Handling Equality in Logic Programming via Basic Folding

Anatoli Degtyarev and Andrei Voronkov[1*]

Computing Science Department, Uppsala University, Box 311, S 751 05, Sweden

Abstract. We introduce a new approach to handling equality in logic programs. The method is based on the transformation of equational logic programs into logic programs without equality using basic superposition and the basic folding. We prove soundness, completeness and the existence of terminating strategies. The efficiency of the method is demonstrated on examples. The implementation is briefly discussed.

1 Introduction

This paper further develops an approach to logic programming with equality introduced in [10]. By a logic program with equality we understand a Horn clause logic program in which equality may occur both in the heads and in the bodies of clauses. The model-theoretic semantics of such programs is well known since long ago [5, 27]. However, there is no uniform generally accepted procedural interpretation. The most natural generalization of SLD-resolution known as SLDE-$^+$-resolution [16] have been proven in general incomplete in [10]. Various definitions of procedural interpretation are actively studied in the theorem proving and logic programming communities.

The main influence on the modern technique of equational logic programming was made by equational theorem proving methods using refinements of inference in the paramodulation-based calculus via ordering strategies and deletion of redundant clauses. This technique appeared in [24, 35, 25] and has been further developed in [20, 32, 37, 34, 3]. In logic programming, it was introduced in many papers, including [14, 22, 31, 15, 17, 1, 30], especially in the framework of the integration of logic and functional programming languages. Further references may be found in [8].

In [10] we introduced a new method called *equality elimination.* This method makes all equality-related reasoning before all other inference steps using a special strategy — basic superposition [28, 4]. Using solutions to equations provided by basic superposition, we transform a logic program with equality into a logic program without equality.

The equality elimination method combines bottom-up solutions of equations with top-down query answering in the transformed logic program. Thus, it is

* Anatoli Degtyarev is on leave from Glushkov Institute of Cybernetics, Kiev. He is supported by a grant from the Swedish Royal Academy of Sciences. Andrei Voronkov is supported by a TFR grant.

similar to tabulation, but instead of tabulating solutions to equations we tabulate *conditional solutions.* The use of conditional solutions is very essential: conditional solutions may generate a finite logic program where unconditional solutions would give an infinite one. Also, solving equations in a bottom-up way gives a complete procedure for logic programs with equality.

The equality elimination method [10] makes strong restrictions on the use of equality reasoning, while preserving completeness. However this method does not guarantee termination — there may be infinite chains of superpositions. The main aim of this paper is to enrich the equality elimination method by a new inference rule called the *basic folding.* By adding the basic folding, we can ensure termination of the equality elimination procedure, thus allowing one to obtain elegant finite programs without equality from equational logic programs. Moreover, the resulting program without equality can sometimes give a unique solution from an infinite number of solutions of the original program.

The basic folding method is complete and sound for *arbitrary* logic programs with equality. Most existing approaches use special conditions on the set of equations which may occur in programs (see, e.g. the recent survey [18]), like confluence and/or termination. Such conditions ensure comparatively efficient operational semantics (e.g. some sophisticated narrowing strategies). Unfortunately, these conditions are not easy to verify. For example, both confluence and termination are not even semi-decidable in general. Even for the decreasing conditional rewrite rules having lots of desirable properties confluence is only semi-decidable. Thus, we have a family of programming languages for which the notion of a program is undecidable. The set of programs is decidable only for non-standard notions of completeness, for example w.r.t. *strict equality* and very refined restrictions on the functional logic programs.

The basic folding method does not impose any conditions on the equality theory. In particular, the set of all positive equations need not be confluent or need not only occur in facts. At the same time, the use of orderings and simplification can significantly reduce both the number of clauses in the final program and the time needed to obtain these clauses.

2 An Informal Example

Since the formal definition of the basic folding is a bit tricky, we shall consider the main ideas on an example. The inference system we use consists of three rules: basic superposition (in fact, substitution of equal for equal with some restrictions), equality solution and the basic folding (introduction of a new definition and its use to "fold" program clauses).

Consider the following program $\mathcal{P}^2$:

[2] As we have shown in [10], this program demonstrates the incompleteness of SLDE$^+$-resolution [16] based on complete sets of E-unifiers [33]

$$v \cdot v = v \leftarrow P(v) \qquad query(x,y) \leftarrow f(x \cdot x) \cdot f(y) = f(x \cdot x) \quad (2)$$
$$P(u) \leftarrow u = f(a \cdot a) \quad (1)$$

There are three instances of the term $f(y \cdot y)$ in this set of clauses. The *basic folding rules* allow us to introduce a new predicate $E(x,y)$ expressing the fact $x = f(y \cdot y)$ together with its definition $E(x,y) \leftarrow x = f(y \cdot y)$ and replace all instances of $f(y \cdot y)$ using $E(x,y)$. Applying the basic folding to (1) and (2) we obtain a new set of clauses

$$v \cdot v = v \leftarrow P(v) \qquad query(x,y) \leftarrow z \cdot f(y) = z, E(z,x) \quad (4)$$
$$P(u) \leftarrow u = y, E(y,x), x = a \quad (3) \qquad E(x,y) \leftarrow x = f(y \cdot y) \quad (5)$$

There is a positive equation $v \cdot v = v \leftarrow P(v)$ in this clause set. It means that we can substitute any term $t \cdot t$ by t provided that $P(t)$ is a logical consequence of the program. The *basic superposition rule* allows us to make this substitution of terms for equal terms. Applying it into clauses (4) and (5) yields two new clauses

$$query(x,y) \leftarrow P(f(y)), f(y) = f(y), E(f(y),x) \quad (6)$$
$$E(x,y) \leftarrow P(y), x = f(y) \quad (7)$$

The *equality solution* rule allows us to solve an equation $s = t$ with unifiable s,t occurring in the body of a clause by removing the equation and applying a most general unifier of s,t to the rest of the clause. We can apply this rule to clauses (3), (5), (6) and (7) obtaining

$$P(u) \leftarrow E(u,a) \quad (8) \qquad query(x,y) \leftarrow P(f(y)), E(f(y),x) \quad (10)$$
$$E(f(y \cdot y),y) \quad (9) \qquad E(f(y),y) \leftarrow P(y) \quad (11)$$

There are further possible applications of these inference rules to the clauses generated so far. The theory developed in Sections 4 and 5 tells us that the derivation we have obtained is *saturated.* It means that we can stop the transformations and remove all clauses containing equality obtaining the logic program without equality consisting of four clauses (8)–(11).

