Logic Programming in Tabular Allegories

Emilio Jesús Gallego Arias*
Universidad Politécnica de Madrid

James B. Lipton
Wesleyan University

submitted 2012; revised 2012; accepted 2012

Abstract
We develop a new compilation scheme and categorical abstract machine for execution of logic programs based on allegories, the categorical version of the calculus of relations. In our case, both operational and denotational semantics are built upon the same theory, thus execution of a query is performed using algebraic reasoning, giving immediate proof of correctness properties. Our work serves two purposes: having an efficient and formal model of a logic programming compiler and machine and building the base for incorporating features typical of functional programming in a declarative way, while maintaining 100% compatibility with existing Prolog programs.

The starting point is the construction of a Regular Lawvere Category from the program’s signature $\Sigma$. Then, the resulting $\Sigma$-allegory captures all the needed theory and meta-theory for Prolog. The categorical framework for reasoning and execution is based on relation composition, thus allowing the enrichment of the base category with types, functions and other constructions typical of functional programming languages.

The single primitive of relation composition encompasses unification, garbage collection, parameter passing, environment creation and destruction. Composition happens between relations tabulated by a pair of maps, whose domain represents the global storage in use while its co-domain represents the number of temporary registers in use. This execution mechanism is efficient, given that shared state is faithfully represented by the notion of projections, corresponding to pointers in the implementation.


1 Introduction
Relational algebras have a broad spectrum of application in both theoretical and practical computer science. In particular, the calculus of binary relations \cite{Pratt1992}, whose main operations are intersection ($\cup$), union ($\cap$), complement $/$, and relation composition ($;$) was shown by Tarski and Givant \cite{Tarski1987}.

* The author wants to acknowledge Wesleyan University for supporting him with Van Vleck funds. This work is part of DESAFIOS10 (TIN2009-14599-C03-00)
to be a complete and adequate model for capturing all first-order logic and set theory in a variable-free manner. The intuition is that conjunction is modeled by $\cup$, disjunction by $\cup$ and existential quantification by composition.

We think this correspondence is very useful for modeling logic programming. It is natural to interpret logic programs as binary relations, and relational algebra provides a suitable framework for algebraic reasoning over them, including execution of queries. In previous work by the authors (Gallego Arias et al. 2011; Lipton and Chapman 1998; Broome and Lipton 1994; Gallego Arias et al. 2012a), we have developed operational and denotational semantics for constraint logic programming using distributive relational algebra with a quasi-projection operator. Pure relational algebraic semantics have very pleasant properties, but individual relations are assumed to range over the same domain, forcing our models to be the union of the set of all finite sequences of terms generated by the signature of the program.

Having an efficient relational machine for the execution of logic programs becomes difficult with this approach. When a predicate call happens, we must split the constraint store in two — the one belonging to the caller environment and the one needed by the called predicate — and merge back the results at return time. Propagating constraint operations that happen inside a procedure call to the outer context is delayed.

We propose to remedy this shortcoming by using typed relations. The theory of allegories (Freyd and Scedrov 1991), provides a categorical setting for distributive relational algebras. In this setting, relations are typed and the semantics for our relations can now be built with 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) $f : C \rightarrow A \times B$ if every pair of the relation is in its image. We may split $f$ into its components $f; \pi_1 : C \rightarrow A$ and $f; \pi_2 : C \rightarrow B$, 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 have an important meaning in the implementation. 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 the a very convenient algebraic style, immediately witnessing the impact of such reasoning on the machine itself. Our philosophy is that in a fully algebraic framework, efficient execution should admit logical 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 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 (Gallego Arias et al. 2012b), using the foundation laid out in this paper, we enrich our categories, adding algebraic data types, constraints, functions and monads to Prolog, all of it without losing source code compatibility with existing programs.

1.1 Structure Of The Paper

We start by reviewing Logic Programming in Sec. 2. In Sec. 3 we introduce the necessary definitions from Category Theory. In Sec. 4 we define Regular Lawvere Categories and their associated distributive allegories. Sec. 5 presents the compilation procedure of a logic program, which amounts to translating predicates to arrows in a Σ-allegory.

In Sec. 6 we introduce our notion of categorical computation, and present an algebraic specification of the categorical machine. The algorithm for pullback calculation, which is the core of the machine is presented separately in Sec. 7. Finally, we discuss related work in Sec. 9 and detail future work and conclusions in Sec. 10.

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 $Σ = ⟨C_Σ, F_Σ⟩$, given by $C_Σ$, the set of constant symbols, and $F_Σ$, the set of term formers or function symbols. $P$ will denote the set of predicate symbols. Function $α : P ∪ F_Σ → \mathbb{N}$ returns the arity of its predicate argument. We assume a set $X$ of so-called logic variables whose members are denoted $x_i$. We write $T_Σ$ for the set of closed terms over $Σ$. We write $T_Σ(X)$ for the set of open terms (in the variables in $X$) over $Σ$. We drop $Σ$ when understood from context. We write sequences of terms using vector notation: $t = t_1, \ldots, t_n$. The length of such a sequence is written $|t| = n$. We assume standard definitions for atoms, predicates, programs and clauses, see (Lloyd 1984).
Operational Semantics: A program state is a sequence of atoms \((A_1, \ldots, A_n)\). We write \(\Box\) for the empty state. Proof search in Logic Programming system is performed by a basic transition, the resolution transition.

