Logic programming beyond Prolog

Article · December 2014

Source: arXiv

1 author:

Maarten H. van Emden
University of Victoria
110 PUBLICATIONS **3,398** CITATIONS

Logic programming beyond Prolog

M.H. van Emden
Department of Computer Science
University of Victoria
Research report DCS-355-IR

Abstract

A program in pure Prolog is an executable specification. For example, merge sort in Prolog is a logical formula, yet shows creditable performance on long lists. But such executable specifications are a compromise: the logic is distorted by algorithmic considerations, yet only indirectly executable via an abstract machine.

This paper introduces *relational programming*, a method that solves the difficulty with Prolog programming by a separation of concerns. It requires writing three texts: (1) the axioms, a logical formula that specifies the problem and is not compromised by algorithmic considerations, (2) the theorem, a logical formula that expresses the algorithm and follows from the axioms, and (3) the code, a transcription of the theorem to an efficient procedural language. Correctness of the code relies on the logical relationship of the theorem with the axioms and relies on an accurate transcription of the theorem to the procedural language.

Sorting is an example where relational programming has the advantage of a higher degree of abstractness: the data to be sorted can be any data type in C++ (the procedural language we use in our examples) that satisfies the axioms of linear order, while the Prolog version is limited to linked lists. Another advantage of relational programs is that they can be shown to have a model-theoretic and fixpoint semantics equivalent to each other and analogous to those of pure Prolog programs.

1 Introduction

We review some advantages and disadvantages of logic programming, discuss how *Elements of Programming* [15] addresses one of the disadvantages, and introduce "relational programming" as way of combining the advantages of logic programming with those of *Elements of Programming*.

1.1 Logic programming

An advantage logic programming is that programs can be declaratively read as definitions in logic of relations yet can often be executed in Prolog. Prolog is a versatile programming language of adequate performance for a variety of applications in certain application areas. See Bratko [1] for a sampling of applications in artificial intelligence. Another advantage is that the formal semantics of logic programs can be defined in three ways: model-theoretic, operational, and according to the fixpoint method and that these can be shown to agree [7, 13].

This paper is motivated by a disadvantage and a disappointment of logic programming, both exemplified by the use of Prolog for sorting. The disadvantage is that in Prolog the only kind of sequence that can be sorted is the linked list. This is the consequence of the fact that in logic programming data are terms of logic. While these are a surprisingly versatile data structure, one might want to sort arrays, for example.

The disappointment has to do with program verification. The fact that a logic program is a text that can be executed as it is written and that is also a definition in logic might lead to the expectation

```
sort(V,W) :-
  split(V, V0, V1),
  sort(V0, W0), sort(V1, W1),
  merge(W0, W1, W).
```

Figure 1: A Prolog program for sorting. It is assumed that suitable definitions of `split` and `merge` have been added. It behaves like merge sort, but is it an acceptable specification?

```
sort(V, W) :- perm(V, W), ord(W).
```

Figure 2: A Prolog program for sorting. This version directly reflects the specification: the output is the sorted version of the input list if it is an ordered permutation of it. It runs as a Prolog program when supplemented with suitable definitions for `perm` and `ord`.

that the executable can serve as its own verification. In this way logic programming would eliminate the verification problem.

However, this is too optimistic, as one can see when one wants to use Prolog for sorting a list. In many situations one can get satisfactory performance with the program in Figure 1. But this is not acceptable as a specification. Acceptable as a specification would be the program in Figure 2, which would even run as a Prolog program, though it would take an amount of time in the order of $n!$ for lists of length n.

This example illustrates that not all definitions in logic are equally suited as specification. If proposed as a specification, the program in Figure 1 has to be rejected as being distorted by algorithmic considerations. A solution to this problem has appeared in *Element of Programming*, a 2009 monograph by Alexander Stepanov and Paul McJones [15].

1.2 "Elements of Programming"

In *Elements of Programming* Stepanov and McJones derive many state-of-the-art algorithms in C++ by a method that uses separate formulas of logic for the specification and for the expression of the algorithm. The first formula is referred to as "axioms"; the second as "theorem"[1]. Because of this separation, the axioms can be pure in the sense of being free of algorithmic considerations. The role of the theorem is to express the idea of the algorithm. That theorems written in logic can express algorithms is familiar in logic programming. The idea evolved independently in EOP.

According to this method, which we will refer to as EOP, the axioms would contain a definition of the sortedness relation, say, as being an ordered permutation. But they would also contain the axioms for linear order. As a result the specification covers any algebraic structure that satisfies the axioms. This is valuable for programming, as the axioms also cover sequences other than Prolog lists; for example arrays and, more generally, iterators.

In EOP the Prolog program for merge sort can take the place of the theorem. In EOP the preferred programming language is not Prolog. It is therefore necessary to transcribe the theorem to code in the preferred language. The result could be a program that sorts arrays, among other possibilities for the data structure.

The following table compares EOP and logic programming.

[1] In what sense the theorem is justified by the axioms is addressed in Section 5.4.

```
 1 typedef char T;
 2 class Seg { // segment of an array
 3 public:
 4   T* bgn; unsigned n;
 5   Seg() {}
 6   Seg(T* bgn, unsigned n): bgn(bgn), n(n) {}
 7   void copy(Seg& w) {
 8     for (unsigned i=0; i<n; ++i)
         *(bgn+i) = *((w.bgn)+i);
 9   }
10 };
11 void merge(Seg& w0, Seg& w1, Seg& w) {
...
28 }
29 void split(Seg& v, Seg& v0, Seg& v1) {
...
33 }
34 void sort(Seg& v, Seg& w) {
35   if ((v.n) <= 1) { w.copy(v); return; }
36         // sort([], []) and sort([x], [x])
37   Seg v0, v1; split(v, v0, v1);
38         // split(v, v0, v1)
39   T a[v0.n], b[v1.n];
40   Seg w0(a, v0.n), w1(b, v1.n);
41         // local storage
42   sort(v0, w0); sort(v1, w1);
43         // sort(v0, w0), sort(v1, w1)
44   merge(w0, w1, w);
45         // merge(w0, w1, w)
46 }
```

Figure 3: The Prolog program in Figure 1 transcribed to C++. The comments in lines 34 – 46 indicate the origins in the logic theorem.

	EOP	logic programming
Specification	Figure 2 plus axioms for linear order	Figure 2
Theorem	Prolog merge sort	Prolog merge sort
Code	Figure 3	Prolog merge sort

Advantages of EOP include the following.

1. The relation to be computed is axiomatized without algorithmic considerations. These are relegated to the theorem.

2. Not only the relation to be computed is axiomatized, but also the data space. For the latter many standard axiomatizations are available in algebra textbooks: linear orders, partial orders, semigroups, monoids, Archimedean monoids, ... [15].

3. The abstractness of logic is exploited more fully than in logic programming: the C++ code can implement any algebraic structure that satisfies the axioms. As Hilbert is said to have remarked in

connection with his axiomatization of Euclid's geometry, instead of points, lines, and planes, one can think of tables, beer mugs, and chairs.

1.3 Relational programming

EOP has made an important contribution to solving the problem of connecting specification to code. In this paper we introduce *relational programming*.

1. **Goal**

   To combine the advantages of logic programming and EOP.

2. **Method**

   Logic programming, as introduced by Kowalski [9, 10], is a package of two components: (1) procedural interpretation of logic, and (2) choice of language in which to express procedures. The choice of language made by Kowalski was (pure) Prolog. To us EOP suggests procedural programming languages other than Prolog.

   In the context of EOP, the right place to insert logic programming is the Theorem. In this way we are less constrained than in logic programming: instead of a formula executable in Prolog, we only aim at one that is easily transcribed to the procedural language of choice.

   Because we are no longer tied to Prolog, we can consider alternatives in logic syntax to clausal form. We define relational programs as the if-halves of the Clark completion [3] of a logic program. These can be more directly transcribed to a conventional procedural language.

3. **Results**

   We show that relational programs in the form of the if-halves of the Clark completion can be given a model-theoretic and a fixpoint semantics in the same manner as in [7], provided that Herbrand interpretations are replaced by interpretations with a fixed interpretation for the function symbols over a freely chosen universe of discourse.

   In principle, logic provides a high degree of abstractness. For example, many structures satisfy the axioms for linear order. When an algorithm is expressed as a relational program that is a theorem[2] with respect to these axioms, its transcriptions are correct with respect to any C++ classes or templates for sequences as long as they conform to the axioms.

