EVOLVING ALGEBRAS: AN ATTEMPT TO DISCOVER SEMANTICS*

YURI GUREVICH
Electrical Engineering and Computer Science Department
The University of Michigan
Ann Arbor, MI 48109-2122, USA

Preface

This tutorial is based on lecture notes from the Fall 1990 course on Principles of Programming Languages at the University of Michigan. (My young friend Quisani did not attend the lectures.) The present version incorporates some changes provoked by the necessity to update the bibliography. The main part of the paper is still the same, however, and the examples are unchanged even though many things happened in the meantime. In particular, we (the collective we) have learned how to build evolving algebras by the method of successive refinements, and the current evolving algebra description of the C programming language in [GH] doesn't look much like the `strcpy` example anymore. Now, we understand better how to compose evolving algebras and how to prove things with evolving algebras. A typical misconception is that the operational approach is necessarily too detailed. Some people think that an approach suited for complexity analysis does not give a good high-level specification language. I believe in a high-level specification language based on evolving algebras; the successive refinement method is then one tool to prove implementation correctness. But this and various other issues (how to incorporate real time into evolving algebras for example) will have to be addressed elsewhere.

*First published in Bulletin of European Assoc. for Theor. Computer Science, no. 43, Feb. 1991, 264–284. Reprinted in "Current Trends in Theoretical Computer Science", Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, 266–292. This version contains slight modifications to reflect the one and only change in the definition of ealgebras that happened in the meantime, namely that the basic EA machine is deterministic; the explanation for the change is found in Subsection 2.7 in [Gu4].

1. Another computation model

Quisani: Somebody told me that you are doing semantics these days.

Author: Somebody was right.

Q: Sure, somebody is usually right. Tell me about your semantics.

A: The original idea was to provide operational semantics for algorithms by elaborating upon what may be called the implicit Turing's thesis: every algorithm is simulated by an appropriate Turing machine [Gu1]. Turing did not claim this explicitly; his thesis was: every computable function is computable by some Turing machine. But his informal proof of the thesis [Tu] gives the stronger version. In the sense of the stronger thesis, Turing machines give operational semantics to algorithms. Unfortunately, this semantics is not very good (and of course I do not claim that semantics was Turing's goal). Turing machine simulation may be very clumsy. In particular, one step of the algorithm may require a long sequence of steps of the simulating Turing machine. I was looking for machines able to simulate algorithms much more closely. In particular, the simulation should be *lock-step* so that the simulating machine makes only a bounded number of steps to simulate one step of the given algorithm. Evolving algebras, or EAs, are supposed to be such machines.

Q: There are abstract machines in the literature better suited to simulate algorithms than Turing machines: Kolmogorov-Uspensky machines [KU], storage modification machines of Schönhage [Sch], various random access machines.

A: Each of these machine models has a fixed level of abstraction which may be very low for some algorithms. An evolving algebra, on the other hand, may be tailored to an arbitrary abstraction level. One may have a whole hierarchy of evolving algebras of various abstraction levels for the same algorithm. See [GH] for an example where you will find EAs for various abstraction levels of the C programming language. A hierarchy of evolving algebras is constructed by Egon Börger and Dean Rosenzweig in [BRo1] where the technique of successive refinements is used to reconstruct the Warren Abstract Machine (a virtual machine model which underlies most of the current Prolog implementations and incorporates crucial optimization techniques) starting from a more abstract EA for Prolog developed by Börger in [Bo1–Bo3].

Q: How do you tailor an EA machine to the abstraction level of an algorithm whose individual steps are complicated algorithms all by themselves? For example, the algorithm may be written in a high level language that allows, say, multiplying integer matrices in one step.

A: You model the given algorithm modulo those algorithms needed to perform single steps. In your case, matrix multiplication will be built in as an operation.

Q: Coming back to Turing, there could be a good reason for him to speak about computable functions rather than algorithms. We don't really know what algorithms are.

A: I agree. Notice, however, that there are different notions of algorithm. On the one hand, an algorithm is an intuitive idea which you have in your head before writing code. The code then implements the algorithm. The same algorithm may be coded in different programming languages. It may have many different abstraction levels. In particular, a source program and the result of its compilation implement versions of the same algorithm that differ in their abstraction levels. We can argue which, if any, of the abstraction levels is the natural one. The question when two such intuitive algorithms are the same may be hard.

On the other hand, there is a more naive notion according to which an algorithm essentially is a program (together with some computing environment which is often implicit). In particular, different programs give different algorithms. I do not want to completely equate programs and algorithms though; one speaks usually about programs in particular programming languages. The notion of algorithm is a little more general. A programming language itself can be seen as an algorithm – a universal algorithm that takes a given program as a part of its data.

Q: Do you want to capture the naive notion of algorithm by means of evolving algebras?

A: The goal is good operational semantics, but it would be great to formalize properly the notion of algorithms, wouldn't it?

Q: To what extent is this formalization goal achieved?

A: Well, I will explain to you the notion of sequential evolving algebras. It seems to me that it captures the notion of sequential algorithms, that every sequential algorithm can be closely simulated by an appropriate sequential EA.

Q: What do you mean "closely simulated"?

A: The simulating EA is on the same abstraction level and does not use much more resources than the given algorithm. In particular the simulation is lock-step.

Q: But who knows what kind of resources the given algorithm uses?

A: Well, give me a sequential algorithm and tell me which resources you care about. Then we will be able, I think, to construct a sequential EA simulating your algorithm closely with respect to the resources in question.

Q: What are your arguments?

A: Speculation and experimentation. In particular, EAs were constructed for a number of sequential programming languages, e.g., Modula-2 [GMr], Prolog [Bo1–Bo3], Prolog III [BS], Protos-L [BB] and C [GH]. In this modeling of programming languages, the goal was direct and natural operational semantics. But these models confirm the thesis as well.

Q: What about nonsequential, say distributed parallel, algorithms?

A: The notion of sequential EA has been generalized to the distributed parallel case [GMs]. In particular, EAs were constructed for Occam [GMs], Parlog [BRi1] and Concurrent Prolog [BRi2]. See also [GR]. I do not know any distributed parallel algorithm that presents a conceptual challenge for EA formalization. There is, however, a difference between the sequential and distributed parallel cases. An arbitrary sequential algorithm can be viewed as a sequential transition system; analyzing such systems, you discover sequential evolving algebras and may justify to an extent the thesis. The notion of distributed parallel algorithms seems open-ended at the moment.

Q: Even if one buys your thesis, he/she may not like your semantics.

A: I agree but hope that he/she will like it. EAs are relatively easy to understand and design. I use them in class. Even a very small program in an unusual language can be difficult to understand directly. Sketching an appropriate EA on a blackboard may help. You can define various machine models as special classes of evolving algebras.

Q: Do you have software to run an EA machine?

A: Yes. For this purpose, we use an EA interpreter written in C here at Michigan. You can run your EA for a specified number of steps and then examine the resulting state, you can run your EA until some predicate becomes true, and so on.

Q: Do you see any use for evolving algebras outside the classroom?

