Compositional Semantics
for the Procedural Interpretation of Logic

M.H. van Emden

Department of Computer Science
University of Victoria
Research Report DCS-307-IR

Abstract

The composition of logic programs out of clauses has been studied semantically, but not the composition of a single clause out of its components. Structurally, a logic program can be regarded as a sentence in clausal form. In his procedural interpretation of logic programs, Kowalski has shown that a positive Horn clause can be viewed as a procedure in the programming sense. This interpretation suggests a composition operator for logic programs, the one where a clause results from composing a head with a body. In this paper we give more detail to the procedural interpretation by giving an algebraic characterization of Kowalski's composition. In addition, we give algebraic characterizations of the composition of goals in a procedure body and for the composition of the predicate symbol with the argument tuple within a goal. A starting point for the semantic operator corresponding to composition of goals is provided by Tarski's cylindric algebra semantics for first-order predicate logic. Tarski's construction is briefly reviewed and suitably modified. The additional semantic operators are shown to be correct with respect to the fixpoint semantics of the logic program as a whole.

1 Introduction

This paper concerns the semantics of the part of Prolog that remains when the built-in predicates have been removed and when unification is enhanced by the occurs check. Let us call this part "pure Prolog". It can be regarded as the result of Kowalski's procedural interpretation of positive Horn clauses [15, 14, 9]. The semantics of pure Prolog has been given by proof theory, by model-theory, and by a fixpoint method [24, 17, 2]. All three approaches follow the syntax of clausal form. As a result, the procedural interpretation has been ignored. The purpose of the present paper is to remedy this defect.

One of the symptoms of the current deficiency in the semantics of Prolog is that procedures can only be recognized in an informal way. As it stands, the procedural interpretation does not provide procedure-

valued expressions that can be substituted for the procedure symbol in a procedure call. Procedures are not "first-class citizens" the way functions can be in functional programming [22].

Compositional semantics does provide this possibility. According to this method, programs are expressions, consisting, if composite, of an operation and its operand(s). The value of the composite expression is the result of the operation on the values of its operands. The method is taken for granted when doing school-room sums: the value of $(4 \div 2) \times (1 + 1)$ is 4 because the value of $4 \div 2$ and $1 + 1$ are both 2 and because $2 \times 2 = 4$. In the late sixties Landin [16] and Scott and Strachey [21] applied the method to expressions that are programs.

In logic programming, compositional semantics seems to have been used only for elucidating how the union of two logic programs affects the definition of a predicate [7, 5]. In this paper we identify the compositions that occur within a clause and give a compositional semantics for these.

There are several advantages to a compositional semantics for a programming language. One such advantage is shared with denotational semantics (which is, among other things, compositional): that it can *guide implementation* [20]. The Warren Abstract Machine [25, 1] is an ingeniously optimized theorem-prover. It is notoriously unsuited to logic programs where some relations are relations in the sense of, and of the size typical in, relational databases. The compositional semantics presented here decomposes logic programs down to single procedure symbols, which take relations as value. This accommodates relations that are not defined in the logic program itself.

Another advantage of compositional semantics is that it forces *a language to be modular*. For example, in a functional language with compositional semantics $E_0 E_1$ is the result of applying the value of E_0, which must be a function, to the value of E_1, which may or may not be a function. The result can be a function, but need not be.

Compositionality requires that the value of $E_0 E_1$ does not change when E_0 is replaced by a different expression with the same value. This forces modularity in the sense that names of auxiliary functions occurring in E_0 do not affect its value, hence are local. Compositional semantics endows logic programs with the same property. The value of a procedure call $p(t_0, \ldots, t_{n-1})$ is obtained by an operation on the value of p (which is a relation) and the argument tuple $\langle t_0, \ldots, t_{n-1} \rangle$. Again, the result depends on the value of the relational expression substituted for p, not on the expression itself.

Contributions of this paper When one attempts a compositional semantics for the procedural interpretation of logic, it becomes apparent that it needs development beyond Kowalski's original formulation. This is done in Section 3.

Section 4 contains no contributions. It needs to be included because cylindric set algebras are a compositional semantics for first-order predicate logic and hence are a candidate for compositional semantics for the procedural interpretation of logic. This section includes enough to show why these algebras are not suitable. We do find, however, an interesting

connection between the tables introduced here and the cylinders of Tarski (see Theorem 2).

Tables, their operations and some of their properties are described in Section 5. This is the basis on which the compositional semantics of Section 6 rests. Implications for modularity are discussed in Section 7.

2 Notation and terminology

In this section we collect terminology and notation that may differ between authors.

2.1 General terminology

Definition 1 (tuple, function, index set, type, restriction, subtuple)
A tuple *is a function t that maps every index i to $t(i)$, which is called the tuple's component at i.*

A function *is a triple consisting a set that is its* domain*, a set that is its* co-domain*, and a* mapping *that associates with every element of the domain a unique element of the co-domain.*

If the function is a tuple, then the domain is usually called "index set".

