Preliminary version. Final version in Journal of Computer Languages (Elsevier), Vol. 23, No. 2–4, pp. 61–85, 1997

Lazy Narrowing with Simplification*

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

Abstract

Languages that integrate functional and logic programming styles with a complete operational semantics are based on narrowing. In order to avoid useless computations, lazy narrowing strategies have been proposed in the past. This paper presents an improvement of lazy narrowing by incorporating deterministic simplification steps into lazy narrowing derivations. These simplification steps reduce the search space so that in some cases infinite search spaces are reduced to finite ones. We consider two classes of programs where this strategy can be applied. Firstly, we show soundness and completeness of our strategy for functional logic programs based on ground confluent and terminating rewrite systems. Then, we show similar results for constructor-based weakly orthogonal (not necessarily terminating) rewrite systems. Finally, we demonstrate the improved operational behavior by means of several examples. Since most functional logic languages are based on programs belonging to one of these classes, our result is a significant step to improve the operational semantics of existing functional logic languages.

1 Introduction

In recent years, a lot of proposals have been made to amalgamate functional and logic programming languages [22]. Functional logic languages with a sound and complete operational semantics are based on *narrowing*, a combination of the reduction principle of functional languages and the resolution principle of logic languages. Narrowing, originally introduced in automated theorem proving [43], is used to *solve* equations by finding appropriate values for variables occurring in arguments of functions. This is done by unifying (rather than matching) an input term with the left-hand side of some rule and then replacing the instantiated input term by the instantiated right-hand side of the rule.

Example 1.1 Consider the following rules defining the addition of two natural numbers which are represented by terms built from 0 and s:

$$\begin{array}{rcll} 0+y & \to & y & \quad (R_1) \\ s(x)+y & \to & s(x+y) & \quad (R_2) \end{array}$$

*This paper is a revised version of papers appeared in the proceedings of ESOP'94 and PLILP'94.

The equation $z + s(0) \approx s(s(0))$ can be solved by a narrowing step with rule R_2 followed by a narrowing step with rule R_1 so that z is instantiated to $s(0)$ and the instantiated equation is reduced to the trivial equation $s(s(0)) \approx s(s(0))$:

$$z + s(0) \approx s(s(0)) \;\rightsquigarrow_{\{z \mapsto s(x)\}}\; s(x + s(0)) \approx s(s(0)) \;\rightsquigarrow_{\{x \mapsto 0\}}\; s(s(0)) \approx s(s(0))$$

Hence we have found the solution $z \mapsto s(0)$ to the given equation. □

In order to ensure completeness in general, the left-hand side of *each* rule must be unified with *each* nonvariable subterm of the given equation. Clearly, this yields a huge search space. The situation can be improved by particular narrowing strategies which restrict the possible positions for the application of the next narrowing step, e.g., basic [26], innermost [14], outermost [11], lazy [41], or needed narrowing [2]. In this paper we consider *lazy narrowing strategies* where narrowing steps are applied at outermost positions in general and at an inner position only if it is demanded and contributes to some later narrowing step at an outer position. Similarly to pure functional programming, such a lazy strategy avoids useless steps in comparison to an eager strategy. However, in the context of functional logic programming, a lazy narrowing strategy can also have an unpleasant behavior if a demanded argument term has infinitely many head normal forms (i.e., if it can be derived to infinitely many terms with a variable or constructor at the top).

Example 1.2 Consider the following rules which may be part of a program for arithmetic operations:

$$\begin{array}{rcll} 0 * x & \to & 0 & (R_3) \\ x * 0 & \to & 0 & (R_4) \end{array} \qquad\qquad \begin{array}{rcll} one(0) & \to & s(0) & (R_5) \\ one(s(x)) & \to & one(x) & (R_6) \end{array}$$

If we want to compute a solution to the equation $one(z) * 0 \approx 0$ by lazy narrowing, we could try to apply rule R_3 to evaluate the left-hand side. In this case the first argument $one(z)$ is demanded and must be evaluated to a term with a constructor at the top. Unfortunately, there are infinitely many possibilities to compute a head normal form $s(0)$ of the term $one(z)$ by instantiating z with $\underbrace{s(\cdots s}_{n}(0)\cdots)$ for arbitrary n. Hence lazy narrowing has an infinite search space in this example and does not compute a solution in a sequential implementation (see [18] for a discussion of problems with sequential implementations of lazy narrowing). However, we could avoid this infinite search space by computing the normal form of both sides of the equation before applying a narrowing step. The normal form of the initial equation is $0 \approx 0$ (reduction of the left-hand side with rule R_4) which is trivially true. □

The idea of reduction to normal form before applying a narrowing step has been mainly proposed with respect to eager narrowing strategies [13, 14, 25, 38, 42]. It has been shown that eager narrowing with normalization is a more efficient control strategy than left-to-right SLD-resolution for equivalent logic programs [14, 21]. On the other hand, only little work has been done to improve the efficiency of outermost or lazy strategies. Echahed [12] has shown the completeness of any narrowing strategy with simplification under strong requirements (uniformity of specifications). Dershowitz et al. [9] have proposed to combine lazy narrowing with simplification and demonstrated the usefulness of inductive consequences for simplification. However, they have not proved completeness of their lazy unification calculus if all terms are simplified to their normal form after each unification step.[1]

The main contribution of this paper is the combination of lazy narrowing with intermediate simplification steps. We show that this combination does not destroy the completeness of lazy narrowing. We prove this result for the following two classes of functional logic programs.[2]

1. *Ground confluent and terminating rewrite systems:* All existing proposals for combining narrowing with simplification require terminating rewrite systems [13, 14, 25, 38, 42]. For this case, narrowing is a method to compute unifiers in the presence of an equational theory (known as E-unification, see [3] for a survey). We will develop a calculus for this class, called *lazy unification with simplification*, and provide a rigorous completeness proof. This calculus has a lazy behavior w.r.t. unification, i.e., functions are only evaluated if their value is required to decide the unifiability of terms. Moreover, we allow to use program rules as well as additional inductive consequences for simplification between narrowing steps. This has been proved to be useful in other (eager) calculi [12, 14, 38].

2. *Weakly orthogonal (not necessarily terminating) rewrite systems:* If the functional logic program is not based on a terminating rewrite system, a lazy narrowing strategy is needed [6, 35, 41]. Since normal forms may not exist in the presence of nonterminating functions, equality between two expressions is interpreted as *strict equality* in such languages (e.g., BABEL [37], K-LEAF [17]), i.e., two expressions are equal iff they are reducible to a same ground constructor term. The confluence of the rewrite system is ensured by syntactic criteria (left-linearity and nonambiguity). Lazy narrowing is a complete method to compute unifiers w.r.t. strict equality for such programs. However, no attempt has been made to use program rules for simplification between narrowing steps. Due to the absence of normal forms for some expressions, full normalization between narrowing steps would be incomplete. Therefore, we propose the integration of *lazy simplification* into lazy narrowing derivations for such programs.

As far as we know, all functional logic languages with a complete operational semantics are based on programs belonging to one of these classes. For instance, programs with the requirements of ALF [19], LPG [4] or SLOG [14] are ground complete and terminating, whereas programs with the requirements of BABEL [37] or K-LEAF [17] are weakly orthogonal. Thus our result is a significant step to improve the operational semantics of existing functional logic languages. We will emphasize this point by discussing the advantage of lazy narrowing with simplification for various classes of functional logic programs.

In the next section we recall basic notions from term rewriting and functional logic programming. In Section 3 we present the lazy unification calculus with simplification and prove its soundness and completeness for ground confluent and terminating rewrite systems. In Section 4 we show

[1] In fact, their completeness proof for lazy narrowing does not hold if eager rewriting is included since rewriting in their sense does not reduce the complexity measure used in their completeness proof and may lead to infinite instead of successful derivations.

[2] For the sake of simplicity, we consider only programs based on unconditional rewrite systems. However, it does not seem difficult to extend the results of Section 4 to conditional rules with extra variables in conditions using the transformation techniques presented in [24].

how to include a deterministic simplification process into lazy narrowing derivations w.r.t. weakly orthogonal rewrite systems. In Section 5 we discuss the usefulness of this simplification process for different classes of functional logic programs. Finally, we conclude with a discussion of related work.

2 Preliminaries

In this section we recall basic notions of term rewriting [8] and functional logic programming [22].

A *signature* is a set $\mathcal{F}$ of *function symbols*.[3] Every $f \in \mathcal{F}$ is associated with an *arity* n, denoted f/n. Let $\mathcal{X}$ be a countably infinite set of *variables*. Then the set $\mathcal{T}(\mathcal{F},\mathcal{X})$ of *terms* built from $\mathcal{F}$ and $\mathcal{X}$ is the smallest set containing $\mathcal{X}$ such that $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F},\mathcal{X})$ whenever $f \in \mathcal{F}$ has arity n and $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{F},\mathcal{X})$. We write f instead of $f()$ whenever f has arity 0. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$ (similarly for the other syntactic constructions defined below, like equation, rewriting rule etc.). A term t is called *ground* if $\mathcal{V}ar(t) = \emptyset$. In the following we assume that $\mathcal{F}$ is a signature with at least one constant.

The execution of functional logic programs requires notions like substitution, unifier and subterm which will be defined next. A *substitution* σ is a mapping from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F},\mathcal{X})$ such that its *domain* $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is finite. We frequently identify a substitution σ with the set $\{x \mapsto \sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. Substitutions are extended to morphisms on $\mathcal{T}(\Sigma,\mathcal{X})$ by $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$ for every term $f(t_1,\ldots,t_n)$. A substitution σ is called *ground* if $\sigma(x)$ is a ground term for all $x \in \mathcal{D}om(\sigma)$. The *composition of two substitutions* ϕ and σ is defined by $\phi \circ \sigma(x) = \phi(\sigma(x))$ for all $x \in \mathcal{X}$. The *union of two substitutions* ϕ and σ is defined by

$$(\phi \cup \sigma)(x) \;=\; \begin{cases} \phi(x) & \text{if } x \in \mathcal{D}om(\phi) \\ \sigma(x) & \text{if } x \in \mathcal{D}om(\sigma) \\ x & \text{otherwise} \end{cases}$$

only if $\mathcal{D}om(\phi) \cap \mathcal{D}om(\sigma) = \emptyset$. The *restriction* $\sigma_{|V}$ of a substitution σ to a set V of variables is defined by $\sigma_{|V}(x) = \sigma(x)$ if $x \in V$ and $\sigma_{|V}(x) = x$ if $x \notin V$. A term s is called *instance* of a term t if there is a substitution σ with $s = \sigma(t)$ (similarly for the other syntactic constructions defined below).

A *unifier* of two terms s and t is a substitution σ with $\sigma(s) = \sigma(t)$. A unifier σ is called *most general* (*mgu*) if for every other unifier σ' there is a substitution ϕ with $\sigma' = \phi \circ \sigma$. Most general unifiers are unique up to variable renaming. By introducing a total ordering on variables we can uniquely choose *the* most general unifier of two terms. A *position* p in a term t is represented by a sequence of natural numbers, $t|_p$ 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 [8] 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 $e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow \cdots$. $\rightarrow$ is called *confluent* if for all $e, e_1, e_2 \in S$ with $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$ there exists an element $e_3 \in S$ with $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$.

[3] In this paper we consider only single-sorted programs. The extension to many-sorted signatures is straightforward [39]. Since sorts are not relevant to the subject of this paper, we omit them for the sake of simplicity.

An *equation* $s \approx t$ is a multiset containing two terms s and t. Thus equations to be unified are symmetric. In order to compute with functional logic programs, we will use the equations specifying functions only in one direction. Hence we define a *rewrite rule* $l \rightarrow r$ as a pair of terms l, r satisfying $l \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ where l and r are called left-hand side and right-hand side, respectively. A rewrite rule is called a *variant* of another rule if it is obtained by a unique replacement of variables by other variables. In the following we assume that $\mathcal{R}$ is a set of rewrite rules, which is also called *term rewriting system*.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{\mathcal{R}} s$ if there exist a position p in t, a rewrite rule $l \rightarrow r$ 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 $t \rightarrow_{\mathcal{R}} s$. A substitution σ is called *irreducible* or *normalized* if $\sigma(x)$ is in normal form for all variables $x \in \mathcal{X}$. A term rewriting system is *(ground) confluent* if the restriction of $\rightarrow_{\mathcal{R}}$ to the set of all (ground) terms is confluent. If $\mathcal{R}$ is (ground) confluent and terminating, then each (ground) term t has a unique normal form which is denoted by $t\downarrow_{\mathcal{R}}$.

