HAL
open science

Attribute grammars and automatic complexity analysis

Marni Mishna

To cite this version:

Marni Mishna. Attribute grammars and automatic complexity analysis. [Research Report] RR-4021, INRIA. 2000. inria-00072620

HAL Id: inria-00072620

https://inria.hal.science/inria-00072620

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Attribute grammars and automatic complexity analysis

Marni Mishna

N ° 4021
Octobre 2000

THÈME 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4021--FR+ENG

Attribute grammars and automatic complexity analysis

Marni Mishna

Thème 2 — Génie logiciel
et calcul symbolique
Projet Algo

Rapport de recherche n° 4021 — Octobre 2000 — 18 pages

Abstract: Decomposable combinatorial structures have been well studied and a systematic manner for determining generating function equations is well known. Attribute grammars have enhanced the study of context-free grammars by giving meaning to constructions. Delest and Fédou [2] showed that attribute grammars extend to combinatorial structures, with applications to random generation. In a similar way, we show attribute grammars can be defined for the family of decomposable structures and yield multivariate generating function equations. From there, averages and higher moments are easily accessible. This idea unifies previous approaches to the analysis of parameters of data-structures.

Key-words: attribute grammars, decomposable structures, algorithm analysis, generating functions

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33) 01 39 63 55 11 – Télécopie : (33) 01 39 63 53 30

Grammaires attribuées et analyse automatique de complexité

Résumé : Les structures decomposables sont bien etudiées, et une méthode systematique permettant d'obtenir des équations de fonctions génératrices est bien connue. Les grammaires attribuées permettent de donner une signification aux constructions des grammaires context-free. Delest et Fédou ont montré que les grammaires attribuées s'étendent à certaines structures combinatoires, avec des applications à la génération aléatoire. De manière semblable, nous montrons que des grammaires attribuées peuvent être definies pour la famille des structures decomposables, ce qui donne lieu à des equations de fonctions génératrices multivariées. De là, moyennes et autres moments sont aisément accessibles. Cett idée unifie les approches précédentes à l'analyse de paramètres des structures de données.

Mots-clé : grammaires attribuées, structures decomposables, analyse d'algorithmes, fonctions génératrices

ATTRIBUTE GRAMMARS AND AUTOMATIC COMPLEXITY ANALYSIS

MARNI MISHNA

Abstract. Decomposable combinatorial structures have been well studied and a systematic manner for determining generating function equations is well known. Attribute grammars have enhanced the study of context-free grammars by giving meaning to constructions. Delest and Fédou [2] showed that attribute grammars extend to combinatorial structures, with applications to random generation. In a similar way, we show attribute grammars can be defined for the family of decomposable structures and yield multivariate generating function equations. From there, averages and higher moments are easily accessible. This idea unifies previous approaches to the analysis of parameters of data-structures.

1. Attribute Grammars

Attribute grammars were initially developed to give semantic interpretation to context-free grammars. Since their introduction by Knuth [11] they have made themselves useful in compilers and as a means of program description. The attribute value, or meaning, of a word can be viewed as the output of a function or program acting on the word. M. P. Delest and J.-M. Fédou [2] illustrated how attribute grammars are useful to the combinatorialist. This concept was further developed by I. Dutour and Fédou [3] who described a systematic approach to enumeration and random generation of object grammars, using attribute grammars to describe properties of the structures.

This work both tailors and extends the latter approach to a particular family of grammars, decomposable structures, and describes an implementation incorporated into the Maple package `combstruct`. Object grammars do not include constructors like set and sequence, and hence an extension is necessary. The notation here is different, but the final q-series functional equations are similar.

From the point of view of decomposable structures, certainly values have been given to structures before under the names additive attributes and recursive parameters [1, 6]. However, this work proposes a framework and makes explicit all the general enumerative consequences.