The set of all functions with domain S and co-domain T is denoted $S \to T$. This set is often referred to as the type *of the functions belonging to it.*

Let f be a function in $S \to T$ and let S' be a subset of S. $f \downarrow S'$ is the restriction *of f to S'. It has S' as domain, T as co-domain and its mapping associates $f(x) \in T$ with every $x \in S'$.*

If t is a tuple with index set I and if I' is a subset of I, then $t \downarrow I'$ is the subtuple *of t defined by I'.*

Definition 2 (relation) *A* relation *with index set I and co-domain T is a set of tuples that have I as index set and T as co-domain. An n-ary relation is a relation that has as index set the set $\{0, \ldots, n-1\}$ of integers.*

Note that a relation need not be an n-ary relation for any n. Indeed, *any set can be the index set of a relation.*

Definition 3 (projection, cylindrification) *Let r be a relation that has I as index set. Let I' be a subset of I. The* projection $\pi_{I'}(r)$ *of r on I' is $\{t \downarrow I' \mid t \in r\}$.*

The cylinder in I *on a relation r' with index set I' is denoted $\pi_I^{-1}(r')$ and is the greatest relation with index set I and co-domain T that has r' as its projection on I'; that is*

$$\pi_I^{-1}(r') = \cup\{\rho \mid \pi_{I'}(\rho) = r' \text{ and } \rho \text{ has index set } I \text{ and co-domain } T\}$$

2.2 Mathematical objects arising in connection with the semantics of logic programs

To serve as semantic objects, three basic objects are defined independently of another; all three are mutually disjoint sets:

- $\mathcal{H}$, an Herbrand universe

- $\mathcal{V}$, a set of variables

- $\mathcal{P}$, a set of predicate symbols, also called "procedure symbols"

From the three basic objects the following are derived:

- $\mathcal{T}_V$, the set of terms that contain no function symbols or constants other than those occurring in $\mathcal{H}$ and no variables other than those occurring in a subset V of $\mathcal{V}$. We write $\mathcal{T}$ for $\mathcal{T}_\mathcal{V}$.

- Substitutions, each of which is a tuple of type $V \to \mathcal{T}_V$, for some subset V of $\mathcal{V}$. If θ is a substitution and $\theta(x) = t$, then we say that θ *substitutes t for x*. We may equate θ with the set $\{x = t \mid \theta(x) = t \text{ and } x \in V\}$ of term equations.

- *Term equations* are equations of the form $t_0 = t_1$, where t_0 and t_1 are terms belonging to $\mathcal{T}$. A set of term equations is said to be in *solved form* if every left-hand side is a variable, and if all these variables are different, and if all variables in the right-hand sides also occur as a left-hand side. If a set of term equations has a solution, then it has a solution in solved form.

  We will not distinguish between term equations in solved form, substitutions, and tuples of elements of T with a subset of $\mathcal{V}$ as index set.

- Relations consisting of tuples of elements of $\mathcal{H}$ that are indexed by $\{0, \ldots, n-1\}$. To distinguish these from the next item, we refer to them as *integer-indexed relations*.

- Relations consisting of tuples of elements of $\mathcal{H}$ that are indexed by a subset V of $\mathcal{V}$ that is characteristic of the relation. We refer to these as *variable-indexed relations*.

- The *Herbrand base*, which is the set of ground atoms.

- *Herbrand interpretations*, which are subsets of the Herbrand base.

- *Relational interpretations*, which are tuples of integer-indexed relations indexed by $\mathcal{P}$.

2.3   Compositional semantics

Compositional semantics assigns the *value $\mathcal{M}(E)$* to the *expression E*. We are interested in expressions that are programs. In this case the value is the behaviour of the program. As "value" and "behaviour" do not match very well, we often use "meaning" instead of "value" as a more neutral term. It also happens to fit well with "semantics".

Compositionality of the semantics means that if E is composed of subexpressions E_0 and E_1, then $\mathcal{M}(E)$ is the result of an operation on $\mathcal{M}(E_0)$ and $\mathcal{M}(E_1)$. An illustrative example [21] is found in the compositional semantics of binary numerals. It specifies how integers are assigned as meanings to binary numerals:

$$\mathcal{M}(\texttt{0}) = 0; \mathcal{M}(\texttt{1}) = 1; \mathcal{M}(\texttt{N0}) = 2\mathcal{M}(\texttt{N}); \mathcal{M}(\texttt{N1}) = 2\mathcal{M}(\texttt{N}) + 1$$

3 The procedural interpretation of positive Horn clauses

3.1 The original procedural interpretation

Kowalski [15] gives the procedural interpretation of Horn clauses as follows:

> A Horn clause $B \leftarrow A_1, \ldots, A_m$, with $m \geq 0$, is interpreted as a *procedure* whose *body* $\{A_1, \ldots, A_m\}$ is a set of procedure calls A_i. Top-down derivations are *computations*. Generation of a new goal statement from an old one by matching the selected procedure call with the *name* B of a procedure $B \leftarrow A_1, \ldots, A_m$ is a *procedure invocation*.
>
> A *logic program* consists of a set of Horn clause procedures and is activated by an initial goal statement.

