

Dynamic Interpretations of Constraint-Based Grammar Formalisms

Lawrence S. Moss

Mathematics and Computer Science Departments, Indiana University, Bloomington, IN 47405 USA.

and

David E. Johnson

Mathematical Sciences Department, Thomas J. Watson Research Center, IBM Research Division, Yorktown Heights, NY 10598 USA.

March 10, 1995

Abstract. We present a rendering of some common grammatical formalisms in terms of evolving algebras. Though our main concern in this paper is on constraint-based formalisms, we also discuss the more basic case of context-free grammars. Our aim throughout is to highlight the use of evolving algebras as a specification tool to obtain grammar formalisms.

Key words: Constraint-based formalism, evolving algebra

1. Introduction

The main point of this paper is to explore the use of *evolving algebra* methods in the specification of grammar formalisms. The subject of evolving algebras was introduced by Yuri Gurevich around 1985 as a means of specifying various operational aspects of computation. It provides a clear and uniform way to specify algorithms, programs, programming languages, operating systems, circuitry, etc., often at a level of detail appropriate for reasoning about the system in question. It is unashamedly operational, and yet the mathematical models produced abstract away from details such as machine execution.

The claimed advantages of using evolving algebra methods are chiefly those associated with specification: by having clear specifications, people can use, test, modify a system, as well as teach it to others. On the other hand, the EA work is not an attempt to answer some of the questions which drive the declarative or denotational approaches to semantics, such as providing mathematical models for difficult features of computation, or giving foundations for program equivalence. The evolving algebra framework does seem to provide some general tools for useful specification, and we believe that these tools might be fruitfully applied in linguistics as well.

We feel that some current grammar formalisms are *dynamic*, on an intuitive level at least. This means that one thinks of some model changing in time as some sort of linguistic data is processed. However, there is a strong tendency to hide the dynamic features of these formalisms and to concentrate on *declarative*

information. Perhaps the easiest way to make the point is to consider context-free production, one of the most basic mathematical models found in linguistics. The main concept there is *derivation*, a dynamic one. It is usually replaced by that of *yield* and *language*, and these are static. Of course, practically every approach in theoretical and computational linguistics uses formal methods which go beyond context-free production. Our point remains that a significant number of these extensions are based on dynamic intuitions, even when they are formalized in declarative terms.

As linguistic formalisms become more sophisticated, there will come a need for clear methods of operational specification. That is, declarative frameworks which downplay the issues of how representational structures are created and manipulated in favor of static accounts do so at the cost of being harder to understand. In addition, if one really ignores the issue of how to find a representation (a parse according to some framework, say, to be concrete), then a computational system is likely to make choices when it searches for a representation; these choices might make using the system different from working within the declarative framework. We are after general tools which could make the operational specification of linguistic systems a straightforward matter. One of our main points in this paper is that work coming from the operational semantics of programming languages might provide such tools.

Another benefit of the incorporation of precise operational specifications would be the ability to do *correctness proofs* for algorithms and implementations used in computational linguistics. Happily, evolving algebra methods have given semantics to Prolog (in a large number of versions) and many other currently-used languages, always including the impure features that are likely to be critical in applied work. In addition, the work is beginning to be applied in the direction of real-time computing. We feel this could be important in designing and specifying linguistic systems (say for comprehension) where timing factors play a leading role.

1.1. THE RESEARCH PROGRAM

At this point, we should clarify the overall point of the paper, and also mention how it fits in to an overall research program concerning evolving algebras and grammar formalisms.

This paper is intended to show that many linguistic formalisms have a dynamic side that is not captured in standard formalisms, and that evolving algebras can capture this in a convenient way. Of course, this claim is not the final goal of the work. As it stands, the evolving algebra methods are overly powerful. What would be needed next are restrictions of the framework, and tools for reasoning about the specifications.

Further, our work to date has been directed towards showing that existing frameworks may be captured in terms of evolving algebras. But eventually we will want to let evolving algebras take the lead, and to develop specification meth-

ods which will enable linguists to propose frameworks which are more directly dynamic.

A question that often comes up is whether the overall program of dynamic grammar formalisms is necessarily linked to evolving algebras. It is not. We feel that it would be interesting to start with a different computational formalism and then take up the same questions that we address. There are many other formalisms to start from, mostly coming from work in concurrency theory. Our choice of evolving algebras has the advantage that it is currently being developed as a specification tool for a wide variety of “real-world” computing problems. It is not of interest solely to theoreticians, but at the same time it is generating a body of theoretical work.

1.2. OTHER WORK ON EVOLVING ALGEBRAS AND GRAMMAR FORMALISMS

We have written two other papers on the topic of evolving algebras and grammar formalisms; these are (JM) and (MJ). Together with this one, they show that distributed evolving algebras can be used to specify a number of grammar formalisms, including Extraposition Grammar, Head Grammar, Attribute Grammar, Indexed Grammar, Tree-Adjoining Grammar, Dynamic Dependency Grammar, and various grammars based on graphs. Given a grammar of one of these types, we provide an evolving algebra whose runs correspond to the parses according to the grammar. This paper takes up feature-based grammars. Moreover, as in most of our other examples, the first step in getting an EA rendering is to consider a context-free skeleton for the grammar, and then to see how the intuitive ideas of the grammar correspond to changes in the EA for context-free production. Here we not only do this, but we also provide a new look at bottom-up context-free production.

In feature-based formalisms, grammars are given by statements in some logic, and parses are essentially models which satisfy a given grammar. Such formalisms are on the surface quite static: the purely logical definitions have absolutely nothing to say about how models are found. On the other hand, the utility of such frameworks is not due solely to their grounding in a formal logic. The fact that satisfiable sentences have minimal models which can be computed, and indeed all of the algorithms based on unification, suggest that some dynamic intuitions are evidently being captured in the formalism.

