In *Proc. of the Twelfth International Conference on Logic Programming*, Tokyo, June 1995, MIT Press, pp. 665–679

On Extra Variables in (Equational) Logic Programming

Michael Hanus
Informatik II, RWTH Aachen
D-52056 Aachen, Germany
hanus@informatik.rwth-aachen.de

Abstract

Extra variables in a clause are variables which occur in the body but not in the head. It has been argued that extra variables are necessary and contribute to the expressive power of logic languages. In the first part of this paper, we show that this is not true in general. For this purpose, we provide a simple syntactic transformation of each logic program into a logic program without extra variables, and we show a strong correspondence between the original and the transformed program. In the second and main part of this paper, we use a similar technique to provide new completeness results for equational logic programs with extra variables. In equational logic programming it is well known that extra variables cause problems since narrowing, the standard operational semantics for equational logic programming, may become incomplete in the presence of extra variables. Using a simple syntactic transformation, we derive a number of new completeness results for narrowing. In particular, we show the completeness of narrowing strategies in the presence of nonterminating functions and extra variables in right-hand sides of rewrite rules. Using these results, current functional logic languages can be extended in a practically useful way.

1 Introduction

Extra variables in a Horn clause $L \Leftarrow B$ are variables in the body B which do not occur in L (other notions are *existential variables* [20], *local variables* [2], or *fresh variables* [19]). It has been argued that extra variables are necessary and contribute to the expressive power of logic languages. For instance, Dershowitz and Okada [7] claim that the restriction of logic programming to clauses without extra variables "is unacceptable since even very simple relations, such as transitivity, require extra variables in conditions." In the first part of this paper, we show that this is not true in general since each clause containing extra variables can be transformed into a clause without extra variables by adding the extra variables as a new argument to the predicate in the head. We state a strong correspondence between the original and the transformed program w.r.t. the declarative and the operational semantics, in order to show that there is no loss due to this transformation.

In the second and main part of this paper, we consider equational logic programs. This class of programs is important since it is a basis for integrating functional and logic programming (see [14] for a recent survey on this subject). In equational logic programming it is well known that extra variables cause problems since narrowing, the standard operational semantics for equational logic programming, may become incomplete in the presence of extra variables. This can be seen by the following example [11]:

Example 1.1 Consider the following equational logic program:[1]

```
a → b          b → c  ⇐  f(X,b)=f(c,X)
a → c
```

This system has all the properties usually required for completeness of narrowing, i.e., it is confluent and terminating. However, narrowing cannot infer the validity of the equation `b=c` since there is only the following infinite derivation (the subterm where a rule is applied is underlined in each step):

$$\begin{aligned} \underline{\mathtt{b}}\mathtt{=c} &\leadsto \mathtt{f(X,\underline{b})=f(c,X),\ c=c} \\ &\leadsto \mathtt{f(X1,\underline{b})=f(c,X1),\ f(X,c)=f(c,X),\ c=c} \leadsto \cdots \end{aligned}$$

In order to prove the condition of the last rule, the extra variable `X` must be instantiated to `a` and the instantiated occurrences must be derived to `c` and `b`, respectively. However, this is not provided by the narrowing calculus. Although narrowing is complete for confluent and terminating equational logic programs without extra variables, this example shows that narrowing becomes incomplete in the presence of extra variables. □

Extra variables are useful from a programming point of view. For instance, the *let* construct used in functional programming to share common subexpressions can be expressed in equational logic programming using extra variables [5]. Therefore, much research has been carried out in order to characterize classes of equational logic programs with extra variables for which narrowing is complete (see Section 3 for a detailed discussion). The aim of the second part of this paper is to provide such completeness results. For this purpose, we transform general equational logic programs into programs without extra variables and discuss conditions for the adequacy of this transformation. The main condition is the property that different occurrences of an extra variable need not be derived to different terms in an instantiated rule (note that this is necessary in Example 1.1). An interesting class satisfying this condition are almost orthogonal programs, which is a reasonable class from a programming point of view. Based on these observations, we characterize new classes of equational logic programs for which narrowing and particular narrowing strategies are complete. For instance, we show the completeness of narrowing and lazy narrowing for a class of programs which allows extra variables in right-hand sides of clause heads. Such programs are very useful in practice but seldom discussed in the narrowing literature.

2 Extra Variables in Logic Programming

In this section we propose a method to avoid extra variables in pure logic programming. We use standard notions from logic programming as to be found in [16]. *Terms* are constructed from variables and function symbols, and (program) *clauses* have the form $L_0 \Leftarrow L_1, \ldots, L_k$, where each literal L_i is a predicate p applied to a sequence of terms $t_1, \ldots, t_n$ (in the following we abbreviate sequences of terms by $\overline{t}$). L_0 is called *head* and $L_1, \ldots, L_k$ is called *body* of the clause. The set of variables occurring in a term t is denoted by

[1] Since the equation in the clause head is always used to derive an instance of the left-hand side to an instance of the right-hand side, we use the arrow '→' instead of the equality symbol in the head.

