Logic Programming in Tabular Allegories*

[Technical Communication]

Emilio Jesús Gallego Arias[1] **and James B. Lipton**[2]

1 **Universidad Politécnica de Madrid**
2 **Wesleyan University**

Abstract

We develop a compilation scheme and categorical abstract machine for execution of logic programs based on allegories, the categorical version of the calculus of relations. Operational and denotational semantics are developed using the same formalism, and query execution is performed using algebraic reasoning. Our work serves two purposes: achieving a formal model of a logic programming compiler and efficient runtime; building the base for incorporating features typical of functional programming in a *declarative* way, while maintaining 100% compatibility with existing Prolog programs.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic

Keywords and phrases Category Theory,Logic Programming,Lawvere Categories,Programming Language Semantics,Declarative Programming

1 Introduction

Relational algebras have a broad spectrum of applications in both theoretical and practical computer science. In particular, the calculus of binary relations [37], whose main operations are intersection ($\cup$), union ($\cap$), relative complement $\setminus$, inversion $(\_)^o$ and relation composition ($;$) was shown by Tarski and Givant [40] to be a complete and adequate model for capturing all first-order logic and set theory. The intuition is that conjunction is modeled by $\cap$, disjunction by $\cup$ and existential quantification by composition.

This correspondence is very useful for modeling logic programming. Logic programs are naturally interpreted by binary relations and relation algebra is a suitable framework for algebraic reasoning over them, including execution of queries.

Previous versions of this work [24, 32, 9, 22], developed operational and denotational semantics for constraint logic programming using distributive relational algebra with a quasi-projection operator. While pure relational algebraic semantics have very pleasant properties, individual relations range over a unique domain or carrier: the union of the set of all finite sequences of terms generated by the signature of the program.

In this approach, operational reasoning is implemented by a rewriting system, but making it efficient is difficult, as we cannot directly reason about the number of logical variables in use. When a predicate call happens, the constraint store is duplicated, with one belonging to the caller environment and one used by the called predicate. At return time, the constraint stores are merged. The propagation of constraints posted inside a procedure call is delayed.

We propose to remedy this shortcoming by using *typed* relations. The theory of allegories [21], provides a categorical setting for distributive relational algebras. In this setting,

* The authors wants to acknowledge Wesleyan University for supporting this work with Van Vleck funds. This work is part of DESAFIOS10 (TIN2009-14599-C03-00)

relations are typed and the semantics for our relations becomes sequences of fixed-length. Now, the notion of categorical product and its associated projections interpret in an adequate way the shared context required to have an efficient execution model.

The most important concepts in our work are the notion of *strictly associative product* and *tabular relation*. Given types A, B (or *objects* in categorical language), we write $A \times B$ for their cartesian product. As usual $A \times (B \times C)$ is isomorphic ($\approx$) to $(A \times B) \times C$. We say our products are *strictly associative* if the isomorphism is an equality. That is, $(A \times B) \times C = A \times (B \times C)$. We are thus allowed to write $A \times B \times C$. This is a crucial fact for our machine, since if we interpret a chosen type H as a memory cell, then a memory region of size n is interpreted as H^n.

Second, we say a relation $R : A \leftrightarrow B$ is tabulated by an injective (monic) function (arrow) $f : C \to A \times B$ if every pair of the relation is in its image. We may split f into its components $f;\pi_1 : C \to A$ and $f;\pi_2 : C \to B$, and state that the pair $(f;\pi_1, f;\pi_2)$ tabulates R. Such a concept is fundamental for two reasons: the types of the tabulations carry important information about the memory use of the machine. The domain of the tabulations corresponds to *global storage* or heap and the co-domain represents the number of *registers* our machine is using at a given state.

The execution mechanism is entirely based on the composition of tabular relations, an operation fully characterized by the pullback of its tabulations. Relation composition models unification, parameter passing, renaming apart, allocation of new temporary variables and garbage collection.

The first important benefit of our use of categorical concepts is the small gap from the categorical specification to the actual machine and proposed implementation. This allows us to reason using a very convenient algebraic style, immediately witnessing the impact of such reasoning on the machine itself. Our philosophy is that in an fully algebraic framework, efficient execution should belong to regular reasoning. Real world implementations usually depart from this view in the name of efficiency, and one key objective of this work is to achieve efficiency without abandoning the algebraic approach. It is also worth noting that in our framework, we replace all the custom theory and meta-theory used in logic programming with category theory. The precise statement is that a Σ-allegory captures all the needed theory and meta-theory for a Logic Program with signature Σ, from set-theoretical semantics down to efficient execution.

The second — and in our opinion, most innovative benefit — is the possibility of seamlessly extending Prolog using constructions typical of functional programming in a fully *declarative way*. In [23], we sketch some of these extensions, adding algebraic data types, constraints, functions and monads to Prolog, all of it without losing source code compatibility with existing programs.

