MAX-PLANCK-INSTITUT FÜR INFORMATIK

On Extra Variables in (Equational) Logic Programming

Michael Hanus

MPI–I–94–246 September 1994

Im Stadtwald
D 66123 Saarbrücken
Germany

Author's Address

Michael Hanus

Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
Germany
michael@mpi-sb.mpg.de

Acknowledgements

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). The responsibility for the contents of this publication lies with the author.

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. Moreover, we show a strong correspondence between the original and the transformed program with respect to the declarative and the operational semantics. In the second 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. Since extra variables are useful from a programming point of view, we characterize new classes of equational logic programs with extra variables for which narrowing and particular narrowing strategies are complete. 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.

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* [PP94], *local variables* [BMPT90], *right-free variables* [BMPT87], or *fresh variables* [Pad92]). It has been argued that extra variables are necessary and contribute to the expressive power of logic languages. For instance, Dershowitz and Okada [DO90] 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 prove 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 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 [Han94b] 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 [GM86]:

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

$$\texttt{a} \rightarrow \texttt{b}$$
$$\texttt{a} \rightarrow \texttt{c}$$
$$\texttt{b} \rightarrow \texttt{c} \Leftarrow \texttt{f(X,b)=f(c,X)}$$

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):

$$\underline{\texttt{b}}\texttt{=c} \leadsto \texttt{f(X,}\underline{\texttt{b}}\texttt{)=f(c,X), c=c} \leadsto \texttt{f(X1,}\underline{\texttt{b}}\texttt{)=f(c,X1), f(X,c)=f(c,X), c=c} \leadsto \cdots$$

In order to prove the condition of the last rule, the extra variable `X` must be instantiated to `a` and then 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 [BG89]. 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

[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 '$\rightarrow$' instead of the equality symbol in the head

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 weakly 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 [Llo87]. *Terms* are constructed from variables and function symbols,[2] 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 $\bar{t}$). 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 $\mathcal{V}ar(t)$ (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

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

A variable $x \in \mathcal{V}ar(C)$ is called *extra variable* if $x \notin \mathcal{V}ar(\bar{t})$. 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

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

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.[3] Moreover, $v_0, v_1, v_2, \ldots$ is a family of new function symbols not occurring in the original program. The following proposition is obvious.

Proposition 2.1 *If C is a clause, then $eev(C)$ is a clause without extra variables.*

We extend the transformation *eev* to programs by applying *eev* to each clause of the program.