$\mathcal{V}ar(t)$ (similarly for other syntactic constructions). A term t is called *ground* if $\mathcal{V}ar(t) = \emptyset$. A *logic program* is a set of clauses. Consider the clause

$$C\colon p(\overline{t}) \Leftarrow q_1(\overline{t_1}), \ldots, q_k(\overline{t_k})$$

A variable $x \in \mathcal{V}ar(C)$ is called *extra variable* if $x \notin \mathcal{V}ar(\overline{t})$. In order to eliminate all extra variables, we apply the transformation *eev* (*e*liminate *e*xtra *v*ariables) to this clause, which is defined by

$$eev(C)\colon p(\overline{t}, v_{n+k}(x_1, \ldots, x_n, y_1, \ldots, y_k)) \Leftarrow q_1(\overline{t_1}, y_1), \ldots, q_k(\overline{t_k}, y_k)$$

where $x_1, \ldots, x_n$ are the extra variables of C and $y_1, \ldots, y_k$ are new variables not occurring in C.[2] Moreover, $v_0, v_1, v_2, \ldots$ is a family of new function symbols not occurring in the original program. We extend the transformation *eev* to programs by applying *eev* to each clause of the program.

Example 2.1 Let P be the program consisting of the following clauses:

```
append([],L,L)
append([E|R],L,[E|RL]) ⇐ append(R,L,RL)
last(L,E) ⇐ append(R,[E],L)
```

Then the transformed program $eev(P)$ contains the following clauses:

```
append([],L,L,v0)
append([E|R],L,[E|RL],v1(Y)) ⇐ append(R,L,RL,Y)
last(L,E,v2(R,Y)) ⇐ append(R,[E],L,Y)
```

□

In the following, we state a strong correspondence between P and $eev(P)$ w.r.t. the declarative and operational semantics. In particular, we show that the initial model of P is identical to the initial model of $eev(P)$ provided that the last argument of all predicates is deleted. For this purpose, we define a mapping on Herbrand interpretations which deletes the additional arguments introduced by *eev*. Let H be a Herbrand interpretation. Then $dla(H)$ (*d*elete *l*ast *a*rgument) is the Herbrand interpretation defined by $dla(H) := \{p(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n, t_{n+1}) \in H\}$.

Theorem 2.2 *Let H be the least Herbrand model of the logic program P, and H' be the least Herbrand model of $P' := eev(P)$. Then $H = dla(H')$.*

This theorem shows that there is no basic difference in the declarative semantics between P and $eev(P)$. Everything which is valid w.r.t. P is also valid w.r.t. $eev(P)$, and vice versa, if we disregard the additional arguments in $eev(P)$. The following theorem shows a similar property for the operational semantics (SLD-resolution).

Theorem 2.3 *Let P be a logic program, $G = p_1(\overline{t_1}), \ldots, p_k(\overline{t_k})$ be a goal, and $x_1, \ldots, x_k$ be new variables.*

1. *If σ is a computed answer for G w.r.t. P, then there are terms $e_1, \ldots, e_k$ such that $\{x_1 \mapsto e_1, \ldots, x_k \mapsto e_k\} \circ \sigma$ is a computed answer for $G' = p_1(\overline{t_1}, x_1), \ldots, p_k(\overline{t_k}, x_k)$ w.r.t. $eev(P)$.*

2. *If σ' is a computed answer for $G' = p_1(\overline{t_1}, x_1), \ldots, p_k(\overline{t_k}, x_k)$ w.r.t. $eev(P)$, then σ' restricted to $\mathcal{V}ar(G)$ is a computed answer for G w.r.t. P.*

[2] The order of the variables in the term $v_{n+k}(x_1, \ldots, x_n, y_1, \ldots, y_k)$ is irrelevant. Therefore, we can fix an arbitrary order for each clause.

The proof of this theorem is based on the fact that each resolution derivation w.r.t. P can be transformed into a resolution derivation w.r.t. $eev(P)$, and vice versa. Thus there is also a strong correspondence between P and $eev(P)$ w.r.t. the derivation trees, i.e., P and $eev(P)$ have the same operational behavior. This shows that the restriction to logic programs without extra variables is not a real restriction, i.e., *extra variables are not an important feature of logic programming.*

3 Extra Variables in Equational Logic Programs

Equational logic programming (see [14] for a survey) amalgamates functional and logic programming styles. It permits the definition of predicates by Horn clauses and the definition of functions by (conditional) equations. Since predicates can be represented as Boolean functions, we assume that all clauses in an equational logic program have the form

$$l \to r \Leftarrow s_1 = t_1, \ldots, s_k = t_k$$

($k \geq 0$), where $l, r, s_1, t_1, \ldots, s_k, t_k$ are terms and l is not a variable. Such a clause is also called *conditional rewrite rule*, and *unconditional rewrite rule* in case of $k = 0$. A *conditional term rewriting system* (CTRS) is a set of conditional rewrite rules. For instance, Example 1.1 is a CTRS. We consider an *equational logic program* as a CTRS.

3.1 Basic Definitions

In order to give a precise definition of the computation with CTRS, we recall basic notions of (conditional) term rewriting [4, 6].

Substitutions and most general unifiers are defined as in logic programming [16]. A *position* p in a term t is represented by a sequence of natural numbers (where Λ denotes the root position), $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [6] for details).

Let $\to$ be a binary relation on a set S. Then $\to^*$ denotes the transitive and reflexive closure of the relation $\to$. We write $e_1 \downarrow e_2$ if there exists an element $e_3 \in S$ with $e_1 \to^* e_3$ and $e_2 \to^* e_3$. $\to$ is called *confluent* if $e_1 \downarrow e_2$ for all $e, e_1, e_2 \in S$ with $e \to^* e_1$ and $e \to^* e_2$.

Let $\mathcal{R}$ be an unconditional term rewriting system, i.e., an equational logic program where all rules have the form $l \to r$ with $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A *rewrite step* (w.r.t. $\mathcal{R}$) is an application of a rewrite rule to a term (rewriting with conditional rules is discussed below), i.e., $t \to_{\mathcal{R}} s$ if there are a position p in t, a rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. In this case we say t is *reducible*. A term t is called *irreducible* or in *normal form* if there is no term s with $t \to_{\mathcal{R}} s$.