We are interested in proving the validity of equations. An equation $s \approx t$ is called *valid* (w.r.t. $\mathcal{R}$) if $s \leftrightarrow^*_{\mathcal{R}} t$. By Birkhoff's Completeness Theorem, this is equivalent to the validity of $s \approx t$ in all models of $\mathcal{R}$. In this case we also write $s =_{\mathcal{R}} t$. If $\mathcal{R}$ is (ground) confluent and terminating, we can decide the validity of a (ground) equation $s \approx t$ by computing the normal form of both sides using an arbitrary sequence of rewrite steps, since $s \leftrightarrow^*_{\mathcal{R}} t$ iff $s\downarrow_{\mathcal{R}} = t\downarrow_{\mathcal{R}}$. In order to compute *solutions* to a nonground equation $s \approx t$, we have to find appropriate instantiations for the variables in s and t. This can be done by *narrowing*. A term t is *narrowable* into a term t' if there exist a nonvariable position p (i.e., $t|_p \notin \mathcal{X}$), a variant $l \rightarrow r$ of a rewrite rule with $\mathcal{V}ar(t) \cap \mathcal{V}ar(l) = \emptyset$, a substitution σ such that σ is a mgu of $t|_p$ and l, and $t' = \sigma(t[r]_p)$. In this case we write $t \leadsto_{[p,l\rightarrow r,\sigma]} t'$ or simply $t \leadsto_{\sigma} t'$.[4] If there is a narrowing sequence $t_1 \leadsto_{\sigma_1} t_2 \leadsto_{\sigma_2} \cdots \leadsto_{\sigma_{n-1}} t_n$, we write $t_1 \leadsto^*_{\sigma} t_n$ with $\sigma = \sigma_{n-1} \circ \cdots \circ \sigma_2 \circ \sigma_1$.

Narrowing is able to solve equations w.r.t. $\mathcal{R}$. For this purpose we introduce two new function symbols $=^?$ and *true* and add the rewrite rule $x =^? x \rightarrow true$ to $\mathcal{R}$. Then narrowing is sound and complete in the following sense.

Theorem 2.1 ([26]) *Let $\mathcal{R}$ be a term rewriting system so that $\rightarrow_{\mathcal{R}}$ is confluent and terminating.*

1. *If $s =^? t \leadsto^*_{\sigma} true$, then $\sigma(s) =_{\mathcal{R}} \sigma(t)$.*
2. *If $\sigma'(s) =_{\mathcal{R}} \sigma'(t)$, then there exist a narrowing derivation $s =^? t \leadsto^*_{\sigma} true$ and a substitution ϕ with $\phi(\sigma(x)) =_{\mathcal{R}} \sigma'(x)$ for all $x \in \mathcal{V}ar(s) \cup \mathcal{V}ar(t)$.*

Since this simple narrowing procedure (enumerating all narrowing derivations) is very inefficient, several authors have proposed restrictions on the admissible narrowing derivations (see [22] for a detailed survey). For instance, Hullot [26] has introduced *basic narrowing* where narrowing steps in positions introduced by substitutions are forbidden. Fribourg [14] has proposed *innermost narrowing* where narrowing is applied only at innermost positions, and Hölldobler [25] has combined innermost and basic narrowing. Krischer and Bockmayr [29] have proposed additional tests during narrowing derivations to eliminate redundant derivations. Narrowing at *outermost* positions is

[4]Since the instantiation of the variables in the rule $l \rightarrow r$ by σ is not relevant for the computed solution of a narrowing derivation, we omit this part of σ in the example derivations in this paper.

complete only if the term rewrite system satisfies strong restrictions [11]. *Lazy narrowing* [6, 35, 41] is influenced by the idea of lazy evaluation in functional programming languages. Lazy narrowing steps are only applied at outermost positions with the exception that arguments are evaluated by narrowing to their head normal form if their values are required for an outermost narrowing step (see [37] for an exact definition of a *lazy narrowing position*). Since lazy strategies are relevant in the context of nonterminating rewrite rules, these strategies have been proved to be complete w.r.t. domain-based interpretations of rewrite rules [17, 37]. *Lazy unification* is very similar to lazy narrowing but manipulates sets of equations rather than terms. It has been proved to be complete for confluent and terminating term rewriting systems w.r.t. standard semantics [9, 34]. Therefore, lazy unification calculi are more appropriate in the context of terminating rewrite systems and standard semantics of equality, whereas lazy narrowing calculi are appropriate in the presence of nonterminating rules. Thus, we follow this distinction.[5]

Another improvement of simple narrowing is *normalizing narrowing* [13] where the term is rewritten to its normal form before a narrowing step is applied. This optimization is important since it prefers deterministic computations: rewriting a term to normal form can be done in a deterministic way since every rewriting sequence yields the same result (if $\mathcal{R}$ is confluent and terminating) whereas different narrowing steps may lead to different solutions and, therefore, all admissible narrowing steps must be considered. In a sequential implementation, rewriting can be efficiently implemented like reductions in functional languages whereas narrowing steps need costly backtracking management like in Prolog. For instance, if $s =_{\mathcal{R}} t$, normalizing narrowing will prove the validity by a pure deterministic computation (reducing s and t to the same normal form) whereas simple narrowing would compute the normal form of s and t by costly narrowing steps. As shown in [14, 21], normalizing narrowing has the desirable effect that functional logic programs are more efficiently executable than pure logic programs.

The idea of normalizing narrowing can also be combined with other narrowing restrictions. Réty [42] has proved completeness of *normalizing basic narrowing*, Fribourg [14] has proposed *normalizing innermost narrowing* and Hölldobler [25] has combined innermost basic narrowing with normalization. Because of these advantages, normalizing narrowing is the foundation of several programming languages which combines functional and logic programming, like ALF [19], LPG [4], or SLOG [14]. However, normalization has not been included in lazy narrowing strategies.[6] Therefore, we will show that deterministic simplification steps can be performed before nondeterministic lazy narrowing steps without destroying the completeness of lazy narrowing. The problems of integrating normalization into basic narrowing [42] shows that such a result is not obvious.

3 Ground Confluent and Terminating Programs

In this section we assume that $\mathcal{R}$ *is a ground confluent and terminating term rewriting system.* First, we present our basic lazy unification calculus to solve a system of equations. The inclusion of a normalization process will be shown in Section 3.2. The "laziness" of our calculus is in the spirit of lazy evaluation in functional programming languages, i.e., terms are evaluated only if their values are needed.

[5] Note that this distinction becomes essential if one considers higher-order rewrite rules [40].

[6] Except for [9, 12], but see the remarks in Section 1.

Lazy narrowing

$f(t_1,\ldots,t_n) \approx t, E \quad \overset{lu}{\Longrightarrow} \quad t_1 \approx l_1,\ldots,t_n \approx l_n, r \approx t, E$

if $t \notin \mathcal{X}$ or $t \in \mathcal{V}ar(f(t_1,\ldots,t_n)) \cup \mathcal{V}ar(E)$ and $f(l_1,\ldots,l_n) \to r$ new variant of a rewrite rule

Decomposition of equations

$f(t_1,\ldots,t_n) \approx f(t'_1,\ldots,t'_n), E \quad \overset{lu}{\Longrightarrow} \quad t_1 \approx t'_1,\ldots,t_n \approx t'_n, E$

Partial binding of variables

$x \approx f(t_1,\ldots,t_n), E \quad \overset{lu}{\Longrightarrow} \quad x \approx f(x_1,\ldots,x_n), x_1 \approx \phi(t_1),\ldots,x_n \approx \phi(t_n), \phi(E)$

if $x \in \mathcal{V}ar(f(t_1,\ldots,t_n)) \cup \mathcal{V}ar(E)$ and $\phi = \{x \mapsto f(x_1,\ldots,x_n)\}$ (where x_i new variable)

Figure 1: The lazy unification calculus

3.1 A Calculus for Lazy Unification

Lazy narrowing as introduced in Section 2 is defined only for constructor-based programs (see also Section 4). Since we do not require constructor-based programs in this section, we present a lazy unification calculus which is slightly more general than lazy narrowing. This lazy unification calculus manipulates sets of equations in the style of Martelli and Montanari [33] rather than terms as in narrowing calculi. Hence we define an *equation system* E to be a multiset of equations (in the following we write such sets without curly brackets if it is clear from the context). A *solution* of an equation system E is a ground substitution σ such that $\mathcal{V}ar(E) \subseteq \mathcal{D}om(\sigma)$ and $\sigma(s) =_{\mathcal{R}} \sigma(t)$ for all equations $s \approx t \in E$.[7] An equation system E is *solvable* if it has at least one solution. A set S of substitutions is a *complete set of solutions* for E iff

1. for all $\sigma \in S$, σ is a solution of E;

2. for every solution θ of E, there exists some $\sigma \in S$ with $\theta(x) =_{\mathcal{R}} \sigma(x)$ for all $x \in \mathcal{V}ar(E)$.

In order to compute solutions of an equation system, we transform it by the rules in Figure 1 until no more rules can be applied. The lazy narrowing transformation applies a rewrite rule to a function occurring outermost in an equation.[8] Actually, this is not a narrowing step as defined in Section 2 since the argument terms may not be unifiable. Narrowing steps can be simulated by a sequence of transformations in the lazy unification calculus but not vice versa since our calculus also allows the application of rewrite rules to the arguments of the left-hand sides. The decomposition transformation generates equations between the argument terms of an equation if both sides have the same outermost symbol. The partial binding of variables can be applied if the variable x occurs at different positions in the equation system. In this case we instantiate the variable only with the outermost function symbol. A full instantiation by the substitution $\phi = \{x \mapsto f(t_1,\ldots,t_n)\}$ may increase the computational work if x occurs several times and the evaluation of $f(t_1,\ldots,t_n)$

[7] We are interested in *ground* solutions since later we will include inductive consequences which are valid in the ground models of $\mathcal{R}$. As pointed out in [38], this ground approach subsumes the conventional narrowing approaches where also nonground solutions are taken into account (as in Theorem 2.1).

[8] Similarly to logic programming, we have to apply rewrite rules with fresh variables in order to ensure completeness.

is costly. In order to avoid this problem of *eager variable elimination* (see [15]), we perform only a partial binding which is also called "root imitation" in [15].

It is possible to add further rules to simplify equation systems like the elimination of trivial equations:

$$t \approx t, E \quad \overset{lu}{\Longrightarrow} \quad E$$

However, these rules are not really necessary and we omit them in our first approach. Later we will see how to add deterministic (failure) rules to reduce the search space of the calculus.

At first sight our lazy unification calculus has many similarities with the lazy unification rules presented in [9, 15, 34, 39]. This is not accidental since these systems have inspired us. However, there are also essential differences. Since we are interested in reducing the computational costs in the E-unification procedure, our rules behave "more lazily". In our calculus it is allowed to evaluate a term only if its value is needed[9] (in several positions). Otherwise, the term is left unevaluated.

Example 3.1 Consider the rewrite rule $0 * x \to 0$. Then the only transformation sequence of the equation $0 * t \approx 0$ (where t may be a costly function) is

$$\begin{array}{rlll} 0 * t \approx 0 & \overset{lu}{\Longrightarrow} & 0 \approx 0,\ t \approx x,\ 0 \approx 0 & \qquad (\textit{lazy narrowing}) \\ & \overset{lu}{\Longrightarrow} & t \approx x,\ 0 \approx 0 & \qquad (\textit{decomposition}) \\ & \overset{lu}{\Longrightarrow} & t \approx x & \qquad (\textit{decomposition}) \end{array}$$

Thus the term t is not evaluated since its concrete value is not needed. Consequently, we may compute solutions which are not normalized. That is a desirable property in the presence of a lazy evaluation mechanism. □

The conventional transformation rules for unification w.r.t. an empty equational theory [33] bind a variable x to a term t only if x does not occur in t. This *occur check* must be omitted in the presence of evaluable function symbols. Moreover, we must also instantiate occurrences of x in the term t which is done in our partial binding rule. The following example shows the necessity of these extensions.

Example 3.2 Consider the rewrite rule $f(c(a)) \to a$. Then we can solve the equation $x \approx c(f(x))$ by the following transformation sequence:

$$\begin{array}{rlll} x \approx c(f(x)) & \overset{lu}{\Longrightarrow} & x \approx c(x_1),\ x_1 \approx f(c(x_1)) & \qquad (\textit{partial binding}) \\ & \overset{lu}{\Longrightarrow} & x \approx c(x_1),\ c(x_1) \approx c(a),\ x_1 \approx a & \qquad (\textit{lazy narrowing}) \\ & \overset{lu}{\Longrightarrow} & x \approx c(x_1),\ x_1 \approx a,\ x_1 \approx a & \qquad (\textit{decomposition}) \\ & \overset{lu}{\Longrightarrow} & x \approx c(a),\ x_1 \approx a,\ a \approx a & \qquad (\textit{partial binding}) \\ & \overset{lu}{\Longrightarrow} & x \approx c(a),\ x_1 \approx a & \qquad (\textit{decomposition}) \end{array}$$

In fact, the initial equation is solvable and $\{x \mapsto c(a)\}$ is a solution of this equation. This solution is also an obvious solution of the final equation system if we disregard the auxiliary variable x_1. □

[9] Although our lazy narrowing rule is more restricted than in other lazy unification calculi, it is not optimal in the sense of [2] since we do not require strongly sequential rewrite systems.

