Logic Colloquium '86 (eds Drake and1 Truss), ©Elsevier, North-Holland 1988, 83–93

Cumulative Logic Programs and Modeling

ABSTRACT The formation process of pure logic programs over a first-order language is iterated to give rise to cumulative logic programs. Such programs turn out to be objects in a combinatory model and are therefore amenable to algebraic manipulation including equation solving. It is pointed out how this fact can be employed in a discipline of modeling for cooperative processes. Other results concern the representability of equational classes, universal classes and algorithmic classes as solution sets of a set of combinatory equations.

1 Combinatory Models

Combinatory algebras, although going back only half a century, belong among the fundamental structures of mathematics just as much as groups and fields. Their study has perhaps been hampered somewhat by the inherent self-referentialness of their most successful constructs and by the fact that there can be no nontrivial computable combinatory algebra. On the other hand, these structures are really ubiquitous since they arise whenever a mathematical structure is furnished with a notion of internal computability. While this fact usually remains hidden - e.g. in computable algebra through a regime of arithmetization, i.e. coding via Göedel numbers - it is becoming increasingly clear that the "true" computational nature of construction theory over a given algebraic or relational structure resides in a set of additional objects, "rules of construction" or "programs", which by themselves form a really quite transparent algebraic structure: just our combinatory algebras. True, this was perhaps not foremost in the minds of the promulgators of combinatory logic and algebra, Schönfinkel, Curry and Church, and perhaps this point of view needed the advent of computer science as well as the construction of mathematically transparent, i.e. set-theoretic, models (Plotkin [10], Scott [11], Engeler [4]). The theme of objectification or reification of construction rules is pushed a step further in the present chapter when we show that, and how, even the mathematical structures on which we compute can become objects in a combinatory algebra. And again, the conceptual link to computer science should not be far from our minds: data types are, in most higher programming languages, themselves objects of constructions, classical such as product types and current such as function types and recursive types, [9].

Definition 1 *An algebraic structure* $\mathcal{C} =< C, \cdot >$ *with one binary operation ("application") is* combinatory complete*, if for every term* $t(x_1.,,.x_n)$*, built up from variables* $x_1, \ldots, x_n$ *using the operation of application alone, there exists an element* T *in* C *such that for all elements* $a_1, \ldots, a_n$ *in* C *we have the identity*

$$t(a_1, \ldots, a_n) = \Big((\ldots (T \cdot a_1) \cdot a_2) \cdot \ldots \Big) \cdot a_n$$

Combinatory complete algebraic structures with more than one element are called *combinatory algebras*, [1].

Historical remark: Schönfinkel and Bernays reduced the above infinite set of existential axioms for combinatory algebras to just two. It is tempting (and historically not unfair) to compare the scheme of combinatory completeness to the full comprehension axiom of set theory and the reduction to finitely many cases to the von Neumann-Bernays axiomatization, in particular $\mathcal{K}$ and $\mathcal{S}$, to Gödel's operations F_i that build up the constructible universe. However, Gödel's construction (over the ordinals) gives only a relative consistency proof, while the term model of combinatory logic (built up by primitive recursion over the natural numbers) is proved consistent, i.e. nontrivial, via the Church-Rosser execution of Hilbert's program for combinatory logic.

In addition to the laws

$$\mathcal{S}xyz = xy(xz)$$

$$\mathcal{K}xy = x$$

which thus characterize combinatory algebras, we may have another element $\mathcal{L}$ in the algebra satisfying

$$\mathcal{L}xy = xy$$

$$\forall z(xy = yz) \supset \mathcal{L}x = \mathcal{L}y$$

Combinatory algebras with such $\mathcal{L}$ are called *combinatory models*, [1]. As we shall see, the set of cumulative logic programs forms an example of a combinatory model.