In this document, the suitable attributes are defined and further specified in the first two sections and the generating function consequences are considered in the third section, including a treatment of pathlength. The fourth section introduces the Maple implementation and is followed by a comparison with the LUO system for automatic average case algorithm analysis developed by the Algorithms Project of INRIA [5, 12]. There is a brief consideration of random generation in section seven.

1.1. **Grammars and Specifications.** The structures considered here are built from basic objects of weight 1, an *Atom* (commonly expressed as Z), and of weight 0, an ϵ. The constructors are disjoint union $|$, cartesian product $\cdot$, sequence *, Set(), MultiSet() and Cycle(). These latter constructors form, respectively, sequences, sets without or with duplication and cycles of their arguments. They shall collectively be referred to as the iterative constructors. They admit simple cardinality restrictions: card(inality) $\leq k$, card $= k$ and card $\geq k$, for any non-negative integer k.

Definition. Let $T = (T_1, T_2, \ldots, T_m)$ be an m-tuple of families of combinatorial objects. A *specification* is a rule $T_i = \Phi_i(T_1, T_2, \ldots, T_m)$ where Φ_i is ϵ, an atom or one of the aforementioned constructors. A notion of derivation trees similar to context free grammar exists. With this view, denote T_j used to define T a *descendant* of T_i and similarly T_i is an *ancestor* of each T_j. A structure which can be described in this way is *decomposable*, and a set of specifications is a *grammar*. If the atoms are labelled with the numeric labels 1 to the size, the grammar is labelled, otherwise it is unlabelled. A grammar is *well defined* if each specification generates a finite number of objects of each size, and uniquely so. The *size* of an object a, or $|a|$ is the number of atoms.

A wide variety of structures are decomposable. For example: plane binary trees ($T = \epsilon | Z \cdot T \cdot T$), non-plane trees ($T = Z \cdot \text{MultiSet}(T)$), permutations ($P = \text{MultiSet}(\text{Cycle}(Z, card > 0))$) and all structures definable with context-free grammars.

Definition. An *attribute* is a function which assigns a value to a structure. For example, the number of atoms is an attribute. An *attribute grammar* is a three-tuple (G, A, P) where G is a grammar, A is a set of attributes and P is a set of rules defining the value of attributes in A for structures in G. An attribute of a structure is *synthesized* if it is strictly a function of descendants and it is *inherited* if it is a function of its ancestors.If the set of defining rules are formulated in such a way that all attributes can always be defined at each node of the derivation tree of each structure, then the grammar is *well defined*. Knuth [11] described an algorithm for determining this property of attribute grammars. If G is well defined, and A contains only synthesized attributes defined (possibly trivially) for each structure, the attribute grammar will be well defined.

The iterative constructors have at least two meaningful possibilities for an attribute. It may be defined as the sum of the values of the elements in the object, or as the value of a chosen element. The first is an *iterative* attribute and the second a *selective* attribute.

1.2. **Example: Binary Trees.** A binary tree is recursively defined as empty, or a node and two children trees, $T = \epsilon \mid N \cdot T \cdot T, N = Atom$. There are many recursively defined properties of trees that one may wish to study, such as the number of internal nodes (numint), or internal pathlength (ipl), the sum of distances from every node to the root. These can all be described by attributes, as is done in Figure 1.

1.3. **Linear Attributes.** A synthesized attribute of a structure defined by $T = \Phi(B_1, \ldots, B_k)$ is *linear* if it is a linear function of the attributes of the descendants. That is, it can be expressed as some function linear in $F_j(B_i)$ where $\{F_j\}$ is a set of attributes. An attribute grammar which

Specification	Attribute definition		
$T = \epsilon \mid N \cdot T \cdot T$	$\text{size}(T)$	$=$	$0 \mid \text{size}(N) + \text{size}(T_1) + \text{size}(T_2)$
$(T_0 = \epsilon \mid N \cdot T_1 \cdot T_2)$	$\text{ipl}(T)$	$=$	$0 \mid \text{size}(T_1) + \text{size}(T_2) + \text{ipl}(T_1) + \text{ipl}(T_2)$
	$\text{numint}(T)$	$=$	$0 \mid 1 + \text{numint}(T_1) + \text{numint}(T_2)$
$N = Atom$	$\text{size}(N)$	$=$	1

FIGURE 1. Binary Trees and some attributes

consists only of linear attributes is aptly named a *linear attribute grammar*. In the example each of the attributes is linear.

