Declarative Compilation for Constraint Logic Programming

Emilio Jesús Gallego Arias[1], James Lipton[2](✉), and Julio Mariño[3]

[1] University of Pennsylvania, Philadelphia, USA
emilioga@cis.upenn.edu
[2] Wesleyan University, Middletown, USA
jlipton@wesleyan.edu
[3] Universidad Politécnica de Madrid, Madrid, Spain
jmarino@fi.upm.es

Abstract. We present a new declarative compilation of logic programs with constraints into variable-free relational theories which are then executed by rewriting. This translation provides an algebraic formulation of the abstract syntax of logic programs. Management of logic variables, unification, and renaming apart is completely elided in favor of algebraic manipulation of variable-free relation expressions. We prove the translation is sound, and the rewriting system complete with respect to traditional SLD semantics.

Keywords: Logic programming · Constraint programming · Relation algebra · Rewriting · Semantics

1 Introduction

Logic programming is a paradigm based on proof search and directly programming with logical theories. This is done to achieve *declarative transparency*: guaranteeing that execution respects the mathematical meaning of the program. The power that such a paradigm offers comes at a cost for formal language research and implementation. Management of logic variables, unification, renaming variables apart and proof search are cumbersome to handle formally. Consequently, it is often the case that the formal definition of these aspects is left outside the semantics of programs, complicating reasoning about them and the introduction of new declarative features.

We address this problem here by proposing a new compilation framework – based on ideas of Tarski [21] and Freyd [9] – that encodes logic programming syntax into a variable-free algebraic formalism: relation algebra. Relation algebras are pure equational theories of structures containing the operations of composition, intersection and convolution. An important class of relation algebras is the so-called *distributive relation algebras with quasi-projections*, which also incorporate union and projections.

We present the translation of constraint logic programs to such algebras in 3 steps. First, for a CLP program P with signature Σ, we define its associated

M. Proietti and H. Seki (Eds.): LOPSTR 2014, LNCS 8981, pp. 299–316, 2015.
DOI: 10.1007/978-3-319-17822-6_17

relation algebra $\mathbf{QRA}_\Sigma$, which provides both the target object language for program translation and formal axiomatization of constraints and logic variables. Second, we introduce a constraint compilation procedure that maps constraints to variable-free relation terms in $\mathbf{QRA}_\Sigma$. Third, a program translation procedure compiles constraint logic programs to an equational theory over $\mathbf{QRA}_\Sigma$.

The *key feature* of the semantics and translation is its variable-free nature. Programs that contain logical variables are represented as ground terms in our setting, thus all reasoning and execution is reduced to algebraic equality, allowing the use of rewriting. The resulting system is sound and complete with respect to SLD resolution. Our compilation provides a solution to the following problems:

– Underspecification of abstract syntax and logic variable management in logic programs: solved by the the inclusion of metalogical operations directly into the compilation process.
– Interdependence of compilation and execution strategies: solved by making target code completely orthogonal to execution.
– Lack of transparency in compilation (for subsequent optimization and abstract interpretation): solved by making target code a low-level yet *fully declarative* translation of the original program.

Variable Elimination and Relation Composition. We illustrate the spirit of translation, and in particular the variable elimination procedure, by considering a simple case, namely the transitive closure of a graph:

```
edge(a,b).          connected(X,X).
edge(b,c).          connected(X,Y) :- edge(X,Z), connected(Z,Y).
edge(a,e).
edge(e,f).
```

In this carefully chosen example, the elimination of variables and the translation to binary relation symbols is immediate:

$$\begin{aligned}\mathbf{edge} &= (a,b) \cup (b,c) \cup (a,e) \cup (a,e) \cup (e,f)\\ \mathbf{connected} &= \mathbf{id} \cup \mathbf{edge};\mathbf{connected}\end{aligned}$$

The key feature of the resulting term is the composition **edge**; **connected**. The logical variable Z is eliminated by the composition of relations allowing the use of variable free object code. A query $\mathbf{connected}(a,X)$ is then modeled by the relation $\mathbf{connected} \cap (a,a)\mathbf{1}$ where $\mathbf{1}$ is the (maximal) universal relation. Computation can proceed by rewriting the query using a suitable orientation of the relation algebra equations and unfolding pertinent recursive definitions.

Handling actual arbitrary constraint logic programs is more involved. First, we use sequences and projection relations to handle predicates involving an arbitrary number of arguments and an unbounded number of logic variables; second, we formalize constraints in a relational way.

Projections and permutations algebraically encode all the operations of logical variables – disjunctive and conjunctive clauses are handled with the help of the standard relational operators $\cap$, $\cup$.

Constraint Logic Programming Conventions. We refer the reader to [16] for basic definitions of logic programming over Horn clauses, and [12] for background on the syntax and semantics of constraint logic programming. In this paper we fix a signature Σ, a set of terms $\mathcal{T}_\Sigma(\mathcal{X})$, and a subset $\mathcal{C}$ of all first-order formulas over Σ closed under conjunction and existential quantification to be the set of *constraint formulas* as well as a Σ-structure $\mathcal{D}$, called the *constraint domain*. Constraint logic programs are sets of Horn clauses. We use vector notation extensively in the paper, to abbreviate Horn clauses with constraints $p \leftarrow q_1, \dots, q_n$, where p is an atomic formula and q_i may be an atomic formula or a constraint. For instance, in our vector notation, a clause is written $p(\boldsymbol{t}[\boldsymbol{x}]) \leftarrow \boldsymbol{q}(\boldsymbol{u}[\boldsymbol{x}, \boldsymbol{y}])$, where the boldface symbols indicate vectors of variables $\boldsymbol{x}, \boldsymbol{y}$, terms $\boldsymbol{t}, \boldsymbol{u}$ (depending on variables $\boldsymbol{x}$, etc...) and predicates $\boldsymbol{q}$ (depending on terms $\boldsymbol{u}$).

2 Relation Algebras and Signatures

In this section, we define $\mathbf{QRA}_\Sigma$, a relation algebra in the style of [9,21] formalizing a CLP signature Σ and a constraint domain $\mathcal{D}$. We define its language, its equational theory and semantics.

2.1 Relational Language and Theory

The relation language R_Σ is built from a set $\mathsf{R}_\mathcal{C}$ of relation constants for constant symbols a set $\mathsf{R}_\mathcal{F}$ of relation constants for function symbols from Σ, and a set of relation constants for primitive predicates $\mathsf{R}_{\mathcal{CP}}$, as well as a fixed set of relation constants and operators detailed below. Let us begin with $\mathsf{R}_\mathcal{C}$. Each constant symbol $a \in \mathcal{C}_\Sigma$ defines a constant symbol $(a, a) \in \mathsf{R}_\mathcal{C}$, each function symbol $f \in \mathcal{F}_\Sigma$ defines a constant symbol R_f in $\mathsf{R}_\mathcal{F}$. Each predicate symbol $r \in \mathcal{CP}_\Sigma$ defines a constant symbol r in $\mathsf{R}_{\mathcal{CP}}$. We write R_Σ for the full relation language:

$$\mathsf{R}_\mathcal{C} = \{(a,a) \mid a \in \mathcal{C}_\Sigma\} \quad \mathsf{R}_\mathcal{F} \;=\; \{\mathsf{R}_f \mid f \in \mathcal{F}_\Sigma,\} \quad \mathsf{R}_{\mathcal{CP}} \;=\; \{\mathsf{r} \mid r \in \mathcal{CP}_\Sigma\}$$
$$\mathsf{R}_{atom} ::= \mathsf{R}_\mathcal{C} \mid \mathsf{R}_\mathcal{F} \mid \mathsf{R}_{\mathcal{CP}} \mid id \mid di \mid \mathbf{1} \mid \mathbf{0} \mid hd \mid tl$$
$$\mathsf{R}_\Sigma \quad ::= \mathsf{R}_{atom} \mid \mathsf{R}_\Sigma{}^\circ \mid \mathsf{R}_\Sigma \cup \mathsf{R}_\Sigma \mid \mathsf{R}_\Sigma \cap \mathsf{R}_\Sigma \mid \mathsf{R}_\Sigma \mathsf{R}_\Sigma$$

The constants $\mathbf{1}, \mathbf{0}, id, di$ respectively denote the universal relation (whose standard semantics is the set of all ordered pairs on a certain set), the empty relation, the identity (diagonal) relation, and identity's complement. Juxtaposition RR represents relation composition (often written R;R) and R° is the inverse of R. We write hd and tl for the head and tail relations. The projection of an n-tuple onto its i-th element is written P_i and defined as $P_1 = hd, P_2 = tl; hd, \dots, P_n = tl^{n-1}; hd$.