2 Logic Programs: Pure and Cumulative

Let A the set of atomic formulas of some first-order language. Ordinary pure logic programs are (finite) sets of Horn formulas

$$a_1 \wedge a_2 \wedge \ldots \wedge a_k \supset b, \quad k \underline{\geq} 0, \quad a_i, b \in A$$

usually written

$$b : -a_1, \ldots, a_n; \quad \text{or } b \leftarrow a_1, \ldots, a_n$$

as in PROLOG. We shall here employ a variant notation, viz.

$$\{a_1, \ldots, a_k\} \rightarrow b$$

and use the formal construction $\rightarrow$ iteratively. Thus,

$$G_0(A) = A$$
$$G_{i+1}(A) = G_i(A) \cup \left\{\alpha \rightarrow a : \alpha \text{ finite}, \alpha \underline{\subseteq} G_i(A), a \in G_i(A)\right\}$$
$$G(A) = \cup_i G_i(A)$$

Note that within this cumulative hierarchy pure logic programs are simply subsets of $G_1(A)$. *Cumulative logic programs* then, are subsets of $G(A)$.

Let D_A be the set of subsets of $G(A)$. We make D_A into an algebraic structure

$$\mathcal{D}_A =< D_A, \cdot >$$

by defining the binary operation $\cdot$ of application on D_A by

$$M \cdot N = \left\{a : \exists \alpha \underline{\subseteq} N. \quad \alpha \rightarrow a \in M\right\}$$

for arbitrary $M, N \in D_A$. It can be easily shown [4] that $\mathcal{D}_A$ is a combinatory algebra; indeed, with

$$\mathcal{L} = \left\{\{\beta \rightarrow u\} \rightarrow (\alpha \rightarrow u) : \beta \underline{\subseteq} \alpha \underline{\subseteq} G(A), u \in G(A), \alpha, \beta \text{ finite}\right\}$$

it is a combinatory model.

The application operation can be used to explain the execution of pure logic programs as a solution process for a corresponding fixpoint equation. To take an example, let us consider the following logic program for the factorial function:

The first-order language is that of number theory, variables $x, y, \ldots$ ranging over natural numbers, terms built up from variables by the operation successor $\mathcal{S}$. The language is augmented by a binary predicate $fac(u, v)$ intended to mean that v is the factorial of u, a ternary predicate $mult(u, v, w)$ for "$w = u \cdot v$", and similarly for addition. The program reads

$$fac(0,1):-$$

$$fac(\mathcal{S}(x),y):-fac(x,z),mult(\mathcal{S}(x),z,y)$$

$$mult(x,0,0):-$$

$$mult(x,\mathcal{S}(y),z):-mult(x,y,u),add(x,u,z)$$

$$add(x,0,x):-$$

$$add(x,\mathcal{S}(y),\mathcal{S}(z)):-add(x,y,z)$$

By our translation, this logic program becomes an element of $\mathcal{D}_A$, but not a finite one: care has to be taken that

$$\Big\{fac(x,z),mult(\mathcal{S}(x),z,y)\Big\}\rightarrow fac(\mathcal{S}((x),y)$$

is present not only for the variables x,y,z but for all terms of the language. Let $P\in D_A$ be formed in this way, i.e.

$$\begin{aligned}P\ :=&\Big\{\emptyset\rightarrow fac(0,1)\Big\}\\ &\bigcup\Big\{\{fac(t_1,t_3),mult(\mathcal{S}(t_1),t_3,t_2)\}\rightarrow fac(\mathcal{S}(t_1,t_2):t_1,t_2,t_3\in Terms\Big\}\\ &\bigcup\ldots\end{aligned}$$

Then $P\cdot\emptyset$, in the sense of the application operation of $\mathcal{D}_A$, yields those atomic formulas which one step of unification/resolution can obtain. $P^{(2)}\cdot\emptyset=P\cdot(P\cdot\emptyset)$ yields answers to those queries for which two steps suffice, etc. Indeed, the iteration

$$\cup_n P^{(n)}\cdot\emptyset$$

yields exactly all answers to possible queries. Now note the form of this expression: it represents the least fixpoint of the equation

$$P\cdot X=X$$

in $\mathcal{D}_A$. Thus, solving this fixpoint equation amounts to obtaining the results of all possible queries to the given logic program (this is the so-called fixed-point or denotational semantics of Prolog, see [2] for an independent formulation). What we have

shown, then, is that logic programming may be considered a discipline of equation-solving, very special equations to be sure, in the algebra $\mathcal{D}_A$. But we observe at once that this discipline, which is a form of backward search familiar from problem-solving à la Polya and from AI methodologies, is - with some modifications - applicable to a wide class of fixed point equations

$$f_i(X_1, \ldots, X_n) = X_i, \quad i = 1, \ldots, n$$

in the whole of $\mathcal{D}_A$. Details of this are given in the next section.