2 Logic Programming

Assume a permutative convention on symbols, i.e., unless otherwise stated explicitly, distinct names f, g stand for different entities (e.g. function symbols) and the same with distinct names i, j, for indices. A first-order language consists of a signature $\Sigma = \langle \mathcal{C}_\Sigma, \mathcal{F}_\Sigma \rangle$, given by $\mathcal{C}_\Sigma$, the set of constant symbols, and $\mathcal{F}_\Sigma$, the set of term formers or function symbols. $\mathcal{P}$ will denote the set of predicate symbols. Function $\alpha : \mathcal{P} \cup \mathcal{F}_\Sigma \to \mathbb{N}$ returns the arity of its predicate argument. We assume a set $\mathcal{X}$ of so-called *logic variables* whose members are denoted x_i. We write $\mathcal{T}_\Sigma$ for the set of closed terms over Σ. We write $\mathcal{T}_\Sigma(\mathcal{X})$ for the set of open terms (in the variables in $\mathcal{X}$) over Σ. We drop Σ when understood from context. We

$$
\begin{tikzcd}
 & & D & & \\
 & & P \arrow[u, two heads] \arrow[dll] \arrow[drr] \arrow[d, phantom, "\lrcorner" rotate=-45] & & \\
 & M \arrow[from=uur, "u"'] \arrow[dl, "f"] \arrow[dr, "g"] & & N \arrow[from=uul, "v"] \arrow[dl, "h"] \arrow[dr, "i"] & \\
A & & B & & C
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
 & D \arrow[dl, "u;f"'] \arrow[dr, "v;i"] & \\
A & & C
\end{tikzcd}
$$

Figure 1 Composition of Relations

write sequences of terms using vector notation: $\vec{t} = t_1, \ldots, t_n$. The length of such a sequence is written $|\vec{t}| = n$. We assume standard definitions for atoms, predicates, programs, clauses, and SLD resolution. For more details see [33].

3 Category Theory

A category $\mathcal{C} = \langle \mathcal{O}, \mathcal{A} \rangle$ consists of a collection of objects $\mathcal{O}$ and typed arrows $\mathcal{A}$. For every object $A \in \mathcal{O}$, there is an identity arrow $id_A : A \to A \in \mathcal{A}$. Given arrows $f : A \to B$ and $g : B \to C$, its composition $f;g : A \to C$ is defined. For $f : A \to B$, we call A the domain of f and B its codomain. Composition is associative and $id_A; f = f; id_B = f$. We assume knowledge of the concepts of *commutative diagram*, product, equalizer, pullback, monic arrow and subobject [6, 5, 30].

For a product $A \times B$, we will write $\pi_1^{A\times B} : A \times B \to A$ and $\pi_2^{A\times B} : A \times B \to B$ for the projections. For arrows $f : C \to A$, $g : C \to B$ we write $\langle f, g\rangle$ for the unique product former. Several definitions exist for Regular Categories [11, 6, 27, 21]; we use the latter presentation.

▶ **Definition 1** (Regular Category). A category $\mathcal{C}$ is a Regular Category if it has products, equalizers, images and pullback transfer covers. A Regular Category can be used to generate a tabular allegory. Indeed, Regular Categories give rise to categories of relations.

3.1 Categorical Relations

▶ **Definition 2** (Monic Pair). $f : C \to A$ and $g : C \to B$ is a monic pair iff $\langle f, g\rangle : C \mapsto A \times B$ is monic. A monic pair is a subobject of $A \times B$, thus we can see it as a relation from A to B:

$$
\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[dr, "g"] & \\
A & & B
\end{tikzcd}
$$

▶ **Definition 3** (Composition of Relations). The composition (u, v) of a relation (f, g) with (h, i) is defined by the diagram on the left in Fig. 1. Note that the purpose of the cover in that diagram is to ensure that the resulting relation remains a monic pair. The right diagram shows the already composed relation.

▶ **Definition 4** (Categories of Relations). For a regular category $\mathcal{C}$, the category $Rel(\mathcal{C})$ of relations has the same objects as $\mathcal{C}$, arrows $A \to B$ are monic pairs $(f : C \to A, g : C \to B)$ and composition is defined as above. $\mathcal{C}$ is a sub-category of $Rel(\mathcal{C})$. The inclusion functor sends an arrow $f : A \to B$ to the pair (id, f). If a morphism of $Rel(\mathcal{C})$ is in $\mathcal{C}$, we call it a *map*.

Given a relation (f, g), its inverse, or reciprocal is (g, f). The natural order-isomorphism $Sub(A \times B) \approx Rel(A, B)$ yields a semi-lattice structure on $Rel(A, B)$.

3.2 Lawvere Categories

A Lawvere category is a category $\mathcal{C}$ with a denumerable set $\mathbb{N}$ of distinct objects, where each object N is the n-th power of the object 1. 0 is the terminal object. We write $!_A : A \to 0$ for the terminal arrow. The product of $T^m \times T^n$ is T^{m+n}. Products are *strictly associative* since addition is associative, thus $((1 \times 1) \times 1) = (1 \times 2) = 3$. Note that this means $(id_2 \times id) : 2 \times 1 \to 2 \times 1 = id_3 : 3 \to 3$, or for $f : 2 \to 2$, $(f \times id_2) = \langle f; \pi_1, f; \pi_2, id_1, id_1 \rangle$, etc…

For a given signature Σ of a logic program, we build the corresponding (free or syntactic) Lawvere Category $\mathcal{C}_\Sigma$ as follows:

- For every constant $a \in \mathcal{T}_\Sigma$, we freely adjoin an arrow $a : 0 \to 1$.
- For every function symbol $f \in \mathcal{T}_\Sigma$ with arity $\alpha(f) = N$, we freely adjoin an arrow $f : N \to 1$.

