Translating Logic Programs into Conditional Rewriting Systems

Femke van Raamsdonk
CWI
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
femke@cwi.nl

Abstract

In this paper a translation from a subclass of logic programs consisting of the simply moded logic programs into rewriting systems is defined. In these rewriting systems conditions and explicit substitutions may be present. We argue that our translation is more natural than previously studied ones and establish a result showing its correctness.

1 Introduction

Logic and functional programming are both instances of declarative programming and hence it is not surprising that the relationship between them has been studied. However, the work so far has in our opinion not yet resulted in clear cut and simple to state results clarifying this relationship. Moreover, most of the work in the area concerns only termination of logic programming, via a translation into term rewriting systems. See Section 5 for a discussion of related work.

The aim of the present paper is to relate in a precise way the operational semantics of logic programming, resolution, to the operational semantics of functional programming, rewriting, thus abstracting from the syntactic details of particular programming languages. We discuss extensively the merits and deficiencies of possible translations and argue that the use of *conditions* and *explicit substitutions* makes it possible to design a natural and intuitive translation. Our translation can be used as a basis for an alternative implementation of a subset of logic programming via a translation to functional programs.

We provide a rigorous result showing the correctness of our translation. This result states that one resolution step using a clause C is translated into one or more rewrite steps, all using the rewrite rule C^*, which is the translation of C. Hence in particular termination of a logic program is implied by termination of its translation. Moreover, a successful resolution sequence is translated into a rewriting sequence that ends in an expression in normal form, from which the computed answer substitution can be read immediately.

2 Preliminaries

We assume the reader to be familiar with logic programming and refer to [1] for an overview. In this section we fix the notation and give the definitions that are less well-known.

We assume a set $\mathcal{V}$ consisting of infinitely many *variables* written as $x, y, z, \ldots$. A *logic program* is a triple of the form $(\mathcal{F}, \mathcal{R}, \mathcal{C})$ with $(f, g \in)$ $\mathcal{F}$ a set of *function symbols*, $(r, f, g \in)$ $\mathcal{R}$ a set of *relation symbols* and $(C, C' \in)$ $\mathcal{C}$ a set of *clauses* over $(\mathcal{F}, \mathcal{R})$. Queries are denoted by $Q, Q', \ldots$, and the empty query is written as $\Box$. Terms are denoted by $s, t, \ldots$ and atoms by $a, b, \ldots$.

Substitutions are denoted by $\sigma, \tau, \ldots$. The identity substitution is denoted by ϵ, and the composition of substitutions σ and τ is denoted by $\sigma\tau$. The result of applying a substitution σ to a term s is denoted by $s\sigma$.

The set of free variables occurring in an expression X is denoted by $\mathsf{V}(X)$. We denote the union of the variables in the domain and the variables in the codomain of a substitution σ by $\mathsf{V}(\sigma)$.

The relation symbols of the logic programs considered in this paper use some arguments as input and some arguments as output. This is formalized using the notion of modes. Modes were introduced by Mellish [9] and further studied by Reddy [10]. A *base mode* is either *input*, denoted by $\downarrow$, or *output*, denoted by $\uparrow$. An *m-ary mode* is a product, denoted using $\times$, of m base modes. Without loss of generality, an m-ary mode is of the form $\downarrow \times \ldots \times \downarrow \times \uparrow \times \ldots \times \uparrow$ with first p times $\downarrow$ and then q times $\uparrow$, and $p + q = m$. Such a mode is denoted by (p, q). In the remainder of this paper, every relation symbol is supposed to have a fixed mode. The following convention will be used.

Notation 2.1 If r is a relation symbol of mode (p, q), then $r(\vec{s}, \vec{t})$ denotes the atom with terms $\vec{s} = s_1, \ldots, s_p$ in the input positions of r and terms $\vec{t} = t_1, \ldots, t_q$ in the output positions of r. Note that p and q may be zero. The length of a sequence $\vec{s}$ is denoted by $|\vec{s}|$.

One of the main differences between logic programming and rewriting is that the resolution relation of a logic program is defined using unification whereas the rewrite relating of a rewriting system is defined using matching. Apt and Etalle identify in [2] several classes of Prolog programs for which unification can be replaced by iterated matching. One of these classes consists of programs that are well-moded and satisfy in addition another restriction; in the present paper these programs are said to be *simply moded*. The class of simply moded logic programs is used as the domain of the translation defined in Section 4. For the definition of simply modedness we first need the definition of well-modedness, which is originally due to Dembiński and Małuszyński [5]. The following definition is taken from [1]. Intuitively, well-modedness is a restriction concerning the flow of information in a program.