Our soundness and completeness theorems of Section 5 imply that the original logic program with equality is in some sense equivalent to this logic program without equality. There is a correct answer substitution $\{a \cdot a/y, a/x\}$ for this program and the query $query(x,y)$. In addition, SLD-tree for this program and the query $query(x,y)$ is finite.

3 Preliminaries

We present here a brief overview of notions and preliminary definitions necessary for understanding the paper. For a more thorough coverage, see books and overviews [26, 19, 12, 18]. We assume the basic knowledge of substitutions, unification and reduction orderings.

All formulas belong to a fixed signature Σ containing at least one constant and a binary predicate symbol $=$. In order to distinguish the equality symbol from identity, the latter is denoted $\approx$. The notation $\rightleftharpoons$ stands for "equal by definition".

We write $A[s]$ to indicate that an expression A contains s as a subexpression and denote by $A[t]$ the result of replacing the occurrence of s in A by t. By $A\sigma$ we denote the result of applying the substitution σ to A.

The domain $dom(\sigma)$ of the substitution σ is the set $\{x \mid x\sigma \neq x\}$. The *range* $ran(\sigma)$ of the substitution σ is defined by $ran(\sigma) \rightleftharpoons \{x\sigma \mid x \in dom(\sigma)\}$. Substitutions θ with the domain $x_1, \ldots, x_n$ is denoted by the set $\{x_1\theta/x_1, \ldots, x_n\theta/x_n\}$.

The symbol $\vdash$ stands for the provability *in first order classical logic with equality.*

Definition 1. An *equation* is an atomic formula of the form $s = t$. A *clause* is any formula of the form $A_1 \wedge \ldots \wedge A_n \supset A$, where $n \geq 0$, denoted by

$$A \leftarrow A_1, \ldots, A_n$$

When $n = 0$, the symbol $\leftarrow$ will be omitted. A *logic program* is a finite set of clauses.

By a *ground expression* (i.e. term, equation, clause etc.) we mean an expression containing no variables.

We assume that reduction orderings are total on the set of ground terms. For the rest of the paper $\succ$ stands for a fixed reduction ordering.

By $var(t)$ (respectively $var(C)$) we denote the set of all variables of a term t (respectively, clause C). For a substitution σ, we define $var(\sigma) \rightleftharpoons var(dom(\sigma)) \cup var(ran(\sigma))$.

Following [4] we distinguish terms occurring in the original formula from terms introduced by a substitution by using *closures.*

Definition 2. A *closure* is a pair $C \cdot \sigma$, where C is a clause, σ a substitution. The application of a substitution ρ to such a closure is defined by $(C \cdot \sigma)\rho \rightleftharpoons C \cdot \sigma\rho$.

The set $var(C \cdot \sigma)$ is defined as $var(C) \cup var(\sigma)$. Two closures $\mathcal{C}_1, \mathcal{C}_2$ have *disjoint variables* iff $var(\mathcal{C}_1) \bigcap var(\mathcal{C}_2) = \emptyset$.

Two closures $C_1 \cdot \sigma_1$ and $C_2 \cdot \sigma_2$ are *variants* iff C_1 is a variant of C_2 and $C_1\sigma_1$ is a variant of $C_2\sigma_2$.

In the following we identify closures which are variants of each other. We note that every closure has a variant $C \cdot \sigma$ such that $dom(\sigma) \subseteq var(C)$.

The closure $C \cdot \sigma$ represents the clause $C\sigma$. Intuitively, C contains only basic positions (positions from the original set of clauses) and σ denotes a substitution introducing non-basic positions which appear as a result of an application of unification in inference rules. When we use a closure $C \cdot \theta$ on either side of the provability symbol $\vdash$, it can be replaced by the clause $C\theta$.

The *basic superposition* rules derive closures from closures. They are defined as in [4]. We always assume that premises of rules have disjoint variables. As usual, to make variables disjoint we can replace closures by their variants.

Definition 3. The *basic right superposition* and the *basic left superposition* rules are the following two inference rules:

$$\frac{(s = t \leftarrow T_1) \cdot \sigma_1 \quad (u[s'] = v \leftarrow T_2) \cdot \sigma_2}{(u[t] = v \leftarrow T_1, T_2) \cdot (\sigma_1 \cup \sigma_2)\rho}$$

and

$$\frac{(s = t \leftarrow T_1) \cdot \sigma_1 \quad (H \leftarrow T_2, u[s'] = v, T_3) \cdot \sigma_2}{(H \leftarrow T_2, T_1, u[t] = v, T_3) \cdot (\sigma_1 \cup \sigma_2)\rho}$$

respectively, where for both rules

1. ρ is a most general unifier of $s\sigma_1$ and $s'\sigma_2$;
2. $t\sigma_1\rho \not\succeq s\sigma_1\rho$;
3. $v\sigma_2\rho \not\succeq u[s']\sigma_2\rho$;
4. s' is not a variable.

and one additional restriction for basic right superposition: the equality symbol $=$ does not occur in T_2.[3]

We always assume that equations are commutative. For example, replacing $s = t$ in the above rule by $t = s$ gives a valid superposition.