A resolution step for a state \(\langle p_1(\vec{u}_1), \ldots, p_n(\vec{u}_n) \rangle\) using a renamed apart clause \(cl: p_1(\vec{v}) \leftarrow q_1(\vec{t}_1), \ldots, q_m(\vec{t}_m)\) is:

\[
\langle p_1(\vec{u}_1), \ldots, p_n(\vec{u}_n) \rangle \xrightarrow{\sigma} \langle \sigma(q_1(\vec{t}_1), \ldots, q_m(\vec{t}_m), p_2(\vec{u}_2), \ldots, p_n(\vec{u}_n)) \rangle
\] (1)

iff \(\sigma(\vec{v}) = \sigma(\vec{u}_1)\) and exists a most general unifier \(\sigma\). A derivation is a sequence of resolution steps, with a successful one ending in \(\Box\). The SLD-derivation is the first successful one by choosing the clauses in order of appearance.

3 Category Theory

A category \(C = \langle O, A \rangle\) consists of a collection of objects \(O\) and typed arrows \(A\). For every object \(A \in O\), there is an identity arrow \(id_A: A \to A \in A\). 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 \(id_A; f = f = f; id_B\). We assume knowledge of the concepts of commutative diagram, product, equalizer, pullback, monic arrow and subobject (Borceux 1994; Barr and Wells 1999; Lambek and Scott 1986).

For a product \(A \times B\), we will write \(\pi_{A \times B}^1: A \times B \to A\) and \(\pi_{A \times B}^2: A \times B \to B\) for the projections. For arrows \(f: C \to A\), \(g: C \to B\) we write \(\langle f, g \rangle\) for the unique product former.

3.1 Regular Categories

Several definitions exist for Regular Categories (Butz 1998; Borceux 1994; Johnstone 2003; Freyd and Scedrov 1991); we use the latter presentation.

Definition 1 (Image)
If \(B'\) is an object, we say \(B'\) allows \(f: A \to B\) if \(f\) factors through \(B'\), i.e. if there are arrows \(s: A \to B'\) and \(t: B' \to B\) such that \(s; t = f\). The image of \(f: A \to B\), if it exists, is the smallest subobject that allows \(f\).

Definition 2 (Cover)
An arrow \(f: A \to B\) is a cover if its image is entire (i.e. an isomorphism) We denote covers by \(A \twoheadrightarrow B\). Every cover is an epimorphism but the converse is not true.

Definition 3 (Regular Category)
We say \(C\) is a Regular Category if it has products, equalizers, images and pullbacks transfer covers. A Regular Category can be used to generate a tabular allegory. In this way, Regular Categories give rise to categories of relations.

3.2 Categorical Relations
Definition 4 (Monic Pair)

$f : C \to A$ and $g : C \to B$ is a monic pair iff $(f, g) : C \to A \times B$ is monic.

A monic pair $(f, g)$ is a subobject of $A \times B$, thus it is a relation from $A$ to $B$:

\[
\begin{array}{c}
\downarrow \quad \downarrow \\
A & \quad & B \\
\end{array}
\]

Definition 5 (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 is given by a monic pair. The right diagram shows the already composed relation. This diagram rewriting is the principal rule of our categorical machine.

Lemma 1
Composition of relations is associative iff the category is regular.

Proof
See Johnstone [Johnstone 2003] or Freyd [Freyd and Scedrov 1991].

Definition 6 (Categories of Relations)
For a regular category $C$, the category $\text{Rel}(C)$ of relations has the same objects as $C$, arrows $A \to B$ are monic pairs $(f : C \to A, g : C \to B)$ and composition is defined as above. $C$ is a sub-category of $\text{Rel}(C)$. The inclusion functor sends an arrow $f : A \to B$ to the pair $(id, f)$. If a morphism of $\text{Rel}(C)$ is in $C$, we call it a map.

Given a relation $(f, g)$, its reciprocal is $(g, f)$. The natural order-isomorphism $\text{Sub}(A \times B) \approx \text{Rel}(A, B)$ yields a semi-lattice structure on $\text{Rel}(A, B)$. 
3.3 Lawvere Categories

A Lawvere category is a category $C$ with a denumerable set $\{T^0, T^1, \ldots, T^n, \ldots\}$ of distinct objects, where each object $T^n$ is the $n$-th power of the object $T^1$. We will write $i$ for $T^i$.