A: Yes. Here are some examples. ISO, the International Standards Organization, adapted Egon Börger's proposal [BD] of an EA based precise semantics of full Prolog [WG17]. (Actually, Egon Börger is not completely happy with the adaptation. In cooperation with Dean Rosenzweig, he prepared a simpler evolving algebra description of full Prolog [BRo2].) Georg Gottlob, Gerti Kappel, and Michael Schrefl used EAs to specify the semantics of characteristic features of object-oriented database models [GKS, KS]. An EA based manual for a programming language could be precise and relatively easy to read.

Q: Do you expect evolving algebras to be used for proving things about algorithms?

A: Yes. The possibility to tailor EAs to any given abstraction level is especially useful. I have already mentioned Börger-Rosenzweig's work on the Warren Abstract Machine [BRo1]. Starting from a Prolog evolving algebra on a high abstraction level (essentially the level of SLD-resolution), a hierarchy of more and more refined evolving algebras is constructed; the final algebra is the first formal abstract specification of WAM in the literature. It is proved that, under appropriate assumptions, every $(i+1)$-st algebra correctly implements the i-th algebra. This is a solid foundation for constructing provably correct compilers from Prolog to WAM; the mentioned proof assumptions give rise to conditions that guarantee correctness. Using similar techniques, my student Jim Huggins is attempting to prove the correctness of the Kermit file transfer protocol.

The EA approach is appropriate for the use of temporal and dynamic logics though we have only started to explore this. In a sense, the EA approach provides a foundation for the use of such logics. If you are looking for models of first-order temporal and dynamic logics, think about evolving algebras of appropriate kinds. (By the way, evolving algebras are called dynamic algebras sometimes, but the latter term is used in the dynamic logic area in a very different sense; see [Pr] for example.)

Q: Now tell me what evolving algebras are.

A: The basic idea is very simple, at least in the sequential case, when time is sequential (the algorithm starts in some initial state S_0 and goes through states S_1, S_2, etc.) and only a bounded amount of work is done on each step. Each state can be represented by a first-order structure: a set with relations and functions. (The term "first-order structure" is misleading. It is logic that is first-order, not structures. Models of second-order logic, for example, can be easily and naturally represented by "first-order" structures. But the term is common and we will use it.) Thus, the run can be seen as a succession of first-order structures, but this isn't a very fruitful way to see the process. How do we get from a state S_i to the next state S_{i+1}? Following the algorithm, we perform a bounded number of changes in S_i. It turns out that the whole algorithm can be rewritten as a finite number of transition rules of very simple form.

By the way, I am thinking about states of the algorithm as something feasible. Certainly, any computer executing the algorithm is able (all the time it is executing correctly) to represent the states. On the other hand, the whole process may be huge, unwieldy and infeasible to represent.

It isn't obvious how to generalize the basic idea to the case of asynchronous distributed algorithms; see [GMs] in this connection.

Q: It seems that you have an evolving first-order structure. Why do you call it an evolving algebra?

A: For technical reasons, it is convenient to replace relations by appropriate functions. In universal algebra, a first-order structure without relations is called an algebra [Gr]; we will use the term "static algebra".

2. Static algebras

Q: Please explain to me what static algebras are exactly.

A: Gladly. Let me start at the very beginning. The term *signature* will be used to mean a finite collection of function names. It is supposed that each function name comes with an indication of its arity, i.e., the number of arguments. A *static algebra* of a signature σ is a nonempty set $\mathcal{S}$, called the *superuniverse*, together with interpretations on $\mathcal{S}$ of function names in σ. A function name of arity r is interpreted as an r-ary function from $\mathcal{S}$ to $\mathcal{S}$, i.e., a function from $\mathcal{S}^r$ to $\mathcal{S}$, and is called a *basic function* of the algebra. A basic function of arity zero is called a *distinguished element*.

Q: I thought a function is always a function of something so that the arity of a function is at least one.

A: Stretching notions is not new to science; recall the notion of zero-speed motion in physics for example. One may insist that zero-speed motion is no motion at all, but it is more convenient to view zero-speed motion as a special case of motion. By the way, in standard terminology, the term "universe", rather than "superuniverse", is used. We reserve the term "universe" for other purposes.

Here is an example of a static algebra (which I will have to modify). Suppose that a signature σ contains zero-ary function names 0, 1 and binary function names $+$ and $\times$. One static algebra of signature σ is obtained as follows: Take the superuniverse to be the set of integers and interpret the function names in the obvious way. There are other natural static algebras of signature σ, but in general a static algebra need not be natural; it may be very arbitrary.

Q: I understand that static algebras will represent snapshots of computational processes. Isn't your definition too restrictive? For example, you may want to extend that arithmetical static algebra by the division operation but the division operation is partial.

A: Instead of generalizing the notion of a static algebra by permitting partial functions, we will restrict attention to algebras with a distinguished element *undef*. In particular, that arithmetical algebra above should be modified by adding a new element *undef*. It can be further extended by adding the division function; we will have $a \div b = \mathit{undef}$ unless both a and b are integers and b divides a. Formally, every r-ary basic function f is defined on every r-tuple $\bar{a}$ of elements of the superuniverse, but we will say that f is undefined at $\bar{a}$ if $f(\bar{a}) = \mathit{undef}$; the set of tuples $\bar{a}$ with $f(\bar{a}) \neq \mathit{undef}$ will be called the *domain* of f.

Q: Still, your static algebras seem too restrictive to me. How do you deal with different data types?

A: One possibility would be to generalize the notion of a static algebra and to consider many-sorted algebras; such algebras are widely used. But this is not necessary. We will suppose that every static algebra contains distinguished elements *true* and *false*. A basic function U, defined on the whole superuniverse, with values in $\{\mathit{true}, \mathit{false}\}$ will be viewed as the set of elements a such that $U(a) = \mathit{true}$ and called a *universe*. We will say that a belongs to U if $U(a) = \mathit{true}$.

Further, we will suppose that every static algebra has the equality relation, a universe *Bool*, comprising two elements *true* and *false*, and the usual boolean operations; the result of any boolean operation is *undef* if at least one of the arguments is outside *Bool*. Thus the arithmetical algebra should be modified again; a universe *Integer* comprising the integers can be declared as well. A relation R on a particular universe U will be represented by (and identified with) the characteristic function of R which is undefined (i.e. equal to *undef*) if at least one of the arguments is outside of U. Now we may further augment the twice modified arithmetic algebra by the standard ordering of integers.

Q: You can dispense with one of the two boolean values as it is done in Lisp. For example, *true* may be dropped and then any element different from *false*, with the possible exception of *undef*, will represent boolean truth.

A: Sure, but I prefer to distinguish between boolean truth and, say, numbers.

Q: Why do you need the trick of coding relations as functions? Why can't a signature contain relation names?

A: The reason is to make updates, introduced below, applicable to relations as well. By the way, ever-present basic functions with predetermined domains and values are called *logical constants*. Logical constants may be omitted when a static algebra is described. For example, the thrice modified arithmetical algebra can be described as follows. It has (1) a universe *Integer* comprising the integer numbers, (2) distinguished integers 0 and 1, (3) the usual total binary operations $+$ and $\times$, the usual partial operation $\div$ and the usual ordering $<$ on *Integer*.