Its semantics can be given as:

> The ground substitution θ is a result of activating program P with goal G if $P \cup \{G\theta\}$ is false in all Herbrand interpretations.

For a more general characterization, see [6, 17, 2].

3.2 A complete procedural interpretation

The procedural interpretation of logic can be formalized by expressing it as a procedural *syntax*. Kowalski proposed, in effect, $B \leftarrow A_0, \ldots, A_{m-1}$ as an alternative syntax for $\{B, \neg A_0, \ldots, \neg A_{m-1}\}$. To make the procedural interpretation not only formal, but also to complete it, we propose the syntax in Definition 4.

Definition 4 (procedural program)   *1. A procedural program is a tuple of procedures with index set $\mathcal{P}^1$.*

    2. A n-ary procedure is a set of n-ary clauses.

    3. An n-ary clause is a pair consisting of a parameter tuple of order n and a procedure body.

    4. A procedure body is a set of procedure calls.

    5. A procedure call is a pair consisting of an n-ary procedure symbol and an argument tuple of order n.

    6. A parameter tuple of order n and an argument tuple of order n are both n-tuples of terms.

Let us consider as example a set $\mathcal{P} = \{\texttt{app},\texttt{mem}\}$ of procedure symbols and the procedural program in Figure 1; let us call is p. As p is a tuple with $\mathcal{P}$ as index set, and as a tuple is a function, p can be specified by
$p(\texttt{app}) = \{(\texttt{nil,y,y}) \texttt{ :- } \{\}, \texttt{ (u.x,y,u.z) :- } \{\texttt{app(x,y,z)}\}\}$
$p(\texttt{mem}) = \{(\texttt{x,y}) \texttt{ :- } \{\texttt{app(u,x.v,y)}\}\}$

---

[1]The procedure symbols in $\mathcal{P}$ index only one procedure. This differs from Prolog where predicate symbols include an arity.

```
app(nil,y,y).                         {app{(nil,y,y) :- {}
app(u.x,y,u.z) :- app(x,y,z).             ,(u.x,y,u.z) :- {app(x,y,z)}
                                          }
mem(x,y) :- app(u,x.v,y).           ,mem{(x,y) :- {app(u,x.v,y)}}
                                    }
```

Figure 1: A Prolog program (left) and an equivalent procedural program (right).

By itself, Definition 4 defines *some* procedural language. It is only of interest in so far as it is related to clausal logic. Similarly, the relational interpretations for procedural programs need to be related to Herbrand interpretations. Hence the following definition.

Definition 5 (correspondence between logic and procedural programs)
An Herbrand interpretation I and a relational interpretation R correspond to each other ($I \sim R$) iff the following holds:
$R(p) = \{\langle a_0, \ldots, a_{n-1}\rangle \mid p(a_0, \ldots, a_{n-1}) \in I\}$ *for all $p \in \mathcal{P}$ and*
$I = \{p(a_0, \ldots, a_{n-1}) \mid p \in \mathcal{P} \text{ and } \langle a_0, \ldots, a_{n-1}\rangle \in R(p)\}$
Let S be a sentence consisting positive Horn clauses (for which we assume Kowalski's notation). Let P be a procedural program. S and P correspond to each other ($S \sim P$) iff the following holds:
$P(p) = \{partuple \text{ :- } body \mid p(partuple) \leftarrow body \in S\}$ *for all $p \in P$ and*
$S = \{p(partuple) \leftarrow body \mid \exists p \in \mathcal{P} \text{ such that } partuple \text{ :- } body \in P(p)\}.$

Each of the syntactical rules of Definition 4 specifies that a certain type of expression is composed of sub-expressions. Compositional semantics then assigns to each of syntactical rules a semantical rule that specifies the corresponding operation on meanings of the constituent sub-expressions.

The next section introduces the mathematical objects that are suitable meanings. Section 6 describes the semantical rules.

Before starting on this we give an informal idea of what is involved. Let us work through the items in Definition 4, starting at the bottom.

Rule 5   Consider the atoms $p(x, v, w)$ and $p(u, w, y)$. Although both involve the same relation p, they are different calls and have different meanings. These meanings are the result of a binary operation with the relation p and the tuple of arguments as operands.

The meaning of the entire call can be viewed as a selection from the tuples that constitute relation p. The selection is specified by the argument tuple, and selects the tuples from the relation that match the argument tuple. Each such match takes the form of a substitution for the variables in the argument tuple. Therefore the result of the operation, which we call *filtering*, is a set of such substitutions.