In order to see *why* we want a dynamic rendering of the graph models of feature logic, consider first the opposite view, as put forward by Shieber (S), p. 54 :

Because of the unique character of constraint-based formalisms, a semantics for a formalism based on a naive redefinition of the notion of derivation is problematic, not only because it introduces a procedural notion where one is not needed, but also because information flow in a derivation is not easily captured by working only top-down – as in a derivation – or, for that matter, only bottom-up. Though possible, such a semantics would be geared to proving

the correctness of a purely top-down or bottom-up algorithm. We would like to be more procedurally agnostic than that.

Our point is that if one does not want to be procedurally agnostic, then having a dynamic rendering of constraint-based formalisms might be useful. There might be more than one such rendering, and the most useful ones would be *high-level*, but still procedural. That is, they should not amount to fully-described implementations, because those would contain too much detail. As we mentioned above, a suitable rendering could be used as the basis of correctness proofs. In fact, by constructing procedural interpretations (see Sections 4 and 4.1), we feel that we have some insight into what the formalism is about, and in what ways it could be extended.¹

In addition to the conceptual interest in dynamism, there are technical points as well: the issue of negation in feature-based work. The gist of the matter is that in much work on feature structures and their applications feature structures are to be regarded as partial descriptions of structures which are “under construction” in some sense (see (K; R) , but for a contrasting view, see (J)). To say that an underspecified structure S satisfies the negation of a positive constraint φ is then a problematic matter: given an underspecified structure S , does $S \models \neg\varphi$ mean that $S \not\models \varphi$? Or does it mean that in the course of extending S , we never satisfy φ ? The second view seems to be the prevalent one. For example, Keller (K), pp. 20–21 writes

... the presence of negative constraints on features and values cannot be adequately accounted for by the absence of the corresponding positive information. The fact that a given feature description does not require that a particular feature label has a particular value cannot be taken as evidence to imply that the feature *does not* have that value. Indeed, this is one of the hallmarks of partial information. Information about features and values may ‘grow’ as additional constraints are taken into account during a computation (i.e., through unification).

As this quotation suggests, there is a dynamic view underlying the use of feature structures. The treatment of negation *should* reflect that view. But standard feature logics work mainly with a single structure, and this makes for a kind of mismatch with negative information. To get around this, we find proposals using three-valued logic (Dawar and Vijay-Shankar (DV-S)) and intuitionistic negation via Kripke semantics (Moshier and Rounds (MR)). Both of these have some connection to dynamic ideas. In addition, Carpenter (C) develops a typed framework to handle certain kind of negation, and his approach seems farther from dynamic intuitions.

¹ In fact, one of the key notions in Shieber’s development is that of a parse tree for constraint formalisms. This inductively defined concept leads to the notion of a yield, and hence of the language determined by a grammar. Our dynamic versions stand in the same relation to these as our dynamic renderings of context-free production stand to ordinary parse trees.

Our point regarding negation in this paper is to show how it is possible to directly incorporate a variety of dynamic styles of negation. In effect, because our overall semantics is dynamic, the extensions of our semantics are relatively minor. (This is in contrast to the more standard work, where adding negation had meant major changes.) We discuss this in Section 5.

2. Evolving Algebras

The formal underpinning of our work is the evolving algebra approach to the operational semantics of programming languages. There are many definitions involved, about as many as would be needed to give a completely rigorous treatment of, say, first-order logic. On the other hand, very few theoretical results are needed in the development.

A very short summary of the formalism would be that it uses many-sorted first-order structures as the *states* of a structure. However, in contrast to all of the standard uses, the constants and function symbols are taken to be *dynamic*. This means that they might change according to a set of *transition rules*. These rules are part of an evolving structure. We also have certain initial structures. Given these, we can speak of a *run* of our structure: it would be a sequence of states which obey the transition rules. The complexity in the formal definition stems from the the ability of an evolving algebra to change its structure by adding or deleting elements, as well as the formal details of how the transition rules work. In addition, the evolving algebras here are all *distributed*, since we feel that these (as opposed to sequential structures) are the best vehicles for rendering the dynamism which we find in grammar formalisms.

The best sources are two papers by Gurevich: his tutorial (G1), and his very detailed presentation in (G2). In addition, our papers (JM; MJ) give a quick summary of the details as they apply in linguistic work. We have decided in this paper not to repeat all of those discussions: first, they are long, and we have nothing new to add in this regard; and second, the one technical result of the paper is stated without proof. This is because our point in this work is not to make technical progress but rather to make a conceptual proposal concerning constraint grammars. We feel that the flavor of the dynamic approach can be gleaned from the statements of how our models work *even in the absence of all the details*. In a sense, this is part of our point. Specifications using evolving algebras are typically easy to read. Of course, the interested reader will have to look up the relevant details in order to begin to use the method. Although the precise definitions are long, they are for the most part elementary.

In addition to the works mentioned above, there is one additional important resource: the annotated bibliography of work on evolving algebras Börger (B). It is expected that this reference will continue to be updated, covering technical

contributions, applications, and introductory papers. It is perhaps the best overall source on the subject.

3. Context-Free Grammars Construed Dynamically

We believe that the image behind context-free derivation is a tree structure growing according to production rules. Further, this growth happens at different points in the tree independently. What we want to do here (following (JM; MJ)), is to see how CF derivation can be understood using the notions of evolving algebra.

