Evolving Algebras and Linear Time Hierarchy*

Andreas Blass† and Yuri Gurevich‡

Keyword Codes: F.1.1; F.1.3; F.3.2
Keywords: Computation by Abstract Devices, Models of Computation;
Computation by Abstract Devices, Complexity Classes;
Logics and Meanings of Programs, Semantics of Programming Languages

1 Introduction

A program Q simulates a program P in *lock-step* under a given representation ρ of states of P as states of Q if there is a constant c such that if P transforms a state A to a state A^+ in time τ then Q transforms $\rho(A)$ into $\rho(A^+)$ in time $\leq c\tau$. (*Cf.* the related notions of real-time computation and simulation [R,S].) The constant c is the *lag factor* of the simulation. We speak about time rather than the number of steps because the number of steps may be too crude a measure of time and a more honest measure may be required. The representation function ρ may be multi-valued, in which case Q transforms any $\rho(A)$ into some $\rho(A^+)$. The inverse function ρ^{-1} is usually single-valued.

Neil Jones [J] calls a universal program U for a language L *efficient* if there is a constant c such that U simulates every L-program in lock-step with lag factor c. He exhibits a programming language I^{su} that admits an efficient universal program and then uses such a program and diagonalization to show that functions I^{su}-computable in linear time form an infinite hierarchy. To show the robustness of his results, Jones mentions a number of languages and machine models which admit efficient universal programs and give the same class $\mathrm{LIN}^{\mathrm{su}}$ of problems solvable in linear time. Among those machines models are the storage modification machines of Schönhage [S] and successor RAMs, that is, random access machines whose only arithmetical operation is the operation of computing the successor of a given natural number.

Jones's thought-provoking paper seems to open a possibility of proving linear time lower bounds for linear time computations. Unfortunately linear time on successor RAMs is too restrictive for many applications, e.g., computational geometry, and in general linear time is not very robust notion. Should one try to extend Jones's results to addition RAMs, or to addition and multiplication RAMs, or to addition, multiplication and division RAMs, or to something else?

*In "IFIP 13th World Computer Congress 1994, Volume I: Technology/Foundations", eds. B. Pehrson and I. Simon, Elsevier, Amsterdam.

†Partially supported by NSF grant DMS-9204276. Mathematics Department, University of Michigan, Ann Arbor, MI 48109-1003, USA. ablass@umich.edu

‡Partially supported by NSF grant CCR 92-04742 and ONR grant N00014-91-J-11861. EECS Department, University of Michigan, Ann Arbor, MI 48109-2122, USA. gurevich@umich.edu

The most versatile machine model we know is evolving algebras (ealgebras, or EAs) [G1,G2]. According to the EA thesis, any algorithm can be simulated by an appropriate ealgebra in lock-step. In this paper we transfer Jones's results mentioned above to evolving algebras. We restrict attention to sequential deterministic ealgebras (without variables); more general ealgebras will be considered elsewhere. The definition of ealgebras in the tutorial [G1] is sufficient for our purposes in this paper. For the reader's convenience, we include in Section 2 a brief review of ealgebras.

Our first result is a universal ealgebra U. (Another universal ealgebra U has been constructed by Anne Preller [P].) To avoid distraction from the main idea, the presentation of U in Section 3 is simplified in that we ignore external functions.

Our main result is the diagonalization theorem for ealgebras formulated in Section 4. It implies the linear time hierarchy theorem for ealgebras. Our proof of the diagonalization theorem cannot be given here because of the lack of space.

Finally, we consider quickly other machine models in Section 5. Schönhage machines are essentially unary ealgebras whose only external function is the nullary function New which brings from the Reserve new elements as needed. The diagonalization and linear hierarchy theorems for Schönhage machines easily follow from those for ealgebras. One has to work a little harder to derive the diagonalization and linear time hierarchy theorems for various sufficiently powerful RAM models (e.g. RAMs with addition and multiplication) from those for ealgebras.

2 Evolving Algebras

We recall the definitions of commands. An update instruction is an expression of the form

$f(t_1, \ldots, t_r) := t_0$

where each t_i is a ground term. Transition rules are defined recursively. Any update instruction is a rule. If k is a natural number, $g_0, \ldots, g_k$ are terms and $R_0, \ldots, R_k$ are sequences of rules then the expression

```
if g_0 then R_0
elseif g_1 then R_1
⋮                                                    (*)
elseif g_k then R_k
endif
```

is a (compound) rule. If the last guard g_k is *true* then the last elseif clause can be abbreviated to

```
else R_k
```

A program P is a sequence of rules. A state of a program P is a static algebra, i.e., a nonempty set (the *carrier*) equipped with interpretations of the function symbols that occur in P. A special distinguished element *undef* is used to deal with partial functions. Distinguished elements *true* and *false* are used to treat predicates as functions. A unary predicate may be regarded as defining a special universe within the carrier.