$\mathbf{QRA}_\Sigma$ (Fig. 1) is the standard theory of distributive relation algebras, plus Tarski's quasiprojections [21], and equations axiomatizing the new relations of R_Σ. Note that products and their projections are axiomatized in a relational, variable-free manner.

$$R \cap R = R \qquad R \cap S = S \cap R \qquad R \cap (S \cap T) = (R \cap S) \cap T$$
$$R \cup R = R \qquad R \cup S = S \cup R \qquad R \cup (S \cup T) = (R \cup S) \cup T$$
$$R\,id = R \qquad R\mathbf{0} = \mathbf{0} \qquad \mathbf{0} \subseteq R \subseteq \mathbf{1}$$
$$R \cup (S \cap R) = R = (R \cup S) \cap R$$
$$R(S \cup T) = RS \cup RT \qquad (S \cup T)R = SR \cup TR$$
$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$
$$(R \cup S)^\circ = R^\circ \cup S^\circ \qquad (R \cap S)^\circ = S^\circ \cap R^\circ$$
$$R^{\circ\circ} = R \qquad (RS)^\circ = S^\circ R^\circ$$
$$R(S \cap T) \subseteq RS \cap RT \qquad RS \cap T \subseteq (R \cap TS^\circ)S$$
$$id \cup di = \mathbf{1} \qquad id \cap di = \mathbf{0}$$

$$hd(hd)^\circ \cap tl(tl)^\circ \subseteq id \qquad (hd)^\circ hd \subseteq id,\ (tl)^\circ tl \subseteq id \qquad (hd)^\circ tl = \mathbf{1}$$
$$\mathbf{1}(c,c)\mathbf{1} = \mathbf{1} \qquad (c,c) \subseteq id$$

Fig. 1. $\mathbf{QRA}_\Sigma$

2.2 Semantics

Let Σ be a constraint signature and $\mathcal{D}$ a Σ-structure. Write $t^{\mathcal{D}}$ for the interpretation of a term $t \in \mathcal{T}_\Sigma$. We define $\mathcal{D}^\dagger$ to be the union of $\mathcal{D}^0 = \{\langle\rangle\}$ (the empty sequence), $\mathcal{D}$ and $\mathcal{D}$-finite products, for example: $\mathcal{D}^2, \mathcal{D}^2 \times \mathcal{D}, \mathcal{D} \times \mathcal{D}^2, \ldots$ We write $\langle a_1, \ldots, a_n\rangle$ for members of the n-fold product associating to the right, that is to say, $\langle a_1, \langle a_2, \ldots, \langle a_{n-1}, a_n\rangle \cdots\rangle\rangle$. Furthermore, we assume right-association of products when parentheses are absent. Note that the 1 element sequence does not exist in the domain, so we write $\langle a\rangle$ for a as a convenience.

Let $\mathsf{R}_{\mathcal{D}} = \mathcal{D}^\dagger \times \mathcal{D}^\dagger$. We make the power set of $\mathsf{R}_{\mathcal{D}}$ into a model of the relation calculus by interpreting atomic relation terms in a certain canonical way, and the operators in their standard set-theoretic interpretation. We interpret *hd* and *tl* as projections in the model.

Definition 1. *Given a structure $\mathcal{D}$ a relational $\mathcal{D}$-**interpretation** is a mapping $[\![\_]\!]^{\mathcal{D}^\dagger}$ of relational terms into $\mathsf{R}_{\mathcal{D}}$ satisfying the identities in Fig. 2. The function α used in this table and elsewhere in this paper refers to the arity of its argument, whether a relation or function symbol from the underlying signature.*

Theorem 1. *Equational reasoning in $\mathbf{QRA}_\Sigma$ is sound for any interpretation:*

$$\mathbf{QRA}_\Sigma \vdash R = S \Longrightarrow [\![R]\!]^{\mathcal{D}^\dagger} = [\![S]\!]^{\mathcal{D}^\dagger}$$

3 Program Translation

We define constraint and program translation to relation terms. To this end, we define a function $\dot{K}$ from constraint formulas with – possibly free – logic variables to a variable-free relational term. $\dot{K}$ is the core of the variable elimination mechanism and will appear throughout the rest of the paper.

$$\begin{array}{llll}
[\![\mathbf{1}]\!]^{\mathcal{D}^\dagger} & = \mathsf{R}_A & [\![tl]\!]^{\mathcal{D}^\dagger} & = \{(\langle a,b\rangle, b) \mid a,b \in \mathcal{D}^\dagger\} \\
[\![\mathbf{0}]\!]^{\mathcal{D}^\dagger} & = \emptyset & [\![R^\circ]\!]^{\mathcal{D}^\dagger} & = ([\![R]\!]^{\mathcal{D}^\dagger})^\circ \\
[\![id]\!]^{\mathcal{D}^\dagger} & = \{(u,u) \mid u \in \mathcal{D}^\dagger\} & [\![R \cup S]\!]^{\mathcal{D}^\dagger} & = [\![R]\!]^{\mathcal{D}^\dagger} \cup [\![S]\!]^{\mathcal{D}^\dagger} \\
[\![di]\!]^{\mathcal{D}^\dagger} & = \{(u,v) \mid u \neq v\} & [\![R \cap S]\!]^{\mathcal{D}^\dagger} & = [\![R]\!]^{\mathcal{D}^\dagger} \cap [\![S]\!]^{\mathcal{D}^\dagger} \\
[\![hd]\!]^{\mathcal{D}^\dagger} & = \{(\langle a,b\rangle, a) \mid a,b \in \mathcal{D}^\dagger\} & [\![(c,c)]\!]^{\mathcal{D}^\dagger} & = \{(c^{\mathcal{D}}, c^{\mathcal{D}})\} \\
[\![RS]\!]^{\mathcal{D}^\dagger} & = [\![R]\!]^{\mathcal{D}^\dagger} ; [\![S]\!]^{\mathcal{D}^\dagger} & & \\
[\![\mathsf{R}_f]\!]^{\mathcal{D}^\dagger} & \multicolumn{3}{l}{= \{(x, \boldsymbol{yu}) \mid x = f^{\mathcal{D}}(a_1,\ldots,a_n) \wedge \ \boldsymbol{y} = \langle a_1,\ldots,a_n\rangle, a_i \in \mathcal{D}, \boldsymbol{u} \in \mathcal{D}^\dagger, n = \alpha(f)\}} \\
[\![\mathsf{r}]\!]^{\mathcal{D}^\dagger} & \multicolumn{3}{l}{= \{(\boldsymbol{xu}, \boldsymbol{xu}) \mid \boldsymbol{x} = \langle a_1,\ldots,a_n\rangle \wedge r^{\mathcal{D}}(a_1,\ldots,a_n), a_i \in \mathcal{D}, \boldsymbol{u} \in \mathcal{D}^\dagger, n = \alpha(r)\}}
\end{array}$$

Fig. 2. Standard interpretation of binary relations.

The reader should keep in mind that there are two kinds of predicate symbols in a constraint logic program: *constraint predicates* r which are translated by the function $\dot{K}$ above to relation terms r, and *defined* or program predicates.

We translate defined predicates – and CLP programs – to equations $\overline{p} \mathrel{\mathring{=}} R$, where $\overline{p}$ will be drawn from a set of definitional variables standing for program predicate names p, and R is a relation term. The set of definitional equations can be both seen as an executable specification, by understanding it in terms of the rewriting rules given in this paper; or as a declarative one, by unfolding the definitions and using the standard set-theoretic interpretation of binary relations.