$0$ 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 $f : 2 \to 2$, $(f \times id_2) = (f; \pi_1, f; \pi_2, id_1, id_1)$, etc.

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

- For every constant $a \in T_0$, we freely adjoin an arrow $a : 0 \to 1$.
- For every function symbol $f \in T_2$ with arity $\alpha(f) = N$, we freely adjoin an arrow $f : N \to 1$.

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

Lawvere Categories are a natural framework for categorically representing algebraic theories. Examples of such categories $C$ may be seen in [Lawvere 1968], and some good treatments are in [Borceux 1994; Hyland and Power 2007].

3.4 Allegories

Definition 7 (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:

\[
\begin{align*}
R \cap R &= R \\
R \cap (S \cap T) &= (R \cap S) \cap T \\
(RS)^\circ &= S^\circ ; R^\circ \\
R; (S \cap T) &\subseteq (R; S \cap R; T) & (R; S \cap T) &\subseteq (R \cap T; S^\circ ; S)\end{align*}
\]

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 $\text{Map}(\mathcal{R})$.

We say 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.
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 $R^o = g^o \circ f$.

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 coincide:

**Lemma 2**

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

**Proof**

See [Freyd and Scedrov 1991] 2.147 and 2.148, 2.154.

Composition of relations in a tabular allegory is thus defined in the same way than for categories of relations arising from a Regular Category. See Def. 5.

A distributive allegory is an allegory with a new relation denoted $0_{AB}$ 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:

\[
\begin{align*}
R \cup R &= R & R \cup S &= S \cup R \\
R \cup (S \cup T) &= (R \cup S) \cup T & 0 \cup S &= S \\
R \cup (R \cap S) &= R & R; 0 &= 0 \\
R(S \cup T) &= RS \cup RT & R \cap (S \cup T) &= (R \cap S) \cup (R \cup T)
\end{align*}
\]

4 Regular Lawvere Categories and $\Sigma$-Allegories

The key idea in order to build our categorical semantics is to visualize a Lawvere Category in terms of $Map(\mathcal{R})$ of a tabular allegory. Our approach can be understood as the convergence of two different efforts:

- Building a typed version of the work with untyped relations in [Gallego Arias et al. 2012a]. Just converting our algebraic theory to a category gives rise to a $\Sigma$-allegory $\mathcal{R}$, where $Map(\mathcal{R})$ is a Regular Lawvere Category.
- Performing a regular completion of Lawvere Categories similar to the ones used as a base category for the indexed category approach [Amato et al. 2009].

In this paper, we build a Regular Lawvere Category $\mathcal{C}$ first, then $Rel(\mathcal{C})$ generates a pre-$\Sigma$-allegory, which is then $\cup$-completed in order to yield a proper $\Sigma$-allegory.

**Definition 8 (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 $?_A : \bot \rightarrow A$ for every object $A$ and applying the quotient $?_A; f = ?_B$ for any arrow $f : A \rightarrow B$.

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 9 (Initial Model)

Given a choice \( \langle \cdot, \rangle \) of product in \( \text{Set} \), and a choice of symbols for the signature \( \Sigma \) generating the Regular Lawvere Category \( \mathcal{C} \) and set \( T_\Sigma \), the initial model of a Regular Lawvere Category \( \mathcal{C} \) — that is to say, the initial object in \( \text{Mod}(\mathcal{C}, \text{Set}) \) — is the functor \( \llbracket \cdot \rrbracket \), with object and arrow components \( (\llbracket \cdot \rrbracket_O, \llbracket \cdot \rrbracket_A) \):

\[
\begin{align*}
\llbracket \bot \rrbracket_O &= \emptyset \quad \llbracket 0 \rrbracket_O = \{\bullet\} \quad \llbracket N \rrbracket_O = T_\Sigma^N \quad N > 0 \\
\llbracket ?N \rrbracket_A &= \emptyset \xrightarrow{\emptyset} \llbracket N \rrbracket_O \\
\llbracket !N \rrbracket_A &= \lambda x. \bullet \\
\llbracket c : 0 \to 1 \rrbracket_A &= \lambda \bullet. c \\
\llbracket f : N \to 1 \rrbracket_A &= \lambda(n_1, \ldots, n_N). f(n_1, \ldots, n_N) \\
\llbracket \pi_i : N \to 1 \rrbracket_A &= \lambda(n_1, \ldots, n_N). n_i \\
\llbracket (t_1, \ldots, t_N) : M \to N \rrbracket_A &= \lambda n. (\llbracket n \rrbracket_A; \llbracket t_1 \rrbracket_A, \ldots, \llbracket n \rrbracket_A; \llbracket t_N \rrbracket_A)
\end{align*}
\]

Lemma 3

The regular completion of a Lawvere Category is indeed a Regular Category.