Coalesce

$x \approx y, E \overset{var}{\Longrightarrow} x \approx y, \phi(E)$

if $x, y \in \mathcal{V}ar(E)$, $x \neq y$, and $\phi = \{x \mapsto y\}$

Trivial

$x \approx x, E \overset{var}{\Longrightarrow} E$

Figure 2: The variable elimination rules

In the rest of this section, we will prove soundness and completeness of our lazy unification calculus. Soundness simply means that each solution of the transformed equation system is also a solution of the initial equation system. Completeness is more difficult since we have to take into account all possible transformations. Therefore, we will show that a solvable equation system can be transformed into another very simple equation system which has "an obvious solution". Such a final equation system is said to be in "solved form". According to [15, 33], we call an equation $x \approx t$ of an equation system E *solved* (in E) if x is a variable which occurs neither in t nor anywhere else in E. In this case variable x is also called *solved* (in E). An equation system is *solved* or in *solved form* if all its equations are solved. A variable or equation is *unsolved* in E if it occurs in E but is not solved.

The lazy unification calculus in the present form cannot transform each solvable equation system into a solved form since equations between variables are not simplified. For instance, the equation system

$$x \approx f(y),\ y \approx z_1,\ y \approx z_2,\ z_1 \approx z_2$$

is irreducible w.r.t. $\overset{lu}{\Longrightarrow}$ but not in solved form since the variables y, z_1, z_2 have multiple occurrences. Fortunately, this is not a problem since a solution can be extracted by merging the variables occurring in unsolved equations. Therefore, we call this system quasi-solved. An equation system is *quasi-solved* if each equation $s \approx t$ is solved or has the property $s, t \in \mathcal{X}$. In the following we will show that a quasi-solved equation system has solutions which can be easily computed by applying the rules in Figure 2 to it. The separation between the lazy unification rules in Figure 1 and the variable elimination rules in Figure 2 has technical reasons that will become apparent later (e.g., applying variable elimination to the equation $y \approx z_1$ may not reduce the complexity measure used in our completeness proofs). However, it is obvious to extract the solutions of a quasi-solved equation system E. For this purpose we transform E by the rules in Figure 2 into a solved equation system which has a direct solution. This is justified by the following propositions.

Proposition 3.3 *Let E and E' be equation systems with $E \overset{var}{\Longrightarrow} E'$. Then E and E' have the same solutions.*

Proof: It is obvious that E and E' have the same solutions if the transformation rule "Trivial" is applied. In case of the rule "Coalesce", E has the form $x \approx y, E_0$, and E' has the form $x \approx y, \phi(E_0)$ with $\phi = \{x \mapsto y\}$. Let σ be a solution of E. Then $\sigma(x) \leftrightarrow^*_{\mathcal{R}} \sigma(y) = \sigma(\phi(x))$. By definition of ϕ and the congruence property of $\leftrightarrow^*_{\mathcal{R}}$, $\sigma(t) \leftrightarrow^*_{\mathcal{R}} \sigma(\phi(t))$ for all terms t. Let $s \approx t \in E_0$. Since σ

is a solution of E, $\sigma(s) \leftrightarrow^*_{\mathcal{R}} \sigma(t)$. Moreover, $\sigma(s) \leftrightarrow^*_{\mathcal{R}} \sigma(\phi(s))$ and $\sigma(t) \leftrightarrow^*_{\mathcal{R}} \sigma(\phi(t))$ which implies $\sigma(\phi(s)) \leftrightarrow^*_{\mathcal{R}} \sigma(\phi(t))$. Therefore, σ is also a solution of $\phi(E_0)$.

If σ is a solution of E', it can be shown in a similar way that σ is also a solution of E_0. ■

Due to this proposition, the transformation $\stackrel{var}{\Longrightarrow}$ preserves solutions. Moreover, it is a terminating relation:

Proposition 3.4 *The relation $\stackrel{var}{\Longrightarrow}$ on equation systems is terminating.*

Proof: Define the complexity of an equation system as the total number of occurrences of unsolved variables in this system. Obviously, both transformation rules of $\stackrel{var}{\Longrightarrow}$ reduce this number. ■

If an equation system is quasi-solved, we can always transform it into a solved system:

Proposition 3.5 *Let E be a quasi-solved equation system. Then there exists a solved equation system E' with $E \stackrel{var}{\Longrightarrow}^* E'$.*

Proof: Let E be a quasi-solved equation system which is not solved. Then there exists an equation $x \approx y \in E$ which is unsolved. Hence $x = y$ or $x, y \in \mathcal{V}ar(E - \{x \approx y\})$. In the first case we apply the rule "Trivial" and in the second case we apply the rule "Coalesce". The result of both cases is a new equation system in quasi-solved form. Since there are no infinite derivations w.r.t. $\stackrel{var}{\Longrightarrow}$ (Proposition 3.4), successive transformation steps w.r.t. $\stackrel{var}{\Longrightarrow}$ will end in a solved equation system. ■

The solutions of an equation system in solved form can be obtained as follows:

Proposition 3.6 *Let E be an equation system in solved form, i.e.,*

$$E = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$$

where $x_1, \ldots, x_n$ are different variables with $x_i \notin \mathcal{V}ar(t_j)$ for $i, j \in \{1, \ldots, n\}$ (recall that equations are multisets, thus we can write solved systems always in this form). Then the substitution set

$$\{\gamma \circ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \mid \gamma \textit{ is a ground substitution with } \mathcal{D}om(\gamma) = \bigcup_{i=1}^{n} \mathcal{V}ar(t_i)\}$$

is a complete set of solutions for E.

Proof: First we show that $\theta := \gamma \circ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a solution of E for an arbitrary ground substitution γ with $\mathcal{D}om(\gamma) = \bigcup_{i=1}^{n} \mathcal{V}ar(t_i)$. Clearly, $\mathcal{D}om(\theta) = \{x_1, \ldots, x_n\} \cup \mathcal{D}om(\gamma) = \mathcal{V}ar(E)$. Consider the equation $x_i \approx t_i \in E$. Since $x_1, \ldots, x_n$ do not occur in any t_i, $\theta(x_i) = \gamma(t_i) = \theta(t_i)$, i.e., θ is a solution of $x_i \approx t_i$. Hence θ is a solution of E.

Next we show that every solution of E is covered by some substitution from the substitution set defined above. Let ξ be a solution of E. Then $\xi(x_i) =_{\mathcal{R}} \xi(t_i)$ for $i = 1, \ldots, n$. Since ξ is a ground substitution with $\mathcal{V}ar(E) \subseteq \mathcal{D}om(\xi)$, the substitution

$$\theta := \xi_{|\bigcup_{i=1}^{n} \mathcal{V}ar(t_i)} \circ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$$

is contained in the above substitution set. We have to show $\xi(x) =_{\mathcal{R}} \theta(x)$ for all $x \in \mathcal{V}ar(E)$:

- By definition of θ and ξ, $\theta(x_i) = \xi(t_i) =_{\mathcal{R}} \xi(x_i)$ for $i = 1, \ldots, n$.
- If $x \in \mathcal{V}ar(t_j)$ for some $j \in \{1, \ldots, n\}$, then $\theta(x) = \xi(x)$ by definition of θ (note that x is different from any x_i since no x_i occurs in t_j).

Altogether, $\theta(x) =_{\mathcal{R}} \xi(x)$ for all $x \in \mathcal{V}ar(E)$. ■

Due to Propositions 3.3, 3.5 and 3.6, it is sufficient to transform an equation system into a quasi-solved form in order to compute its solutions. Hence we can state soundness and completeness results by concentrating on quasi-solved forms. The next lemma shows the soundness if a transformation rule of the lazy unification calculus is applied.

Lemma 3.7 *Let E and E' be equation systems with $E \overset{lu}{\Longrightarrow} E'$. Then each solution σ of E' is also a solution of E.*

Proof: Assume that $E \overset{lu}{\Longrightarrow} E'$ and σ is a solution of E'. Clearly, $\mathcal{V}ar(E) \subseteq \mathcal{D}om(\sigma)$ since $\mathcal{V}ar(E) \subseteq \mathcal{V}ar(E') \subseteq \mathcal{D}om(\sigma)$. There are three cases corresponding to the applied transformation rule:

1. The lazy narrowing rule has been applied. Then $E = f(t_1, \ldots, t_n) \approx t, E_0$, $f(l_1, \ldots, l_n) \to r$ is a variant of a rewrite rule and $E' = t_1 \approx l_1, \ldots, t_n \approx l_n, r \approx t, E_0$. Since σ is a solution of E', $\sigma(t_i) \leftrightarrow^*_{\mathcal{R}} \sigma(l_i)$ (for $i = 1, \ldots, n$) and $\sigma(r) \leftrightarrow^*_{\mathcal{R}} \sigma(t)$. These equivalences imply $\sigma(f(t_1, \ldots, t_n)) \leftrightarrow^*_{\mathcal{R}} \sigma(f(l_1, \ldots, l_n))$ by the congruence property of $\leftrightarrow^*_{\mathcal{R}}$. Since $f(l_1, \ldots, l_n) \to r$ is a variant of a rewrite rule, $\sigma(f(l_1, \ldots, l_n)) \to_{\mathcal{R}} \sigma(r) \leftrightarrow^*_{\mathcal{R}} \sigma(t)$. Hence $\sigma(f(t_1, \ldots, t_n)) \leftrightarrow^*_{\mathcal{R}} \sigma(t)$, i.e., σ is a solution of E.
2. The decomposition rule has been applied. Then $E = f(t_1, \ldots, t_n) \approx f(t'_1, \ldots, t'_n), E_0$ and $E' = t_1 \approx t'_1, \ldots, t_n \approx t'_n, E_0$. Since σ is a solution of E', $\sigma(t_i) \leftrightarrow^*_{\mathcal{R}} \sigma(t'_i)$ (for $i = 1, \ldots, n$). Hence $\sigma(f(t_1, \ldots, t_n)) \leftrightarrow^*_{\mathcal{R}} \sigma(f(t'_1, \ldots, t'_n))$ by the congruence property of $\leftrightarrow^*_{\mathcal{R}}$.
3. The partial binding rule has been applied. Then $E = x \approx f(t_1, \ldots, t_n), E_0$ and $E' = x \approx f(x_1, \ldots, x_n), x_1 \approx \phi(t_1), \ldots, x_n \approx \phi(t_n), \phi(E_0)$ where $\phi = \{x \mapsto f(x_1, \ldots, x_n)\}$. Since σ is a solution of E', we have

   (a) $\sigma(x) \leftrightarrow^*_{\mathcal{R}} \sigma(f(x_1, \ldots, x_n))$

   (b) $\sigma(x_i) \leftrightarrow^*_{\mathcal{R}} \sigma(\phi(t_i))$ (for $i = 1, \ldots, n$)

   (c) σ solution of $\phi(E_0)$

   By definition of ϕ, (a) and the congruence property of $\leftrightarrow^*_{\mathcal{R}}$,

   $$\sigma(\phi(t)) \leftrightarrow^*_{\mathcal{R}} \sigma(t) \quad \text{for all terms } t \qquad (*)$$

   Hence σ is also a solution of E_0. Moreover,

   $$\begin{array}{rcll} \sigma(x) & \leftrightarrow^*_{\mathcal{R}} & \sigma(f(x_1, \ldots, x_n)) & \text{(by (a))} \\ & \leftrightarrow^*_{\mathcal{R}} & \sigma(f(\phi(t_1), \ldots, \phi(t_n))) & \text{(by (b))} \\ & \leftrightarrow^*_{\mathcal{R}} & \sigma(f(t_1, \ldots, t_n)) & \text{(by (*))} \end{array}$$

   Hence σ is a solution of $x \approx f(t_1, \ldots, t_n)$.

$\blacksquare$

The following soundness theorem can be proved by a simple induction on the transformation steps using the previous lemma.

Theorem 3.8 *Let E and E' be equation systems with $E \overset{lu}{\Longrightarrow}^* E'$. Then each solution σ of E' is a solution of E.*

The completeness proof is more difficult since we have to consider all possible transformation sequences. Therefore, we show that for each solution of an equation system there is a derivation into a quasi-solved form that has the same solution. Note that the solution of the quasi-solved form cannot be identical to the required solution, because new additional variables are generated during the derivation (by lazy narrowing and partial binding transformations). However, this is not a problem since we are interested in solutions w.r.t. the variables of the initial equation system.

Theorem 3.9 *Let E be a solvable equation system with solution σ. Then there exists a derivation $E \overset{lu}{\Longrightarrow}^* E'$ with E' in quasi-solved form such that E' has a solution σ' with $\sigma'(x) =_{\mathcal{R}} \sigma(x)$ for all $x \in \mathcal{V}ar(E)$.*

Proof: We show the existence of a derivation from E into a quasi-solved equation system by the following steps:

1. We define a reduction relation $\Rightarrow$ on pairs of the form (σ, E), where E is an equation system and σ is a solution of E, with the property that $(\sigma, E) \Rightarrow (\sigma', E')$ implies $E \overset{lu}{\Longrightarrow} E'$ and $\sigma'(x) = \sigma(x)$ for all $x \in \mathcal{V}ar(E)$.
2. We define a terminating ordering $\succ$ on these pairs.
3. We show: If E has a solution σ but E is not in quasi-solved form, then there exists a pair (σ', E') with $(\sigma, E) \Rightarrow (\sigma', E')$ and $(\sigma, E) \succ (\sigma', E')$.