3.1 Constraint Translation

We fix a canonical list $x_1, \ldots, x_n$ of variables occurring in all terms, so as to translate them to variable-free relations in a systematic way. There is no loss of generality as later, we transform programs into this canonical form.

Definition 2 (Term Translation). *Define a translation function $K : \mathcal{T}_\Sigma(\mathcal{X}) \to \mathsf{R}_\Sigma$ from first-order terms to relation expressions as follows:*

$$\begin{array}{ll}
K(c) & = (c,c)\mathbf{1} \\
K(x_i) & = P_i^\circ \\
K(f(t_1,\ldots,t_n)) & = \mathsf{R}_f ; \bigcap_{i \le n} P_i ; K(t_i)
\end{array}$$

This translation is extended to vectors of terms as follows $K(\langle t_1,\ldots,t_n\rangle) = \bigcap_{i\le n} P_i ; K(t_i)$.

The semantics of the relational translation of a term is the set of all of the instances of that term, paired with the corresponding instances of its variables. For instance, the term $x_1 + s(s(x_2))$ is translated to the relation $+ ; (P_1 ; P_1^\circ \cap P_2 ; \mathsf{s} ; \mathsf{s} ; P_2^\circ)$.

Lemma 1. *Let $t[\boldsymbol{x}]$ be a term of $\mathcal{T}_\Sigma(\mathcal{X})$ whose free variables are among those in the sequence $\boldsymbol{x} = x_1, \ldots, x_m$. Then, for any sequences $\boldsymbol{a} = a_1, \ldots, a_m \in \mathcal{D}^\dagger$, $\boldsymbol{u} \in \mathcal{D}^\dagger$ and any $b \in \mathcal{D}$ we have*

$$(b, \boldsymbol{au}) \in [\![K(t[\boldsymbol{x}])]\!]^{\mathcal{D}^\dagger} \iff b = t^{\mathcal{D}}[\boldsymbol{a}/\boldsymbol{x}]$$

We will translate constraints over m variables to partially coreflexive relations over the elements that satisfy them. A binary relation R is *coreflexive* if it is contained in the identity relation, and it is *i-coreflexive* if its i-th projection is contained in the *identity relation*: $P_i^\circ; R; P_i \subseteq id$. Thus, for a variable x_i *free* in a constraint, the translation will be i-coreflexive.

We now formally define two *partial identity relation expressions* I_m, Q_i for the translation of existentially quantified formulas, in such a way that if a constraint $\varphi[\boldsymbol{x}]$ over m variables is translated to an m-coreflexive relation, the formula $\exists x_i.\ \varphi[\boldsymbol{x}]$ corresponds to a coreflexive relation in all the positions but the i-th one, as x_i is no longer free. In this sense Q_i may be seen as a *hiding* relation.

Definition 3. *The partial identity relation expressions I_m, Q_i for $m, i > 0$ are defined as:*

$$I_m := \bigcap_{1 \le i \le m} P_i (P_i)^\circ \qquad Q_i = I_{i-1} \cap J_{i+1} \qquad J_i = tl^i; (tl^\circ)^i$$

I_m is the identity on sequences up to the first m elements. Q_i is the identity on all but the i-th element, with the i-th position relating arbitrary pairs of elements.

Definition 4 (Constraint Translation). *The $\dot{K} : \mathcal{L}_{\mathcal{D}} \to \mathsf{R}_\Sigma$ translation function for constraint formulas is:*

$$\begin{array}{ll} \dot{K}(p(t_1, \ldots, t_n)) & = (\bigcap_{i \le n} K(t_i)^\circ; P_i^\circ); \mathsf{p}; (\bigcap_{i \le n} P_i; K(t_i)) \\ \dot{K}(\varphi \wedge \theta) & = \dot{K}(\varphi) \cap \dot{K}(\theta) \\ \dot{K}(\exists x_i.\ \varphi) & = Q_i; \dot{K}(\varphi); Q_i \end{array}$$

As an example, the translation of the constraint $\exists x_1, x_2. s(x_1) \le x_2$ is

$$Q_1; Q_2; (P_1^\circ; \mathsf{s}^\circ; P_1 \cap P_2^\circ; P_2); \le; (P_1; \mathsf{s}; P_1^\circ \cap P_2; P_2^\circ); Q_1; Q_2$$

Lemma 2. *Let $\varphi[\boldsymbol{x}]$ be a constraint formula with free variables among $\boldsymbol{x} = x_1, \ldots, x_m$. Then, for any sequences $\boldsymbol{a} = a_1, \ldots, a_m$, $\boldsymbol{u}$ and $\boldsymbol{u}'$ of members of $\mathcal{D}$*

$$(\boldsymbol{au}, \boldsymbol{au}') \in [\![\dot{K}(\varphi[\boldsymbol{x}])]\!]^{\mathcal{D}^\dagger} \iff \mathcal{D} \models \varphi[\boldsymbol{a}/\boldsymbol{x}]$$

3.2 Translation of Constraint Logic Programs

To motivate the technical definitions below, we illustrate the program translation procedure with an example. Assume a language with constant 0, a unary function symbol s, constraint predicate $=$ and program predicate *add*. We can write the traditional Horn clause definition of Peano addition:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

This program is first *purified:* the variables in the head of the clauses defining each predicate are chosen to be a sequence of fresh variables x_1, x_2, x_3, with all bindings stated as equations in the tail.

$$\begin{aligned}&add(x_1,x_2,x_3) \longleftarrow x_1 = 0, x_2 = x_3.\\&add(x_1,x_2,x_3) \longleftarrow \exists x_4, x_5 x_1 = s(x_4), x_3 = s(x_5), add(x_4,x_2,x_5))\end{aligned}$$

The clauses are combined into a single definition similar to the Clark completion of a program. We also use the variable permutation π sending $x_1, x_2, x_3, x_4, x_5 \mapsto x_4, x_2, x_5, x_1, x_3$ to rewrite the occurrence of the predicate *add* in the tail so that its arguments coincide with those in the head:

$$\begin{aligned}add(x_1,x_2,x_3) \leftrightarrow{} & (x_1 = 0, x_2 = x_3)\\&\vee\ \exists x_4, x_5, x_1 = s(x_4), x_3 = s(x_5), w_\pi\, add(x_1,x_2,x_3).\end{aligned}$$

Now we apply relational translation $\dot{K}$ defined above to all relation equations, and eliminate the existential quantifier using the *partial identity operator* I_3 defined above. We represent the permutation π using the relation expression W_π that simulates its behavior in a variable-free manner and replace the predicate *add* with a corresponding *relation variable* $\overline{add}$. (A formal definition of W_π and its connection with function w_π is given below, see Definition 7 and Lemma 4.)

$$\overline{add} \mathrel{\mathring{=}} \dot{K}(x_1 = o \wedge x_2 = x_3) \cup I_3((\dot{K}(x_1 = s(x_4) \wedge x_3 = s(x_5)) \cap W_\pi\, \overline{add}\, W_\pi^o)))$$

Now we give a description of the general translation procedure. We first process programs to their complete database form as defined in [6], which given the executable nature of our semantics reflects the choice to work within the minimal semantics. The main difference in our processing of a program P to its completed form P' is that a strict policy on variable naming is enforced, so that the resulting completed form is suitable for translation to relational terms.