Proof

Every pair of arrows has an equalizer, thanks to the existence of the initial object. Every arrow has an image as can be checked inductively over the set of arrows. Pullbacks transfer covers as can be checked case by case. The only covers are the identity arrows and the projections. \( \square \)

4.1 \( \Sigma \)-Allegories

Regular Lawvere Categories are not enough to model disjunctive clauses in logic programs, as they don't tabulate distributive allegories. A distributive allegory is tabulated by a Pre-Topos \( \text{Freyd and Scedrov 1991} \), which is a regular category whose subobjects form a complete lattice, not just a semi-lattice.

Definition 10 (\( \Sigma \)-Allegory)

Given a Regular Lawvere Category \( \mathcal{C} \), we define a \( \Sigma \)-allegory \( \mathcal{R}_\cup \) as the distributive allegory generated from the allegory \( \mathcal{R} \approx \text{Rel}(\mathcal{C}) \) by freely adding all union arrows and taking the quotient by the distributive laws. This means that an inclusion functor \( F : \mathcal{R} \to \mathcal{R}_\cup \) exists. Recall that all the arrows in \( \mathcal{R} \) are tabular, thus, it is easy to see that all the arrows in \( \mathcal{R}_\cup \) that possess a union-free representation are tabular.

This compromise means that the conjunctive part of resolution is handled by pullbacks of tabulations whereas backtracking is controlled by pure allegorical laws.

5 Translation of the Program

The compilation procedure and interpretation for a given logic program coincide. The method is almost identical to the one used in \( \text{Gallego Arias et al. 2012a} \), and
in order to help the reader we have included an actual example in Appendix A. First, we complete every predicate in a similar way to Clark’s (Clark 1977). Every term occurring as an argument in the head and tail is assigned to a new variable \( x_i \). The set of \( n \) variables occurring in the terms is renamed from \( y_1 \) to \( y_n \). After that process, clauses are of the form:

\[
p(\vec{x}') \leftarrow \vec{x} = \vec{\bar{y}}(\vec{y}), p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n).
\]

\( \vec{x}' \) a prefix of \( \vec{x} \), \( \vec{x}_i \) a selection of variables in \( \vec{x} \) and \( \vec{y} \) 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:

\[
p(\vec{x}') \leftarrow \vec{x} = \vec{\bar{y}}(\vec{y}), p_1(w_1(\vec{x})), \ldots, p_n(w_n(\vec{x})).
\]

The main idea behind the relational translation is to interpret a predicate by the coreflexive relation generated by the set of ground terms which make it true. Using this scheme, \( \land \) corresponds to intersection of relations. We first build a coreflexive relation between sequences of terms \( K(\vec{t}) \), of type \( I \rightarrow \vec{I} \). Such relation, is tabulated by an arrow \( |\vec{g}| : |\vec{t}| \rightarrow |\vec{l}| \), that is to say, it constructs the terms from a supply of fresh variables corresponding to \( \vec{g} \). Then we wrap the predicates into a relational projection \( W \) generated from \( w_i \). The candidate translation is:

\[
\vec{p} = K(\vec{t}) \cap W_1; \vec{p}_1; W_1^\circ \cap \cdots \cap W_n; \vec{p}_n; W_n^\circ\]

We are close to the final form, however we must solve two remaining technical problems. First, as \( |\vec{x}'| \leq |\vec{x}| \), the type of the above term is not the correct one for \( \vec{p} \). Define \( M = |\vec{x}'|, N = |\vec{x}| \). Then, we introduce a partial identity relation \( I_{MN} : M \rightarrow N \) and wrap the clause with it. When composed with a vector of type \( M \), its codomain will represent a vector of size \( N \) and behave as an identity on first \( M \) elements. For instance, \( I_{12} \) is \( \pi_1^2 : 1 \rightarrow 2 \), whose set-theoretic semantics is \( \forall a, b \in T_S,(a, b) \in [I_{12}] \). The final translation for the clause is:

\[
I_{MN}; (K(\vec{t}) \cap W_1; \vec{p}_1; W_1^\circ \cap \cdots \cap W_n; \vec{p}_n; W_n^\circ); I_{MN}^\circ
\]

\( I_{MN} \) plays the role of environment creation and destruction. The second technical problem we face is that \( W_1; \vec{p}_1; W_1^\circ \) is not in general a coreflexive relation. If \( R, S \) are coreflexive \( R \cap S = R \circ S \), a transformation highly convenient for our purposes. Say \( A_i = N - \alpha(p_i) \), then we can replace the projections \( W \) by permutations and replace \( ; \) for \( \cup \). The final translation of a clause is the arrow of the \( \Sigma \)-allegory:

\[
I_{MN}; (K(\vec{t}); W_1; (id_{A_1} \times \vec{p}_1); W_1^\circ; \cdots ; W_n; (id_{A_n} \times \vec{p}_n); W_n^\circ); I_{MN}
\]

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

\[
p(\vec{x}) \leftarrow c_1 \cup \cdots \cup c_m \rightarrow \vec{p} = C_1 \cup \cdots \cup C_m
\]

where \( C_i \) is the arrow corresponding to the translation of the clause \( c_i \).