Consider the trivial grammar $S \rightarrow AB$, $A \rightarrow a$, $A \rightarrow AA$, $B \rightarrow b$. This grammar generates the string aab , and there is only one way to get it. Here is our intuitive understanding of this derivation. Initially we have a tree consisting of a single node, say n_0 , labeled S . Then according to the production $S \rightarrow AB$, n_0 sprouts two children. The left child n_1 is labeled A and the right child n_2 is labeled B . After this, the root of the tree will not evolve further, and n_1 and n_2 evolve independently. Later n_1 sprouts two children n_3 and n_4 labeled A , and independently of this n_2 has one child n_5 labeled b . Meanwhile, n_3 and n_4 have children n_6 and n_7 labeled a , respectively.

At this point, the derivation is complete except for the calculation of the yield aab . This is a recursive process going back up the tree. The only important points are that terminals get their yield directly, while nodes higher in the tree get their yields by concatenating the yields of their children.

The process *could* be linearized as in the standard presentation; that is, it could be presented as if it happened one step at a time. But it is, we claim, more natural to think of the evolution as many independent actions happening concurrently and independently.

Here is a description of the distributed EA machinery which is appropriate for CF derivations. We have a universe \mathcal{NODE} of nodes of an evolving parse tree. (A “universe” is just a set. Our models will have many universes, and they will also have functions between various of these universes.) This universe is the universe of active processes of the EA. The \mathcal{NODE} universe has a distinguished element *root*, and it comes with partial functions *first-child*, *last-child*, and *next-sibling*. (Later on, we also assume that it comes with partial functions child_k which give the k -th child of a given node.) In addition, there is a universe \mathcal{SYM} of terminal and nonterminal symbols of the grammar. There is also a map $Type : \mathcal{NODE} \rightarrow \mathcal{SYM}$. Of course, all functions mentioned will be dynamic since we think of a tree as growing in time.

We also have a universe \mathcal{MODE} of modes, and a function *mode* from nodes to modes. This function is dynamic; processes change their mode in the evolution of a structure. As in the Gurevich and Moss (GM) EA treatment of the programming language Occam, the modes will be *starting*, *working*, *reporting*, and *dormant*. The initial state of our machine will have a single n in \mathcal{NODE} , and this node will be in

starting mode. We also write, e.g., “ n changes to working” rather than “ $\text{mode}(n) := \text{working}$.” That is, mode is technically a function, but we prefer a notation closer to ordinary English. For the calculation of the yield of a parse tree, we will need a universe \mathcal{STR} of strings on our alphabet. This universe comes with concatenation as a primitive operation. Finally, there will be a map output from \mathcal{NODE} to \mathcal{STR} .

In addition to universes, an evolving algebra also has *transition rules*. These are exactly analogous to the transition table for an automaton or Turing machine. In fact, evolving algebras are essentially automata where we allow the collection of states to be the first-order structures of some signature, and where we generalize transition rules appropriately.

Below we list two transition rule schemata, (1) and (2), and two rules (3) and (4), for our machine. These correspond to: (1) the spawning of new nodes in a parse tree; (2) the base case of the recursive definition of yield; (3) the general step in the calculation of yield; and (4) the way the machine closes down.

- (1) If $\text{type}(n) = X$ and n is starting,
 then n changes to working,
 and n has k new children m_1, \dots, m_k
 of types Y_1, \dots, Y_k respectively,
 and all children of n change to starting.

We understand rule (1) as a universally quantified statement; for all nodes n , if n is in starting mode, then this rule may fire and the overall structure is accordingly updated. (1) is a schema in the sense that each rule $X \rightarrow Y_1 \cdots Y_k$ with X a nonterminal of the grammar gives a rule of this form. In specifying that the children of n be new, we make use of a central feature of evolving algebras.

- (2) If n is starting, and $\text{type}(n) = x$,
 then n changes to reporting,
 and $\text{output}(n) = \langle \text{type}(n) \rangle$.

Rule (2) is a schema, one instance for each terminal symbol x . Also $\langle x \rangle$ is the one-element sequence $\langle x \rangle$ corresponding to a terminal symbol x .

- (3) If n is working, and all children of n are reporting,
 then $\text{output}(n) = \text{concatenate}_{n \rightarrow m}(\text{output}(m))$,
 all children m of n change to dormant,
 and n changes to reporting.

In rule (3), we use the notation $\text{concatenate}_{n \rightarrow m}(\text{output}(m))$ to mean the obvious concatenation (in order) of the yields of children of n .

- (4) If root is reporting,
 then root changes to dormant.

Rule (4) is included merely to tidy the overall picture. A rule of this form is more useful in computational settings where nodes might cycle through the different modes many times in the course of a computation. This possibility does not seem to be used the linguistic formalisms we have considered.

Looking back at the grammar in the beginning of this section, we see that it uses four instances of schema (1), two instances of schema (2), as well as rules (3) and (4).

We can now imagine the evolution of a machine loaded with a fixed grammar. The machine begins with its root in starting mode, and with the value S associated with that mode. S immediately becomes working, and some rule is nondeterministically chosen to apply. Then this process continues on all of the children of the root, and these continuations are independent and totally concurrent. Furthermore, we incorporate a notion of being “finished” for a given node in such a way that a node becomes finished after its children have, and the output of such a node is the concatenation of outputs of its children. (Incidentally, it is possible to simplify our evolving algebra by using fewer modes; in fact, two modes suffice. We have presented the algebra this way in part because it matches the presentation of (GM) and in part because it will be easier to adapt this particular rendering to more complicated formalisms.)

Incidentally, one of the critical questions for evolving algebra systems for distributed computation is the question of what to do if more than one rule is ready to fire in a given state of a computation. The matter is delicate, and in different applications we might make different choices. Here, we take the view that any one of the rules whose antecedent is satisfied may act, but only one can do so. This is critical for the way we understand CF productions: if a node n in a parse tree corresponds to a nonterminal X , then n may have children according to one (and only one) production rule whose left side is X . In our terms, this corresponds to having more than one rule ready to fire when n is in starting mode.