2. Linear Attribute Specifications

The initial goal is to extract enumerative information about decomposable structures and their properties. In order to do so, a general description of suitable attributes is required. First consider linear, synthesized, and then other possibilities.

Fédou and Delest concentrated on a class of attributes called q-linear. On context-free grammars this is equivalent to synthesized, linear attributes. In the work of Dutour and Fédou, the q-linear attributes are also synthetic and linear, but over a wider range of structures. However, this wider range of structures does not include constructors such as set, and cycle. There are meaningful definitions of linear, synthetic attributes, as well as enumeration consequences occur for these constructors, and so they are treated here.

Consider the explicit definition of linear attributes. Let $AG = (G, \{F_i\}, P)$ be an attribute grammar. Given a specification for a structure, the attribute depends on the operator. For example, the attribute value of a member in a union depends on the value of the union element from which it was derived. So, if $C = A \mid B$, the set of all possible values for $F_i(C)$ is a disjoint union of linear combinations of attribute values of elements of A and linear combinations of attribute values of elements of B. This is written

$$F_i(C) = \gamma_i + \sum_{j=1}^{k} \alpha_{ij} F_j(A) \quad \Bigg| \quad \delta_i + \sum_{j=1}^{k} \beta_{ij} F_j(B),$$

where the $\gamma_i, \alpha_{ij}, \delta_i, \beta_{ij}$ are all integer constants.

Similarly, for products, a valid attribute operator for $C = A \cdot B$ is a linear combination of attributes of A and B. Thus, the range of attributes of $F_i(C)$ is

$$F_i(C) = \gamma_i + \sum_{j=1}^{k} \alpha_{ij} F_j(A) + \sum_{j=1}^{k} \beta_{ij} F_j(B).$$

This will also be written

$$F_i(C) = \gamma_i + \cdot(\sum_{j=1}^{k} \alpha_{ij} F_j(A) + \sum_{j=1}^{k} \beta_{ij} F_j(B)),$$

with a dot to facilitate notation readability.

Denote by Φ the iterative attribute which sums over all sub-elements of type Φ. That is, The iterative operator $A = \Phi(B)$ has a general form for a linear attribute of

$$F_i(A) = \Phi(\delta_i + \sum_{j=1}^{k} \alpha_{ij} F_j(B)) + \gamma_i.$$

In the case of sets, for example, this would have the interpretation that for each set $a \in A$,

$$F_i(a) = \gamma_i + \sum_{b \in a} \left(\delta_i + \sum_{j=1}^{k} \alpha_{ij} F_j(b)) \right).$$

The range of values of $F_i(A)$ is the set formed by considering all structures of type A and summing the value of the linear function over them.

With this notation the general specification of a structure is

$$A = \Phi_1(B_1^{(1)}, \ldots, B_{k_1}^{(1)}) | \ldots | \Phi_n(B_1^{(n)}, \ldots, B_{k_n}^{(n)}),$$

and an attribute of this specification has the general form:

$$F_i(A) = \bigcup_{m=1}^{n} \Phi_m(\delta_i^m + \sum_j \sum_{k=1}^{k_m} \alpha_{ij}^{(m,k)} F_j(B_k^{(m)})) + \gamma_i.$$

In this general expression Φ_i is a non union constructor and indexed, lower case greek letters indicate integer constants.