As such sets are best presented in tabular form, we call the result of the the filtering operation on a relation and an argument tuple a

table (see Definition 6).

Rule 4 We define the *product* operation on tables (see Definition 8) by means of which procedure bodies obtain values. These values are tables. Theorem 2 shows how product is related to the semantic counterpart of conjunction in Tarski's cylindric set algebra.

Rule 3 The meaning of a *clause* is the n-ary relation that results from an operation on the meanings of the constituents of the clause: the parameter tuple and the body. As a parameter tuple has itself as meaning, we define an operation, which we call *projection*, on a parameter tuple of order n and a table (see Definition 11). The operation yields an n-ary relation.

This completes the preview of the novel semantic operations: filtering, product, and projection. The remaining operations, those arising from Rules 1 and 2, will not require any explanation beyond the following few lines. In Rule 2, a procedure symbol is combined with a set of clauses. As the meaning of a clause is an n-ary relation, a set of such clauses denotes the union of these relations, that is, an n-ary relation again. Rule 2 merely creates a pair consisting of a procedure symbol and a relation.

Rule 1 combines into a set a number of procedures, each of which is a pair of a procedure symbol and a relation. The semantic object corresponding to a program is therefore a tuple of procedures indexed by $\mathcal{P}$, the set of procedure symbols.

4   Compositional semantics for logic

Though there does not seem to exist any compositional semantics for the procedural interpretation of logic, one does exist for logic that is parsed in the conventional way. It is called *algebraic logic*, so named by Halmos [10]. Algebraic logic is what would be called compositional semantics if it would concern a programming language[2]. Therefore, algebraic logic is a good starting point for a compositional semantics of logic programs.

Algebraic logic assigns elements of an algebra as meanings to formulas of logic; it assigns operations of the algebra as meaning to the connectives that compose logical formulas. A better known alternative to the meaning algebras used by Halmos are the cylindric set algebras of Tarski [11, 23] of which we give a brief sketch here. Tarski's approach is based on the algebraic interpretation of propositional logic due to Boole [3].

4.1   Propositional logic and Boolean algebra

In general, a Boolean algebra is any algebra that satisfies certain defining axioms. A Boolean *set* algebra is a special case. It is described as the tuple $\langle S, \cup, \cap, \sim, \emptyset, U \rangle$ where S is a set of subsets of U that contains $\emptyset$ and

---

[2]This comparison does not do justice to algebraic logic. The translation of logic to algebra is only the first step in algebraic logic. The goal is to use *substantial* results in algebra to obtain, by mere translation, *important* results in logic that had so far only been proved in an *ad hoc* manner.

U and is closed under union, intersection, and complementation (here denoted as $\sim$).

A special case of a Boolean set algebra is the one where U is the Cartesian product D^n, for some given non-empty set D. Recall that the Cartesian product D^n is the set of all n-tuples of elements of D.

In the Boolean set algebra, we choose $U = D^0 = \{\langle\rangle\}$ and $S = \{\{\}, \{\langle\rangle\}\}$. As a result, the algebra has two elements: $\{\}$ and $\{\langle\rangle\}$. Boolean addition, multiplication, and complementation then become set union, set intersection, and set complement, respectively. Let $\mathcal{M}$ be the mapping from propositional formulas to the elements of the Boolean algebra. We have that $\mathcal{M}(p_0 \vee p_1) = \mathcal{M}(p_0) \cup \mathcal{M}(p_1)$, $\mathcal{M}(p_0 \wedge p_1) = \mathcal{M}(p_0) \cap \mathcal{M}(p_1)$, and $\mathcal{M}(\neg p) = \sim \mathcal{M}(p)$ when we define $\mathcal{M}(true) = \{\langle\rangle\}$ and $\mathcal{M}(false) = \{\}$.

4.2   Predicate logic and cylindric set algebra

Tarski sought an algebra that would do for first-order predicate logic what Boolean algebra does for propositional logic. The result was cylindric set algebra [23, 11].

In model theory, formulas correspond to relations. If this intuitively attractive feature is to be retained, a puzzle needs to be solved. Consider $\mathcal{M}(p(x, y) \wedge p(y, z))$. As the formula has three free variables, this should be a ternary relation. As conjunction means the same in predicate logic as in propositional logic, this ternary relation should be the result of set intersection. But the arguments of the set intersection are derived from binary predicates.

Another part of the puzzle is that $p(x, y)$ and $p(y, z)$ should both denote binary relations, but these should be different and cannot both be the relation denoted by p.

Tarski solved these conundrums by mapping every formula to a relation consisting tuples indexed by *all* the variables in the language. He assumed a countable infinity of variables in the language, in a given order. In this way he could identify each variable with a natural number. Thus this meaning algebra has as elements relations that are subsets of the Cartesian product D^ω.