The confluence of the rewrite relation $\to_{\mathcal{R}}$ is a basic requirement to apply rules only in one direction during equational reasoning. In order to ensure confluence even for nonterminating rewrite systems,[3] we need some syntactic restrictions on the rewrite rules. A rewrite rule $l \to r$ is called *left-linear* if

[3]We do not require the termination of the rewrite system since this cannot be checked automatically. Moreover, such a requirement excludes important functional programming techniques like programming with infinite data structures.

there are no multiple occurrences of the same variable in l. An unconditional term rewriting system $\mathcal{R}$ is called *orthogonal* if each rule $l \to r \in \mathcal{R}$ is left-linear and for each non-variable subterm $l|_p$ of l there exists no rule $l' \to r' \in \mathcal{R}$ such that $l|_p$ and l' unify (where $l' \to r'$ is not a variant of $l \to r$ in case of $p = \Lambda$). $\mathcal{R}$ is *almost orthogonal* if all rules are left-linear and for each pairs of rules $l \to r, l' \to r' \in \mathcal{R}$, nonvariable subterm $l|_p$ of l, and mgu σ for $l|_p$ and l', p is the root position Λ and the terms $\sigma(r)$ and $\sigma(r')$ are identical.

An important property of almost orthogonal systems is the confluence of the rewrite relation (see [15] for a comprehensive survey on results for orthogonal systems).

If $\mathcal{R}$ is a CTRS, we denote by $\mathcal{R}_u := \{l \to r \mid l \to r \Leftarrow C \in R\}$ the *unconditional part* of $\mathcal{R}$. A CTRS $\mathcal{R}$ is called (*almost*) *orthogonal* if $\mathcal{R}_u$ is (almost) orthogonal.

3.2 Equational Logic Programs

The computation mechanism of unconditional term rewrite systems was defined by the rewrite relation $\to_{\mathcal{R}}$ in the previous section. If we want to define the computation with a CTRS, we have to explain the evaluation of the condition in a rewrite step. Due to [4, 7], there are different possibilities. Here we consider *normal* CTRS where $t_1, \ldots, t_k$ are ground normal forms w.r.t. $\mathcal{R}_u$ for each condition $s_1 = t_1, \ldots, s_k = t_k$, and such a condition is provable if every s_i is reducible to t_i. Note that this definition of conditional rewriting is recursive, but we can also provide an iterative definition. Let $\mathcal{R}$ be a normal CTRS. We inductively define the following unconditional term rewriting systems $\mathcal{R}_n$ $(n \geq 0)$ by:

$$\mathcal{R}_0 := \{l \to r \mid l \to r \in \mathcal{R}\}$$
$$\mathcal{R}_{n+1} := \{\sigma(l) \to \sigma(r) \mid l \to r \Leftarrow s_1 = t_1, \ldots, s_k = t_k \in \mathcal{R} \text{ and } \sigma(s_i) \to_{\mathcal{R}_n} t_i\}$$

We have $s \to_{\mathcal{R}} t$ iff $s \to_{\mathcal{R}_n} t$ for some $n \geq 0$. The restriction to normal CTRS is essential, otherwise the rewrite relation may not be confluent even for orthogonal CTRS (see [4]). On the other hand, normal CTRS have the following confluence property [15]:

Theorem 3.1 *The rewrite relation of an almost orthogonal normal CTRS is confluent.*

Therefore, we consider in the following only normal CTRS as *equational logic programs* (this restriction is also made in the functional logic languages BABEL [18] and K-LEAF [10]). This is not a restriction from a logic programming point of view, since each logic program can be transformed into a almost orthogonal normal CTRS by representing predicates as Boolean functions and eliminating multiple occurrences of variables in left-hand sides by introducing new variables and new equations for them in the condition part (see [18] for details).

In practice, most equational logic programs are *constructor-based*, i.e., the set of function symbols is divided into a set of *constructors* $\mathcal{C}$ and a set of *defined functions* or *operations* $\mathcal{D}$ (see, for instance, the functional logic languages ALF [12], BABEL [18], K-LEAF [10], or SLOG [9]). A *constructor term* is a term containing only variables and symbols from $\mathcal{C}$. In a *constructor-based term rewrite system*, the left-hand side of each clause must be of the form $f(t_1, \ldots, t_n)$, where $f \in \mathcal{D}$ and $t_1, \ldots, t_n$ are constructor terms. Additionally, in a *constructor-based normal CTRS*, each conditional rule $l \to r \Leftarrow s_1 = t_1, \ldots, s_k = t_k$ has the property that $t_1, \ldots, t_k$ are ground constructor terms.

In constructor-based normal CTRS we cannot write arbitrary equations in conditions. However, we can provide an explicit definition of an equality function $\equiv$ between constructor terms by the following rules (this *strict equality* is the only sensible notion of equality for possible nonterminating systems, since normal forms may not exist [10, 18]):

$$\begin{array}{rcll} c \equiv c & \to & true & \text{for all 0-ary } c \in \mathcal{C} \\ c(x_1,\ldots,x_n) \equiv c(y_1,\ldots,y_n) & \to & \wedge_{i=1}^{n}(x_i \equiv y_i) & \text{for all } n\text{-ary } c \in \mathcal{C} \\ true \wedge x & \to & x & \end{array}$$

The reduction of $s \equiv t$ to *true* is equivalent to the reduction of s and t to a same ground constructor term ([1], Proposition 1). In the rest of this paper, we assume that an equation $s \equiv t$ in a condition of a constructor-based normal CTRS denotes the equation $(s \equiv t) = true$.