Example 2.2 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)
```

□

[2] As usual, we assume that there is at least one 0-ary function symbol.

[3] 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.

In the following, we show 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. This implies a strong correspondence between validity w.r.t. P and $eev(P)$. However, there is a small problem in the comparison of the validity between both programs. Due to the introduction of the new function symbols $v_0, v_1, \ldots$, the languages of P and $eev(P)$ differ. Consequently, the classes of interpretations are different (in particular, the domains of the interpretations and the denotations of function symbols are different). However, the following proposition shows that the introduction of new function symbols does not influence the set of logical consequences.

Proposition 2.3 (Extended signatures) *Let $\Sigma = (\mathcal{F}, \mathcal{P})$ be a signature (i.e., $\mathcal{F}$ is a set of function symbols and $\mathcal{P}$ is a set of predicate symbols), $\Sigma' = (\mathcal{F}', \mathcal{P})$ with $\mathcal{F} \subseteq \mathcal{F}'$ be an extended signature, P be a Σ-program, and G be a Σ-goal. Then: G is Σ-valid w.r.t. P iff G is Σ'-valid w.r.t. P.*

Proof: "*if*": Let $I = (U, \delta)$ be a Σ-interpretation which is a model of P, i.e., U is the universe or domain of the interpretation and δ is a denotation for each function and predicate symbol such that δ_f is a mapping from U^n into U for each n-ary function symbol f and δ_p is a mapping from U^n into $\{true, false\}$ for each n-ary predicate symbol p. Since the language of P contains at least one constant symbol, $U \neq \emptyset$. Let $u \in U$ be some fixed element. We define a Σ'-interpretation $I' = (U, \delta')$ by $\delta'_f := \delta_f$ for all $f \in \mathcal{F}$, $\delta'_f(e_1, \ldots, e_n) := u$ for all n-ary functions $f \in \mathcal{F}' \backslash \mathcal{F}$, and $\delta'_p := \delta_p$ for all $p \in \mathcal{P}$. Since all clauses in P are Σ-clauses and I is a model of P, I' is also a model of P. Since G is Σ'-valid w.r.t. P, G is valid in I'. Thus G is also valid in I (note that each variable assignment in I is also a variable assignment in I').

"*only if*": Let $I' = (U', \delta')$ be a Σ'-interpretation which is a model of P. We define a Σ-interpretation $I = (U, \delta)$ by $U' := U$, $\delta_f := \delta'_f$ for all $f \in \mathcal{F}$ and $\delta_p := \delta'_p$ for all $p \in \mathcal{P}$. Clearly, I is a model of P since all clauses are valid in I. Hence G is valid in I, and, therefore, G is also valid in I' (note that each variable assignment in I' is also a variable assignment in I). ■

As a consequence of this proposition, we can avoid explicitly referring to the signature when we talk about validity. In the rest of this paper, we assume that the new function symbols $v_0, v_1, \ldots$ always belongs to our logic language (but, of course, they do not occur in the original program P).

The Herbrand base w.r.t. $eev(P)$ contains an additional argument for each predicate in comparison to the Herbrand base w.r.t. P. However, it can be shown that the initial models are equivalent if the additional arguments are 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

$$dla(H) := \{p(t_1, \ldots, t_n) \mid p(t_1, \ldots, t_n, t_{n+1}) \in H\} \ .$$

Now we can establish the precise correspondence between the declarative semantics of P and $eev(P)$:

Theorem 2.4 *Let H be the least Herbrand model of the logic program P, and H' be the least Herbrand model of the transformed program $P' := eev(P)$. Then $H = dla(H')$.*

Proof: The least Herbrand model H can be computed as the least fixpoint of the transformation T_P, i.e., $H = lfp(T_P) = T_P \uparrow \omega$, where T_P is the following transformation on Herbrand interpretations

[Llo87]:

$$T_P(I) := \{L \mid L \Leftarrow L_1, \ldots, L_k \text{ is a ground instance of a clause in } P \text{ with } \{L_1, \ldots, L_k\} \subseteq I\}$$

We define a sequence of Herbrand interpretations by $H_0 := \emptyset$ and $H_{i+1} := T_P(H_i)$ $(i \geq 0)$. We assume analogous definitions for H', $T_{P'}$ and H'_i. First, we show by induction on i: $H_i = dla(H'_i)$ for all $i \geq 0$.

Since the case $i = 0$ is trivial, consider $i > 0$: By induction hypothesis, $H_{i-1} = dla(H'_{i-1})$.

"$\subseteq$": Let $p(\bar{t}) \in H_i$. By definition of T_P, there is a ground instance $p(\bar{t}) \Leftarrow q_1(\overline{t_1}), \ldots, q_k(\overline{t_k})$ of a clause in P with $q_j(\overline{t_j}) \in H_{i-1}$ for $j = 1, \ldots, k$. Since $H_{i-1} = dla(H'_{i-1})$ and by definition of dla, there are ground terms e_j with $q_j(\overline{t_j}, e_j) \in H'_{i-1}$. By definition of eev, $p(\bar{t}, e) \Leftarrow q_1(\overline{t_1}, e_1), \ldots, q_k(\overline{t_k}, e_k)$ is a ground instance of the corresponding clause in P' for some ground term e. Hence $p(\bar{t}, e) \in H'_i$, and thus $p(\bar{t}) \in dla(H'_i)$. Therefore, $H_i \subseteq dla(H'_i)$.

"$\supseteq$": Let $p(\bar{t}) \in dla(H'_i)$. By definition of dla, there is a ground term e with $p(\bar{t}, e) \in H'_i$. By definition of $T_{P'}$, there is a ground instance $p(\bar{t}, e) \Leftarrow q_1(\overline{t_1}, e_1), \ldots, q_k(\overline{t_k}, e_k)$ of a clause in P' with $q_j(\overline{t_j}, e_j) \in H'_{i-1}$ for $j = 1, \ldots, k$. Since $H_{i-1} = dla(H'_{i-1})$ and by definition of dla, $q_j(\overline{t_j}) \in H_{i-1}$. By definition of eev, $p(\bar{t}) \Leftarrow q_1(\overline{t_1}), \ldots, q_k(\overline{t_k})$ is a ground instance of the corresponding clause in P. Hence $p(\bar{t}) \in H_i$ by definition of T_P. Therefore, $H_i \supseteq dla(H'_i)$.

Finally, we have to show: $H = dla(H')$.

"$\subseteq$": Let $p(\bar{t}) \in H$. Since $H = T_P \uparrow \omega$, $p(\bar{t}) \in H_n$ for some n. Since $H_n = dla(H'_n)$, there is a ground term e with $p(\bar{t}, e) \in H'_n$, which implies $p(\bar{t}, e) \in H'$. Thus $p(\bar{t}) \in dla(H')$, and $H \subseteq dla(H')$.

"$\supseteq$": Let $p(\bar{t}) \in dla(H')$. By definition of dla, there is a ground term e with $p(\bar{t}, e) \in H'$. Since $H' = T_{P'} \uparrow \omega$, $p(\bar{t}, e) \in H'_n$ for some n. Since $H_n = dla(H'_n)$, $p(\bar{t}) \in H_n$, which implies $p(\bar{t}) \in H$. Therefore, $H \supseteq dla(H')$. ■

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 $eev(P)$. In the following, we will show a similar property for the operational semantics. We consider SLD-resolution with the leftmost computation rule as the operational semantics. The commitment to the leftmost computation rule is for the sake of simplicity of the proofs, but these proofs can also be extended to an arbitrary computation rule. First, we show that each computed answer w.r.t. P is covered by a computed answer w.r.t. $eev(P)$.

Theorem 2.5 *Let P be a logic program, $G = p_1(\overline{t_1}), \ldots, p_k(\overline{t_k})$ be a goal and σ be a computed answer for G w.r.t. P. If $x_1, \ldots, x_k$ are new variables, 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. $eev(P)$.*

Proof: Since σ is a computed answer for G, there is a resolution sequence

$$G \vdash_{\sigma_1} G_1 \vdash_{\sigma_2} \cdots \vdash_{\sigma_n} G_n = \square \qquad (*)$$

w.r.t. P ($\square$ denotes the empty goal), where $\sigma_n \circ \cdots \circ \sigma_1(x) = \sigma(x)$ for all $x \in \mathcal{V}ar(G)$. We show that there is a resolution sequence

$$G' \vdash_{\sigma'_1} G'_1 \vdash_{\sigma'_2} \cdots \vdash_{\sigma'_n} G'_n = \square$$

w.r.t. $eev(P)$ with $\sigma'_i = \rho_i \circ \sigma_i$ and $dom(\rho_i) \cap (\mathcal{V}ar(G) \cup \mathcal{V}ar(G_1) \cup \cdots \cup \mathcal{V}ar(G_n)) = \emptyset$ $(i = 1, \ldots, n)$. We construct this sequence by induction on n.

$n = 1$: Then sequence $(*)$ has the form $p(\overline{t_1}) \vdash_{\sigma_1} \square$, and there is a clause $p(\overline{s}) \in P$ (or a variant if $\overline{s}$ and $\overline{t_1}$ have variables in common) so that σ_1 is an mgu for $\overline{t_1}$ and $\overline{s}$. By definition of eev, $p(\overline{s}, v_0) \in eev(P)$. If x_1 is a new variable, $\sigma'_1 := \{x_1 \mapsto v_0\} \circ \sigma_1$ is an mgu for $p(\overline{t_1}, x_1)$ and $p(\overline{s}, v_0)$, and $p(\overline{t_1}, x_1) \vdash_{\sigma'_1} \square$ is a resolution step. Thus the claim holds for $n = 1$.

$n > 1$: Let $p(\overline{s}) \Leftarrow q_1(\overline{s_1}), \ldots, q_l(\overline{s_l})$ be (the variant of) the clause used in the first resolution step in $(*)$, i.e., σ_1 is an mgu for $p_1(\overline{t_1})$ and $p(\overline{s})$ and $G_1 = \sigma_1(q_1(\overline{s_1}), \ldots, q_l(\overline{s_l}), p_2(\overline{t_2}), \ldots, p_k(\overline{t_k}))$. Then $p(\overline{s}, v_m(\ldots, y_1, \ldots, y_l)) \Leftarrow q_1(\overline{s_1}, y_1), \ldots, q_l(\overline{s_l}, y_l)$ is (a variant of) a clause in $eev(P)$, and $\sigma'_1 := \rho_1 \circ \sigma_1$ with $\rho_1 = \{x_1 \mapsto v_m(\ldots, y_1, \ldots, y_l)\}$ is an mgu for $p(\overline{s}, v_m(\ldots, y_1, \ldots, y_l))$ and $p_1(\overline{t_1}, x_1)$. Thus

$$G' \vdash_{\sigma'_1} G'_1 = \sigma'_1(q_1(\overline{s_1}, y_1), \ldots, q_l(\overline{s_l}, y_l), p_2(\overline{t_2}, x_2), \ldots, p_k(\overline{t_k}, x_k))$$

is a resolution step w.r.t. $eev(P)$. Moreover, $\sigma'_1(y_j) = y_j$ for $j = 1, \ldots, l$ and $\sigma'_1(x_j) = x_j$ for $j = 2, \ldots, k$. Therefore, we can apply the induction hypothesis to the remaining resolution sequence. Hence there is a resolution sequence

$$G'_1 \vdash_{\sigma'_2} G'_2 \vdash_{\sigma'_3} \cdots \vdash_{\sigma'_n} G'_n = \square$$

w.r.t. $eev(P)$ with $\sigma'_i = \rho_i \circ \sigma_i$ and $dom(\rho_i) \cap (\mathcal{V}ar(G) \cup \mathcal{V}ar(G_2) \cup \cdots \cup \mathcal{V}ar(G_n)) = \emptyset$ $(i = 2, \ldots, n)$. If we combine this sequence with the first step, we obtain a resolution sequence

$$G' \vdash_{\sigma'_1} G'_1 \vdash_{\sigma'_2} \cdots \vdash_{\sigma'_n} G'_n = \square$$

w.r.t. $eev(P)$. Since the variables in the applied clauses can be freely chosen, we can ensure that $dom(\rho_i) \cap (\mathcal{V}ar(G) \cup \mathcal{V}ar(G_1) \cup \cdots \cup \mathcal{V}ar(G_n)) = \emptyset$ $(i = 1, \ldots, n)$.

Altogether, $\sigma'_n \circ \cdots \circ \sigma'_1 = \rho_n \circ \sigma_n \circ \cdots \circ \rho_1 \circ \sigma_1 = \rho \circ \sigma_n \circ \cdots \circ \sigma_1$ for some substitution ρ with $\rho(x) = x$ for all $x \in \mathcal{V}ar(G) \cup \mathcal{V}ar(G_1) \cup \cdots \cup \mathcal{V}ar(G_n)$ (the last equality holds since the domains of all ρ_i have no variables in common with $\mathcal{V}ar(G) \cup \mathcal{V}ar(G_1) \cup \cdots \cup \mathcal{V}ar(G_n)$). Thus the computed answer of the constructed resolution sequence for G' is $\{x_1 \mapsto \rho(x_1), \ldots, x_k \mapsto \rho(x_k)\} \circ \sigma$. ■

The next theorem shows that the opposite direction of the previous theorem is also true.

Theorem 2.6 *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. 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.*

Proof: Since σ' is a computed answer for G, there is a resolution sequence

$$G' \vdash_{\sigma_1} G'_1 \vdash_{\sigma_2} \cdots \vdash_{\sigma_n} G'_n = \square \qquad (*)$$

w.r.t. $eev(P)$, where $\sigma'_n \circ \cdots \circ \sigma'_1(x) = \sigma'(x)$ for all $x \in \mathcal{V}ar(G')$. We show that there is a resolution sequence

$$G \vdash_{\sigma_1} G_1 \vdash_{\sigma_2} \cdots \vdash_{\sigma_n} G_n = \square$$

w.r.t. P with $\sigma'_i = \rho_i \circ \sigma_i$ for some substitution ρ_i and $dom(\rho_i) \cap (\mathcal{V}ar(G) \cup \mathcal{V}ar(G_1) \cup \cdots \cup \mathcal{V}ar(G_n)) = \emptyset$ $(i = 1, \ldots, n)$. We construct this sequence by induction on n.

$n = 1$**:** Then sequence $(*)$ has the form $p(\overline{t_1}, x_1) \vdash_{\sigma'_1} \square$, and there is a clause $p(\overline{s}, v_0) \in eev(P)$ (or a variant) so that σ'_1 is an mgu for $p(\overline{t_1}, x_1)$ and $p(\overline{s}, v_0)$. By definition of eev, $p(\overline{s}) \in P$. Since x_1 is a new variable, σ'_1 has the form $\{x_1 \mapsto v_0\} \circ \sigma_1$, where σ_1 is an mgu for $p(\overline{t_1})$ and $p(\overline{s})$. Thus $p(\overline{t_1}) \vdash_{\sigma_1} \square$ is a resolution step, and the claim holds for $n = 1$.

$n > 1$**:** Let $p(\overline{s}, v_m(\ldots, y_1, \ldots, y_l)) \Leftarrow q_1(\overline{s_1}, y_1), \ldots, q_l(\overline{s_l}, y_l)$ be (the variant of) the clause used in the first resolution step in $(*)$, i.e., σ'_1 is an mgu for $p_1(\overline{t_1}, x_1)$ and $p(\overline{s}, v_m(\ldots, y_1, \ldots, y_l))$ and $G'_1 = \sigma'_1(q_1(\overline{s_1}, y_1), \ldots, q_l(\overline{s_l}, y_l), p_2(\overline{t_2}, x_2), \ldots, p_k(\overline{t_k}, x_k))$. Then $p(\overline{s}) \Leftarrow q_1(\overline{s_1}), \ldots, q_l(\overline{s_l})$ is (a variant of) a clause in P, and σ'_1 has the form $\rho_1 \circ \sigma_1$, where $\rho_1 = \{x_1 \mapsto v_m(\ldots, y_1, \ldots, y_l)\}$ and σ_1 is an mgu for $p(\overline{s})$ and $p_1(\overline{t_1})$ (note that x_1 is a new variable). Thus

$$G \vdash_{\sigma_1} G_1 = \sigma_1(q_1(\overline{s_1}), \ldots, q_l(\overline{s_l}), p_2(\overline{t_2}), \ldots, p_k(\overline{t_k}))$$

is a resolution step w.r.t. P. Note that $\sigma'_1(y_j) = y_j$ for $j = 1, \ldots, l$ and $\sigma'_1(x_j) = x_j$ for $j = 2, \ldots, k$. Therefore, we can apply the induction hypothesis to the remaining resolution sequence. Hence there is a resolution sequence

$$G_1 \vdash_{\sigma_2} G_2 \vdash_{\sigma_3} \cdots \vdash_{\sigma_n} G_n = \square$$

w.r.t. P with $\sigma'_i = \rho_i \circ \sigma_i$ and $dom(\rho_i) \cap (\mathcal{V}ar(G) \cup \mathcal{V}ar(G_2) \cup \cdots \cup \mathcal{V}ar(G_n)) = \emptyset$ $(i = 2, \ldots, n)$. If we combine this sequence with the first step, we obtain a resolution sequence

$$G \vdash_{\sigma_1} G_1 \vdash_{\sigma_2} \cdots \vdash_{\sigma_n} G_n = \square$$

w.r.t. P. Since the variables in the applied clauses can be freely chosen, we can ensure that $dom(\rho_i) \cap (\mathcal{V}ar(G) \cup \mathcal{V}ar(G_1) \cup \cdots \cup \mathcal{V}ar(G_n)) = \emptyset$ $(i = 1, \ldots, n)$.

Since the domains of all ρ_i have no variables in common with $\mathcal{V}ar(G) \cup \mathcal{V}ar(G_1) \cup \cdots \cup \mathcal{V}ar(G_n)$, $\sigma'_n \circ \cdots \circ \sigma'_1(x) = \sigma_n \circ \cdots \circ \sigma_1(x)$ for all variables $x \in \mathcal{V}ar(G)$. This implies the claim. ■

The proofs of Theorem 2.5 and 2.6 show 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.*

Application of the transformation *eev*

The purpose of the transformation *eev* was to show that all extra variables can be eliminated in logic programs. Although this seems to be only of theoretical interest, there is an interesting application of this transformation in equational logic programming. Equational logic programming is a basis for integrating functional and logic programming languages [Han94b], since it permits the definition of predicates by Horn clauses and the definition of functions by (conditional) equations. The standard operational semantics for equational logic programs is narrowing, a combination of term reduction and resolution (see next section for details). Completeness results for narrowing strategies are often stated under the assumption that no extra variables occur in conditions. Therefore, it is sometimes argued that equational logic programming is less powerful than logic programming due to these

restrictions. The results in this section show that this is wrong, since it is possible to eliminate all extra variables in a logic program, and then represent all predicates as Boolean functions.

We do not elaborate this idea, since such a translation method does not exploit the power of equational logic programming. One reason to use extra variables in logic programming is the missing ability to write nested function calls. For instance, if the predicate `append` defined in Example 2.2 is given, the following clause defines a predicate `conc3` to concatenate three lists:

```
conc3(L1,L2,L3,L) ⇐ append(L1,L2,M), append(M,L3,L)
```

The extra variable `M` in this clause is introduced because we cannot write nested function calls. However, in an equational logic language, we define `conc3` as a function by the following equation (provided that `append` is also defined as a function from two lists into a list):

```
conc3(L1,L2,L3) = append(append(L1,L2),L3)
```

Therefore, it is better to use directly an equational logic language instead of transforming logic programs by representing predicates as Boolean functions (other advantages of equational logic programming in comparison to pure logic programming are discussed in [Han92]). However, extra variables are also a useful feature in equational logic programming. Due to the incompleteness of some narrowing strategies in the presence of extra variables, it seems that there is no simple way to avoid extra variables in equational logic programs. In the next section, we show how to eliminate extra variables similarly to pure logic programs, and we discuss classes of programs where this method yields interesting new results.

3 Extra Variables in Equational Logic Programming

Equational logic programming (see [Han94b] 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, they are considered as syntactic sugar for the sake of simplicity. Therefore, we assume that all clauses in an equational logic program have the form

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

($n \geq 0$), 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 $n = 0$. 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 [BK86, DJ90].

Substitutions and most general unifiers are defined as in logic programming [Llo87]. 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 [DJ90] for details).

Let $\rightarrow$ be a binary relation on a set S. Then $\rightarrow^*$ denotes the transitive and reflexive closure of the relation $\rightarrow$, and $\leftrightarrow^*$ denotes the transitive, reflexive and symmetric closure of $\rightarrow$. $\rightarrow$ is called *terminating* if there are no infinite chains $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow \cdots$. We write $e_1 \downarrow e_2$ if there exists an element $e_3 \in S$ with $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$. $\rightarrow$ is called *confluent* if $e_1 \downarrow e_2$ for all $e, e_1, e_2 \in S$ with $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$.

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

The confluence of the rewrite relation $\rightarrow_{\mathcal{R}}$ is a basic requirement to apply rules only in one direction during equational reasoning. In order to ensure confluence even for nonterminating rewrite systems,[4] we need some syntactical restrictions on the rewrite rules. A rewrite rule $l \rightarrow r$ is called *left-linear* if there are no multiple occurrences of the same variable in l. Two rewrite rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ (with disjoint variables) have a *critical pair* $\langle \sigma(r_2), \sigma(l_2[r_1]_p) \rangle$ if σ is an mgu for l_1 and the nonvariable subterm $l_2|_p$ (in case of $p = \Lambda$ we additionally require that $l_1 \rightarrow r_1$ is not a variant of $l_2 \rightarrow r_2$). An unconditional term rewriting system $\mathcal{R}$ is called *orthogonal* if all rules in $\mathcal{R}$ are left-linear and there are no critical pairs between variants of rules in $\mathcal{R}$. $\mathcal{R}$ is called *weakly orthogonal* if all rules in $\mathcal{R}$ are left-linear and all critical pairs between variants of rules in $\mathcal{R}$ are trivial, i.e., they describe an overlap at the root position $p = \Lambda$ and have the form $\langle t, t \rangle$.

An important property of weakly orthogonal systems is the confluence of the rewrite relation (see [Klo92] for a comprehensive survey on results for orthogonal systems).

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

3.2 Equational Logic Programs

The computation mechanism of unconditional term rewrite systems was defined by the rewrite relation $\rightarrow_{\mathcal{R}}$ 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 [BK86, DO90], we can distinguish the following possibilities. A condition $s_1 = t_1, \ldots, s_k = t_k$ is satisfied if

(i) (*semi-equational systems*) $s_1 \leftrightarrow^* t_1, \ldots, s_k \leftrightarrow^* t_k$ (i.e., the left-hand side of each condition can be converted into the right-hand side by equational reasoning),

(ii) (*join systems*) $s_1 \downarrow t_1, \ldots, s_k \downarrow t_k$ (i.e., the left- and right-hand side of each condition can be reduced to a same term), or

(iii) (*normal systems*) $s_1 \rightarrow^* t_1, \ldots, s_k \rightarrow^* t_k$ (i.e., the left-hand side of each condition is reducible to the right-hand side), where $t_1, \ldots, t_k$ are ground normal forms w.r.t. the unconditional part of the CTRS.

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

Note that all three definitions of conditional rewriting are recursive, but we can provide an iterative definition (this is only done for the case (ii) but it is obvious to define the other cases). Let $\mathcal{R}$ be a CTRS. We inductively define the following unconditional term rewriting systems $\mathcal{R}_n$ ($n \geq 0$) by:

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

Note that $\mathcal{R}_n \subseteq \mathcal{R}_{n+1}$ for all $n \geq 0$. We have $s \to_{\mathcal{R}} t$ iff $s \to_{\mathcal{R}_n} t$ for some $n \geq 0$. The minimal n is called the *depth* of the rewrite step $s \to_{\mathcal{R}} t$.

Semi-equational systems have a complex proof procedure for conditions. The rewrite relation of join systems may not be confluent even for orthogonal CTRS. This can be shown by the following example [BK86].

Example 3.1 Consider the rewrite rules:

```
f(X) → a ⇐ X ↓ f(X)
   b → f(b)
```

Now `f(f(b))` can be rewritten to `a` (since `f(b)`$\downarrow$`f(f(b))`) and to `f(a)` (since `b`$\downarrow$`f(b)`). However, `a`$\downarrow$`f(a)` does not hold. □

A similar example shows that the rewrite relation of normal systems may not be confluent if we allow nonground t_i's in the condition [BK86]. Fortunately, this cannot occur for normal systems [Klo92]:

Theorem 3.2 *The rewrite relation of a weakly 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 [MR92] and K-LEAF [GLMP91]). This is not a restriction from a logic programming point of view, since each logic program can be transformed into a weakly 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 [MR92] 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* [Han90], *BABEL* [MR92], *K-LEAF* [GLMP91], *LPG* [BE86], or *SLOG* [Fri85]). 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 [GLMP91, MR92]):

$$\begin{array}{rcll} c \equiv c & \to & true & \text{for all 0-ary constructors } c \\ c(x_1,\ldots,x_n) \equiv c(y_1,\ldots,y_n) & \to & x_1 \equiv y_1 \wedge \cdots \wedge x_n \equiv y_n & \text{for all } n\text{-ary constructors } c \\ c(x_1,\ldots,x_n) \equiv d(y_1,\ldots,y_m) & \to & false & \text{for all constructors } c \neq d \text{ or } n \neq m \\ true \wedge x & \to & x & \\ false \wedge x & \to & false & \end{array}$$

The reduction of $s \equiv t$ to $true$ is equivalent to the reduction of s and t to a same ground constructor term ([AEH94], 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 [MH94]:

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 3.3 The equational logic program in Example 1.1 is a 2-CTRS, and the following equational version of Example 2.2 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 $G_1 \leadsto_{\sigma_1} G_2 \leadsto_{\sigma_2} \cdots \leadsto_{\sigma_{n-1}} G_n$, we write $G_1 \leadsto^*_\sigma G_n$ 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 $t_1 = t_1, \ldots, t_k = t_k$.

Although we have defined narrowing steps on normal equational goals, it should be clear how to extend narrowing steps to other classes of equational programs (e.g., join systems). The only difference is due to the fact that the right-hand sides t_i need not be ground in other classes of programs. Therefore, it is necessary to introduce narrowing steps with the rule $x = x \to \mathit{true}$ and consider only goals of the form $\mathit{true}, \ldots, \mathit{true}$ as trivial. When we discuss completeness results for narrowing strategies w.r.t. other classes of programs, we assume this extended definition of narrowing.

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. The declarative semantics of pure logic programs is described by the notions of the least Herbrand model and logical consequences. In equational logic programming the declarative semantics of a program is described as the set of all equalities which are derivable by equational reasoning (see, for instance, [GM87]). If the rewrite relation of a term rewriting system is confluent (as usually required for equational logic programs), then the application of rewrite steps is equivalent to equational reasoning. Therefore, we use the following specialized definition of completeness. 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 $G \leadsto^*_{\sigma'} G'$, 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 [MH94] for a comprehensive survey). Hussmann [Hus85] and Kaplan [Kap87] showed completeness of narrowing for confluent and terminating 1-CTRS. However, Example 1.1 shows that narrowing is incomplete for confluent and terminating 2-CTRS. In order to ensure completeness in the presence of extra variables, Giovannetti and Moiso [GM86] proposed the notion of level-confluence. A CTRS $\mathcal{R}$ is *level-confluent* if each rewrite relation $\to_{\mathcal{R}_n}$ $(n \geq 0)$ is confluent. For instance, weakly orthogonal normal 2-CTRS are level-confluent. Narrowing is complete for level-confluent and terminating 2-CTRS [GM86] and 3-CTRS [MH94].

The completeness results for narrowing defined so far are more or less of theoretical interest if we want to use it as the operational semantics of equational logic programs. Since the position where the next narrowing step is applied is not fixed, simple narrowing as defined above has a huge search space. Therefore, refined *narrowing strategies* which restrict the number of possible narrowing derivations are needed. The development of good strategies that do not destroy completeness was an active research topic during the last decade (see [Han94b] for a detailed survey). We discuss only the most important strategies.

Basic narrowing [Hul80] and its refinement [BKW92] reduces the possible narrowing positions by disregarding narrowing steps in positions introduced by substitutions in previous steps. *Innermost narrowing* [Fri85] selects an innermost position for the next narrowing step, i.e., a function call is evaluated by narrowing only if all its arguments were completely evaluated before. Alternatively, *outermost narrowing* [Ech88] selects an outermost position for the next narrowing step. All these strategies are complete under additional requirements. The most important requirement is the termination of the rewrite relation. However, termination is difficult to check due its undecidability,

and such a requirement excludes important functional programming techniques like programming with infinite data structures. Therefore, we are mainly interested in strategies which do not require termination. For this purpose, lazy evaluation strategies have been proposed for languages like BABEL [MR92] and K-LEAF [GLMP91]. In order to ensure the confluence of the rewrite relation, these languages are based on weakly orthogonal normal CTRS (where the non-overlapping requirement is slightly weakened in BABEL). It is well-known that *lazy narrowing* is complete for weakly orthogonal normal 2-CTRS, 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 [MR92] for details). 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* [AEH94] is an optimal strategy for inductively sequential programs, which is a subclass of unconditional orthogonal programs, and for weakly orthogonal programs it has been shown that the combination of lazy narrowing with intermediate simplification steps yields a better behavior [Han94a]. 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 list 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 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 $\{x_1, \ldots, x_n\} = (\mathcal{V}ar(\widehat{r}) \cup \mathcal{V}ar(\widehat{C}))\backslash\mathcal{V}ar(\widehat{\overline{t}})$.[5] 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. Similarly to the transformation for pure logic programs, we have the following property.

Proposition 3.4 *If R is a conditional rewrite rule, then $eev(R)$ is a conditional rewrite rule without extra variables.*

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 3.5 Let $\mathcal{R}$ be the equational logic program of Example 1.1 (although it is not a normal CTRS, it can be transformed into a normal system by replacing "=" by "≡" in the condition). Then $eev(\mathcal{R})$ is the following program:

$$\begin{array}{lll} \texttt{a}(v_1\texttt{(Y)}) & \to \texttt{b(Y)} & \\ \texttt{a}(v_0) & \to \texttt{c} & \\ \texttt{b}(v_2\texttt{(X,Z)}) & \to \texttt{c} & \Leftarrow \texttt{f(X,b(Z))=f(c,X)} \end{array}$$

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 programs (cf. Theorem 2.4), 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 3.5, since this would require an equality between `a(`v_0`)` and some instance of `a(`v_1`(Y))`. A deeper analysis of this example shows that in the original program the term `a` can be rewritten to the terms `b` and `c`, which is necessary

[5] 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 weakly orthogonal if the original programs are weakly orthogonal (see Proposition 3.14). Therefore, we fix the same ordering principle for all rules. A possible choice is a left-to-right innermost ordering for all variables in $\widehat{r}, \widehat{C}$.

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` (only `a`(v_1`(Y))` is reducible to `b(Y)` and `a`(v_0) is reducible to `c`).

Hence we can see that 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 (it is interesting to note that a similar phenomenon is the reason for the divergence in Example 3.1). 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 [BvEG$^+$87]. In order to avoid repeating all details of term graph rewriting, we assume familiarity with graphs to represent shared subterms (see [BvEG$^+$87] for details). We only cite the following result, which is important in our framework.

Theorem 3.6 ([BvEG$^+$87]) *If $\mathcal{R}$ is an unconditional weakly 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 weakly orthogonal systems is essential. Otherwise, rewriting with sharing is incomplete as the following example shows.

Example 3.7 Consider the following rewrite rules [BvEG$^+$87]:

```
g(X)   → f(X,X)        a → b
f(a,b) → c             b → a
```

Then there is the following reduction of `g(a)` to the normal form `c` in the traditional term rewriting sense (without sharing):

```
g(a) → f(a,a) → f(a,b) → c
```

However, if rule variables are shared, both occurrences of `a` in the derived term `f(a,a)` are identical. Therefore, a rewrite step applied to one of these occurrences also replaces the other occurrence. Consequently, there is only the following sequence with sharing:

```
g(a) → f(a,a) → f(b,b) → f(a,a) → ···
```

Hence rewriting with sharing fails to compute the normal form `c`. □

To apply the result of Theorem 3.6 in our framework, we have to extend it to conditional rewrite systems. Example 3.1 shows that this is not possible in general: there, `f(b)` is reducible to `a` in the traditional sense, but `f(b)` is not reducible if all occurrences of variable `X` in the rule are shared. Fortunately, 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.

For the sake of simplicity, we assume in subsequent proofs that all conditional rules have the form $l \to r \Leftarrow s = t$. This is not a restriction, since this form can be obtained by joining all equations of the condition into one equation.

Theorem 3.8 *Let $\mathcal{R}$ be a weakly orthogonal normal CTRS (with extra variables). Then all variables in rewrite rules can be shared during the computation of a normal form.*

Proof: By induction on the sum d of the depths of all rewrite steps (including rewrite steps in conditions) in the derivation $t \to^*_{\mathcal{R}} t_0$ (where t_0 is a normal form).

$d = 0$**:** Then the normal form is computed without a conditional rule, i.e., in all rewrite steps a rule from $\mathcal{R}_0$ is applied. Since $\mathcal{R}$ is weakly orthogonal, $\mathcal{R}_0$ is a weakly orthogonal unconditional rewrite system. Thus all rule variables in the derivation can be shared by Theorem 3.6.

$d > 1$**:** W.l.o.g. assume that the first step is a conditional rewrite step, i.e., there is an application of a conditional rule R: $l \to r \Leftarrow s = v \in \mathcal{R}$ to term t at position p with substitution σ. Since $\mathcal{R}$ is a normal CTRS, v is a ground normal form w.r.t. $\mathcal{R}_u$ and $\sigma(s) \to^*_{\mathcal{R}} v$. Let $x_1, \ldots, x_n$ be all variables occurring in R. Let f' and $cond\_f$ be new function symbols. Then add the following unconditional orthogonal rules to $\mathcal{R}$:

$$\begin{array}{rcl} f'(x_1, \ldots, x_n) & \to & cond\_f(s, r) \\ cond\_f(v, x) & \to & x \end{array}$$

If the first rewrite step with rule R has depth d_1, $t \to_{\mathcal{R}_{d_1}} t[\sigma(r)]_p \to^*_{\mathcal{R}} t_0$ with $\sigma(s) \to^*_{\mathcal{R}_{d_1-1}} v$. Consider the modified term $t' := t[\sigma(f'(x_1, \ldots, x_n))]_p$. Then there is the following derivation:

$$\begin{array}{rll} t' & \to_{\mathcal{R}_0} & t[cond\_f(\sigma(s), \sigma(r))]_p \\ & \to^*_{\mathcal{R}_{d_1-1}} & t[cond\_f(v, \sigma(r))]_p \\ & \to_{\mathcal{R}_0} & t[\sigma(r)]_p \\ & \to^*_{\mathcal{R}} & t_0 \end{array} \qquad (*)$$

The sum of the depths of this derivation is smaller than the sum of the depths of the original derivation $t \to^*_{\mathcal{R}} t_0$. By the induction hypothesis, all variables in this derivation can be shared, in particular, the variables $x_1, \ldots, x_n$. Since the derivation of the condition and the right-hand side of rule R is identical to the derivation $(*)$, all variables in rule R in the original derivation can also be shared. ■

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 introduce a mapping dv which deletes all extension arguments in terms:

$$\begin{array}{rcll} dv(x) & = & x & \text{for all variables } x \\ dv(f(t_1, \ldots, t_n, t_{n+1})) & = & f(dv(t_1), \ldots, dv(t_n)) & \end{array}$$

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 a weakly orthogonal normal CTRS.

Theorem 3.9 *Let $\mathcal{R}$ be a weakly 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 $t' \to^*_{\mathcal{R}'} \widehat{s}$.*

Proof: By induction on the number n of rewrite steps (including rewrite steps in conditions) in the derivation $t \to^*_{\mathcal{R}} s$.

$n = 0$**:** Then $t = s$. For $t' := \widehat{s}$ we have $t' \to^*_{\mathcal{R}'} \widehat{s}$ and $dv(t') = s = t$.

$n > 0$**:** Consider the first step of the derivation. There is a rewrite rule R: $l \to r \Leftarrow u = u_0$ (where u_0 is a ground normal form w.r.t. $\mathcal{R}_u$),[6] a position p in t and a substitution σ such that $t \to_{\mathcal{R}} t[\sigma(r)]_p \to^*_{\mathcal{R}} s$ and $\sigma(u) \to^*_{\mathcal{R}} u_0$. By induction hypothesis, there are extended terms t'_1, u' with $dv(t'_1) = t[\sigma(r)]_p$, $dv(u') = \sigma(u)$ and $t'_1 \to^*_{\mathcal{R}'} \widehat{s}$, $u' \to^*_{\mathcal{R}'} \widehat{u_0}$. Consider the transformed rule

$$eev(R)\colon f(\widehat{\overline{t}}, v_n(x_1, \ldots, x_n)) \to \widehat{r} \Leftarrow \widehat{u} = \widehat{u_0}$$

(provided that $l = f(\overline{t})$). By Theorem 3.8, all variables of rule R can be shared. Therefore, they are also replaced by identical terms in t'_1 and u'. This implies the existence of a substitution σ' with $t'_1|_p = \sigma'(\widehat{r})$ and $u' = \sigma'(\widehat{u})$, where the only difference between σ and σ' is the fact that σ' additionally instantiates the extension variables in $\widehat{r}$ and $\widehat{u}$.

Let $t|_p = f(\overline{v})$ (i.e., $\overline{v} = \sigma(\overline{t})$). Then $\widehat{t}|_p = f(\widehat{\overline{v}}, x)$ for some new variable x. Since the extension variables in $\widehat{\overline{t}}$ are disjoint from the extension variables in $\widehat{r}$ and $\widehat{u}$, $\sigma'(\widehat{\overline{t}}) = \sigma(\widehat{\overline{t}})$. Moreover, all extension variables in $f(\widehat{\overline{v}}, x)$ are different. Hence there exists a substitution ρ for the extension variables in $f(\widehat{\overline{v}}, x)$ with $\rho(f(\widehat{\overline{v}}, x)) = \sigma'(f(\widehat{\overline{t}}, v_n(x_1, \ldots, x_n)))$ (note that $\overline{t} = \sigma(\overline{t}) = \sigma'(\overline{t})$), in particular, $\rho(x) = v_n(\sigma'(x_1), \ldots, \sigma'(x_n))$. Let $t_0 = \widehat{t}[\rho(f(\widehat{\overline{v}}, x))]_p$. Clearly, $dv(t_0) = t[dv(\rho(f(\widehat{\overline{v}}, x)))]_p = t[f(\overline{v})]_p = t$ and $eev(R)$ is applicable to t_0 at position p, since $t_0|_p = \rho(f(\widehat{\overline{v}}, x)) = \sigma'(f(\widehat{\overline{t}}, v_n(x_1, \ldots, x_n)))$ and $\sigma'(\widehat{u}) = u' \to^*_{\mathcal{R}'} \widehat{u_0}$:

$$t_0 \quad \to_{\mathcal{R}'} \quad t_0[\sigma'(\widehat{r})]_p = \widehat{t}[\sigma'(\widehat{r})]_p =: t''_1$$

Now we have:

1. $dv(t''_1) = t[\sigma(r)]_p = dv(t'_1)$, i.e., the only difference between t''_1 and t'_1 is the instantiation of extension arguments.
2. $t''_1|_p = \sigma'(\widehat{r}) = t'_1|_p$

Moreover, $t''_1 = \widehat{t}[\sigma'(\widehat{r})]_p$, i.e., all extension arguments not in the subterm at position p are different new variables. This implies the existence of a substitution τ for these extension variables with $\tau(t''_1) = t'_1$. Since τ influences only variables outside the subterm at position p, the first rewrite step is also applicable to $t' := \tau(t_0)$:

$$\tau(t_0) \quad \to_{\mathcal{R}'} \quad \tau(t_0)[\sigma'(\widehat{r})]_p = t'_1$$

Moreover, $dv(t') = dv(\tau(t_0)) = t$ and $t'_1 \to^*_{\mathcal{R}'} \widehat{s}$. Hence $t' \to^*_{\mathcal{R}'} \widehat{s}$, which proves the claim. ■

This theorem implies that all strict equalities w.r.t. $\mathcal{R}$ are also valid w.r.t. $eev(\mathcal{R})$.

[6] An unconditional rule $l \to r$ is considered as a conditional rule $l \to r \Leftarrow \mathit{true} = \mathit{true}$ in order to avoid an additional case distinction.

Corollary 3.10 *Let $\mathcal{R}$ be a constructor-based weakly orthogonal normal CTRS. If the strict equality $t \equiv t'$ is valid w.r.t. $\mathcal{R}$, then there are terms u, u' with $dv(u) = t$, $dv(u') = t'$ so that $u \equiv u'$ is valid w.r.t. $eev(\mathcal{R})$.*

Proof: Let $\mathcal{R}' = eev(\mathcal{R})$. If the strict equality $t \equiv t'$ is valid w.r.t. $\mathcal{R}$, there is a ground constructor term s with $t \to^*_{\mathcal{R}} s$ and $t' \to^*_{\mathcal{R}} s$. By Theorem 3.9, there are terms u, u' with $dv(u) = t$, $dv(u') = t'$ so that $u \to^*_{\mathcal{R}'} \widehat{s}$ and $u' \to^*_{\mathcal{R}'} \widehat{s}$. By replacing all extension arguments of $\widehat{s}$ by arbitrary ground constructor terms in these derivations, this implies the validity of the strict equality $u \equiv u'$ w.r.t. $\mathcal{R}'$. ■

We do not explicitly prove the converse of this corollary, since this is a consequence of the fact that each narrowing derivation w.r.t. $eev(\mathcal{R})$ corresponds to a narrowing derivation w.r.t. $\mathcal{R}$. In order to prove this fact, we need two auxiliary propositions. The first lemma shows a relationship between the function dv and substitutions. In this lemma it is assumed that dv is extended on substitutions σ by $dv(\sigma)(x) := dv(\sigma(x))$.

Lemma 3.11 *Let t be an extended term and σ be a substitution on extended terms. Then $dv(\sigma(t)) = dv(\sigma)(dv(t))$.*

Proof: By structural induction on t. Let $\sigma' := dv(\sigma)$.

$t = x$**:** By definition, $\sigma'(x) = dv(\sigma(x))$. Hence $dv(\sigma(x)) = \sigma'(x) = \sigma'(dv(x))$.

$t = f(t_1, \ldots, t_n, v)$ $(n \geq 0)$**:** In this case we have

$$\begin{array}{rcll} dv(\sigma(t)) & = & dv(f(\sigma(t_1), \ldots, \sigma(t_n), \sigma(v))) & \\ & = & f(dv(\sigma(t_1)), \ldots, dv(\sigma(t_n))) & \text{(by definition of } dv\text{)} \\ & = & f(\sigma'(dv(t_1)), \ldots, \sigma'(dv(t_n))) & \text{(by induction hypothesis)} \\ & = & \sigma'(f(dv(t_1), \ldots, dv(t_n))) & \\ & = & \sigma'(dv(t)) & \text{(by definition of } dv\text{)} \end{array}$$

■

The following important theorem relates most general unifiers for extended terms to most general unifiers for their non-extended equivalents.

Theorem 3.12 *Let $\widehat{u_1}, \widehat{u_2}$ be extended terms which are variable disjoint, and θ be a most general unifier for $\widehat{u_1}$ and $\widehat{u_2}$. Then $dv(\theta)|_V$ with $V = \mathcal{V}ar(u_1) \cup \mathcal{V}ar(u_2)$ is a most general unifier for u_1 and u_2.*

Proof: Martelli and Montanari [MM82] showed that most general unifiers can be computed (up to variable renaming) by applying the following transformation rules to a system of equations (we omit the failure rules since we will apply these rules to unifiable terms):

Delete: $\dfrac{x = x, E}{E}$

Decompose: $\dfrac{f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n), E}{s_1 = t_1, \ldots, s_n = t_n, E}$

Commute: $\dfrac{t = x, E}{x = t, E}$ if t is not a variable

Instantiate: $\dfrac{x = t, E}{x = t, \sigma(E)}$ if $x \in \mathcal{V}ar(E) \backslash \mathcal{V}ar(t)$ and $\sigma = \{x \mapsto t\}$

We denote by $E \Rightarrow E'$ an application of one of these transformation steps. In order to unify two terms s and t, these rules are applied to the initial system $s = t$.

Let EV be the set of extension variables, i.e., $EV := (\mathcal{V}ar(\widehat{u_1}) \cup \mathcal{V}ar(\widehat{u_2})) \backslash V$. First, we prove the following invariants for all equation systems E which are derived by applying transformation rules starting from the initial system $\widehat{u_1} = \widehat{u_2}$:

1. If $s = t \in E$, then all extension arguments in s and t are variables disjoint from V, and $t_1, \ldots, t_{n-1} \notin EV$ for all subterms $f(t_1, \ldots, t_n)$ of s and t.
2. For all $x = t \in E$ or $t = x \in E$:
   (a) If $x \in EV$, then $t \in EV$.
   (b) If $x \in V$, then $t \notin EV$.

We prove these invariants by induction on the number k of transformation steps:

$k = 0$**:** This case is trivial, since it is easy to check that all invariants hold for the initial equation $\widehat{u_1} = \widehat{u_2}$.

$k > 0$**:** Assume that the invariants hold for the equation system E and $E \Rightarrow E'$. We prove the invariant for E' by a case distinction on the transformation rules:

Delete: The invariants trivially hold by the induction hypothesis since an equation is deleted in E'.

Decompose: Then E has the form $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n), E_0$ and E' has the form $s_1 = t_1, \ldots, s_n = t_n, E_0$. Invariant 1 holds for all new equations, since it holds for the equation $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \in E$. Furthermore, $s_1, t_1, \ldots, s_{n-1}, t_{n-1} \notin EV$ and $s_n, t_n \in EV$ by invariant 1 for E. Thus invariant 2 also holds for all new equations in E'.

Commute: The invariants trivially hold for E' since they hold for E.

Instantiate: Then E has the form $x = t, E_0$ and E' has the form $x = t, \sigma(E_0)$ with $\sigma = \{x \mapsto t\}$.

If $x \in V$, then $t \notin EV$ by invariant 2 for E. Invariant 1 for E implies that extension arguments are not altered by σ. Thus invariant 1 holds for E'. Since $t \notin EV$, invariant 2 also holds.

If $x \notin V$, then $t \in EV$ by invariant 2 for E, i.e., σ replaces an extension variable by another extension variable. Hence invariants 1 and 2 also hold for E'.

Next we show: If $E_1 \Rightarrow E_2 \Rightarrow \cdots$ is a sequence of transformation steps with $E_1 = \{\widehat{u_1} = \widehat{u_2}\}$, then $E'_1 \Rightarrow^= E'_2 \Rightarrow^= \cdots$, where $\Rightarrow^=$ is the reflexive closure of $\Rightarrow$, and

$$E'_i := \{dv(s) = dv(t) \mid s = t \in E_i \text{ and } s, t \notin EV\} \ .$$

We prove this claim by showing that either $E'_{k+1} = E'_k$ or $E'_k \Rightarrow E'_{k+1}$ is a valid transformation step. The proof is done by a case distinction on the transformation applied in step $E_k \Rightarrow E_{k+1}$.