Q: Did we finish with logical constants?

A: Almost. Sometimes it is necessary to suppose that there is auxiliary infinite (or sufficiently large finite) universe *Reserve* disjoint from other universes and a special function that selects an element of *Reserve*. The role of *Reserve* will be clear soon.

Q: I am confused. I thought that we will be dealing with finite feasible static algebras reflecting snapshots of real computer computations. If this is the case, then there is no room for infinite universes.

A: In principle it is possible to keep everything finite and feasible. For example, the *Reserve* can reflect real computational resources. However, it is much more practical to divide concerns. You permit whatever universes and basic functions are convenient and, if necessary, you keep track of various resources.

Finally, let me recall the definition of *closed terms* of a given signature (the first clause is superfluous but it seems to make comprehension easier):

- Every zero-ary function name is a closed term.
- If f is a function name of arity r and $t_1, \ldots, t_r$ are closed terms then $f(t_1, \ldots, t_r)$ is a closed term.

We will use the usual abbreviations and conventions in order to increase readability. Here are some terms in the arithmetical signature mentioned above:
$0, 1+1, (1+1) \times (1+1)$.

3. Transition rules

Q: How do you program the evolution of an evolving algebra?

A: The *basic language* of transition rules is very modest and even minimal in some justifiable sense. Think about static algebras as states of evolving algebras. What is a simplest atomic change of a static algebra?

Q: To change one function at one place, I guess. There are, however, other atomic changes: Adding an element to the superuniverse, removing an element from the superuniverse, adding a new basic function, removing a new basic function.

A: Adding elements to the superuniverse can be avoided with the help of *Reserve*. To extend a universe U, place a *Reserve* element a in U; technically, set U to *true* and Reserve to *false* at a. To remove an element a from U, set $U(a) = \mathit{false}$.

Q: I see. And what about adding and removing basic functions.

A: I do not believe that this is ever necessary if your EA properly formalizes the given algorithm.

Q: Suppose the algorithm is given by a program which declares new functions from time to time. This isn't unusual, right?

A: Right. But the new functions can be treated as data. For example, you may have a whole universe F of, say, unary operations on some universe U. To deal with this situation, you may have a binary basic function *Apply*. If $f \in F$ and $u \in U$ then $\mathit{Apply}(f, u)$ will be the result of applying f to u.

Q: What if my program declares functions of *a priori* unbounded arities? Better yet, it may declare a function that takes an arbitrary number of arguments.

A: Use the usual tricks. A function with arbitrarily many arguments can be viewed as a unary function of lists, for example.

Q: Thus, your basic language allows no extension or contraction of either the signature or the superuniverse.

A: That is right. Commands of the basic language will be called *transition rules* or simply *rules*. The only primitive transition rule is a *local function update* of the form

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

where f is (the name of) a basic function, r is the arity of f and every t_i is a closed term. Is the meaning clear?

Q: I think so. You evaluate all terms t_i. If a_i is the value of t_i then you reset $f(a_1, \ldots, a_r)$ to a_0.

A: That is right. For brevity, let me call local function updates simply updates. A slightly more complicated rule is a *guarded update*

if b ***then*** u ***endif***

where b (the *guard*) is any term and u is any update. If b evaluates to *true* on the given static algebra then perform u; otherwise do nothing. Let me stress that all evaluations are done in the given static algebra.

Q: I think you can get rid of guarded updates. Introduce a logical constant, say, *Cond(x,y,z)* interpreted in any static algebra as follows: If x is *true* then the result is y; otherwise the result is z. Then a guarded update above has exactly the same effect as the unguarded update

$f(t_1,\ldots,t_r) := \mathit{Cond}(b, t_0, f(t_1,\ldots,t_r))$

A: You are right. With *Cond*, a guarded update is equivalent to (i.e. has exactly the same effect on any static algebra as) some unguarded update. Unfortunately, the use of *Cond* makes rules more difficult to read.

Q: By the way, *Cond* corresponds naturally to a command like this:

if b ***then*** u_1 ***else*** u_2 ***endif***

A: This command, which is a legal rule as well, is equivalent to a set of two regular conditional updates:

if b ***then*** u_1 ***endif***

if $b \neq true$ ***then*** u_2 ***endif***

A set of guarded updates, written usually as a list, is executed in parallel, so that the order is immaterial. All guarded updates on the list are performed simultaneously. For example, the rule

$a := f(a)$
$b := f(a)$

sets a and b to the same value. Imagine a demon who evaluates all relevant terms in a given state and then makes the necessary changes; the result is a new state.

Q: But different guarded updates may contradict each other. For example, you may have

$a := true$
$b := false$

where a and b evaluate to the same value. Better yet, you may have

$a := true$
$a := false$

What is the meaning then?

A: The computation halts. The basic EA machine is deterministic. There is an extension of the basic model with explicit nondeterminism; see Subsection 4.1 in [Gu4].

Q: Why did you choose to interpret a list of rules as a set rather than a sequence? The sequence approach is more usual, it is clear, it avoids nicely the problems of contradicting updates and lends itself more naturally to the use of macros.

A: I admit that the sequence interpretation may work better in many situations though the use of guards allows one to ensure the desired sequence of actions in the set approach as well and the set interpretation has its own advantages. First, it permits a certain concurrency; this is convenient and helps to achieve a better simulation of the given algorithm. Second the set interpretation allows a natural transition to asynchronous distributed computing, but let us stick to sequential computing for the time being.

Q: Is it difficult to define basic transition rules in full generality?

A: No. Here is the definition.

- Any local function update is a rule.
- If k is a natural number, $b_0, \dots, b_k$ are terms and $C_0, \dots, C_{k+1}$ are sets of rules then

  ```
  if $b_0$ then $C_0$
  elseif $b_1$ then $C_1$
        ⋮
  elseif $b_k$ then $C_k$
  else $C_{k+1}$
  endif
  ```

  as well as

  ```
  if $b_0$ then $C_0$
  elseif $b_1$ then $C_1$
        ⋮
  elseif $b_k$ then $C_k$
  endif
  ```

  is a rule.

I hope that the meaning is obvious. Every rule is equivalent to a set of guarded updates; this is easy to check by induction. Therefore every set of rules is equivalent to a set of guarded updates.

Q: And what is a program?

A: A program is a set of rules. Recall that we deal with sequential algorithms; only a bounded amount of work is done at each step.

Q: Is the definition of sequential EAs complete?

A: No, not yet. Let us see an example.

4. Example 1: A stack machine

A: Consider a stack machine for computing expressions given in reverse Polish notation, or RPN. An expression $(1 + 23) \times (45 + 6)$ written in RPN would appear as

$1\ 23 + 45\ 6 + \times$

Q: Are you talking about arithmetical operations over integers?

A: Not necessarily. Suppose we have a nonempty set *Data* and a nonempty set *Oper* of (for simplicity) total binary operations on *Data*. For example, *Data* may be the set of integers and *Oper* may be $\{+, \times\}$. We suppose that the RPN expression is given to us in the form of a list where each entry denotes a datum or an operation. The stack machine reads one entry of the list at a time from the input file. If the entry denotes a datum, it is pushed onto the stack. If the entry denotes an operation, the machine pops two items from the stack, applies the operation and pushes (the notation for) the result onto the stack. At the beginning, the stack is empty. In the case of the RPN expression above (with the usual operations), the stack goes through the following states: (), (1), (23 1), (24), (45 24), (6 45 24), (51 24), (1224).