3 Continuity, Equation Solving and Process Modeling

In a separate paper [6] we put forward a method for the modeling of cooperative processes based on a representation of both processes and their interactions by elements of an appropriate domain $\mathcal{D}_A$ of cumulative logic programs. In outline, this representation proceeds as follows: Imagine two processes X and Y interacting in such a way that either depends on the other in a well-determined way. If we describe the known facts about X by a set of formulas, again denoted by X, and do the same with Y then the interaction between the processes takes on the form

$$X = f(X, Y)$$
$$Y = g(X, Y)$$

where f and g are set-functions transforming (pairs of) sets of formulas into sets of formulas. The most immediate case is to consider X and Y as subsets of $A = G_0(A)$, but more generally we should think of them as subsets of $G(A)$. More importantly, it is reasonable to expect that the functions f and g enjoy a certain continuity property:

$$f(X, Y) = \bigcup \left\{ f(\alpha, \beta) : \alpha \underline{\subseteq} X, \beta \underline{\subseteq} Y, \alpha, \beta \text{ finite} \right\}$$
$$g(X, Y) = \bigcup \left\{ g(\alpha, \beta) : \alpha \underline{\subseteq} X, \beta \underline{\subseteq} Y, \alpha, \beta \text{ finite} \right\}$$

This expectation is based on the constraints posed on modeling processes quite generally: we never have but finite amounts of information about the properties of the processes which we model, accumulating this information is what leads to our "knowledge" of X. In any case, postulating f and g to be continuous in this sense (a notion of continuity which can be linked to a well-known topology on $G(A)$), we have in $\mathcal{D}_A$ an important and useful fact:

f is continuous iff there is an $F \in D_A$ such that

$F \cdot u \cdot v = f(u, v)$ identically

The proof, well known in combinatory algebra and in denotational semantics, is given below for completeness in the present context: If f is continuous, define

$$F := \left\{ \alpha \to (\beta \to x) : x \in f(\alpha; \beta), \alpha, \beta \underline{\subseteq} G(A), \text{ finite} \right\}$$

and verify the identity. If, conversely, $f : D_A^2 \to D_A$ is given by

$$f(X, Y) = (F \cdot X) \cdot Y, \quad \text{for some} \quad F \in D_A$$

continuity follows by verifying

$$F \cdot X \cdot Y = \bigcup \left\{ F \cdot \alpha \cdot \beta : \alpha \underline{\subseteq} X, \beta \underline{\subseteq} Y, \alpha, \beta \text{ finite} \right\}$$

which is immediate from the definition of the application operation.

Thus, the interaction equations from above are rendered

$$X = F \cdot X \cdot Y, \quad Y = G \cdot X \cdot Y$$

for appropriate $F, G \in D_A$. Indeed, as we have pointed out elsewhere, the sets F and G are often quite easy to generate directly, and having obtained them, the least solution of the pair of fixpoint equations is the pair $X = \bigcup_i X_i, Y = \bigcup_i Y_i$, where $X_0 = Y_0 = \emptyset$ and $X_{i+1} = F \cdot X_i \cdot Y_i,\ Y_{i+1} = G \cdot X_i \cdot Y_i$.

Thus, whenever a set of cooperative processes lends itself to description of the form $X_i = f_i(X_1, \ldots, X_n), i = 1, \ldots, n$, for continuous f_i, we have here a full theoretical solution, one that generalizes the fixpoint semantics of its special subcase, logic programs. A practical solution, consisting in a discipline of evaluating such systems of fixpoint equations in analogy to interpreters of logic programs will be presented in a forthcoming paper.