We are interested in the influence of extra variables to the completeness of narrowing strategies for equational logic programs. In contrast to pure logic programming, equational logic programming allows a refined classification of rules according to the occurrence of extra variables. Each conditional rule $l \to r \Leftarrow C$ is classified according to the following table [17]:

Type	Requirement
1	$\mathcal{V}ar(r) \cup \mathcal{V}ar(C) \subseteq \mathcal{V}ar(l)$
2	$\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$
3	$\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(C)$
4	no restrictions

All variables in a conditional rule which do not occur in the left-hand side l are called *extra variables*. An n-CTRS contains only rules of type n, i.e., a 1-CTRS does not contain extra variables, a 2-CTRS may contain extra variables only in the condition, and a 3-CTRS may contain extra variables in the right-hand side, but these extra variables must also occur in the condition.

Example 3.2 The program in Example 1.1 is a 2-CTRS, and the following equational version of Example 2.1 is a constructor-based normal 3-CTRS:

```
append([],    L) → L
append([E|R],L) → [E|append(R,L)]
last(L) → E  ⇐  append(R,[E])≡L
```

□

3.3 Conditional Narrowing

In equational logic programming we are interested in solving equational goals, i.e., we want to compute a substitution such that terms rewrite to some normal forms under this substitution. Due to the restriction on conditions in rules introduced in the previous section, we define a (*normal equational*) *goal* (w.r.t. a normal CTRS $\mathcal{R}$) as a sequence of equations $s_1 = t_1, \ldots, s_k = t_k$, where $t_1, \ldots, t_k$ are ground normal forms w.r.t. $\mathcal{R}_u$. Since it is straightforward to extend the definitions of Section 3.1 to goals, we will use them in the following. For instance, we use notions like "subterms of goals" and apply rewrite steps to goals.

A *narrowing step* transforms a goal G into another goal by applying a rule to some subterm of G. More precisely, G narrows to G', denoted $G \leadsto_\sigma G'$, if there exist a nonvariable position p in the goal G (i.e., $G|_p$ is not a variable), a variant $l \to r \Leftarrow C$ of a rewrite rule in $\mathcal{R}$ and a substitution σ such that σ is a mgu of $G|_p$ and l, and $G' = \sigma(C, G[r]_p)$. Since $\mathcal{R}$ is a normal CTRS, it is clear that G' is again a well-defined goal. If there is a narrowing sequence $G_1 \rightsquigarrow_{\sigma_1} G_2 \rightsquigarrow_{\sigma_2} \cdots \rightsquigarrow_{\sigma_{n-1}} G_n$, we write $G_1 \rightsquigarrow^*_{\sigma} G_n$ with $\sigma = \sigma_{n-1} \circ \cdots \circ \sigma_2 \circ \sigma_1$. A narrowing sequence is *successful* if the final goal G_n is *trivial*, i.e., it has the form $t_1 = t_1, \ldots, t_k = t_k$.

The important property of evaluation strategies for (equational) logic programs is their completeness, i.e., their ability to compute all answers which are valid w.r.t. the declarative semantics. In our context we say narrowing is *complete* w.r.t. the equational logic program $\mathcal{R}$ if, for all goals G and substitutions σ so that $\sigma(G)$ can be rewritten to a trivial goal, there exists a narrowing derivation $G \rightsquigarrow^*_{\sigma'} G'$, where G' is a trivial goal and $\sigma = \phi \circ \sigma'$ for some substitution ϕ. That is, each valid answer σ is subsumed by a more general answer σ' computed by narrowing.

There are many results for the completeness of narrowing w.r.t. different classes of programs (see [17] for a comprehensive survey). However, simple narrowing defined so far is more or less of theoretical interest due to its huge search space. In order to reduce the search space and to avoid superfluous work, lazy narrowing strategies have been proposed for languages like BABEL [18] and K-LEAF [10], where lazy narrowing selects an outermost position but also allows narrowing steps at an inner position if the value at this position is demanded by some rule (see [18] for details). It is well-known that *lazy narrowing* is complete for almost orthogonal normal 2-CTRS. However, there are many cases where 2-CTRS are too restricted and 3-CTRS are appropriate, but no completeness results are known for this class. Moreover, there are operationally better strategies than lazy narrowing. For instance, *needed narrowing* [1] is an optimal strategy for inductively sequential programs, which is a subclass of unconditional orthogonal programs, and for almost orthogonal programs it has been shown that the combination of lazy narrowing with intermediate simplification steps yields a better behavior [13]. Again, there are no results for these refined strategies w.r.t. extra variables.

In order to avoid separate completeness proofs w.r.t. extra variables for all these (and possible future) extensions, we present a systematic method to eliminate extra variables in equational logic programs. The method is based on the ideas presented in Section 2, but the incompleteness of narrowing in the presence of extra variables shows that this method cannot work in general. Therefore, we will discuss conditions for the adequacy of our method.

3.4 Eliminating Extra Variables in Conditional Rules

In this section we present a transformation on equational logic programs to eliminate all extra variables. The purpose of this transformation is to provide a general method to derive completeness results in the presence of extra variables. This method consists of the following steps:

1. Transform an equational logic program into a new program without extra variables.
2. Apply a complete narrowing strategy to the transformed program (note that more such strategies are known if extra variables do not occur).
3. Check the correspondence of narrowing derivations between the original and the transformed program.

In this section we discuss conditions for the correctness of steps 1 and 3. Applications of the entire method are discussed in Section 3.5.

