In *Proc. 5th European Symposium on Programming*
Edinburgh, April 1994, Springer LNCS 788, pp. 272–286

Lazy Unification with Simplification

Michael Hanus

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

Abstract. Unification in the presence of an equational theory is an important problem in theorem-proving and in the integration of functional and logic programming languages. This paper presents an improvement of the proposed lazy unification methods by incorporating simplification into the unification process. Since simplification is a deterministic computation process, more efficient unification algorithms can be achieved. Moreover, simplification reduces the search space so that in some case infinite search spaces are reduced to finite ones. We show soundness and completeness of our method for equational theories represented by ground confluent and terminating rewrite systems which is a reasonable class w.r.t. functional logic programming.

1 Introduction

Unification is not only an important operation in theorem provers but also the most important operation in logic programming systems. Unification in the presence of an equational theory, also known as *E-unification*, is necessary if the computational domain in a theorem prover enjoys certain equational properties [26] or if functions should be integrated into a logic language [10]. Therefore the development of E-unification algorithms is an active research topic during recent years (see, for instance, [29]).

Since E-unification is a complex problem even for simple equational axioms, we are interested in efficient E-unification methods in order to incorporate such methods into functional logic programming languages. One general method to improve the efficiency of implementations is the use of a *lazy strategy*. "Lazy" means that evaluations are performed only if it is necessary to compute the required solutions. In the context of unification this corresponds to the idea that terms are manipulated at outermost positions. Hence *lazy unification* means that equational axioms are applied to outermost positions of equations. For instance, consider the following equations for addition and multiplication on natural numbers which are represented by terms of the form $s(\cdots s(0)\cdots)$:

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

If we have to unify the terms $0*(s(0)+s(z))$ and 0, we could apply equational axioms to inner subterms starting with $s(0)+s(z)$ (*innermost* or *eager strategy*) or to outermost subterms (*outermost* or *lazy strategy*). This will lead to the following two derivations (the subterms manipulated in the next step are underlined):

$$0*(\underline{s(0)+s(z)})\approx 0 \;\Rightarrow\; 0*(s(\underline{0+s(z)}))\approx 0 \;\Rightarrow\; \underline{0*(s(s(z)))}\approx 0 \;\Rightarrow\; 0\approx 0$$
$$\underline{0*(s(0)+s(z))}\approx 0 \;\Rightarrow\; 0\approx 0$$

Obviously, the second lazy unification derivation should be preferred.

There are many proposals for such lazy unification strategies. For instance, Martelli et al. [22] have proposed a lazy unification algorithm for confluent and terminating equational axioms. Due to the confluence requirement, equations are only applied in one direction. However, their method is not pure lazy since equations are applied to inner subterms in equations of the form $x \approx t$ where the variable x occurs in t. Gallier and Snyder [11] have proved the completeness of a lazy unification method for arbitrary equational theories where equations can be applied in both directions. *Narrowing* is a method to compute E-unifiers in the presence of confluent axioms. It is a combination of the reduction principle of functional languages with syntactic unification in order to instantiate variables. Lazy narrowing were proposed by Reddy [27] as the operational principle of functional logic languages. Recently, Antoy, Echahed and Hanus [1] have proposed a narrowing strategy for programs where the functions are defined by case distinctions over the data structures. This strategy reduces only needed redexes, computes no redundant solutions, and is optimal w.r.t. the length of narrowing derivations.

From a practical point of view the disadvantage of E-unification is its inherent nondeterminism. In the area of narrowing there are many proposals for the inclusion of a deterministic simplification process in order to reduce the nondeterminism [8, 9, 19, 24, 28], but all these proposals are based on an eager narrowing strategy. On the other hand, only little work has been done to improve the efficiency of outermost or lazy strategies. Echahed [7] has shown the completeness of any narrowing strategy with simplification under strong requirements (uniformity of specifications). Dershowitz et al. [6] have proposed to combine lazy unification 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. 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. Therefore we will formulate a calculus for lazy unification which includes simplification and give a rigorous completeness proof. The distinguishing features of our framework are:

- We consider a ground confluent and terminating equational specification in order to apply equations only in one direction and to ensure the existence of normal forms. This is reasonable if one is interested in declarative programming rather than theorem proving.
- The unification calculus is lazy, i.e., functions are not evaluated if their value is not required to decide the unifiability of terms. Consequently, we may compute *reducible solutions* as answers according to the spirit of lazy evaluation. For instance, in contrast to other "lazy" unification methods we do not allow any evaluation of t in the equation $x \approx t$ if x occurs only once.
- We include a deterministic simplification process in our unification calculus. In order to restrict nondeterministic computations as much as possible, we allow to use additional inductive consequences for simplification which has been proved to be useful in other calculi [7, 9, 24].

After recalling basic notions from term rewriting, we present in Section 3 our basic lazy unification calculus. In Section 4 we include a deterministic simplification process into the lazy unification calculus. Finally, we show in Section 5 some important optimizations for constructor-based specifications. Due to lack of space we omit the details of some proofs, but the interested reader will find them in [17].

2 Computing in equational theories

In this section we recall the notations for equations and term rewriting systems [5] which are necessary in our context.

Let the *signature* $\mathcal{F}$ be a set of *function symbols*[1] and $\mathcal{X}$ be a countably infinite set of *variables*. Then $\mathcal{T}(\mathcal{F},\mathcal{X})$ denotes the set of *terms* built from $\mathcal{F}$ and $\mathcal{X}$. $\mathcal{V}ar(t)$ is the set of variables occurring in t. A *ground term* t is a term without variables, i.e., $\mathcal{V}ar(t) = \emptyset$. 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 $x \in \mathcal{X}$. 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 $\sigma' = \phi \circ \sigma$. 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 [5] for details). The outermost position Λ is also called *root* position.

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 $e_2 \rightarrow^* e_3$.

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 equational specifications, we will use the specified equations 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. A *term rewriting system* $\mathcal{R}$ is a set of rewrite rules. In the following we assume a given *term rewriting system* $\mathcal{R}$.

A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \rightarrow_{\mathcal{R}} s$ if there exists a position p, a rewrite rule $l \rightarrow r$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. A term t is called *reducible* if we can apply a rewrite rule to it, and t is called *irreducible* or in *normal form* if there is no term s with $t \rightarrow_{\mathcal{R}} s$. 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. Hence we call an equation $s \approx t$ *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 non-ground 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

[1] In this paper we consider only single-sorted programs. The extension to many-sorted signatures is straightforward [25]. Since sorts are not relevant to the subject of this paper, we omit them for the sake of simplicity.

t is *narrowable* into a term t' if there exist a non-variable position p (i.e., $t|_p \notin \mathcal{X}$), a variant $l \to r$ of a rewrite rule and 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 $t \rightsquigarrow_\sigma t'$.

Narrowing is able to solve equations w.r.t. $\mathcal{R}$ by deriving both sides of an equation to syntactically unifiable terms. Due to the huge search space of simple narrowing, several authors have proposed restrictions on the admissible narrowing derivations (see [18] for a detailed survey). *Lazy narrowing* [3, 23, 27] 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. Since lazy strategies are important in the context of non-terminating rewrite rules, these strategies have been proved to be complete w.r.t. domain-based interpretations of rewrite rules [13, 23]. *Lazy unification* is very similar to lazy narrowing but manipulates sets of equations rather than terms. It has been proved to be complete for canonical term rewriting systems w.r.t. standard semantics [6, 22].

From a practical point of view the most essential improvement of simple narrowing is *normalizing narrowing* [8] 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 gives the same result (if $\mathcal{R}$ is confluent and terminating). As shown in [9, 16], normalizing narrowing has the important effect that equational logic programs are more efficiently executable than pure logic programs. Normalization can also be combined with other narrowing restrictions [9, 19, 28]. Because of these important advantages, normalizing narrowing is the foundation of several programming languages which combines functional and logic programming like ALF [15], LPG [2] or SLOG [9]. However, normalization has not been included in lazy narrowing strategies.[2] Therefore we will present a lazy unification calculus which includes a normalization process where the term rewrite rules as well as additional inductive consequences are used for normalization.