Obviously, not all interaction equations for all types of cooperating processes are in the form of fixpoint equations. They arise quite often from some law such as physical laws (e.g. conservation laws or minimalization principles) which do not always yield autonomous differential equations. The typical case is rather of the form

$$f_i(X_1, \ldots, X_n) = g_i(X_1, \ldots, X_n), \quad i = 1, \ldots, n.$$

What does $\mathcal{D}_A$, as an algebraic structure, have to offer towards manipulating and perhaps solving such equations? Here is one result:

Lemma 2 *Let Γ be a finite set of equations in $\mathcal{D}_A$ of the form*

$$\begin{aligned} \forall y_1 \cdots y_n \cdot f_i(A_1, \cdots, A_k, X_1, \cdots, X_m, y_1, \cdots, y_n) \\ = g_i(A_1, \cdots, A_k, X_1, \cdots, X_m, y_1, \cdots, y_n), \\ f_i, g_i \text{ continuous} \quad i = 1, \ldots, p \end{aligned}$$

where A_j are constants of $D_A, X_1, \cdots, X_m$ are the unknowns and $y_1, \cdots, y_n$ are variables. Then we can effectively construct new constants A and B from the f_i, g_i, A_i such that solving Γ for $X_1, \cdots, X_m$ is effectively equivalent to solving the equation $A \cdot Z = B \cdot Z$ for Z.

But we have to beware of the enormous generality of the problem formulated here as that of solving one deceptively simple type of equation in an algebraic structure with deceptively simple fundamental laws. Very little progress has been reported in the general case ([5], [8], [9]).

4 Combinatory Representation of Model Classes

We now come to the claim that computations or "constructions" on algebraic or relational structures are most transparently thought of as additional elements of suitably expanded structure in which they obey rather simple algebraic laws. To this claim is added the further insight that various model classes, thus expanded, are describable as solutions of combinatory equations similar to the ones encountered in the previous section. The research program expressed in the above has been carried out for algebraic structures and equational classes in [7]; we report here on these results and expand them to relational structures, universal classes, and algorithmic classes.

To fix ideas, let

$$\mathcal{H}_0 =< H_0, R, f, c >$$

be a relational structure with universe H_0, binary relation R, binary operation f and distinguished element c. Consider a combinatory model

$$\mathcal{D}_A =< D_A, \cdot >$$

where for the present we disregard the particular structure of A and only ask that A is an infinite superset of H_0. To realize $\mathcal{H}_0$ within $\mathcal{D}_A$ we make use, as in [7], of a suitable element $h \in D_A$, which acts by left-multiplication as a retraction

$$h \cdot (h \cdot x) = h \cdot x, \quad \forall x \in D_A.$$

and whose retract $H = \Big\{ h \cdot x : x \in D_A \Big\} = \Big\{ x \in D_A : h \cdot x = x \Big\}$ consists of

$$H = \Big\{ G(A), \emptyset \Big\} \cup \Big\{ \{a\} : a \in H_0 \underline{\subseteq} A \Big\}.$$

This set serves as the universe of the representation of $\mathcal{H}_0$ in $\mathcal{D}_A$. Actually, what is represented is not $\mathcal{H}_0$ but its "completion"

$$\overline{\mathcal{H}}_0 =< \overline{H}_0, \overline{R}, \overline{f}, \overline{c} >$$

defined as follows:

$$\overline{H}_0 = H_0, \cup \{\bot, \top\}, \quad \bot, \top \notin H_0, \quad \bot \neq \top, \quad \bot, \top \in A.$$

$$\overline{R}(x,y) = \begin{cases} \textbf{true} & \text{if } x, y \in H_0 \text{ and } R(x,y) = \textbf{true} \\ \textbf{false} & \text{if } x, y \in H_0 \text{ and } R(x,y) = \textbf{false} \\ \bot & \text{if } x = \bot \text{ or } y = \bot \\ \top & \text{otherwise} \end{cases}$$

$$\overline{f}(x,y) = \begin{cases} f(x,y) & \text{if } x, y \in H_0 \\ \bot & \text{if } x = \bot \text{ or } y = \bot \\ \top & \text{otherwise} \end{cases}$$

$$\overline{c} = c, \quad (\in H_0 \underline{\subseteq} \overline{H}_0).$$

The binary operation f of $\overline{\mathcal{H}}_0$ is realized by another element of $\mathcal{D}_A$, which we denote again by f, and which satisfies

$$h \cdot (f \cdot (h \cdot x) \cdot (h \cdot y)) = f \cdot (h \cdot x) \cdot (h \cdot y), \quad \forall x, y \in D_A.$$