In order to eliminate extra variables in equational logic programs, we transform each rewrite rule by adding new arguments to each function occurring in the rule. Since functions can be nested, we have to add new arguments in each subterm. For this purpose, we denote by $\widehat{t}$ the term obtained from t by adding a new variable argument to each function occurring in t, i.e., $\widehat{t}$ can be defined as follows:

$$\begin{array}{lcll} \widehat{x} & = & x & \text{for all variables } x \\ \widehat{t} & = & f(\widehat{t_1},\ldots,\widehat{t_n},y) & \text{if } t = f(t_1,\ldots,t_n) \text{ and } y \text{ is a new variable} \end{array}$$

The new arguments added to each function call are called *extension arguments* and the new variables introduced in these arguments are called *extension variables*. Terms that contain extension arguments for each subterm (which may be instantiated) are called *extended terms*. Although the names of the extension variables are not fixed, we consider in the following the transformation $\widehat{\ }$ as a mapping from terms into terms (this can be formalized by taking a list of new variables as an additional argument to $\widehat{\ }$, but for the sake of readability we avoid this formalism). The transformation will also be applied to lists of terms and equations. We omit the straightforward definition.

Each conditional rewrite rule R: $f(\overline{t}) \to r \Leftarrow C$ is transformed into a rule $eev(R)$ by applying the transformation $\widehat{\ }$ to $\overline{t}$, r and C, and adding the extra variables to the left-hand side, i.e.,

$$eev(R)\text{: } f(\widehat{\overline{t}}, v_n(x_1,\ldots,x_n)) \to \widehat{r} \Leftarrow \widehat{C}$$

where $\{x_1,\ldots,x_n\} = (\mathcal{V}ar(\widehat{r}) \cup \mathcal{V}ar(\widehat{C}))\backslash\mathcal{V}ar(\widehat{\overline{t}})$.[4] The transformed clause may not be a normal one, but this causes no problems since the requirement for normal CTRS is only necessary for the original programs in order to ensure the confluence of the original rewrite relation.

We extend eev to sets of rewrite rules by applying it to each rule. For the sake of readability, we use the following obvious optimization in concrete examples: Introduce extension arguments only in function calls of the form $f(\overline{s})$ where there is some rewrite rule $f(\overline{t}) \to r \Leftarrow C$ for f. In particular, extension arguments are not introduced in constructor terms if $\mathcal{R}$ is a constructor-based program.

Example 3.3 Let $\mathcal{R}$ be the program of Example 1.1. Then $eev(\mathcal{R})$ is the following program:

```
a(v1(Y)) → b(Y)          b(v2(X,Z)) → c  ⇐  f(X,b(Z))=f(c,X)
a(v0)    → c
```

It is not necessary to add extension arguments to the functions `c` and `f` since there are no rewrite rules for them. □

The elimination of extra variables in equational logic programs seems to be very similar to pure logic programs. However, there is an essential difference. The transformation does not change the meaning in the case of pure logic

[4] In contrast to pure logic programming, the order of the variables in the term $v_n(x_1,\ldots,x_n)$ is relevant to ensure that the transformed programs are almost orthogonal if the original programs are almost orthogonal (see Proposition 3.8). Therefore, we fix the same ordering principle for all rules. A possible choice is a left-to-right innermost ordering for all variables in $\widehat{r}, \widehat{C}$.

programs (cf. Theorem 2.2), but this is no longer true in the equational case. The *meaning of an equational logic program* is the set of valid equalities. For instance, `b=c` is valid w.r.t. Example 1.1 (since the instantiated condition `f(a,b)=f(c,a)` can be rewritten to the trivial equation `f(c,b)=f(c,b)`, i.e., `b` $\to_{\mathcal{R}_1}$ `c`). However, no instance of the equation `b(V)=c` is valid w.r.t. the transformed program in Example 3.3. In the original program the term `a` can be rewritten to `b` as well as `c`, which is necessary to prove the condition of the last rule. However, in the transformed program, there is no term which is simultaneously reducible to `b(Y)` and `c`.

The meanings of the original and the transformed program differ whenever it is necessary to rewrite an instance of a variable to different terms in the original program. The inversion of this observation yields a criterion for the adequacy of the transformation. We can ensure that the original and the transformed program have the same meaning if all occurrences of the same variable are reduced to an identical term, i.e., if the same rewrite steps are applied to all occurrences of a variable (in the instantiated rule). This can be expressed by the notion of *sharing*, which means that all occurrences of a rule variable are represented only once. Sharing is also a well-known implementation technique in functional and logic languages. Sharing in rewriting can be formally treated in the framework of term graph rewriting [3]. In order to avoid repeating all details of term graph rewriting, we assume familiarity with graphs to represent shared subterms (see [3] for details). We only cite the following result, which is important in our framework.

Theorem 3.4 ([3]) *If $\mathcal{R}$ is an unconditional almost orthogonal term rewriting system, then graph rewriting (where all variables in rules are shared) is a sound and complete implementation of term rewriting; in particular, the normal forms (w.r.t. traditional term rewriting) of terms are also computable if all rule variables are shared.*

The restriction to almost orthogonal systems is essential. Otherwise, rewriting with sharing is incomplete (see [3]). To apply the result of Theorem 3.4 in our framework, we have to extend it to conditional rewrite systems. Although this is not possible in general, sharing is a complete implementation for the class of programs which we consider as equational logic programs. This also shows that the restriction to *normal* CTRS is sensible from an implementation point of view.