Definition 2.2

1. A query $r_1(\vec{s}_1, \vec{t}_1), \ldots, r_m(\vec{s}_m, \vec{t}_m)$ is said to be *well-moded* if

$$\mathsf{V}(\vec{s}_i) \subseteq \bigcup_{j=1}^{i-1} \mathsf{V}(\vec{t}_j)$$

   for every $i \in \{1, \ldots, m\}$.

2. A clause $r_0(\vec{t}_0, \vec{s}_{m+1}) \leftarrow r_1(\vec{s}_1, \vec{t}_1), \ldots, r_m(\vec{s}_m, \vec{t}_m)$ is *well-moded* if

$$\mathsf{V}(\vec{s}_i) \subseteq \bigcup_{j=0}^{i-1} \mathsf{V}(\vec{t}_j)$$

   for every $i \in \{1, \ldots, m+1\}$.

3. A logic program $(\mathcal{F}, \mathcal{R}, \mathcal{C})$ is *well-moded* if every clause in $\mathcal{C}$ is well-moded.

Note that if $r_1(\vec{s}_1, \vec{t}_1), \ldots, r_m(\vec{s}_m, \vec{t}_m)$ is a well-moded query, then $\mathsf{V}(\vec{s}_1) = \emptyset$. The concept of well-modedness is important because computing well-moded queries in well-moded programs yields computed answer substitution that are ground (see [1]). Modes play an important rôle in the programming language Mercury [11].

Definition 2.3

1. A query $r_1(\vec{s}_1, \vec{t}_1), \ldots, r_m(\vec{s}_m, \vec{t}_m)$ is said to be *simply moded* if
   (a) it is well-moded,
   (b) the terms $\vec{t}_1, \ldots, \vec{t}_m$ are distinct variables.

2. A clause $r_0(\vec{s}_0, \vec{t}_0) \leftarrow r_1(\vec{s}_1, \vec{t}_1), \ldots, r_m(\vec{s}_m, \vec{t}_m)$ is *simply moded* if
   (a) it is well-moded,
   (b) the terms $\vec{t}_1, \ldots, \vec{t}_m$ are distinct variables.

3. A logic program $(\mathcal{F}, \mathcal{R}, \mathcal{C})$ is *simply moded* if every clause in $\mathcal{C}$ is simply moded.

Note that our terminology differs from the one used in [2]: simply moded in the sense of Definition 2.3 is equivalent to the conjunction of well-modedness and simply modedness in the sense of [2].

In the remainder of this paper we restrict attention to simply moded logic programs and queries. We will make use of the following notion of resolution.

Definition 2.4 Let $(\mathcal{F}, \mathcal{R}, \mathcal{C})$ be a simply moded logic program. A *resolution step* is defined as a pair written as

$$\langle Q; \sigma \rangle \Rightarrow_C \langle Q'; \sigma' \rangle$$

with Q, Q' queries, σ, σ', τ substitutions and C a clause in $\mathcal{C}$ such that:

1. $\mathsf{V}(C) \cap \mathsf{V}(\langle Q; \sigma \rangle) = \emptyset$,
2. $Q = a_1, a_2, \ldots, a_n$,
3. $C = h \leftarrow b_1, \ldots, b_m$,
4. τ is a most general unifier of h and a_1,
5. $Q' = b_1\tau_1, \ldots, b_m\tau_1, a_2\tau_2, \ldots, a_n\tau_2$,
6. $\sigma' = \sigma\tau_2$,

with τ_1 the substitution τ restricted to $\mathsf{V}(h)$ and τ_2 the substitution τ restricted to $\mathsf{V}(a_1)$.

A sequence of resolution steps is called a *resolution sequence*. A resolution sequence is *successful* if it ends in $\langle \Box; \sigma \rangle$, for some substitution σ. We write $\Rightarrow$ instead of $\Rightarrow_C$ if it is clear or irrelevant which clause is used in the resolution step.