We need one more inference rule. The *equality solution rule* derives closures from closures:

Definition 4. The *equality solution rule* is the following inference rule on closures:

$$\frac{H \leftarrow T_1, s = t, T_2 \cdot \sigma}{H \leftarrow T_1, T_2 \cdot \sigma\rho}$$

where ρ is a most general unifier of $s\sigma, t\sigma$ and T_1 has no occurrences of $=$.

[3] The reason why *left* superposition is a superposition into the tail (i.e. into the *right* hand side of the clause) is historical: superposition is often defined as an inference rule on *sequents*, i.e. expressions of the form $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$, then left superposition is a superposition in one of A_i. A clause $H \leftarrow T$ corresponds to the sequent $T \rightarrow H$, hence left superposition becomes a superposition into the tail.

In the subsequent sections we shall describe a program transformation system, i.e. a system transforming programs (sets of closures) into programs (sets of closures). We shall formalize it as a transition system on sets of closures. We have to include usual inference rules, like basic superposition, in such transition systems. To this end we represent inference rules on closures as transition rules on sets of closures.

Let Ω be any calculus of closures, i.e. any set of inference rules inferring closures from closures. The calculus Ω can be considered as an inference system Ξ allowing to derive *sets of closures* from sets of closures in the following way: the inference rule $S \Rightarrow S \cup \{\mathcal{C}\}$ (S can be transformed into $S \cup \{\mathcal{C}\}$), where S is a set of closures, is an inference rule of Ξ iff there are closures $\mathcal{C}_1, \ldots, \mathcal{C}_n \in S$ such that the rule $\mathcal{C}_1 \cdots \mathcal{C}_n / \mathcal{C}$ belongs to Ω. Using this correspondence, basic superposition and equality solution will also be considered as rules on sets of closures. The notation $\Rightarrow$ for the inference rules acting on sets of closures is chosen in order to stress that such rules can be considered as state transitions. We need to consider ordinary inference rules as rules on sets of closures because we shall also use basic folding rules *replacing* closures by other closures which are easily formalized as state transitions.

4 The Basic Folding

The basic folding is based on a transformation of sets of closures removing the equality predicate from the program. Unlike the equality elimination method studied in [10], there are strategies for the basic folding which always terminate resulting in a finite program. The essence of the basic folding method is the following. We systematically try to solve equations in the bodies of closures using basic superposition. When there are growing terms so that basic superposition transformations do not terminate, we introduce a new predicate definition via the basic folding rule and "fold" the closure containing the terms.

Let $\mathcal{X} = x_0, x_1, x_2, \ldots$ be a fixed countable sequence of variables.

Definition 5. A term t is called $\mathcal{X}$*-normal* iff its variables (in the order of their first occurrences in t) are $x_1, \ldots, x_n$. The $\mathcal{X}$*-normal form* of a term t is the variant of t which is $\mathcal{X}$-normal.

For example, the $\mathcal{X}$-normal form of the term $g(x, y, f(a, x))$ is $g(x_1, x_2, f(a, x_1))$.

Let S be a set of closures. By $\mathcal{R}(S)$ we denote the set of clauses $\mathcal{R}(S) \rightleftharpoons \{C\sigma \mid C \cdot \sigma \in S\}$.

Let $\mathcal{P}$ be a logic program in a signature Σ. We assume that the program contains a definition of the predicate *query* of n arguments.

Definition 6. An *answer* to the program $\mathcal{P}$ is any tuple $\langle t_1, \ldots, t_n \rangle$ of terms, where n is the arity of the predicate *query* in $\mathcal{P}$. Such an answer is a *correct answer to* $\mathcal{P}$ iff $\mathcal{P} \vdash query(t_1, \ldots, t_n)$.

This definition can be rephrased for closures:

Definition 7. An *answer* to the set of closures S is any tuple $\langle t_1, \ldots, t_n \rangle$ of terms, where n is the arity of the predicate *query* in S. Such an answer is a *correct answer to* S iff $\mathcal{R}(S) \vdash query(t_1, \ldots, t_n)$.

From now on we assume that the signature Σ is fixed and all logic programs are written in this signature. Let Pr be a countable set of predicate symbols disjoint from predicate symbols of Σ. Let there be a mapping that associates with each $\mathcal{X}$-normal term t a unique predicate symbol $\mathcal{E}_t \in Pr$. The arity of $\mathcal{E}_t$ is the number of different variables in t plus 1. We shall use the new predicate symbols $\mathcal{E}_t$ for introducing new predicate definitions for the atoms $x_0 = t$.

Basic superposition only allows for superpositions in the basic positions. Below we formalize the notion of a position.

Definition 8. A *position* is a finite sequence of positive integers. The notions of a *position in a term* t and the subterm $t\,|_\pi$ of t at the position π are defined inductively as follows:

1. The *empty position* ε is a position in any term t and $t\,|_\varepsilon \rightleftharpoons t$.
2. Let the term t have the form $f(t_1, \ldots, t_n)$, where $n \geq 1$, π be a position in t_i and $t_i\,|_\pi \approx s$. Then $i\pi$ is a position in t and $t\,|_{i\pi} \rightleftharpoons s$.

A position π in t is a *non-variable position* iff $t\,|_\pi$ is not a variable.

Definition 9. Let τ be a substitution with $dom(\tau) = \{y_1, \ldots, y_m\}$. The *flattening* of τ (denoted $\mathcal{S}_\tau$) is a sequence of equations $y_1 = y_1\tau, \ldots, y_m = y_m\tau$.

Let t be an $\mathcal{X}$-normal term. Then Def_t is the closure

$$\mathcal{E}_t(x_0, x_1, \ldots, x_n) \leftarrow x_0 = t \cdot \varepsilon$$

The following definition is the main definition of this section.