Definition 5 (General Purified Form for Clauses). *For a clause* $p(\boldsymbol{t}[\boldsymbol{y}]) \leftarrow \boldsymbol{q}(\boldsymbol{v}[\boldsymbol{y}])$*, let* $h = \alpha(p)$*,* $y = |\boldsymbol{y}|$*,* $v = |\boldsymbol{v}|$*, and* $m = h + y + v$*. Assume vectors:*

$$\begin{array}{lllll}\boldsymbol{x} & = \boldsymbol{x}_h\boldsymbol{x}_t = \boldsymbol{x}_h\boldsymbol{x}_y\boldsymbol{x}_v & = x_1,\ldots,x_h, & x_{h+1},\ldots,x_{h+y}, & x_{h+y+1},\ldots,x_m\\ \boldsymbol{x}_h & & = x_1,\ldots,x_h & & \\ \boldsymbol{x}_t & = \quad \boldsymbol{x}_y\boldsymbol{x}_v & = & x_{h+1},\ldots,x_{h+y}, & x_{h+y+1},\ldots,x_m\\ \boldsymbol{x}_y & & = & x_{h+1},\ldots,x_{h+y} & \\ \boldsymbol{x}_v & & = & & x_{h+y+1},\ldots,x_m\end{array}$$

the clause's GPF form is:

$$p(\boldsymbol{x}_h) \leftarrow \exists^{h\uparrow}.((\boldsymbol{x}_h = \boldsymbol{t}[\boldsymbol{x}_y] \wedge \boldsymbol{x}_v = \boldsymbol{v}[\boldsymbol{x}_y]), \boldsymbol{q}(\boldsymbol{x}_v))$$

$\exists^{n\uparrow}$ denotes existential closure with respect to all variables whose index is greater than n. $\boldsymbol{x}_h$ and $\boldsymbol{x}_t$ stand for head and tail variables. A program is in GPF form iff every one of its clauses is. After the GPF step, we perform Clark's completion.

Definition 6 (Completion of a Predicate). *We define Clark's completed form for a predicate p with clauses $cl_1, \dots, cl_n$ in GPF form:*

$$\left.\begin{array}{l} p(\boldsymbol{x}_h) \leftarrow_{cl_1} tl_1 \\ \dots \\ p(\boldsymbol{x}_h) \leftarrow_{cl_n} tl_k \end{array}\right\} \xRightarrow{\textit{Clark's comp.}} p(\boldsymbol{x}_h) \leftrightarrow tl_1 \vee \cdots \vee tl_k$$

The above definition easily extends to programs. Completed forms are translated to relations by using $\dot{K}$ for the constraints, mapping conjunction to $\cap$ and $\vee$ to $\cup$. Existential quantification, recursive definitions and parameter passing are handled in a special way which we proceed to detail next.

Existential Quantification: Binding Local Variables. Variables *local* to the tail of a clause are existentially quantified. For technical reasons — simpler rewrite rules — we use the *partial identity* relation I_n, rather than the Q_n relation defined in the previous sections. I_n acts as an existential quantifier for all variables of index greater than a given number.

Lemma 3. *Let $\boldsymbol{a} = a_1, \dots, a_n \in \mathcal{D}$, $\boldsymbol{x} = x_1, \dots, x_n$, let φ be a constraint over m free variables, with $m > n$, $\boldsymbol{y}$ a vector of length k such that $n + k = m$, and $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{D}^\dagger$, then:*

$$(\boldsymbol{au}, \boldsymbol{av}) \in [\![I_n; \dot{K}(\varphi[\boldsymbol{xy}]); I_n]\!]^{\mathcal{D}^\dagger} \iff \mathcal{D} \models (\exists^{n\uparrow}.\varphi[\boldsymbol{xy}])[\boldsymbol{a}/\boldsymbol{x}]$$

Recursive Predicate Definitions. We shall handle recursive predicate definitions by extending the relational language with a set of definitional symbols $\overline{p}, \overline{q}, \overline{r}, \dots$ for predicates. Then, a recursive predicate $\overline{p}$ is translated to a definitional equation $\overline{p} \mathrel{\hat{=}} R(\overline{p}_1, \dots, \overline{p}_n)$, spelled out in Definition 8 where the notation $R(\overline{p}_1, \dots, \overline{p}_n)$ indicates that relation R resulting from the translation may depend on predicate symbols $\overline{p}_1, \dots, \overline{p}_n$. Note that R is monotone in $\overline{p}_1, \dots, \overline{p}_n$. Consequently, using a straightforward fixed point construction we can extend the interpretation $[\![\_]\!]^{\mathcal{D}^\dagger}$ to satisfy $[\![\overline{p}]\!]^{\mathcal{D}^\dagger} = [\![R(\overline{p}_1, \dots, \overline{p}_n)]\!]^{\mathcal{D}^\dagger}$, thus preserving soundness when we adjoin the definitional equations to $\mathbf{QRA}_\Sigma$. The details are given in Subsect. 3.3, below.

Parameter Passing. The information about the order of parameters in each pure atomic formula $p(x_{i_1}, \dots, x_{i_r})$ is captured using permutations. Given a permutation $\pi : \{1..n\} \to \{1..n\}$, the function w_π on formulas and terms is defined in the standard way by its action over variables. We write W_π for the corresponding relation:

Definition 7 (Switching Relations). *Let $\pi : \{1..n\} \to \{1..n\}$ be a permutation. The* switching relation expression *W_π, associated to π is:*

$$W_\pi = \bigcap_{j=1}^{n} P_{\pi(j)}(P_j)^\circ.$$

```
male(terach). male(haran). male(isaac). male(lot).

female(sarah). female(milcah). female(yiscah).

father(terach,haran). father(haran,lot). ↩
    father(haran,milcah).

mother(sarah,isaac).

parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).

sibling(S1,S2) ← S1≠S2, parent(Par,S1), parent(Par,S2).

brother(Brother,Sib) ← male(Brother), sibling(Brother,Sib).
```

Fig. 3. Biblical family relations in prolog.

Lemma 4. *Fix a permutation π and its corresponding w_π and W_π. Then:*

$$[\![\dot{K}(w_\pi(p(x_1,\dots,x_n)))]\!] = [\![W_\pi \dot{K}(p) W_\pi^\circ]\!]$$

The Translation Function. Now we may define the translation for defined predicates.

Definition 8 (Relational Translation of Predicates). *Let $h, p(\boldsymbol{x}_h)$ be as in Definition 5. The translation function Tr from completed predicates to relational equations is defined by:*

$$\begin{array}{lll}
Tr(p(\boldsymbol{x}_h) \leftrightarrow cl_1 \vee \dots \vee cl_k) & = (\overline{p} \mathrel{\dot{=}} Tr_{cl}(cl_1) \cup \dots \cup Tr_{cl}(cl_k)) & \\
Tr_{cl}(\exists^{h\uparrow}.\boldsymbol{p}) & = I_h; (Tr_l(p_1) \cap \dots \cap Tr_l(p_n)); I_h & \\
Tr_l(\varphi) & = \dot{K}(\varphi) & \varphi \textit{ a constraint} \\
Tr_l(p_i(\boldsymbol{x}_i)) & = W_\pi; \overline{p_i}; W_\pi^\circ \quad \textit{such that } \pi(x_1,\dots,x_{\alpha(p_i)}) = \boldsymbol{x}_i &
\end{array}$$

where $\boldsymbol{x}_i$ is the original sequence of variables in p_i in the Clark completion of the program, and π a permutation that transforms the ordered sequence of length $\alpha(p)$ starting at x_1 to $\boldsymbol{x}_i$.

We will sometimes write $I_n(R)$ for $I_n R I_n$ and $W_\pi(R)$ for $W_\pi R W_i^\circ$.

Example 1. Figure 3 shows a fragment of a constraint logic program to represent a family relations database [20].