A few remarks concerning the previous definition are appropriate. First, note that always the leftmost atom is selected. Second, the expressions that are transformed are pairs consisting of a query and a substitution. In some other definitions of the resolution relation, see for instance [1], the expressions that are transformed are queries. We consider the first option to be more natural; moreover it is closer to actual implementations. Third, we make essential use of the form of simply moded clauses and programs, which also ensures that instead of unification iterated matching can be used [2]. Suppose, using the notation of Definition 2.4, that $a_1 = r(\vec{s}_1, \vec{t}_1)$ and $h = r(\vec{s}, \vec{t})$ are unifiable. Then to start with $\vec{s}_1$ and $\vec{s}$ are unifiable, which means, since $\mathsf{V}(\vec{s}_1) = \emptyset$, that there is a substitution τ_1 such that $\vec{s}\tau_1 = \vec{s}_1$. We call this substitution the *matching substitution* since it matches the input part of the head of a clause with the input part of an atom. Second, since $\vec{t}_1$ are distinct variables, the substitution τ_2 that assigns $\vec{t}\tau_1$ to $\vec{t}_1$ is a unifier of $\vec{t}_1$ and $\vec{t}\tau_1$. Because τ_2 expresses which values are computed for the variables $\vec{t}_1$, we call τ_2 the *computation substitution*.

In the remainder of the paper we will tacitly make use of the following result, which combines results of [1] and [2].

Theorem 2.5 *Let $(\mathcal{F}, \mathcal{R}, \mathcal{C})$ be a simply moded logic program and let Q be a simply moded query. If $\langle Q; \sigma \rangle \Rightarrow \langle Q'; \sigma' \rangle$, then the query Q' is also simply moded.*

3 Conditional Rewriting

In this section we define conditional rewriting systems that will serve as the codomain of the translation defined in Section 4. They differ from the usual ones in several respects. First, conditions are treated on an object-level instead of on a meta-level. Second, the expressions that are transformed are not terms, but environments with conditions. Moreover, the conditions may contain explicit substitutions.

We assume a set $\mathcal{V}$ consisting of infinitely many *variables* written as $x, y, z, \ldots$ and a set $\mathcal{T} = \{\mathsf{T}_i \mid i \geq 0\}$ of symbols for *tupling*. The arity of $\mathsf{T}_i \in \mathcal{T}$ is i. A conditional rewriting system is specified by a pair $(\mathcal{F}, \mathcal{RR})$ consisting of a set of *function symbols* and a set of *conditional rewrite rules*. We assume that $\mathcal{T} \subseteq \mathcal{F}$.

The set of terms is denoted by Terms and terms are denoted by $s, t, \ldots$ as in the previous section. We write T instead of T_0 and s instead of $\mathsf{T}_1(s)$.

We assume further a binary operator $\otimes$ on the set of terms. A *condition* is an element of the monoid $(\mathsf{Terms}, \otimes, \mathsf{T})$, so T is the (left- and right-) neutral element for $\otimes$. We will work modulo the equality relation of the monoid. The set of conditions is denoted by Con and conditions are denoted by $c, d, \ldots$. Note that $\mathsf{Terms} \subseteq \mathsf{Con}$. We will make use of the following syntactic constructs.

Definition 3.1

1. An *environment* is inductively defined as follows.
   (a) $[\,]$ is an environment (the empty environment),
   (b) if e is an environment, and for some $m > 0$, $x_1, \ldots, x_m$ are variables and $s_1, \ldots, s_m$ are terms, then $e[\mathsf{T}_m(x_1, \ldots, x_m) := \mathsf{T}_m(s_1, \ldots, s_m)]$ is an environment.
2. A *condition with explicit substitution* is a pair consisting of a condition c and an environment e, denoted using juxtaposition by ce. Conditions with explicit substitutions are like conditions denoted by $c, c', \ldots$. Terms with explicit substitutions are denoted as terms without explicit substitutions.
3. A *conditional term* is a pair consisting of a term and a condition, where both the term and the condition may contain explicit substitutions. A conditional term is denoted by $s \Leftarrow c$.
4. A *conditional environment* is a pair consisting of an environment and a condition, where the condition may contain explicit substitutions. A conditional environment is denoted by $e \Leftarrow c$.

We write s instead of $s \Leftarrow \mathsf{T}$ and e instead of $e \Leftarrow \mathsf{T}$. Further we adopt the equality $e[x := s\tilde{e}] = e[x := s]\tilde{e}$. The definition of a conditional rewrite rule is as follows.

Definition 3.2 A *conditional rewrite rule* is a pair $l \to (r \Leftarrow c)$ such that

1. l is a non-variable term,
2. $r \Leftarrow c$ is a conditional term,
3. $\mathsf{V}(r \Leftarrow c) \subseteq \mathsf{V}(l)$.