The concept of a *run* of an evolving algebra is a generalization of the concept from automata theory. It is a sequence of states related by the transition rules. Actually, this is the concept for a *sequential* evolving algebra. For a *distributed* one we need more involved definitions; see (G2; GM; JM; MJ).

We can now describe one particular run of this machine, the one corresponding to the parse whose yield is aab . Let n_0, n_1, \dots, n_7 be the nodes of the parse tree (in the usual sense) for aab . In the course of a derivation, the leaves just assume starting, reporting, and dormant modes, and the other five nodes assume these and also working mode. We need nodes corresponding to all of the nodes at all stages. So we take G to be any set with the right number of elements. (The exact identity of the nodes of G is not important, since we will be labeling the nodes shortly.) We

write the elements of G as, e.g., $n_3^{starting}$. This corresponds to the starting mode of n_3 in the parse tree. We abbreviate this by writing 3_s .

A *run* here has a binary relation \rightarrow representing causal changes between the states, and a labeling function ℓ which gives the full state description corresponding to each of the points. Here are some of the pairs in \rightarrow : $0_s \rightarrow 0_w$, $0_s \rightarrow 1_s$, $0_s \rightarrow 2_s$, $1_s \rightarrow 1_w$, $1_s \rightarrow 3_s$, $1_s \rightarrow 4_s$, $2_s \rightarrow 2_w$, $2_s \rightarrow 5_s$, $5_s \rightarrow 5_r$, $5_r \rightarrow 5_d$, $5_r \rightarrow 2_d$, \dots , $0_w \rightarrow 0_r$, $0_w \rightarrow 1_d$, $0_w \rightarrow 2_d$, $1_r \rightarrow 0_r$, $1_r \rightarrow 1_d$, $1_r \rightarrow 0_r$, $2_r \rightarrow 0_r$, $2_r \rightarrow 2_d$, $2_r \rightarrow 0_r$, $0_r \rightarrow 0_d$. For example, after the ellipsis in this list, the first nine pairs correspond to the last step of the calculation of the yield. In terms of the original parse tree, the yields of n_1 and n_2 are concatenated to get the yield of n_0 . In terms of the EA, we are applying rule (3). The nine pairs exhibit all of the mode changes in (3). The reason for putting all nine pairs into the \rightarrow relation is in order to exhibit as much causal information as possible.

The labeling function ℓ is more involved and so we shall only discuss one value for it, that of $\ell(2_w)$. The idea is that $\ell(2_w)$ should indicate the following: the process involved 2, that its type is B , the mode is *working*, the parent is 0, the first child is 5, and the yield is undefined. This exemplifies the way that ℓ generally works.

The transition rules (1)–(4) can be regarded in two ways: they are either rule schemata and can therefore be instantiated to a particular grammar G to give a fixed EA M_G . Alternatively, they can be considered as the set of transition rules of a single machine representing context-free derivation in the abstract. Such a machine would have many initial states, and each of these would have a concrete grammar embedded in it. This embedding would be uniform, and it would result in a little more structure. (For example, we might have a universe of rules.) These alternatives parallel the choice in EA work on programming languages: we might consider an EA for a particular program or an EA for the overall language.

The definition of a run is a labeled digraph meeting certain conditions. Because we have used so many modes, such a digraph is typically too large to draw completely. Indeed, since each node of the tree must go through *starting*, *working*, *reporting*, and *dormant* modes, the run will have exactly four times as many nodes as the parse tree. The edge relation on the run will be fairly dense as well; see (GM) for an example spelled out completely. Nevertheless, we can prove results about the set of runs of the EA above. The most important of these says that given a CFG G and its corresponding EA M_G , there is a correspondence between the parse trees for s according to G and runs of M_G whose output is s :

Proposition 1. For all strings $s \in STR$, s is the output of some run of M_G iff $s \in L(G)$.

The proof of this proposition consists of two inductive arguments, one on the length of derivations in the grammar, and the other on the length of runs of the evolving algebra.

Our formalization of CF derivation is not the only one possible in the EA framework. For example, we made use of concatenation of strings. Instead of this, we could have “programmed” concatenation into the rules. However, this would not be very illuminating. Indeed the main point of the EA framework is to provide a high-level language for *transfer of control* as opposed to the processing of actual data. In addition, we could have used other modes, different functions, etc. The careful writing of transition rules is something of an art.

3.1. CONTEXT-FREE PRODUCTION VIA GNF

One question which comes to mind after seeing an EA rendering of context-free production is whether the *top-down* version of context-free production is somehow intrinsic to the EA work. The answer is that it is not: We believe that *any* intuitively plausible and dynamic account of a grammar formalism can be rendered as some sort of evolving algebra. In this section, we discuss the situation with regard to production via CFG’s in Greibach Normal Form (GNF).

Consider a CFG G in GNF. This means that all of the productions are of the form $A \rightarrow a\alpha$, where $\alpha \in NT^*$. (That is, α is a string of nonterminals, possibly the empty string). Consider the following simple grammar G in GNF: $S \rightarrow bAB$, $A \rightarrow a$, $A \rightarrow aAA$, $B \rightarrow b$. One string which is generated by this grammar is $baaab$. Before presenting the formal rules of an evolving algebra, we discuss the intuitive idea behind the bottom-up parse of this string s in G .