Consider the translation of the program predicates mother, parent, sibling and brother. We write the program in general purified form:

$$\begin{aligned}
mother(x_1,x_2) &\iff (x_1 = sarah) \wedge (x_2 = isaac)\\
parent(x_1,x_2) &\iff father(x_1,x_2) \vee mother(x_1,x_2)\\
sibling(x_1,x_2) &\iff \exists x_3.\ x_1 \neq x_2 \wedge parent(x_3,x_1) \wedge parent(x_3,x_2)\\
brother(x_1,x_2) &\iff male(x_1) \wedge sibling(x_1,x_2)
\end{aligned}$$

Letting σ_1 and σ_2 be the permutations $\langle 1,2,3\rangle \longrightarrow \langle 2,3,1\rangle$ and $\langle 1,2,3\rangle \longrightarrow \langle 3,2,1\rangle$ respectively we obtain

$$\begin{aligned}
\overline{mother} &= \dot{K}(x_1 = sarah) \cap \dot{K}(x_2 = isaac)\\
\overline{parent} &= \overline{father} \cup \overline{mother}\\
\overline{sibling} &= \dot{K}(x_1 \neq x_2) \cap I_2[W_{\sigma_1}\overline{parent}W^o_{\sigma_1} \cap W_{\sigma_2}\overline{parent}W^o_{\sigma_2}]I_2\\
\overline{brother} &= \overline{male} \cap \overline{sibling}
\end{aligned}$$

The query $brother(X, milcah)$ leads to the rewriting of the term $\dot{K}(x_2 = milcah) \cap \overline{brother}$ to $\dot{K}(x_2 = milcah) \cap \dot{K}(x_1 = lot)$.

3.3 The Least Relational Interpretation Satisfying Definitional Equations

Let P be a program and $\overline{p}_1, \ldots, \overline{p}_n$ be a sequence of *relation variables*, one for each predicate symbol p_i in the language of P. We define the extended relation calculus $\mathsf{R}_\Sigma(\overline{p}_1, \ldots, \overline{p}_n)$ to be the set of terms generated by $\overline{p}_1, \ldots, \overline{p}_n$ and the terms of R_Σ. More formally

$$\begin{aligned}
&\mathsf{R}_{atom} &&::= \overline{p}_1 \mid \cdots \mid \overline{p}_n \mid \mathsf{R}_{\mathcal{C}} \mid \mathsf{R}_{\mathcal{F}} \mid \mathsf{R}_{\mathcal{CP}} \mid id \mid di \mid \mathbf{1} \mid \mathbf{0} \mid hd \mid tl\\
&\mathsf{R}_\Sigma(\overline{p}_1, \ldots, \overline{p}_n) &&::= \mathsf{R}_{atom} \mid {\mathsf{R}_\Sigma}^\circ \mid \mathsf{R}_\Sigma \cup \mathsf{R}_\Sigma \mid \mathsf{R}_\Sigma \cap \mathsf{R}_\Sigma \mid \mathsf{R}_\Sigma\mathsf{R}_\Sigma
\end{aligned}$$

Observe that the relational translation of Definition 8 maps programs to sets of definitional equations $\overline{p}_i \mathrel{\hat{=}} R_i(\overline{p}_1, \ldots, \overline{p}_n)$ over $\mathsf{R}_\Sigma(\overline{p}_1, \ldots, \overline{p}_n)$. Let $\mathcal{F}$ be the set of all n such definitional equations.

Given a structure $\mathcal{D}$ we now lift the definition of $\mathcal{D}$-*interpretation* given in Definition 1 to the extended relation calculus. An extended interpretation $[\![\ ]\!] : \mathsf{R}_\Sigma(\overline{p}_1, \ldots, \overline{p}_n) \longrightarrow \mathsf{R}_\mathcal{D}$ is a function satisfying the identities in Fig. 2 as well as mapping each relation variable $\overline{p}_i$ to an arbitrary member $[\![\overline{p}_i]\!]$ of $\mathsf{R}_\mathcal{D}$. Given a structure $\mathcal{D}$ for the language of a program, its action is completely determined by its values at the $\overline{p}_i$. Note that the set $\mathcal{I}$ of all such interpretations forms a CPO, a complete partial order with a least element, under pointwise operations. That is to say, any *directed* set $\{[\![\ ]\!]_d : d \in \Lambda\}$ of interpretations has a supremum $\bigvee_{d\in\Lambda}[\![\ ]\!]_d$ given by $T \mapsto \bigcup_{d\in\Lambda}[\![T]\!]_d$. The directedness assumption is necessary. For example, to show that a pointwise supremum of interpretations $\bigvee_{d\in\Lambda}[\![\ ]\!]_d$ preserves composition (one of the 13 identities of Fig. 2), we must show that for any relation terms R and S we have $\bigcup_{d\in\Lambda}[\![RS]\!]_d = \bigcup_{d\in\Lambda}[\![R]\!]_d; \bigcup_{d\in\Lambda}[\![S]\!]_d$. However the right hand side of this identity is equal to $\bigcup_{d,e\in\Lambda\times\Lambda}[\![R]\!]_d; [\![S]\!]_e$. But since the family of interpretations is directed, for every pair d, e of indices in Λ there is an $m \in \Lambda$ with $[\![\ ]\!]_d, [\![\ ]\!]_e \leq [\![\ ]\!]_m$, hence $\bigcup_{d,e\in\Lambda\times\Lambda}[\![R]\!]_d; [\![S]\!]_e \leq \bigcup_{m\in\Lambda}[\![R]\!]_m[\![S]\!]_m$. The reverse inequality is immediate and we obtain $\bigcup_{d\in\Lambda}[\![R]\!]_d; \bigcup_{d\in\Lambda}[\![S]\!]_d = \bigcup_{d\in\Lambda}[\![RS]\!]_d$.

The least element of the collection $\mathcal{I}$ is the interpretation $[\![\ ]\!]_0$ given by $[\![\overline{p_i}]\!]_0 = \emptyset$ for all i $(1 \leq i \leq n)$.

In the remainder of this section, the word *interpretation* will refer to an extended $\mathcal{D}$-interpretation.

Lemma 5. *Let $[\![\;]\!]$ and $[\![\;]\!]'$ be interpretations. If for all i $[\![\overline{p}_i]\!] \subseteq [\![\overline{p}_i]\!]'$ then $[\![\;]\!] \leq [\![\;]\!]'$.*

Proof. By induction on the structure of extended relations. For all relational constants c we have $[\![c]\!] = [\![c]\!]'$ We will consider one of the inductive cases, namely that of composition. Suppose $[\![R]\!] \subseteq [\![R]\!]'$ and $[\![S]\!] \subseteq [\![S]\!]'$. Then we must show that $[\![RS]\!] \subseteq [\![RS]\!]'$. But this follows immediately by a set-theoretic argument, since $(x, u) \in [\![R]\!]$ and $(u, y) \in [\![S]\!]$ imply, by inductive hypothesis, that $(x, u) \in [\![R]\!]'$ and $(u, y) \in [\![S]\!]'$. It can also be proved using the axioms of $\mathbf{QRA}_{\Sigma}$ by showing that $A \cup A' = A'$ and $B \cup B' = B'$ imply $AB \cup A'B' = A'B'$. We leave the remaining cases to the reader.

We will now define a operator $\Phi_{\mathcal{F}}$ from interpretations to interpretations, show it continuous and define the *interpretation generated by* $\mathcal{F}$ as its least fixed point. This interpretation will be the least extension of a given relational $\mathcal{D}$-interpretation satisfying the equations in $\mathcal{F}$.

Definition 9. *Let P be a program, with predicate symbols $\{p_1, \ldots, p_n\}$. Fix a structure $\mathcal{D}$ for the language of P. Let $\mathcal{F}$ be the set of definitional equations $\{\overline{p}_i \mathrel{\dot{=}} R_i(\overline{p}_1, \ldots, \overline{p}_n) : i \in \mathbb{N}\}$ produced by the translation Tr of P of Definition 8. Let $\mathcal{I}$ be the set of extended $\mathcal{D}$-interpretations, with poset structure induced pointwise. Then we define the operator $\Phi_{\mathcal{F}} : \mathcal{I} \longrightarrow \mathcal{I}$ as follows*

$$\Phi_{\mathcal{F}}([\![\;]\!])(\overline{p}_i) = [\![R_i(\overline{p}_1, \ldots, \overline{p}_n)]\!].$$

Theorem 2. *$\Phi_{\mathcal{F}}$ is a continuous operator, that is to say it preserves suprema of directed sets.*

Proof. Let $\{[\![\;]\!]_d : d \in \Lambda\}$ be a directed set of interpretations. By Lemma 5 it suffices to show that for all p_i

$$\Phi_{\mathcal{F}}(\bigvee_{d \in \Lambda} [\![\;]\!]_d)(\overline{p}_i) = (\bigvee_{d \in \Lambda} \Phi_{\mathcal{F}}([\![\;]\!]_d))(\overline{p}_i).$$

Let $[\![\;]\!]^* = \bigvee_{d \in \Lambda} [\![\;]\!]_d$. Then $\Phi_{\mathcal{F}}(\bigvee_{d \in \Lambda} [\![\;]\!]_d)(\overline{p}_i) = [\![R_i(\overline{p}_1, \ldots, \overline{p}_n)]\!]^*$, which in turn is the union $\bigcup_{d \in \Lambda} [\![R_i(\overline{p}_1, \ldots, \overline{p}_n)]\!]_d$. But this is equal to $\bigcup_{d \in \Lambda} \Phi_{\mathcal{F}}([\![\;]\!]_d)(\overline{p}_i)$. Therefore $\Phi_{\mathcal{F}}(\bigvee_{d \in \Lambda} [\![\;]\!]_d) = \bigvee_{d \in \Lambda} \Phi_{\mathcal{F}}([\![\;]\!]_d)$.