**Definition 11 (Term Translation)**

The translation function \( K \) takes a sequence of terms \( \vec{t} \), using \( \vec{y} = [y_1, \ldots, y_{|\vec{y}|}] \).
Definition 12 (I Relation)
The relation $I_{MN}$, with $M < N$ is tabulated by $(\langle \pi_1, \ldots, \pi_M \rangle, \text{id}_N)$. See Fig. 2.

We may interpret $I_{MN}$ as a creator — and its converse a destroyer — of local variables. This relation formalizes the intuition that the reciprocal of a projection creates a new variable, indeed $I_{12} = \pi^T_1$. As we will see, every $I_{MN}$ relation is immediately followed by the term instantiation.

Definition 13 (W Relation)
For a projection $w : N \rightarrow M$, with $N \ge M$ and $K = N - M$, we denote by $w' : N \rightarrow N$ any of its extensions to a permutation such that the following equations are satisfied: \[ w'(K) = w^{-1}(1), \ldots, w'(K+M) = w^{-1}(M) \]. Having fixed a $w'$, $W$ is tabulated by $(N, \langle \pi_{w'\left(1,\ldots,M\right)} \rangle)$. See Fig. 2.

Basically the $W$ relation puts the parameters needed by a predicate in the right order and in the last position of the vector. We don’t care about the order of the other elements as they don’t play a role.

Theorem 1 (Adequacy of the Translation)
Given a predicate $p$ of arity $N$ translated to the arrow $p : N \rightarrow N$, the initial model maps $\bar{p}$ to the subobject $\Sigma^N \xrightarrow{\bar{p}} T'_{\Sigma^N}$ such that its image is precisely the set of ground terms making $p$ true.

Proof
For each non-recursive clause, is easy to check that the semantics of its translation are the relations such that $p(\vec{t})$ iff $\vec{t} \in [\bar{p}]$. For the recursive case, assign $[\bar{p}]_A = \emptyset$ and the fixpoint of the interpretation coincides with the result of the van Emden-Kowalski fixpoint operator. \qed
6 Specification of The Machine

We present an algebraic specification of the allegorical machine for logic programming. It is based on diagram rewriting such as the one seen in Fig. 1, which corresponds to the first rule of the machine.

Motivation and Basic Principles In our previous non-categorical version (Gallego Arias et al. 2012a), lack of typing information and tabulations forced us to duplicate the constraint store on predicate call:

\[
\hat{K}(\varphi) \cap I_m; R; I^0_m \rightarrow I_m; (I_m; \hat{K}(\varphi); I^0_m \cap R); I^0_m \cap \hat{K}(\varphi)
\]

where \(I_m\) is \(\bigcup_{N \geq M} I_{MN}\). In this context, the only available tool for reasoning is the modular law, which while highly convenient from an algebraic point of view, forces the partial duplication of the relation \(\hat{K}(\varphi)\), which amounts to unnecessary duplication and delay of substitutions happening in the inner context. This can be fixed using category theory, so we have the needed structure to abandon the modular law and derive a more efficient rule, and by making the relation that wraps the predicates coreflexive.

Thus, we convert \(I^0_{MN}; \overline{p}; I_{MN}\) to the coreflexive relation \((id_{N-M} \times \overline{p})^\circ; (id_{N-M} \times \overline{p})\). It is safe to do this as in our translation scheme a call to \(p\) will always occur intersected with a coreflexive relation:

\[
\hat{K} \cap I_{MN}; \overline{p}; I^0_{MN} = (\hat{K} \cap id) \cap I_{MN}; \overline{p}; I^0_{MN} = \hat{K} \cap (id \cap I_{MN}; \overline{p}; I^0_{MN})
\]

indeed:

\[
id \cap I_{MN}; \overline{p}; I^0_{MN} = (id_{N-M} \times \overline{p})^\circ; (id_{N-M} \times \overline{p})
\]

Note that we abuse notation and we 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 is clear from the context. The wrapping in an environment of type \(N\) for \(\overline{p} : P \rightarrow M\), given \(K = N - M\) is:

\[
\begin{array}{c}
K \times P \\
(id_K \times \overline{p})
\end{array}
\]

\[
\begin{array}{c}
N \\
(id_K \times \overline{p})
\end{array}
\]

As we will see, this is fundamental for the correct behavior of the machine, the first \(K\) slots will maintain the link with the unfolding of \(\overline{p}\).