The choice of the two 0-ary relations on D for the two elements of the Boolean algebra for propositional logic is now clear: the number of variables in a propositional formula is 0.

A first-order predicate logic formula without free variables is either true or false. It is mapped accordingly to the full or empty ω-ary relation over D; that is, to D^ω or $\emptyset$. At first sight it might seem right to map a formula $F[x_0, \ldots, x_{n-1}]$ with free variables $x_0, \ldots, x_{n-1}$ to the relation that consists of all the tuples $\langle a_0, \ldots, a_{n-1} \rangle$ such that $F[a_0, \ldots, a_{n-1}]$ is true. By mapping instead this formula to the *cylinder* on this relation with respect to all variables, Tarski ensured that $\mathcal{M}(p_0 \vee p_1) = \mathcal{M}(p_0) \cup \mathcal{M}(p_1)$ and $\mathcal{M}(p_0 \wedge p_1) = \mathcal{M}(p_0) \cap \mathcal{M}(p_1)$, just as in the case of propositional logic.

Going back to the above puzzle, we see that $\mathcal{M}(p(x, y))$ and $\mathcal{M}(p(y, z))$ are not binary relations but ω-ary relations that are cylinders on a binary relation. Though the binary relation denoted by p in these formulas is the

same, the cylinders on $\mathcal{M}(p(x,y))$ and $\mathcal{M}(p(y,z))$ are different. In this way $\mathcal{M}(p(x,y)) \cap \mathcal{M}(p(y,z))$ is a cylinder on a ternary relation.

Thus Tarski devised a compositional semantics for first-order predicate logic. He simplified the language to contain as connectives only conjunction, disjunction, and negation. The presence of the negation connective makes it possible to do with a single quantifier, the existential one. There are no function symbols. An atomic formula can be of the form $x = y$.

For this language a suitable algebra for a compositional semantics is the *cylindric set algebra* $\langle S, \cup, \cap, \sim, \emptyset, D^\omega, C_k, \delta_{i,j} \rangle$ for all natural numbers i, j, and k. This algebra is a Boolean algebra (for the first six items). In addition, there are $\delta_{i,j}$, the i,j diagonal relations: the subsets of D^ω consisting of the tuples where the elements indexed by i and j are equal. The specification of cylindric set algebras also includes for all $k \in \omega$ the cylindrification operations C_k, which are defined by $C_k r$ being the subset of D^ω consisting of the tuples that differ from a tuple in r in at most the k-th component.

S is the set that contains $\emptyset$, D^ω, as well as all the diagonal relations $\delta_{i,j}$ and that is closed under the Boolean operations as well as under C_k.

4.3 Cylindric set algebra for the compositional semantics of procedural programs?

Cylindric set algebra interprets formulas as relations; relations are a suitable model for the procedures of a procedure-oriented language. These facts might suggest that cylindric set algebras be used for a compositional semantics for the procedural interpretation of logic.

The following are reasons not to do so.

- Tarski's choice of language for first-order predicate logic is no more procedure-oriented than clausal form is.

- Tarski's semantics does not specify by what operation, for example, the binary relation $\mathcal{M}(p(x,y,x))$ arises from the ternary relation p and the argument tuple $\langle x, y, x \rangle$. That is, his compositionality stops short of the atomic formula.

Accordingly, we create an independent alternative, centered around the concept of *table*. Surprisingly, one of the operations on tables reflects the way Tarski uses cylinders to algebraize conjunction.

5 Tables

Some of the semantic objects for the procedural programs of Definition 4 are familiar; they have been introduced in Section 2. This section is devoted to the one novel type of semantic object.

Definition 6 (table) *A table on a subset V of $\mathcal{V}$ is a set of tuples each of which has type $V \to \mathcal{T}_V$. If the set of tuples is empty, then we have the* null *table, which we write as $\bot$. If V is empty and the set of tuples is not, then the table is the* unit *table, which we write as $\top$.*

As there is only one function of type $\{\} \to \mathcal{T}_V$ for any subset V of $\mathcal{V}$, we have that $\top = \{\langle\rangle\}$.

To every table there corresponds a unique variable-indexed relation, which we call the result of *grounding* the table.

Definition 7 (grounding, table equivalence) *Let t be a table with tuples of type $V \to \mathcal{T}_V$. $\Gamma(t)$, the result of grounding t, is the variable-indexed relation consisting of the tuples of type $V \to \mathcal{H}$ each of which is a ground instance of a tuple in t.*

Tables t_0 and t_1 are equivalent *if $\Gamma(t_0) = \Gamma(t_1)$.*

In this section we define and discuss the product, filtering, and projection operations. These operations are adapted from [12], where filtering is called "application".

5.1   Product

As we will see, compositional semantics assigns tables as values to the calls in a procedure body as well as to the body itself. The co-occurrence of calls in a body corresponds to the *product* operation of the corresponding tables. An example will be given in Section 6.1.