Delete: Then E_k has the form $x = x, E$. If $x \in V$, then $x = x \in E'_k$ and this step is also a valid transformation step on E'_k. If $x \notin V$, then $x = x \notin E'_k$ and $E'_k = E'_{k+1}$.

Decompose: Then E_k has the form $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n), E$ and $f(dv(s_1), \ldots, dv(s_{n-1})) = f(dv(t_1), \ldots, dv(t_{n-1})) \in E'_k$. By invariant 1 for E_k, $s_1, t_1, \ldots, s_{n-1}, t_{n-1} \notin EV$ and $s_n, t_n \in EV$. Thus $dv(s_1) = dv(t_1), \ldots, dv(s_{n-1}) = dv(t_{n-1}) \in E'_{k+1}$, i.e., $E'_k \Rightarrow E'_{k+1}$ is a valid decomposition step.

Commute: Trivial.

Instantiate: Then E_k has the form $x = t, E$ and $\sigma = \{x \mapsto t\}$.

If $x \notin V$, then $t \in EV$ by invariant 2 for E_k, i.e., $x = dv(t)$ do not occur in E'_k and E'_{k+1}, and σ replaces an extension variable by another extension variable which occur only in extension arguments by invariant 1 for E_k. Thus $E'_k = E'_{k+1}$.

If $x \in V$, then $t \notin EV$ by invariant 2 for E_k. Thus E'_k has the form $x = dv(t), E'$, and E'_{k+1} contain the same set of equations (except for the application of σ). Thus there is a subset E_0 of E with $E' = dv(E_0)$, and E'_{k+1} has the form $x = dv(t), dv(\sigma(E_0))$. By a straightforward extension of Lemma 3.11 to equation systems, $dv(\sigma(E_0)) = dv(\sigma)(dv(E_0))$ and $dv(\sigma) = \{x \mapsto dv(t)\}$. Hence $E'_k \Rightarrow E'_{k+1}$ is a valid transformation.

Martelli and Montanari [MM82] showed that the mgu θ can be computed (up to variable renaming) by repeated application of the transformation $\Rightarrow$ to the initial equation $\widehat{u_1} = \widehat{u_2}$ until a solved form $x_1 = s_1, \ldots, x_n = s_n$ (where all x_i occur only once) is obtained. Then $\theta = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}$. We have shown that there is also a valid transformation sequence from $u_1 = u_2$ into $\{x_i = dv(s_i) \mid x_i \in V\}$ (note that $x_i, s_i \notin EV$ is equivalent to $x_i \in V$ by invariant 2). Thus $\theta' := \{x_i \mapsto dv(s_i) \mid x_i \in V\}$ is an mgu for u_1 and u_2 with $\theta' = dv(\theta)|_V$. ■

Now we are able to show 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 3.13 *Let $\mathcal{R}$ be a normal CTRS such that $eev(\mathcal{R})$ is weakly orthogonal and G be a goal. If there is a narrowing derivation $\widehat{G} \leadsto^*_\sigma G_1$, where G_1 is a trivial goal, then there is a narrowing derivation $G \leadsto^*_\phi G_0$ 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.*

Proof: By induction on the number k of narrowing steps in the derivation $\widehat{G} \leadsto^*_\sigma G_1$.

$k = 0$**:** Then $G_1 = \widehat{G}$ and σ is the identity substitution. Clearly, $G \leadsto^*_\phi G$ with $dv(G_1) = G$ and ϕ is the identity.

$k > 0$**:** Consider rule R: $f(\overline{t}) \to r \Leftarrow C$, where its extended version $eev(R)$: $l \to \widehat{r} \Leftarrow \widehat{C}$ with $l = f(\widehat{\overline{t}}, v_n(x_1, \ldots, x_n))$ is applied at position p in the first narrowing step. Let σ_0 be the mgu for $\widehat{G}|_p$ and $f(\widehat{\overline{t}}, v_n(x_1, \ldots, x_n))$ computed in the first step. Then the narrowing derivation has the structure

$$\widehat{G} \quad \leadsto_{\sigma_0} \quad \sigma_0(\widehat{C}, \widehat{G}[\widehat{r}]_p) \quad \leadsto^*_{\sigma_1} \quad G_1$$

Note that p is also a position in G since $\widehat{G}|_p$ is not a variable and $\widehat{G}$ has only variables in extension arguments. Consider the slightly modified left-hand side $l' = f(\widehat{\overline{t}}, x)$, where x is a new variable. Then l' and $\widehat{G}|_p$ are extended terms with different variables in all extension arguments. By Theorem 3.12, $\phi_0 := dv(\sigma_0)|_{V_0}$ with $V_0 = \mathcal{V}ar(\overline{t}) \cup \mathcal{V}ar(G)$ is an mgu for $f(\overline{t})$ and $G|_p$ (note that σ_0 also replaces the variable z, provided that $\widehat{G}|_p = f(\overline{s}, z)$, by a term $v_n(\cdots)$, but this has no influence on ϕ_0). Thus there is a narrowing step

$$G \quad \leadsto_{\phi_0} \quad \phi_0(C, G[r]_p)$$

with

$$\begin{array}{rcll} dv(\sigma_0(\widehat{C}, \widehat{G}[\widehat{r}]_p)) & = & dv(\sigma_0)(dv(\widehat{C}, \widehat{G}[\widehat{r}]_p)) & \text{(by Lemma 3.11)} \\ & = & dv(\sigma_0)(C, G[r]_p) & \\ & = & \phi_0(C, G[r]_p) & \end{array}$$

In order to apply the induction hypothesis, we have to show that $G' := \sigma_0(\widehat{C}, \widehat{G}[\widehat{r}]_p)$ is an extended goal, where all extension arguments are different variables. Since l is a linear term and l and $\widehat{G}|_p$ are variable disjoint, the mgu σ_0 cannot identify different extension variables in $\widehat{G}$. All extension arguments in $\widehat{r}$ and $\widehat{C}$ are also new variables. Thus G' can only contain multiple occurrences of extension variables if there are multiple occurrences of variables in r or C. By Theorem 3.8, all these variables can be shared (we have proved that variables can be shared only for rewrite derivations, but since completeness of narrowing is proved by lifting rewrite derivations to narrowing derivations, the sharing theorem can also be lifted to narrowing derivations). So, if G' has multiple occurrences of extension variables, we can replace them by different new variables and identify these different variables afterwards (this is always possible since they are instantiated to terms which are identical up to renaming due to the possible sharing). Therefore, we can assume that G' is an extended term. Since the derivation $G' \leadsto^*_{\sigma_1} G_1$ has less than k steps, by induction hypothesis there is a derivation $G'_1 := \phi_0(C, G[r]_p) \leadsto^*_{\phi_1} G_0$ with $dv(G_1) = G_0$ and $dv(\sigma_1(x)) = \phi_1(x)$ for all $x \in \mathcal{V}ar(G'_1)$. Moreover, the narrowing positions in both derivations are identical, and the applied rules correspond via the transformation eev. Hence

$$G \quad \leadsto_{\phi_0} \quad G'_1 \quad \leadsto^*_{\phi_1} \quad G_0$$

is the required narrowing derivation and, for all $x \in \mathcal{V}ar(G)$,

$$\begin{array}{rcll} dv(\sigma(x)) & = & dv(\sigma_1(\sigma_0(x)) & \\ & = & dv(\sigma_1)(dv(\sigma_0(x))) & \text{(by Lemma 3.11)} \\ & = & dv(\sigma_1)(dv(\sigma_0)(x)) & \text{(by Lemma 3.11)} \\ & = & dv(\sigma_1)(\phi_0(x)) & \\ & = & \phi_1(\phi_0(x)) & \end{array}$$

The last equality holds since $dv(\sigma_1)(x) = \phi_1(x)$ for all $x \in \mathcal{V}ar(G'_1)$ and w.l.o.g. $\phi_1(x) = x = \sigma_1(x)$ for all $x \in \mathcal{V}ar(\phi_0(G)) \backslash \mathcal{V}ar(G'_1)$. ■

If $\mathcal{R}$ is a weakly 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 weakly orthogonal (Theorem 3.13). The following proposition shows that this is always the case.

Proposition 3.14 *If $\mathcal{R}$ is a weakly orthogonal CTRS, then $eev(\mathcal{R})$ is weakly orthogonal.*

Proof: We have to show that all critical pairs in $eev(\mathcal{R})$ are trivial. For this purpose, consider the variable-disjoint variants

$$f(\overline{t_1}) \to r_1 \Leftarrow C_1 \quad \text{and} \quad g(\overline{t_2}) \to r_2 \Leftarrow C_2$$

of rules in $\mathcal{R}$, and the corresponding transformed rules

$$l'_1 \to \widehat{r_1} \Leftarrow \widehat{C_1} \quad \text{and} \quad l'_2 \to \widehat{r_2} \Leftarrow \widehat{C_2}$$

where $l'_1 = f(\widehat{\overline{t_1}}, v_m(x_1, \ldots, x_m))$ and $l'_2 = g(\widehat{\overline{t_2}}, v_n(y_1, \ldots, y_n))$. Assume that there is an overlap between these rules at nonvariable position p in l'_2, i.e., there is a mgu σ with $\sigma(l'_1) = \sigma(l'_2|_p)$. Since p is a nonvariable position and all arguments of $v_n(y_1, \ldots, y_n)$ are variables and also all other extension arguments of l'_2, p is a position in $g(\overline{t_2})$ and $g(\overline{t_2})|_p = dv(l'_2|_p)$. By Theorem 3.12 (after replacing the terms $v_m(x_1, \ldots, x_m)$ and $v_n(y_1, \ldots, y_n)$ by new variables), $\sigma' := dv(\sigma)|_V$ with $V = \mathcal{V}ar(f(\overline{t_1})) \cup \mathcal{V}ar(g(\overline{t_2})|_p)$ is a mgu for $f(\overline{t_1})$ and $g(\overline{t_2})|_p$. Since $\mathcal{R}_u$ is weakly orthogonal, it contains only trivial critical pairs, i.e., $p = \Lambda$ and $\sigma'(r_1) = \sigma'(r_2)$. This implies

$$\begin{array}{rcll} dv(\sigma(\widehat{r_1})) & = & dv(\sigma)(dv(\widehat{r_1})) & \text{(by Lemma 3.11)} \\ & = & \sigma'(r_1) & \\ & = & \sigma'(r_2) & \\ & = & dv(\sigma)(dv(\widehat{r_2})) & \\ & = & dv(\sigma(\widehat{r_2})) & \text{(by Lemma 3.11)} \end{array}$$

Hence $\sigma(\widehat{r_1})$ and $\sigma(\widehat{r_2})$ are identical if we ignore the extension arguments. All extension arguments in $\widehat{r_1}$ and $\widehat{r_2}$ are new variables occurring in $v_m(x_1, \ldots, x_m)$ and $v_n(y_1, \ldots, y_n)$, respectively. Since σ is an mgu for l'_1 and l'_2, $m = n$ and $\sigma(x_i) = \sigma(y_i)$ for $i = 1, \ldots, m$, i.e., σ identifies all corresponding extension arguments (here it is essential that the orderings of $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ are the same, cf. Footnote 5). Thus $\sigma(\widehat{r_1}) = \sigma(\widehat{r_2})$, i.e., the critical pair is trivial. Hence $eev(\mathcal{R})$ is weakly 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.[7] 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 result shows the general method.

Theorem 3.15 *Let $\mathcal{R}$ be a weakly 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 $\mathcal{R}$.*

Proof: Let $\mathcal{R}' = eev(\mathcal{R})$, G be a goal and σ be a solution for G, i.e., $\sigma(G)$ can be rewritten to the trivial goal G_0. By a straightforward extension of Theorem 3.9 to goals, there is an extended goal G' with $dv(G') = \sigma(G)$ and $G' \to^*_{\mathcal{R}'} \widehat{G_0}$. Thus there is a substitution σ' with $G' = \sigma'(\widehat{G})$ ($dv(\sigma')$ is identical to σ on $\mathcal{V}ar(G)$, but σ' additionally instantiates extension variables in $\widehat{G}$). Since N is a complete narrowing strategy w.r.t. $\mathcal{R}'$, there is a narrowing derivation $\widehat{G} \leadsto^*_{\phi'} \widehat{G_0}$ (the extension variables in $\widehat{G_0}$ can be considered as new constants) with $\sigma' = \tau \circ \phi'$ for some substitution τ. By Theorem 3.13 (note that $eev(\mathcal{R})$ is weakly orthogonal by Proposition 3.14), there is a narrowing derivation $G \leadsto^*_{\phi} G_0$ with $dv(\phi'(x)) = \phi(x)$ for all $x \in \mathcal{V}ar(G)$. Moreover, the narrowing positions in both derivations are identical and the rules correspond via the transformation eev, i.e., it is also a narrowing derivation computed by N.[8] Therefore, for all $x \in \mathcal{V}ar(G)$,

$$\begin{array}{rcll} \sigma(x) & = & dv(\sigma')(x) & \text{(by definition of } \sigma') \\ & = & dv(\sigma'(x)) & \\ & = & dv(\tau(\phi'(x))) & \\ & = & dv(\tau)(dv(\phi'(x))) & \text{(by Lemma 3.11)} \\ & = & dv(\tau)(\phi(x)) & \end{array}$$

i.e., σ is an instance of an answer computed by N for the goal G. This implies the completeness of N for $\mathcal{R}$. ■

Concrete applications of this result are shown in the following section.

3.5 Application of Extra Variable Elimination

3.5.1 Inductively Sequential Systems

If the termination of the rewrite relation is not required, a lazy narrowing strategy is necessary to compute solutions to goals. For instance, BABEL's lazy narrowing strategy primarily selects an outermost position but also allows narrowing steps at an inner position if the value at this position is demanded by some rule [MR92]. This narrowing strategy is complete for weakly orthogonal normal

[7] An exception is the needed narrowing strategy [AEH94] which additionally assigns a unifier because the unifier in a needed narrowing step is not necessarily a most general one.

[8] Here we assume that the narrowing strategy N does not depend on the extension arguments since these are always variables, cf. proof of Theorem 3.13. Although this is true for all known narrowing strategies, it must be checked for each new narrowing strategy in order to apply this theorem.

2-CTRS. However, it is well known that lazy narrowing may perform superfluous narrowing steps due to the interaction of redex selection and rule selection. As an alternative, needed narrowing is proposed in [AEH94]. 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. The precise definition can be found in [AEH94]. 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 3.3 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 [BK86] 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:

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

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.[9] Moreover, there is a strong correspondence between the rewrite derivations (see [BK86], 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 3.16 *If $\mathcal{R}$ is an inductively sequential CTRS, then $uc(eev(\mathcal{R}))$ is an unconditional inductively sequential rewrite system.*

Example 3.17 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
```