Such f is easily constructed from the multiplication table of f:

$$\begin{array}{rcl} f & := & \Big\{\{a\} \to (\{b\} \to f(a,b)) : a,b \in H_0\Big\} \\ & \cup & \Big\{\{a\} \to (\{b\} \to a) : a,b \in A,\ a \text{ or } b \notin H_0\Big\} \\ & \cup & \Big\{\alpha \to (\beta \to u) : u \in G(A), \emptyset \neq \alpha \neq \{a\}\ \forall a \in A \text{ or } \emptyset \neq \beta \neq \{b\}\ \forall b \in A\Big\} \end{array}$$

The realization of $c \in H_0$ in $\mathcal{D}_A$ is simply $\{c\}$ which we denote, by abuse of notation again, as c. So far we have simply restated the approach of [7]. What is added is the treatment of relations. For this we shall use two particular elements of $\mathcal{D}_A$, namely the combinators $\mathcal{K}$ and $\mathcal{KI}$ which will play the role of truth and falsity:

$$r \cdot \{a\} \cdot \{b\} = \begin{cases} \mathcal{K} \text{ if } R(a,b) \text{ is true and } a,b \in H_0 \\ \mathcal{KI} \text{ if } R(a,b) \text{ is false and } a,b \in H_0 \end{cases}$$

$\mathcal{K}$ satisfies the identity $\mathcal{K}xy = x$, and $\mathcal{KI}$ satisfies $(\mathcal{KI})xy = y$ in $\mathcal{D}_A$; thus the relation R is actually realized as a test r. Explicitly, r is given by

$$\begin{array}{rcl} r & := & \Big\{\{a\} \to (\{b\} \to u) : a,b \in H_0, R(a,b) \text{ true }, u \in \mathcal{K}\Big\} \\ & \cup & \Big\{\{a\} \to (\{b\} \to v) : a,b \in H_0, R(a,b) \text{ false } v \in \mathcal{KI}\Big\} \\ & \cup & \Big\{\alpha \to (\beta \to u) : u \in G(A), \emptyset \neq \alpha \neq \{a\}, a \in A \text{ or } \emptyset \neq \beta \neq \{b\}, b \in A\Big\} \end{array}$$

In this way, the relational structure $\mathcal{H}_0$ gives rise to four elements h, r, f, c of $\mathcal{D}_A$ which in turn determine a relational structure

$$\mathcal{D}_A[h; r, f, c]$$

where the Boolean values **true, false**, $\bot, \top$ of $\overline{R}$ in $\overline{\mathcal{H}}_0$ are realized by the values $\mathcal{K}, \mathcal{KI}, \emptyset$ and $G(A)$ in $\mathcal{D}_A$ respectively. It is easy to show that

$$\mathcal{D}_A[h; r, f, c] \cong \overline{\mathcal{H}}_0.$$

We now recall from [3] the notion of an algorithmic class of structures. For any similarity type of relational structure $\mathcal{H}_0 =< H, R, f, c >$ consider a programming language which embodies

assignments $\quad x_i := x_j, \quad x_i := c, \quad x_i := f(x_j, x_k)$

tests $\quad$ *if* $R(x_i, x_j)$ *then* ... *else* ...

corresponding to the operations and relations of the structures. If $\pi(x_1, \ldots x_n)$ is a program in this language, we interpret

$$\mathcal{H}_0 \models \pi(x_1, \ldots x_n)$$

to mean that π halts for whatever initial values from H_0 are given to π at the start of its execution. The following program is a typical example:

$z := x$;

if $x \underline{\leq} 0$ or $y \underline{\leq} 0$ *then* halt *else*

while $z < y$ *do* $z : z + x$ *od.*

This program halts for all x, y from an ordered abelian group iff this group is archimedean; the archimedean property is an "algorithmic property".