Definition 10 (basic folding). The *basic left folding rule* is defined as follows. Assume that S is a set of closures and $\mathcal{C} \in S$ is a closure of the form:

$$H \leftarrow T_1, A[t\tau], T_2 \cdot \theta$$

where t is not a variable. Let the term t' be an $\mathcal{X}$-normal term, variables $x_1, \ldots, x_n$ are all variables of t', the variables $x_0, x_1, \ldots, x_n$ do not occur in $var(\mathcal{C}) \cup var(\tau)$, $dom(\tau) = \{y_1, \ldots, y_m\}$ and $y_1, \ldots, y_m$ do not occur in $var(\mathcal{C})$. Let $\mathcal{C}'$ be the closure

$$H \leftarrow T_1, A[x_0], \mathcal{E}_{t'}(x_0, x_1, \ldots, x_n), \mathcal{S}_\tau, T_2 \cdot \sigma \cup \theta$$

such that

1. $dom(\sigma) = \{x_1, \ldots, x_n\}$;
2. Every non-variable position π in t is a non-variable position in t';
3. $t'\sigma \approx t\theta$;
4. $\mathcal{C}$ is not a variant of $Def_{t'}$.

Then the basic left folding rule is the rule $S \Rightarrow S \setminus \{\mathcal{C}\} \cup \{\mathcal{C}', Def_{t'}\}$.

The *basic right folding rule* is defined in the same way as the basic left folding rule, but with $\mathcal{C}, \mathcal{C}'$ defined as

$$A[t\tau] \leftarrow T \cdot \theta \quad \text{and} \quad A[x_0] \leftarrow \mathcal{E}_{t'}(x_0, x_1, \ldots, x_n), \mathcal{S}_\tau, T \cdot \sigma \cup \theta$$

respectively. We also say that $\mathcal{C}'$ is obtained by a basic folding *into* $\mathcal{C}$.

Below we shall show that one can use a weaker form of the folding: $S \Rightarrow S \setminus \{\mathcal{C}\} \cup \{\mathcal{C}'\}$ provided that the predicate $\mathcal{E}_{t'}$ occurs in S.

Consider an example.

Example 1. Let $\mathcal{C}$ be the closure

$$A(f(y,z)) \leftarrow B(y,z), A(f(g(z),y)) \cdot \{b/z, h(a)/y\}$$

Following the notation of Definition 10, let

$$\begin{aligned}
\theta &\rightleftharpoons \{b/z, h(a)/y\}\\
t' &\rightleftharpoons f(x_1, h(x_2))\\
t &\rightleftharpoons f(y_1, y)\\
Def_{t'} &\rightleftharpoons \mathcal{E}_{f(x_1,h(x_2))}(x_0, x_1, x_2) \leftarrow x_0 = f(x_1, h(x_2)) \cdot \varepsilon\\
\tau &\rightleftharpoons \{g(z)/y_1\}\\
\mathcal{S}_\tau &\rightleftharpoons y_1 = g(z)
\end{aligned}$$

Note that t' has more non-variable positions than t. The condition $t\theta \approx t'\sigma$ yields $\sigma = \{y_1/x_1, a/x_2\}$. Then $\mathcal{C}'$ has the form

$$\begin{aligned}
A(f(y,z)) \leftarrow\ & B(y,z), A(x_0), \mathcal{E}_{f(x_1,h(x_2))}(x_0, x_1, x_2), y_1 = g(z)\\
& \cdot\{b/z, h(a)/y, y_1/x_1, a/x_2\}
\end{aligned}$$

In the definition of the basic folding, we allow to "fold" closures using the definition for a term t' which can be different from t. For completeness, it is enough to only fold using t. However such a restriction of the basic folding would disallow some elegant applications of the folding, like the one used below in Example 2. In fact the idea of using t' instead of t has been obtained during experiments with our first implementation.

Further explanation of the intended meaning of the substitution τ used in the basic folding rules is given in Example 2 in Section 6. We need to keep $\mathcal{S}_\tau$ for not losing non-variable positions which may result in incompleteness.

The basic folding method uses the logical system $\mathcal{BF}$ defined below that acts on sets of closures of a special kind. Before applying the inference rules of $\mathcal{BF}$, we transform the equational logic program into an *initial* set of closures:

Definition 11. A closure $C \cdot \sigma$ is *basic* iff it satisfies the following condition: whenever an atomic formula $P(t_1, \ldots, t_n)$ occurs in C and P is a predicate symbol different from $=$, then all terms t_i are variables. A set of closures S is called *initial* iff

1. All closures in S are basic and have the form $C \cdot \varepsilon$;
2. Predicate symbols $\mathcal{E}_t$ do not occur in S.

There are many ways of transforming a logic program into an initial set of closures. One standard way is the following: all clauses C of the form

$$H \leftarrow T_1, A(t_1, \ldots, t_n), T_2 \quad \text{or} \quad A(t_1, \ldots, t_n) \leftarrow T$$

respectively, where the predicate symbol A is different from equality, are replaced by

$$H \leftarrow T_1, y_1 = t_1, \ldots, y_n = t_n, A(y_1, \ldots, y_n), T_2$$

or

$$A(y_1, \ldots, y_n) \leftarrow y_1 = t_1, \ldots, y_n = t_n, T$$

respectively, where $y_1, \ldots, y_n$ are new variables. This transformation is applied until all non-equality predicate symbols only contain variables as arguments. Another, more complicated way is described in [10] and is based on the introduction of new predicate symbols for all complementary pairs of literals in the program.

Definition 12. The inference system $\mathcal{BF}$ is the set of inference rules acting on sets of closures and consisting of 3 inference rules:

1. basic (left and right) superposition;
2. equality solution;
3. basic (left and right) folding.

A $\mathcal{BF}$*-derivation* Δ is any sequence of sets $S_0 \Rightarrow S_1 \Rightarrow \ldots$ such that every S_{i+1} is obtained from S_i by an application of an inference rule of $\mathcal{BF}$.