Theorem 3.5 *Let $\mathcal{R}$ be an almost orthogonal normal CTRS (with extra variables). Then all variables in rewrite rules can be shared during the computation of a normal form.*

Now we want to relate rewrite proofs in $\mathcal{R}$ with rewrite proofs in the transformed system $eev(\mathcal{R})$. In order to compare extended terms with original terms, we define a mapping dv to delete extension arguments by $dv(x) = x$ for all variables x and $dv(f(t_1, \ldots, t_n, t_{n+1})) = f(dv(t_1), \ldots, dv(t_n))$. Clearly, $dv(\widehat{t}) = t$ for all terms t. The following theorem shows that every normal form computation w.r.t. $\mathcal{R}$ can also be performed for the extended terms w.r.t. $eev(\mathcal{R})$, provided that $\mathcal{R}$ is an almost orthogonal normal CTRS.

Theorem 3.6 *Let $\mathcal{R}$ be an almost orthogonal normal CTRS (with extra variables), t be a term and $\mathcal{R}' = eev(\mathcal{R})$. If $t \to^*_{\mathcal{R}} s$ (where s is a normal form), then there is an extended term t' with $dv(t') = t$ and $t' \to^*_{\mathcal{R}'} \widehat{s}$.*

This theorem implies that all strict equalities w.r.t. $\mathcal{R}$ are also valid w.r.t. $eev(\mathcal{R})$. The next theorem shows that each narrowing derivation w.r.t. $eev(\mathcal{R})$ corresponds to a narrowing derivation w.r.t. $\mathcal{R}$, i.e., if there is a narrowing derivation on the extended level, then there is also a narrowing derivation on the original level. This property will be used to state new completeness results for narrowing strategies in the presence of extra variables. Remember that all trivial goals have the form $t_1 = t_1, \ldots, t_n = t_n$, where $t_1, \ldots, t_n$ are in normal form (not necessarily ground if they contain extension arguments).

Theorem 3.7 *Let $\mathcal{R}$ be a normal CTRS such that $eev(\mathcal{R})$ is almost orthogonal and G be a goal. If there is a narrowing derivation $\widehat{G} \rightsquigarrow^*_\sigma G_1$, where G_1 is a trivial goal, then there is a narrowing derivation $G \rightsquigarrow^*_\phi G_0$ with $dv(G_1) = G_0$ and $dv(\sigma(x)) = \phi(x)$ for all $x \in \mathcal{V}ar(G)$. Moreover, the narrowing positions in both derivations are identical, and the applied rules correspond via the transformation eev.*

If $\mathcal{R}$ is an almost orthogonal normal CTRS and we want to apply our transformation in order to show the completeness of sophisticated narrowing strategies, we have to ensure that the transformed program $eev(\mathcal{R})$ is also almost orthogonal (Theorem 3.7). The following proposition shows that this is always the case.

Proposition 3.8 *If $\mathcal{R}$ is an almost orthogonal CTRS, then $eev(\mathcal{R})$ is almost orthogonal.*

We mentioned in Section 3.3 that simple narrowing has a huge search space and, therefore, sophisticated narrowing strategies are needed in practice. In general, a narrowing strategy restricts the number of possible narrowing steps, i.e., it can be seen as a mapping which assigns to each goal a set of pairs of positions and rules.[5] However, a narrowing strategy should not destroy completeness, and completeness results are often known only for equational logic programs without extra variables. In order to overcome these problems, we can apply the results of this section to transfer completeness results for narrowing strategies from programs without extra variables to programs which may contain extra variables. The following main result shows the general method.

Theorem 3.9 *Let $\mathcal{R}$ be an almost orthogonal normal CTRS (with extra variables) and N be a narrowing strategy which is complete for $eev(\mathcal{R})$. Then N is also complete for $\mathcal{R}$.*

The following section contains concrete applications of this result.

3.5 Application of Extra Variable Elimination

3.5.1 Inductively Sequential Systems

Lazy narrowing is complete for almost orthogonal normal 2-CTRS [18]. However, it is well known that lazy narrowing may perform superfluous narrowing

[5] An exception is the needed narrowing strategy [1] which additionally assigns a unifier because the unifier in a needed narrowing step is not necessarily a most general one.

steps due to the interaction of redex selection and rule selection. As an alternative, needed narrowing is proposed in [1]. The needed narrowing strategy is optimal w.r.t. the length of the derivations and the number of computed solutions. Needed narrowing is defined for the class of inductively sequential systems. These are particular constructor-based orthogonal unconditional rewrite systems (see [1] for a precise definition). Roughly speaking, in inductively sequential systems all rules defining a function can be organized in a hierarchical structure, called definitional tree, which represents a unique selection of a rule by a case distinction on the arguments for each ground function call. For instance, the rules for `append` in Example 3.2 are inductively sequential, since a unique selection of a rule can be made by the first argument of `append`: if this argument is an empty list (`[]`), the first rule is selected, and the second rule is selected if this argument is a nonempty list (`[·|·]`). On the other hand, the rules of Example 1.1 are not inductively sequential, since the first as well as the second rule can be applied to the term '`a`'.