A model of a Lawvere Category $\mathcal{C}$ is a functor $F : \mathcal{C} \to Set$ which preserves finite products and pullbacks. A homomorphism of $\mathcal{C}$-models is a natural transformation. The category of models $Mod(\mathcal{C}, Set)$ for $\mathcal{C}$ is the usual functor category.

Lawvere Categories are a natural framework for categorically representing algebraic theories. Examples of such categories $\mathcal{C}$ may be seen in [31], and some good treatments are in [6, 26].

3.3 Allegories

▶ **Definition 5** (Allegory). An allegory $\mathcal{R} = \{\mathcal{O}, \mathcal{A}\}$ is an enriched category, with objects $\mathcal{O}$ and *relations* $\mathcal{A}$. We write $R; S : A \to C$ for composition of relations $R : A \to B$ and $S : B \to C$. When there is no confusion possible we may also write RS for $R; S$. We add two new operations:

- For every relation $R : A \to B$ and $S : A \to B$, $(R \cap S) : A \to B$ is a relation.
- For every relation $R : A \to B$, $R^\circ : B \to A$ is a relation.

We write $R \subseteq S$ for $R \cap S = R$. The new operations obey the following laws:

$$\begin{array}{rclrcl} R \cap R & = & R & R \cap S & = & S \cap R \\ R \cap (S \cap T) & = & (R \cap S) \cap T & R^{\circ\circ} & = & R \\ (RS)^\circ & = & S^\circ; R^\circ & (R \cap S)^\circ & = & (R^\circ \cap S^\circ) \\ R; (S \cap T) & \subseteq & (R; S \cap R; T) & (R; S \cap T) & \subseteq & (R \cap T; S^\circ); S \end{array}$$

A *map* is a relation such that $R^\circ; R \subseteq id$ and $id \subseteq R; R^\circ$. We use capital letters for relations and small letters for maps. A relation R is coreflexive iff $R \subseteq id$. For an allegory $\mathcal{R}$, we shall denote its subcategory of maps by $Map(\mathcal{R})$. A pair of maps f, g tabulates a relation R iff $f^\circ; g = R$ and $f; f^\circ \cap \; g; g^\circ = 1$. The latter condition is equivalent to stating that f, g form a monic pair.

$$
\begin{tikzcd}
 & C \arrow[dr, "g"] & \\
A \arrow[ur, "f^\circ"] \arrow[rr, leftrightarrow, "R"] & & B
\end{tikzcd}
=
\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[dr, "g"] & \\
A \arrow[rr, leftrightarrow, "R"] & & B
\end{tikzcd}
\qquad
\begin{tikzcd}
 & C \arrow[dl, "g"'] \arrow[dr, "f"] & \\
B \arrow[rr, leftrightarrow, "R^\circ"] & & A
\end{tikzcd}
$$

It is easy to prove that a tabulation is unique up to isomorphism. A coreflexive relation $R \subseteq id$ is tabulated by a pair of the form (f, f). If $R = f^\circ; g$, then $R^\circ = g^\circ; f$.

An allegory is a tabular allegory iff every relation has a tabulation. For an allegory $\mathcal{R}$, $Map(\mathcal{R})$ is a regular category. The following lemma tells us that a tabular allegory really is the relational extension generated by its maps and that the concepts of regular category and tabular allegory intimately connected:

▶ **Lemma 6.** *If $\mathcal{R}$ is a tabular allegory then $\mathcal{R} \approx Rel(Map(\mathcal{R}))$. If $\mathcal{C}$ is a regular category then $\mathcal{C} \approx Map(Rel(\mathcal{C}))$. If $\mathcal{R} \approx Rel(\mathcal{C})$ then $Map(\mathcal{R}) \approx \mathcal{C}$.*

Proof. See [21] 2.147 and 2.148, 2.154. ◀

Composition of relations in a tabular allegory is thus defined in the same way as for categories of relations arising from a regular category, see Def. 3.

A distributive allegory is an allegory with a new relation denoted $\mathbf{0}_{AB}$ for every object A, B, and given relations R, S with the same type, $R \cup S$ is an arrow. They obey the following laws:

$$
\begin{array}{rcl@{\qquad}rcl}
R \cup R & = & R & R \cup S & = & S \cup R \\
R \cup (S \cup T) & = & (R \cup S) \cup T & \mathbf{0} \cup S & = & S \\
R \cup (R \cap S) & = & R & R; \mathbf{0} & = & \mathbf{0} \\
R(S \cup T) & = & RS \cup RT & R \cap (S \cup T) & = & (R \cap S) \cup (R \cup T)
\end{array}
$$

4 Regular Lawvere Categories and Σ-Allegories

The key idea is to use Lem. 6 to build an allegory from a Lawvere category. In order to do that, we need to define the concept of Regular Lawvere Category (RLC) $\mathcal{C}$ first. Then $Rel(\mathcal{C})$ generates a pre-Σ-allegory. However, this category is not distributive, so we $\cup$-complete it in order to obtain what we call a Σ-allegory.

▶ **Definition 7** (Regular Lawvere Category)**.** Given a Lawvere Category $\mathcal{C}$, we build its regular completion $\hat{\mathcal{C}}$ by adjoining an initial object $\bot$, the corresponding initial arrows $?_A : \bot \to A$ for every object A and applying the quotient $?_A; f =?_B$ for any arrow $f : A \to B$.