   Our examples suggest that the transcription of relational programs to C++ is a routine task that can be reliably executed. A problem in coding is that the same function can be written in many different ways. One of the guidelines in software engineering is to suppress this variability and to write code in as stereotyped a fashion as possible. Though this is widely accepted as a guideline, there seems to be no agreement which stereotype to choose from the many candidates. Writing C++ as transcription of a relational program may prove to be a welcome contribution to software engineering[3].

   Limitations

   Logic programs can be nondeterministic and/or reversible. These possibilities are lost in translation to a conventional procedural language.

   Using Prolog ensures that there is no discrepancy between the logic program and what is executed. In relational programming the possibility of error is introduced by the transcription of the relational program to conventional code.

[2] Not in the standard sense of logic. The standard sense requires the theorem to be true in all models of the axioms. In relational programming it is only the case that the minimal model of the theorem satisfies the axioms. See Section 5.4.

[3] It should be noted that Prolog also allows many different variations for the same programming task. The choice of the if-halves of the Clark completion in relational programming removes much of this frivolous variability.

2 Notation and terminology

Not all texts agree on the notations and terminology in set theory and logic that we need in this paper. Therefore we collect in this section the necessary material, terminology, and notation.

2.1 Set theory

Sets We use the conventional notations $\mathcal{N}$, $\mathcal{Z}$, and $\mathcal{Q}$. for the sets of natural numbers, integers, and rational numbers respectively. For $n \in \mathcal{N}$ we often need the set $\{0, \ldots n-1\}$. It is convenient to denote this set as n.

Functions The set of functions that take arguments in a set S and have values in a set T is denoted $S \to T$. This set is said to be the *type* of a function $f \in (S \to T)$. We write $f(a)$ for the element of T that is the value of f for argument $a \in S$.

Suppose we have $f \in S \to T$ and $g \in T \to U$. Then the composition $g \circ f$ of f and g is the function $h \in S \to U$ defined by $x \mapsto g(f(x))$.

Tuples We regard an n-tuple over a set D as an object $d = (d_0, \ldots, d_{n-1})$ in which an element of D is associated with each of the indexes $0, \ldots, n-1$. It is convenient to view such a d as a function of type $n \to D$. This formulation allows us to consider tuples of which the index set is a set other than $\{0, \ldots, n-1\}$. Hence we define a tuple as an element of the function set $I \to D$, where I is an arbitrary countable set to serve as index set. $I \to D$ is the *type* of the tuple. When $t \in I \to D$ is regarded as a tuple, we often write t_i instead of $t(i)$.

Example If t is a tuple in $\{x, y, z\} \to R$, then we may have $t_x = 1.1$, $t_y = 1.21$, and $t_z = 1.331$. A more compact notation would be welcome; we use $t = \begin{array}{c|c|c} x & y & z \\ \hline 1.1 & 1.21 & 1.331 \end{array}$, where the order of columns is immaterial.

Example $t \in \{0, 1, 2\} \to \{a, b, c\}$, where $t = \begin{array}{c|c|c} 2 & 1 & 0 \\ \hline c & c & b \end{array}$. In cases like this, where the index set is of the form $\{0, \ldots, n-1\}$, we use the compact notation $t = (b, c, c)$, using the conventional order of the index set.

Example $(x_0, \ldots, x_{n-1})$ is a tuple of type $n \to \{x_0, \ldots, x_{n-1}\}$. It can be convenient to write $\vec{x}$ for $(x_0, \ldots, x_{n-1})$ when n is apparent from the context.

Example Suppose we have tuple $t \in n \to D$ for some set D and $(x_0, \ldots, x_{n-1}) \in (n \to \{x_0, \ldots, x_{n-1}\})$. In the absence of repeated elements in $(x_0, \ldots, x_{n-1})$, the inverse function $(x_0, \ldots, x_{n-1})^{-1}$ exists. If we set $A = t \circ (x_0, \ldots, x_{n-1})^{-1}$, then we have $(A(x_0), \ldots, A(x_{n-1})) = (t_0, \ldots, t_{n-1}) = t$.

Relations A relation is a set of tuples with the same type. This type is the *type* of the relation. If $J \to D$ is the type of the relation, then J is the index set of the relation and D is the domain of the relation.

Being subsets of $J \to D$, relations of that type are partially ordered by set inclusion.

A relation of type $n \to D$ and is commonly denoted $D \times \cdots \times D = D^n$. This is an example of a relation consisting of tuples indexed by numbers. In general this is not necessarily the case.

Example
$sum = \{(x, y, z) \in (\{0, 1, 2\} \to \mathcal{R}) \mid x + y = z\}$ is a relation of type $\{0, 1, 2\} \to \mathcal{R}$. Compare this relation to the relation
$\sigma = \{s \in (\{x, y, z\} \to \mathcal{R}) \mid s_x + s_y = s_z\}$. As their types are different, they are different relations; $(2, 2, 4) \in sum$ is not the same tuple as $s \in \sigma$ where $s = \begin{array}{c|c|c} x & y & z \\ \hline 2 & 2 & 4 \end{array}$.

2.2 Logic

2.2.1 Syntax

A *signature* L consists of

1. A set of *constant symbols*.
2. Sets of n-ary predicate symbols for nonnegative integers n. These include "=" for $n = 2$, and *true* and *false* for $n = 0$. We denote the arity of a predicate symbol q by $|q|$.
3. Sets of n-ary function symbols for positive integers n. We denote the arity of a function symbol f by $|f|$.

The language of logical formulas is determined by a signature enhanced with a set V of *variables*.

To avoid notational minutiæ we give the syntax in an abstract form.

A *term* is a variable, a constant symbol, or a pair consisting of a k-ary function symbol and a tuple of k terms.

An *atomic formula* (or *atom*) is a pair consisting of a k-ary predicate symbol and a tuple of k terms. The index set of this tuple is $\{0, \ldots, k-1\}$.

A *conjunction* is a tuple C consisting of a set of formulas and an indication that C is a conjunction.

A *disjunction* is a tuple C consisting of a tuple of formulas and an indication that C is a disjunction.

An *implication* is a pair of formulas consisting of conclusion and a condition together with an indication that the pair is an implication.

An *existential quantification* is a pair expression E consisting of a variable and a formula, together with and an indication that E is an existential quantification.

A *universal quantification* is a pair expression A consisting of a variable and a formula, together with and an indication that A is a universal quantification.

2.2.2 Structures and interpretations

A *structure* consists of a universe D, which is a set, and numerically-indexed relations and functions over D.

L is the signature of a structure with universe D when

- each constant in L names an element of D,
- each predicate symbol q in L names a relation of type $|q| \to D$,
- and each function symbol f in L names a function of type $D^{|f|} \to D$.

Definition 1 *An $(F, =)$-interpretation for an L-formula E is an L-structure S with universe D and a mapping F of every n-ary function symbol of E to an n-ary function of S. The binary predicate symbol "=" is mapped to the identity relation over D.*

For a given F, $(F, =)$-interpretations only differ in the relations that are the interpretations of the predicate symbols other than '='. Thus we can view an $(F, =)$-interpretation I as a vector of relations indexed by the set Q of predicate symbols. The component I_q of I that is indexed by an $|q|$-ary predicate symbol q is a relation of type $|q| \to D$.

Definition 2 *Let I_0 and I_1 be $(F, =)$-interpretations with the same signature, the same mapping F and the same universe. We define $I_0 \preceq I_1$ to mean that $[I_0]_q \subseteq [I_1]_q$ for all $q \in Q$, where Q is the set of predicate symbols in L.*
We denote by $\sqcup S$ the least upper bound, if it exists, of a set S of $(F, =)$-interpretations.

Note that $\preceq$ is a partial order.

Example Let L be the signature with constants 0 and 1 and with binary operators $+$ and $\times$. D consists of the natural numbers $\mathcal{N}$ and F maps the symbols to the usual functions over D. With these parameters in place, the $(F, =)$-interpretation gives terms the values that are conventional for arithmetic expressions.

Example Let any signature L be given. The set of L-terms can itself be the universe of a structure with signature L. Let F map every function symbol f in L to the function that with map

$$(a_0, \ldots, a_{|f|-1}) \mapsto f(a_0, \ldots, a_{|f|-1}).$$

This $(F, =)$-interpretation is the Herbrand interpretation for signatures without predicate symbols.

3 Relational programming

EOP has made an important contribution to solving the problem of connecting specification to code. The goal of this paper is to introduce *relational programming*, which aims to improve on EOP while maintaining its advantages. The method we will follow in making these improvements is to modify LP.