The desired evolving algebra has *Data* and *Oper* as universes. *Arg1* and *Arg2* are distinguished elements of *Data*. To handle operations in *Oper*, the EA has a ternary function *Apply* such that *Apply(f,x,y) = f(x,y)* for all f in *Oper* and all x, y in *Data*. To handle the input, the EA has a universe *List* of all lists composed of data and operations. The basic functions *Head* and *Tail* have the usual meaning. If L is a list then *Head(L)* is the first element of L and *Tail(L)* is the remaining list. *EmptyList* has the obvious meaning. F is a distinguished list. Initially, F is the input. Finally, the evolving algebra has a universe *Stack* of all stacks of data with the usual operations *Push* (from *Data* $\times$ *Stack* to *Stack*), *Pop* (from *Stack* to *Stack*) and *Top* (from *Stack* to *Data*). S is a distinguished stack. Initially, S is empty.

Q: Since the stack machine deals with only one list and has only one stack, I do not understand why you need the universes *List* and *Stack*.

A: The EA should be on the abstraction level of the given algorithm. Operations like *Push* and *Pop* should be provided with their natural environment. We can manage without the universes *List* and *Stack* by implementing the input and the stack as arrays, but this would be a lower level of abstraction. Here are the transition rules.

```
if Head(F) is a datum then
    S := Push(Head(F), S)
    F := Tail(F)
endif
```

```
if Head(F) is an operation then
    if Arg1 = undef then
        Arg1 := Top(S)
        S := Pop(S)
    elseif Arg2 = undef then
        Arg2 := Top(S)
        S := Pop(S)
    else S := Push(Apply(Head(F), Arg1, Arg2), S)
        F := Tail(F)
        Arg1 := undef
        Arg2 := undef
    endif
endif
```

The phrase "*Head(F)* is a datum" means of course that *Head(F)* belongs to *Data*, i.e., *Data(Head(F)) = true*. The meaning of the phrase "*Head(F)* is an operation" is similar.

Q: You suppose that the input file contains a legal RPN expression.

A: That is true. When a given algorithm has an error handling routine, it can be formalized as well.

Q: Assume that there is a bound *Max* on the depth of the stack. What changes should be made?

A: Let me suppose for simplicity that the machine simply freezes if it attempts to push a datum onto a stack of the maximal depth. Then you may want to introduce a universe of natural numbers with successor operation $n+1$ and predecessor operation $n-1$ and distinguished natural numbers *Depth* and *Max*. Initially, *Depth* is zero. Augment the guard of the first of the two rules above by the conjunct $Depth < Max$ and add updates *Depth* := *Depth* ± 1 in appropriate places.

5. Static, dynamic and external functions

Q: You speak about basic functions like a programmer; a function has a name and may change. In mathematics, a function is a particular mapping. If you set sin(1) to 1/2 then the result isn't the sin function anymore.

A: You are right. Basic functions are the current interpretations of the corresponding function names. In every state of an EA, basic functions are particular mappings, but the same function name may be interpreted differently in different states. Actually, it is convenient to distinguish between *static* and *dynamic* basic functions. The distinction is syntactical: Dynamic functions appear in function updates as subjects for updating. Static functions do not change during the evolution of the algebra whereas dynamic functions may change. In the stack machine example, for instance, the only dynamic functions are S, F, $Arg1$ and $Arg2$.

Q: I see a problem with evolving algebras. They are completely isolated. How do you deal with input, output, interrupts and other interactions of the given algorithm with the outside world? Do you want to pretend that all reactions of the outside world are written in files ahead of time?

A: Let us examine the problem. In an attempt to formalize the given algorithm as an evolving algebra, we may discover that some basic functions depend on the outside world; let f be one of them. In the case that the algorithm is given by a program, f may reflect, for example, whatever the user types at the keyboard or some activity of the operating system. I presume that f is not a subject of updating by our transition rules; in other words, f is syntactically static. Nevertheless f may have different values in different states. One way to deal with this situation is to add another argument to f which will allow us to distinguish between different evaluations. For example, we may pretend, as you said, that f reads from a file and use the position in the file as a new argument. The pretense creates the illusion that the program of an evolving algebra is the only means to change its state. This illusion may be awkward to maintain and we use the following more radical way to deal with the problem.

The basic functions of an evolving algebra $\mathcal{A}$ are partitioned into *internal static* functions (in short, static functions), *internal dynamic functions* (in short, dynamic functions), and *external* functions. External functions cannot be changed by rules of $\mathcal{A}$, but they may be different in different states of $\mathcal{A}$. From the point of view of $\mathcal{A}$, an external function is an oracle, an unpredictable black box that is used but not controlled by $\mathcal{A}$.

Q: Let me see if I understand you. I imagine again the demon responsible for the evolution of $\mathcal{A}$, the one who executes the rules. In the case of an internal function f, the demon knows exactly how to compute f. For example, he may have a fixed algorithm for computing the default version of f and a complete table to account for the deviation from the default. In the case of an external function f, the demon has an unpredictable magic box for evaluating f. Whenever he needs to evaluate f, he enters the appropriate number of arguments and miraculously gets the result.

A: That is right. From the point of view of the demon, an external function is a nondeterministic function.

Q: I guess one can have rules like

$Output := t$

too, right?

A: Sure. Nothing dramatic happens in the given algebra when this rule is fired; syntactically your *Output* is just another distinguished element. But of course such rules are extremely important for communication. You may have several communication channels (and distinguished elements like *Output-on-channel-5*) and even a whole universe of channels (and a basic output function with an argument that is a channel). You may have a net of evolving algebras, but for the time being let us concentrate on a single evolving algebra (possibly communicating with the outside world).

Q: I have a question related to the types of basic functions. Formally, all r-ary basic functions are of the same type: In each state, they map the r-th power of the superuniverse to the superuniverse. It is clear, however, that you really think in terms of multiple universes and try to specify the type of basic functions in terms of universes.

A: This is very true. Essentially, we deal with many-sorted algebras though it is convenient, technically speaking, to have the superuniverse around.

Q: The case of a static function seems clear; here typing, i.e. prescribing a type, is a part of the initial state description. But what is the meaning of typing a dynamic function? Is it a declaration of intent?

A: It is an *integrity constraint*. Integrity constraints are statements (in indicative rather than imperative mood) that should be satisfied in any state during the evolution of a given EA. Often integrity constraints are implicit, but they can be stated explicitly. In the stack machine example, for instance, we have the following integrity constraints: *Arg1* and *Arg2* are data (i.e. belong to *Data*), F is a list, and S is a stack. It is easy to see that these four statements indeed hold in every state of the EA.

Q: Are there other kinds of integrity constraints?

A: Yes, usually we expect external functions to have values of certain types.

Q: What happens if an integrity constraint is violated?

A: The ideal machine breaks down.