6.1 The Machine

We define transition rules for the machine as a diagram rewriting system. The basic diagrams are a tabulation \((f \mid g)\), a union \(R_1 \cup \cdots \cup R_n\) and a procedure call diagram
\( (f \mid (g, [R])) \). A pair of arrows above the transition indicates the pullback result:

\[
(f \mid g); (f' \mid g') \xrightarrow{(h\cdot h')} (h; f \mid h'; g) \\
(f \mid (g_K, g_N)); (id_K \times p_N) \Rightarrow (f \mid (g_K, [g_N; p_1])) \cup \\
\vdots \cup \cup \\
(f \mid (g_K, [g_N; p_n])) \\
(f \mid (g, [(g' \mid g')]))) \Rightarrow (f \mid (g, g)) \\
(f \mid (g, [E])) \Rightarrow (h; f \mid (h; g, [E']))) \text{ iff } E \Rightarrow E' \\
R \cup S \Rightarrow R' \cup S \text{ iff } R \Rightarrow R' \\
0 \cup S \Rightarrow S
\]

**Theorem 2 (Operational equivalence)**

\( \langle p_1(\vec{u}_1), \ldots, p_n(\vec{u}_n) \rangle \rightarrow \cdots \rightarrow \square \) is the SLD derivation with substitution \( \sigma \) iff

\[
K(\vec{u}); W_1; p_1; W_1; \ldots; W_n; p_n; W_n \Rightarrow K(\sigma(\vec{u})) \cup R
\]

**Proof**

Fix an increasing renaming scheme for the transition in Eq. (1). It is easy to show by induction that a resolution transition corresponds to a relational transition. The SLD strategy corresponds to our choice of always executing first the left part of a union, where incomplete derivations correspond to \( 0 \).

\( \square \)

### 7 The Pullback Algorithm

The core of the machine is pullback calculation. We present a pullback calculation algorithm for an arbitrary Regular Lawvere Category \( C \) generated from a signature \( \Sigma \). For convenience reasons, the basis of our algorithm is non-commutative diagram rewriting. Our notion of diagram coincides with [Barr and Wells 1999], it is a finite subcategory of \( C \). The equational theory of \( C \) is the basis for our computation procedure.

In order 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 diagram rewriting used in our algorithm doesn’t change the shape of the diagram. Our notion of substitution is arrow composition followed of normalization modulo the product equational theory.

**Definition 14 (Pullback Problem)**

A pullback problem is given by two arrows \( f : N \to M \) and \( g : N' \to M \).

**Definition 15 (Arrow Normalization)**

We write \( \to_R^1 \) for the associated normalizing relation based on \( \to_R \):

\[
\begin{align*}
\langle f, g \rangle & \rightarrow_R \langle h; f, h; g \rangle \\
(f, g); \pi_1 & \rightarrow_R f \\
(f, g); \pi_2 & \rightarrow_R g \\
N \rightarrow_R M & f : M \to N
\end{align*}
\]
Definition 16 (Starting Diagram)
For a pullback problem, build the pre-starting diagram $\mathcal{P}$:

$$
\begin{array}{c}
N \times N' \xrightarrow{\pi_1; f} M \\
\downarrow \pi_2; g \\
M
\end{array}
$$

where the puncture mark indicates possible failure of commutativity. Given that products are strictly associative $\pi_2$ behaves as renaming operation, see the example $f = \langle \pi_1 \rangle$, $g = \langle \langle \pi_1, \pi_2 \rangle; f \rangle$, 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 $\pi_1; f \to^1_R f'$ and $\pi_2; g \to^1_R g'$, the starting diagram $\mathcal{P}$ is:

$$
\begin{array}{c}
N + N' \xrightarrow{id = \langle \pi_1, \ldots, \pi_{N+N'} \rangle} N + N' \xrightarrow{f'} M \\
\downarrow g' \\
M
\end{array}
$$

We call $N + N'$ the type of the pullback problem.

Definition 17 (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 18 (Auxiliary Substitution)
The helper substitution function is $S(i, f : N \to 1, h : N \to N) = h'$, where $\langle \pi_1, \ldots, \pi_{i-1}, f, \pi_{i+1}, \ldots, \pi_N \rangle; h \to^1_R h'$. This function replaces any $\pi_i$ in $h$ for $f$.

Definition 19 (Pullback Calculation Algorithm)
The input of the algorithm is two arrows $f : N \to M$ and $g : N' \to M$. Build the starting diagram $\mathcal{P}$, which produces arrows $f'$ and $g'$, and a type of the problem $N + N' = M$.

$f'$ and $g'$ are of the form $\langle f_1, \ldots, f_M \rangle$, $\langle g_1, \ldots, g_M \rangle$, build the initial set $S = \{f_1 \approx g_1, \ldots, f_M \approx g_M\}$. The initial state is $(S \mid \langle \pi_1, \ldots, \pi_{N+N'} \rangle)$. Transform the state $(S \mid h)$ iteratively until $S = \emptyset$ as follows:

- Pick an equation from $S$ such that $S = \{f \approx g\} \cup S'$. Compute $h; f \to^1_R f'$ and $h; g \to^1_R g'$. Then, do case analysis on $f' \approx g'$:

  \[
  \begin{array}{l}
  !_M; a \approx !_M; b \Rightarrow \text{Fail} \\
  !_M; a \approx h; f \Rightarrow \text{Fail} \\
  g; f \approx g'; f' \Rightarrow \text{Fail} \\
  !_M; a \approx (S' \mid h) \Rightarrow \text{Fail} \\
  !M; a \approx \langle S' \mid h \rangle \\
  g; f \approx g' ; f' \Rightarrow (\langle g_1 \approx g'_1 \rangle \cup \ldots \langle g_n \approx g'_n \rangle \cup S' \mid h)
  \end{array}
  \]

When $S = \emptyset$, our diagram is commutative but may not be an equalizer due to having a domain which is “too big”. Adjusting $h$ to be a monic arrow — that is to say, discarding the $K$ unused elements of $M$ — is enough to solve the problem. 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 enforce the occurs check, and in that case the resulting diagram won’t be a pullback. To get full soundness we would need to implement the occurs check in rule 7.

**Lemma 4**

$h' : M' \rightarrow M$ is an equalizer of the original diagram.

## 8 Implementation Discussion

We briefly present the most important points about the efficient implementation of the machine presented in Sec. 6 and Sec. 7. As previously said, the implementation is based in the interpretation of projections as pointers. That way, any $\pi_i$ appearing inside a term is a pointer to cell $i$. Together with a `deref` operation, this view fully captures the arrow normalization system that implements substitution in our pullback algorithm.

The set of equations $S$ defined in the algorithm is directly implemented as checks against a set of registers. For a pullback between $(1; f, \pi_1)$ and $(a, b)$, we assume that the registers are $X_1 = 1; f$ and $X_2 = \pi_1$ and emit the instructions `testc a, X1` and `testc b, X2`. For a compound term, we assume the existence of a stack of registers. We may even have a counter and omit the second parameter to `testc`. We have two memory zones, one for variables and one for registers. Currently we choose to allocate terms in the register space, but this may change in the future.

Note that the model presented here forces garbage collection and compaction, as every unused slot is eliminated by the pullback algorithm. We may fix our model by creating 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 $I_{MN}; K(i)$. The second one is a peephole optimizer.

## 9 Related Work

Algebraic approaches to logic programming have been tried in (Kinoshita and Power 1996; Amato et al. 2009; Finkelstein et al. 2003; Amato and Lipton 2001; Corradini and Asperti 1992; Asperti and Martini 1989). 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 (Seres et al. 1999; Todoran and Papaspyrou 2000; Brisset and Ridoux 1993; Pirog and Gibbons 2011).

In (Braßel and Christiansen 2008), the authors study relational semantics for
logic programming language, modeling adequately the interactions between function call and non-determinism.

In (Bruni et al. 2001) the authors propose a diagram-based semantics for Logic Programming. An very interesting related work is (McPhee and Mcphee 1995). 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 (Cousineau et al. 1987).

Several approaches to virtual machine generation (Morales et al. 2005; Diehl et al. 2000) and compiler verification (Russinoff 1992) for Prolog exist.


In (van Emden 2006), 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 (Aameri and Winter 2011), 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 (Kahl 2011). 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 foundation of category and allegory theory down to an actual machine 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 to the semantics induce modifications to the algorithm as can be seen in (Gallego Arias et al. 2012b). In the other direction, a good example is the effect that memory layout have 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 a definitive instruction set. We don’t want it to be specific to an operational choice like SLD resolution, 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 (Gallego Arias et al. 2012b), and some of them, such as higher-order types, may require that we add a second primitive of reduction to our machine.

In the future, we expect to mechanize all the theory presented here, and indeed we hope that effort will bring us close to the goal of having a fully verified imple-
agement. We are also working on extending the application of Regular Lawvere Categories given here to Pre-Logoi.

References


Logic Programming in Tabular Allegories


Appendix A An Example

We will use as example the classical add predicate implementing Peano addition:

\[
\begin{align*}
\text{add} & \left( o, X, X \right). \\
\text{add} & \left( s(X), Y, s(Z) \right) :- \text{add} \left( X, Y, Z \right).
\end{align*}
\]

A.1 Translation

We perform the renaming procedure similar to Clark’s completion:

\[
\begin{align*}
\text{add} \left( X_1, X_2, X_3 \right) & :- X_1 = o, X_2 = Y_1, X_3 = Y_1. \\
\text{add} \left( X_1, X_2, X_3 \right) & :- X_1 = s(Y_1), X_2 = Y_2, X_3 = s(Y_3), X_4 = Y_1, X_5 = Y_3, \\
& \quad \text{add} \left( X_4, X_2, X_5 \right).
\end{align*}
\]