The *limit* S_ω of the $\mathcal{BF}$-derivation Δ is the set $\bigcup_j \bigcap_{k \geq j} S_k$ of closures. The derivation Δ is called *fair* iff for every set S such that S can be deduced from S_ω by an application of basic superposition or equality solution, and for every closure $\mathcal{C} \in S \setminus S_\omega$, we have $\mathcal{C} \in S_i$, for some i.

A finite derivation $\Delta \rightleftharpoons (S_0 \Rightarrow \ldots \Rightarrow S_n)$ is called *saturated* iff for every set S such that S can be deduced from S_n by basic superposition or equality solution, and for every closure $\mathcal{C} \in S \setminus S_n$, we have $\mathcal{C} \in S_i$, for some $i \leq n$.

Note that a derivation is saturated iff it is finite and fair.

For a set of closures S, by S^- we denote a set of closures obtained from S by the removal of *all* closures in which the equality symbol $=$ occurs.

Given an equational logic programs, the basic folding method tries to construct a saturated derivation. When such a derivation $S_0 \Rightarrow \ldots \Rightarrow S_n$ is found, all clauses containing the equality predicate are removed from the final set of clauses $\mathcal{R}(S_n)$. In Section 5 we discuss the existence of saturated derivations and the soundness and completeness of the method. The soundness and completeness mean that for any fair derivation the initial equational logic program S_0 and the final logic program $\mathcal{R}(S_\omega^-)$ without equality are, in some sense, equivalent. In Section 6 we consider examples. Section 7 discusses related works. In Section 8 we briefly describe an implementation of the basic folding. Some experimental results may be found in [8].

5 Soundness and Completeness

In this section we formulate the main properties of the basic folding, namely soundness, completeness and termination. Complete proofs of these properties may be found in [8].

Theorem 13 (soundness). *Let S_0 be an initial set of closures, $\Delta = (S_0 \Rightarrow S_1 \Rightarrow \ldots)$ be any $\mathcal{BF}$-derivation, and S_ω be the limit of Δ. Then any correct answer for S_ω^- is a correct answer for S_0.*

The proof of this theorem is based on the conservativity of the basic folding rule.

Theorem 14 (completeness). *Let S_0 be an initial set of closures, $\Delta = (S_0 \Rightarrow S_1 \Rightarrow \ldots)$ be any fair derivation, and S_ω be the limit of Δ. If S_0 has a correct answer then S_ω^- has a correct answer.*

The proof of this theorem [8] is based on numerous proof transformations in the system consisting of resolution, basic superposition, equality solution and some other rules. The proof uses a result of [28, 29] on the completeness of a basic superposition calculus with ordering constraint inheritance.

Note that Theorem 14 and the compactness theorem imply that there is n such that $\mathcal{R}(S_n^-)$ has a correct answer. Also note that by Theorem 13 every correct answer to $\mathcal{R}(S_n^-)$ is a correct answer to $\mathcal{R}(S_0)$.

We can use a better folding rule which does not introduce the same definition twice: it is possible not to add the closure $Def_{t'}$ into the new set of closures provided only that the predicate $\mathcal{E}_{t'}$ occurs in S. The completeness of this strategy is shown in [8].

The completeness theorem here may seem not very striking since it does not characterize the relation between the computed answer substitutions and all correct answer substitutions. Thus, it looks like a satisfiability statement. Let us consider this aspect in detail. It was precisely one of the main aims of the basic folding to avoid the generation of *all* correct answers. The known operational semantics of functional logic programming (see [18]) also compute only *some* answers. The set of computed answer substitutions for the known operational principles of functional logic programs is *complete* in the sense that for every correct answer substitution θ there is a computed answer substitution σ such that $\sigma \leq_E \theta$. The problem is that the a complete set of answer substitutions w.r.t. an equational theory E does not give much information about the set of all correct answer substitutions. An answer substitution σ for a query Q gives no information on whether another substitution θ is also a correct answer (the problem $x\theta =_E x\sigma$ may be undecidable, even when θ, σ are ground and E is represented by a canonical conditional term rewrite system without extra variables [21]). It means that if we would like to know whether θ is a correct answer substitution to Q we have to ask the query $Q\theta$ instead. But we can as well ask $Q\theta$ in our setting and our completeness theorem guarantees a correct answer in the same way as the completeness theorems for functional logic programs. Thus, the difference between the two formulations of completeness is really marginal. In addition, the complete sets of solutions (complete sets of E-unifiers) do not behave well for equational logic programs in general. For example, in [10] we show that the minimal model of a program is not a fixpoint when we use complete sets of unifiers in the definition of the immediate consequence operator.

Following [30], we could have changed the completeness statement in a way more traditional for logic programming via answer substitutions using a suitable notion of minimality. However it requires several new notions and a different technique. We plan to further investigate this subject.

Theorem 15 (existential termination). *For any initial set of closures S_0 there is a saturated derivation $S_0 \Rightarrow \ldots \Rightarrow S_n$.*

The proof of this theorem is based on the folding of the terms in the heads of all clauses defining equality that is similar (but not equivalent) to the transformation of [13]. However, this strategy does not give elegant programs (see the discussion after Example 2 below). We shall consider more interesting strategies in Section 6.

Note that by Theorems 13 and 14 it follows that for every saturated derivation $S_0 \Rightarrow \ldots \Rightarrow S_n$

1. Every correct answer to $\mathcal{R}(S_n^-)$ is a correct answer to $\mathcal{R}(S_0)$;
2. If $\mathcal{R}(S_0)$ has a correct answer then $\mathcal{R}(S_n^-)$ has a correct answer.

The above three theorems form the theoretical basis for the basic folding. The main aim of the method is to obtain a saturated derivation from the initial set of closures representing a logic program with equality. Then we can remove all closures containing equality from the limit set of the saturated derivation. The above theorems guarantee that the resulting logic program without equality is sound and complete w.r.t. the initial equational logic program.