[9] Proposition 2.5.3 in [BK86] is not true in the presence of extra variables.

```
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
```

□

Since needed narrowing is an optimal and complete strategy for inductively sequential unconditional systems, we can apply the results of the previous section (as in Theorem 3.15), and we obtain the following new result.[10]

Theorem 3.18 *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.*

This result can be extended to *overlapping rules with excluding conditions.* For instance, the two rules

$$\begin{array}{lllll} R_1\colon & l & \to & r_1 & \Leftarrow \quad s = u_1 \\ R_2\colon & l & \to & r_2 & \Leftarrow \quad s = u_2 \end{array}$$

with identical left-hand sides but different ground constructor terms u_1, u_2 can be translated into the following unconditional rules:

$$\begin{array}{rcl} l & \to & cond_{R_1R_2}(s, r_1, r_2) \\ cond_{R_1R_2}(u_1, x, y) & \to & x \\ cond_{R_1R_2}(u_2, x, y) & \to & y \end{array}$$

Example 3.19 The following rules define an ordered insert function on lists (which may be a part of a sort function, see [Han92, p. 6]):

```
insert(E,[])    → [E]
insert(E,[F|L]) → [E,F|L]          ⇐ leq(E,F) = true
insert(E,[F|L]) → [F|insert(E,L)] ⇐ leq(E,F) = false
```

These rules can be translated into the following unconditional rules which are inductively sequential:

```
insert(E,[])    → [E]
insert(E,[F|L]) → condInsert(leq(E,F),[E,F|L],[F|insert(E,L)])
condInsert(true ,X,Y) → X
condInsert(false,X,Y) → Y
```

□

Using this translation method, we obtain an optimal narrowing strategy for a large class of equational logic programs.

[10] Note that needed narrowing steps do not always compute mgu's. However, Theorem 3.12 also holds for specialized unifiers computed in needed narrowing steps.

3.5.2 Extra Variables in Right-Hand Sides

Current functional logic languages like BABEL [MR92] and K-LEAF [GLMP91]) permit extra variables in conditions but not in the right-hand side of conditional rules. However, as observed by several authors [DOS87, Klo92, MH94], it makes good sense to allow extra variables also in right-hand sides if they occur in conditions (3-CTRS). Example 3.3 shows a sensible use of extra variables in right-hand sides. The following example [Klo92] shows that such extra variables can be a replacement for the *let* construct of functional languages.

Example 3.20 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.

Example 3.21 Consider the following rewrite rules:

```
a → X ⇐ g(X)=true
g(b) → true
g(c) → true
```

According to the definition of conditional rewriting in Section 3.2, `a` can be rewritten to `b` as well as `c`. Thus the rewrite relation is not confluent. □

To ensure the confluence of the rewrite relation and completeness of narrowing, additional restrictions are needed. Middeldorp and Hamoen [MH94] showed that narrowing is complete for level-confluent and terminating 3-CTRS. However, the completeness of refined strategies like basic narrowing is an open problem. In [BG89, DO90, Pad92] 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 weakly 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 3.8 and Proposition 3.14 to functional CTRS. Since Example 3.21 shows that 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 the confluence of $\to_{\mathcal{R}}$ is only needed to ensure that all valid equational goals can be proved by rewriting. Confluence is not necessary to apply our transformation method to such CTRS. Actually, using our transformation method, we can show completeness of narrowing

for CTRS satisfying conditions 1 and 2 w.r.t. goals provable by rewriting. However, completeness results are usually stated w.r.t. all valid goals. Therefore, we consider only CTRS where rewriting is sufficient to verify all valid goals. Note that each weakly orthogonal normal 2-CTRS is functional (by Theorem 3.2), while a 4-CTRS cannot be functional. Hence the class of functional CTRS lies between the classes of weakly orthogonal normal 2-CTRS and 3-CTRS.

We want to apply our transformation to show the completeness of narrowing strategies for functional CTRS. Therefore, we have to ensure that the transformed systems are weakly orthogonal. Since the proof of Proposition 3.14 does not depend on variable restrictions in the right-hand side, we immediately have the following result.

Proposition 3.22 *If $\mathcal{R}$ is a functional CTRS, then $eev(\mathcal{R})$ is weakly orthogonal.*

Hence we can apply Theorem 3.13 also to functional CTRS. Moreover, it is easy to check that the proof of Theorem 3.8 is also valid for functional CTRS, which implies the validity of Theorem 3.9 for functional CTRS. Thus Theorem 3.15 is also valid for functional CTRS:

Theorem 3.23 *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, Middeldorp and Hamoen [MH94] showed the completeness of simple narrowing for level-confluent and terminating 3-CTRS. However, they could not state any result for basic narrowing. Since basic conditional narrowing is complete for confluent and decreasing rewrite systems [MH94], Theorem 3.23 implies the following result.

Corollary 3.24 *Let $\mathcal{R}$ be a functional CTRS such that $eev(\mathcal{R})$ is decreasing. Then basic conditional narrowing is complete for $\mathcal{R}$.*

As mentioned in Section 3.2, most equational logic languages are constructor-based. For such constructor-based languages it is possible to define sophisticated narrowing strategies even for nonterminating rewrite systems. Such strategies are based on the principle of lazy evaluation [GLMP91, MR92], which can be made optimal in the case of inductively sequential programs [AEH94]. However, completeness results for lazy narrowing strategies are only known for constructor-based normal 2-CTRS [MR92] since variables in right-hand sides are usually excluded. Our transformation method yields new completeness results for functional CTRS by applying Theorem 3.23 to the completeness result of lazy narrowing [MR92] for weakly orthogonal normal 2-CTRS.