This completion effectively replaces the Lawvere Category concept of *existence* of an equalizer by the question: What is the domain of the equalizing arrow? Arrows not having an equalizer in $\mathcal{C}$ are equalized by $\bot$ in $\hat{\mathcal{C}}$.

▶ **Definition 8** (Initial Model)**.** Given a choice $\langle , \rangle$ of product in Set, and a choice of symbols for the signature Σ generating the Regular Lawvere Category $\mathcal{C}$ and set $\mathcal{T}_\Sigma$, the initial model of a Regular Lawvere Category $\mathcal{C}$ — that is to say, the initial object in $Mod(\mathcal{C}, Set)$ — is the functor $[\![]\!]$, with object and arrow components $([\![]\!]_O, [\![]\!]_A)$:

$$[\![\bot]\!]_O = \emptyset \qquad [\![0]\!]_O = \{\bullet\} \qquad [\![N]\!]_O = \mathcal{T}_\Sigma^N \quad N > 0$$

$$
\begin{array}{lcl}
[\![?_N]\!]_A & = & \emptyset \xrightarrow{\emptyset} [\![N]\!]_O \\
[\![!_N]\!]_A & = & \lambda x.\ \bullet \\
[\![c : 0 \to 1]\!]_A & = & \lambda \bullet .\ c \\
[\![f : N \to 1]\!]_A & = & \lambda(n_1, \ldots, n_N).\ f(n_1, \ldots, n_N) \\
[\![\pi_i : N \to 1]\!]_A & = & \lambda(n_1, \ldots, n_N).\ n_i \\
[\![\langle t_1, \ldots, t_N\rangle : M \to N]\!]_A & = & \lambda n.\langle [\![n]\!]_A ; [\![t_1]\!]_A, \ldots, [\![n]\!]_A ; [\![t_N]\!]_A \rangle
\end{array}
$$

▶ **Lemma 9.** *The regular completion of a Lawvere Category is a regular category.*

4.1 Σ-Allegories

A RLC cannot model disjunctive clauses in logic programs, as it doesn't tabulate distributive allegories, which are tabulated by a Pre-Logos [21]: regular categories whose subobjects form a complete lattice, not just a semi-lattice.

▶ **Definition 10** (Σ-Allegory)**.** Given a Regular Lawvere Category $\mathcal{C}$, we define a Σ-allegory $\mathcal{R}_\cup$ as the distributive allegory generated from the allegory $\mathcal{R} \approx Rel(\mathcal{C})$ by freely adding all union arrows and taking the quotient by the distributive laws. An inclusion functor $F : \mathcal{R} \to \mathcal{R}_\cup$ exists, and it is easy to see that all the relations in $\mathcal{R}_\cup$ that possess a union-free representation are tabular.

5 Translation of the Program

The translation procedure is almost identical to the one defined in [22]. A predicate is translated to a coreflexive relation. We use two helper relations, a partial identity I, is meant to *create* and *destroy* local (or existentially quantified) variables, and a permutation W, which puts the arguments in the right order for relation composition.

▶ **Definition 11** (I Relation)**.** The relation I_{MN}, with $M < N$ is tabulated by $(\langle \pi_1, \ldots, \pi_M\rangle, id_N)$.

This relation formalizes the intuition that the reciprocal of a projection creates a new variable, indeed $I_{12} = \pi_1^\circ$.

▶ **Definition 12** (W Relation)**.** For a projection $w : N \to M$, with $N \geq M$ and $K = N - M$, we denote by $w' : N \to N$ any of its extensions to a permutation such that the following equations are satisfied: $\{w'(K) = w^{-1}(1), \ldots, w'(K + M) = w^{-1}(M)\}$. For a given w', W is tabulated by $(N, \langle \pi_{w'(1)}, \ldots, \pi_{w'(N)}\rangle)$.

First, we complete every predicate in a similar way to Clark's [15]. The set of n variables occurring in the terms is renamed from y_1 to y_n. Then, every term t_i occurring as an argument in the head and tail is replaced by a fresh variable x_i, and the equation $x_i = t_i$ is added to the clause. After that process, clauses are of the form:

$$p(\vec{x}') \leftarrow \vec{x} = \vec{t}(\vec{y}), p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n).$$

$\vec{x}'$ a prefix of $\vec{x}$, $\vec{x}_i$ a selection of variables in $\vec{x}$ and $\vec{t}$ a sequence of terms using variables in $\vec{y}$. We replace $\vec{x}_i$ for projections $w_i(\vec{x})$ such $w_i(\vec{x}) = \vec{x}_i$. Clauses are now of the form:

$$p(\vec{x}') \leftarrow \vec{x} = \vec{t}(\vec{y}), p_1(w_1(\vec{x})), \ldots, p_n(w_n(\vec{x})).$$

Now we are ready to transform the clause into a relational term. The equation $\vec{x} = \vec{t}(\vec{y})$ is translated to a coreflexive relation between sequences of terms $K(\vec{t})$, of type $|\vec{t}| \to |\vec{t}|$, tabulated by an arrow $|\vec{y}| \to |\vec{t}|$.