We begin with the string s , considered as five independent units m_1, \dots, m_5 together with type identifications. (These will be elements of a universe \mathcal{NODE} later on.) The derivation may be understood as their interlocking search for their own justification for being parts of a string in the grammar. The three a tokens, m_2 , m_3 , and m_4 , non-deterministically decide to look for justification via the second and third rules, and the two b tokens m_1 and m_5 look at the first and fourth rules. In the only correct parse, m_1 decides to look for justification from the first rule $S \rightarrow bAB$. This will add three new elements, n_1 labeled S , n_2 labeled A , and n_3 labeled B . It also adds a structure: n_1 is the parent, and the children in order are m_1 , n_2 , and n_3 . Then m_2 will fire according to the third rule, $A \rightarrow aAA$, and this will bring into being two new A -nodes, say n_4 and n_5 . But firing this rule *requires* the presence of an A node, and of course this is where n_2 comes in. Applying $A \rightarrow aAA$ means that three children are added to n_2 . In order, they are m_2 , n_4 and n_5 . Later on, m_3 and m_4 become the children of n_4 and n_5 , respectively. These two actions happen independently. Also independent of this, and of the event between m_2 and n_2 , we have the final b -node, m_5 attaching itself as the child of n_3 . At this point, the parse tree is essentially complete, and we may calculate its yield as in the CFG case.

In the last paragraph we presented an informal account of the bottom-up derivation of s in G . Now we render derivation according to G as an evolving algebra. We use the same modes as before, and we only need to replace rule schemas (1) and

(2) by the two rule schemas which appear below. The first of these corresponds to a GNF rule of the form $X \rightarrow xY_1, Y_2, \dots, Y_k$ with $X \neq S$.

The second rule covers the case when X is the start symbol. The reason for the difference is that the start symbol is not required to be present for the rule to fire; indeed, it cannot be so required. Instead, for each rule of the form $S \rightarrow xY_1, Y_2, \dots, Y_k$ we have the second transition rule below.

- (5) If $\text{type}(n) = X$ and n is starting,
and $\text{type}(m_0) = x$ and m_0 is starting,
then n changes to working,
 m_0 changes to reporting, and $\text{output}(m_0) = \langle \text{type}(m_0) \rangle$,
 $\text{first-child}(n)$ changes to m_0 ,
and n has k new children m_1, \dots, m_k ,
of types Y_1, \dots, Y_k respectively,
and all these children of n change to starting.
- (6) If $\text{type}(m_0) = x$ and m_0 is starting,
then m_0 changes to reporting, and $\text{output}(m_0) = \langle \text{type}(m_0) \rangle$,
let n be a new node of type S , n changes to working,
 $\text{first-child}(n)$ changes to m_0 ,
and n has k additional children m_1, \dots, m_k ,
of types Y_1, \dots, Y_k respectively,
and all these children of n change to starting.

The collection of universes, functions, and transition rules gives an evolving algebra. We need to say what the permissible starting states are, and what overall constraints must hold in a run. Here we insist that in the starting state all nodes are in starting mode, and that there be no other structure present. The only overall condition on a run is that there should be exactly one application of rule (6).

At this point, the general definition of a run applies. We would have an exact analog of Proposition 1 for CFG's in GNF and the yields of runs of our machine.

The point of all this is that we have a different formalization of the concept of a CF derivation which is rigorous, easily understandable, and completely on a par with our earlier work on top-down CF production. That is, the definition of a run for the distributed evolving algebra just described is the same as for the earlier, top-down account EA account of CFG's.

3.2. BOTTOM-UP PRODUCTION

GNF is a convenient starting point to get a bottom-up interpretation of context-free production. However, it is not the complete story. As our rules show, the intuition formalized in GNF is partly top-down, and partly bottom-up. The top-down part comes in with the spawning of new children in (5) and (6). To get a completely bottom-up account, we want to replace both of those rules by ones which only spawn *parents* of pre-existing nodes.

Suppose we have a CF rule of the form $A \rightarrow aAbAB$. Here is how this would get turned into a rule of our evolving algebra:

- (7) If $\text{type}(n_1) = a$ and n_1 is starting,
 and $\text{type}(n_2) = A$ and n_2 is starting,
 and $\dots \text{type}(n_5) = B$ and n_5 is starting,
 then let m be a new node,
 let $n_1 = \text{first-child}(m)$, \dots , $n_5 = \text{fifth-child}(m)$,
 m changes to starting,
 and n_1, \dots, n_5 change to working.

As a special case of (7), we have the case of a production with just a terminal on the right; for example, $A \rightarrow a$. In this case, the rule would say that a node of type a can, at the appropriate moment, call introduce a parent node of type A .

The rest of the machinery would be the same as in the rendering of top-down production. That is, the modes reporting and dormant would work just as before. However, there need be no root in the bottom-up approach.

In contrast to the GNF rendering, this time we do not need a special rule for the productions involving the start symbol S . What we need is a way to insist that the overall yields be associated to a single node of type S . Perhaps the easiest way to do this to modify the rule (4) to say

- (8) If $\text{type}(n) = S$, and n is reporting,
 then n changes to dormant.

But this would also mean that we must stipulate that the yield of a run is the output of some node n of type S . (The production rules would take care of the requirement that all of the nodes in the original structure will be accessible from n . That is, from the grammar $S \rightarrow ab$, there will be no run having a node n of type S with the string $abab$ as output.)

We stress that in this rendering of CF production, we must change the definition of a run slightly. This is not a real problem as far as evolving algebra methodology is concerned.

4. Dynamic Models for Constraint Logic

We briefly review the definitions of constraint logic and its graph models. The material here is now quite standard; see, for example, Carpenter (C), Keller (K), Rounds (R), or Shieber (S).