6 Examples

We have informally considered an example in Section 2. Here we consider another example. Its aim is to demonstrate that the basic folding can yield very natural programs. This example (with a small modification) was checked experimentally [8]. In the presentation below, we shall sometimes replace closures by their variants. Instead of a closure $C \cdot \varepsilon$ we write the clause C.

Example 2. Consider the following program defining the function ap (append) and the predicate r (reverse) for lists as follows:

$$
\begin{array}{ll}
ap(\texttt{[]}, x) = x & r(\texttt{[}x \,|\, y\texttt{]}, ap(z, \texttt{[}x\texttt{]})) \leftarrow r(y, z) \\
ap(\texttt{[}x \,|\, y\texttt{]}, z) = \texttt{[}x \,|\, ap(y, z)\texttt{]} & query(x) \leftarrow r(\texttt{[}a, b\texttt{]}, x) \\
r(\texttt{[]}, \texttt{[]}) &
\end{array}
$$

It can be translated into the following initial set of closures:

$$
\begin{array}{l}
ap(\texttt{[]}, x) = x \\
ap(\texttt{[}x \,|\, y\texttt{]}, z) = \texttt{[}x \,|\, ap(y, z)\texttt{]} \\
r(x, x) \leftarrow x = \texttt{[]} \\
r(u, v) \leftarrow u = \texttt{[}x \,|\, y\texttt{]}, v = ap(z, \texttt{[}x\texttt{]}), r(y, z) \\
query(x) \leftarrow y = \texttt{[}a, b\texttt{]}, r(y, x)
\end{array}
\tag{12}
$$

For the folding we shall use the new predicate $\mathcal{E}_{ap(x_1, [x_2])}$, for simplicity denoting it by E. Comparison of terms will be made according to the recursive path ordering with precedence $ap > \texttt{[|]} > \texttt{[]}$. Then in both equations for ap the left hand side is greater than the right hand side. The basic left folding into (12) yields the set

$$ap(\texttt{[]}, x) = x \tag{13}$$
$$ap(\texttt{[}x \,|\, y\texttt{]}, z) = \texttt{[}x \,|\, ap(y, z)\texttt{]} \tag{14}$$
$$r(x, x) \leftarrow x = \texttt{[]}$$
$$E(x, y, z) \leftarrow x = ap(y, \texttt{[}z\texttt{]}) \tag{15}$$
$$r(u, v) \leftarrow u = \texttt{[}x \,|\, y\texttt{]}, v = w, E(w, z, x), r(y, z)$$
$$query(x) \leftarrow y = \texttt{[}a, b\texttt{]}, r(y, x)$$

Basic left superpositions of (13) and (14) into (15) yield

$$
\begin{array}{l}
ap(\texttt{[]}, x) = x \\
ap(\texttt{[}x \,|\, y\texttt{]}, z) = \texttt{[}x \,|\, ap(y, z)\texttt{]} \\
r(x, x) \leftarrow x = \texttt{[]} \\
E(x, y, z) \leftarrow x = ap(y, \texttt{[}z\texttt{]}) \\
E(x, y, z) \leftarrow x = u \cdot \{\texttt{[]}/y, \texttt{[}z\texttt{]}/u\}
\end{array}
$$

$$E(x,y,z) \leftarrow x = [u\,|\,ap(v,w)] \cdot \{[u\,|\,v]/y, [z]/w\} \tag{16}$$
$$r(u,v) \leftarrow u = [x\,|\,y], v = w, E(w,z,x), r(y,z)$$
$$query(x) \leftarrow y = [a,b], r(y,x)$$

Applying the basic left folding to (16) we obtain

$$ap([\,], x) = x$$
$$ap([x\,|\,y], z) = [x\,|\,ap(y,z)]$$
$$r(x,x) \leftarrow x = [\,] \tag{17}$$
$$E(x,y,z) \leftarrow x = ap(y, [z]) \tag{18}$$
$$E(x,y,z) \leftarrow x = u \cdot \{[\,]/y, [z]/u\} \tag{19}$$
$$E(x,y,z) \leftarrow x = [u\,|\,w], E(w,v,z) \cdot \{[u\,|\,v]/y\} \tag{20}$$
$$r(u,v) \leftarrow u = [x\,|\,y], v = w, E(w,z,x), r(y,z) \tag{21}$$
$$query(x) \leftarrow y = [a,b], r(y,x) \tag{22}$$

After applying equality solution to (17)–(22) we obtain a saturated set of closures. Now we can remove all closures containing equality and replace closures by clauses:

$$r([\,], [\,])$$
$$E(ap(y, [z]), y, z)$$
$$E([z], [\,], z) \tag{23}$$
$$E([u\,|\,x], [u\,|\,v], z) \leftarrow E(x,v,z) \tag{24}$$
$$r([x\,|\,y], v) \leftarrow E(v,z,x), r(y,z)$$
$$query(x) \leftarrow r([a,b], x)$$

Note that clauses (23)–(24) look like the usual recursive definition of the *append* predicate[4].

SLD-resolution yields four solutions for this program:

$$\{ap([b], [a])/x\} \qquad \{[b\,|\,ap([\,], [a])]/x\}$$
$$\{[b,a]/x\} \qquad \{ap(ap([\,], [b]), [a])/x\}$$

The program transformation by the basic folding in general depends on the query. Note, however, that neither the goal clause (i.e. the clause containing the *query* predicate) nor any clause obtained from it by the rules of $\mathcal{BF}$ never contains positive equations. This means that it is possible to separate the $\mathcal{BF}$ derivations of the program without the goal and the program with the goal in the

[4] In fact, $E(x,y,z)$ does not denote $x = ap(y,z)$ but rather $x = ap(y, [z])$.

following way. First, we construct a saturated derivation of the program without the goal. Then we add the goal clause and consider possible applications of the inference rules to the goal clause (or its successors). It is quite possible that there are no basic superpositions into the goal clause or that we can fold the goal clause using the definitions already introduced. In this case there is no additional overhead for adding the goal. This situation is quite general: for instance, in our append-reverse example the transformation is the same for any goal not using the function symbol *ap*. At the same time, the goal-oriented transformation may give better results (the partial evaluation effect).