Q: Do you suppose that integrity constraints constitute a part of the description of the EA?

A: This is not necessary. The effect of integrity constraints in question can be achieved by additional rules. In practice, however, an integrity constraint may be a convenient way to avoid writing boring transition rules. For example, stating a constraint that an external function f produces values of certain type allows one to avoid writing rules for what to do if f produces a value of a wrong type.

6. Example 2: A Turing machine

A: For the second example, I would like to describe a generic Turing machine as an evolving algebra.

Three universes of the EA, called *Control, Char* and *Displacement*, are nonempty finite sets. *InitialState* and *CurrentState* are distinguished elements of *Control*, *Blank* is a distinguished element of *Char*. *CurrentState* is a dynamic distinguished element; initially, *CurrentState* = *InitialState*. Another universe, *Tape*, is a countable infinite set; call its elements *cells*. *Head* is a dynamic distinguished cell. *Move* is a possibly partial function from *Tape* $\times$ *Displacement* to *Tape*.

Q: What kind of Turing machines are you talking about? Is the tape linear? Is it one-way infinite or two-way infinite?

A: The kind of Turing machine is determined by further restrictions on *Displacement* and *Move*. You can choose *Displacement* $= \{+1, -1\}$, and *Move* to be such that *Tape* with unary operations *Move(c,*$+1$*)*, *Move(c,*-1*)* is isomorphic to natural numbers with the successor and (partial) predecessor operations. This will give you a model with a linear one-way infinite tape. You can choose *Displacement* and *Move* in a way that gives rise to two linear tapes or to one two-dimensional tape, and so on.

We need some additional basic functions though. A dynamic function *TapeCont* maps cells to characters, and functions *NewState, NewChar* and *Shift* map the cartesian product *Control* $\times$ *Char* to *Control, Char* and *Displacement* respectively. It is required that *TapeCont(c)* = *Blank* for all but finitely many cells c. The transition rules are as follows.

CurrentState := NewState(CurrentState, TapeCont(Head))

TapeCont(Head) := NewChar(CurrentState, TapeCont(Head))

Head := Move(Head, Shift(CurrentState, TapeCont(Head))

Recall that all rules fire every time in a simultaneous fashion.

Q: By the way, the usual definition of Turing machine is shorter.

A: What usual definition?

Q: A Turing machine is a quadruple, namely, a set of control states, a set of characters, a transition function and the initial state.

A: In isolation, this short definition does not really define Turing machines. For example, one cannot derive from that definition that a Turing machine has a tape. Recall that after giving the short definition, one proceeds to define configurations, etc. These additional definitions are absorbed by the general EA framework. The short definition provides, however, a convenient notation for Turing machines.

Q: You did not define what your Turing machines compute.

A: I have to repeat the usual definitions.

Q: As an exercise, let me list the integrity constraints: *CurrentState* belongs to *Control*, *Head* belongs to *Tape*, and *TapeCont* maps *Tape* to *Char*.

A: That is correct. Again all constraints are easy to check in advance and again they can be stated in the form $t = true$ where t is a closed term.

Q: But the constraint on *TapeCont* is a universal statement: For every cell, the value of *TapeCont* is a character.

A: Taking for granted that the constraint is satisfied initially, it reduces to the constraint *Char(TapeCont(Head))* = *true*.

Q: I guess, it isn't always so easy to check whether a constraint $t = true$ will be eventually violated.

A: This is an undecidable problem. To prove this, modify the Turing machine example by introducing a subuniverse *Nonhalting* of the universe *Control*. In an obvious way, the halting problem for Turing machines reduces to the problem whether the constraint *CurrentState* belongs to *Nonhalting* is violated. But of course the *Nonhalting* universe is very artificial and unnatural. Often, constraints are checkable in advance. Certainly, your demon, able to evaluate a bounded number of closed terms at a time, should have no trouble to check, in each state, the validity of a constraint $t = true$.

Q: You defined the tape to be infinite. I prefer finite tapes growing if and when necessary.

A: The predefined geometry of Turing tapes makes the distinction somewhat artificial. In this connection it makes more sense to consider Kolmogorov-Uspensky machines [KU] with (finite at every moment) tapes of changing geometry. (We discussed KU machines once [Gu3].) The tape of a KU machine gives rise to a dynamic universe in a very natural sense. To handle this situation, we may use a countable universe *Reserve* (also dynamic) and an external distinguished element *New* subject to the integrity constraint $New \in Reserve$. Whenever it is required that the tape gets another cell, *New* is deleted from *Reserve* and added to the tape. Of course, when we evaluate *New* next time, it belongs to *Reserve* again. Actually, adding a new element to the tape requires some work because, contrary to *Reserve*, the tape has a relatively rich structure. Let me skip the details.

Q: By the way, the halting problem easily reduces to the following decision problem: Given an EA, tell whether it will eventually reach a state with contradicting updates. Thus, this consistency problem for EAs is undecidable. I imagine that for certain applications only consistent EAs are appropriate. What do you do?

A: Often very simple syntactic restrictions suffice to ensure consistency. Sometimes, it may be appropriate to require that, for each basic function f, the guards of different updates of f are mutually exclusive. This guarantees that no basic function is updated twice at the same step. (Since the problem of mutual exclusivity of given boolean formulas is co-NP complete, one may want to require some kind of explicit mutual exclusivity.) In general, every evolving algebra $\mathcal{A}$ can be modified into a consistent variant $\mathcal{A}'$ making 2 steps per each step of $\mathcal{A}$ and halting (with setting an error flag to *true* if desired) when $\mathcal{A}$ encounters inconsistency. The idea is to check the consistency of $\mathcal{A}$'s transition rules in given state of $\mathcal{A}$ before executing them.

7. Example 3: The strcpy function

Q: Will you show me an example where the algorithm is given by a program?

A: Yes. Consider the piece $\mathcal{P}$ of C code in Figure 1 which is taken from Kernighan and Ritchie's book "The C Programming Language" [KR, page 105].

```
void strcpy (char *s, char *t)
{
     while (*s++ = *t++)
          ;
}
```

Figure 1: The strcpy function.

Q: I know too little about C. What does $\mathcal{P}$ do and what does this unpronounceable "strcpy" stand for? Is $\mathcal{P}$ well formed? The `while` loop does not seem to do anything.

A: $\mathcal{P}$ is well formed and defines an algorithm that copies a character string from one place in memory to another. I guess, "strcpy" abbreviates "string-copying". The pointer variables `t` and `s` point initially to the first memory locations in the original place and the new place respectively; `*s` is the content of the memory location `s`. The equality sign denotes assignment. The expression `*s++` evaluates to the content of the location `s` and has a side effect of resetting `s` to the next location. Kernighan and Ritchie admit that the code is "cryptic at first sight" [KR, page 106].

I should make a remark about the notion of a string in C. Usually, people have in mind strings of printable characters, but this is not important for us here. The important restriction is that a string does not contain the so-called null character which is used to indicate the end of a string representation in memory. The memory can be viewed as a linearly ordered set of memory locations holding characters. Each character fits into one location. (There isn't much difference in C between characters, bytes and very small integers. In particular, the null character corresponds to the number zero.) Because of the necessary null character at the end, it takes $l+1$ memory locations to store a string of l non-null characters.