Relational programming modifies LP in the following ways.

1. Instead of Horn clauses, it uses *Clark formulas*, formulas exemplified by the if-halves of Clark completions of Horn clauses.
2. Model-theoretic semantics is defined in terms of the semantics of logic.
3. The model-intersection property is expressed in terms of $(F, =)$-interpretations, a generalization of Herbrand interpretations.
4. Fixpoint semantics is given in terms of a mapping of the set of $(F, =)$-interpretations to itself using the semantics of logic formulas.

We review Clark completions and give the general form of a relational program.

3.1 Example of a Clark completion

Clark completions are so called because they make explicit certain assumptions left implicit in logic programs. For example the Clark completion of the program in Figure 1 is

$$\begin{aligned}
\forall v, w.sort(v, w) \quad & \leftrightarrow \\
& (v = nil \wedge w = nil) \vee \\
& (\exists v_0, v_1, w_0, w_1.split(v, v_0, v_1) \wedge \\
& sort(v_0, w_0) \wedge sort(v_1, w_1) \wedge \\
& merge(w_0, w_1, w) \\
&)
\end{aligned}$$

In general, the Clark completion consists of a translation of the Prolog program as well as equality axioms for terms. In the absence of function symbols, as is the case here, there are no equality axioms in the Clark completion.

For each predicate symbol in a left-hand side of a Prolog program there is a sentence in the Clark completion of the form of the above example, namely of an implication with an atom on one side and the other side a disjunction of conjunctions of atoms. Without loss of generality we may assume that both sides have the same set of free variables. Each conjunction is existentially quantified over the variables that do not occur in the left-hand side.

3.2 General form of relational programs

If-halves of Clark completions are examples of relational programs. But relational programs do not have to be related in this way to any logic program.

Independently of any logic program, we define relational programs to be logic sentences of the form given as follows.

Definition 3 *A relational program is a sentence of the form*

$$\bigwedge_{q\in Q}\left[\forall\left[A_q \leftarrow \bigvee_{r\in R_q}\exists\bigwedge_{s\in S_{qr}} B_{qrs}\right]\right].$$

It will be convenient to abbreviate the expression in Definition 3 to $A \leftarrow B$, which stands for

$$\bigwedge_{q\in Q}\forall A_q \leftarrow B_q.$$

According to this definition this further elaborates as shown in the following table.

	$\bigwedge_{q\in Q}\forall A_q \leftarrow B_q$	A_q is header shared by alternative procedure bodies
B_q	$\bigvee_{r\in R_q} B_{qr}$	disjunction of alternative procedure bodies
B_{qr}	$\exists\bigwedge_{s\in S_{qr}} B_{qrs}$	procedure body

Example
Consider the relational program

$$\begin{aligned}\forall x.\ even(x) &\leftarrow x = 0 \ \vee\ (\exists y.\ x = s(y) \wedge odd(y))\\ \forall x.\ odd(x) &\leftarrow x = s(0) \ \vee\ (\exists y.\ x = s(y) \wedge even(y))\end{aligned}$$

This conforms to Definition 3 as shown in the table below.

B_{qr}	$r = 0$	$r = 1$
$q = even$	$S_{qr} = \{0\}$ $x = 0$	$S_{qr} = \{0,1\}$ $\exists y.\ \underbrace{x = s(y)}_{S_{qr0}} \wedge \underbrace{odd(y)}_{S_{qr1}}$
$q = odd$	$S_{qr} = \{0\}$ $x = s(0)$	$S_{qr} = \{0,1\}$ $\exists y.\ \underbrace{x = s(y)}_{S_{qr0}} \wedge \underbrace{even(y)}_{S_{qr1}}$

3.3 Example: de Bruijn's algorithm

Multiplication of integers can be done by repeated addition. This is a slow process unless one makes use of the opportunities to halve the multiplier in conjunction with doubling the multiplicand. This method has been recorded in the Rhind papyrus, an ancient Egyptian document. Similarly, division by repeated

subtraction is a slow process unless a similar trick is used. That such a trick is available is less widely known. It may have first appeared in print in [6], where Dijkstra attributes the algorithm to N.G. de Bruijn.

De Bruijn's algorithm is one of the many examples where Stepanov and McJones [15] derive executable code from a declarative statement concerning a mathematical structure. They take care to axiomatically characterize the most general structure to which the algorithm is applicable. As a consequence the resulting code is most widely usable. In case of de Bruijn's algorithm the most general structure is apparently the "Archimedian monoid", examples of which include the integers, the rational numbers, the binary fractions $n/2^k$, the ternary fractions $n/3^k$, and the real numbers. When a is divided by b with quotient m and remainder u, a, b, and u are monoid elements and m is m is an integer.

An Archimedean monoid is an ordered additive monoid (0 as neutral element, + as binary operation) in which the Archimedean property holds. This property takes different forms for different ordered algebras. In the case of an additive monoid we define the Archimedean property by the axiom $\forall a, b. \exists m, u.\ q(a, b, m, u)$ where $q(a, b, m, u)$ stands for

$$(0 \leq a \wedge 0 < b) \rightarrow (m \cdot b + u = a \wedge 0 \leq u < b).$$

Here $m \cdot b$ stands for $\underbrace{b + \cdots + b}_{m\ times}$.

Although not stated in the form of a theorem, the top two equations on page 82 in [15] state a theorem that holds in Archimedean monoids. We reformulate these equations in two steps, first informally and then formally as Theorem 1.

These equations can be reformulated as follows. Suppose that dividing a by b results in quotient m and remainder u. Suppose that dividing a by $b + b$ results in quotient n and remainder v. Then we have

1. if $a < b$, then $m = 0$ and $u = a$

2. if $b \leq a < b + b$, then $m = 1$ and $u = a - b$

3. if $b + b \leq a$ and $v < b$, then $m = 2n$ and $u = v$

4. if $b + b \leq a$ and $b \leq v$, then $m = 2n + 1$ and $u = v - b$.

Re items 2 and 4: for monoid elements x and y, $x - y$ is only used when $y \leq x$ and is shorthand for the z that exists such that $y + z = x$. As + is the only operation in the monoid, we write $b + b$ rather than $2b$. But m and n are integers so that we see expressions such as $m = 2n$ and $m = 2n + 1$ in items 3 and 4.

In the second step we formalize this as follows.

Theorem 1 *Let $q(a, b, m, u)$ mean that dividing in an Archimedean monoid a by b gives m with remainder u. We assume $0 \leq a$ and $0 < b$. Then we have*

$$\begin{aligned}
&\forall a, b, m, u.\ q(a, b, m, u) \leftarrow \\
&\quad [(a < b \wedge m = 0 \wedge u = a) \vee \\
&\quad (b \leq a \wedge a < b + b \wedge m = 1 \wedge u = a - b) \vee \\
&\quad (b + b \leq a \wedge q(a, b + b, n, v) \wedge aux(b, m, u, n, v))]
\end{aligned}$$

where

```
bool q(a, b, m, u){
  assert(0 <= a && 0 < b);
  if (a < b) { m = 0; u = a; return true; }
  if (b <= a && a < b+b) {
    m = 1; u = a-b; return true;
  }
  if (b+b <= a) { loc n; loc v; // local variables
    return (q(a, b+b, n, v) && aux(b, m, u, n, v)) ?
           true : false;
  }
  return false;
}
bool aux(b, m, u, n, v){
  if (v < b) { m = 2*n; u = v; return true; }
  if (b <= v) { m = 2*n+1; u = v - b; return true; }
  return false;
}
```

Figure 4: Pseudocode in C style for quotient and remainder in Archimedean monoids. For a compilable and executed C++ version see Figure 5.

$$\forall b, m, u, n, v.\ aux(b, m, u, n, v) \leftarrow$$
$$[(v < b \wedge m = 2n \wedge u = v) \vee$$
$$(b \leq v \wedge m = 2n + 1 \wedge u = v - b)]$$

The theorem is transcribed in C-style pseudocode in Figure 4. For the compilable and executed version in C++ see Figure 5.

4 Semantics of logic programs

Without loss of generality one can regard a program in a high-level language as a set of procedure definitions, possibly mutually recursive. The procedures can be regarded as closed, that is, their interactions with the rest of the program are completely determined by the manner in which the output parameters depend on the input parameters. In other words, the meaning of a procedure is the set of possible tuples of parameter values. In mathematics relations are defined as sets of tuples, which makes the meaning of a procedure into a *relation*.