To execute an update instruction $f(t_1, \ldots, t_r) := t_0$ in a state A, evaluate all terms t_i is A and set f at $(A(t_1), \ldots, A(t_r))$ to $A(t_0)$. To execute a sequence $u_1, \ldots, u_k$ of update instructions at A, evaluate all the relevant terms in A and then make all the changes. What happens if some update instructions contradict each other at A? In this paper, we use eager execution to resolve conflicts. Make all the changes from left to right; later changes may override earlier ones. For example, if $k = 2$, u_1 is $f(a) := 3$, u_2 is $f(b) := 5$ and $A(a) = A(b) = 1$ then $f(1)$ is set first to 3 and then finally to 5. At a given state, the compound rule $(*)$ reduces to a sequence of update instructions obtained recursively as follows. Find the first i (if any) for which $A(g_i) = \mathit{true}$ and reduce all the rules in R_i to sequences of update instructions. If there is no such i then $(*)$ yields the empty sequence. To execute P at a state A, execute the concatenation of the update instruction sequences produced by the rules in P.

A pure run of the ealgebra with program P is a sequence of states in which each except the initial state is obtained by executing P in the previous state. "Pure" refers to the run's being uninfluenced by any external environment. If there are environmental influences, they change certain functions, called external functions. A typical example of an external function is the input provided by the user. Another example is the nullary function New mentioned above.

Time A relatively obvious EA program U simulates an arbitrary EA program P in lock-step where the lag factor c depends on P. Is it possible to make c independent of P? The answer is no as long as we identify time with the number of steps. The number of updates performed by U in c steps is bounded and thus U cannot compute the next state of P in c steps if P performs more than c updates. However this answer is misleading. Ealgebras were designed to make the specification job easier. Introducing more functions and rules, one may increase forever the amount of work that an ealgebra is able to do in one step. We need a more honest measure of time. The following definition will do.

Given a state A, we define inductively the time needed to compute and execute at A. The time to compute a term $f(t_1, \ldots, t_r)$ is

$$\mathrm{Time}(f(t_1, \ldots, t_r), A) = 1 + \sum_{i=1}^{r} \mathrm{Time}(t_i, A).$$

The time to execute an update instruction $f(t_1, \ldots, t_r) := t_0$ is

$$\mathrm{Time}((f(t_1, \ldots, t_r) := t_0), A) = 1 + \sum_{i=0}^{r} \mathrm{Time}(t_i, A).$$

Let C be a clause "(else)if g then $R_1, \ldots, R_k$". The time to execute C is

$$\mathrm{Time}(C, A) = \begin{cases} 1 + \mathrm{Time}(g, A) + \sum_{i=1}^{k} \mathrm{Time}(R_i, A), & \text{if } A(g) = \mathit{true}; \\ 1 + \mathrm{Time}(g, A), & \text{otherwise.} \end{cases}$$

The time to execute a compound rule R with clauses $C_0, \ldots, C_k$ is

$$\mathrm{Time}(R, A) = 1 + \sum_{i=0}^{j} \mathrm{Time}(C_i, A),$$

where j is the first i such that the guard g_i in C_i evaluates to *true* in A, and $j = k$ if there is no such i. Let P be a program with rules $R_1, \ldots, R_k$. The time to execute one step of P at state A is:

$$\mathrm{Time}(P, A) = 1 + \sum_{i=1}^{k} \mathrm{Time}(R_i, A).$$

The time of a run $A_0, \ldots, A_k$ of program P is $\sum_{i=0}^{k-1} \mathrm{Time}(P, A_i)$.

3 Universal Ealgebra

In the spirit of the EA approach, we would like to avoid coding as much as possible. However, some coding is necessary in order to represent an arbitrary ealgebra P together with a state A of P as a state $P + A$ of a universal program.

Parse Trees An *oriented tree* is a single-universe static algebra whose vocabulary consists of the nullary function root and the unary functions Parent, FirstChild, LastChild and NextSib. The interpretations are obvious; in particular, Parent(root)=*undef*. A *labeled tree* is obtained by adding a new universe of labels and a labeling function from tree nodes to labels. To *compose* labeled trees $T_1, \ldots, T_k$ under label w, take disjoint copies of $T_1, \ldots, T_k$ and make their roots children of a new root labeled w; denote the result $\mathcal{C}(T_1, \ldots, T_k; w)$.