2 and 3 implies that each solvable equation system can be transformed into a quasi-solved form. By 1, the solution of this quasi-solved form is the required solution of the initial equation system.

In the sequel we will show 1 and 3 in parallel. First we define the terminating ordering $\succ$. For this purpose we use the *strict subterm ordering* $\succ_{sst}$ on terms defined by $t \succ_{sst} s$ iff there is a position p in t with $t|_p = s \neq t$. Since R is a terminating rewrite system, the relation $\rightarrow_{\mathcal{R}}$ on terms is also terminating. Let $\ggcurly$ be the transitive closure of the relation $\rightarrow_{\mathcal{R}} \cup \succ_{sst}$. Then $\ggcurly$ is also terminating [28].[10] Now we define the following ordering on pairs (σ, E): $(\sigma, E) \succ (\sigma', E')$ iff

$$\{\sigma(s), \sigma(t) \mid s \approx t \in E \text{ is unsolved in } E\} \ggcurly_{mul} \{\sigma'(s'), \sigma'(t') \mid s' \approx t' \in E' \text{ is unsolved in } E'\} \quad (*)$$

where $\ggcurly_{mul}$ is the multiset extension[11] of the ordering $\ggcurly$ (all sets in this definition are multisets). $\ggcurly_{mul}$ is terminating (note that all multisets considered here are finite) since $\ggcurly$ is terminating [7].

[10]Note that the use of the relation $\rightarrow_{\mathcal{R}}$ instead of $\ggcurly$ (as done in [9]) is not sufficient for the completeness proof since $\rightarrow_{\mathcal{R}}$ has not the subterm property [7] in general.

[11]The multiset ordering $\ggcurly_{mul}$ is the transitive closure of the replacement of an element by a finite number of elements that are smaller w.r.t. $\ggcurly$ [7].

Now we will show that we can apply a transformation step to a solvable but unsolved equation system such that its complexity is reduced. Let E be an equation system not in quasi-solved form and σ be a solution of E. Since E is not quasi-solved, there must be an equation which has one of the following forms:

1. There is an equation $E = s \approx t, E_0$ with $s, t \notin \mathcal{X}$: Let $s = f(s_1, \ldots, s_n)$ with $n \geq 0$ (the other case is symmetric). Consider an innermost derivation of the normal forms of $\sigma(s)$ and $\sigma(t)$:

   (a) No rewrite step is performed at the root of $\sigma(s)$ and $\sigma(t)$: Then t has the form $t = f(t_1, \ldots, t_n)$ and $\sigma(s)\downarrow_{\mathcal{R}} = \sigma(t)\downarrow_{\mathcal{R}} = f(u_1, \ldots, u_n)$. Since $\sigma(s)$ and $\sigma(t)$ are not reduced at the root, $\sigma(s_i)\downarrow_{\mathcal{R}} = u_i = \sigma(t_i)\downarrow_{\mathcal{R}}$ for $i = 1, \ldots, n$. Now we apply the decomposition transformation and obtain the equation system

   $$E' \;=\; s_1 \approx t_1, \ldots, s_n \approx t_n, E_0$$

   Obviously, σ is a solution of E'. Moreover, the complexity of the new equation system is reduced because the equation $s \approx t$ is unsolved in E and each $\sigma(s_i)$ and $\sigma(t_i)$ is smaller than $\sigma(s)$ and $\sigma(t)$, respectively, since $\succ\!\!\succ$ contains the strict subterm ordering $\succ_{sst}$. Hence $(\sigma, E) \succ (\sigma, E')$.

   (b) A rewrite step is performed at the root of $\sigma(s)$, i.e., the innermost rewriting sequence of $\sigma(s)$ has the form

   $$\sigma(s) \rightarrow^*_{\mathcal{R}} f(s'_1, \ldots, s'_1) \rightarrow_{\mathcal{R}} \theta(r) \rightarrow^*_{\mathcal{R}} \sigma(s)\downarrow_{\mathcal{R}}$$

   where $f(l_1, \ldots, l_n) \rightarrow r$ is a new variant of a rewrite rule, $\theta(l_i) = s'_i$ and $\sigma(s_i) \rightarrow^*_{\mathcal{R}} s'_i$ for $i = 1, \ldots, n$. An application of the lazy narrowing transformation yields the equation system

   $$E' \;=\; s_1 \approx l_1, \ldots, s_n \approx l_n, r \approx t, E_0$$

   We combine σ and θ to a new substitution $\sigma' = \sigma \cup \theta$ (this is always possible since θ does only work on the variables of the new variant of the rewrite rule). Note that $\mathcal{V}ar(E') \subseteq \mathcal{D}om(\sigma')$. σ' is a solution of E' since

   $$\sigma'(s_i) = \sigma(s_i) \rightarrow^*_{\mathcal{R}} s'_i = \theta(l_i) = \sigma'(l_i)$$

   and

   $$\sigma'(r) = \theta(r) \rightarrow^*_{\mathcal{R}} \sigma(s)\downarrow_{\mathcal{R}} \leftrightarrow^*_{\mathcal{R}} \sigma(t) = \sigma'(t)$$

   Since the transitive closure of $\rightarrow_{\mathcal{R}}$ is contained in $\succ\!\!\succ$, $\sigma(s_i) \succ\!\!\succ \sigma'(l_i)$ (if $\sigma(s_i) \neq \sigma'(l_i)$) and $\sigma(s) \succ\!\!\succ \sigma'(r)$. Since $s \approx t$ is unsolved in E, the term $\sigma(s)$ is contained in the left multiset of the ordering definition $(*)$, and it is replaced by a selection of the smaller terms $\sigma(s_1), \ldots, \sigma(s_n), \sigma'(l_1), \ldots, \sigma'(l_n), \sigma'(r)$ ($\sigma(s) \succ\!\!\succ \sigma(s_i)$ since $\succ\!\!\succ$ contains the strict subterm ordering). Therefore, the new equation system is smaller w.r.t. $\succ$, i.e., $(\sigma, E) \succ (\sigma', E')$.

2. There is an equation $E = x \approx t, E_0$ with $t = f(t_1, \ldots, t_n)$ and x unsolved in E: Hence $x \in \mathcal{V}ar(t) \cup \mathcal{V}ar(E_0)$. Again, we consider an innermost derivation of the normal form of $\sigma(t)$:

(a) A rewrite step is performed at the root of $\sigma(t)$. Then we apply a lazy narrowing step and proceed as in the previous case.

(b) No rewrite step is performed at the root of $\sigma(t)$, i.e., $\sigma(t)\downarrow_{\mathcal{R}} = f(t'_1, \ldots, t'_n)$ and $\sigma(t_i)\downarrow_{\mathcal{R}} = t'_i$ for $i = 1, \ldots, n$. We apply the partial binding transformation and obtain the equation system

$$E' \;=\; x \approx f(x_1, \ldots, x_n), x_1 \approx \phi(t_1), \ldots, x_n \approx \phi(t_n), \phi(E_0)$$

where $\phi = \{x \mapsto f(x_1, \ldots, x_n)\}$ and x_i are new variables. We extend σ to a substitution σ' by adding the bindings $\sigma'(x_i) = t'_i$ for $i = 1, \ldots, n$, i.e., $\mathcal{V}ar(E') \subseteq \mathcal{D}om(\sigma')$. Then

$$\sigma'(f(x_1, \ldots, x_n)) = f(t'_1, \ldots, t'_n) = \sigma(t)\downarrow_{\mathcal{R}} \leftrightarrow^*_{\mathcal{R}} \sigma(t) \leftrightarrow^*_{\mathcal{R}} \sigma(x) = \sigma'(x)$$

Moreover, $\sigma'(\phi(x)) = \sigma'(x)\downarrow_{\mathcal{R}}$ which implies $\sigma'(s) \leftrightarrow^*_{\mathcal{R}} \sigma'(\phi(s))$ for all terms s. Hence $\sigma'(\phi(t_i)) \leftrightarrow^*_{\mathcal{R}} \sigma'(t_i) \leftrightarrow^*_{\mathcal{R}} t'_i = \sigma'(x_i)$. Altogether, σ' is a solution of E'.

It remains to show that this transformation reduces the complexity of the equation system. Since $\sigma'(\phi(x)) = \sigma(x)\downarrow_{\mathcal{R}}$, we have $\sigma(x) \rightarrow^*_{\mathcal{R}} \sigma'(\phi(x))$. Hence $\sigma(E_0)$ is equal to $\sigma'(\phi(E_0))$ (if $\sigma(x) = \sigma'(\phi(x))$) or $\sigma'(\phi(E_0))$ is smaller w.r.t. $\ggcurly_{mul}$. Therefore, it remains to check that $\sigma(t)$ is greater than each $\sigma'(x_1), \ldots, \sigma'(x_n), \sigma'(\phi(t_1)), \ldots, \sigma'(\phi(t_n))$ w.r.t. $\ggcurly$ (note that the equation $x \approx t$ is unsolved in E, but the equation $x \approx f(x_1, \ldots, x_n)$ is solved in E'). First of all, $\sigma(t) \ggcurly \sigma(t_i)$ since $\ggcurly$ includes the strict subterm ordering. Moreover, $\sigma(t_i) \rightarrow^*_{\mathcal{R}} \sigma'(x_i)$, i.e., $\sigma'(x_i)$ is equal or smaller than $\sigma(t_i)$ w.r.t. $\ggcurly$ for $i = 1, \ldots, n$. This implies $\sigma(t) \ggcurly \sigma'(x_i)$. Similarly, $\sigma'(\phi(t_i))$ is equal or smaller than $\sigma(t_i)$ w.r.t. $\ggcurly$ since $\sigma'(\phi(x)) = \sigma(x)\downarrow_{\mathcal{R}}$. Thus $\sigma(t) \ggcurly \sigma'(\phi(t_i))$. Altogether, $(\sigma, E) \succ (\sigma', E')$.

■

We want to point out that there exist also other orderings on substitution/equation system pairs to prove the completeness of our calculus. However, the ordering chosen above is tailored to a simple proof for the completeness of lazy unification with simplification as we will see in the next section.

Propositions 3.3, 3.5, 3.6 and Theorems 3.8 and 3.9 imply that a complete set of solutions for a given equation system E can be computed by enumerating all derivations in the lazy unification calculus from E into a quasi-solved equation system. Due to the nondeterminism in the lazy unification calculus, there are many unsuccessful and often infinite derivations. Therefore, we will show in the next section how to reduce this nondeterminism by integrating a deterministic simplification process into the lazy unification calculus. More determinism can be achieved by dividing the set of function symbols into constructors and defined functions. This will be the subject of Section 3.3.

3.2 Integrating Simplification Into Lazy Unification

The lazy unification calculus admits a high degree of nondeterminism even if there is only one reasonable derivation. This is due to the fact that functional expressions are processed “too lazy”.

Example 3.10 Consider the rewrite rules

$$\begin{array}{rclcrcl} f(a) & \rightarrow & c & \qquad & g(a) & \rightarrow & a \\ f(b) & \rightarrow & d & & g(b) & \rightarrow & b \end{array}$$

and the equation $f(g(b)) \approx d$. Then there are the following four different derivations in our lazy unification calculus:

$$
\begin{array}{lclclcl}
f(g(b)) \approx d & \stackrel{lu}{\Longrightarrow} & g(b) \approx a,\ c \approx d & \stackrel{lu}{\Longrightarrow} & b \approx a,\ a \approx a,\ c \approx d & \stackrel{lu}{\Longrightarrow} & b \approx a,\ c \approx d \\
f(g(b)) \approx d & \stackrel{lu}{\Longrightarrow} & g(b) \approx a,\ c \approx d & \stackrel{lu}{\Longrightarrow} & b \approx b,\ b \approx a,\ c \approx d & \stackrel{lu}{\Longrightarrow} & b \approx a,\ c \approx d \\
f(g(b)) \approx d & \stackrel{lu}{\Longrightarrow} & g(b) \approx b,\ d \approx d & \stackrel{lu}{\Longrightarrow} & b \approx a,\ a \approx b,\ d \approx d & \stackrel{lu}{\Longrightarrow} & b \approx a,\ a \approx b \\
f(g(b)) \approx d & \stackrel{lu}{\Longrightarrow} & g(b) \approx b,\ d \approx d & \stackrel{lu}{\Longrightarrow} & b \approx b,\ b \approx b,\ d \approx d & \stackrel{lu}{\Longrightarrow}{}^{*} & \emptyset
\end{array}
$$

The first three derivations do not end in a quasi-solved form, only the last derivation is successful. However, if we first compute the normal form of $f(g(b))$, which is d, then there is only one possible derivation: $d \approx d \stackrel{lu}{\Longrightarrow} \emptyset$. Hence we will show that the lazy unification calculus remains to be sound and complete if the (deterministic!) normalization of terms is included. □