Let C and L be arbitrary sets of *constants* and *labels*, respectively. We assume that L contains the symbols *first-child*, *second-child*, etc., which we used in the last section, and we also assume the label output belongs to L . Let L^* be the set of all finite sequences from L . \mathcal{L} is the smallest set containing the constant 1 (for truth), each $c \in C$, each pair $\langle x, y \rangle$ of elements of L^* (written $(x \approx y)$), and closed under the following formation rules: if $\phi, \psi \in \mathcal{L}$, then so are $\phi \wedge \psi$ and $\neg\phi$; if $\phi \in \mathcal{L}$ and $l \in L$, then $(l : \phi) \in \mathcal{L}$.

An \mathcal{L} -*structure* (over L and C) is a tuple $\mathcal{G} = \langle G, \delta, \alpha \rangle$ such that $\delta : G \times L \rightarrow G$ and $\alpha : G \rightarrow \text{Power}(C)$ are partial functions. Thinking of \mathcal{G} as a graph, we often refer to the members of G as *nodes*. We extend δ from a partial function on $G \times L$ to a partial function on $G \times L^*$, by $\delta(n, \epsilon) \simeq n$, and $\delta(n, lw) \simeq \delta(\delta(n, l), w)$.

For each \mathcal{G} , we define the *satisfaction relation* $\models_{\mathcal{G}}$ on $G \times \mathcal{L}$ by the following recursion:

$$\begin{array}{ll}
 n \models_{\mathcal{G}} 1 & \text{always} \\
 n \models_{\mathcal{G}} \phi \wedge \psi & \text{if } n \models_{\mathcal{G}} \phi \text{ and } n \models_{\mathcal{G}} \psi \\
 n \models_{\mathcal{G}} l : \phi & \text{if } \delta(n, l) \downarrow \text{ and } \delta(n, l) \models_{\mathcal{G}} \phi \\
 n \models_{\mathcal{G}} (x \approx y) & \text{if } \delta(n, x) \simeq \delta(n, y) \\
 n \models_{\mathcal{G}} c & \text{if } c \in \alpha(n)
 \end{array}$$

Given this logical system, a *grammar* may be taken to be a pair $Gr = \langle \mathcal{S}, s \rangle$, where \mathcal{S} is a finite set of sentences of \mathcal{L} , and $s \in \mathcal{S}$ is a *start rule*. We assume that all of the sentences are conjunctions of path equalities and sentences of the form $l_1 : \dots : \dots : l_l : c$, for $c \in A \cup \{1\}$. That is, every sentence in this basic language is equivalent to a conjunction of sentences of this form by rules of KR logic (see (R)).

The productions in a grammar are of two forms: phrasal productions and lexical productions. Phrasal productions are of the form $\langle a, \phi \rangle$, where $a \in N$ and $\phi \in \mathcal{L}$. Lexical productions are of the form $\langle w, \phi \rangle$, where w is a lexical item and $\phi \in \mathcal{L}$. The general form of a phrasal production may be taken in this treatment to be a conjunction of path equalities and atomic sentences. Later in this paper, we consider negation, and for that we find several alternatives. One could further extend our framework to cover disjunction, implication, regular path expressions, etc.

A structure \mathcal{G} satisfies Gr iff every node of G satisfies some sentence in Gr . Further, the *yield* of such a satisfying structure \mathcal{G} is the least function y so that for all nodes $n \in G$, if n satisfies a lexical rule with output $y(n)$ then $y(n)$ is the

lexical item associated with that rule, and if n satisfies a phrasal rule, then $y(n)$ is the concatenation of $y(\text{first-child}(n))$ with $y(\text{second-child}(n))$, etc.²

Now given these definitions, the next main concept is that of a *parse tree*. As it happens, Shieber does define bounded-depth parse trees as certain “models”, given by an inductive definition. This definition leans on the notion of a model, and (S) presents several classes of models before turning to the graph models (which have the nicest properties of all). We restrict attention to the graph models themselves.

It might be interesting to note that Shieber feels that “The definition of parse tree can play the role that derivation does in a context-free grammar.” This is not surprising, since it is an inductive definition with roughly the same flavor as that of a parse tree. Our purpose here is to present other formulations of this, one closely connected to our renderings of CFG’s.

The basic idea is to use *structure sharing* in the transition rules. This means that rule schema (1), which demanded *new* children, must be replaced by a schema which allows for the children to be already present. (This feature has already been used in our work on bottom-up rendering.)

A top-down derivation, then, will work much as in the CF case, except that structure sharing is permitted when nodes are in starting mode. To enforce the path equalities, we need only check them at the time of reporting. (This is because all of our models are non-deterministic. More deterministic models are of course needed in parsing, and for that real ingenuity is needed. There would be no problem in expressing such an algorithm as a evolving algebra. But doing so would mean working at a lower level of abstraction and hence not illustrate the power of the approach to formalize intuitive concepts.)

To see how a set of transition rules could be constructed, consider the rule

$$c \wedge (\text{first-child} : l_1 : l_2 \approx l_3) \wedge \text{second-child} : l_2 : a \quad (\rho)$$

Here l , m , and n belong to L , as do *first-child* and *second-child*. It is natural in an evolving algebra to associate functions to the elements of L . As we have already been doing with *first-child* and *second-child*, we will have functions l from nodes to nodes. All of these functions are dynamic.

It is natural to associate unary functions from nodes to truth values. Accordingly, we write $c?(n) \simeq 1$ to say that node n belongs to the extension of c . (In other words, $c \in \alpha(n)$.) Of course, c too is dynamic, and this means that in the course of a derivation (i.e., a run of our structure), it will be possible to change c . In the evolving algebra for constraint grammars, this only happens when certain nodes are created. But it is conceivable that grammar formalisms might want to change whether $c(n)$ holds or not many times in the course of a run.

