALF. This translation ensures that the set of answers to a goal remains the same and the translated programs have at least the same efficiency (search space) as the original programs. This is due to the correspondence between SLD-derivations and innermost basic narrowing derivations. However, in many cases the search space is reduced by simplifying goals (rewriting) and comparing both sides of an equation (rejection) which is effective for the class of generate-and-test programs. This improved control behaviour requires the addition of negative knowledge or the transformation of predicates into functions between arguments. Fortunately, there are well-known tools for both tasks. The necessary negative knowledge can be derived by intensional negation of the program, and the validity of a functional transformation can be checked by completion procedures for equational specifications.

Of course, similar effects or, in some cases, better effects can be obtained by other methods to influence the control of logic programs, e.g., delay declarations for predicates or inserting cuts. But the advantage of our transformation method is the declarative nature of the approach. Since ALF's proof strategy is complete, any solution to the original logic program is also computed w.r.t. the new strategy. This may be not the case in other methods where goals can flounder (because of delay declarations) or solutions are lost (because of inserting "red" cuts).

We do not propose to use our method for the automatic translation of logic programs into functional logic programs. The motivation for our method was to show that functional logic languages are superior to pure logic languages since it is possible to translate any logic program into a functional equivalent which has the same set of answers but is often more efficient. Hence we should directly use functional logic languages instead of pure logic languages. Nevertheless, the presented transformation techniques point to important aspects for improving the efficiency of functional logic programs: functional dependencies reduce the number of possible search paths, and negative knowledge supports the early detection of failures.

In order to increase the power of logic programming, it is necessary to improve the operational behaviour in a declarative way such that logic programs become more deterministic without loosing logically important answers. The integration of functions is one possibility as shown in this paper. Further improvements can be achieved by including constraints over specific domains [31] or type information which influences the search space [20, 22, 43].

Acknowledgements. The author is grateful to Andreas Schwab for improving and debugging the current implementation of ALF, and to Alexander Bockmayr, Rita Loogen and Michael Gollner for their comments on a previous version of this paper.

References

1. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and Functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 17–23, San Francisco, 1987.
2. R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A Transformational Approach to Negation in Logic Programming. *Journal of Logic Programming (8)*, pp. 201–228, 1990.
3. M. Bellia and G. Levi. The Relation between Logic and Functional Languages: A Survey. *Journal of Logic Programming (3)*, pp. 217–236, 1986.
4. 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.
5. 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.

6. P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science 59*, pp. 3–23, 1988.
7. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling Control. *Journal of Logic Programming (6)*, pp. 135–162, 1989.
8. M.M.T. Chakravarty and H.C.R. Lock. The Implementation of Lazy Narrowing. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 123–134. Springer LNCS 528, 1991.
9. P.H. Cheong and L. Fribourg. Efficient Integration of Simplification into Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 359–370. Springer LNCS 528, 1991.
10. S.K. Debray and D.S. Warren. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pp. 451–481, 1989.
11. D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations, and Equations.* Prentice Hall, 1986.
12. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pp. 243–320. Elsevier, 1990.
13. N. Dershowitz and M. Okada. Conditional Equational Programming and the Theory of Conditional Term Rewriting. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 337–346, 1988.
14. 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.
15. 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.
16. H. Ganzinger. A Completion Procedure for Conditional Equations. *J. of Symb. Computation*, Vol. 11, pp. 51–81, 1991.
17. E. Giovannetti and C. Moiso. A completeness result for E-unification algorithms based on conditional narrowing. In *Proc. Workshop on Foundations of Logic and Functional Programming*, pp. 157–167. Springer LNCS 306, 1986.
18. 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.
19. 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.
20. 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.
21. M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.
22. M. Hanus. Parametric Order-Sorted Types in Logic Programming. In *Proc. of the TAPSOFT '91*, pp. 181–200. Springer LNCS 494, 1991.
23. M. Hanus. An Abstract Interpretation Algorithm for Residuating Logic Programs. Technical Report MPI-I-92-217, Max-Planck-Institut für Informatik, Saarbrücken, 1992.
24. M. Hanus. Incremental Rewriting in Narrowing Derivations. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming.* Springer LNCS, 1992.
25. M. Hanus and A. Schwab. ALF User's Manual. FB Informatik, Univ. Dortmund, 1991.

26. 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.
27. R. Harper, D.B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, 1986.
28. S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNCS 353, 1989.
29. J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.
30. H. Hussmann. Unification in Conditional-Equational Theories. In *Proc. EUROCAL '85*, pp. 543–553. Springer LNCS 204, 1985.
31. J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. of the 14th ACM Symposium on Principles of Programming Languages*, pp. 111–119, Munich, 1987.
32. J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.
33. R. Loogen. From Reduction Machines to Narrowing Machines. In *Proc. of the TAP-SOFT '91*, pp. 438–457. Springer LNCS 494, 1991.
34. A. Middeldorp and E. Hamoen. Counterexamples to Completeness Results for Basic Narrowing. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming.* Springer LNCS, 1992.
35. A. Mück. Compilation of Narrowing. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*, pp. 16–29. Springer LNCS 456, 1990.
36. L. Naish. *Negation and Control in Prolog.* Springer LNCS 238, 1987.
37. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 15–26. Springer LNCS 528, 1991.
38. K. Nakamura. Control of logic program execution based on the functional relation. In *Proc. Third International Conference on Logic Programming (London)*, pp. 505–512. Springer LNCS 225, 1986.
39. W. Nutt, P. Réty, and G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, Vol. 7, pp. 295–317, 1989.
40. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer, 1988.
41. U.S. Reddy. Transformation of Logic Programs into Functional Programs. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 187–196, Atlantic City, 1984.
42. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.
43. G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* Dissertation, FB Informatik, Univ. Kaiserslautern, 1989.
44. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.
45. D. Turner. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, pp. 1–16. Springer LNCS 201, 1985.
46. K. Verschaetse, D. De Schreye, and M. Bruynooghe. Generation And Compilation of Efficient Computation Rules. In *Proc. Seventh International Conference on Logic Programming*, pp. 700–714. MIT Press, 1990.
47. D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.
48. D. Wolz. Design of a Compiler for Lazy Pattern Driven Narrowing. In *Recent Trends in Data Type Specification*, pp. 362–379. Springer LNCS 534, 1990.