It is well-known [14, 21] that the inclusion of inductive consequences for normalization may have an essential effect on the search space reduction in normalizing narrowing strategies. Therefore, we will also allow the use of additional inductive consequences for normalization. A rewrite rule $l \to r$ is called *inductive consequence* (of $\mathcal{R}$) if $\sigma(l) =_{\mathcal{R}} \sigma(r)$ for all ground substitutions σ with $\mathcal{D}om(\sigma) = \mathcal{V}ar(l)$. For instance, the rule $x + 0 \to x$ is an inductive consequence of the term rewriting system

$$
\begin{array}{rcl}
0 + y & \to & y \\
s(x) + y & \to & s(x + y)
\end{array}
$$

If we want to solve the equation $s(x)+0 \approx s(x)$, our basic lazy unification calculus would enumerate the solutions $\{x \mapsto 0\}$, $\{x \mapsto s(0)\}$, $\{x \mapsto s(s(0))\}$, and so on, i.e., this equation has an infinite search space. Using the inductive consequence $x+0 \to x$ for normalization, the equation $s(x)+0 \approx s(x)$ is reduced to $s(x) \approx s(x)$ and then transformed into the quasi-solved form $x \approx x$ representing the solution set where x is replaced by any ground term.[12]

In the following, we assume that $\mathcal{I}$ is a set of inductive consequences of $\mathcal{R}$ (the set of *simplification rules*) so that the rewrite relation $\to_{\mathcal{I}}$ is terminating. We will use rules from $\mathcal{R}$ for lazy narrowing and rules from $\mathcal{I}$ for simplification. Note that each rule from $\mathcal{R}$ is also an inductive consequence and can be included in $\mathcal{I}$. However, we do not require that all rules from $\mathcal{R}$ must be used for normalization. This is reasonable if there are duplicating rules where one variable of the left-hand side occurs several times on the right-hand side, like $f(x) \to g(x, x)$. If we normalize the equation $f(s) \approx t$ with this rule, then the term s is duplicated. This may increase the computational costs if the evaluation of s is necessary and costly. In such a case it would be better to use this rule only in lazy narrowing steps.

In order to include simplification into the lazy unification calculus, we define a relation $\Rightarrow_{\mathcal{I}}$ on systems of equations. $s \approx t \Rightarrow_{\mathcal{I}} s' \approx t'$ iff s' and t' are normal forms of s and t w.r.t. $\to_{\mathcal{I}}$, respectively. $E \Rightarrow_{\mathcal{I}} E'$ iff $E = e_1, \ldots, e_n$ and $E' = e'_1, \ldots, e'_n$ where $e_i \Rightarrow_{\mathcal{I}} e'_i$ for $i = 1, \ldots, n$. Note

[12] In larger single-sorted term rewriting systems, it can be difficult to find inductive consequences. E.g., $x + 0 \to x$ is not an inductive consequence if there is a constant a since $a + 0 =_{\mathcal{R}} a$ is not valid. However, in practice specifications are many-sorted and then inductive consequences must be valid only for all well-sorted ground substitutions. Therefore, we want to point out that all results in this paper can also be extended to many-sorted term rewriting systems in a straightforward way.

that $\Rightarrow_{\mathcal{I}}$ describes a deterministic computation process.[13] $E \overset{lus}{\Longrightarrow} E'$ is a derivation step in the *lazy unification calculus with simplification* if $E \Rightarrow_{\mathcal{I}} \overline{E} \overset{lu}{\Longrightarrow} E'$ for some $\overline{E}$.

The following lemma shows the soundness of one rewrite step with a simplification rule. The formulation of soundness differs from Lemma 3.7 since we have to consider the fact that goal variables may be deleted by normalization.

Lemma 3.11 *Let $s \approx t$ be an equation, $s \rightarrow_{\mathcal{I}} s'$ be a rewrite step, and σ' be a solution of $s' \approx t$. Then any ground substitution σ with $\mathcal{V}ar(s \approx t) \subseteq \mathcal{D}om(\sigma)$ and $\sigma(x) =_{\mathcal{R}} \sigma'(x)$ for all $x \in \mathcal{V}ar(s' \approx t)$ is a solution of $s \approx t$.*

Proof: Let $s \rightarrow_{\mathcal{I}} s'$ and σ' be a solution of $s' \approx t$, i.e., $\sigma(s') =_{\mathcal{R}} \sigma(t)$. We consider a ground substitution σ with $\mathcal{V}ar(s \approx t) \subseteq \mathcal{D}om(\sigma)$ and $\sigma(x) =_{\mathcal{R}} \sigma'(x)$ for all $x \in \mathcal{V}ar(s' \approx t)$. Obviously, $\sigma(s) \rightarrow_{\mathcal{I}} \sigma(s')$ using the same rewrite rule from $\mathcal{I}$. Hence $\sigma(s) =_{\mathcal{R}} \sigma(s')$ since $\mathcal{I}$ consists of inductive consequences of $\mathcal{R}$ and $\sigma(s)$ and $\sigma(s')$ are ground terms. By $\sigma(s') =_{\mathcal{R}} \sigma(t)$, this implies $\sigma(s) =_{\mathcal{R}} \sigma(t)$, i.e., σ is a solution of $s \approx t$. ■

Now we can state the soundness of the calculus $\overset{lus}{\Longrightarrow}$:

Theorem 3.12 *Let E and E' be equation systems with $E \overset{lus}{\Longrightarrow}{}^{*} E'$ where E' is in quasi-solved form, and σ' be a solution of E'. Then any ground substitution σ with $\mathcal{V}ar(E) \subseteq \mathcal{D}om(\sigma)$ and $\sigma(x) =_{\mathcal{R}} \sigma'(x)$ for all $x \in \mathcal{V}ar(E')$ is a solution of E.*

Proof: By Lemma 3.11, we can show the soundness of $\Rightarrow_{\mathcal{I}}$ with a simple induction on the sequence of rewrite steps. Combining this result with Lemma 3.7 shows the soundness of one $\overset{lus}{\Longrightarrow}$ step. Then the theorem follows by another simple induction on the number of $\overset{lus}{\Longrightarrow}$ steps. ■

For the completeness proof we have to show that solutions are not lost by the application of simplification rules:

Lemma 3.13 *Let E be an equation system and σ be a solution of E. If $E \Rightarrow_{\mathcal{I}} E'$, then σ is a solution of E'.*

Proof: By definition of rewrite rules, $\mathcal{V}ar(E') \subseteq \mathcal{V}ar(E)$. Let $s \approx t \in E$, $\sigma(s) =_{\mathcal{R}} \sigma(t)$ and $s \approx t \Rightarrow_{\mathcal{I}} s' \approx t'$. Hence $s \rightarrow_{\mathcal{I}}^{*} s'$ and $t \rightarrow_{\mathcal{I}}^{*} t'$ which implies $\sigma(s) \rightarrow_{\mathcal{I}}^{*} \sigma(s')$ and $\sigma(t) \rightarrow_{\mathcal{I}}^{*} \sigma(t')$. Since σ is a ground substitution with $\mathcal{V}ar(E) \subseteq \mathcal{D}om(\sigma)$ and $\mathcal{I}$ are inductive consequences, $\sigma(s) =_{\mathcal{R}} \sigma(s')$ and $\sigma(t) =_{\mathcal{R}} \sigma(t')$. Hence $\sigma(s') =_{\mathcal{R}} \sigma(t')$, i.e., σ is a solution of all equations in E'. ■

The last lemma would imply the completeness of the calculus $\overset{lus}{\Longrightarrow}$ if a derivation step with $\Rightarrow_{\mathcal{I}}$ does not increase the ordering used in the proof of Theorem 3.9. Unfortunately, this is not the case in general since the termination of $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{I}}$ may be based on different orderings (e.g., $\mathcal{R} = \{a \rightarrow b\}$ and $\mathcal{I} = \{b \rightarrow a\}$). In order to avoid such problems, we require that the relation $\rightarrow_{\mathcal{R} \cup \mathcal{I}}$ is terminating which is not a real restriction in practice.

[13] If there exist more than one normal form w.r.t. $\rightarrow_{\mathcal{I}}$, it is sufficient to select *don't care* one of these normal forms.

Theorem 3.14 *Let $\mathcal{I}$ be a set of inductive consequences of the ground confluent and terminating rewrite system $\mathcal{R}$ such that $\rightarrow_{\mathcal{R}\cup\mathcal{I}}$ is terminating. Let E be a solvable equation system with solution σ. Then there exists a derivation $E \stackrel{lus}{\Longrightarrow}^* E'$ such that E' is in quasi-solved form and has a solution σ' with $\sigma'(x) =_{\mathcal{R}} \sigma(x)$ for all $x \in \mathcal{V}ar(E)$.*

Proof: In the proof of Theorem 3.9, we have shown how to apply a transformation step to an equation system not in quasi-solved form such that the solution is preserved. We can use the same proof for the transformation $\stackrel{lus}{\Longrightarrow}$ since Lemma 3.13 shows that normalization steps preserve solutions. The only difference concerns the ordering where we use $\rightarrow_{\mathcal{R}\cup\mathcal{I}}$ instead of $\rightarrow_{\mathcal{R}}$, i.e., $\ggcurly$ is now defined to be the transitive closure of the relation $\rightarrow_{\mathcal{R}\cup\mathcal{I}} \cup \succ_{sst}$. Clearly, this does not change anything in the proof of Theorem 3.9. Moreover, the relation $\Rightarrow_{\mathcal{I}}$ does not increase the complexity w.r.t. this ordering but reduces it if simplification rules are applied since $\rightarrow_{\mathcal{I}}$ is contained in $\ggcurly$. ■

Theorems 3.12 and 3.14 show that we can integrate the deterministic simplification process into the lazy unification calculus without loosing soundness and completeness. Note that the rules from $\mathcal{I}$ can only be applied if their left-hand sides can be matched with a subterm of the current equation system. If these subterms are not sufficiently instantiated, the rewrite rules are not applicable and hence we loose potential determinism in the unification process.

Example 3.15 Consider the rules

$$\begin{array}{rcl} zero(s(x)) & \rightarrow & zero(x) \\ zero(0) & \rightarrow & 0 \end{array}$$

(assume that these rules are contained in $\mathcal{R}$ as well as in $\mathcal{I}$) and the equation system $zero(x) \approx 0, x \approx 0$. Then there exists the following derivation in our calculus (this derivation is also possible in the unification calculi in [15, 34]):

$$\begin{array}{lll} \multicolumn{2}{l}{zero(x) \approx 0,\ x \approx 0} & \\ \stackrel{lus}{\Longrightarrow} & x \approx s(x_1),\ zero(x_1) \approx 0,\ x \approx 0 & (\textit{lazy narrowing with first rule}) \\ \stackrel{lus}{\Longrightarrow} & x \approx s(x_1),\ x_1 \approx s(x_2),\ zero(x_2) \approx 0,\ x \approx 0 & (\textit{lazy narrowing with first rule}) \\ \stackrel{lus}{\Longrightarrow} & x \approx s(x_1),\ x_1 \approx s(x_2),\ x_2 \approx s(x_3),\ zero(x_3) \approx 0,\ x \approx 0 & (\textit{lazy narrowing with first rule}) \\ \stackrel{lus}{\Longrightarrow} & \cdots & \end{array}$$

This infinite derivation could be avoided if we apply the partial binding rule in the first step:

$$\begin{array}{rcll} zero(x) \approx 0,\ x \approx 0 & \stackrel{lus}{\Longrightarrow} & zero(0) \approx 0,\ x \approx 0 & (\textit{partial binding}) \\ & \Rightarrow_{\mathcal{I}} & 0 \approx 0,\ x \approx 0 & (\textit{rewriting with second rule}) \\ & \stackrel{lus}{\Longrightarrow} & x \approx 0 & (\textit{decomposition}) \end{array}$$

In the next section we will present an optimization which prefers the latter derivation and avoids the first infinite derivation. □

3.3 Constructor-based Systems

In most existing functional logic programming languages, a distinction is made between operation symbols to construct data terms, called *constructors*, and operation symbols to operate on data

Decomposition of constructor equations

$c(t_1,\ldots,t_n) \approx c(t'_1,\ldots,t'_n), E \quad \overset{luc}{\Longrightarrow} \quad t_1 \approx t'_1,\ldots,t_n \approx t'_n, E$

if $c \in \mathcal{C}$

Full binding of variables to ground constructor terms

$x \approx t, E \quad \overset{luc}{\Longrightarrow} \quad x \approx t, \phi(E)$

if $x \in \mathcal{V}ar(E)$, $t \in \mathcal{T}(\mathcal{C},\emptyset)$ and $\phi = \{x \mapsto t\}$

Partial binding of variables to constructor terms

$x \approx c(t_1,\ldots,t_n), E \quad \overset{luc}{\Longrightarrow} \quad x \approx c(x_1,\ldots,x_n), x_1 \approx \phi(t_1),\ldots,x_n \approx \phi(t_n), \phi(E)$

if $c \in \mathcal{C}$, $x \in \mathcal{V}ar(c(t_1,\ldots,t_n)) \cup \mathcal{V}ar(E)$, $x \notin cvar(c(t_1,\ldots,t_n))$ and $\phi = \{x \mapsto c(x_1,\ldots,x_n)\}$
(x_i new variable)