A function symbol that occurs as the head-symbol of the left-hand side of a conditional rewrite rule is said to be a *defined symbol.* A function symbol that is not a defined symbol is a *constructor symbol.* We assume the symbols in $\mathcal{T}$ to be constructor symbols. A *context* is an expression with one hole in it. The result of replacing the hole [ ] in a context $C[\,]$ by an expression X is denoted by $C[X]$.

Definition 3.3 Let $(\mathcal{F}, \mathcal{RR})$ be a conditional rewriting system. The rewrite relation $\to$ on conditional environments is defined as follows. We have

$$(e \Leftarrow c) \;\to_\rho\; (e' \Leftarrow d\sigma \otimes c')$$

if there is a rewrite rule $\rho = l \to (r \Leftarrow d)$ in $\mathcal{RR}$, a substitution σ and a context $C[\,]$ such that

1. $(e \Leftarrow c) = C[l\sigma]$,
2. $(e' \Leftarrow c') = C[r\sigma]$.

The relation $\to$ is defined as the union $\cup_{\rho \in \mathcal{RR}} \to_\rho$.

The transitive closure of a relation $\to$ is denoted by $\to^+$ and the reflexive-transitive closure is denoted by $\twoheadrightarrow$. Note that we consider only one level of conditions. For instance, we have $e \Leftarrow c_1 \otimes c_2$ instead of $(e \Leftarrow c_1) \Leftarrow c_2$. A conditional environment that cannot be rewritten is said to be in *normal form.* Note that a conditional environment without defined symbols is in normal form.

Example 3.4 As an example, we consider the conditional rewriting system defined by the following rewrite rules:

$$\begin{array}{lcl} a & \to & (b \Leftarrow c) \\ c & \to & \mathsf{T} \end{array}$$

We have the following rewrite sequence:

$$[x := a] \to ([x := b] \Leftarrow c) \to [x := b].$$

4 The Translation

In this section we define the translation from simply moded logic programs into conditional rewriting systems.

Statics. The symbols used in a logic program are variables, function symbols with a fixed arity and relation symbols with a fixed mode. These symbols should be translated into symbols used in a conditional rewriting system. We suppose variables to be universal. Both the function symbols and the relation symbols of a logic program are translated into function symbols as follows.

Definition 4.1

1. A variable x is translated into itself.
2. A function symbol $f \in \mathcal{F}$ of arity m is translated into a function symbol f^* of arity m.
3. A relation symbol $r \in \mathcal{R}$ of mode (p, q) is translated into a function symbol r^* of arity p.

The set $\{f^* \mid f \in \mathcal{F}\}$ is denoted by $\mathcal{F}^*$ and the set $\{r^* \mid r \in \mathcal{R}\}$ is denoted by $\mathcal{R}^*$. We will write f and r instead of f^* and r^*. The translation of function symbols is extended homomorphically to the set of atoms. Note that the translation is the identity on the set of terms of a logic program.

Dynamics. The dynamics of a logic program is prescribed by its set of clauses. In a rewriting system, the rewrite rules determine how expressions can be transformed. We will define how to translate simply moded clauses into conditional rewriting rules with explicit substitution as defined in Section 3. First we motivate the use of the three main particularities of the class of rewriting systems used as codomain: tupling, conditions and explicit substitutions.

First we discuss the use of *tupling*. Consider a clause of the form $h \leftarrow$, with an empty body. Such a clause is of the form

$$f(s_1, \ldots, s_p, t_1, \ldots, t_q) \leftarrow$$

with the first p arguments input and the last q arguments output. The natural translation of such a clause is

$$f(s_1, \ldots, s_p) \rightarrow \mathsf{T}_q(t_1, \ldots t_q)$$

with T_q a symbol for tupling of arity q as defined in Section 3. We identify the tuple of arity 0 with 'true'. Hence a simply moded clause $(\mathcal{F}, \mathcal{R})$ will be translated into a rewrite rule over the alphabet $\mathcal{F}^* \cup \mathcal{R}^* \cup \mathcal{T}$ with $\mathcal{T} = \{\mathsf{T}_i \mid i \geq 0\}$ as defined in Section 2.