We will use the results of the previous section to extend needed narrowing to conditional rewrite rules with extra variables in a simple way. A CTRS $\mathcal{R}$ is called *inductively sequential* if it is a constructor-based normal CTRS and its unconditional part $\mathcal{R}_u$ is inductively sequential. Since inductively sequential systems are orthogonal, we can use the method proposed in [4] to translate inductively sequential normal CTRS into an unconditional system. For this purpose, we introduce for each conditional rule R: $l \to r \Leftarrow s = u$ of $\mathcal{R}$ (where u is a ground constructor term) a new function symbol $cond_R$ and replace R by the following unconditional rules:

$$\begin{array}{rcl} l & \to & cond_R(s,r) \\ cond_R(u,x) & \to & x \end{array}$$

We denote by $uc(\mathcal{R})$ the new unconditional system obtained from $\mathcal{R}$. Since u is a ground *constructor* term, the new unconditional system is inductively sequential if the original system is an inductively sequential CTRS without extra variables.[6] Moreover, there is a strong correspondence between the rewrite derivations (see [4], Proposition 2.5.4). In order to deal with extra variables, we have to translate $\mathcal{R}$ by the transformation eev before applying uc. The following proposition is obvious since the introduction of extension arguments does not influence the non-overlapping of left-hand sides.

Proposition 3.10 *If $\mathcal{R}$ is an inductively sequential CTRS, then $uc(eev(\mathcal{R}))$ is an unconditional inductively sequential rewrite system.*

Example 3.11 Consider the following inductively sequential CTRS $\mathcal{R}$ which defines the Boolean function `member` on the basis of the function `append`:

```
append([],    L) → L
append([E|R],L) → [E|append(R,L)]
member(E,L) → true ⇐ append(L1,[E|L2])≡L
```

Then the transformed system $uc(eev(\mathcal{R}))$ consists of the following rules:

```
append([],    L,v0)          → L
append([E|R],L,v1(X))        → [E|append(R,L,X)]
member(E,L,v3(L1,L2,X)) → cond(append(L1,[E|L2],X)≡L,true)
cond(true,X) → X
```

□

[6] Proposition 2.5.3 in [4] is not true in the presence of extra variables.

Since needed narrowing is an optimal and complete strategy for inductively sequential unconditional systems, we can apply the results of the previous section (as summarized in Theorem 3.9), and we obtain the following new result.

Theorem 3.12 *Needed narrowing is complete for inductively sequential CTRS (with extra variables). Moreover, it is optimal w.r.t. the length of the derivations and the number of computed solutions.*

Since this result can be easily extended to *overlapping rules with excluding conditions*, we obtain with our translation method an optimal narrowing strategy for a large class of equational logic programs.

3.5.2 Extra Variables in Right-Hand Sides

Current functional logic languages like BABEL [18] and K-LEAF [10]) permit extra variables in conditions but not in the right-hand side of conditional rules. However, as observed by several authors [8, 15, 17], it makes good sense to allow extra variables also in right-hand sides if they occur in conditions (3-CTRS). Example 3.2 shows a sensible use of extra variables in right-hand sides. The following example [15] shows that such extra variables can be a replacement for the *let* construct of functional languages.

Example 3.13 The Fibonacci numbers can be computed by the following conditional rules:

```
fib(0)    → <0,1>
fib(s(X)) → <Z,Y+Z> ⇐ fib(X) ≡ <Y,Z>
```

□

However, an unrestricted use of extra variables in right-hand sides leads to nonconfluent rewrite relations even for non-overlapping normal CTRS. To ensure the confluence of the rewrite relation and completeness of narrowing, additional restrictions are needed. Middeldorp and Hamoen [17] showed that narrowing is complete for level-confluent and terminating 3-CTRS. In [5, 7, 19] 3-CTRS with a special rewrite relation are proposed, where extra variables are instantiated only to irreducible terms and all such instantiations of conditional rules must be *decreasing* (i.e., the left-hand side must greater than the conditions and right-hand side w.r.t. a termination ordering). Narrowing is complete for such rewrite systems. Since we do not want to restrict ourselves to terminating rewrite systems, we need other conditions. For this purpose, we call a CTRS $\mathcal{R}$ *functional* if the following conditions hold:

1. $\mathcal{R}$ is a normal CTRS.
2. The unconditional part $\mathcal{R}_u$ is almost orthogonal (where we use the same definition as in Section 3.1 but do not require $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ for all $l \to r \in \mathcal{R}_u$).
3. $\to_{\mathcal{R}}$ is confluent.

Conditions 1 and 2 are necessary to extend Theorem 3.5 and Proposition 3.8 to functional CTRS. Since these conditions are not sufficient for the confluence of the rewrite relation, we have the explicit confluence condition 3. We will discuss sufficient conditions ensuring confluence below. Note that each almost orthogonal normal 2-CTRS is functional (by Theorem 3.1), while a 4-CTRS cannot be functional. Hence the class of functional CTRS lies between the classes of almost orthogonal normal 2-CTRS and 3-CTRS.

We want to apply our transformation to show the completeness of narrowing strategies for functional CTRS. Since functional CTRS are transformed by *eev* into almost orthogonal CTRS, it is easy to check that Theorem 3.9 is also valid for functional CTRS:

Theorem 3.14 *Let $\mathcal{R}$ be a functional CTRS and N be a narrowing strategy which is complete for $eev(\mathcal{R})$. Then N is also complete for $\mathcal{R}$.*

We can use this result to show the completeness of various narrowing strategies for equational logic programs with extra variables in right-hand sides. For instance, completeness results for lazy narrowing strategies are only known for constructor-based normal 2-CTRS [18]. Our transformation method yields new completeness results for functional CTRS by applying Theorem 3.14 to the completeness result of lazy narrowing [18] for constructor-based almost orthogonal normal 2-CTRS.

Corollary 3.15 *Lazy narrowing is complete for constructor-based functional CTRS.*

To obtain a further interesting result, we apply Theorem 3.14 to inductively sequential systems with extra variables in right-hand sides. For this purpose, we use the same translation techniques as introduced in Section 3.5.1 and we immediately obtain the following proposition.