Figure 3: Deterministic transformations for constructor-based rewrite systems

terms, called *defined functions* or *operations* (see, for instance, the functional logic languages ALF [19], BABEL [37], K-LEAF [17], SLOG [14], or the RAP system [16]). Such a distinction allows to optimize our unification calculus. Therefore, we assume in this section that the signature $\mathcal{F}$ is divided into two sets $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$, called constructors and defined functions, with $\mathcal{C} \cap \mathcal{D} = \emptyset$. A *constructor term* t is built from constructors and variables, i.e., $t \in \mathcal{T}(\mathcal{C},\mathcal{X})$. The distinction between constructors and defined functions comes with the restriction that for all rewrite rules $l \to r$ the outermost symbol of l is always a defined function.[14]

A basic property of such constructor-based term rewriting systems is the irreducibility of constructor terms. Due to this fact, we can specialize the rules of our basic lazy unification calculus. Therefore, we define the deterministic transformations in Figure 3. *Deterministic transformations* are intended to be applied as long as possible before any transformation $\overset{lu}{\Longrightarrow}$ is used. Hence they can be integrated into the deterministic normalization process $\Rightarrow_{\mathcal{I}}$. It is obvious that this modification preserves soundness and completeness. The decomposition transformation for constructor equations must be applied in any case in order to obtain a quasi-solved equation system since a lazy narrowing step $\mathcal{R}$ cannot be applied to constructor equations. The full binding of variables to ground constructor terms is an optimization which combines subsequent applications of partial binding transformations. This transformation decreases the complexity used in the proof of Theorem 3.14 since a constructor term is always in normal form. The partial binding transformation for constructor terms performs an eager (partial) binding of variables to constructor terms since a lazy narrowing step cannot be applied to the constructor term. Moreover, this binding transformation is combined with an *occur check* since it cannot be applied if $x \in cvar(c(t_1,\ldots,t_n))$ where *cvar*

[14]In constructor-based systems, it is often required that all rules have the form $f(t_1,\ldots,t_n) \to r$ with $f \in \mathcal{D}$ and $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{C},\mathcal{X})$. However, this stronger requirement is not necessary for the results in this section.

Clash of constructor equations

$c(t_1, \ldots, t_n) \approx d(t'_1, \ldots, t'_m), E \quad \stackrel{luc}{\Longrightarrow} \quad \text{FAIL}$

if $c, d \in \mathcal{C}$ and $c \neq d$

Occur check

$x \approx c(t_1, \ldots, t_n), E \quad \stackrel{luc}{\Longrightarrow} \quad \text{FAIL}$

if $x \in cvar(c(t_1, \ldots, t_n))$

Figure 4: Failure rules for constructor-based rewrite systems

denotes the set of all variables occurring outside terms headed by defined function symbols:

$$\begin{array}{rcll} cvar(x) & = & \{x\} & \\ cvar(c(t_1, \ldots, t_n)) & = & \bigcup_{i=1}^{n} cvar(t_i) & \text{if } c \in \mathcal{C} \\ cvar(f(t_1, \ldots, t_n)) & = & \emptyset & \text{if } f \in \mathcal{D} \end{array}$$

This restriction avoids infinite derivations of the following kind:

$$\begin{array}{rcll} x \approx c(x) & \stackrel{lu}{\Longrightarrow} & x \approx c(x_1),\ x_1 \approx c(x_1) & (partial\ binding) \\ & \stackrel{lu}{\Longrightarrow} & x \approx c(x_1),\ x_1 \approx c(x_2),\ x_2 \approx c(x_2) & (partial\ binding) \\ & \stackrel{lu}{\Longrightarrow} & \cdots & \end{array}$$

It is obvious that an equation of the form $x \approx c(t_1, \ldots, t_n)$ with $x \in cvar(c(t_1, \ldots, t_n))$ is unsolvable.

A further optimization can be added if all functions are reducible on ground constructor terms, i.e., for all $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \emptyset)$ there exists a term t with $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{R}} t$. In this case all ground terms have a ground constructor normal form and the partial binding transformation of $\stackrel{lu}{\Longrightarrow}$ can be completely omitted which increases the determinism in the lazy unification calculus.

If we invert the deterministic transformation rules, we obtain a set of failure rules shown in Figure 4. *Failure rules* are intended to be tried during the deterministic transformations. If a failure rule is applicable, the derivation can be safely terminated since the equation system cannot be transformed into a quasi-solved system.

3.4 Using Inductive Consequences

In Section 5 we will discuss the advantages of using program rules for simplification between lazy narrowing or unification steps for various classes of functional logic programs. Therefore, we provide in this section only an example which demonstrates the advantages of using inductive consequences for simplification in our lazy unification calculus. Since inductive consequences are only used for simplification, they do not increase the search space. Formally, this is confirmed by the fact that lazy unification derivations correspond to rewrite derivations (Lemma 3.7) and the application of inductive consequences reduces the complexity of goals (Theorem 3.14).

Example 3.16 Consider the following rewrite rules for addition and multiplication on natural numbers where $\mathcal{C} = \{0, s\}$ are constructors and $\mathcal{D} = \{+, *\}$ are defined functions:

$$\begin{array}{rclcrclc} 0+y & \to & y & (R_1) & \qquad 0*y & \to & 0 & (R_3) \\ s(x)+y & \to & s(x+y) & (R_2) & \qquad s(x)*y & \to & y+x*y & (R_4) \end{array}$$

If we use this confluent and terminating set of rewrite rules for lazy narrowing ($\mathcal{R}$) as well as for normalization ($\mathcal{I}$) and add the inductive consequence $x * 0 \to 0$ to $\mathcal{I}$, then our lazy unification calculus with simplification has a finite search space for the equation $x * y \approx s(0)$. This is due to the fact that the following derivation can be terminated using the inductive consequence and the clash rule:

$$\begin{array}{rll} x*y = s(0) & \stackrel{lu}{\Longrightarrow} \; x \approx s(x_1),\; y \approx y_1,\; y_1 + x_1 * y_1 \approx s(0) & (\textit{lazy narrowing, } R_4) \\ & \stackrel{lu}{\Longrightarrow} \; x \approx s(x_1),\; y \approx y_1,\; y_1 \approx 0,\; x_1 * y_1 \approx y_2,\; y_2 \approx s(0) & (\textit{lazy narrowing, } R_1) \\ & \stackrel{luc}{\Longrightarrow} \; x \approx s(x_1),\; y \approx 0,\; y_1 \approx 0,\; x_1 * 0 \approx y_2,\; y_2 \approx s(0) & (\textit{bind variable } y_1) \\ & \stackrel{luc}{\Longrightarrow} \; x \approx s(x_1),\; y \approx 0,\; y_1 \approx 0,\; x_1 * 0 \approx s(0),\; y_2 \approx s(0) & (\textit{bind variable } y_2) \\ & \Rightarrow_{\mathcal{I}} \; x \approx s(x_1),\; y \approx 0,\; y_1 \approx 0,\; 0 \approx s(0),\; y_2 \approx s(0) & (\textit{reduce } x_1 * 0) \\ & \stackrel{luc}{\Longrightarrow} \; \text{FAIL} & (\textit{clash between } 0 \textit{ and } s) \end{array}$$

The equation $x_1 * 0 \approx s(0)$ could not be transformed into the equation $0 \approx s(0)$ without the inductive consequence. Consequently, an infinite derivation would occur in our basic unification calculus of Section 3.1.

Note that other lazy unification calculi [15, 34] or lazy narrowing calculi [37, 41] have an infinite search space for this equation. It is also interesting to note that a normalizing innermost narrowing strategy as in [14, 20] has also an infinite search space even if the same simplification rules are available. This shows the advantage of combining a lazy strategy with simplification. □

4 Rewrite Systems with Nonterminating Rules

In this section we consider rewrite systems which are not necessarily terminating. Similarly to lazy evaluation in functional languages, lazy narrowing has at least two advantages in comparison to other (eager) narrowing strategies:

1. Since lazy narrowing applies narrowing steps at inner positions only if it is demanded by some rule, useless narrowing steps (steps at inner positions which do not contribute to the result) are avoided.[15]

2. Since lazy narrowing evaluates functions only if their results are demanded, it can deal with nonterminating functions and infinite data structures. Other narrowing strategies (like basic, innermost, or outermost narrowing) require a terminating set of rewrite rules and cannot deal with infinite data structures.

The next example should emphasize the latter point.

[15] To be precise, the avoidance of useless narrowing steps depends on the lazy narrowing strategy. Although this is one of the motivations of all lazy strategies, the only strategy for which this property has been formally proved is needed narrowing [2].

Example 4.1 The following rules define a function $from(n)$, which computes an infinite list of naturals starting from n, and a function $first(n, l)$, which computes the first n elements of the list l ($[]$ denotes the empty list and $[e|l]$ denotes a nonempty list with first element e and tail l):

$$\begin{array}{rcl} from(n) & \to & [n|from(s(n))] \\ first(0, l) & \to & [] \\ first(s(n), [e|l]) & \to & [e|first(n, l)] \end{array}$$

The first rule of this rewrite system is nonterminating. Lazy evaluation of the expression $first(s(s(0)), from(0))$ yields the result $[0, s(0)]$, whereas an eager evaluation does not terminate due to the nonterminating eager evaluation of $from(0)$. Similarly, lazy narrowing applied to the equation $first(x, from(y)) \approx [0, s(0)]$ computes the solution $\{x \mapsto s(s(0)), y \mapsto 0\}$, whereas an eager narrowing strategy runs into an infinite loop. □

Since narrowing applies rules only in one direction from left to right, the confluence of the rewrite relation is an essential requirement for the completeness of all narrowing strategies. However, confluence is an undecidable property of a rewrite system if it is not terminating. Therefore, functional logic languages with nonterminating rewrite systems have the following requirements on rewrite rules:

1. *Constructor-based:* The signature $\mathcal{F}$ is divided into two disjoint sets $\mathcal{C}$ and $\mathcal{D}$, called *constructors* and *defined functions*. Moreover, if $l \to r$ is a rewrite rule, then l has the form $f(t_1, \ldots, t_n)$ with $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

2. *Left-linearity:* All rules are left-linear, i.e., no variable appears more than once in the left-hand side of any rule.

3. *Nonambiguity:* If $l_1 \to r_1$ and $l_2 \to r_2$ are two different rules, then l_1 and l_2 are not unifiable (*strong nonambiguity*), or if l_1 and l_2 have a most general unifier σ, then $\sigma(r_1)$ and $\sigma(r_2)$ are identical (*weak nonambiguity*).

Rewrite systems with these properties are called *constructor-based (weakly) orthogonal systems.* These conditions ensure the uniqueness of normal forms if they exist. Due to the presence of nonterminating functions, the completeness results for lazy strategies are stated with respect to domain-based interpretations of rewrite rules [17, 37]. In particular, the equality of two expressions holds only if both sides are reducible to the same ground constructor term. The completeness of lazy narrowing w.r.t. this semantics is formally stated in [37]. We will show that deterministic simplification steps can be included in lazy narrowing derivations without destroying completeness for such rewrite systems, i.e., we assume that $\mathcal{R}$ is a constructor-based weakly orthogonal term rewriting system.

Loogen and Winkler [32] have shown how to increase deterministic computations in the implementation of such programs: if no goal variable has been bound in a narrowing step, then all attempts to apply alternative rules at the same position can be ignored due to the nonambiguity of the rules. In this case a "cut" can be executed to eliminate the choice point for alternative rules. Since the execution of this "cut" depends on the run-time behavior of the program (whether or not a goal variable has been bound during unification), it is called *dynamic cut* in [32]. The dynamic cut can be implemented by a special `POP` instruction which checks whether a goal variable has been bound during unification and, if this did not happen, removes the last choice point. The advantage of this method is its simple implementation, but it has also two disadvantages:

1. The dynamic cut removes choice points which have been created but are not needed in the further computation process. Hence it does not avoid the *creation* of choice points (one of the most expensive operations in the implementation of logic languages): if a choice point is not needed in a deterministic computation, it is created and then deleted after the unification of the rule's left-hand side.

2. The detection of deterministic computations depends on the order of the rules. If a rule which enables a deterministic computation step is not at the beginning, nondeterministic steps may be performed even if a deterministic step is possible.

The following example discusses the second disadvantage in more detail.

Example 4.2 Consider the rules of Example 1.2 and the goal equation $0 * one(x) \approx 0$. Using the dynamic cut technique, first a choice point for the rules R_3 and R_4 is created, rule R_3 is applied to narrow the left-hand side yielding the trivial equation $0 \approx 0$, and then the choice point is removed since no goal variable (x) has been bound in the narrowing step (dynamic cut). Hence the attempt to apply rule R_4 is avoided by the dynamic cut. However, if we try to solve the equation $one(x) * 0 \approx 0$, the dynamic cut has no effect. As before, first a choice point for the rules R_3 and R_4 is created, then an attempt to apply rule R_3 is made.[16] Since it is necessary to evaluate the first argument in order to decide the applicability of this rule, $one(x)$ is a lazy narrowing redex which is evaluated by applying rules R_5 or R_6 (this evaluation has an infinite search space and does not terminate in a sequential implementation, cf. Example 1.2). In any case the goal variable x will be bound and therefore the dynamic cut has no effect. □

Although the dynamic cut has some disadvantages since it is applied *after* a narrowing attempt, the nonambiguity of the rules is the key to exploit deterministic computations in functional logic programs. In the following we will show that we can apply deterministic rewrite steps before a narrowing step. This technique avoids the creation of superfluous choice points and is independent on the order of rules (if we use all rules also for rewrite steps).