Definition 8 (product) *Let τ_0 and τ_1 be tables consisting of tuples with index sets V_0 and V_1, respectively. The* product *$\tau_0 * \tau_1$ of these tables is defined as a table with $V_0 \cup V_1$ as index set. The product table $\tau_0 * \tau_1$ contains a tuple t if and only if there is a tuple t_0 in τ_0 and a tuple t_1 in τ_1 such that the set of equations $t_0 \cup t_1$ is solvable and has t as solved form.*

Theorem 1   • *Product is commutative and associative.*

- *The null table $\bot$ is an absorbing element: $\bot * \tau = \tau * \bot = \bot$ for all tables τ.*

- *The top table $\top$ is a unit: $\top * \tau = \tau * \top = \tau$ for all tables t.*

- *$\tau * \tau$ and τ are equivalent.*

Commutativity and associativity give the obvious meaning to $*S$, where S is a set of tables, assuming that $*\{\} = \top$.

Definition 9 (cylinder on table) *The cylinder $\pi^{-1}(T)$ on a table T with index set $V \in \mathcal{V}$ is a table where $\mathcal{V}$ is the index set and where every tuple t' is obtained from a tuple t in T by defining $t'(v) = t(v)$ for every $v \in V$ and $t'(v) = v$ for every $v \in \mathcal{V} \setminus V$.*

This definition of "cylinder" is independent of Tarski's notion, which is the one in Definition 3. The two notions are connected as follows.

Lemma 1 *Let T be a table with index set V, a subset of $\mathcal{V}$. We have that $\Gamma(\pi^{-1}(T)) = \pi^{-1}(\Gamma(T))$. The first occurrence of π^{-1} is the cylindrification on tables from Definition 9; the second one is Tarski's cylindrification on relations as in Definition 3.*

The distinguishing feature of Tarski's use of cylindric set algebra as semantics for first-order predicate logic is that conjunction in logic simply translates to intersection in the algebra. And this is the case even though the conjunction may be between two formulas with sets V_0 and V_1 of free variables. There is no restriction on these sets: they may be disjoint, one may be a subset of the other, or neither may be the case. Tarski's device works because the intersection is not between relations with V_0 and V_1 as index sets, but between *cylinders* on these relations in the set of all variables. This crucial idea reappears in the product of tables defined here. The connection is made apparent by the following theorem.

Theorem 2 *Let τ_i be a table with set V_i of variables, for $i \in \{0, 1\}$.*
$$\Gamma(\tau_0 * \tau_1) = \pi_{V_0 \cup V_1}(\pi_{\mathcal{V}}^{-1}(\Gamma(\tau_0)) \cap \pi_{\mathcal{V}}^{-1}(\Gamma(\tau_1))).$$

5.2   Filtering: from relations to tables

Just as in a functional programming language a function is applied to the n-tuple of its arguments, we think of the combination of a procedure symbol with its argument tuple as an binary operation. Consider therefore a call consisting of a procedure symbol and an argument tuple of order n. The procedure symbol has as value an integer-indexed relation of order n. It combines with the argument tuple to produce a table. This is the operation we call *filtering*. An example of this operation can be found in Section 6.1.

Definition 10 (filtering) *Let p be an integer-indexed relation of order n and let t be an n-tuple of terms with V as set of variables. The result of the filtering $p : t$ is a table where V is the index set of the tuples. For every tuple $\langle a_0, \ldots, a_{n-1} \rangle$ in p for which the set $\{t_0 = a_0, \ldots, t_{n-1} = a_{n-1}\}$ equations is solvable, the table contains a tuple which is the solved form of these equations.*

In functional programming, an expression $E_0 E_1$ can denote function application. Here E_0 is an expression that evaluates to a function, and it is this function that is applied. Filtering is the relational counterpart: in $p : t$ the first operand p has a relation as value; it is filtered by the tuple t; the result is a table.

5.3   Projection: from tables to integer-indexed relations

Finally, a clause is a contribution to a procedure, which is an integer-indexed relation of order n. This relation, which is the clause's value, is somehow produced by a combination of the parameter tuple of the clause and the table that is the value of its body. We call this operation *projection*. An example of this operation can be found in Section 6.1.

Definition 11 (projection) *Let T be a table consisting of tuples whose index set is a subset V of $\mathcal{V}$. The result of projecting T on an n-tuple of terms, denoted $\langle t_0, \ldots, t_{n-1} \rangle / T$, is an integer-indexed relation consisting of n-tuples of ground terms. The relation contains such a tuple if and only if it is a ground instance of $\langle t_0 \theta, \ldots, t_{n-1} \theta \rangle$, for some θ in T.*

6 Compositional semantics

The operations of product, filtering, and projection are intended to be the semantical counterparts of the way in which procedural programs are put together syntactically. But so far only the intention exists.