3 A calculus for lazy unification

In the rest of this paper we assume that $\mathcal{R}$ *is a ground confluent and terminating term rewriting system.* This section presents our basic lazy unification calculus to solve a system of equations. The inclusion of a normalization process will be shown in Section 4. 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.

Our lazy unification calculus manipulates sets of equations in the style of Martelli and Montanari [21] 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 $\sigma(s) =_\mathcal{R} \sigma(t)$ for all equations $s \approx t \in E$.[3] 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

[2] Except for [6, 7], but see the remarks in Section 1.

[3] 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 [24], this ground approach subsumes the conventional narrowing approaches where also non-ground solutions are

Lazy narrowing

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

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 rule

Decomposition of equations

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

Partial binding of variables

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

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

Figure 1. The lazy unification calculus

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.[4] 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 $f(t_1,\ldots,t_n)$ is costly. In order to avoid this problem of *eager variable elimination* (see [11]), we perform only a partial binding which is also called "root imitation" in [11].

At first sight our lazy unification calculus has many similarities with the lazy unification rules presented in [6, 11, 22, 25]. 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 rules it is allowed to evaluate a term only if its value is needed (in several positions). Otherwise, the term is left unevaluated.

Example 1. Consider the rewrite rule $0 * x \to 0$. Then the only transformation sequence of the equation $0 * t \approx 0$ (where t is a costly function) is

$$0 * t \approx 0 \quad \stackrel{lu}{\Longrightarrow} \quad 0 \approx 0,\ t \approx x,\ 0 \approx 0 \quad \stackrel{lu}{\Longrightarrow} \quad t \approx x,\ 0 \approx 0 \quad \stackrel{lu}{\Longrightarrow} \quad t \approx x$$

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

taken into account.

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

Coalesce	$x \approx y, E \quad \overset{var}{\Longrightarrow} \quad x \approx y, \phi(E)$		if $x, y \in \mathcal{V}ar(E)$ and $\phi = \{x \mapsto y\}$
Trivial	$x \approx x, E \quad \overset{var}{\Longrightarrow} \quad E$		

Figure 2. The variable elimination rules

The conventional transformation rules for unification w.r.t. an empty equational theory [21] 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 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:

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

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. □

In the rest of this section we will show 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 called in "solved form". According to [11, 21] we call an equation $x \approx t \in E$ *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 obtain 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 always possible because $\stackrel{var}{\Longrightarrow}$ is terminating, preserves solutions, and transforms each quasi-solved system into a solved one (see [17] for details). Moreover, the solutions of an equation system in solved form can be obtained as follows:

Proposition 1. *Let $E = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ be an equation system in solved form. Then the substitution set*

$$\{\gamma \circ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} \mid \gamma \text{ is a ground substitution}\}$$

is a complete set of solutions for E.

Therefore it is sufficient to transform an equation system into a quasi-solved form. The soundness of the lazy unification calculus is implied by the following theorem which can be proved by a case analysis on the applied transformation rule [17].

Theorem 2. *Let E and E' be equation systems with $E \stackrel{lu}{\Longrightarrow} E'$. Then each solution σ of E' is also a solution of E.*

For the completeness 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 since new additional variables are generated during the derivation (by lazy narrowing and partial binding transformations). But this is not a problem since we are interested in solutions w.r.t. the variables of the initial equation system.

Theorem 3. *Let E be a solvable equation system with solution σ. Then there exists a derivation $E \stackrel{lu}{\Longrightarrow}{}^* E'$ 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 $E \stackrel{lu}{\Longrightarrow} E'$ 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 term rewriting 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 [20].[5] Now we define the following ordering on pairs (σ, E): $(\sigma, E) \succ (\sigma', E')$ iff

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

[5] Note that the use of the relation $\rightarrow_{\mathcal{R}}$ instead of $\ggcurly$ (as done in [6]) is not sufficient for the completeness proof since $\rightarrow_{\mathcal{R}}$ has not the subterm property [4] in general.

where $\ggcurly_{mul}$ is the multiset extension[6] 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 [4].