The next lemma is due to Loogen and Winkler [32] and shows that it is not necessary to consider alternative rules for narrowing if one rule is applicable without binding goal variables. This is a consequence of the nonambiguity condition on rewrite rules.

Lemma 4.3 *Let $R_1 = l_1 \to r_1$ and $R_2 = l_2 \to r_2$ be two different program rules and t be a term which has no variables in common with R_1 and R_2. If $\sigma(l_1) = t$, i.e., t is narrowable by rule R_1 without instantiating any goal variable, then rule R_2 need not be considered, because either R_2 is not applicable or the result of applying R_2 yields an instance of the application of R_1.*

This means that lazy narrowing is complete in the sense of [37] even if the lazy narrowing derivation starting with an application of rule R_2 to t is ignored. Hence we could try to match the left-hand

[16] Note that we consider a sequential implementation where the rules are applied in the given textual order.

side of some rule with the current goal before applying a narrowing step. If this is possible, we can perform the corresponding rewrite step and, by the previous lemma, ignore all other rules, i.e., we perform a deterministic computation step. Although this solves the problems exemplified in Example 4.2, it is not sufficient to exploit many possible deterministic computations. In general, rewrite steps must also be performed at inner positions in order to enable rewrite steps at outer positions. For instance, consider the rules of Examples 1.1 and 1.2 and the goal equation $(0+0)*z \approx 0$. A rewrite step by applying rules R_3 or R_4 to the left-hand side of the equation is not possible. Hence we try to perform a narrowing step, i.e., generate a choice point for the rules R_3 or R_4, and so on. However, if we apply a rewrite step to the subterm $(0+0)$ before the narrowing attempt, the equation is simplified to $0*z \approx 0$ using rule R_1, and we could further simplify the equation to the trivial one $0 \approx 0$ using rule R_3. Therefore, we could solve the equation without any nondeterministic narrowing step. The following lemma shows that deterministic rewrite steps at inner positions does not influence the applicability of narrowing steps at outer positions.

Lemma 4.4 *Let t, t' be terms such that $t \rightarrow_{\mathcal{R}} t'$ is a rewrite step at position p. Then all narrowing rules which are applicable to t at a position p', where $p' \neq p$ is a position not below p, are also applicable to t' with the same substitution of variables occurring in t.*

Proof: This lemma is a consequence of the requirement for constructor-based rules: the subterm $t|_p$ must have a defined function symbol at the top since $t \rightarrow_{\mathcal{R}} t'$ is a rewrite step at position p. If a narrowing rule is applicable to t at position p', i.e., there is a rule $l \rightarrow r$ and a mgu σ of $t|_{p'}$ and l, and p' is a position above p (the case of independent positions is trivial since variables in t are not instantiated by the rewrite step), then there must be a variable position p'' in l (i.e., $l|_{p''} \in \mathcal{X}$) such that $\sigma(l)|_{p''}$ contains the subterm $t|_p$ (since all proper subterms of l contain only constructors and variables). But then there is also a unifier σ' of $t'|_{p'}$ and l which can be obtained by modifying σ for the variable $l|_{p''}$ (note that l has no multiple occurrences of variables, hence $\sigma'_{|\mathcal{V}ar(t)} = \sigma_{|\mathcal{V}ar(t)}$). Hence we can apply rule $l \rightarrow r$ to t' at position p'. ∎

The following theorem justifies deterministic rewrite steps at arbitrary lazy narrowing positions (see [37] for a detailed definition of lazy narrowing positions).

Theorem 4.5 *Let t, t' be terms such that $t \rightarrow_{\mathcal{R}} t'$ is a rewrite step at lazy narrowing position p. Then lazy narrowing is complete even if we ignore all alternative narrowing rules applicable to t.*

Proof: Let $t \rightsquigarrow_{[p',R,\sigma]} t''$ be an alternative lazy narrowing step. We show that we do not loose any solutions by ignoring this step and continuing with t' instead of t''.

$p' = p$: By Lemma 4.3 applied to position p, t'' is an instance of t'. Hence all solutions computed by narrowing derivations starting from t'' are also computed by narrowing derivations starting from t'.

p' is a position below p: Since p' is a lazy narrowing position, the narrowing step at p' is demanded by some rule which may be applicable at position p at some later point. However, similarly to the previous case, this alternative step at position p can be ignored without destroying completeness. Consequently, this narrowing step at position p' can also be ignored.

p' is a position not below p and $p' \neq p$: By Lemma 4.4, this alternative narrowing step is also applicable to t' with the same substitution of variables occurring in t. Hence we can ignore this step without destroying completeness. ■

As a consequence of this theorem, we can deterministically apply rewrite rules at any lazy narrowing position before a narrowing step. A simple induction shows that we can also deterministically apply a finite sequence of rewrite steps at lazy narrowing positions. I.e., we can combine *lazy narrowing with lazy simplification* (where lazy simplification positions are defined similarly to lazy narrowing positions [37]) without destroying completeness. However, this is only true for finite sequences of simplification steps (due to the proof by induction). Nevertheless, an infinite loop caused by simplification occurs in lazy narrowing derivations without simplification, too, since rewrite steps are also particular narrowing steps. The only difference is that the order of rule applications in simplification steps may be different from the order of rule applications in narrowing steps. Hence it may be the case that the simplification process runs into an infinite loop, whereas lazy narrowing without simplification first computes an answer and then runs into an infinite loop.

Example 4.6 Consider the rules of Example 1.2 and the following rule defining a nonterminating function:

$$\mathit{inf} \ \rightarrow \ \mathit{inf}$$

If the goal equation $x * \mathit{inf} \approx 0$ should be solved, a lazy simplification strategy tries to evaluate the subterm *inf* to the constructor 0 in order to apply rule R_4 to the left-hand side of the equation (i.e., the second argument of $*$ is a lazy narrowing position). Since the evaluation of *inf* loops, the simplification process does not terminate and no solution is computed. On the other hand, lazy narrowing without simplification narrows the left-hand side of the equation by applying rule R_3. This binds goal variable x to 0 and yields the trivial equation $0 \approx 0$. However, after the computation of this solution an attempt to apply the alternative rule R_4 to the left-hand side is made which yields the same infinite loop as in the simplification process. □

Note that this different behavior is due to a particular sequential implementation of the strategy. In an implementation which collects all answers until the entire search space has been examined, we obtain no answer in both cases due to the infinite search space.

In order to ensure the termination of the simplification process even if we blindly apply all possible lazy simplification steps, we include only a *terminating subset* of the program rules for simplification. Since lazy narrowing is already complete without simplification, it is not necessary to perform rewrite steps with all possible program rules, but we can arbitrarily restrict the set of rules used for rewrite steps. In the light of the previous example, it is a reasonable decision to include a rule set with a terminating rewrite relation for simplification. This ensures the termination of the simplification process. The selection of this subset of rewrite rules could be done by the programmer or by the system (e.g., include only those rewrite rules for which a termination proof can be constructed). We have made the experience that, for most practical examples, termination proofs can be automatically constructed using syntactic termination orderings from term rewriting [7]. This is the case for all rules presented so far (of course, except for the $first$-rule of Example 4.1 and the *inf*-rule of Example 4.6). An example where a terminating subset of all program rules is used for simplification will be given in Section 5.3.

5 Application to Functional Logic Programs

In this section we discuss the usefulness of integrating simplification into lazy narrowing derivations with respect to different classes of functional logic programs. In general, we consider constructor-based confluent rewrite systems. However, there are various subclasses of such rewrite systems with different implications on the usefulness of integrating simplification. We will discuss the following three subclasses in more detail: inductively sequential systems [1] where the rules for each function can be organized in a hierarchical structure, orthogonal systems satisfying the strong nonambiguity condition (no overlapping in the left-hand sides of the rules), and weakly orthogonal systems with overlapping left-hand sides.

5.1 Inductively Sequential Programs

In many functional as well as functional logic programs, functions are defined by a case distinction on the different constructors occurring in the data type of the arguments. For instance, the definition of the addition function on natural numbers (cf. Example 1.1) is based on a case distinction for the first argument with respect to the constructors 0 and s. As another example consider the following rules defining a less-or-equal function on naturals:

$$\begin{array}{rcllll} 0 & \leq & x & \to & true & (R_1) \\ s(x) & \leq & 0 & \to & false & (R_2) \\ s(x) & \leq & s(y) & \to & x \leq y & (R_3) \end{array}$$

Here is the main case distinction on the constructors of the first argument: if this argument is 0, then only rule R_1 is applicable. If this argument has the constructor s at the top, then a further case distinction on the second argument is necessary to distinguish between rules R_2 and R_3. Altogether, the rules can be organized in a hierarchical structure representing the various case distinctions. Such hierarchical structures have been introduced by Antoy [1] under the name *definitional trees*. A program for which the rules of each function symbol can be organized in a definitional tree is called *inductively sequential*. Antoy, Echahed and Hanus [2] have defined for inductively sequential programs a narrowing strategy, called *needed narrowing*, which is optimal in the following sense: (1) it reduces only needed subterms in a narrowing step, i.e., subterms which must be reduced in any possible successful narrowing derivation, (2) it computes the shortest narrowing derivations if common subterms are shared, and (3) the solutions computed by two different narrowing derivations are independent. The needed narrowing steps are computed using the structure of definitional trees. Thus it can be efficiently implemented by pattern matching, and the strategy has an outermost (lazy) behavior.

Due to the optimality of needed narrowing the natural question arises whether the inclusion of simplification has an effect for this class of programs. To answer this question, we recall the applicability conditions for a rewrite step. A functional expression can be reduced by a rewrite step if the arguments of the function call are sufficiently instantiated such that the left-hand side of some rule can be matched with the current call. Since the program is inductively sequential, there is always at most one rule matching the current call and this rule will be selected in the next narrowing step without instantiating any goal variables (see [2] for a detailed description of the strategy). Therefore, a possible lazy reduction step is also computed by the needed narrowing strategy as a narrowing step, i.e., the inclusion of simplification steps has no effect. This is formally justified by the following proposition.

Proposition 5.1 *Let $\mathcal{R}$ be a set of inductively sequential rules. Then the integration of simplification does not shorten any needed narrowing derivation.*

Proof: By definition, rewrite steps are also particular narrowing steps. Thus any narrowing derivation with intermediate simplification steps is also a pure narrowing derivation. Since needed narrowing computes the shortest narrowing derivations [2], simplification cannot shorten any needed narrowing derivation. ■

Hence it is unnecessary to integrate simplification with program rules in narrowing derivations for the class of inductively sequential programs. Therefore, narrowing derivations can be optimized for such programs only if inductive consequences are added as simplification rules. Actually, Example 3.16 is an inductively sequential program and we have shown that simplification with the additional inductive consequence $x * 0 \to 0$ can reduce the search space.

5.2 Orthogonal Programs

The main example where we have demonstrated the improvements of simplification with respect to lazy narrowing (Example 1.2) has the property that two rules have overlapping left-hand sides. In the following we will show that the inclusion of simplification is useful even if there are no overlapping rules.

Example 5.2 Consider the following rewrite rules:

$$\begin{array}{lcll} f(0,s(x),y) & \to & 0 & (R_1) \\ f(s(x),y,0) & \to & 0 & (R_2) \\ f(y,0,s(x)) & \to & 0 & (R_3) \end{array} \qquad \begin{array}{rcll} one(0) & \to & s(0) & (R_4) \\ one(s(x)) & \to & one(x) & (R_5) \end{array}$$

This is an orthogonal term rewriting system since all rules are left-linear and do not overlap in the left-hand sides. However, it is not inductively sequential since there is no argument which represents a case distinction on the constructors 0 and s. In fact, simplification has an effect if we consider the goal equation $f(one(z),0,s(0)) \approx 0$. Naive lazy narrowing first tries to apply rule R_1 to the left-hand side of this equation. Since the first argument of the rule's left-hand side is 0, the evaluation of the actual argument $one(z)$ is required in order to decide the unifiability of the first argument.[17] Similarly to Example 1.2, the evaluation of $one(z)$ has an infinite search space and a sequential implementation does not compute any result since all evaluations of $one(z)$ yields $s(0)$ as the result which is not unifiable with the demanded value 0. However, if we simplify the goal equation before the attempt to apply a narrowing step, we use rule R_3 for a rewrite step which yields the trivial equation $0 \approx 0$. Hence the infinite search space is avoided. □

[17] We assume that arguments are unified from left to right, otherwise a similar example can be constructed.