² This is our formulation of the notion of yield. Like all inductively defined sets, it can be understood iteratively, and this is incorporated in our rules below. The definition of Shieber (S), p. 55 is an alternative way of understanding the recursion.

Another detail we need is that each grammar rule is associated with a type. (This contrasts with the CFG case, where types corresponded to the terminal and non-terminal symbols of the grammar.) In a constraint grammar, we have no such easy access to type information. In fact, our evolving algebra will simply guess a type ρ (that is, a rule of the grammar) for each node n when n is created, and then when n is in reporting mode, we check whether the conditions expressed in ρ are fulfilled.

We can now list our transition rules. First, the analog of (1) is the following rule which introduces nodes into a structure:

- (7) If $\text{type}(n) = \rho$ and n is starting,
 then n changes to working, n has 2 new children m_1 and m_2 ,
 and there also exist nodes m_3, m_4 , and m_5 ,
 (none of these necessarily new or distinct from others),
 $\text{first-child}(n)$ changes to m_1 , $l_1(m_1)$ changes to m_3 ,
 $l_2(m_3)$ changes to m_4 , $l_3(n)$ changes to m_4 ,
 $\text{second-child}(n)$ changes to m_2 , $l_2(m_2)$ changes to m_5 ,
 $a?(m_5)$ changes to 1, $c?(n)$ changes to 1,
 and all these new nodes change to starting,
 and types are chosen for them.

The nodes mentioned in this rule are just those which rule (ρ) demands. In the top-down view of this section, it may be that these nodes are lacking when the rule is applied. In that case, they would be introduced into the structure. The parenthetical remark on the introduced nodes means that the nodes need not be new; they can be nodes which already exist in the structure. This is the way that our rendering takes care of structure sharing: by permitting introduced nodes to be non-distinct, or to coincide with pre-existing nodes.

Of course, (7) is really a schema, and it encodes the rules of the grammar in a uniform way.

For lexical rules, we have the following schema:

- (8) If $\text{type}(n) = \rho$ and n is starting, and $\text{type}(n) = x$,
 then n changes to reporting,
 and $\text{output}(n) = \langle \text{type}(n) \rangle$.

On the other hand, for other rules we calculate the yield in working mode exactly as in (3).

- (9) If $\text{type}(n) = \rho$, n is working,
 $c?(n) \simeq 1$, $\text{first-child}(l_1(l_2(n))) = l_3(n)$,
 $\text{second-child}(l_2(n)) \downarrow$,
and all numbered children of n are reporting,
then $\text{output}(n) = \text{concatenate}_{n \rightarrow m}(\text{output}(m))$,
all children m of n change to dormant,
and n changes to reporting.

In other words, we use (ρ) as a node-admissibility condition, to be checked at the time of reporting. By the “numbered children” of a node n , we mean $\text{first-child}(n)$, $\text{second-child}(n)$, etc. Or course, this rule is calculating the yield according to the fixed point equation.

Finally, we have an analog of (4), where ρ is the start production.

- (10) If $\text{type}(n) = \rho$,
and n is reporting,
then n changes to dormant.

Unlike the schemata above, this is a single rule, one per grammar.

We also must stipulate that there be only one application of this rule in each run. This is the analog of insisting that in a parse tree for a CFG there is exactly one application of a rule whose left-hand side is S .

This concludes our description of an evolving algebra whose runs correspond to the top-down parses of a grammar which is given by formulas of KR logic.

4.1. BOTTOM-UP PARSSES

It is also possible to obtain bottom-up renderings of constraint grammars. Our ideas are a combination of those from Section 3.2 with the work of the last section. We again discuss a simple example (ρ) . The main new operations in the bottom-up approach are found in the following fundamental rule:

- (11) If n and m are unifiable,
then $\text{identify}(n, m)$.

Even before we define the terms, one can well imagine that a rule like (11) would be at the heart of bottom-up renderings of constraint formalisms. The predicate “unifiable” holds of two nodes n and m just in case none of the partial functions defined on n and m disagree. (For functions of more than one argument, we would require that interchanging n and m in any argument tuple does not change any

defined value.) We stress that this predicate, like all dynamic predicates, will be applied to nodes at certain places in their evolution. We do not expect (or desire) that this predicate be monotone. Also identify means the obvious thing: merge the two nodes to one, and define the functions on them by taking cases.

Note also that the dynamic apparatus having to do with unification is treated abstractly. We are working at a higher level of abstraction than if we had exact algorithms for unification. Our starting point is the observation that for most work on constraint formalisms, the exact details of implementation are not needed for the workings of grammar. Therefore, in order to choose a formalism at the right level of abstraction, we would be wise to avoid such details.

Having (11), we can now go on to discuss the the remainder of the rules having to do with (ρ) from the last section. Rule (11) is in some sense a dual to (7); the former allows merging of nodes at any time, while the latter only allows identification at the time of creation. As with the bottom-up rendering of CF production, the idea is to turn (ρ) into a rule (ρ') which *demand*s the presence of the intermediate nodes given by (7). The antecedent of (ρ') would contain the path equalities of (ρ) and the statement that the modes are all working. Then the conclusion would be that a new node is created, of the appropriate type, and all the modes are set as they should be.

We have found that figuring out the details of various evolving algebra renderings often brings to light the sensitive features of a formalism. Also, when formalisms can be defined in different but equivalent ways (as it often happens), this is often a sign that different dynamic intuitions are at work. So the resulting evolving algebras would be different.

5. Negation

As we have noted in Section 1.2, the issue of negation in feature formalisms is complicated, in part because the concept rests on dynamic ideas which the formalism prefers to avoid. At this point, we have sketched out two possible evolving algebra renderings of production via constraint grammars. In our terms, we can locate a number of possible sources of negative information, and also suggest treatments for these, or connections to the literature.