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 reducible 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 $\ggcurly$ 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 extend σ to a new substitution σ' with $\sigma'(x) = \theta(x)$ for all $x \in \mathcal{D}om(\theta)$ (this is always possible since θ does only work on the variables of the new variant of the rewrite rule). 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 $\ggcurly$, $\sigma(s_i) \ggcurly \sigma'(l_i)$ (if $\sigma(s_i) \neq \sigma'(l_i)$) and $\sigma(s) \ggcurly \sigma'(r)$. 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 the smaller terms $\sigma(s_1), \ldots, \sigma(s_n), \sigma'(l_1), \ldots, \sigma'(l_n), \sigma'(r)$ ($\sigma(s) \ggcurly \sigma(s_i)$ since $\ggcurly$ 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.

[6] 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. $\ggcurly$ [4].

(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 $i = 1, \ldots, n$. 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 in the above proof is tailored to a simple proof for the completeness of lazy unification with simplification as we will see in the next section.

The results of this section 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 5.

4 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. Consider the rewrite rules

$$f(a) \rightarrow c \qquad\qquad g(a) \rightarrow a$$
$$f(b) \rightarrow d \qquad\qquad g(b) \rightarrow b$$

and the equation $f(g(b)) \approx d$. Then there are four different derivations in our lazy unification calculus, but only one derivation is successful. If we would first compute the normal form of $f(g(b))$, which is d, then there is only one possible derivation: $d \approx d \overset{lu}{\Longrightarrow} \emptyset$. 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 [9, 16] 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. For instance, the rule $x + 0 \to x$ is an inductive consequence of the term rewriting system

$$0 + y \;\to\; y \qquad\qquad s(x) + y \;\to\; s(x + y)$$

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.[7]

In the following we assume that $\mathcal{S}$ is a set of inductive consequences of $\mathcal{R}$ (the set of *simplification rules*) so that the rewrite relation $\to_{\mathcal{S}}$ is terminating. We will use rules from $\mathcal{R}$ for lazy narrowing and rules from $\mathcal{S}$ for simplification. Note that each rule from $\mathcal{R}$ is also an inductive consequence and can be included in $\mathcal{S}$. But 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 which 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 $\Rightarrow_{\mathcal{S}}$ on systems of equations. $s \approx t \Rightarrow_{\mathcal{S}} s' \approx t'$ iff s' and t' are normal forms of s and t w.r.t. $\to_{\mathcal{S}}$, respectively. $E \Rightarrow_{\mathcal{S}} E'$ iff $E = e_1, \ldots, e_n$ and $E' = e_1', \ldots, e_n'$ where $e_i \Rightarrow_{\mathcal{S}} e_i'$ for $i = 1, \ldots, n$. Note that $\Rightarrow_{\mathcal{S}}$ describes a deterministic computation process.[8] $E \overset{lus}{\Longrightarrow} E'$ is a derivation step in the *lazy unification calculus with simplification* if $E \Rightarrow_{\mathcal{S}} \overline{E} \overset{lu}{\Longrightarrow} E'$ for some $\overline{E}$.

The soundness of the calculus $\overset{lus}{\Longrightarrow}$ can be shown by a simple induction on the computation steps using Theorem 2 and the following lemma which shows the soundness of one rewrite step with a simplification rule:

Lemma 4. *Let $s \approx t$ be an equation and $s \to_{\mathcal{S}} s'$ be a rewrite step. Then each solution of $s' \approx t$ is also a solution of $s \approx t$.*

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

Lemma 5. *Let E be an equation system and σ be a solution of E. If $E \Rightarrow_{\mathcal{S}} E'$, then σ is a solution of E'.*

---

[7] In larger single-sorted term rewriting systems it would 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.

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

This lemma would imply the completeness of the calculus $\stackrel{lus}{\Longrightarrow}$ if a derivation step with $\Rightarrow_{\mathcal{S}}$ does not increase the ordering used in the proof of Theorem 3. Unfortunately, this is not the case in general since the termination of $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{S}}$ may be based on different orderings (e.g., $\mathcal{R} = \{a \rightarrow b\}$ and $\mathcal{S} = \{b \rightarrow a\}$). In order to avoid such problems, we require that the relation $\rightarrow_{\mathcal{R}\cup\mathcal{S}}$ is terminating which is not a real restriction in practice.