▶ **Definition 13** (Term Translation)**.** The translation function K takes a sequence of terms $\vec{t}$, using $\vec{y} \equiv [y_1, \ldots, y_{|\vec{y}|}]$ variables and returns a coreflexive tabular relation $K(\vec{t}) : |\vec{t}| \to |\vec{t}|$ with tabulation $f : |\vec{y}| \to |\vec{t}|$.

$$\begin{array}{lcl} K(\vec{t}[\vec{y}]) & = & \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n)\rangle^\circ; \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n)\rangle \\ \text{where} & & \\ K_{\vec{y}}(a) & = & !_{|\vec{y}|}; a : |\vec{y}| \to 1 \\ K_{\vec{y}}(y_i) & = & \pi_i : |\vec{y}| \to 1 \\ K_{\vec{y}}(f(t_1, \ldots, t_n)) & = & \langle K_{\vec{y}}(t_1), \ldots, K_{\vec{y}}(t_n)\rangle; f : \alpha(\vec{y}) \to 1 \end{array}$$

The tabulation could be seen as a constructor for $\vec{t}$ from a supply of fresh variables $\vec{y}$. We must wrap the predicates with the relational projection W_i generated from w_i, let $A_i = N - \alpha(p_i)$:

$$K(\vec{t}); W_1; (id_{A_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{A_n} \times \overline{p_n}); W_n^\circ$$

Note that we have replaced $\cup$ by composition. This is possible thanks to the fact that the relation $(id_{A_1} \times \overline{p_1})$ is coreflexive, thus the equation $A \cap B = A; B$ holds. Note that the presented arrow has type $N = |vt|$, while the arrow for the predicate should have a type of M. We use the $I_{MN} : M \to N$ to fix this and we obtain the final form. The final translation for the clause is:

$$I_{MN}; (K(\vec{t}); W_1; (id_{A_1} \times \overline{p_1}); W_1^\circ; \ldots; W_n; (id_{A_n} \times \overline{p_n}); W_n^\circ); I_{MN}^\circ$$

A predicate p consisting of several clauses is then translated using $\cup$:

$$\begin{array}{l} p(\vec{x}) \leftarrow cl_1 \vee \cdots \vee cl_m \quad \to \\ \overline{p} = C_1 \cup \cdots \cup C_m \end{array}$$

where C_i is the arrow corresponding to the translation of the clause cl_i.

▶ **Theorem 14** (Adequacy of the Translation)**.** *Given a predicate p of arity N translated to the arrow $\overline{p} : N \to N$, the initial model maps $\overline{p}$ to the subobject $\xrightarrow{[\![\overline{p}]\!]_A} \mathcal{T}_\Sigma^N$ such that its image is precisely the set of ground terms making p true.*

6 Specification of The Machine

We abuse notation to profit from the fact that a coreflexive relation is uniquely tabulated by a monic $f^\circ; f$ to write f for $f^\circ; f$ when it can be deduced from the context.

We define the categorical machine as a set of transition rules over relations. We write $(f \mid g)$ for tabular relations. Then, $(f \mid g); (f' \mid g')$ is rewrote to $(h; f \mid h'; g')$ using the pullback (h, h') of g, f'. This corresponds to a substitution, where the arrow $h : M \to N$ takes a current state of the machine using N variables to a state using M variables, and $h' : M' \to N'$ does the same, usually instantiating the translations of a clause to the right variables. This mechanism is also used for variable creation/destruction. The pair of arrows (h, h') above the transition arrow denotes the result of the pullback.

A union $R_1 \cup \cdots \cup R_n$ is used to represent disjunctive search, while predicate calls are represented as $(f \mid \langle g, [R]\rangle)$, where R is the relation pertaining to the call in-progress. Note that g and the left tabulation of R share the same domain, allowing the propagation of substitutions resulting from reducing R to the outer context.

$$\begin{array}{llll}
(f \mid g);(f' \mid g') & \xRightarrow{(h,h')} & (h;f \mid h';g') & \\
(f \mid \langle g_K, g_N\rangle);(id_K \times \overline{p_N}) & \Rightarrow & (f \mid \langle g_K, [(g_N \mid g_N);p_1]\rangle) \cup & \\
 & & \vdots \qquad\qquad\qquad\qquad \cup & \\
 & & (f \mid \langle g_K, [(g_N \mid g_N);p_n\rangle]) & \\
(f \mid \langle g, [(g' \mid g')]\rangle) & \Rightarrow & (f \mid \langle g, g\rangle) & \\
(f \mid \langle g, [E]\rangle) & \Rightarrow & (h;f \mid \langle h;g, [E']\rangle) & \text{iff } E \Rightarrow E' \\
R \cup S & \Rightarrow & R' \cup S & \text{iff } R \Rightarrow R' \\
\mathbf{0} \cup S & \Rightarrow & S &
\end{array}$$

The first rule represents composition of tabular relations. The second one represents predicate call. First, disjunctive predicates are unfolded using the rule $f;(R \cup S) = f;R \cup f;S$. Computing the predicate call is performed by the relation $(g_N \mid g_N);p_1$. The third rule deals with return. The three last rules encode the search strategy of the machine.

▶ **Theorem 15** (Operational equivalence). $\langle p_1(\vec{u}_1), \ldots, p_n(\vec{u}_n)\rangle \to \cdots \to \square$ *is the SLD derivation with substitution σ iff*

$$K(\vec{u});W_1;\overline{p_1};W_1^\circ;\ldots;W_n;\overline{p_n};W_n^\circ \Rightarrow K(\sigma(\vec{u})) \cup R$$

7 The Pullback Algorithm

The core of the machine is pullback calculation. We present a pullback calculation algorithm for an arbitrary Regular Lawvere Category $\mathcal{C}$ generated from a signature Σ. The equational theory of $\mathcal{C}$ is the basis for the algorithm.