Note how we arrived at relations as meanings of procedures for high-level programming languages in general, without reference to logic. Now, according to the procedural interpretation of logic a set of positive Horn clauses is to be interpreted as a set of procedure definitions with the procedures named by predicate symbols. According to the classical semantics of logic, which predates the advent of computers and programming languages, predicate symbols are to be interpreted as relations. The procedural interpretation of logic connects procedures and relations in the same way as we already did in the context of high-level programming languages in general.

However, the mere fact that predicate symbols are to be interpreted as relations, combined with the fact there exists a precisely defined semantics of first-order predicate logic, does not make it immediately obvious how to use a sentence of logic as a definition of the relations referred to in this sentence.

This was done in [7] by the model-theoretic semantics for logic programs. This type of semantics is possible because a logic program is a sentence of logic. That paper exploited the fact that logic

programs can also be read as procedure definitions to define for such programs an operational and a fixpoint semantics. As one can see in [7], these three characterizations of meaning are equivalent. The simplicity of that paper is based on the fact that all three semantics are within the framework of Herbrand interpretations. Relational programs are not so restricted, so it is not to be expected that they have a model-theoretic semantics.

In this paper we show that, if relational programs are interpreted with $(F, =)$-interpretations, they do have a model-theoretic semantics and that this can be shown to be equivalent to their fixpoint semantics. In our definitions and proofs we mimic those of [7]. The possibility of doing so relies on the fact that $(F, =)$-interpretations are a generalization of Herbrand interpretations.

The remainder of this section is a précis of the results in [7]. In Sections 5 and 6 we present model-theoretic and fixpoint semantics for relational programs.

4.1 Syntax

Logic programs are written in the clausal form of first-order predicate logic. For background and details on this syntax see [14, 11]. A *clause* is an expression of the form

$$A_0, \ldots, A_{m-1} \leftarrow B_0, \ldots, B_{n-1}$$

where A_i and B_j are atomic formulas, for all $i \in m$ and $j \in n$. A *Horn clause* is a clause where $m \leq 1$. It is a *positive* Horn clause if $m = 1$. A logic program is a set of positive Horn clauses.

4.2 Model-theoretic semantics

The *Herbrand universe* of a logic program $\mathcal{P}$ is the set of ground terms containing no function symbols or constants other than those occurring in $\mathcal{P}$. The *Herbrand base* of a logic program $\mathcal{P}$ is the set of all ground atoms containing no predicate symbols other than those occurring in $\mathcal{P}$ and no terms other than those occurring in the Herbrand universe of $\mathcal{P}$. An *Herbrand interpretation* for $\mathcal{P}$ is a subset of the Herbrand base for $\mathcal{P}$.

A clause of a program $\mathcal{P}$ is false in Herbrand interpretation $\mathcal{I}$ iff there exists a ground instance $A \leftarrow B_0, \ldots, B_{n-1}$ of a clause in $\mathcal{P}$ such that $B_i \in \mathcal{I}$ for all $i \in n$ and $A \notin \mathcal{I}$. A logic program is true in $\mathcal{I}$ if no clause in it is false. An *Herbrand model* of a program $\mathcal{P}$ is an Herbrand interpretation in which $\mathcal{P}$ is true.

Theorem 2 *The intersection of a non-empty set of Herbrand models of a logic program $\mathcal{P}$ is an Herbrand model of $\mathcal{P}$ [7].*

Note that the Herbrand base is itself an Herbrand model (all clauses in $\mathcal{P}$ are positive Horn clauses), so that the set of Herbrand models is non-empty. Theorem 2 implies that every logic program has a least model. It is this model that is its meaning under model-theoretic semantics.

4.3 Fixpoint semantics

Associated with every logic program $\mathcal{P}$ there is a mapping $\mathcal{T}_\mathcal{P}$ that maps the set of Herbrand interpretations of $\mathcal{P}$ to itself.

Definition 4 *Let $\mathcal{P}$ be a logic program and $\mathcal{I}$ an Herbrand interpretation for it. $\mathcal{T}_\mathcal{P}(\mathcal{I})$ is the set of all A such that there is a ground instance $A \leftarrow B_0, \ldots, B_{n-1}$, $(n \geq 0)$ of a clause in $\mathcal{P}$ such that $B_i \in \mathcal{I}$ for all $i \in n$ [7].*

Theorem 3 *Let $\mathcal{P}$ be a logic program and $\mathcal{I}$ an Herbrand interpretation for it. $\mathcal{I}$ is a model of $\mathcal{P}$ iff $\mathcal{T}_\mathcal{P}(\mathcal{I}) \subseteq \mathcal{I}$ [7].*

It is easy to see that $\mathcal{T}_\mathcal{P}(\mathcal{I}) = \mathcal{I}$ when $\mathcal{I}$ is the minimal model of $\mathcal{P}$ and that this $\mathcal{I}$ is the least fixpoint. This least fixpoint is the meaning of $\mathcal{P}$ under fixpoint semantics. A loose summary of these results is that for logic programs the model-theoretic semantics coincides with the fixpoint semantics.

Scott and Strachey were the first to use fixpoints the characterize the semantics of computer programs. [7] is but one of many studies to follow their example. Most of them use the monotonicity of $\mathcal{T}$ to show the existence of the least fixpoint and use the continuity of $\mathcal{T}$ to show that it is equal to the least upper bound of the set containing $\mathcal{T}^n_\mathcal{P}(\bot)$ for all natural numbers n. In [7] the role of $\bot$ is played by the empty Herbrand interpretation and properties of logic programs are used to provide a shortcut to both results. The first, the existence of the least fixpoint, is obtained along the lines of the above remark. Moreover, [7] proves the following.

Theorem 4 *If $\mathcal{P}$ is a logic program, then $\bigcup_{n=0}^{\infty} \mathcal{T}_\mathcal{P}(\emptyset)$ is the least fixpoint of $\mathcal{T}_\mathcal{P}$.*

The result is obtained via the completeness of hyperresolution, an inference rule that is similar to an application of $\mathcal{T}_\mathcal{P}$.

4.4 Operational semantics

In [7] an operational semantics is defined for logic programs. As this relies on the choice of Prolog as procedural language, it is not relevant here.

5 Model-theoretic semantics of relational programs

The semantics of logic programs could be simplified because the interpretations were restricted to Herbrand interpretations. For relational programs the interpretations are $(F, =)$-interpretations for any universe, so the semantics of logic is needed in its full generality.

5.1 Semantics of formulas

Consider an L-formula E and an interpretation for it.

An interpretation I will be the basis of the determination of the meaning M^I of the terms and formulas of logic.

Definition 5 *Semantics of variable-free terms and formulas under interpretation I is defined as follows.*

- $M^I(c) = I(c)$ *if c is a constant.*
- $M^I(f(t_0, \ldots, t_{n-1})) = (I(f))(M^I(t_0), \ldots, M^I(t_{n-1}))$ *if f is a function symbol.*
- $q(t_0, \ldots, t_{k-1})$ *is satisfied by I iff* $(M^I(t_0), \ldots, M^I(t_{k-1})) \in I(q)$ *if q is a predicate symbol.*
- *A conjunction $\{E_0, \ldots, E_{n-1}\}$ of formulas is satisfied by I iff E_i is satisfied by I for all $i \in n$.*
- *A disjunction $\{E_0, \ldots, E_{n-1}\}$ of formulas is satisfied by I iff E_i is satisfied by I for at least one $i \in n$.*

We now consider meanings of formulas with a set V of free variables, possibly, but not typically, empty. Let α be an *assignment*, which is a function in $V \to D$, assigning an individual in D to every variable. In other words, α is a tuple of elements of D indexed by V. As meanings of expressions with variables depend on α, we write M^I_α for the function mapping a term to an element of the universe D.

Definition 6 *M^I_α is defined as follows.*

- $M^I_\alpha(t) = \alpha(t)$ *if t is a variable*

- $M^I_\alpha(c) = I(c)$ *if* c *is a constant*
- $M^I_\alpha(f(t_0, \ldots, t_{n-1})) = (I(f))(M^I_\alpha(t_0), \ldots, M^I_\alpha(t_{n-1}))$.
- $q(t_0, \ldots, t_{k-1})$ *is satisfied by* I *with* α *iff* $(M^I_\alpha(t_0), \ldots, M^I_\alpha(t_{k-1})) \in I(q)$

Now that satisfaction of atoms is defined, we can continue inductively with satisfaction of complex formulas.