By one induction, we define parse trees $\mathrm{PT}(s) = \mathrm{ParseTree}(s)$ for various syntactical objects s related to a program P. These parse trees are labeled trees, and the universe of labels consists of the function symbols in the vocabulary of P and the special labels "update", "clause", "compound-rule", and "program".

$$\mathrm{ParseTree}(f(t_1, \ldots, t_r)) = \mathcal{C}(\mathrm{PT}(t_1), \ldots, \mathrm{PT}(t_r); f).$$

If u is an update instruction $f(t_1, \ldots, t_r) := t_0$ then

$$\mathrm{ParseTree}(u) = \mathcal{C}(\mathrm{PT}(f(t_1, \ldots, t_r)), \mathrm{PT}(t_0); \mathrm{update}).$$

If C is a clause "(else)if g then $R_1, \ldots, R_m$" then

$$\mathrm{ParseTree}(C) = \mathcal{C}(\mathrm{PT}(g), \mathrm{PT}(R_1), \ldots, \mathrm{PT}(R_m); \mathrm{clause}).$$

If R is a compound rule with clauses $C_1, \ldots, C_k$ then

$$\mathrm{ParseTree}(R) = \mathcal{C}(\mathrm{PT}(C_1), \ldots, \mathrm{PT}(C_k); \mathrm{compound\text{-}rule}).$$

Finally, if the program P is a sequence of rules $R_1, \ldots, R_k$ then

$$\mathrm{ParseTree}(P) = \mathcal{C}(\mathrm{PT}(R_1), \ldots, \mathrm{PT}(R_k); \mathrm{program}).$$

The Apply Function A program P may have functions of higher arity than those of the universal program U. Thus the rules of U cannot deal directly with functions of P. We solve this problem by allowing the state $P + A$ to have not only elements of A but also sequences of those elements of length $\leq r$ where r is the highest arity of P's functions.

The vocabulary of our universal algebra contains the nullary function name Nil and the binary function names Append, Apply among other symbols. Nil and Append are interpreted in the obvious way; in particular, $\mathrm{Append}(s, x) \neq \mathit{undef}$ only if s is a sequence of length $< r$ and x is a single element. If f belongs to the universe of labels and is one of the function symbols and s is a sequence whose length equals the arity of f then $\mathrm{Apply}(f, s) = A(f(s))$ in a state $P + A$.

The Strategy for a Universal Program States $P + A$ sketched above are called *pose* states. When P makes one step and transforms A to the next state A^+, U performs a computation

$$B_0 = P + A, B_1, \ldots, B_k = P + A^+$$

going from a pose state B_0 to a pose state B_k via nonpose states $B_1, \ldots, B_{k-1}$. To make the distinction between pose and nonpose states sharp, we specify a special pose term in the vocabulary of U such that a state B of U is a pose state if and only if the pose term evaluates to *true* at B.

Theorem 3.1 *There exist an EA program U and a constant c such that U simulates any EA program P in lock-step with lag factor c.*

The computation of U splits into two phases: evaluate and update. During the evaluation phase the parse tree of P is traversed in the depth-first fashion except that certain parts are skipped because of the control structure of compound rules (specifically, elseif) and the values of guards. All visited terms are evaluated and relevant updates are marked. During the update phase, the marked updates are executed. The nullary function C (an allusion to "control") gives the current position in the parse tree of P.

Abbreviations and Aliases

F, L, N, Pa, S abbreviate
FirstChild(C), LastChild(C), NextSib(C), Parent(C), Status.
"C is the last sib" stands for NextSib(C)=*undef*.
"C is a term" stands for Vocabulary(Label(C)). The same for Pa.
"C is a clause" stands for Label(C)=clause. The same for Pa.
"C is a guard" abbreviates "C is a term and Pa is a clause"
FU, NU(C), LU abbreviate
FirstActiveUpdate, NextActiveUpdate(C), LastActiveUpdate.
Val(C) abbreviates Apply(Label(C),Arg(C))

Initial State It is supposed that the initial state of U satisfies the following conditions:

C=root and therefore Label(C)=program; halt=*false*; phase=evaluate; S=start.
Arg(x)=ValSeq(x)=Nil for all x such that Term(Label(x)) holds.
FU, LU are undefined. NU(x) is nowhere defined.

The pose term is "Label(C)=program and S=start". In particular, the initial state is a pose state, and we assume that it satisfies an earlier description of pose states.

Rules Advance Rule

```
if phase=evaluate and ¬(halt) then
   if S=start then
      if F=undef then S:=finish else C:=F endif
   elseif C is the last sib or
         C is a guard and Val(C)≠true or
         C is a clause and Val(F)=true then
      C:=Pa
   else C:=N, S:=start endif
endif
```

Advance Rule regulates the movement of C during the evaluation phase. The nullary function Status with values *start* and *finish* indicates whether whether the evaluation of the subtree rooted at C starts or finishes.