Second we discuss the use of *conditions*. As an example we consider the following logic program:

$$\begin{array}{rcl} even(0) & \leftarrow & \\ odd(s(x)) & \leftarrow & even(x) \end{array}$$

This program is translated into the following conditional rewriting system:

$$\begin{array}{rcll} even(0) & \to & \mathsf{T} & \\ odd(s(x)) & \to & \mathsf{T} & \Leftarrow \quad even(x) \end{array}$$

with modes $even, odd$: $\downarrow$. Intuitively, the resolution sequence

$$\langle odd(s(0)); \epsilon \rangle \Rightarrow \langle even(0); \epsilon \rangle \Rightarrow \langle \Box; \epsilon \rangle$$

corresponds to the rewrite sequence

$$odd(s(0)) \to (\mathsf{T} \Leftarrow even(0)) \to \mathsf{T}.$$

Note that the use of conditions on an object level is essential, if we want every resolution step to correspond to at least one rewriting step. Using 'normal' conditional rewriting, the second rewrite step would take place on a meta-level and would hence not be observable.

Third we discuss the use of *explicit substitutions*. To make the discussion more concrete, consider the logic program for addition of natural numbers:

$$\begin{array}{rcl} add(0, x, x) & \leftarrow & \\ add(s(x), y, s(z)) & \leftarrow & add(x, y, z) \end{array}$$

with mode add: $\downarrow \times \downarrow \times \uparrow$. A naive but elegant way of translating this logic program yields the following conditional rewriting system:

$$\begin{array}{rcll} add(0, x) & \to & x & \\ add(s(x), y) & \to & s(z) & \Leftarrow \quad add(x, y) \twoheadrightarrow z \end{array}$$

Note that this conditional rewriting system is not in the format defined in Section 2; it is in fact a conditional rewriting system with so-called *extra variables*, since the variable z in the last rewrite rule does not appear in the left-hand side. This is, ignoring some notational differences, the translation used by Ganzinger and Waldmann in [6]. Although for the purpose of that paper, which is to provide a method to prove termination of logic programs, this translation is satisfactory, for the purposes of the present paper it is not, for the following reasons. First, every successful resolution sequence starting in a simply moded query is translated into a rewrite sequence consisting of one rewrite step, so the translation does not give any indication of the cost of a computation expressed in the number of transformation steps. This is the case since resolution steps at object-level are translated into rewriting steps at meta-level. Second, we think that the conditional part of a rewriting

rule should be used to check a condition and not to calculate the value of a variable. Reconsidering the second rewrite rule reveals that the problems mentioned above can be solved by turning the condition of the second rewrite rule into a substitution, yielding the following rewrite rule:

$$add(s(x), y) \rightarrow s(z)[z := add(x, y)].$$

If the condition is evaluated in the usual way, then this rewrite rule takes the following form:

$$add(s(x), y) \rightarrow s(add(x, y)).$$

In this way the logic program for addition is translated into the usual rewriting system for addition. However, for the moment we choose not to evaluate the conditions in right-hand sides of rewrite rules for the following reason. Translating the clause

$$f(x, a) \quad \leftarrow \quad f(a, y)$$

with mode $f: \downarrow \times \uparrow$ into a rewrite rule with explicit conditions yields

$$f(x) \quad \rightarrow \quad a[y := f(a)].$$

If the condition is evaluated, the rule takes the form

$$f(x) \quad \rightarrow \quad a.$$

In that way a non-terminating logic program is translated into a terminating rewriting system. In the translation with explicit substitution, the infinite resolution sequence is translated into an infinite rewrite sequence in which almost all rewrite steps take place in 'garbage'. Finally, the order in which the explicit substitutions occur in the translation of a clause is determined by the flow of information, which is in simply moded clauses from left to right. See for an illustration Example 4.3.

This discussion motivates the following definition of the translation of a moded (not necessarily simply moded) clause.

Definition 4.2 Let

$$C = h \leftarrow b_1, \ldots, b_m$$

with $h = r(\vec{s}, \vec{t})$ and $m \geq 0$ be a moded clause over $(\mathcal{F}, \mathcal{R})$ with distinct variables in the output positions of the body. Define for every $p \in \{1, \ldots, m+1\}$: $C_p = h \leftarrow b_p, \ldots, b_m$.