- *A conjunction* $\{E_0, \cdots, E_{n-1}\}$ *is satisfied by* I *with* α *iff the formulas* E_i *are satisfied by* I *with* α*, for all* $i \in n$.
- *A disjunction* $\{E_0, \cdots, E_{n-1}\}$ *is satisfied by* I *with* α *iff the formulas* E_i *are satisfied by* I *with* α*, for at least one* $i \in n$.
- *If* E *is a formula, then* $\exists x.E$ *is satisfied by* I *with* α *iff there is a* $d \in D$ *such that* E *is satisfied by* I *with* $\alpha_{x|d}$ *where* $\alpha_{x|d}$ *is an assignment that maps* x *to* d *and maps the other variables according to* α.
- *If* E *is a formula, then* $\forall x.E$ *is satisfied by* I *with* α *iff for all* $d \in D$, E *is satisfied by* I *with* $\alpha_{x|d}$ *where* $\alpha_{x|d}$ *is an assignment that maps* x *to* d *and maps the other variables according to* α.

So far, M^I has assigned meanings to variable-free terms and to formulas without free variables. This is now extended as follows to formulas with free variables.

Definition 7 *If* t *is a term with set* V_t *of variables, then* $M^I(t)$ *is the function of type* $(V_t \to D) \to D$ *that maps* $\alpha \in (V_t \to D)$ *to* $M^I_\alpha(t) \in D$.

Example Suppose that t is $s(s(x))$, t' is $x + 2$, $V_t = \{x\}$, $D = \mathcal{Z}$, I maps the function symbol + to addition among the integers $\mathcal{Z}$ and maps s to the successor function. Now $M^I(t)$ and $M^I(t')$ are the same function in $(\{x\} \to \mathcal{Z}) \to \mathcal{Z}$.

Example t is $x + 2 \times y + 3 \times z$, $V_t = \{x, y, z\}$, $D = \mathcal{Z}$, I maps the function symbol + to addition among integers and maps $\times$ to multiplication. Then $M^I(t)$ is the function of type $(\{x, y, z\} \to \mathcal{Z}) \to \mathcal{Z}$ with map $\alpha \mapsto M^I_\alpha(t)$ e.g.

$M^I(t)(\ \begin{array}{c|c|c} x & y & z \\ \hline 3 & 2 & 1 \end{array}\ ) = M^I_\alpha(t) = 10$ where $\alpha = \begin{array}{c|c|c} x & y & z \\ \hline 3 & 2 & 1 \end{array}$.

The following definition follows Tarski *et al.* [8], Cartwright [2], page 377, and Clark [4]. It does for formulas what Definition 7 does for terms.

Definition 8 *Let* E *be a formula with set* V_E *of free variables. We define*

$$M^I(E) = \{\alpha \in (V_E \to D) \mid E \text{ is satisfied by } I \text{ with } \alpha\}.$$

Thus $M^I(E)$ is a relation of type $V_E \to D$.

Example $M^I(x \times x + y \times y < 2 \wedge x > y) = \{ \begin{array}{c|c} x & y \\ \hline 1 & 0 \end{array} \}$ where $D = \mathcal{Z}$ and $\times$, $+$, and $<$ have the usual interpretations.

If V_E is empty, then Definition 8 gives the semantics of a closed formula, a sentence. This conforms to the conventional definition of satisfaction if we identify the relation $\{\}$ with "not satisfied by I" and identify being satisfied by I with the relation that is the singleton set containing the empty tuple. In fact, one may define logical implication in terms of Definition 8.

Definition 9 *If A and T are sentences, then $A \models T$ holds iff for all interpretations I, $M^I(T)$ is non-empty whenever $M^I(A)$ is non-empty.*

5.2 Model-theoretic semantics of relational programs

In Section 4 we noted that the mere fact that predicate symbols are to be interpreted as relations, combined with the fact we have a precisely defined semantics of first-order predicate logic, does not make it immediately obvious how to use a sentence of logic as a definition of the relations referred to in this sentence. In Section 4 we mentioned that this was done in [7] for logic programs. The solution given in that paper relied on Herbrand interpretations. In the absence of that crutch we have to consider afresh the question:

> *How do we use a sentence of logic to define the relations named in the sentence?*

Formulas are connected to relations by the fact that their predicate symbols are interpreted as relations. Hence a plausible answer to the question is:

> *A sentence defines a set of relations as the relations in the interpretation that makes the sentence true.*

However, there may be more than one such interpretation, or there may not be any. So this approach does not work. The less obvious approach in the remainder of this section does. The reason that it does so is that it allows us to show that there exists at least one interpretation and that there is *the least* interpretation that makes the sentence true. The relations in this least interpretation are, by our definition, the ones defined by the relational program.

To show that a relational program has a least $(F, =)$-model for given F our starting point is the definition of $(F, =)$-model. As relational programs are sentences of logic, the special case of an empty set of free variables of Definition 8 applies. This makes a new definition superfluous. But to emphasize this point we do add the following.

Definition 10 *An $(F, =)$-interpretation I for a relational program P is a model of P if P is true in I.*

The following characterization will turn out to be useful.

Lemma 1 *An $(F, =)$-interpretation I is a model of a relational program $A \leftarrow B$ with set Q of predicate symbols iff $M^I(A_q) \supseteq M^I(B_q)$ for all $q \in Q$.*

Proof
(If)
Suppose I is not a model of P.

I falsifies $\forall(A_q \leftarrow B_q)$ for some $q \in Q \Rightarrow$ (1)
$\exists\alpha \in (V_q \rightarrow D).\ A_q \leftarrow B_q$ is not true in I with $\alpha \Rightarrow$ (2)
$\exists\alpha \in (V_q \rightarrow D).\ B_q$ is true and A_q is false in I with $\alpha \Rightarrow$ (3)
$\exists\alpha \in (V_q \rightarrow D).\ \alpha \in M^I(B_q)$ and $\alpha \notin M^I(A_q) \Rightarrow$ (4)
$\exists q \in Q.\ M^I(A_q) \not\supseteq M^I(B_q)$.

(1,2,3): Definition 6; (4): Definition 8.

(Only if)
Assume I is a model of P.

$\alpha \in M^I(B_q) \Rightarrow$ (1)
$\exists r \in R_q.\ B_{qr}$ true in I with $\alpha \Rightarrow$ (2)
A_q true in I with $\alpha \Rightarrow$ (3)
$\alpha \in M^I(A_q)$

(1): Definition 8; (2): assumption; (3): Definition 8.

Whether, and how, this allows a relational program to define relations depends on the properties of the models.

5.3 The model-intersection property

Lemma 2 *Let L be a non-empty set of $(F,=)$-interpretations as defined in Section 2.2.2. Let $q(t_0,\ldots,t_{|q|-1})$ be an atomic formula set V of variables. We have*

$$M^{\cap L}(q(t_0,\ldots,t_{|q|-1})) = \cap_{I\in L} M^I(q(t_0,\ldots,t_{|q|-1}))$$

Proof

$$\begin{array}{lcl}
M^{\cap L}(q(t_0,\ldots,t_{|q|-1})) & = & (1)\\
\{\alpha \in V \to D \mid q(t_0,\ldots,t_{|q|-1}) \text{ true in } M^{\cap L} \text{ with } \alpha\} & = & (2)\\
\{\alpha \in V \to D \mid (M^{\cap L}_\alpha(t_0),\ldots M^{\cap L}_\alpha(t_{|q|-1})) \in [\cap L]_q\} & = & (3)\\
\{\alpha \in V \to D \mid (M_\alpha(t_0),\ldots M_\alpha(t_{|q|-1})) \in [\cap L]_q\} & = & (4)\\
\{\alpha \in V \to D \mid \forall I \in L.\ (M_\alpha(t_0),\ldots M_\alpha(t_{|q|-1})) \in I_q\} & = & (5)\\
\{\alpha \in V \to D \mid \forall I \in L.\ q(t_0,\ldots,t_{|q|-1}) \text{ true in } I \text{ with } \alpha\} & = & (6)\\
\cap_{I\in L}\{\alpha \in V \to D \mid q(t_0,\ldots,t_{|q|-1}) \text{ true in } I \text{ with } \alpha\} & = & (7)\\
\cap_{I\in L} M^I(q(t_0,\ldots,t_{|q|-1}))
\end{array}$$

(1) Definition 8, (2) Definition 6, (3) the fact that L only contains $(F,=)$-interpretations, so that the meaning of terms is independent of the interpretation, (4) the definition of L, (5) Definition 6, (6) the definition of L, and (7) is by Definition 8.