Theorem 6. *Let $\mathcal{S}$ be a set of inductive consequences of the ground confluent and terminating term rewriting system $\mathcal{R}$ such that $\rightarrow_{\mathcal{R}\cup\mathcal{S}}$ 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 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 5 shows that normalization steps preserve solutions. The only difference concerns the ordering where we use $\rightarrow_{\mathcal{R}\cup\mathcal{S}}$ instead of $\rightarrow_{\mathcal{R}}$, i.e., $\ggcurly$ is now defined to be the transitive closure of the relation $\rightarrow_{\mathcal{R}\cup\mathcal{S}} \cup \succ_{sst}$. Clearly, this does not change anything in the proof of Theorem 3. Moreover, the relation $\Rightarrow_{\mathcal{S}}$ does not increase the complexity w.r.t. this ordering but reduces it if inductive consequences are applied since $\rightarrow_{\mathcal{S}}$ is contained in $\ggcurly$. □

These results show that we can integrate the deterministic simplification process into the lazy unification calculus without loosing soundness and completeness. Note that the rules from $\mathcal{S}$ 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 4. Consider the rules

$$zero(s(x)) \rightarrow zero(x) \qquad\qquad zero(0) \rightarrow 0$$

(assume that these rules are contained in $\mathcal{R}$ as well as in $\mathcal{S}$) 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 [11, 22]):

$$\begin{array}{ll} zero(x) \approx 0,\ x \approx 0 & \\ \stackrel{lus}{\Longrightarrow} x \approx s(x_1),\ zero(x_1) \approx 0,\ x \approx 0 & (\textit{lazy narrowing}) \\ \stackrel{lus}{\Longrightarrow} x \approx s(x_1),\ x_1 \approx s(x_2),\ zero(x_2) \approx 0,\ x \approx 0 & (\textit{lazy narrowing}) \\ \stackrel{lus}{\Longrightarrow} \cdots & \end{array}$$

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

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

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

Decomposition of constructor equations

$c(t_1,\ldots,t_n) \approx c(t'_1,\ldots,t'_n), E \stackrel{luc}{\Longrightarrow} t_1 \approx t'_1,\ldots,t_n \approx t'_n, E$ if $c \in \mathcal{C}$

Full binding of variables to ground constructor terms

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

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

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

Figure 3. Deterministic transformations for constructor-based rewrite systems

5 Constructor-based systems

In practical applications of equational logic programming a distinction is made between operation symbols to construct data terms, called *constructors*, and operation symbols to operate on data terms, called *defined functions* (see, for instance, the functional logic languages ALF [15], BABEL [23], K-LEAF [13], SLOG [9], or the RAP system [12]). 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.

The important 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 $\stackrel{lu}{\Longrightarrow}$ is used. Hence they can be integrated into the deterministic normalization process $\Rightarrow_{\mathcal{S}}$. 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 6 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 $cvar$ denotes the set of all variables occurring outside terms headed by defined function symbols. This restriction avoids infinite derivations of the following kind:

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

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

Clash	$c(t_1, \ldots, t_n) \approx d(t'_1, \ldots, t'_m), E \stackrel{luc}{\Longrightarrow}$ FAIL	if $c, d \in \mathcal{C}$ and $c \neq d$
Occur check	$x \approx c(t_1, \ldots, t_n), E \stackrel{luc}{\Longrightarrow}$ FAIL	if $x \in cvar(c(t_1, \ldots, t_n))$

Figure 4. Failure rules for constructor-based rewrite systems

with $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{R}} t$. In this case all ground terms have a ground constructor normal form and therefore 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.

6 Examples

In this section we demonstrate the improved computational power of our lazy unification calculus with simplification by means of two examples. The first example shows that simplification reduces the search space in the presence of rewrite rules with overlapping left-hand sides.