Corollary 3.16 *Let $\mathcal{R}$ be a functional CTRS such that the unconditional part $\mathcal{R}_u$ is inductively sequential.*[7] *Then needed narrowing is complete for $\mathcal{R}$, and it is an optimal strategy w.r.t. the length of the derivations and the number of computed solutions.*

Thus needed narrowing is a complete and optimal strategy for the programs in Examples 3.2 and 3.13.

Due to these results, it is no problem to extend equational logic languages like BABEL [18] or K-LEAF [10] by permitting extra variables in right-hand sides. However, the use of these extra variables must be restricted so that the programs are functional. The first two conditions of functional CTRS are easy to check, but the confluence condition 3 is usually hard to verify. In some cases it is possible to show confluence by proving that the rewrite system $\mathcal{R}$ is level-confluent, i.e., we may show that each unconditional rewrite system $\mathcal{R}_n$ is confluent for all $n \geq 0$. For instance, it is relatively easy to show that the rewrite system in Example 3.2 is level-confluent. However, from a practical point of view, it is desirable to have syntactic criteria to ensure the confluence of a 3-CTRS. Fortunately, for constructor-based programs there is an interesting subclass of functional CTRS which has a simple syntactic characterization.[8] Note that in constructor-based systems each conditional rule can be written in the form $l \rightarrow r \Leftarrow s \equiv t$.

[7] Since the property of inductive sequentiality depends only on the left-hand sides of the rewrite rules, the definition can simply be extended to rules with extra variables in right-hand sides.

[8] Recently, Suzuki et al. [21] have independently characterized a class of level-confluent 3-CTRS with similar restrictions.

Proposition 3.17 *A constructor-based normal CTRS $\mathcal{R}$ is functional if the following conditions holds:*

1. *The unconditional part $\mathcal{R}_u$ is almost orthogonal.*
2. *For each rule $l \to r \Leftarrow s \equiv t$ with extra variables in r, t is a constructor term, $\mathcal{V}ar(s) \subseteq \mathcal{V}ar(l)$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(t)$.*

As a consequence of this proposition, the rewrite system in Example 3.13 is functional. It is straightforward to refine the proposition to conditional rules with more than one strict equation in the condition part.

4 Conclusions

In this paper we have discussed the necessity and problems of extra variables in pure logic programming and equational logic programming. In the first part, we have shown that extra variables are unnecessary for pure logic programming since all occurrences of extra variables during a computation can be moved into the initial goal. Although this transformation does not change the declarative and operational semantics of pure logic programs, it does not generally work for equational logic programs, since it is known that the presence of extra variables may cause incompleteness of narrowing, the standard operational semantics of equational logic programs. Nevertheless, we have shown that this transformation works for the important subclass of almost orthogonal normal programs. As a consequence of this result, we have provided a general method to lift completeness results for narrowing without extra variables to programs with extra variables. Using this method, we could prove various new completeness results like completeness and optimality of needed narrowing and completeness of lazy narrowing in the presence of extra variables. Programs with such properties often occur if programming techniques like infinite data structures (e.g., streams) and *let* constructs from functional programming are simultaneously used. Therefore, our results are a contribution to extend current functional logic languages in a practically useful way, since such extensions give the programmer more expressivity and allow a more efficient execution of programs. Our method can also be helpful to simplify completeness proofs for possibly more sophisticated narrowing strategies that will be developed in the future.

Acknowledgements. The author is grateful to Aart Middeldorp and Enno Ohlebusch for their comments on this paper. The research described in this paper was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103 and by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics).

References

[1] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pp. 268–279, Portland, 1994.

[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] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term Graph Rewriting. In *Proc. Parallel Architectures and Languages Europe (PARLE'87)*, pp. 141–158. Springer LNCS 259, 1987.

[4] J.A. Bergstra and J.W. Klop. Conditional Rewrite Rules: Confluence and Termination. *Journal of Computer and System Sciences*, Vol. 32, No. 3, pp. 323–362, 1986.

[5] 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.

[6] 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.

[7] N. Dershowitz and M. Okada. A Rationale for Conditional Equational Programming. *Theoretical Computer Science*, Vol. 75, pp. 111–138, 1990.

[8] N. Dershowitz, M. Okada, and G. Sivakumar. Confluence of Conditional Rewrite Systems. In *Proc. 1st Int. Workshop on Conditional Term Rewriting Systems*, pp. 31–44. Springer LNCS 308, 1987.

[9] 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.

[10] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. *Journal of Computer and System Sciences*, Vol. 42, No. 2, pp. 139–185, 1991.

[11] 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.

[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. Combining Lazy Narrowing and Simplification. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pp. 370–384. Springer LNCS 844, 1994.

[14] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

[15] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

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

[17] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, Vol. 5, pp. 213–253, 1994.

[18] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The Language BABEL. *Journal of Logic Programming*, Vol. 12, pp. 191–223, 1992.

[19] P. Padawitz. Generic Induction Proofs. In *Proc. of the 3rd Intern. Workshop on Conditional Term Rewriting Systems*, pp. 175–197. Springer LNCS 656, 1992.

[20] M. Proietti and A. Pettorossi. Completeness of Some Transformation Strategies for Avoiding Unnecessary Logical Variables. In *Proc. Eleventh International Conference on Logic Programming*, pp. 714–729. MIT Press, 1994.

[21] T. Suzuki, A. Middeldorp, and T. Ida. Level-Confluence of Conditional Rewrite Systems with Extra Variables in Right-Hand Sides. Technical Report ISE-TR 94-116, Univ. of Tsukuba, 1994. To appear in Proc. RTA'95.