Definition below formalizes this intention. It defines the meaning $\mathcal{M}(P)$ of a procedural program P as a tuple with index set $\mathcal{P}$ of integer-indexed relations. This meaning depends on a relational interpretation I (Definition 5) that assigns relations to the procedure symbols in $\mathcal{P}$. We indicate this dependence by a subscript, as in $\mathcal{M}_I$.

Definition 12 gives the compositional semantics for procedural programs. As Definition 5 shows, a procedural program is just another way of writing a set of positive Horn clauses. The semantics of these has been defined in three equivalent ways: model-theoretically, proof-theoretically, and by means of fixpoints. The main theorem (3) of this paper relates the compositional semantics of procedural programs to the established semantics of the corresponding clausal sentences.

Definition 12   *1. For every procedural program prog, $\mathcal{M}_I(prog)$ is the tuple with index set $\mathcal{P}$ such that for every $prsym \in \mathcal{P}$ the prsym-component is $\mathcal{M}_I(prog(prsym))$.*

   2. For every procedure proc, $\mathcal{M}_I(proc) = \cup\{\mathcal{M}_I(clause) \mid clause \in proc\}$

   3. For every clause with pars as parameter tuple and B as body, $\mathcal{M}_I(pars \; \text{:-} \; B) = pars/\mathcal{M}_I(B)$ (use of projection)

   4. For every procedure body B, we have $\mathcal{M}_I(B) = *\{\mathcal{M}_I(call) \mid call \in B\}$ (use of table product)

   5. For every call with prsym as procedure symbol and args as argument tuple, $\mathcal{M}_I(prsym \; args) = \mathcal{M}_I(prsym) : args$ (use of filtering)

   6. For every $prsym \in \mathcal{P}$ we have that $\mathcal{M}_I(prsym) = I(prsym)$

Here the numbering follows that of the syntactical rules of Definition 4.

Theorem 3 *Let I be a relational interpretation and I' the corresponding (Definition 5) Herbrand interpretation. Let P be a procedural program and P' the corresponding (Definition 5) set of positive Horn clauses. We have*

$$T_{P'}(I') \sim \mathcal{M}_I(P),$$

where T is the immediate-consequence operator for logic programs.

We only know a cumbersome, though straightforward, proof of this theorem.

T has a unique least fixpoint [24, 17, 2]. The partial order among Herbrand interpretations (set inclusion) translates according to the correspondence in Definition 5 to a partial order among relational interpretations (component-wise inclusion). Hence there is, for each procedural program P, a unique least relational interpretation I such that $I = \mathcal{M}_I(P)$.

Definition 13 $\mathcal{M}(P) = \mathcal{M}_{I_m}(P)$ *where I_m is the least relational interpretation I such that $I = \mathcal{M}_I(P)$.*

Theorem 4 *Let P' be a logic program and let P be the corresponding procedural program. Then we have $lfp(T_{P'}) \sim \mathcal{M}(P)$.*

This relates the compositional semantics of procedural programs to the mutually equivalent least fixpoint, proof-theoretical, and model-theoretical semantics of logic programs.

6.1 An example

Consider the procedural program clause
```
(f(y),z) :- {p(x,f(y)), p(f(x),z)}.
```
Here $\mathcal{M}(p)$ is an integer-indexed relation with $\{0,1\}$ as index set. Let us assume that $\mathcal{M}(p) = $

0	a	f(a)	f(a)	f(b)
1	f(b)	b	f(b)	f(a)

. Here the four 2-tuples, indexed by $\{0,1\}$, are displayed vertically.

The value of a call is a table; that is, a variable-indexed relation.
$$\mathcal{M}(p(x,f(y))) = \mathcal{M}(p) : \langle x, f(y) \rangle = $$

x	a	f(a)	f(b)
y	b	b	a

.

Similarly, $\mathcal{M}(p(f(x),z)) = \mathcal{M}(p) : \langle f(x), z \rangle = $

x	a	a	b
z	b	f(b)	f(a)

.

The value of the body is the product of the above two tables:
$$\mathcal{M}(p(x,f(y)),p(f(x),z)) = \mathcal{M}(p(x,f(y))) * \mathcal{M}(p(f(x),z)) = $$

x	a	a
y	b	b
z	b	f(b)

.

Finally, the meaning of the entire clause
```
(f(y),z) :- {p(x,f(y)), p(f(x),z)}
```
is obtained by projection:
$$\langle f(y), z \rangle / \mathcal{M}(p(x,f(y)),p(f(x),z)) = $$

0	f(b)	f(b)
1	b	f(b)

.