1. The translation of C_p, denoted by C_p^*, is defined by induction on $m + 1 - p$.

   (a) Suppose that $p = m + 1$. The translation of C_p is defined as follows:

   $$C_p^* = r(\vec{s}) \rightarrow \mathsf{T}_i(\vec{t}) \quad \text{with } i = |\vec{t}|.$$

(b) Suppose that $1 \leq p < m+1$ and let $C^*_{p+1} = l_{p+1} \to (r_{p+1} \Leftarrow c_{p+1})$.

i. If $b_p = r_p(\vec{s}_p, \vec{t}_p)$ with $|\vec{t}_p| = i > 0$, then

$$C^*_p = l_{p+1} \to r_{p+1}[\mathsf{T}_i(\vec{t}_p) := r_p(\vec{s}_p)] \Leftarrow c_{p+1}[\mathsf{T}_i(\vec{t}_p) := r_p(\vec{s}_p)].$$

ii. If $b_p = r_p(\vec{s}_p)$, then

$$C^*_p = l_{p+1} \to (r_{p+1} \Leftarrow r_p(\vec{s}_p) \otimes c_{p+1}).$$

2. The translation of C, denoted by C^*, is defined as follows: $C^* = C^*_1$.

Note that we have $l_p = r(\vec{s})$ for every $p \in \{1, \ldots, m+1\}$ in the previous definition. Another observation is that the relation symbols of a logic program are translated into defined symbols, and its function symbols are translated into constructor symbols.

Example 4.3 Consider the simply moded clause

$$C = f(x, z) \leftarrow g(x, y), h(y), g'(y, z)$$

with modes f, g, g': $\downarrow \times \uparrow$ and h: $\downarrow$. Following Definition 4.2, we find the following:

$C_4 = f(x, z) \leftarrow$	$C^*_4 = f(x) \to z$
$C_3 = f(x, z) \leftarrow g'(y, z)$	$C^*_3 = f(x) \to z[z := g'(y)]$
$C_2 = f(x, z) \leftarrow h(y), g'(y, z)$	$C^*_2 = f(x) \to z[z := g'(y)] \Leftarrow h(y)$
$C_1 = f(x, z) \leftarrow g(x, y), h(y), g'(y, z)$	$C^*_1 = f(x) \to z[z := g'(y)][y := g(x)] \Leftarrow h(y)[y := g(x)]$

The following result states that the translation of a simply moded clause is a conditional rewrite rule.

Proposition 4.4 *Let C be a simply moded clause. Then C^* is a conditional rewrite rule.*

Expressions. In a resolution step, a pair consisting of a query and a substitution is transformed into another pair consisting of a query and a substitution. We now define the translation of such pairs into conditional environments. First, a substitution is translated into a conditional environment as follows.

Definition 4.5 Let $\sigma = \{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$ be a substitution. Its translation, denoted by σ^*, is defined as follows:

$$\sigma^* = [x_1 := s_1] \ldots [x_m := s_m].$$

Note that the translation of a substitution is a conditional environment in normal form, since a term in a logic program is translated into a term not containing defined symbols.

A moded query is also translated into a conditional environment.

Definition 4.6 Let $Q = a_1, \ldots, a_m$ with $m \geq 0$ be a moded query over $(\mathcal{F}, \mathcal{R})$ with distinct variables in the output positions. Define for every $p \in \{1, \ldots, m+1\}$: $Q_p = a_p, \ldots, a_m$.

1. The translation of Q_p, denoted by Q_p^*, is defined by induction on $m + 1 - p$ as follows.
   (a) Suppose that $p = m + 1$. Then:
   $$Q_p^* = [\,].$$
   (b) Suppose that $1 \leq p < m + 1$. Suppose that $Q_{p+1}^* = e_{p+1} \Leftarrow c_{p+1}$.
      i. If $a_p = r_p(\vec{s}_p, \vec{t}_p)$ with $|\vec{t}_p| = i > 0$, then
      $$Q_p^* = e_{p+1}[\mathsf{T}_i(\vec{t}_p) := r_p(\vec{s}_p)] \Leftarrow c_{p+1}[\mathsf{T}_i(\vec{t}_p) := r_p(\vec{s}_p)].$$
      ii. If $a_p = r_p(\vec{s}_p)$, then
      $$Q_p^* = e_{p+1} \Leftarrow r_p(\vec{s}_p) \otimes c_{p+1}.$$
2. The translation of Q, denoted by Q^*, is defined as follows: $Q^* = Q_1^*$.

Using the previous two definitions, the translation of a pair consisting of a query and a substitution is defined as follows.

Definition 4.7 Let Q be a query with translation $Q^* = e \Leftarrow c$ and let σ be an substitution with translation $\sigma^* = \tilde{e}$. Then: $\langle Q; \sigma \rangle^* = \tilde{e}\, e \Leftarrow c$.