To improve the presentation, we reduce the pullback problem to its equivalent equalizer formulation. We start with a non commutative diagram and rewrite it until we reach a commutative one, which is an equalizer, and thus we obtain a pullback. The notion of substitution is an arrow composition followed by normalization modulo the product equational theory.

▶ **Definition 16** (Pullback Problem). A pullback problem is given by two arrows $f : N \to M$ and $g : N' \to M$.

▶ **Definition 17** (Arrow Normalization). We write $\to_R^!$ for the associated normalizing relation based on $\to_R$:

$$\begin{array}{lllllll}
h;\langle f, g\rangle & \to_R & \langle h;f, h;g\rangle & \qquad & \langle f, g\rangle;\pi_1 & \to_R & f \\
\langle f, g\rangle;\pi_2 & \to_R & g & & f;!_N & \to_R & !_M \qquad f : M \to N
\end{array}$$

▶ **Definition 18** (Starting Diagram). For a pullback problem, its pre-starting diagram $\mathcal{P}$ is:

$$\begin{tikzcd} N \times N' \arrow[r, shift left, "\pi_1;f"] \arrow[r, shift right, "\pi_2;g"'] \arrow[r, phantom, "\not\vdash"] & M \end{tikzcd}$$

Products are strictly associative, so π_2 is a renaming, for instance if $f = \langle \pi_1 \rangle$ and $g = \langle\langle \pi_1, \pi_2\rangle;f\rangle$, then $\pi_2 : 3 \to 2$ is equal to $\langle \pi_2, \pi_3\rangle$, and $\pi_2;g = \langle\langle \pi_2, \pi_3\rangle;f\rangle$. If $\pi_1;f \to_R^! f'$ and $\pi_2;g \to_R^! g'$, the starting diagram $\mathcal{P}$ is:

$$\begin{tikzcd} N + N' \arrow[r, "id = \langle \pi_1, \ldots, \pi_{N+N'}\rangle"] & N + N' \arrow[r, shift left, "f'"] \arrow[r, shift right, "g'"'] \arrow[r, phantom, "\not\vdash"] & M \end{tikzcd}$$

$N + N'$ is the *type* of the pullback problem.

▶ **Definition 19** (Algorithm State). For a pullback problem of type N, the algorithm state is $(S \mid h)$, $h : N \to N$ an arrow and S an ordered set of equations $f \approx g$ between arrows $f, g : N \to 1$.

▶ **Definition 20** (Auxiliary Substitution). The helper substitution function is $S(i, f : N \to 1, h : N \to N) = h'$, where $\langle \pi_1, \ldots, \pi_{i-1}, f, \pi_{i+1}, \ldots, \pi_N \rangle; h \to^!_R h'$. This function replaces any π_i in h for f.

▶ **Definition 21** (Pullback Calculation Algorithm). The input of the algorithm is two arrows $f_0 : N \to M$ and $g_0 : N' \to M$. First, build the starting diagram $\mathcal{P}$, which produces arrows f'_0 and g'_0, and a type of the problem $N + N' = N_T$. f'_0 and g'_0 are of the form $\langle f_1, \ldots, f_M \rangle$, $\langle g_1, \ldots, g_M \rangle$, then build the initial set $S = \{f_1 \approx g_1, \ldots, f_M \approx g_M\}$. The initial state is $(S \mid \langle \pi_1, \ldots, \pi_{N+N'} \rangle)$. The algorithm proceeds to transform the state $(S \mid h)$ iteratively until $S = \emptyset$ using the following rules

- Pick an equation from S such that $S = \{f \approx g\} \cup S'$. Compute $h; f \to^!_R f'$ and $h; g \to^!_R g'$. Then, do case analysis on $f' \approx g'$:

$$
\begin{array}{lllll@{\qquad}lllll}
!_M; a & \approx & !_M; b & \Rightarrow & Fail & \pi_i & \approx & \pi_j & \Rightarrow & (S' \mid S(j, \pi_i, h)) \\
!_M; a & \approx & h; f & \Rightarrow & Fail & \pi_i & \approx & g; f & \Rightarrow & (S' \mid S(i, g; f, h)) \\
g; f & \approx & g'; f' & \Rightarrow & Fail & !_M; a & \approx & \pi_i & \Rightarrow & (S' \mid S(i, !_M; a, h)) \\
!_M; a & \approx & !_M; a & \Rightarrow & (S' \mid h) & g; f & \approx & g'; f & \Rightarrow & (\{g_1 \approx g'_1\} \cup \ldots \\
 & & & & & & & & & \{g_n \approx g'_n\} \cup S' \mid h)
\end{array}
$$

When $S = \emptyset$, our diagram is commutative but may not be an equalizer due to having an incorrect domain. We create a new arrow from h such that it is a monic. Discarding the K unused elements of M — is enough. Compose $h : M \to M$ with any extension of id_{M-K} to M to obtain $h' : (M - K) \to M$. This process is similar to *garbage collection* and memory de-fragmentation. If the algorithm fails, the equalizer is the initial arrow. Like many actual Prolog implementations, we don't implement occur-check. To get full soundness we would need to implement the occurs check in rule 7.