Q: How can an assignment be the condition of a `while` loop?

A: An assignment statement returns the value assigned. Nonzero values represent truth and 0 represents falsity in C. The value 0 is returned when the null character is encountered.

We will construct an evolving algebra $\mathcal{A}$ modeling the `strcpy` algorithm. First, we will describe the universes of $\mathcal{A}$ and most of the basic functions. Then we will describe the transition rules and the remaining basic functions. Since $\mathcal{P}$ uses pointers to memory locations, $\mathcal{A}$ has universes *Char*, *Loc* and *Addr*. Elements of *Char* will be called *characters*; *null* is a distinguished character. The universe *Loc* comes equipped with a successor function. Elements of *Loc* will be called *memory locations*. (C allows more arithmetic on memory locations, but only the successor function is used in $\mathcal{P}$.) A dynamic function *LocCont* from *Loc* to *Char* assigns characters to memory locations. Elements of *Addr* will be called *addresses*. A dynamic function *AddrCont* assigns memory locations to addresses. (According to C, pointer addresses are composed of memory locations and *AddrCont* is defined by *LocCont*, but this information is not necessary for explaining $\mathcal{P}$ and will be ignored.) We assume that the two content functions are total (on their respective domains).

Q: An address probably identifies a little block of a memory where the address of a pointer can be stored. What is the difference between *Addr* and *Loc*?

A: You are right of course. The size of the block depends on the implementation. For our purposes, there is no reason to go into these details.

To reflect (a somewhat simplified version of) the parse tree for $\mathcal{P}$ corresponding to the official grammar of C and shown in Figure 2, $\mathcal{A}$ has a universe *Parsetree* that comprises nodes of the parse tree. A distinguished element *Root* and unary functions *Child1, Child2, Child3, Parent* have the obvious meaning. In addition, there is a universe of labels. A function *Label* assigns labels to nodes as shown in Figure 2. (The labels do not include expressions in parentheses which are added for greater clarity.) Each label is a distinguished element and therefore can be mentioned in rules.

In order to simulate $\mathcal{P}$, we should know at each moment where we are in the program. To this end, a dynamic distinguished element C will be the currently active node of *Parsetree*. Initially, C is the root. It will be convenient to abbreviate *Child1(C), Child2(C), Child3(C)* as *C1, C2, C3* respectively. Let me write a few transition rules governing the movement of C. I presume that C never stays two moments in a row at the same place; in other words, it will never have the same value at two consecutive states.

```
if C = Root then
    if Val(C1) = undef then C := C1
    elseif Val(C2) = undef then C := C2
    elseif Val(C3) = undef then C := C3
    endif
endif
```

Q: What is *Val*?

A: *Val* is a function on *Parsetree*. Think about each node n of the parse tree as the root of the corresponding subtree. When the code corresponding to the subtree has been executed, the resulting value is assigned to n. For example, the execution in

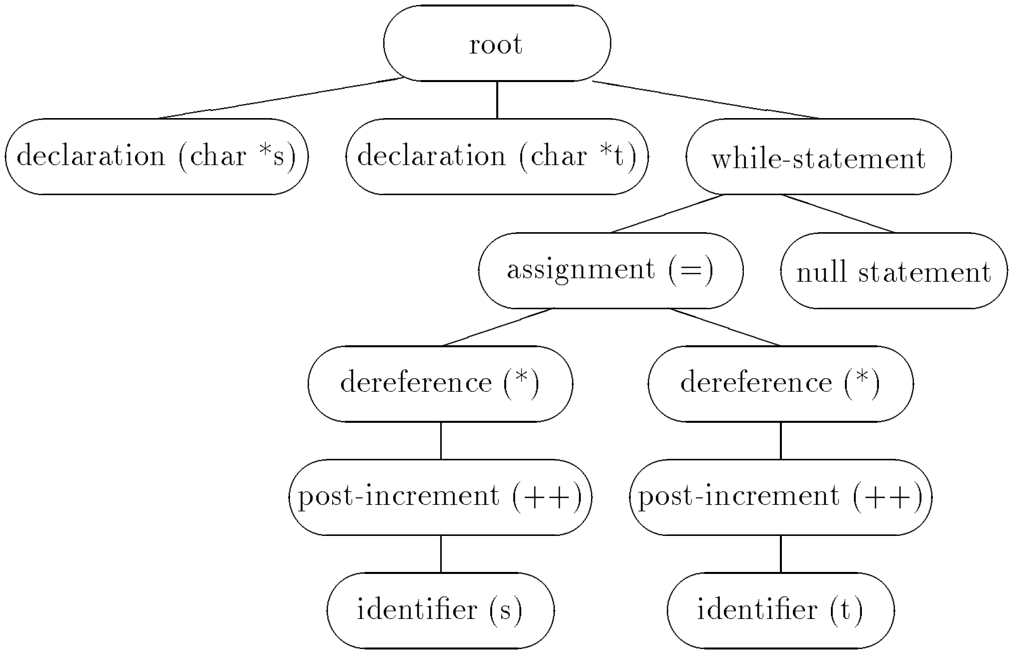

Figure 2: Our parse tree for `strcpy`.

question may be an expression evaluation; then the value of the expression is assigned to n. Initially, *Val* is nowhere defined.

Q: What if n corresponds to a statement that does not return any value?

A: When the statement has been executed, the value *done* will be assigned to n.

Q: What will be the value of a declaration node?

A: You'll see. Here are some additional rules governing the movement of C and using some obvious abbreviations.

```
if C is a leaf then C := Parent(C) endif

if C has exactly one child then
      if Val(C1) = undef then C := C1
      else   C := Parent(C)
             Val(C1) := undef
      endif
endif
```

Q: I understand that the case of the while-statement node is an exception, but what about the case of the assignment node? Is it treated the same way as the case of the root node?

A: No. The problem is that the definition of C does not specify whether the left or the right expression is evaluated first. This was a deliberate decision [KR, pp. 53–54]; the order of evaluation is left to the implementer. To reflect the implementer's decision, we use an external function *ChooseChild*.

```
if Label(C) = assignment then
    if Val(C1) = undef and Val(C2) = undef then C:=ChooseChild(C)
    elseif Val(C1) = undef then C := C1
    elseif Val(C2) = undef then C := C2
    else  C := Parent(C)
          Val(C1) := undef
          Val(C2) := undef
    endif
endif
```

Q: Why do you reset the values of *C1, C2* to *undef*?

A: To have a proper environment when C comes down to the assignment node next time.

Q: I do not see why do you need *ChooseChild*. Since you have allowed nondeterminism to resolve contradicting rules, you may as well use it. To choose a child, just say that C gets *C1* if *Val(C1)* is undefined and that C gets *C2* if *Val(C2)* is undefined. If the values of both *C1* and *C2* are undefined then one of them is nondeterministically chosen.