One might ask whether the order in which the environments $\tilde{e}$ and e are concatenated in the previous definition is essential. This is indeed the case. Consider for instance the following (simply moded) clause

$$C = f(x, g(y)) \leftarrow h(x, y)$$

with modes f, h: $\downarrow \times \uparrow$. Its translation is

$$C^* = f(x) \rightarrow g(y)[y := h(x)].$$

We have the resolution step

$$\langle f(a, z); \epsilon \rangle \Rightarrow_C \langle h(a, y); \{z \mapsto g(y)\} \rangle.$$

Using Definition 4.7, this resolution step is translated into the following rewrite step:

$$[z := f(a)] \rightarrow_{C^*} [z := g(y)][y := h(a)].$$

Note that the translation is not correct if the order in which $\{z \mapsto g(y)\}^*$ and the environment part of $h(a, y)^*$ are concatenated.

The Main Result. The main result is that using the translation of the present paper, a resolution step using a clause C corresponds to a rewrite sequence consisting of at least one step using the translation of C. In a diagram:

$$\begin{array}{ccc} \langle Q;\sigma\rangle & \Rightarrow_C & \langle Q';\sigma'\rangle \\ \downarrow & & \downarrow \\ \langle Q;\sigma\rangle^* & \rightarrow^+_{C^*} & \langle Q';\sigma'\rangle^* \end{array}$$

This is expressed in the following theorem, which is proved by induction on the translation of the clause C. In the proof we work modulo three natural equations concerning conditional environments.

Theorem 4.8 *If* $\langle Q;\sigma\rangle \Rightarrow_C \langle Q';\sigma'\rangle$, *then* $\langle Q;\sigma\rangle^* \rightarrow^+_{C^*} \langle Q';\sigma'\rangle^*$.

A corollary of Theorem 4.8 is that termination of a logic program is implied by termination of its translation. Since the translation of $\langle \Box;\sigma\rangle$ is a conditional environment in normal form, we have moreover the following result.

Theorem 4.9 *A successful resolution sequence is translated into a rewrite sequence ending in a conditional environment in normal form.*

A resolution sequence that ends with failure might be translated into an infinite rewrite sequence. However, it is easy to imagine a marking device that indicates which part of the translation of a query Q corresponds to the left-most atom of Q. In this way, termination with failure can also be detected in the translation of a logic program.

Finally, the translation of the clauses of a logic program can be put into a more readable form by evaluating the explicit substitutions in the usual way. Then the statement of Theorem 4.8 doesn't hold anymore; instead we obtain the weaker result that $\langle Q;\sigma\rangle \Rightarrow_C \langle Q';\sigma'\rangle$ implies $\langle Q;\sigma\rangle^* \twoheadrightarrow_{C^*} \langle Q';\sigma'\rangle^*$.

Example. As an example we consider quicksort. Using the notational conventions of Prolog, this program consists of the following clauses:

```
q([], []).
q([X | Xs], Ys) :-
  p(X, Xs, Ls, Bs),
  q(Ls, Ls'), q(Bs, Bs'),
  app(Ls', [X | Bs'], Ys).
p(X, [], [], []).
p(X, [Y | Ys], [Y | Ls], Bs) :-
  X >= Y, p(X, Ys, Ls, Bs).
p(X, [Y | Ys], Ls, [Y | Bs]) :-
```

```
  X < Y, p(X, Ys, Ls, Bs).
app([], Xs, Xs).
app([X | Xs], Ys, [X | Zs]) :-
  app(Xs, Ys, Zs).
```

with modes q: $\downarrow \times \uparrow$, p: $\downarrow \times \downarrow \times \uparrow \times \uparrow$, app: $\downarrow \times \downarrow \times \uparrow$. Translating this program yields a conditional rewriting system. If we evaluate some of the explicit conditions in this conditional rewriting system, and use the notational conventions of Gofer, then we obtain the following functional program.

```
q([])           =  []
q(x:xs)         =  app(q(ls), (x:q(bs)))
                   where (ls, bs) = p(x, xs)
p(x, [])        =  ([], [])
p(x, (y:ys))    =  ((y:ls), bs) if x >= y
                =  (ls, (y:bs)) if x < y
                   where (ls, bs) = p(x, ys)
app([], xs)     =  xs
app((x:xs), ys) =  (x:app(xs, ys))
```

5 Concluding Remarks

Related Work. The first to study a translation from logic programs into functional programs is Reddy [10]. The logic programs that are translated in this paper are not subject to any restriction; as a consequence, the translation yields functional programs that are in fact only notationally different from the logic programs they are derived from.