8 Implementation Discussion

We briefly present the most important points about the efficient implementation of the machine presented in Sec. 6 and Sec. 7. An implementation should be based on the interpretation of projections as pointers, with any π_i appearing inside a term being a pointer to a cell i.

The codomain of the tabulations may be seen as a set of registers, thus, for a pullback between $\langle !_1; f, \pi_1 \rangle$ and $\langle a, b \rangle$, we may assume that the registers are $X_1 = !_1; f$ and $X_2 = \pi_1$ and emit instructions `testc a, X1` and `testc b, X2`.

Note that the model presented here *forces* garbage collection and compaction. Every unused slot is eliminated by the pullback algorithm. We may fix our model by creating $\mathbb{N}$ copies of the object T with their corresponding products. Then, the T_i object becomes a representative of the memory cell i, and the denotational model captures the instantiation of a variable as the variation of the tabulation domain from $(T_1 \times T_2 \times T_3)$ to $(T_1 \times T_3)$. This yields a memory behavior close to a standard WAM without garbage collection.

In order for the code to look reasonable we need to implement two optimizer engines. The first one is an algebraic one and perform tasks like statically computing the tabulation of $I_{MN}; K(\vec{t})$. The second one is a peephole optimizer.

9 Related Work

Algebraic approaches to logic programming have been tried in [29, 3, 20, 2, 16, 4]. The most important difference with our work is that all of them are based on the notion of *indexed category* and don't make a proposal for a concrete implementation. As in our proposal, the use of pullbacks is key point.

A different line of work is interpretation of logic programming as functional programs. The most representative works are [39, 41, 8, 36]. In [7], the authors study relational semantics for lazy functional logic programming language, modeling adequately the interactions between function call and non-determinism. In [10] the authors propose a diagram-based semantics for Logic Programming. An very interesting related work is [34]. This is the only proposal that we know of for the use of tabular allegories in programming. Unfortunately, McPhee's work does not develop an executable model. The use of category theory as a foundational tool for a machine is not new, the best known work is [17].

Several approaches to virtual machine generation [35, 18] and compiler verification [38] for Prolog exist. Several relation-based programming languages exist [19, 14, 13, 12, 25]. In [42], a similar effort to our semantics is developed, but the framework chosen is Tarski's cylindrical algebras instead Freyd's allegories. The author doesn't consider the implementation and efficiency of his approach.

In [1], the authors propose a first-order encoding for allegories. This is related our previous relation rewriting approach and indeed we consider their work very useful for mechanizing our theory. An encoding of allegories in a dependently-typed programming language is presented in [28]. We think Kahl's approach may help us to certify our compiler.

10 Conclusions and Future Work

We have presented an algebraic approach to Logic Programming, from the semantic base of category and allegory theory down to an actual machine based on which can be efficiently implemented. Our approach is new and has important advantages. First, as the algebraic connection between the different layers of the machine is not lost, reasoning in a layer is immediately reflected by the others. Additions on the semantics foster modifications to the algorithm as can be seen in [23]. In the other direction, a good example is the effect that memory layout has on incorporating T_i objects representing memory cells. Second, the correctness of the machine is easy to check. Composition of relations together with the equation $R;(S \cup T) = R;S \cup R;T)$ capture in a simple way the operational semantics and memory layout of Prolog. Our framework is well suited to prove semantic properties, given that our semantics are compositional and use the well established frameworks of category theory and relation algebra. Third, the use of such frameworks favors the reuse of existing technologies in other areas of programming.

We are actively working on an definitive instruction set. We don't want it to be specific to an operational choice like SLD, given that our approach is well suited to accommodate other strategies like breadth-first search. On the other hand, we are already developing extensions to Prolog in [23], and some of them, such as higher-order types may require that we add second primitive of reduction to our machine.

In the future, we will mechanize all the theory presented here, and indeed we hope that effort will bring us close to the goal of having a fully verified implementation. We are working in extending Regular Lawvere Categories to Pre-Logos.

References

1 Bahar Aameri and Michael Winter. A first-order calculus for allegories. In Harrie de Swart, editor, *Relational and Algebraic Methods in Computer Science*, volume 6663 of *Lecture Notes in Computer Science*, pages 74–91. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21070-9_8.

2 Gianluca Amato and James Lipton. Indexed categories and bottom-up semantics of logic programs. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 438–454. Springer, 2001.

3 Gianluca Amato, James Lipton, and Robert McGrail. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, 410(46):4626 – 4671, 2009. Abstract Interpretation and Logic Programming: In honor of professor Giorgio Levi.

4 Andrea Asperti and Simone Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *ICLP*, pages 337–352, 1989.

5 Michael Barr and Charles Wells. *Category theory for computing science (3. ed.)*. Sentre de REcherches Mathématiques, 1999.

6 Francis Borceux. *Handbook of Categorical Algebra 2. Categories and Structures*, volume 51 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.

7 Bernd Braßel and Jan Christiansen. A relation algebraic semantics for a lazy functional logic language. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *RelMiCS*, volume 4988 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2008.