2.1. **Self-Referential Attributes.** It is notationally convenient at times for an attribute of a structure to be expressed in terms of another attribute of the structure itself. For example, the internal pathlength of the binary tree in Figure 1 could be rewritten with the rule $\mathrm{ipl}(T) = \mathrm{ipl}(T_1) + \mathrm{ipl}(T_2) + \mathrm{size}(T) - 1$. The reference to the size of the whole tree, $\mathrm{size}(T)$, is the self-reference. Even though T_1 and T_2 are both structures of type T, the attribute values $\mathrm{ipl}(T_1)$ and $\mathrm{ipl}(T_2)$ are not considered self-referential since T_1 and T_2 are descendants.

Grammars which are well defined with all self-references occurring outside of iterative attributes, have no greater expressive power than those without self-references since the attributes can be expressed strictly in terms of its descendants. An attribute grammar defined without any self-referential attributes is in *standard form.* The idea is akin to Knuth's observation that all inherited attributes can be rewritten as synthesized attributes.

Proposition 1. *Any linear, self-referential attribute in a well defined attribute grammar can be rewritten as a linear attribute in standard form.*

Proof. Let A be some decomposable structure and $\mathbf{F} = \{F_i\}_{i=1}^n$ be a set of attributes defined for A. Since the attribute grammar is well defined they all have values on A. Thus, there exists some total ordering of evaluation of $\mathbf{F}$, say $F_1', F_2', \ldots, F_n'$. The first in this total ordering must be either constant or depend on values of descendants. In either case all references to $F_1'(A)$ can be replaced

by its value: either a constant or the value stated in terms of the descendants of A. This process can be iterated through the partial order until all attribute values of A are expressed in terms of descendants.

□

3. Generating Functions

Information about combinatorial structures can be conveniently encoded into ordinary and exponential generating functions. The name of the class doubles as the name of the generating function. For example, in the unlabelled case, $A(z) = \sum_{a \in A} z^{|a|} = \sum_{n>0} a_n z^n$ where a_n is the number of structures of size n. The labelled case is written $A(z) = \sum_{n>0} \frac{a_n z^n}{n!}$. A decomposable structure's generating function satisfies relations governed by the following folk theorem.

Theorem 2 (Folk theorem of combinatorial analysis). *Given a specification for a class C, a set of equations for the corresponding generating functions is obtained automatically by the following translation rules. (The function ϕ is Euler's totient function.)*

Specification	$C(z)$	*Universe*
$C = A \mid B$	$A(z) + B(z)$	*either*
$C = A \cdot B$	$A(z)B(z)$	*either*
$C = A^*$	$\frac{1}{1-A(z)}$	*either*
$C = \mathrm{Set}(A)$	$\exp(A(z))$	*labelled*
$C = \mathrm{Cycle}(A)$	$\ln(\frac{1}{1-A(z)})$	*labelled*
$C = \mathrm{Set}(A)$	$\exp(\sum_{k>0}(-1)^{k-1}\frac{A(z^k)}{k})$	*unlabelled*
$C = \mathrm{Cycle}(A)$	$\sum_{k>0}\phi(k)/k \ln(\frac{1}{1-A(z^k)})$	*unlabelled*
$C = \mathrm{Multiset}(A)$	$\exp(\sum_{k>0}\frac{A(z^k)}{k})$	*unlabelled*

For specification $\Phi(B_1, \ldots, B_k)$ let $\mathcal{G}_\Phi(B_1, \ldots, B_k)$ denote the corresponding generating functions translation.

Multivariate generating functions are used to keep track of multiple properties; here, the value of attributes. The attribute generating function in an unlabelled universe with attributes $\{F_i\}$ is $A(z_1, z_2, ..., z_k) = \sum_{a \in A} z_1^{|a|} z_2^{F_2(a)} \cdots z_k^{F_k(a)}$, and in a labelled universe each term is divided by $|a|!$. By convention, the first parameter z_1 tracks size size. This uses the fact that the grammar is well defined to ensure that the equations are meaningful. They satisfy very similar relationships as those in the folk theorem. This has been observed before with respect to additive attributes, and the relationships have been used before. Notice, for example, the lone attribute size reduces the problem to the univariate case.