By Kleene's fixed point theorem $\Phi_{\mathcal{F}}$ has a least fixed point $[\![\;]\!]^{\dagger}$ in $\mathcal{I}$. This fixed point is, in fact, the union of all $\Phi_{\mathcal{F}}^{(n)}([\![\;]\!]_0), (n \in \mathbb{N})$. By virtue of its being fixed by $\Phi_{\mathcal{F}}$ we have $[\![\overline{p}_i]\!]^{\dagger} = [\![R_i(\overline{p}_1, \ldots, \overline{p}_n)]\!]^{\dagger}$. That is to say, all equations in $\mathcal{F}$ are true in $[\![\;]\!]^{\dagger}$, which is the least interpretation with this property under the pointwise order.

4 A Rewriting System for Resolution

In this section, we develop a rewriting system for proof search based on the equational theory $\mathbf{QRA}_{\Sigma}$, which will be proven equivalent to the traditional

$$\begin{array}{llll}
m_1 & : I_m(\dot{K}(\psi)) & \xrightarrow{P} \dot{K}(\exists^{m\uparrow}.\psi) & \text{Hiding meta-reduction} \\
m_1* & : I_m(\mathbf{0}) & \xrightarrow{P} \mathbf{0} & \\
m_2 & : W_\pi(\dot{K}(\psi)) & \xrightarrow{P} \dot{K}(w_\pi(\psi)) & \text{Permutation meta-reduction} \\
m_2* & : W_\pi(\mathbf{0}) & \xrightarrow{P} \mathbf{0} & \\
m_3 & : \dot{K}(\psi_1) \cap \dot{K}(\psi_2) & \xrightarrow{P} \dot{K}(\psi_1 \wedge \psi_2) & \mathcal{D} \models \psi_1 \wedge \psi_2 \\
m_3 & : \dot{K}(\psi_1) \cap \dot{K}(\psi_2) & \xrightarrow{P} \mathbf{0} & \mathcal{D} \not\models \psi_1 \wedge \psi_2 \\
m_4 & : \dot{K}(\psi) \cap \overline{q} & \xrightarrow{P} \dot{K}(\psi) \cap (\Theta) & \text{where } \overline{q} \mathrel{\dot{=}} \Theta \in Tr(P)
\end{array}$$

Fig. 4. Constraint meta-reductions

operational semantics for CLP. In Sect. 5 we will show that answers obtained by resolution correspond to answers yielded by our rewriting system and conversely.

The use of ground terms permits the use of rewriting, overcoming the practical and theoretical difficulties that the existence of logic variables causes in equational reasoning. Additionally, we may speak of *executable* semantics: we use the same function to compile and interpret CLP programs in the relational denotation.

For practical reasons, we don't rewrite over the full relational language, but we will use a more compact representation of the relations resulting from the translation.[1]

Formally, the signature of our rewriting system is given by the following term-forming operations over the sort $\mathcal{T}_R$: $\mathsf{I} : (\mathbb{N} \times \mathcal{T}_R) \to \mathcal{T}_R$, $\mathsf{W} : (\mathsf{Perm} \times \mathcal{T}_R) \to \mathcal{T}_R$, $\mathsf{K} : \mathcal{L}_\mathcal{D} \to \mathcal{T}_R$, $\cup : (\mathcal{T}_R \times \mathcal{T}_R) \to \mathcal{T}_R$ and $\cap : (\mathcal{T}_R \times \mathcal{T}_R) \to \mathcal{T}_R$. Thus, for instance, the relation $I_n; R; I_n$ is formally represented in the rewriting system as $\mathsf{I}(n, \mathsf{R})$, provided R can be represented in it. In practice we make use of the conventional relational notation I_n, W_π when no confusion can arise.

4.1 Meta-Reductions

We formalize the interface between the rewrite system and the constraint solver as meta-reductions (Fig. 4). Every meta-reduction uses the constraint solver in a black-box manner to perform constraint manipulation and satisfiability checking.

Lemma 6. *All meta-reductions are sound: if* $m_i : l \xrightarrow{P} r$ *then* $[\![l]\!]^{\mathcal{D}^\dagger} = [\![r]\!]^{\mathcal{D}^\dagger}$.

4.2 A Rewriting System for SLD Resolution

We present a rewriting system for proof search in Fig. 5. We prove local confluence. Later we will prove that a query rewrites to a term in the canonical form $\dot{K}(\psi) \cup R$ iff the leftmost branch of the associated SLD-tree of the program is finite.

[1] There is no problem in defining the rewriting system using the general relational signature, but we would need considerably more rules for no gain.

$$\begin{array}{ll}
p_1 : \mathbf{0} \cup R & \xmapsto{P} R \\
p_2 : \mathbf{0} \cap R & \xmapsto{P} \mathbf{0} \\
p_3 : W_\pi(R \cup S) & \xmapsto{P} W_\pi(R) \cup W_\pi(S) \\
p_4 : I_n(R \cup S) & \xmapsto{P} I_n(R) \cup I_n(S) \\
p_5 : (R \cup S) \cap T & \xmapsto{P} (R \cap T) \cup (S \cap T) \\
p_6 : \dot{K}(\psi) \cap (R \cup S) & \xmapsto{P} (\dot{K}(\psi) \cap R) \cup (\dot{K}(\psi) \cap S) \\
p_7 : \dot{K}(\psi) \cap (R \cap W_\pi(\overline{q_i})) & \xmapsto{P} (\dot{K}(\psi) \cap R) \cap W_\pi(\overline{q_i}) \\
p_8 : \dot{K}(\psi) \cap W_\pi(\overline{q}) & \xmapsto{P} W_\pi^\circ(W_\pi(\dot{K}(\psi)) \cap \overline{q}) \\
p_9 : \dot{K}(\psi) \cap I_m(R) & \xmapsto{P} I_m(I_m(\dot{K}(\psi)) \cap R) \cap \dot{K}(\psi)
\end{array}$$

Fig. 5. Rewriting system for *SLD*.

Lemma 7. $\xmapsto{P}$ *is sound: if* $p_i : l \xmapsto{P} r$ *then* $[\![l]\!]^{\mathcal{D}^\dagger} = [\![r]\!]^{\mathcal{D}^\dagger}$.

Lemma 8. *If we give higher priority to* p_7 *over* p_8, $\xmapsto{P}$ *is locally confluent.*

A *left outermost strategy* gives priority to p_7 over p_8.

5 Operational Equivalence

We prove that our rewriting system over relational terms simulates "traditional" SLD proof search specified as a transition-based operational semantics (i.e. [7, 12]). For reasons of space, we give a high-level overview of the proof. The full details can be found in the online technical report.