8 Pascal Brisset and Olivier Ridoux. Continuations in lambda-prolog. In *ICLP*, pages 27–43, 1993.

9 Paul Broome and James Lipton. Combinatory logic programming: computing in relation calculi. In *ILPS '94: Proceedings of the 1994 International Symposium on Logic programming*, pages 269–285, Cambridge, MA, USA, 1994. MIT Press.

10 Roberto Bruni, Ugo Montanari, and Francesca Rossi. An interactive semantics of logic programming. *THEORY AND PRACTICE OF LOGIC PROGRAMMING*, 1(6):647–690, 2001.

11 Carsten Butz. Regular categories and regular logic. Technical Report LS-98-2, BRICS, October 1998.

12 Dave Cattrall and Colin Runciman. Widening the representation bottleneck: A functional implementation of relational programming.

13 Dave Cattrall and Colin Runciman. A relational programming system with inferred representations. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *Lecture Notes in Computer Science*, pages 475–476. Springer Berlin / Heidelberg, 1992. 10.1007/3-540-55844-6_156.

14 D.M. Cattrall. *The Design and Implementation of a Relational Programming Systema*. PhD thesis, University of York, 1992.

15 Keith L. Clark. Negation as failure. In Gallaire and Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1977.

16 Andrea Corradini and Andrea Asperti. A categorial model for logic programs: Indexed monoidal categories. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 666 of *Lecture Notes in Computer Science*, pages 110–137. Springer, 1992.

17 Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Sci. Comput. Program.*, 8(2):173–202, 1987.

18 Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Comp. Syst.*, 16(7):739–751, 2000.

19 B. Dwyer. Programming using binary relations: a proposed programming languague. Technical report, University of Adelaide, 1994.

20 Stacy E. Finkelstein, Peter J. Freyd, and James Lipton. A new framework for declarative programming. *Theor. Comput. Sci.*, 300(1-3):91–160, 2003.

21 P.J. Freyd and A. Scedrov. *Categories, Allegories.* North Holland Publishing Company, 1991.

22 Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Constraint logic programming with a relational machine. Technical report, Universidad Politécnica de Madrid, 2012. `http://babel.ls.fi.upm.es/~egallego/iclp/clprm.pdf`.

23 Emilio Jesús Gallego Arias, James Lipton, and Julio Mariño. Extensions to logic programming in tabular allegories: Algebraic data types, functions, constraints and monads. 2012. Submitted to ICLP 2012 `http://babel.ls.fi.upm.es/~egallego/iclp/lpta-ext.pdf`.

24 Emilio Jesús Gallego Arias, James Lipton, Julio Mariño, and Pablo Nogueira. First-order unification using variable-free relational algebra. *Logic Journal of IGPL*, 19(6):790–820, 2011.

25 Patrick A. V. Hall. Relational algebras, logic, and functional programming. In Beatrice Yormark, editor, *SIGMOD Conference*, pages 326–333. ACM Press, 1984.

26 Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. *Electr. Notes Theor. Comput. Sci.*, 172:437–458, 2007.

27 Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium.* Oxford University Press, 2003.

28 Wolfram Kahl. Dependently-typed formalisation of relation-algebraic abstractions. In Harrie de Swart, editor, *Relational and Algebraic Methods in Computer Science*, volume 6663 of *Lecture Notes in Computer Science*, pages 230–247. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21070-9_18.

29 Yoshiki Kinoshita and A. John Power. A fibrational semantics for logic programs. In Roy Dyckhoff, Heinrich Herre, and Peter Schroeder-Heister, editors, *ELP*, volume 1050 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 1996.

30 J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

31 F. William Lawvere. *Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the context of Functorial Semantics of Algebraic Theories.* PhD thesis, Columbia University, 1968.

32 Jim Lipton and Emily Chapman. Some notes on logic programming with a relational machine. In Ali Jaoua, Peter Kempf, and Gunther Schmidt, editors, *Using Relational Methods in Computer Science*, Technical Report Nr. 1998-03, pages 1–34. Fakultät für Informatik, Universität der Bundeswehr München, July 1998.

33 J. W. Lloyd. *Foundations of logic programming.* Springer-Verlag New York, Inc., New York, NY, USA, 1984.

34 Richard McPhee and Richard Mcphee. Towards a relational programming language, 1995.

35 José F. Morales, Manuel Carro, Germán Puebla, and Manuel V. Hermenegildo. A generator of efficient abstract machine implementations and its application to emulator minimization. In Maurizio Gabbrielli and Gopal Gupta, editors, *ICLP*, volume 3668 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2005.

36 Maciej Pirog and Jeremy Gibbons. A functional derivation of the warren abstract machine. 2011. Submitted for publication.

37 Vaughan R. Pratt. Origins of the calculus of binary relations. In *Logic in Computer Science*, pages 248–254, 1992.

38 David M. Russinoff. A verified prolog compiler for the warren abstract machine. *Journal of Logic Programming*, 13:367–412, 1992.

39 Silvija Seres, J. Michael Spivey, and C. A. R. Hoare. Algebra of logic programming. In *ICLP*, pages 184–199, 1999.

40 Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*, volume 41 of *Colloquium Publications*. American Mathematical Society, Providence, Rhode Island, 1987.

41 Eneia Todoran and Nikolaos S. Papaspyrou. Continuations for parallel logic programming. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 257–267, New York, NY, USA, 2000. ACM.

42 Maarten H. van Emden. Compositional semantics for the procedural interpretation of logic. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2006.