We call a class A of similar relational structures an algorithmic class if there is a set Π of programs such that

$$\mathcal{H}_0 \in A \text{ iff } \mathcal{H}_0 \models \pi \text{ for all } \pi \in \Pi$$

Even the most rudimentary control structures of a programming language suffice to show that every universal class (in the sense of model theory) is an algorithmic class; the above example shows that the converse is not true.

Now, if $\mathcal{H}_0$, or more precisely, $\overline{\mathcal{H}}_0$, is embedded in $\mathcal{D}_A$ we can make use of the combinators of $\mathcal{D}_A$ such as the composition operator B and the fix-point operator Y to compose operations f and tests r into elements of $\mathcal{D}_A$ which correspond to (recursive) programs on $\mathcal{H}_0$: For each program $\pi(x_1, \ldots, x_n)$ there is an element $P \in \mathcal{D}_A$ such that for $\overline{\mathcal{H}}_0 \cong \mathcal{D}_A[h; r, f, c]$ we have:

$$\mathcal{H}_0 \models \pi(x_1, \ldots, x_n) \text{ iff } P \cdot h \cdot r \cdot f \cdot c \cdot x_1 \cdot \cdots \cdot x_n = K \text{ identically in} \overline{\mathcal{H}}_0$$

P is obtained by a quite straightforward construction in combinatory algebra [1]. These equations, then, are the proposed translations of the proper axioms for the class A; they correspond to the translation of equations Eq in [7]. The remainder of the translation can be lifted from that paper and we have:

Theorem *For every set* Π *of programs and every similarity type of relational structures* $\mathcal{H}_0 =< H_0, R, f, c >$ *we can effectively find a set* Γ *of equations for elements* h, r, f, c *in* $\mathcal{D}_A$ *such that, disregarding cardinality restrictions:*

(a) For every model $\mathcal{H}_0$ of the algorithmic properties Π the realization $\mathcal{D}_A[h; r, f, c] \cong \overline{\mathcal{H}}_0$ is such that h, r, f, c satisfy Γ.

(b) If $h, r, f, c \in \mathcal{D}_A$ satisfy Γ and are nontrivial, then $\mathcal{D}_A[h; r, f, c] \cong \overline{\mathcal{H}}_0$ for some model $\mathcal{H}_0$ of the algorithmic properties Π.

The proof of this theorem is a simple extension of the proof of Theorem 2 [7].

Bibliography

[1] H. Barendregt. *The Lambda Calculus* North-Holland, Amsterdam, 2nd ed. (1984).

[2] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language, *J. ACM* 23 (1976), 733–742.

[3*] E. Engeler. Algorithmic properties of structures, *Math. Systems Theory* 1 (1967), 183–195.

[4*] E. Engeler. Algebras and combinators, *Algebra Universalis* 13 (1981), 389–392.

[5*] E. Engeler. Equations in Combinatory Algebras. In: *Proceedings of Logics of Programs '83* (ed. D. Kozen) *LN in Computer Science* 164, Springer (1984), 193–205.

[6] E. Engeler. Modeling of cooperative processes. In: *Comput. Theory and Logic* (ed. E. Börger) *LN Computer Science* 270, Springer (1987), 143–153.

[7*] E. Engeler. Representation of varieties in combinatory algebras, *Algebra Universalis* 25 (1988), 85–95.

[8] T. Fehlmann. *Theorie und Anwendung des Graphmodells der kombinatorischen Logik*. Berichte des Instituts fuer Informatik, ETH Zurich, No. 41 (1981).

[9] R. Maeder. Graph algebras, algebraic and denotational semantics, Dissertation ETH No. 8065 Zurich (1986).

[10] G. Plotkin. A set-theoretic definition of application, Memorandum MIP-R-95, School of AI, University of Edinburgh (1972).

[11] D. S. Scott. Models for the λ-calculus. Manuscript (1969).