Our logical system is highly non-deterministic. There are many ways of defining a concrete strategy for applying rules. In fact, the strategy of applying the basic folding used in our proof of Theorem 15 guarantees termination, but does not give interesting programs. For instance, this strategy applied to Example 2 below generates 15 clauses, one of them has the form

$$C(x, y, z) \leftarrow A(\texttt{[}y \,|\, z\texttt{]}, u, v), C(u, w, x'), C(x, w, y'), A(y', x', w)$$

where A and C are new predicate symbols denoting the relations $x = ap(y, z)$ and $x = \texttt{[}y \,|\, z\texttt{]}$. This clause (not alone) encodes the equation $ap(\texttt{[}x \,|\, y\texttt{]}, z) = \texttt{[}x \,|\, ap(y, z)\texttt{]}$.

A more interesting strategy for the basic folding is to check for repeated occurrences of the same subterm in equations and to apply the folding to these subterms. Unlike works on folding-unfolding transformations of programs, both these strategies of the folding are easily mechanizable. In fact, the subterm $ap(y, \texttt{[}z\texttt{]})$ used for a folding in Example 2, has been found by our system completely automatically.

Our method is also applicable to subsets of Prolog with equality. If equality axioms are defined by definite clauses without any extralogical features our method still works, even if the rest of the program uses some non-logical control primitives. For example, our method is compatible with cuts of the form

$$A \leftarrow T, [!B_1, \ldots, B_n!], T'.$$

used in the languages Arity Prolog and Gödel.

Our method is, however, incompatible with unrestricted cuts, like

$$A \leftarrow A_1, \ldots, !, \ldots, A_n$$

In the future, we are going to extend the basic folding to non-Horn clauses with the aim of applications in automated theorem proving.

7 Related work

In this paper we are trying to "compile out" equational reasoning with the aim of obtaining an equivalent logic program without equality, which can be executed by ordinary SLD-resolution. This reminds the process of completion of equational theories which can also be characterized as a transformation of an equational theory to a canonical form. As a rule, this kind of processes does not terminate, i.e. one cannot obtain a finite canonical program (see e.g. "unfailing completion" [2]). However, any query which is a consequence of the program can be answered at a finite stage of this process by narrowing. For Horn clause theories it has been developed in [17]. Our approach is very different from the cited papers because we can always obtain a finite program using the generation of recursive rules.

The basic strategy without ordering restrictions has been introduced by the first author in 1979 (published in Russian, see the survey [9]). The motivation was to reduce the search space by restricting the paramodulation at the term level. The basic superposition inference systems have been introduced in [4, 28].

The *modification method* of Brand [7] is somewhat similar to the flattening strategy of the basic folding. He has shown how to get rid of the equality substitution axioms by pulling nonvariable terms out from argument lists. The modification method transforms the original Horn clause program with equality into "flat" Horn clauses without equality. Then one can use SLD-resolution as the inference mechanism for the transformed program. In logic programming community, this method has been considered in [23, 6]. Brand does not introduce new defining predicates for equations, he puts the equations themselves into the bodies of clauses. After the transformation, he treats the equality predicate as the ordinary binary predicate Eq defined by the fact $Eq(x, x)$. The essential difference between our method and that of Brand is that we try to *solve* all equations in the bodies by specialized applications of equality rules: basic superposition and equality solution. It allows us to automatically "subsume" the axioms of symmetry and transitivity, and also to essentially decrease the number of logical consequences due to the use of orderings. According to the Brand's method, these axioms must be explicitly applied to all positive occurrences of the predicate Eq. Compare the program obtained in Section 2 by the basic folding method with the program obtained by the Brand's method:

$$
\begin{aligned}
&Eq(x, x)\\
&Eq(v, w) \leftarrow Eq(v \cdot v, y), Eq(y, w), P(v)\\
&Eq(y, w) \leftarrow Eq(v \cdot v, y), Eq(v, w), P(v)\\
&P(u) \leftarrow Eq(f(z), u), Eq(x \cdot x, z), Eq(a, x)\\
&query(x, y) \leftarrow Eq(v \cdot z, v), Eq(f(w), v), Eq(x \cdot x, w), Eq(f(y), z)
\end{aligned}
$$

This logic program is much more complicated and the SLD-tree for this program and the query $query(x, y)$ is infinite. The program generated by the Brand's method for Example 2 yields an infinite number of solutions.

The transformation of programs into the homogeneous form [13] allows one to get rid of the axioms of transitivity and predicate replaceability (i.e. they become "built-in") but the symmetry and the function replaceability axioms remain. Flattening transformation of [23] is analogous to the TE-transformation by Brand and it subsumes all axioms but reflexivity and symmetry. (Symmetry can further be avoided by, in fact, doubling the number of clauses describing equality.) But as we have already demonstrated in Section 2, the set of clauses obtained by such a transformation is unnecessarily large and not always compatible with the Prolog technology. Programs obtained as the result of our transformation are more elegant because we partially solve equations using very strong techniques of equality reasoning. This is not achieved by other methods which try to move all equality reasoning to SLD-resolution.

The automatic introduction of new function symbols for defining some useful subterms have been used in automated deduction systems [36, 38].