5.3 Weakly Orthogonal Programs

In Sections 5.1 and 5.2 we have shown that the boundary of the usefulness of simplification in lazy narrowing derivations is between inductively sequential and orthogonal systems. If we do not include inductive consequences for simplification, we conjecture that, for practical applications, the most interesting class, where simplification is useful, is the class of weakly orthogonal programs which have rules with overlapping left-hand sides. Example 1.2 contains such a simple program, but the recursively defined constant function *one* may not convince the reader. Therefore, we will demonstrate the positive effects of simplification by a more natural example.

Example 5.3 Consider the following rules defining the Boolean operator $\vee$ and the predicate *even* on natural numbers:

$$\begin{array}{rcll} true \vee b & \to & true & (R_1) \\ b \vee true & \to & true & (R_2) \\ false \vee false & \to & false & (R_3) \end{array} \qquad \begin{array}{rcll} even(0) & \to & true & (R_4) \\ even(s(0)) & \to & false & (R_5) \\ even(s(s(x))) & \to & even(x) & (R_6) \end{array}$$

This rewrite system is weakly orthogonal since rules R_1 and R_2 overlap. Now consider the goal equation $even(z) \vee true \approx true$ (note that this goal equation could also be the result of the more general equation $even(z) \vee b \approx true$ where the Boolean variable b has been bound to $true$ in the preceding computation). Naive lazy narrowing without simplification tries to apply a narrowing step with rule R_1. Since the value of the first $\vee$-argument is demanded by this rule, the subterm $even(z)$ is evaluated to a constructor-headed term by narrowing. There are infinitely many possibilities to do this, in particular, the constructor $true$ is derived by instantiating variable z with the values $s^{2*i}(0)$, $i \geq 0$. Therefore, lazy narrowing without simplification has an infinite search space and computes the additional specialized solutions $\{z \mapsto s^{2*i}(0)\}$. Moreover, in a sequential implementation of lazy narrowing by backtracking [18], only the infinite set of specialized solutions would be computed without ever trying the second $\vee$-rule. On the other hand, if the equation is first simplified by applying rule R_2 to the left-hand side, we immediately obtain the trivial equation $true \approx true$ and avoid the infinite search space. □

We have mentioned that our method is complete even in the presence of nonterminating functions if a terminating subset of the program rules is used for simplification. This is demonstrated by a modification of the previous example.

Example 5.4 Consider the rules for $\vee$ of Example 5.3 (R_1, R_2, R_3) and the following new rules for *not*, *even* and *odd*:

$$\begin{array}{rcll} not(true) & \to & false & (R_4) \\ not(false) & \to & true & (R_5) \end{array} \qquad \begin{array}{rcll} even(x) & \to & not(odd(x)) & (R_6) \\ odd(x) & \to & not(even(x)) & (R_7) \end{array}$$

Although *even* and *odd* are nonterminating functions, it is an admissible program. We use the terminating subset of the rules $\{R_1, R_2, R_3, R_4, R_5\}$ for simplification.[18] Consider the goal equation $even(z) \vee not(false) \approx true$. Lazy narrowing without simplification tries to compute the head normal form of the subterm $even(z)$ since its value is demanded by rule R_1. Since this computation is nonterminating, naive lazy narrowing has an infinite search space. The same holds for lazy

[18] Note that the termination property of this subset can be automatically checked.

narrowing with the dynamic cut operator [32]. However, lazy narrowing with simplification tries to apply rewrite steps first. No simplification rule is applicable to the entire left-hand side of the goal equation since the arguments are not in head normal form. Due to the lazy simplification strategy, we try to evaluate the arguments by simplification steps. The subterm $even(z)$ cannot be further simplified since rule R_6 is not included in the set of simplification rules. The second argument $not(false)$ can be simplified to $true$ by R_5 which causes the simplification of the complete left-hand side to $true$ by R_2. Hence we obtain the trivial equation $true \approx true$ and the infinite search space is avoided. □

5.4 A Benchmark

In order to test our new execution strategy on larger programs, we have implemented an interpreter for lazy narrowing with simplification in Prolog [23]. An interesting class of programs, where simplification has a relevant effect on the search space, are "generate-and-test" programs. A typical example for such programs is the "permutation sort" program, where a list is sorted by enumerating all permutations and checking whether they are sorted. In the Prolog version of this program ([44], p. 55), *all* permutations are enumerated and checked. However, if we execute the same program by lazy narrowing with simplification (in this case predicates are considered as Boolean functions, see [14], p. 182), then the simplification process cuts some parts of the search space so that not all permutations are completely enumerated. Therefore, we obtain the following execution times in seconds (Sicstus-Prolog 2.1 on a Sparc10) to sort the list `[`n`,...,2,1]` for different values of n:

Length n	Prolog	Lazy	Lazy+Simp
4	0.01	0.02	0.04
5	0.01	0.1	0.1
6	0.05	0.8	0.2
7	0.3	5.4	0.5
8	2.6	45.9	1.1
9	23.6	420.1	2.5
10	240.9	4389.2	5.5

The column "Lazy+Simp" contains the execution times for lazy narrowing with simplification, the column "Lazy" the times for pure lazy narrowing without simplification, and the column "Prolog" the times for the direct implementation of permutation sort in Prolog. The search spaces of "Prolog" and "Lazy" are essentially the same. The slow timings of "Lazy" is due to the overhead of the lazy narrowing interpreter (which is also written in Prolog). However, the last column shows that this overhead can be compensated by the search space reduction due to the simplification process.

6 Conclusions and Related Work

In this paper we have shown how to improve the execution mechanism of functional logic languages, where we have considered the most important classes of programs: ground confluent and terminating rewrite systems, and weakly orthogonal and possibly nonterminating rewrite systems. The basic idea of our improvement is the integration of a deterministic simplification process into lazy narrowing derivations. This can be done in a simple way by using the program rules (or a terminating subset of the program rules in the presence of nonterminating rules) as simplification rules. The simplification strategy must be identical to the narrowing strategy in order to avoid additional computation steps caused by the simplification process. For particular and practically relevant classes of functional logic programs (orthogonal and weakly orthogonal programs) this has the positive effect that the search space is reduced without destroying completeness. Although we have emphasized the effect of simplification to the search space, the inclusion of simplification can also have an effect on the run time even if the search space is not reduced. For instance, if all program rules are used for simplification, ground goals are evaluated by simplification without generating any choice point, whereas a lazy narrowing implementation would generate (and afterwards delete) choice points. Hence lazy narrowing with simplification combines the features from functional and logic programming also from an implementation point of view.

We have mentioned in the introduction and in Section 2 that the idea of exploiting deterministic computations by including simplification in functional logic languages has been proposed mainly for eager narrowing strategies like basic [38, 42], innermost [14] or innermost basic narrowing [25]. Echahed [12] has shown how to integrate normalization (with inductive consequences) in any narrowing strategy, but he requires strong restrictions on the set of rules (termination and uniformity, which is stronger than inductive sequentiality). As far as we know, the present paper is the first attempt to include simplification into narrowing derivations even in the presence of nonterminating functions.[19] The only related work for this class of programs is the paper of Loogen and Winkler [32] which proposes the dynamic cut to detect deterministic narrowing steps after the unification phase. As discussed in Section 4, this does not avoid the generation of choice points, and the cut of infinite derivation paths depends on the order of rules. The basic difference of our method is that we check the applicability of a deterministic computation step before we apply a nondeterministic step. Hence we prefer deterministic computations to nondeterministic computations. This qualifies our execution method as the operational principle of efficient functional logic languages.

Loogen et al. [31] have proposed to improve lazy narrowing strategies by reordering the unification steps in rule applications. For this purpose they use a version of definitional trees [1] extended to weakly orthogonal rewrite systems. In order to handle overlapping left-hand sides, they introduce nondeterministic choice nodes in definitional trees. However, these choice nodes have the effect that possible deterministic computations are not detected. For instance, the infinite search spaces of naive lazy narrowing in Examples 1.2, 5.2 and 5.3 would also occur with respect to their improved strategy.

Another alternative to improve lazy narrowing has been proposed by Moreno-Navarro et al. [36]. They use information about demanded arguments to avoid reevaluations of expressions during unification with different rules. Since they do not change the order of argument evaluations and rules, the infinite search spaces avoided by simplification still occur in their approach.

The integration of simplification into lazy narrowing derivations requires new implementation techniques for functional logic languages. Current efficient implementations of lazy narrowing are mainly based on extensions of reduction machines used for the implementation of functional lan-

[19] The combination of lazy narrowing with deterministic reduction steps has been also considered by Josephson and Dershowitz [27]. However, they provide no completeness proof but refer to [10] where only the completeness of naive narrowing without simplification and without a particular lazy strategy is proved for terminating conditional rules.

guages [5, 18, 30, 35]. The inclusion of simplification requires the implementation of an intermediate reduction process. This could be done by techniques proposed for the efficient implementation of normalizing narrowing [19, 20] or by the implementation of demons waiting for the sufficient instantiation of function arguments [27].

Acknowledgements. The author is grateful to the anonymous referees for their suggestions to improve the paper. The research described in this paper was mainly made during the author's stay at the Max-Planck-Institut für Informatik in Saarbrücken, Germany. It 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.

References

[1] S. Antoy. Definitional Trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 143–157. Springer LNCS 632, 1992.

[2] 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.

[3] F. Baader and J.H. Siekmann. Unification Theory. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pp. 41–125. Oxford University Press, 1994.

[4] 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.

[5] M.M.T. Chakravarty and H.C.R. Lock. The Implementation of Lazy Narrowing. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 123–134. Springer LNCS 528, 1991.

[6] J. Darlington and Y. Guo. Narrowing and unification in functional programming - an evaluation mechanism for absolute set abstraction. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 92–108. Springer LNCS 355, 1989.

[7] N. Dershowitz. Termination of Rewriting. *J. Symbolic Computation*, Vol. 3, pp. 69–116, 1987.

[8] 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.

[9] N. Dershowitz, S. Mitra, and G. Sivakumar. Equation Solving in Conditional AC-Theories. In *Proc. of the 2nd International Conference on Algebraic and Logic Programming*, pp. 283–297. Springer LNCS 463, 1990.

[10] N. Dershowitz and D.A. Plaisted. Equational Programming. In J.E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11*, pp. 21–56. Oxford Press, 1988.

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

[12] R. Echahed. Uniform Narrowing Strategies. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 259–275. Springer LNCS 632, 1992.

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

[14] 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.

[15] J.H. Gallier and W. Snyder. Complete Sets of Transformations for General E-Unification. *Theoretical Computer Science*, Vol. 67, pp. 203–260, 1989.

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

[17] 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.

[18] W. Hans, R. Loogen, and S. Winkler. On the Interaction of Lazy Evaluation and Backtracking. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pp. 355–369. Springer LNCS 631, 1992.

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

[20] M. Hanus. Efficient Implementation of Narrowing and Rewriting. In *Proc. Int. Workshop on Processing Declarative Knowledge*, pp. 344–365. Springer LNAI 567, 1991.

[21] 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.

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

[23] M. Hanus. Efficient Translation of Lazy Functional Logic Programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pp. 252–266. Springer LNCS 1048, 1995.

[24] M. Hanus. On Extra Variables in (Equational) Logic Programming. In *Proc. International Conference on Logic Programming*, pp. 665–679. MIT Press, 1995.

[25] S. Hölldobler. *Foundations of Equational Logic Programming*. Springer LNCS 353, 1989.

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

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

[28] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, Vol. 15, No. 4, pp. 1155–1194, 1986.

[29] S. Krischer and A. Bockmayr. Detecting Redundant Narrowing Derivations by the LSE-SL Reducibility Test. In *Proc. RTA'91*. Springer LNCS 488, 1991.

[30] R. Loogen. Relating the Implementation Techniques of Functional and Functional Logic Languages. *New Generation Computing*, Vol. 11, pp. 179–215, 1993.

[31] R. Loogen, F. Lopez Fraguas, and M. Rodríguez Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 184–200. Springer LNCS 714, 1993.

[32] R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional Logic Languages. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pp. 335–346. Springer LNCS 528, 1991. Extended version to appear in *Theoretical Computer Science*, 1995.

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

[34] A. Martelli, G.F. Rossi, and C. Moiso. Lazy Unification Algorithms for Canonical Rewrite Systems. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2, Rewriting Techniques*, chapter 8, pp. 245–274. Academic Press, New York, 1989.

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

[36] J.J. Moreno-Navarro, H. Kuchen, J. Marino-Carballo, S. Winkler, and W. Hans. Efficient Lazy Narrowing Using Demandedness Analysis. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pp. 167–183. Springer LNCS 714, 1993.

[37] 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.

[38] W. Nutt, P. Réty, and G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, Vol. 7, pp. 295–317, 1989.

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

[40] C. Prehofer. Higher-Order Narrowing. In *Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*, pp. 507–516, 1994.

[41] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.

[42] P. Réty. Improving basic narrowing techniques. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 228–241. Springer LNCS 256, 1987.

[43] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, Vol. 21, No. 4, pp. 622–642, 1974.

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