Root Rule

```
if Label(C)=program and ¬(halt) and phase=evaluate and S=finish then
   if FU=undef then halt:=true else phase:=update, C:=FU, FU:=undef endif
endif
```

Root Rule applies at the end of the evaluation phase. If FU=undef then there is and there will be nothing to update; the universal program halts. Otherwise, C jumps to FU and the update phase starts.

Term Rule

```
if C is a term and S=finish and Pa is a term then
   if C≠LastChild(Pa) then ValSeq(N):=Append(ValSeq(C),Val(C))
   else Arg(Pa):=Append(ValSeq(C),Val(C)) endif
endif
```

Term Rule applies at the end of the evaluation of a compononent t_i of a term $f(t_1, \ldots, t_r)$. At this moment ValSeq(C) is the value of $(t_1, \ldots, t_{i-1})$. The value of $(t_1, \ldots, t_i)$ is computed. If $i < r$, that value is stored as ValSeq(N) at the next sibling N of C; otherwise it is stored at the parent node Pa as Arg(Pa).

Update Rule

```
if Label(C)=update then
   if phase=evaluate and Val(L)≠Apply(Label(F),Arg(F)) then
      if FU=undef then FU:=C, LU:=C else NU(LU):=C, LU:=C endif
```

```
   else
      Apply(Label(F),Arg(F)):=Val(L)
      if Defined(NU(C)) then C:=NU(C), NU(C):=undef
      else C:=root, phase:=evaluate, S:=start, LU:=undef endif
   endif
endif
```

Update Rule applies when C is at the root of an update parse tree. During the evaluation phase, relevant updates are marked for future execution. The very first relevant update is marked with FU. The function NU is set to lead from one relevant update to the next one. LU marks the last relevant update. During the update phase, the marked updates are executed starting with FU and finishing with LU. As each update is executed, it becomes unmarked.

4 Diagonalization

Theorem 4.1 (Diagonalization) *Let σ be a vocabulary with sufficiently many nullary and unary function symbols. There exist a σ-program D and a constant c_σ such that, for any σ-program P, the run of D on input ParseTree(P) simulates in lock-step with lag factor c_σ the run of P on ParseTree(P) except that the output is changed.*

Theorem 4.2 (Linear Time Hierarchy) *For any constant a, there is a function computable by a σ-program in time $c_\sigma \cdot a \cdot n$ but not computable by any σ-program in time $a \cdot n$.*

The diagonalization is harder than constructing a universal machine U. U can use functions like Append and Apply unknown to P but D has no such luck. P may play with any functions available to D. Our D does not use sequences or the Apply function. Notice also that D cannot start by copying the whole input as the universal Turing machine does, as such copying might take much more time than P uses. If we were interested only in linear-time simulation, we could restrict attention to programs P that read their whole input. Then D would have enough time to copy the input before proceeding with the simulation. But in order to obtain lock-step simulation we must program D to copy pieces of its input when they are needed rather than all at once. Because of this, the pose states of D are not uniquely determined by the corresponding states of P. They do, however, uniquely determine the corresponding states of P. The notion of simulation of runs, used in Theorem 4.1, should be defined accordingly.

5 Other Machine Models

Schönhage Machines Schönhage machines [S] can be lock-step simulated by unary ealgebras (that is, ealgebras without functions of arity ≥ 2) whose only external function is the nullary function New which brings from the Reserve new elements as needed. To get a uniform lock-step simulation, one needs honest time counting for both Schönhage machines and ealgebras. Furthermore, every such unary ealgebra can be simulated in lock step by an appropriate Schönhage machine. Thus, the diagonalization and linear time hierarchy theorems for Schönhage machines [J] are consequences of Theorems 4.1 and 4.2.

Random Access Machines Consider RAMs that have a particular set of operations available for manipulating the contents and addresses of their registers and that have each register's content bounded in length by $O(\log n)$ where n is the input size. Such RAMs can be uniformly simulated in lock-step by an ealgebra that has those operations built into its initial state. Diagonalization and linear hierarchy theorems for sufficiently powerful RAMs can be deduced from Theorems 4.1 and 4.2 by designing RAMs to simulate the relevant ealgebras in lock-step.

References

G1 Yuri Gurevich, "Evolving Algebras: An Attempt to Discover Semantics", Bull. EATCS, 43 (1991), 264–284, and (slightly revised) in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292.

G2 Yuri Gurevich, "Evolving Algebras 1993: Lipari Guide", in "Specification and Validation Methods", Ed. E. Börger, Oxford University Press, to appear.

J Neil D. Jones, "Constant Time Factors *Do* Matter", ACM Symp. on Theory of Computing, 1993, 602–611

P Anne Preller, Private Communication, February 1994.

R Michael Rabin, "Real Time Computations", Israel J. of Math., 1:4 (1963), 203–211.

S Arnold Schönhage, "Storage modification machines", SIAM J. on Computing, 9 (1980), 490-508.