To describe this action, let $\alpha = [\alpha_{ij}]_{k \times k}$, and $\gamma = [\gamma_i]_{1 \times k}$ be matrices over some ring containing the range of values of attributes. If $\mathbf{z} = (z_1, \ldots, z_k)$, then denote $\mathbf{z}^\alpha = (z_1^{\alpha_{11}} \cdots z_k^{\alpha_{k1}}, \ldots, z_1^{\alpha_{1k}} \cdots z_k^{\alpha_{kk}})$ and $\mathbf{z}^\gamma = z_1^{\gamma_1} \cdots z_k^{\gamma_k}$.

Theorem 3. *Given the grammar specification*

$$A = \Phi_1(B_1^{(1)}, \ldots, B_{k_1}^{(1)}) \mid \ldots \mid \Phi_n(B_1^{(n)}, \ldots, B_{k_n}^{(n)})$$

where each Φ_i is a grammar constructor, and given the set of attributes

$$F_i(A) = \bigcup_{m=1}^{n} \Phi_m(\delta_i^{(m)} + \sum_j \sum_{k=1}^{k_m} \alpha_{ij}^{(m,k)} F_j(B_k^{(m)})) + \gamma_i^{(m)}$$

the multivariate generating function $A(\mathbf{z})$ satisfies

$$A(\mathbf{z}) = \sum_m \mathbf{z}^{\gamma^{(m)}} \mathcal{G}_{\Phi_m}(\mathbf{z}^{\delta^{(m)}} B_1^{(m)}(\mathbf{z}^{\alpha^{(m,1)}}), \ldots, \mathbf{z}^{\delta^{(m)}} B_{k_m}^{(m)}(\mathbf{z}^{\alpha^{(m,k_m)}})),$$

where $\mathcal{G}_{\Phi_m}$ is the generating function transformation based on the Theorem 2.

Proof. The result can be directly proved by a case by case analysis of each structure type. The general structure for such proofs is illustrated with the following proof of the unlabelled set transformation.

Here the statement is simplified to $A(\mathbf{z}) = \mathbf{z}^\gamma \mathcal{G}_{\mathrm{Set}}(\mathbf{z}^\delta B(\mathbf{z}^\alpha))$.

$$\begin{aligned}
A(\mathbf{z}) &= \sum_{a \in A} z_1^{F_1(a)} z_2^{F_2(a)} \ldots z_k^{F_k(a)} \\
&= \sum_{a \in A} \prod_{i=1}^{k} z_i^{\gamma_i + \sum_{b \in a} \delta_i + \sum_{j=1}^{k} \alpha_{ij} F_j(b)} \\
&= \mathbf{z}^\gamma \prod_{b \in B} (1 + \mathbf{z}^\delta \prod_{i=1}^{k} z_i^{\sum_{j=1}^{k} \alpha_{ij} F_j(b)}) \\
&= \mathbf{z}^\gamma \prod_{(e_1, e_2, \ldots, e_k)} (1 + \mathbf{z}^\delta \prod_{j=1}^{k} (\prod_{i=1}^{k} z_i^{\alpha_{ij}})^{e_j})^{N_B(e_1, e_2, \ldots, e_k)} \\
&= \mathbf{z}^\gamma \exp(\sum_{(e_1, \ldots, e_k)} N_B(e_1, \ldots, e_k) \ln(1 + \mathbf{z}^\delta \prod_{j=1}^{k} (\prod_{i=1}^{k} z_j^{\alpha_{ij}})^{e_j})) \\
&= \mathbf{z}^\gamma \exp(\sum_{k \geq 1} (-1)^{k+1} \frac{\mathbf{z}^{k\delta} B(\mathbf{z}^{k\alpha})}{k}) \\
&= \mathbf{z}^\gamma \mathcal{G}_{\mathrm{Set}}(\mathbf{z}^\delta B(\mathbf{z}^\alpha))
\end{aligned}$$

Here $N_B(e_1, ..., e_k)$ is the number of $b \in B$ that satisfy $F_i(b) = e_i$ for $i = 1, \ldots, k$. This is precisely the coefficient of $z_1^{e_1} \cdots z_k^{e_k}$ in the generating function of B.