Note that we have two kinds of variables, the ones starting by X which may only appear as arguments to predicates and the Y variables, which represent the “real” variables used inside the predicate. Externally, add only uses three X variables, but internally it needs two more. In our relational translation, we will capture this fact by using a relation \( I_{35} : 3 \rightarrow 5 \) that takes care of creating \( X_4 \) and \( X_5 \). Recall that \( \langle f, g \rangle \) is the categorical product constructor. Then, storing all our X variables in such a product, we may try to express add in a relational pseudo-notation:

\[
\begin{align*}
\text{add} & = \langle o, Y_1, Y_1 \rangle \\
& \quad \cup \ I_{35}; (\langle s(Y_1), Y_2, s(Y_3), Y_1, Y_3 \rangle \cap (id_2 \times \text{add}); I_{35}^o)
\end{align*}
\]

the recursive call to \( \text{add} \) is wrapped into a vector of size 5, but we are calling it with the wrong parameters! The above expression is equivalent to \( \text{add}(X_3, X_4, X_5) \). We need to call it with the right parameters, so we compose the call with a permutation of the vector. We replace Y variables by categorical projections and the actual translation is:

\[
\begin{align*}
\text{add} & = \langle o, \pi_1, \pi_1 \rangle^o; \langle o, \pi_1, \pi_1 \rangle \\
& \quad \cup \ I_{35}; (\langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle^o; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \text{add}); W^o; I_{35}^o)
\end{align*}
\]

where \( I_{35} : 3 \rightarrow 5 = \langle \pi_1, \pi_2, \pi_3 \rangle^o \) and \( W : 5 \rightarrow 5 = \langle \pi_1, \pi_3, \pi_4, \pi_2, \pi_5 \rangle \). In order to save space we will abuse notation and will write \( f \) for a coreflexive relation \( f^o; f \).
With this abuse in mind, the tabulation of \( \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle \) is:

![Diagram](image)

The reader can see how domain of the tabulations reflects the number of free variables in use by the machine, information which is usually associated to global storage. The codomain of the tabulations — the actual domain of the relations — should be interpreted as the number or working “temporal registers” that are used for parameter passing and unification.

### A.2 Execution

A query \( add(s(X), Y, Z) \) is translated to \( \langle \pi_1 s, \pi_2, \pi_3 \rangle; add \) and its execution trace is:

\[
\begin{align*}
&\langle \pi_1 s, \pi_2, \pi_3 \rangle; \text{add} \\
&\langle \pi_1 s, \pi_2, \pi_3 \rangle; \langle o, \pi_1, \pi_1 \rangle \cup \ldots \quad \Rightarrow \\
&0 \cup \langle \pi_1 s, \pi_2, \pi_3 \rangle; I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \text{add}); W^o; I_{35}^o \quad \Rightarrow \\
&\langle \pi_1 s, \pi_2, \pi_3 \rangle; I_{35}; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \text{add}); W^o; I_{35}^o \quad \Rightarrow \\
&\langle \pi_1 s, \pi_2, \pi_3 \rangle; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_4, \pi_5 \rangle; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \text{add}); W^o; I_{35}^o \quad \Rightarrow \\
&\langle \pi_1 s, \pi_2, \pi_3 s \rangle; \langle \pi_1 s, \pi_2, \pi_3 s, \pi_1, \pi_3 \rangle; W; (id_2 \times \text{add}); W^o; I_{35}^o \quad \Rightarrow \\
&\langle \pi_1 s, \pi_2, \pi_3 s \rangle; \langle \pi_1 s, \pi_3 s, \pi_1, \pi_2, \pi_3 \rangle; (id_2 \times \text{add}); W^o; I_{35}^o \quad \Rightarrow \\
&\langle \pi_1 s, \pi_2, \pi_3 s \rangle; \langle \pi_1 s, \pi_3 s, \pi_1, \pi_2, \pi_3 \rangle; (id_2 \times \text{add}); W^o; I_{35}^o \quad \Rightarrow \\
&\langle \pi_1 s, \pi_3 s \rangle; \langle \pi_1 s, \pi_3 s, \langle \langle \pi_1, \pi_2, \pi_3 \rangle; \langle o, \pi_1, \pi_1 \rangle \rangle \rangle; W^o; I_{35}^o \cup \ldots \quad \Rightarrow \\
&\langle os, \pi_1, \pi_1 s \rangle; \langle os, \pi_1 s, \langle \langle o, \pi_1, \pi_1 \rangle \rangle \rangle; W^o; I_{35}^o \cup \ldots \quad \Rightarrow \\
&\langle os, \pi_1, \pi_1 s \rangle; \langle os, \pi_1 s, \langle o, \pi_1, \pi_1 \rangle \rangle; W^o; I_{35}^o \cup \ldots \quad \Rightarrow \\
&\langle os, \pi_1, \pi_1 s \rangle; \langle os, \pi_1 s, o, \pi_1, \pi_1 \rangle; I_{35}^o \cup \ldots \quad \Rightarrow \\
&\langle os, \pi_1, \pi_1 s \rangle \cup \ldots \quad \Rightarrow
\end{align*}
\]

then \( \langle os, \pi_1, \pi_1 s \rangle \) is translated back to the answer \( X = o, Z = s(Y) \).