Theorem 5 *If L is a non-empty set of $(F,=)$-models of P, then $\cap L$ is an $(F,=)$-model of P.*

Proof. We assume that $\cap L$ is not a model and show that this leads to a contradiction.

$$\begin{array}{lcl}
\cap L \text{ is not a model} & \Rightarrow & (1)\\
\cap L \text{ falsifies } A_q \leftarrow B_q \text{ for at least one } q \in Q & \Rightarrow & (2)\\
M^{\cap L}(A_q) \not\supseteq M^{\cap L}(B_q) \text{ for at least one } q \in Q & \Rightarrow & (3)\\
\exists \alpha \in V_q \to D.\ \ \alpha \in M^{\cap L}(B_q) \text{ and } \alpha \notin M^{\cap L}(A_q) & \Rightarrow & (4)\\
\exists \alpha \in V_q \to D.\ \ \forall I \in L.\ \alpha \in M^I(B_q) \text{ and } \alpha \notin M^I(A_q) & \Rightarrow & (5)\\
\exists \alpha \in V_q \to D.\ \ \forall I \in L.\ \alpha \in M^I(A_q) \text{ and } \alpha \notin M^I(A_q) & \Rightarrow & (6)\\
\exists \alpha \in V_q \to D.\ \ \alpha \in M^{\cap L}(A_q) \text{ and } \alpha \notin M^{\cap L}(A_q) & &
\end{array}$$

(4) is by the definition of L and Lemma 2, (5) is by the fact that I is a model of P, and (6) is by the fact that A_q is an atomic formula and by Lemma 2.

5.4 Theorem and axioms

Now that we have identified a convincingly distinguished one among the typically multiple models of a relational program, we are in a position to describe the way the relational program is justified as a theorem by the axioms of the mathematical structure.

The usual way in which a theorem is justified is that it is a logical implication of the axioms. This is the case for example with the theorem that states that in a multiplicative group the unit element is unique. Here the theorem is true in all models of the axioms. However, we do not claim this relationship between Theorem 1 and the axioms for Archimedean monoids. Our theorems are relational programs, each of which has a unique minimal model. We only claim that the relational program is justified by the axioms in the sense that the axioms are true in the minimal model of the relational program.

Kowalski described a similar situation in [12], section 5.2. His example is the logic program

$$plus(0, X, X) \quad \leftarrow \tag{1}$$

$$plus(s(X), Y, s(Z)) \quad \leftarrow \quad plus(X, Y, Z) \tag{2}$$

It is desired to prove that the *plus* relation defined by this logic program is single-valued in the third argument:

$$\forall x, y, u, v.\ plus(x, y, u) \wedge plus(x, y, v) \rightarrow u = v \tag{3}$$

(3) is not a logical consequence of (1) and (2).

For example, result of adding

$$plus(0, 0, s(0)), plus(s(0), 0, s(s(0))), plus(s(s(0)), 0, s(s(s(0)))), \ldots$$

to the minimal model results in a model of (1) and (2).

As pointed out by [12], this problem was addressed by Clark and Tärnlund [5]. The replaced the program by what later came to be called the Clark completion and added an axiom schema for induction. We prefer to view such a proof as a demonstration that (3) is true in the minimal model of (1) and (2).

(3) is often referred to as a "property" of (1) and (2). Indeed it is the aim of program verification to prove that programs have certain properties. Logic can be used in more ways than one for this purpose.

We propose to define a first-order sentence E to be a property of a relational program P whenever E is true in the minimal $(F, =)$-model of P.

6 Fixpoint semantics of relational programs

Given a relational program P of the form $A \leftarrow B$, we use the vector B of right-hand sides to define a map T_P from the set of $(F, =)$-interpretations of P to itself. We plan to show that $I \supseteq T_P(I)$ has a unique least solution and that this equals the least model of P.

At first sight it might seem that one can simply define $T_P(I) = M^I(B)$, by analogy with Definition 4. However, the q-component of $M^I(B)$ is not an interpretation for the predicate symbol q, which is what the q-component of $T_P(I)$ has to be.

Consider one of the conjuncts of P: $q(x_0, \ldots, x_{|q|-1}) \leftarrow B_q$. $M^I(B_q)$ is a relation consisting of tuples t indexed by the set $\{x_0, \ldots, x_{|q|-1}\}$. The q-component of $T_P(I)$ is a relation consisting of tuples indexed by the set $\{0, \ldots, |q| - 1\}$. As there are no repeated occurrences of a variable in $q(x_0, \ldots, x_{|q|-1})$, the inverse $(x_0, \ldots, x_{|q|-1})^{-1}$ exists so that $t \circ (x_0, \ldots, x_{|q|-1})^{-1}$ is a tuple indexed by $\{x_0, \ldots, x_{|q|-1}\}$. This observation suggests the following definition.

Definition 11 *Let P be a relational program of the form $A \leftarrow B$, with Q as set of predicate symbols. For every $q \in Q$, A_q is $q(x_0, \ldots, x_{|q|-1})$. We define T_P as a map from the set of $(F, =)$-interpretations for P to itself. $T_P(I)$ is defined as the vector of relations indexed by Q that has the q-component*

$$\{t \in (|q| \rightarrow D) \mid B_q \text{ is true in } I \text{ with } t \circ (x_0, \ldots, x_{|q|-1})^{-1}\}.$$

Theorem 6 *For any relational program P and $(F, =)$-interpretations I_0 and I_1, $I_0 \preceq I_1$ implies $T_P(I_0) \preceq T_P(I_1)$. That is, T_P is monotonic.*

Proof
It suffices to show that $[I_0]_q \subseteq [I_1]_q$ implies $[T_P(I_0)]_q \subseteq [T_P(I_1)]_q$ for every $q \in Q$, where Q is the set of predicate symbols in P.
$t \in [T_P(I_0)]_q \Rightarrow$
B_q true in I_0 with $t \circ (x_0, \ldots, x_{|q|-1})^{-1} \Rightarrow$
B_{qr} true in I_0 with $t \circ (x_0, \ldots, x_{|q|-1})^{-1}$ for at least one $r \in R_q \Rightarrow$
B_{qrs} true in I_0 with $t \circ (x_0, \ldots, x_{|q|-1})^{-1}$ for at least one $r \in R_q$ and all $s \in S_q r \Rightarrow (I_0 \preceq I_1)$
B_{qrs} true in I_1 with $t \circ (x_0, \ldots, x_{|q|-1})^{-1}$ for at least one $r \in R_q$ and all $s \in S_q r \Rightarrow (B_{qrs}$ are all atomic formulas)
B_{qr} true in I_1 with $t \circ (x_0, \ldots, x_{|q|-1})^{-1}$ for at least one $r \in R_q \Rightarrow$
B_q true in I_1 with $t \circ (x_0, \ldots, x_{|q|-1})^{-1} \Rightarrow t \in [T_P(I_1)]_q$

6.1 Existence of (least) fixpoint

Theorem 3 connects models of logic programs and the mapping $\mathcal{T}$. The following concerns its counterpart for relational programs and the mapping T.

Theorem 7 *Let I be an $(F, =)$-interpretation of a relational program P with set Q of predicate symbols. Then we have that I is a model of P iff $T_P(I) \subseteq I$.*

Proof
(Only if) Assume I is a model of P and assume t is an element of the q-component of $T_P(I)$ for some $q \in Q$.

$t \in [T_P(I)]_q$	$\Rightarrow$ (1)
B_q true I with $t \circ (x_0, \ldots, x_{\|q\|-1})^{-1}$	$\Rightarrow$ (2)
A_q true I with $t \circ (x_0, \ldots, x_{\|q\|-1})^{-1}$	$\Rightarrow$ (3)
$q(x_0, \ldots, x_{\|q\|-1})$ true in I with $t \circ (x_0, \ldots, x_{\|q\|-1})^{-1}$	$\Rightarrow$ (4)
$t \in I_q$	

(1): Definition 11, (2): I is a model, (3): A_q is the atom $q(x_0, \ldots, x_{|q|-1})$, and (4): by Section 5.1.

(If) Assume $T_P(I) \subseteq I$ and assume that $\alpha \in M^I(B_q)$ for some $q \in Q$.