Example 5. Consider the following ground confluent and terminating rewrite system defining the Boolean operator $\vee$ and the predicate *even* on natural numbers:

$$\begin{array}{rcl@{\qquad\qquad}rcl} true \vee b & \rightarrow & true & even(0) & \rightarrow & true \\ b \vee true & \rightarrow & true & even(s(0)) & \rightarrow & false \\ false \vee false & \rightarrow & false & even(s(s(x))) & \rightarrow & even(x) \end{array}$$

If we want to solve the equation $even(z) \vee true \approx true$, the lazy unification calculus without simplification could apply a lazy narrowing step with the first $\vee$-rule. This yields the equation system

$$even(z) \approx true, \; true \approx b, \; true \approx true$$

Now there are infinitely many solutions to the new equation $even(z) \approx true$ by instantiating the variable z with the values $s^{2*i}(0)$, $i \geq 0$, i.e., the lazy unification calculus without simplification (cf. Section 3) has an infinite search space. The same is true for other lazy unification calculi [11, 22] or lazy narrowing calculi [23, 27]. Moreover, in a sequential implementation of lazy narrowing by backtracking [14] only an infinite set of specialized solutions would be computed without ever trying the second $\vee$-rule. But if we use our lazy unification calculus with simplification where all rewrite rules are used for simplification (i.e., $\mathcal{R} = \mathcal{S}$), then the initial equation $even(z) \vee true \approx true$ is first simplified to $true \approx true$ by rewriting with the second $\vee$-rule. Hence our calculus has a finite search space. $\square$

If the left-hand sides of the rewrite rules do not overlap, i.e., if the functions are defined by a case distinction on one argument, then there exists a lazy narrowing strategy (*needed narrowing* [1]) which is optimal w.r.t. the length of derivations. However, unsuccessful infinite derivations can be avoided also in this case by our lazy unification calculus with simplification if inductive consequences are added to the set of simplification rules.

Example 6. Consider the following rewrite rules for the addition and multiplication on natural numbers where $\mathcal{C} = \{0, s\}$ are constructors and $\mathcal{D} = \{+, *\}$ are defined functions:

$$
\begin{array}{rclr}
0+y & \rightarrow & y & (1) \\
s(x)+y & \rightarrow & s(x+y) & (2)
\end{array}
\qquad
\begin{array}{rclr}
0*y & \rightarrow & 0 & (3) \\
s(x)*y & \rightarrow & y+x*y & (4)
\end{array}
$$

If we use this confluent and terminating set of rewrite rules for lazy narrowing ($\mathcal{R}$) as well as for normalization ($\mathcal{S}$) and add the inductive consequence $x * 0 \rightarrow 0$ to $\mathcal{S}$, 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:

$$
\begin{array}{ll}
x * y \approx s(0) & \\
\stackrel{lu}{\Longrightarrow} x \approx s(x_1),\ y \approx y_1,\ y_1 + x_1 * y_1 \approx s(0) & (\textit{lazy narrowing, rule 4}) \\
\stackrel{lu}{\Longrightarrow} x \approx s(x_1),\ y \approx y_1,\ y_1 \approx 0,\ x_1 * y_1 \approx y_2,\ y_2 \approx s(0) & (\textit{lazy narrowing, rule 1}) \\
\stackrel{luç}{\Longrightarrow} x \approx s(x_1),\ y \approx 0,\ y_1 \approx 0,\ x_1 * 0 \approx y_2,\ y_2 \approx s(0) & (\textit{bind variable } y_1) \\
\stackrel{luç}{\Longrightarrow} x \approx s(x_1),\ y \approx 0,\ y_1 \approx 0,\ x_1 * 0 \approx s(0),\ y_2 \approx s(0) & (\textit{bind variable } y_2) \\
\Rightarrow_{\mathcal{S}} x \approx s(x_1),\ y \approx 0,\ y_1 \approx 0,\ 0 \approx s(0),\ y_2 \approx s(0) & (\textit{reduce } x_1 * 0) \\
\stackrel{luç}{\Longrightarrow} \text{FAIL} & (\textit{clash between } 0 \textit{ and } s)
\end{array}
$$

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.