A: You are right. As an implementor, I would prefer the *ChooseChild* version though. It is more explicit. Now, let me give you the remaining rules. They are grouped with respect to the location of C.

```
if Label(C) = declaration then
    Val(C) := ChooseAddr
    AddrCont(ChooseAddr) := PredefinedLoc(C)
endif
```

Here *ChooseAddr* is a zero-ary external basic function that returns an address.

Q: Why should it be external?

A: $\mathcal{P}$ doesn't tell us how to allocate addresses.

Q: You suppose, I guess, that both occurrences of *ChooseAddr* evaluate to the same value.

A: Yes, *ChooseAddr* has a definite value in each state (where it is evaluated).

Q: And what is the *PredefinedLoc* function? You did not mention it earlier.

A: A call to `strcpy` should provide two parameters, namely, the initial values of `s` and `t`. *PredefinedLoc* carries this information.

Q: Is it an internal or external function?

A: In this case, the distinction does not matter. Since the two parameters should be known when $\mathcal{P}$ starts, *PredefinedLoc* can as well be an internal function, a static internal function. Further,

```
if Label(C) = identifier then Val(C) := AddrCont(Val(Decl(C))) endif
```

Q: What is *Decl*?

A: *Decl* maps each identifier node to the declaration node where the identifier in question is declared. The left identifier node is mapped to the left declaration node and the right identifier node is mapped to the right declaration node. Here is the rule for post-increment nodes.

```
if Label(C) = post-increment and Val(C1) $\neq$ undef then
    Val(C) := Val(C1)
    AddrCont(Val(Decl(C1))) := Val(C1) + 1
endif
```

Q: Let me understand this complicated second update. It would be easier for me to speak about a specific identifier node, say, the left one, which is related to the pointer variable `s`. Then *Decl(C1)* is the left declaration node and *Val(Decl(C1))* is the address of `s`. On the other hand, *Val(C1)* is the memory location pointed to by `s`. Thus, we are changing `s` to point to the next location (the successor of the old value of `s`).

A: That is right. Notice that *Val(C)* does not reflect the change. But the next time when our identifier node becomes active, it will have a new value.

The dereferencing operator (also known as the indirection operator) of C is formalized by our *LocCont* function. However, in the case of the left child of the assignment node, we are not really interested in dereferencing.

```
if Label(C) = dereference and Val(C1) $\neq$ undef then
    if C = Child1(Parent(C)) then Val(C) := Val(C1) endif
    if C = Child2(Parent(C)) then Val(C) := LocCont(Val(C1)) endif
endif
```

The assignment-node rule should be clear now.

```
if Label(C) = assignment and Val(C1) $\neq$ undef and Val(C2) $\neq$ undef then
    LocCont(Val(C1)) := Val(C2)
    Val(C) := Val(C2)
endif
```

Q: I do not understand why the result of the assignment statement is *Val(C2)* rather than simply *done*.

A: As I mentioned before, an assignment statement returns the assigned value. This feature allows C programmers to use commands like $a = b = c = 0$. We take advantage of this feature to indicate when the string copy operation should halt. When *null* has been copied, it is passed up to the while-statement node halting execution of the `while` statement.

```
if Label(C) = while-statement then
    if Val(C1) = undef then C := C1
    elseif Val(C1) $\neq$ null then
            C := C2
            Val(C1) := undef
    else    C := Parent(C)
            Val(C) := done
            Val(C1) := undef
    endif
endif
```

Q: Is all that a part of your EA description of C?

A: No, not really, but the description used to look like that.

8. Sequential and nonsequential evolving algebras

Q: Now I understand what the program of a sequential EA is; it is essentially a finite set of conditional function updates. But I still do not understand what a sequential EA is exactly. Do you identify a sequential EA with its program?

A: It is reasonable to identify sequential EAs and their programs and this is done sometimes [GR]. Often it is convenient, however, to avoid a formal definition. For example, many authors define precisely the language of a first-order theory, the set of theorems, etc. but leave the notion of first-order theory itself informal.

Q: What do you gain by leaving the notion of sequential EAs informal?

A: Usually, constructing an EA for a given algorithm $\mathcal{F}$ involves more than just writing a program. You indicate a class of static structures which includes presumably the presentations of all possible or all relevant states of $\mathcal{F}$. You may want also to indicate a class of initial static structures. See for example how Turing machines are formalized above. It may be convenient to view all that as a part of the definition of the EA.

Q: What non-basic rules are most common?

A: Most common is the generalization of basic rules where the terms are allowed to have free individual variables. This generalization is used even in the sequential case (as a syntactic sugar) to make the program more succinct [BRo1 and BRo2], but it is indispensable in the case of unbounded parallelism. Consider, for example, a tree algorithm that colors red – in one step – all children of the currently active node whenever the active node is green. This may be expressed by the rule

```
if Color(C) = green and x is a child of C then
    Color(x) := red
endif
```

which is not equivalent to any (finite) set of basic rules. In the paper on Occam, we used (limited) universal quantification [GMs, BRi1]. To show why one may need universal quantification, let us suppose that the tree algorithm, mentioned above, colors the active node C green if all children of C are red. This may be expressed by the rule

```
if (∀ child x of C)(Color(x) = red) then
    Color(C) := green
endif
```

Q: You mentioned asynchronous distributed computing a while ago. I have been wondering; how do you deal with such computations? It seems that going from one state to another is a basis of the EA approach. In the case of a distributed asynchronous computation, it may be unclear what the states are.

A: Syntactically, an evolving algebra for a distributed asynchronous algorithm may look like one that works in (discrete) sequential time. The difference is in the definition of a run. Instead of one demon, you may have a multitude of demons responsible for different rules. Whenever conditions are right, demons perform the required changes, but some demons may work with lightning speed whereas some others may be lazy. Let me again refer you to the paper [GMs] with Larry Moss on the programming language Occam. By the way, the sequential interpretation and the parallel interpretation of a list of commands coexist in Occam as well as in Parlog. This is one way to generalize the basic language of transition rules.

Q: Can one compose evolving algebras?

A: Of course [GR], but this is a topic for another conversation.

Acknowledgement. The EA project would probably die without the active participation of Egon Börger. I benefited greatly from the collaboration with Andreas Blass, Larry Moss and Dean Rosenzweig. The `strcpy` example was "lifted" from Jim Huggins' homework for the Principles of Programming Languages class. I am very greatful to Bob Blakley, Erich Grädel, Angelica Kappel, Gerti Kappel, Anatol Slissenko, Ed Wimmers and Moshe Vardi for their comments.

References

[BB] Christoph Beierle and Egon Börger, "Correctness proof for the WAM with types", Springer LNCS 626 (1992), 15–34.

The Börger-Rosenzweig's correctness proof for compiling Prolog to WAM [BRo1] is extended to the Protos-L language which extends Prolog with polymorphic (dynamic) types.

[Bl] Bob Blakley, "A Smalltalk Evolving Algebra And Its Uses", Ph. D. Thesis, University of Michigan, 1992.

An early student work on evolving algebras (the late date of 1992 is accidental). A reduced version of Smalltalk (called Mumble) is formalized and studied.

[Bo1] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control", CSL'89, 3rd Workshop on Computer Science Logic (eds. E. Börger et al.), Springer LNCS 440 (1990), 36–64.