Corollary 3.25 *Let $\mathcal{R}$ be a functional CTRS. Then lazy narrowing is complete for $\mathcal{R}$.*

To obtain a further interesting result, we apply Theorem 3.23 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 3.26 *Let $\mathcal{R}$ be a functional CTRS such that the unconditional part $\mathcal{R}_u$ is inductively sequential.*[11] *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 3.3 and 3.20.

Due to these results, it is no problem to extend equational logic languages like BABEL [MR92] or K-LEAF [GLMP91] 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 3.3 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. Toyama and Oyamaguchi [TO94] characterized a confluent class of semi-equational CTRS with extra variables in right-hand sides (*left-right separated* CTRS), but this class is too restricted for equational logic programming due to the strong requirements on variable occurrences and the form of conditions. Fortunately, for constructor-based programs there is an interesting subclass of functional CTRS which has a simple syntactic characterization. Note that in constructor-based systems each conditional rule can be written in the form $l \to r \Leftarrow s \equiv t$.

Proposition 3.27 *Let $\mathcal{R}$ be a constructor-based normal CTRS which satisfies the following conditions:*

1. *The unconditional part $\mathcal{R}_u$ is weakly 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 $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(t)$.*

Then $\mathcal{R}$ is functional.

Proof: Bergstra and Klop [BK86] proved the confluence of (weakly) orthogonal normal 2-CTRS. More precisely, they have shown the stronger notion of level-confluence. Since we also require $\mathcal{R}_u$ to be weakly orthogonal, the only new divergence in rewrite derivations w.r.t. $\mathcal{R}$ (in comparison to weakly orthogonal normal 2-CTRS) is due to an overlap of a rule R with itself, where R has extra variables in the right-hand side which are instantiated to different terms. We show that all these divergencies can be joined by rewriting in a lower level. This implies the level-confluence of $\mathcal{R}$.

Let R be the rewrite rule $l \to r \Leftarrow s \equiv t$ containing extra variables in r so that t is a constructor term, $\mathcal{V}ar(s) \subseteq \mathcal{V}ar(l)$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(t)$. Consider a divergent rewrite computation $t_1 \; {}_{\mathcal{R}_n}\!\!\leftarrow t \to_{\mathcal{R}_n} t_2$ in level n caused by this rule, i.e., there are a position p in t and substitutions σ_1, σ_2 with $t|_p = \sigma_1(l) = \sigma_2(l)$, $\sigma_1(s \equiv t) \to^*_{\mathcal{R}_{n-1}} true$, $\sigma_2(s \equiv t) \to^*_{\mathcal{R}_{n-1}} true$, $t_1 = t[\sigma_1(r)]_p \neq t[\sigma_2(r)]_p = t_2$. We have to show: $\sigma_1(r) \downarrow_{\mathcal{R}_{n-1}} \sigma_2(r)$.

By definition of strict equality, there are ground constructor terms u_1, u_2 with $\sigma_1(s) \to^*_{\mathcal{R}_{n-1}} u_1$, $\sigma_1(t) \to^*_{\mathcal{R}_{n-1}} u_1$, $\sigma_2(s) \to^*_{\mathcal{R}_{n-1}} u_2$, $\sigma_2(t) \to^*_{\mathcal{R}_{n-1}} u_2$. Since $\sigma_1(l) = \sigma_2(l)$, $\sigma_1(x) = \sigma_2(x)$ for all

[11] 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.

$x \in \mathcal{V}ar(l)$. Thus $\mathcal{V}ar(s) \subseteq \mathcal{V}ar(l)$ implies $\sigma_1(s) = \sigma_2(s)$ and $u_1 = u_2$ by confluence of the lower level $\rightarrow_{\mathcal{R}_{n-1}}$. Since t is a constructor term and $u_1 = u_2$, $\sigma_1(x) \downarrow_{\mathcal{R}_{n-1}} \sigma_2(x)$ for all $x \in \mathcal{V}ar(t)$. Therefore, $\sigma_1(x) \downarrow_{\mathcal{R}_{n-1}} \sigma_2(x)$ for all $x \in \mathcal{V}ar(t) \cup \mathcal{V}ar(l)$. This implies $\sigma_1(r) \downarrow_{\mathcal{R}_{n-1}} \sigma_2(r)$ by $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(t)$. ■

As a consequence of this proposition, the rewrite system in Example 3.20 is functional. It is straightforward to refine the proposition to conditional rules with more than one strict equation in the condition part. Requirement 2 can be replaced by the relaxed requirement that, for all conditional rules

$$l \rightarrow r \Leftarrow s_1 \equiv t_1, \ldots, s_k \equiv t_k$$

with extra variables in r, the terms $t_1, \ldots, t_k$ are constructor terms, $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(t_j)$ $(i = 1, \ldots, k)$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \bigcup_{j=1}^{k} \mathcal{V}ar(t_j)$. For instance, the conditional rule

```
f(X) → Z+Z ⇐ X+X≡Y, Y*Y≡Z
```

satisfies this relaxed requirement. Rewrite rules with similar restrictions but additional termination requirements are considered in [BG89] as "quasi-reductive rules." Rewriting with quasi-reductive rules is always terminating. This allows to deal with nontrivial critical pairs and rewrite systems which are not constructor-based, but it is too restricted from a functional programming perspective.

We conclude this section by discussing some advantages of 3-CTRS in comparison to 2-CTRS. Current equational logic languages with a lazy operational semantics (BABEL [MR92], K-LEAF [GLMP91]) do not permit extra variables in right-hand sides. This restriction to 2-CTRS requires the representation of some functions as relations, namely those functions which could be defined by rules with extra variables in right-hand sides (if we do not want to define the functions by completely different rewrite rules). For instance, the function `last` of Example 3.3 can be defined as a relation by the following rule without extra variables in the right-hand side:

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

However, such a representation requires the flattening of originally nested function calls. E.g., an original goal like `0*last([1,2])≡N` must be transformed into the new goal

```
last([1,2],E)≡true, 0*E≡N .
```

This new goal has a worse operational behavior than the original one, due to the fact that all equations in a goal must be proved in order to verify the entire goal. In particular, the function call `last([1,2],E)` is evaluated. However, if the function '`*`' is defined by the rule

```
0*X → 0
```

the original goal is verified without evaluating the function call `last([1,2])` provided that a "good" narrowing strategy like needed narrowing [AEH94] is used:

$$\texttt{0*last([1,2])} \equiv \texttt{N} \quad \leadsto_{\{\}} \quad \texttt{0} \equiv \texttt{N} \quad \leadsto_{\{\texttt{N} \mapsto \texttt{0}\}} \quad \texttt{true}$$

Although this example might look artificial, avoiding unnecessary evaluations of subterms becomes important in the presence of unbound variables, since different bindings of a variable causes different subterms to be evaluated. For instance, if we have to solve the goal `Z*last([1,2])≡N`, the function call `last([1,2])` need not be evaluated if `Z` is bound to `0`, but it must be evaluated if `Z` is bound to a nonzero value. A detailed discussion on this subject can be found in [AEH94].

One of the advantages of integrating functions into logic programming is the possible nesting of expressions. In nested expressions, it is not necessary to evaluate all subexpressions but only those which are needed to compute the overall result (see [AEH94] for more details). As shown by the previous example, 3-CTRS permits more nested expressions than 2-CTRS. Hence we can obtain a better operational behavior. Another advantage of 3-CTRS is their ability to express *let* constructs of functional languages (see Example 3.20 and [BG89] for a more detailed discussion). *let* constructs enable the programmer to express explicit sharing of values in order to avoid multiple evaluations of the same expression.

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 weakly 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. As far as we know, these are the first completeness results for narrowing calculi in the presence of nonterminating functions and extra variables in right-hand sides of rules. 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.

References

[AEH94] 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.

[BE86] D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. European Symposium on Programming*, pp. 119–132. Springer LNCS 213, 1986.

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

[BK86] 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.

[BKW92] A. Bockmayr, S. Krischer, and A. Werner. An Optimal Narrowing Strategy for General Canonical Systems. In *Proc. of the 3rd Intern. Workshop on Conditional Term Rewriting Systems*, pp. 483–497. Springer LNCS 656, 1992.

[BMPT87] R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. Intensional Negation of Logic Programs: Examples and Implementation Techniques. In *Proc. of the TAPSOFT '87*, pp. 96–110. Springer LNCS 250, 1987.

[BMPT90] 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.

[BvEG+87] 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.

[DJ90] 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.

[DO90] N. Dershowitz and M. Okada. A Rationale for Conditional Equational Programming. *Theoretical Computer Science*, Vol. 75, pp. 111–138, 1990.

[DOS87] 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.

[Ech88] R. Echahed. On Completeness of Narrowing Strategies. In *Proc. CAAP'88*, pp. 89–101. Springer LNCS 299, 1988.

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

[GLMP91] 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.

[GM86] 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.

[GM87] J.A. Goguen and J. Meseguer. Models and Equality for Logical Programming. In *Proc. of the TAPSOFT '87*, pp. 1–22. Springer LNCS 250, 1987.

[Han90] 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.

[Han92] M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–23. Springer LNCS 631, 1992.

[Han94a] M. Hanus. Combining Lazy Narrowing and Simplification. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming*. Springer LNCS (to appear), 1994.

[Han94b] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.

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

[Hus85] H. Hussmann. Unification in Conditional-Equational Theories. In *Proc. EUROCAL '85*, pp. 543–553. Springer LNCS 204, 1985.

[Kap87] S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination, and confluence. *Journal of Symbolic Computation*, Vol. 4, No. 3, pp. 295–334, 1987.

[Klo92] 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.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer, second, extended edition, 1987.

[MH94] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, Vol. 5, pp. 213–253, 1994.

[MM82] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.

[MR92] 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.

[Pad92] P. Padawitz. Generic Induction Proofs. In *Proc. of the 3rd Intern. Workshop on Conditional Term Rewriting Systems*, pp. 175–197. Springer LNCS 656, 1992.

[PP94] 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.

[TO94] Y. Toyama and M. Oyamaguchi. Church-Rosser Property and Unique Normal Form Property of Non-Duplicating Term Rewriting Systems. In *Fourth Int. Workshop on Conditional Term Rewriting Systems.* Springer LNCS (to appear), 1994.