1. Negations of atomic conditions. The assertion $n \not\approx c$ can be understood classically, at any point in a run. Alternately, it can be understood using some variety three-valued semantics, as is done by Dawar and Vijay-Shankar (DVS).
2. Negations of path equalities. We have the same options for interpreting assertions of the form $n \not\approx (x \approx y)$. In addition, we can express that $\delta(n, x)$ and $\delta(n, y)$ are not unifiable. Further, one might take a different sort of prescriptive

approach and obtain a definition that forced n and m never to be unifiable in the future, even if it happened that they were unifiable when the rule applied. (To do this, we could simply add a new function symbol and force a difference. Alternatively, we could encode the condition that $n \not\approx m$ in the transition rules, for example at the time of reporting.)

Further, we can also modify our work to handle disjunctions, in the same general ways. In addition, we can envision options reminiscent of temporal logic. For example, we could have $\varphi U \psi$, φ holds until ψ does. This kind of operator arises in temporal specifications, and we suspect that it could be useful in syntax. For more details of how temporal operators could work in evolving algebras, see (GM; B). Along with disjunction and negation, we have a number of possibilities for implication.

There is also no problem to extend our models of feature structures to cover the “regular path” versions of the logic, such as \mathcal{L}_{EKR1} , (see Moshier (M)) or Regular Rounds-Kasper logic (see Keller (K)). Both of these have operators which allow for non-local properties to be checked. However, since the descriptions are very simple, they can be directly implemented in our system. Alternatively, their semantics can be worked in as a primitive feature of the system.

6. Summary Points on Static and Dynamic Linguistic Models

Our aim in this paper has been to show that dynamic renderings are available for a well-known and well-used class of static grammar formalisms. While the constraint grammar formalisms may be studied and used statically (simply as logics), we feel that a more dynamic view might also be useful. One reason is that unification is often supplemented with various features: negative constraints, structure modification, defaults, etc. From a logical point of view, these are often problematic. In effect, they give rise to *non-monotonic* phenomena. This is unfortunate, since it means that the use of a computational system will depend on technical concerns which were far from the original grammatical motivation. And the order in which operations are carried out could become significant. For example, is a given constraint checked after every unification operation or is it checked once at the end of all unification operations?

In other words, an inescapably dynamic quality has crept into what was once a rather static picture. The real claim behind declarative formalisms is not that one can simply formulate such a system, but that it could also have a nice computational interpretation. To the extent that declarative formalisms do not have a nice computational interpretation, and hence their computational use requires significant enhancements, alterations, and innovations, the attractiveness or usefulness of declarative approaches is compromised.

We believe that some sort of constrained procedural formalism would allow one to come to terms with these problems.

Assuming this point, we would like to detail three further steps in the program of using dynamic ideas directly. The most pressing technical problem is to limit the specification language. The evolving algebra machinery in its full strength is too expressive in that one can easily represent unsolvable problems in it. However, what we are using is just a fragment (our structures have no memory, for example). After working through our examples in this paper and in (JM; MJ), we are in a position to study questions of expressive power. The main goal would be an efficient decidability result for this fragment. Another project would be to develop some tools to make it easier to actually specify grammar formalisms. We are well aware that the evolving algebra framework and notation as it stands might not be optimized for that application. Finally, it would be most interesting to propose uses of this kind of formalism that went beyond what is available in constraint formalisms. One candidate area is real-time sentence processing: since evolving algebra methods are being used in specification and verification of real-time problems, there is a body of conceptual work to draw on.

References

- E. Börger. Annotated bibliography on evolving algebras. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994. Paper available by anonymous ftp from: apollo.di.unipi.it in the directory pub/Papers/boerger.
- B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science 32. Cambridge University Press, 1992.
- A. Dawar and K. Vijay-Shankar. A three-valued interpretation of negation in feature structure descriptions. *Computational Linguistics*, 16(1):11–21, 1990.
- Y. Gurevich and L.S. Moss. Algebraic operational semantics and occam. In *Proceedings of 3rd Workshop on Computer Science Logic*, Springer-Verlag LNCS volume 440, 1990.
- Y. Gurevich. Evolving algebras: A tutorial introduction. *Bulletin of the European Association for Theoretical Computer Science*, 43:264–286, 1991.
- Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.
- D. E. Johnson and L. S. Moss. Grammar formalisms viewed as evolving algebras. *Linguistics and Philosophy*, to appear, 1995.
- M. Johnson. Attribute-Value Logic and the Theory of Grammar. CSLI Lecture Notes Number 16, CSLI, 1988.
- B. Keller. *Feature Logics, Infinitary Descriptions, and Grammar*. CSLI Lecture Notes Number 44, CSLI, 1993.
- M. A. Moshier. *Extensions to Unification Grammar for the Description of Programming Languages*. PhD thesis, University of Michigan, 1988.
- M. A. Moshier and W. C. Rounds. A logic for partially specified data structures. In *Proceedings of 14th ACM Symposium on Principles of Programming Languages* 156–167.
- L. S. Moss and D. E. Johnson. Evolving algebras and mathematical models of language. In *Applied Logic: How, What, and Why?*, 143–176. Kluwer Academic Publishers, Dordrecht, to appear 1995.
- W. C. Rounds. Feature logic. In J. van Benthem and A. ter Meulen (eds.), *Handbook of Logic and Language*. North Holland, Amsterdam, to appear.
- S. M. Shieber. *Constraint-Based Grammar Formalisms*. Bradford Books, MIT Press, Cambridge, Massachusetts 1992.