In [7], Krishna Rao, Kapur and Shyamasundar define a translation from well-moded logic programs into (unconditional) term rewriting systems. The main result of the paper is that termination of a logic program is implied by termination of its translation. The definition of the translation is quite complex and different from the translation defined in the present paper. For instance, the translation of a relation symbol with more than one output position is not unique but introduces for every output position a new function symbol.

The paper [6] by Ganzinger and Waldmann is concerned with termination of logic programs. Well-moded logic programs are translated into conditional rewriting systems in the naive but elegant way as discussed in Section 4, modulo some differences in notation. An earlier result by Ganzinger states that conditional rewriting systems that are *quasi-reductive* are terminating. The main result of [6] is that a logic program is terminating if its translation into a conditional rewriting system is quasi-reductive. This line of research is continued by Avenhaus and Loría-Sáenz, who present in [4] a critical pair criterion for quasi-reductive conditional rewriting systems. A corollary of the

results presented in [6] and [4] is that a well-moded logic program terminates in a unique result, if its translation (in the naive way) is quasi-reductive and satisfies the critical pair criterion of [4].

Marchiori proposes in [8] a translation from the class of simply moded logic programs into (unconditional) term rewriting systems. This translation is very complex and differs quite a lot from the one of the present paper.

Arts and Zantema present in [3] an algorithm to translate logic programs into constructor systems. Moreover, they present a technique to prove termination of constructor systems that is particularly suitable to prove termination of the translation of logic programs, using their translation.

Conclusions. The translation proposed in the present paper is very intuitive and given the correctness result moreover suitable to be used as a basis for an implementation of a subclass of logic programs via functional programs.

The format of rewriting presented in Section 3 is tailor-made for the translation; nevertheless some of its features, in particular the use of conditions at an object-level, are also interesting purely from a rewriting point of view.

We conjecture that the main result can also be proved for a larger class of logic programs obtained by relaxing the restriction of well-modedness.

Acknowledgements

I gratefully acknowledge helpful and motivating discussions with Krzysztof Apt. I further wish to thank Gilles Barthe, Sandro Etalle, Pieter Hartel, Aart Middeldorp, Vincent van Oostrom, Wim Vree and Mark Wielaard for valuable remarks.

This research was supported by NWO/SION project number 612-33-003, entitled 'Parallel declarative programming: transforming logic programs to lazy functional programs'.

References

[1] K.R. Apt. *From Logic Programming to Prolog.* Prentice-Hall, 1997.

[2] K.R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS '93)*, pages 1–19, Berlin, Germany, 1993. Springer Verlag.

[3] T. Arts and H. Zantema. Termination of logic programs using semantic unificition. In M. Proietti, editor, *Proceedings of the 5th International Workshop on Logic Program Synthesis and Transformation (LOPSTR*

'95), volume 1048 of *Lecture Notes in Computer Science*, pages 219–233, Utrecht, The Netherlands, September 1995. Springer Verlag.

[4] J. Avenhaus and C. Loría-Sáenz. On conditional rewrite systems with extra variables and deterministic logic programs. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning (LPAR '94)*, volume 822 of *Lecture Notes in Artificial Intelligence*, pages 215–229, Kiev, Ukraine, July 1994. Springer Verlag.

[5] P. Dembiński and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, Boston, USA, 1985.

[6] H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In M. Rusinowitch and J.L. Rémy, editors, *Proceedings of the third international workshop on conditional term rewriting systems (CTRS '92)*, pages 430–437, Pont-à-Mousson, July 1992.

[7] M.R.K. Krishna Rao, D. Kapur, and R.K. Shyamasundar. A transformation methodology for proving termination of logic programs. In E. Börger, G. Jäger, H. Kleine Büning, and M.M. Richter, editors, *Proceedings of the 5th Workshop on Computer Science Logic (CSL '91)*, pages 213–226, Berne, Switzerland, October 1991. Springer Verlag.

[8] M. Marchiori. Logic programs as term rewriting systems. In G. Levi and M. Rodríguez-Artalejo, editors, *Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP '94)*, pages 223–241, Madrid, Spain, September 1994.

[9] C.S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report DAI Research Paper 163, University of Edinburgh, 1981.

[10] U.S. Reddy. Transformation of logic programs into functional programs. In *Proceedings of the Symposium on Logic Programming*, pages 187–196, Atlantic City, New Jersey, USA, February 1984. IEEE Computer Society, Silver Spring, MD.

[11] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.