8 Implementation

Our method has been implemented by Yu. Koval' as a program transformation system written in Sicstus Prolog (about 1500 lines of Prolog code). For some more complicated examples a straightforward implementation of our technique can generate many unrelated or excessive closures. In order to make the program transformation faster and more efficient we introduced a few simplification rules. *Basic subsumption* removes closures subsumed by other closures. Basic subsumption is described e.g. in [4]. On closures without equality, i.e. clauses used in the resulting program, basic subsumption is equivalent to subsumption (because the resulting closures have no basic positions). *Tautology deletion* removes closures $s = t \leftarrow T \cdot \theta$ such that $s\theta \approx t\theta$ (tautologies). It is also applicable to closures $A \leftarrow \ldots B \ldots \cdot \theta$ such that $A\theta \approx B\theta$. *Double literal elimination* removes the literal B in a closure $H \leftarrow \ldots A \ldots B \ldots \cdot \theta$ such that $A\theta \approx B\theta$ and A is not an equation.

One more strategy that can be applied but is not yet implemented is the simplification in the basic setting [4].

Currently, our system uses a method for generating initial closures based on the analysis of all complementary pairs of literals [10].

The implementation was powerful enough to transform (completely automatically) both examples of our paper. Despite a different way of generating initial closures, the final programs looked very similar to the ones described in the paper. For instance, the program of Section 2 had been transformed into a program having a finite SLD-tree. For the program of Example 2, the predicate definition of the append relation has been automatically generated. It is interesting that the original folding rule proposed in [10] has been strengthened due to an analysis of the experimental results.

References

1. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *ACM Symposium on Principles of Programming Languages*, pages 268–279, Portland, 1994.
2. L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In H. Aït Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Academic Press, 1989.

3. L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.
4. L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 462–476, Saratoga Springs, NY, USA, June 1992. Springer Verlag.
5. G. Birkhoff. Subdirect unions in universal algebras. *Bull. Amer. Math. Soc.*, 50:764–768, 1944.
6. P.G. Bosco, E. Giovanetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.
7. D. Brand. Proving theorems with the modification method. *SIAM Journal of Computing*, 4:412–430, 1975.
8. A. Degtyarev, Yu. Koval, and A. Voronkov. Handling equality in logic programming via basic folding. UPMAIL Technical Report 101, Uppsala University, Computing Science Department, March 1995.
9. A. Degtyarev and A. Voronkov. Methods of handling equality in machine theorem proving (in Russian). *Kibernetika*, (3), 1986. English translation in Cybernetics 22 (1986), pp. 298–307.
10. A. Degtyarev and A. Voronkov. A new procedural interpretation of Horn clauses with equality. In Leon Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming*, pages 565–579. The MIT Press, 1995.
11. A. Degtyarev and A. Voronkov. Equality elimination for the inverse method and extension procedures. In C.S. Mellish, editor, *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, volume 1, pages 342–347, Montréal, August 1995.
12. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–309. North Holland, Amsterdam, 1990.
13. E.W. Elcock. Prolog: Subsumption of equality axioms by the homogeneous form. *Journal of Logic Programming*, pages 45–56, 1989.
14. L. Fribourg. Slog: A logic programming language interpreter based on clausal superposition and rewriting. In *IEEE International Symposium on Logic Programming*, pages 172–184, Boston, 1985.
15. U. Furbach, S. Hölldobler, and J. Schreiber. Horn equality theories and paramodulation. *Journal of Automated Reasoning*, 5:309–337, 1989.
16. J.H. Gallier and S. Raatz. Extending SLD-resolution to equational Horn clauses using E-unification. *Journal of Logic Programming*, 6(3):3–44, 1989.
17. H. Ganzinger. A completion procedure for conditional equations. *Journal of Symbolic Computations*, 11:51–81, 1991.
18. M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19,20:583–628, 1994.
19. S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 1989.
20. J.M. Hullot. Canonical forms and unification. In *5th CADE*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334, 1980.
21. S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984.
22. S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination and confluence. *Journal of Symbolic Computations*, 4(3):295–334, 1987.

23. E. Knill, P.T. Cox, and T. Pietrzykowski. Equality and abductive residua for Horn clauses. *Theoretical Computer Science*, 120:1–44, 1993.
24. D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.
25. D.S. Lankford. Canonical inference. Technical report, Department of Mathematics, South-Western University, Georgetown, Texas, 1975.
26. J.W. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer Verlag, 1987.
27. A. Maĺcev. Subdirect products of models. *Dokl. Akad. Nauk SSSR (in Russian)*, 109:264–266, 1956.
28. R. Nieuwenhuis and A. Rubio. Basic superposition is complete. In *ESOP'92*, volume 582 of *Lecture Notes in Computer Science*, pages 371–389. Springer Verlag, 1992.
29. R. Nieuwenhuis and A. Rubio. Theorem proving with ordering and equality constrained clauses. *Journal of Symbolic Computations*, 11(1):1–32, 1995.
30. R. Nieuwenhuis. On narrowing, refutation proofs and constraints. In J. Hsiang, editor, *Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 56–70, 1995.
31. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, 1988.
32. G.E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM Journal of Computing*, 12(1):82–100, 1983.
33. G. Plotkin. Building-in equational theories. In Meltzer and Michie, editors, *Machine Intelligence*, volume 7, pages 73–90. Edinburgh University Press, Edinburgh, 1972.
34. M. Rusinowitch. Theorem proving with resolution and superposition: an extension of the Knuth and Bendix completion procedure as a complete set of inference rules. *Journal of Symbolic Computations*, 11:21–49, 1991.
35. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the Association for Computing Machinery*, 21(4):622–642, 1974.
36. S. Winker. Robbins algebra: Conditions that make a near-boolean algebra boolean. *Journal of Automated Reasoning*, 6(4):465–489, 1990.
37. H. Zhang and D. Kapur. First-order theorem proving using conditional rewrite rules. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 1–20, 1988.
38. H. Zhang. Automatic proofs of equality problems in Overbeek's competition. *Journal of Automated Reasoning*, 11(3):333–351, 1993.