$\alpha \in M^I(B_q)$	$\Rightarrow$ (1)
B_q true I with α	$\Rightarrow$ (2)
B_q true I with $t \circ (x_0, \ldots, x_{\|q\|-1})^{-1}$	$\Rightarrow$ (3)
$t \in [T_P(I)]_q$	$\Rightarrow$ (4)
$t \in I_q$	$\Rightarrow$ (5)
$q(x_0, \ldots, x_{\|q\|-1})$ true I with $t \circ (x_0, \ldots, x_{\|q\|-1})^{-1}$	$\Rightarrow$ (6)
$q(x_0, \ldots, x_{\|q\|-1})$ true I with α	$\Rightarrow$ (7)
A_q true in I with α	$\Rightarrow$ (8)
$\alpha \in M^I(A_q)$	

(1): by Definition 8, (2): define $\alpha = t \circ (x_0, \ldots, x_{|q|-1})^{-1}$, where $x_0, \ldots, x_{|q|-1}$ is the enumeration of the variables of B_q that occurs in A_q and $t \in I_q$, (3): by Definition 11, (4): by assumption $T_P(I) \subseteq I$, (5): by Section 5.1, (6): using definition of α, (7): A_q is the atom $q(x_0, \ldots, x_{|q|-1})$, and (8): by Definition 8.
Hence $M^I(A_q) \supseteq M^I(B_q)$ for all $q \in Q$, so that I is a model, via Lemma 1.

Corollary 1 *For any relational program P, any universe D, and any mapping F from function symbols to functions over D, T_P has a unique least fixpoint.*

6.2 Approximations to the least fixpoint

The standard method for showing that the least upper bound of $\{T_P^n(\bot)\}$ (where $\bot$ is the least element in the partially ordered set of $(F,=)$-interpretations) is the least fixpoint is to show that the set of $(F,=)$-interpretation is 'complete' (in a suitable sense) and that the T_P is 'continuous' (in a suitable sense). However, we prefer to exploit the fact that the corresponding fact has been proved for logic programs (Theorem 4). The properties of logic programs allowed this to be proved in [7] without relying on completeness or continuity.

To enable us to exploit for our purpose the result about fixpoint semantics of logic programs we establish a correspondence between logic programs and relational programs and between Herbrand interpretations and $(F,=)$-interpretations. Our key result will be to show that $T_P(M^I(\mathcal{I})) = M^I(\mathcal{T}_\mathcal{P}(\mathcal{I}))$, where P is a relational program, T_P the corresponding mapping, $\mathcal{P}$ is the logic program corresponding to P (Definition 12), $M^I(\mathcal{I})$ is the $(F,=)$-interpretation corresponding to the Herbrand interpretation $\mathcal{I}$ (Definition 13), and similarly for $M^I(\mathcal{T}_\mathcal{P}(\mathcal{I}))$.

Definition 12 *Let P be the relational program*

$$\bigwedge_{q\in Q}\left[\forall\left[A_q \leftarrow \bigvee_{r\in R_q}\exists\bigwedge_{s\in S_{qr}} B_{qrs}\right]\right].$$

The logic program consisting of the clauses

$$A_q \leftarrow B_{qr0},\ldots,B_{qr(|S_{qr}|-1)}$$

for all $q\in Q$ and for all $r\in R_q$ is the logic program corresponding to P.

M^I has been defined for individual formulas, but not for sets of formulas. There does not seem to be a use for such an extension, except for the special case of Herbrand interpretations, regarded as sets of ground atomic formulas.

Definition 13 *Let $\mathcal{I}$ be an Herbrand interpretation and let I be an $(F,=)$-interpretation. We define*

$$[M^I(\mathcal{I})]_q = \{(M^I(g_0),\ldots,M^I(g_{|q|-1})) \mid q(g_0,\ldots,g_{|q|-1})\in\mathcal{I}\}.$$

For $M^I(\emptyset)$ we write $\bot$.

Lemma 3 *Let P be a relational program with set Q of predicate symbols and let $\mathcal{P}$ be the corresponding logic program. Let $\mathcal{I}$ be an Herbrand interpretation for $\mathcal{P}$. We have $T_P(M^I(\mathcal{I})) = M^I(\mathcal{T}_\mathcal{P}(\mathcal{I}))$.*

Proof
First we show that $T_P(M^I(\mathcal{I})) \supseteq M^I(\mathcal{T}_\mathcal{P}(\mathcal{I}))$. We show this for any $q\in Q$ and let $q(g_0,\ldots,g_{|q|-1})\in\mathcal{I}$ and $(d_0,\ldots,d_{|q|-1}) = (M^I(g_0),\ldots,M^I(g_{|q|-1}))$.

The desired inclusion is shown by the following chain of inferences.

$(d_0,\ldots,d_{|q|-1})\in[M^I(\mathcal{T}_\mathcal{P}(\mathcal{I}))]_q \Rightarrow$
$q(g_0,\ldots,g_{|q|-1})\in\mathcal{T}_\mathcal{P}(\mathcal{I}) \Rightarrow$
$\exists$ a ground instance $q(g_0,\ldots,g_{|q|-1})\leftarrow\mathcal{B}_0,\ldots,\mathcal{B}_{k-1}$ of a clause $\mathcal{C}\in\mathcal{P}$ such that $\mathcal{B}_i\in\mathcal{I}\ \forall i\in k \Rightarrow$
(assume that $\mathcal{B}_i$ has the form $q^i(h_0,\ldots,h_{|q^i|-1})$)
$(M^I(h_0),\ldots,M^I(h_{|q^i|-1}))\in[M^I(\mathcal{I})]_q \Rightarrow$
[assume that $\forall\ q^i(x_0,\ldots,x_{|q|-1})\leftarrow\bigvee_{j\in n}\exists B_{q^ij}$ is the conjunct in P that causes $\mathcal{C}$ to be in $\mathcal{P}$ according to Definition 12 and that B_{q^ij} is $B_0,\ldots,B_{m-1}$ which is $\bigwedge_{k\in m}\mathcal{B}_k$ because these conjuncts are atomic formulas and appear as such in $\mathcal{P}$] $\Rightarrow$
B_{q^ij} is true in $M^I(\mathcal{I})$ with assignment $(M^I(h_0),\ldots,M^I(h_{|q^i|-1}))\circ(x_0,\ldots,x_{|q|-1})^{-1} \Rightarrow$

$(M^I(g_0), \ldots, M^I(g_{|q^i|-1})) \in [T_P(M^I(\mathcal{I}))]_q \Rightarrow$
$(d_0, \ldots, d_{|q|-1}) \in [T_P(M^I(\mathcal{I}))]_q$.

It remains to be shown that $T_P(M^I(\mathcal{I})) \subseteq M^I(\mathcal{T}_\mathcal{P}(\mathcal{I}))$. Once more, per predicate symbol $q \in Q$.
$(d_0, \ldots, d_{|q|-1}) \in [T_P(M^I(\mathcal{I}))]_q \Rightarrow$
there exists $q(x_0, \ldots, x_{|q|-1} \leftarrow B_q$ in P and B_q is true in $M^I(\mathcal{I})$ with assignment $(d_0, \ldots, d_{|q^i|-1}) \circ (x_0, \ldots, x_{|q|-1})^{-1} \Rightarrow$
there exists a disjunct B_{qr} such that all its atoms B_{qrs} are true in $M^I(\mathcal{I})$ with $(d_0, \ldots, d_{|q|-1}) \circ (x_0, \ldots, x_{|q|-1})^{-1}$ $\Rightarrow$
there exists a clause in $\mathcal{P}$ with ground instance $q(g_0, \ldots, g_{|q|-1}) \leftarrow \mathcal{B}_0, \ldots, \mathcal{B}_{k-1}$ such that $\forall i \in k\ \mathcal{B}_i \in \mathcal{I}$ and $M^I(g_j) = d_j\ \forall j \in |q| \Rightarrow$
$q(g_0, \ldots, g_{|q|-1}) \in [\mathcal{T}_\mathcal{P}(\mathcal{I})]_q \Rightarrow$
$(d_0, \ldots, d_{|q|-1}) \in [M^I(\mathcal{T}_\mathcal{P}(\mathcal{I}))]_q$.

Lemma 4 *Under the conditions of Lemma 3 we have $T_P^n(\bot) = M^I(\mathcal{T}_\mathcal{P}^n(\emptyset))$ for all $n \in \mathcal{N}$.*

Proof By induction on n. The induction basis $n = 0$ is the case of Lemma 3 where $\mathcal{I} = \emptyset$.