Recall a *resolvent* is a sequence of atoms or constraints $\boldsymbol{p}$. We write $\square$ for the empty resolvent. We assume given a constraint domain $\mathcal{D}$ and its satisfaction relation $\mathcal{D} \models \varphi$. A *program state* is an ordered pair $\langle \boldsymbol{p} \mid \varphi \rangle$ where $\boldsymbol{p}$ is a resolvent and φ is a constraint (called the *constraint store*). The notation $cl : p(\boldsymbol{u}[\boldsymbol{y}]) \leftarrow \boldsymbol{q}(\boldsymbol{v}[\boldsymbol{z}])$ indicates that $p(\boldsymbol{u}[\boldsymbol{y}]) \leftarrow \boldsymbol{q}(\boldsymbol{v}[\boldsymbol{z}])$ is a program clause with label cl. Then, the standard operational semantics for SLD resolution can be defined as the following transition system over program states:

Definition 10 (Standard SLD Semantics).

$$\begin{array}{lll}
\langle \varphi, \boldsymbol{p} \mid \psi \rangle & \xrightarrow{cs}_l & \langle \boldsymbol{p} \mid \psi \wedge \varphi \rangle \qquad \textit{iff } \mathcal{D} \models \psi \wedge \varphi \\
\langle p(\boldsymbol{t}[\boldsymbol{x}]), \boldsymbol{p} \mid \varphi \rangle & \xrightarrow{res_{cl}}_l & \langle \boldsymbol{q}(\boldsymbol{v}[\sigma(\boldsymbol{z})]), \boldsymbol{p} \mid \varphi \wedge (\boldsymbol{u}[\sigma(\boldsymbol{y})] = \boldsymbol{t}[\boldsymbol{x}]) \rangle \\
& \textit{where:} & cl : p(\boldsymbol{u}[\boldsymbol{y}]) \leftarrow \boldsymbol{q}(\boldsymbol{v}[\boldsymbol{z}]) \\
& & \mathcal{D} \models \varphi \wedge (\boldsymbol{u}[\sigma(\boldsymbol{y})] = \boldsymbol{t}[\boldsymbol{x}]) \\
& & \sigma \textit{ a renaming apart for } \boldsymbol{y}, \boldsymbol{z}, \boldsymbol{x}
\end{array}$$

Taking the previous system as a reference, the proof proceeds in two steps: we first define a new transition system that internalizes renaming apart and proof search, and we prove it equivalent to the standard one.

Second, we show a simulation relation between the fully internalized transition system and a transition system defined over relations, which is implemented by the rewriting system of Sect. 4.

With these two equivalences in place, the main theorem is:

Theorem 3. *The rewriting system of Fig. 5 implements the transition system of Definition 10. Formally, for every transition* $(r_1, r_2) \in (\rightarrow_l)^*$,

$$\exists n.(Tr(r_1), Tr(r_2)) \in (\xmapsto{P})^n$$

and

$$\forall r_3.(Tr(r_1), r_3) \in (\xmapsto{P})^n \Rightarrow Tr(r_2) = r_3$$

Thus, given a program P, relational rewriting of translation will return an answer constraint $K(\varphi)$ iff SLD resolution from P reaches a program state $\langle \Box \mid \varphi' \rangle$, with $\varphi \iff \varphi'$.

In the next section, we briefly describe the main intermediate system used in the proof.

5.1 The Resolution Transition System

The crucial part of the SLD-simulation proof is the definition of a new extended transition system over program states that will internalize both renaming apart and the proof-search tree. It is an intermediate system between relation rewriting and traditional proof search.

The first step towards the new system is the definition of an extended notion of state. In the standard system of Definition 10, a state is a resolvent plus a constraint store. Our extended notion of state includes:

– A notion of *scope*, which is captured by a natural number which can be understood as the number of global variables of the state.
– A notion of *substate*, which includes information about parameter passing in the form of a *permutation*.
– A notion of clause *selection*, and
– a notion of *failure* and *parallel state*, which represents failures in the search tree and alternatives.

Such states are enough to capture all the meta-theory of constraint logic programming except recursion, which operates meta-logically by replacing predicate symbols by their definitions. Formally:

Definition 11. *The set* $\mathcal{PS}$ *of* resolution states *is inductively defined as:*

– $\langle \mathit{fail} \rangle$.
– $\langle \boldsymbol{p} | \varphi \rangle_n$*, where* $p_i \equiv P_i(\boldsymbol{x}_i)$ *is an atom, or a constraint* $p_i \equiv \psi$*,* $\boldsymbol{x}_i$ *a vector of variables,* φ *a constraint store and* n *a natural number.*
– $\langle {}^{\pi} PS, \boldsymbol{p} | \varphi \rangle_n$*, where* PS *is a resolution state, and* π *a permutation.*
– $\langle {}^{\pi} \blacktriangleright PS, \boldsymbol{p} | \varphi \rangle_n$*, the "select state". It represents the state just before selecting a clause to proceed with proof search.*
– $\big(PS_1 \,\rule{2pt}{8pt}\, PS_2\big)$*. The bar is parallel composition, capturing choice in the proof search tree.*

$$
\begin{array}{rcl}
\langle \psi, \boldsymbol{p} \mid \varphi \rangle_n & \xrightarrow{constraint}_p & \langle \boldsymbol{p} \mid \varphi \wedge \psi \rangle_n \\
\langle \psi, \boldsymbol{p} \mid \varphi \rangle_n & \xrightarrow{fail}_p & \langle \mathit{fail} \rangle \\
 & & \boxed{\text{if } \varphi \wedge \psi \text{ is not satisfiable}} \\
\langle p(\boldsymbol{x}), \boldsymbol{p} \mid \varphi \rangle_n & \xrightarrow{call}_p & \left\langle^{\pi} \blacktriangleright (\langle \boldsymbol{q}_1 \mid \top \rangle_h \mathbin{\pmb{\vert}} \ldots \mathbin{\pmb{\vert}} \langle \boldsymbol{q}_k \mid \top \rangle_h), \boldsymbol{p} \mid \varphi \right\rangle_n \\
 & & \boxed{\text{if } p(\boldsymbol{x}_h) \leftarrow \exists^{h\uparrow}.(\boldsymbol{q}_1 \vee \ldots \vee \boldsymbol{q}_k) \in P',\ \pi(\boldsymbol{x}) = \boldsymbol{x}_h} \\
\left\langle^{\pi} \blacktriangleright (\langle \boldsymbol{q} \mid \psi \rangle_h \mathbin{\pmb{\vert}} PS), \boldsymbol{p} \mid \varphi \right\rangle_n & \xrightarrow{select}_p & \left(\left\langle^{\pi} \langle \boldsymbol{q} \mid \psi \wedge \Delta^{\pi}_h(\varphi) \rangle_h, \boldsymbol{p} \mid \varphi \right\rangle_n \mathbin{\pmb{\vert}} \left\langle^{\pi} \blacktriangleright PS, \boldsymbol{p} \mid \varphi \right\rangle_n \right) \\
\left\langle^{\pi} \langle \Box \mid \psi \rangle_h, \boldsymbol{p} \mid \varphi \right\rangle_n & \xrightarrow{return}_p & \langle \boldsymbol{p} \mid \nabla^{\pi}_h(\psi, \varphi) \rangle_n \\
\left\langle^{\pi} \langle \mathit{fail} \rangle, \boldsymbol{p} \mid \varphi \right\rangle_n & \xrightarrow{return}_p & \langle \mathit{fail} \rangle \\
\left\langle^{\pi} PS, \boldsymbol{p} \mid \varphi \right\rangle_n & \xrightarrow{sub}_p & \left\langle^{\pi} PS', \boldsymbol{p} \mid \varphi \right\rangle_n \\
 & & \boxed{\text{if } PS \neq \langle \Box \mid \psi \rangle_n,\ PS \neq \langle \mathit{fail} \rangle, \text{ and } PS \to_p PS'} \\
(\langle \mathit{fail} \rangle \mathbin{\pmb{\vert}} PS) & \xrightarrow{backtrack}_p & PS \\
(PS_1 \mathbin{\pmb{\vert}} PS_2) & \xrightarrow{seq}_p & (PS'_1 \mathbin{\pmb{\vert}} PS_2) \\
 & & \boxed{\text{if } PS \neq \langle \mathit{fail} \rangle, \text{ and } PS_1 \to_p PS'_1}
\end{array}
$$

(We omit the case in *select* where the left side has no PS component which happens when the number of clauses for a given predicate is one ($k = 1$))

Fig. 6. Resolution transition system

The *resolution transition system* $\to_P \subseteq (\mathcal{PS} \times \mathcal{PS})$ is shown in Fig. 6. The two first transitions deal with the case where a constraint is first in the resolvent, failing or adding it to the constraint store in case it is satisfiable.