7 Implications for modularity

Suppose P and P' are procedural programs with the same Herbrand universe. If $p(t_0, \ldots, t_{n-1})$ is a call in P, then the meaning of p is $(\mathcal{M}(P))(p)$. But p is a special case of an expression that has an n-ary integer-indexed relation as value. Such an expression could also be $(\mathcal{M}(P'))(p')$ if p' is a procedure symbol in P' paired with an n-ary procedure. The value of this expression is a set of n-tuples of ground terms. This value is independent of the procedure symbols occurring in P'. Hence these symbols are "encapsulated" in the expression $(\mathcal{M}(P'))(p')$.

The module mechanism at work in this way is a natural one in the sense of being an unavoidable consequence of the compositionality of the semantics.

8 Related work

Modules for logic programs can be obtained via proof theory [19, 18]. An entirely different approach is to base it on decompositions of the

immediate-consequence operator as done by Brogi et al. [4]. It is baffling that the various approaches to modularity are so difficult to relate. Several more are mentioned by Brogi et al. [4], who also seem at a loss in relating them to their own work.

Additional details about the operations on tables and relations, there called "table-relation algebra", can be found in [12, 13].

9 Concluding remarks

The procedural programs of Definition 4 are the result of the desire to give a procedural interpretation not only of an entire clause, but also of the composition of head and body within a clause as well as of the compositions that can be recognized in the body. Thus procedural programs are but another way of parsing a set of positive Horn clauses.

But suppose that in 1972 one had never heard of clausal logic and that the motivation was to characterize in what way languages with procedures, such as Algol, are of a higher level than their predecessors. A higher level of programming in such languages is achieved by using procedure calls as much as possible. That suggests the ultimate altitude in level of programming: procedure bodies contain procedure calls only.

What about data structures for a pure procedural language? Just as Lisp simplified by standardizing all data structures to lists, one could make a similar choice by standardizing on trees. In this way a pure procedure-oriented language would arise that coincides with the procedural programs of Definition 4.

Functional programming languages have an obvious semantics in the form of functions as defined in mathematics. The semantics of Algol-like languages is defined in terms of transitions between computational states. These transitions are specified directly or indirectly in terms of assignments. In this way one might think that procedure-oriented programming languages are of inherently lower level than functional programming languages.

It is not necessary to specify procedures in terms of state transitions. A procedure is more directly specified as the set of all possible combinations of values of the arguments of a call. That is, as a set of tuples of the same arity, which is a relation.

In this way the procedural programs of Defintion 4 become as high-level as functional programs and obtain a semantics that is as mathematical.

One might argue that this gives procedural programs a significance that extends beyond logic programming. For example, they may be a way to describe Colmerauer's view [8] that Prolog is not necessarily a logic programming language. In the procedural interpretation described here, the Herbrand universe can be replaced by a sufficiently similar data structure, such as the rational trees.

10 Acknowledgements

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine.* MIT Press, 1991.

[2] K.R. Apt. Logic programming, 1990.

[3] George Boole. *An Investigation of the Laws of Thought.* Dover, 1854. Dover edition not dated.

[4] Antonio Brogi, Paolo Mancarella, Dino Pedreschi, and Franco Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16:1361– 1398, 1994.

[5] François Bry. A compositional semantics for logic programs and deductive databases. In *Proc. Joint Int. Conf. Symp. Logic Programming*, 1996.

[6] K.L. Clark. *Predicate Logic as a Computational Formalism.* PhD thesis, Imperial College, 1980.

[7] Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL)*, 1993.

[8] A. Colmerauer. Sur les bases théoriques de Prolog. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, 1979.

[9] A. Colmerauer, H. Kanoui, R. Paséro, and P. Roussel. Un système de communication homme-machine en français. Technical report, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II, 1972.

[10] Paul R. Halmos. *Algebraic Logic.* Chelsea, 1962.

[11] Leon Henkin, J. Donald Monk, and Alfred Tarski. *Cylindric Algebras, Parts I,II.* Studies in Logic and the Foundations of Mathematics. North-Holland, 1985.

[12] H. Ibrahim and M.H. van Emden. Towards applicative relational programming. Unpublished draft., 1991.

[13] Husain Ibrahim. Applicative expressions for relational programming. Master's thesis, University of Victoria, 1992.

[14] R.A. Kowalski. Predicate logic as programming language. In *Proc. IFIP 74*, pages 569–574, 1974.

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

[16] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1963.

[17] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

[18] Francis G. McCabe. *Logic and Objects*. Prentice Hall, 1992.

[19] Dale Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79 – 108, 1989.

[20] Peter D. Mosses. Denotational semantics, 1990.

[21] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, pages 19–46, Polytechnic Institute of Brooklyn, 1971.

[22] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey approach to Programming Language Theory*. MIT Press, 1977.

[23] A. Tarski. A representation theorem for cylindric algebras. *Bull. Amer. Math. Soc.*, 58:65 – 66, 1952.

[24] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[25] David H.D. Warren. An abstract Prolog instruction set. Technical Report Technical Note 309, SRI International, 1983.