Assume true for n.
$T_P^{n+1}(\bot) =$
$T_P(T_P^n(\bot) =$ (induction assumption)
$T_P(M^I(\mathcal{T}_\mathcal{P}^n(\emptyset)) =$ (Lemma 3)
$M^I(\mathcal{T}_\mathcal{P}(\mathcal{T}_\mathcal{P}^n(\emptyset))) =$
$M^I(\mathcal{T}_\mathcal{P}^{n+1}(\emptyset))$.

Lemma 5 *If the least upper bound L of $\{T^n(\bot) \mid n \in \mathcal{N}\}$ exists, then $L = M^I(\bigcup_{n=0}^\infty \mathcal{T}_\mathcal{P}^n(\emptyset))$.*

Proof In a partially ordered set one can show that $X = Y$ by showing $X \preceq Y$ and $Y \preceq X$.
For all $n \in \mathcal{N}$ we have $T^n(\bot) \preceq M^I(\bigcup_{n=0}^\infty \mathcal{T}_\mathcal{P}^n(\emptyset))$, so that $L \preceq M^I(\bigcup_{n=0}^\infty \mathcal{T}_\mathcal{P}^n(\emptyset))$.

For all $n \in \mathcal{N}$ we have, by Lemma 4, $M^I(\mathcal{T}_\mathcal{P}^n(\emptyset)) \preceq L$. Hence $M^I(\bigcup_{n=0}^\infty \mathcal{T}_\mathcal{P}^n(\emptyset)) \preceq L$.

Theorem 8 *If P is a relational program, then*

$$\sqcup\{T_p^n(\bot) \mid n \in \mathcal{N}\}$$

is the least fixpoint of T.

Proof
$\sqcup\{T_p^n(\bot) \mid n \in \mathcal{N}\} =$ (Lemma 5)
$M^I(\bigcup_{n=0}^\infty \mathcal{T}_\mathcal{P}^n(\emptyset)) =$ (Theorem 4)
$M^I(\mathcal{T}(\bigcup_{n=0}^\infty \mathcal{T}_\mathcal{P}^n(\emptyset)]) =$ (Lemma 3)
$T_P(M^I(\bigcup_{n=0}^\infty \mathcal{T}_\mathcal{P}^n(\emptyset))) =$ (Lemma 5)
$T_P(\sqcup\{T_p^n(\bot) \mid n \in \mathcal{N}\})$.

This shows that $\sqcup\{T_p^n(\bot) \mid n \in \mathcal{N}\}$ is a fixpoint of T_P.

It remains to show that it is the least fixpoint. Let f be any fixpoint of T_P. We have $\bot \preceq f$. By monotonicity of T_P we have that $T(\bot) \preceq T(f) = f$. Similarly, for all $n \in \mathcal{N}$, $T^n(\bot) \preceq f$. As f is an upper bound for all n, we have for the least upper bound $\sqcup\{T_p^n(\bot) \mid n \in \mathcal{N}\} \preceq f$, which shows that $\sqcup\{T_p^n(\bot) \mid n \in \mathcal{N}\}$ is not only a fixpoint, but also the least.

7 Future work

Transcription of relational programs to C++ is easy enough. However, those who are oppressed by the size and complexity of C++ might be interested in the language resulting from eliminating everything not needed for the transcription of relational programs.

Conversely, formal logic was formed more than a century ago and has, with few exceptions, only been used for theoretical purposes. Even textbooks on abstract algebra give the axioms informally. Logic lacks facilities for writing large formulas in a structured fashion. It may benefit from some of the structuring facilities that allow programs of many thousands of lines to be written in conventional programming languages.

8 Conclusions

- The work reported here suggests the following method of programming. First, express an algorithm in the form of a relational program P. Second, determine the most general family of structures to which the algorithm is applicable and write a list A of axioms characterizing this family such that A is true in the minimal model of P. Finally, transcribe the relational program to a program P' in a suitable procedural language. The result is an efficient program P' that has property A in a sense that is defined in terms of the semantics of first-order predicate logic. In the case of C++ as the procedural language, P' can be compiled to efficient code, even though it is written in an unusual style. The compiler's optimization capabilities can take care of the superficially apparent inefficiencies.

- Fixpoint and model-theoretic semantics of logic programs with respect to Herbrand interpretations generalize to these semantics for relational programs with respect to $(F, =)$-interpretations.

- Kowalski's Procedural Interpretation of Logic has not only procedurally interpreted Horn clauses, but also limited the language for expressing procedures to pure Prolog. The work reported here interprets Horn clauses as procedures, but leaves open the choice of procedural language; we do not propose to replace Prolog, but to expand the scope of logic programming.

9 Acknowledgements

Thanks to Philip Kelly, Paul McJones, and Areski Nait-Abdallah for useful discussions.

References

[1] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2001, 3rd ed.

[2] Robert Cartwright. Recursive programs as definitions in first-order logic. *SIAM Journal of Computing*; vol. 13, no. 2, May 1984, pages 374–408.

[3] Keith L. Clark. Negation as Failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[4] Keith L. Clark. Logic-programming schemes and their implementations. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 487–541. MIT Press, 1991.

[5] Keith L. Clark and S.-Å Tärnlund. A first-order theory of data and programs. *Proc. IFIP Congress*, pages 939–944, Toronto, 1977.

[6] E.W. Dijkstra Notes on structured programming. In *Structured Programming*, O-J. Dahl, E.W. Dijkstra, C.A.R. Hoare, eds. Academic Press, 1972.

[7] Emden, M. van, and Kowalski, R. The Semantics of Predicate Logic as a Programming Language *JACM* Vol. 23, No. 4, (1976) 733-742.

[8] Henkin, L., Monk, J. D., and Tarski, A. *Cylindric Algebras, Part I.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1971.

[9] Kowalski, R.A. Predicate logic as a programming language. DCL Memo 70, School of Artificial Intelligence, Univ. of Edinburgh, U.K., Nov. 1973.

[10] Kowalski, R.A. Predicate logic as a programming language. In: *Proceedings of IFIP 1974* North-Holland, Amsterdam, 1974, 569-574.

[11] Kowalski, R.A. *Logic for Problem-Solving* North Holland Elsevier, 1979.

[12] Kowalski, R.A. Logic Programming. in: *Computational Logic* J. Siekmann, D. Gabbay, and J. Woods, eds. Volume 9 in *History of Logic*. Elsevier, 2014.

[13] Lloyd, J.W. *Foundations of Logic Programming* Springer-Verlag, 1987 (2nd ed).

[14] J. A. Robinson A Machine-Oriented Logic Based on the Resolution Principle Journal of the ACM, vol. 12 (1965), pp. 23–41.

[15] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley, 2009.

A Appendix: Programs

```
 1 #include <stdio.h>
 2 #include <assert.h>
 3
 4 typedef unsigned QT; // quotient type
 5 class AM { // Archimedean Monoid
 6 public:
 7   double val; //floating-point for Archimedean Monoid
 8   AM (): val(0) {}
 9   AM (double val): val(val) {}
10   static AM zero() { return AM(0); }
11   friend AM operator+(const AM& x, const AM& y)
12     { return AM(x.val + y.val); }
13   friend AM operator-(const AM& x, const AM& y)
14     { return AM(x.val - y.val); }
15   friend bool operator<(const AM& x, const AM& y)
16     { return x.val < y.val; }
17   friend bool operator<=(const AM& x, const AM& y)
18     { return x.val <= y.val; }
19 };
20 bool aux(const AM& b, QT& m, AM& u,
21          const QT& n, const AM& v){
22   if (v < b) { m = 2*n; u = v; return true; }
23   if (b <= v) { m = 2*n+1; u = v - b; return true; }
24   return false;
25 }
26 bool q(const AM& a, const AM& b, QT& m, AM& u){
27   assert(0 <= a && 0 < b);
28   if (a < b) { m = 0; u = a.val; return true; }
29   if (b <= a && a < b+b) {
30     m = 1; u = a-b; return true;
31   }
32   if (b+b <= a) { QT n; AM v;
33     return (q(a, b+b, n, v) && aux(b, m, u, n, v)) ?
34     true : false;
35   }
36   return false;
37 }
38
39 int main() {
40   AM a(1000000001.1), b(17), u; QT m;
41   if (q(a, b, m, u)) {
42     printf("%d %lf\n", m, u.val/b.val);
43     printf("%lf\n", a.val/b.val);
44   } else assert(false);
45 }
```

Figure 5: The C++ program for quotient and remainder in Archimedean monoids transcribed from Theorem 1.