Note that other lazy unification calculi [11, 22] or lazy narrowing calculi [23, 27] have an infinite search space for this equation. It is also interesting to note that a normalizing innermost narrowing strategy as in [9] has also an infinite search space even if the same inductive consequences are available. This shows the advantage of combining a lazy strategy with a simplification process. □

7 Conclusions

In this paper we have presented a calculus for unification in the presence of an equational theory. In order to obtain a small search space, the calculus is designed in the spirit of lazy evaluation, i.e., functions are not evaluated if their result is not required to solve the unification problem. The most important property of our calculus is the inclusion of a deterministic simplification process. This has the positive effect that our calculus is more efficient (in terms of the search space size) than other lazy unification calculi or eager narrowing calculi (like basic narrowing, innermost narrowing) with simplification. We think that our calculus is the basis of efficient implementations of future functional logic languages.

Acknowledgements. The author is grateful to Harald Ganzinger for his pointer to a suitable termination ordering and to two anonymous referees for their helpful remarks. The research described in this paper was supported in part by the German Ministry for Research and Technology (BMFT) under grant ITS 9103 and by the ESPRIT Basic Research Working Group 6028 (Construction of Computational Logics).

References

1. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, pp. 268–279, Portland, 1994.
2. D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. ESOP'86*, pp. 119–132. Springer LNCS 213, 1986.
3. 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.

4. N. Dershowitz. Termination of Rewriting. *J. Symbolic Computation*, Vol. 3, pp. 69–116, 1987.
5. 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.
6. N. Dershowitz, S. Mitra, and G. Sivakumar. Equation Solving in Conditional AC-Theories. In *Proc. ALP'90*, pp. 283–297. Springer LNCS 463, 1990.
7. R. Echahed. Uniform Narrowing Strategies. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pp. 259–275. Springer LNCS 632, 1992.
8. 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.
9. L. Fribourg. SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 172–184, Boston, 1985.
10. J.H. Gallier and S. Raatz. Extending SLD-Resolution to Equational Horn Clauses Using E-Unification. *Journal of Logic Programming (6)*, pp. 3–43, 1989.
11. J.H. Gallier and W. Snyder. Complete Sets of Transformations for General E-Unification. *Theoretical Computer Science*, Vol. 67, pp. 203–260, 1989.
12. A. Geser and H. Hussmann. Experiences with the RAP system – a specification interpreter combining term rewriting and resolution. In *Proc. ESOP 86*, pp. 339–350. Springer LNCS 213, 1986.
13. 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.
14. W. Hans, R. Loogen, and S. Winkler. On the Interaction of Lazy Evaluation and Backtracking. In *Proc. PLILP'92*, pp. 355–369. Springer LNCS 631, 1992.
15. M. Hanus. Compiling Logic Programs with Equality. In *Proc. PLILP'90*, pp. 387–401. Springer LNCS 456, 1990.
16. M. Hanus. Improving Control of Logic Programs by Using Functional Logic Languages. In *Proc. PLILP'92*, pp. 1–23. Springer LNCS 631, 1992.
17. M. Hanus. Lazy Unification with Inductive Simplification. Technical Report MPI-I-93-215, Max-Planck-Institut für Informatik, Saarbrücken, 1993.
18. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *To appear in Journal of Logic Programming*, 1994.
19. S. Hölldobler. *Foundations of Equational Logic Programming*. Springer LNCS 353, 1989.
20. 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.
21. A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 2, pp. 258–282, 1982.
22. 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.
23. 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.
24. W. Nutt, P. Réty, and G. Smolka. Basic Narrowing Revisited. *Journal of Symbolic Computation*, Vol. 7, pp. 295–317, 1989.
25. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.
26. G.D. Plotkin. Building-in Equational Theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pp. 73–90, 1972.
27. U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 138–151, Boston, 1985.
28. P. Réty. Improving basic narrowing techniques. In *Proc. of the Conference on Rewriting Techniques and Applications*, pp. 228–241. Springer LNCS 256, 1987.
29. J.H. Siekmann. An Introduction to Unification Theory. In *Formal Techniques in Artificial Intelligence*, pp. 369–425. Elsevier Science Publishers, 1990.