[Bo2] Egon Börger, "A Logical Operational Semantics for Full Prolog. Part II: Built-in Predicates for Database Manipulations", in Proc. of Mathematical Foundations of Computer Science 1990 (ed. B. Rovan), Springer LNCS 452, 1–14.

[Bo3] Egon Börger, "A Logical Operational Semantics of Full Prolog. Part III: Built-in Predicates for Files, Terms, Arithmetic and Input-Output", in: Y. Moschovakis (ed.), "Logic from Computer Science", Mathematical Sciences Research Institute Publications, vol. 21, Springer Verlag, 1992, 17–50.

These are the original 3 papers of Börger where he gives a complete evolving-algebra formalization of Prolog with all features discussed in the WG17 of ISO. An improved (tree based) version is found in [BRo2].

[BD] Egon Börger and K. Daessler, "PROLOG: DIN Papers for Discussion", in ISO/IEC JTCI SC22 WG17 No. 58, National Physical Laboratory, Middlesex, 1990, 114 pages.

A version of [Bo1–3] proposed to WG17 as a complete formal semantics of Prolog.

[BRi1] Elvinia Riccobene and Egon Börger, "A Formal Specification of Parlog", in "Semantics of Programming Languages and Model Theory" (eds. M. Droste and Y. Gurevich), Gordon and Breach, 1993, to appear. [Preliminary version in Springer LNCS 567 (1992) and 592 (1992).]

An evolving algebra formalization of Parlog, a well known parallel version of Prolog.

[BRi2] Elvinia Riccobene and Egon Börger, "A Mathematical Model of Concurrent Prolog", CSTR-92-15, Dept. of Computer Science, University of Bristol, 1992.

An evolving algebra formalization of Ehud Shapiro's Concurrent Prolog.

[BRo1] Egon Börger and Dean Rosenzweig, "The WAM – Definition and Compiler Correctness", TR-14/92, Dipartimento di Informatica, Universita di Pisa, June 1992, 57 pages. [Previous version in Springer LNCS 533 and 592.]

The first substantial example of the successive refinement method in the area. A hierarchy of evolving algebras provides a solid foundation for constructing provably correct compilers from Prolog to WAM. Various refinement steps take care of different distinctive features ("orthogonal components" in the authors' metaphor) of WAM making the specification as well as the correctness proof modular and extendible; an example of such an extension is found in [BB].

[BRo2] Egon Börger and Dean Rosenzweig, "A Simple Mathematical Model for Full Prolog", TR-33/92, Dipartimento di Informatica, Universita di Pisa, October 1992, 23 pages.

An improved version of [Bo1–3].

[BS] Egon Börger and Peter Schmitt, "A formal operational semantics for languages of type Prolog III", CSL'90, 4th Workshop on Computer Science Logic (eds. E. Börger et al.), Springer LNCS 533, 1991, 67–79.

An evolving algebra formalization of Alain Colmerauer's constraint logic programming language Prolog III.

[GR] Paola Glavan and Dean Rosenzweig, "Communicating Evolving Algebras — Part 1", Manuscript, University of Zagreb, Croatia, October 1992.

The first part of a two-part paper, presented at CSL'92, aimed to "demonstrate the power of the framework by providing simple and transparent models of the (polyadic) π-calculus of Milner, the Chemical Abstract Machine of Berry and Boudol, and TCCS of de Niccola and Henessy".

[GKS] Georg Gottlob, Gerti Kappel and Michael Schrefl, "Semantics of Object-Oriented Data Models – The Evolving Algebra Approach", in "Next Generation Information Technology" (eds. J. W. Schimdt and A. A. Stogny), Springer LNCS 504, 144–160.

"In this paper we show how evolving algebras can be used in particular to define the operational semantics of object creation, of overriding and dynamic binding, and of inheritance at the type level (type specialization) and at the instance level (object specialization)."

[Gr] George Gratzer, "Universal Algebra", Van Nostrand, 1968.

[Gu1] Yuri Gurevich, “Logic and the challenge of computer science”, In “Current Trends in Theoretical Computer Science” (ed. E. Börger), Computer Science Press, 1988, 1–57.

The introduction and the first use of evolving algebras (at the end of the paper).

[Gu2] Yuri Gurevich, “Algorithms in the world of bounded resources”, in “The universal Turing machine – a half-century story”, (ed. R. Herken), Oxford University Press, 1988, 407–416.

[Gu3] Yuri Gurevich, “Kolmogorov machines and related issues: The column on logic in computer science”, Bulletin of EATCS, No. 35, June 1988, 71–82.

[Gu4] Yuri Gurevich, “Evolving Algebra 1993: Lipari Guide”, in “Specification and Validation Methods”, Ed. E. Boerger, Oxford University Press, 1995, 9–36.

[GMr] Yuri Gurevich and James Morris, “Algebraic operational semantics and Modula-2”, CSL’87, 1st Workshop on Computer Science Logic, Springer LNCS 329 (1988), 81–101.

An abstract of [Mo].

[GMs] Yuri Gurevich and Larry Moss “Algebraic Operational Semantics and Occam”, Springer LNCS 440, 176–192.

The first application of evolving algebras to distributed parallel computing with the challenge of true concurrency.

[GH] Yuri Gurevich and James Huggins, “The Evolving Algebra Semantics of C”, CSE-TR-141-92, EECS Department, University of Michigan, 1992.

The method of successive refinements is used to give a more succinct semantics of the C programming language.

[JM] David E. Johnson and Lawrence S. Moss, “Dynamic Syntax”, 1993 (to appear).

Distributed Evolving Algebras are used to model formalisms for natural language syntax.

[KS] Gerti Kappel and Michael Schrefl, “Cooperation Contracts”, Proc. 10th International Conference on the Entity Relationship Approach (ed. T. J. Teorey), San Mateo, California, 1991, ER Institute, 285–307.

The authors introduce the concept of cooperative message handling and use evolving algebras to give formal semantics.

[KR] Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, Prentice-Hall, 1988, Englewood Cliffs, NJ.

[KU] A. N. Kolmogorov and V. A. Uspensky, “To the Definition of an Algorithm”, Uspekhi Mat. Nauk 13:4 (1958), 3–28 (Russian); English translation in AMS Translations, ser. 2, vol. 21 (1963), 217–245

[Mo] James Morris, “Algebraic operational semantics for Modula 2”, Ph.D. thesis, University of Michigan, 1988.

The earliest formalization of a real-life language. In the meantime, the methodology has developed enabling more elegant descriptions, but one has to start somewhere.

[Pr] Vaughan R. Pratt, “Dynamic algebras and the nature of induction”, 12th ACM Symposium on Theory of Computation, 1980.

[Sch] Arnold Schönhage A, “Storage Modification Machines”, SIAM J. on Computing 9:3 (1980), 490–508.

[Tu] Alan M. Turing, “On computable numbers with an application to the Entscheidungsproblem,” Proc. London Math. Soc. 42 (1937) 230–265; correction, *ibid*, No. 43 (1937), 544–546.

[WG17] “PROLOG. Part 1, General Core, Committee Draft 1.0”, ISO/IEC JTCI SC22 WG17 No. 92, 1992.