When the head of the resolvent is a defined predicate, the *call* transition will replace it by its definition, properly encapsulated by a select state equipped with the permutation capturing argument order.

The *select* transition performs two tasks: first, it modifies the current constraint store adding the appropriate permutation and scoping (n, π); second, it selects the first clause for proof search.

The *return* transitions will either propagate failure or undo the permutation and scoping performed at call time.

sub, *backtrack*, and *seq* are structural transitions with a straightforward interpretation from a proof search perspective.

Then, we have the following lemma:

Lemma 9. *For all queries $\langle \boldsymbol{p} | \varphi \rangle_n$, the first successful $\to_l$ derivation using a SLD strategy uniquely corresponds to a $\to_p$ derivation:*

$$\langle \boldsymbol{p} | \varphi \rangle_n \to_l \ldots \to_l \langle \Box \mid \varphi' \rangle_n \quad \Longleftrightarrow \quad \langle \boldsymbol{p} | \varphi \rangle_n \to_p \ldots \to_p \left(\langle \Box \mid \varphi' \rangle_n \mathbin{\pmb{\vert}} PS \right)$$

for some resolution state PS.

Corollary 1. *The transition systems of Definition 10 and Fig. 6 are answer-equivalent: for any query they return the same answer constraint.*

With this lemma in place, the proof of Theorem 3 is completed by showing a simulation between the resolution system and a transition system induced by relation rewriting.

6 Related and Future Work

Previous Work: The paper is the continuation of previous work in [4,10,15] considerably extended to include constraint logic programming, which requires a different translation procedure and a different rewriting system.

In particular, the presence of constraints in this paper permits a different translation of the Clark completion of a program and plays a crucial role in the proof of completeness, which was missing in earlier work. The operational semantics is also new.

Related Work: A number of solutions have been proposed to the syntactic specification problem. There is an extensive literature treating abstract syntax of logic programming (and other programming paradigms) using encodings in higher-order logic and the lambda calculus [18], which has been very successful in formalizing the treatment of substitution, unification and renaming of variables, although it provides no special framework for the management and progressive instantiation of logic variables, and no treatment of constraints. Our approach is essentially orthogonal to this, since it relies on the complete elimination of variables, substitution, renaming and, in particular, existentially quantified variables. Our reduction of management of logic variables to variable free rewriting is new, and provides a complete solution to their formal treatment.

An interesting approach to syntax specification is the use of nominal logic [5,22] in logic programming, another, the formalization of logic programming in categorical logic [1,2,8,13,19] which provides a mathematical framework for the treatment of variables, as well as for derivations [14]. None of the cited work gives a solution that simultaneously includes logic variables, constraints and proof search strategies however.

Bellia and Occhiuto [3] have defined a new calculus, the C-expression calculus, to eliminate variables in logic programming. We believe our translation into the well-understood and scalable formalism of relations is more applicable to extensions of logic programming. Furthermore the authors do not consider constraints.

Future Work: A complementary approach to this work is the use of category theory, in particular the Freyd's theory of *tabular allegories* [9] which extends the relation calculus to an abstract category of relations providing native facilities for generation of fresh variables and a categorical treatment of monads. A first attempt in this direction has been published by the authors in [11]. It would be interesting to extend the translation to hereditarily Harrop or higher order logic [17] by using a stronger relational formalism, such as Division and Power Allegories. Also, the framework would yield important benefits if it was extended to include relation and set constraints explicitly.

7 Conclusion

We have developed a declarative relational framework for the compilation of Constraint Logic programming that eliminates logic variables and gives an algebraic treatment of program syntax. We have proved operational equivalence to the classical approach. Our framework has several significant advantages.

Programs can be analyzed, transformed and optimized entirely within this framework. Execution is carried out by rewriting over relational terms. In these two ways, specification and implementation are brought much closer together than in the traditional logic programming formalism.

References

1. Amato, G., Lipton, J., McGrail, R.: On the algebraic structure of declarative programming languages. Theor. Comput. Sci. **410**(46), 4626–4671 (2009), http://www.sciencedirect.com/science/article/B6V1G-4WV15VS-7/2/5475111b9a9642244a208e9bd1fcd46a (abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi)
2. Asperti, A., Martini, S.: Projections instead of variables: a category theoretic interpretation of logic programs. In: ICLP, pp. 337–352 (1989)
3. Bellia, M., Occhiuto, M.E.: C-expressions: a variable-free calculus for equational logic programming. Theor. Comput. Sci. **107**(2), 209–252 (1993)
4. Broome, P., Lipton, J.: Combinatory logic programming: computing in relation calculi. In: ILPS'94: Proceedings of the 1994 International Symposium on Logic programming, pp. 269–285. MIT Press, Cambridge (1994)
5. Cheney, J., Urban, C.: Alpha-prolog: a logic programming language with names, binding, and alpha-equivalence (2004)
6. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press (1977)
7. Comini, M., Levi, G., Meo, M.C.: A theory of observables for logic programs. Inf. Comput. **169**(1), 23–80 (2001)
8. Finkelstein, S.E., Freyd, P.J., Lipton, J.: A new framework for declarative programming. Theor. Comput. Sci. **300**(1–3), 91–160 (2003)
9. Freyd, P., Scedrov, A.: Categories, Allegories. North Holland Publishing Company, Amsterdam (1991)
10. Gallego Arias, E.J., Lipton, J., Mariño, J., Nogueira, P.: First-order unification using variable-free relational algebra. Log. J. IGPL **19**(6), 790–820 (2011). http://jigpal.oxfordjournals.org/content/19/6/790.abstract
11. Gallego Arias, E.J., Lipton, J.: Logic programming in tabular allegories. In: Dovier, A., Costa, V.S. (eds.) Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4–8, 2012, Budapest, Hungary. LIPIcs, vol. 17, pp. 334–347. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik (2012)
12. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. J. Log. Program. **19/20**, 503–581 (1994). http://citeseer.ist.psu.edu/jaffar94constraint.html
13. Kinoshita, Y., Power, A.J.: A fibrational semantics for logic programs. In: Dyckhoff, R., Herre, H., Schroeder-Heister, P. (eds.) ELP. LNCS, vol. 1050, pp. 177–191. Springer, Heidelberg (1996)
14. Komendantskaya, E., Power, J.: Coalgebraic derivations in logic programming. In: Bezem, M. (ed.) CSL. LIPIcs, vol. 12, pp. 352–366. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik (2011)

15. Lipton, J., Chapman, E.: Some notes on logic programming with a relational machine. In: Jaoua, A., Kempf, P., Schmidt, G. (eds.) Using Relational Methods in Computer Science, pp. 1–34. Technical report Nr. 1998-03, Fakultät für Informatik, Universität der Bundeswehr München, July 1998
16. Lloyd, J.W.: Foundations of Logic Programming. Springer, New York (1984)
17. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Ann. Pure Appl. Log. **51**(1–2), 125–157 (1991)
18. Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI'88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pp. 199–208. ACM, New York (1988)
19. Rydeheard, D.E., Burstall, R.M.: A categorical unification algorithm. In: Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming, pp. 493–505. Springer, New York (1986)
20. Sterling, L., Shapiro, E.: The Art of Prolog. The MIT Press, Cambridge (1986)
21. Tarski, A., Givant, S.: A Formalization of Set Theory Without Variables, Colloquium Publications, vol. 41. American Mathematical Society, Providence (1987)
22. Urban, C., Pitts, A.M., Gabbay, M.J.: Nominal unification. Theor. Comput. Sci. **323**(1–3), 473–497 (2004)