□

3.1. **Example: Pathlength.** The generating function equations for binary tree with path length are consequences of the definitions. By applying the attribute grammar version of the folk theorem one has the equation $T(z,u) = 1+z\cdot T(zu,u)^2$. Pathlength of labelled non-plane trees $P = Z\cdot\mathrm{Set}(P)$ is similar with $\mathrm{ipl}(P) = \mathrm{Set}(\ \mathrm{size}(P) + \mathrm{ipl}(P))$ leading to the equation $P(z,u) = z\exp(P(zu,u))$. These well known equations admit several interesting calculations, presented in the next section with the help of Maple.

Attributes can also be used to count occurrences of substructures. For example, the number of cycles in a permutation, described by the construction $P = \mathrm{Set}(\mathrm{Cycle}(Z))$ (labelled) is determined by the attribute grammar, $\{\mathrm{num}(P) = \mathrm{Set}(1)\}$ which yield $P(z,u) = z\exp(u\ln((1-z)^{-1}))$.

3.2. **Additional Attribute Types.** A selective attribute determines the value of the set, cycle or sequence based on some representative. The attributes described so far can yield a constant selective attribute. Instead, consider an attribute of a structure which is a linear function of an attribute value of a randomly chosen element. In the labelled case one can describe the probability generating function where a variable marks the expected value of the set, cycle or sequence. Assume that the value of the empty set is 0.

To illustrate the generating function transformations, consider the simplest case, $A = \Phi(B)$ with attribute function $F(A) = \mathrm{Select}(\beta F(B) + \alpha)$. This has the interpretation that given an iterative structure a of type A, the value is $\beta F(b) + \alpha$ where $b \in B$ is chosen at random from the elements of a. The generating function is interpreted with $[z^n u^k]A(z,u)$ as the number of elements of type A and size n with the expected value of F equal to k. In the case of sequences, this value for a given sequence is given by summing over all values in the sequence and then dividing by the size of the sequence. The labelled set and cycle constructions can be similarly determined. Unlabelled cycles also have easy access in the generating function to the size of the cycle and hence a simple description of this value is also possible. Table 1 summarizes these values.

$A = B^*$	$A(z,u) = u^\alpha \sum_{k>0} B^k(z, u^{\beta/k})$
$A = Set(B)$	$A(z,u) = u^\alpha \sum_{k>0} B^k(z, u^{\beta/k})/k!$
$A = Cycle(B)$	$A(z,u) = u^\alpha \sum_{k>0} B^k(z, u^{\beta/k})/k$
$A = Cycle(B)$(unlabelled)	$A(z,u) = u^\alpha \sum_{k>0} \phi(k)/k \sum_{j\geq 1} B(z^k, u^{\beta/j})^j/j$

TABLE 1. Selective Attribute Generating functions for $\beta F(b) + \alpha$

Each of these are obtained by expanding the generating function equations and setting u to the power one over the cardinality of the object.

4. Maple Implementation

The main Maple package for combinatorial structure manipulation (combstruct) provides a template for the definition of attribute grammars.[1]

4.1. **Grammar Syntax.** The attribute grammar syntax mirrors closely that of the `combstruct` grammars. This allows for maximal clarity and eliminates ambiguity. Now, Union, Prod, and the other `combstruct` operators act as functions to describe attributes. Further, self-referential attributes are permitted. Thus, if a grammar contains $A = B \mid C$, the attribute $F(A)$ in a corresponding attribute grammar is described

```
F(A)=Union(b_1*F_1(B)+...+b_k*F_k(B),c_1*F_1(C)+...+c_k*F_k(C))
                + a_1*F_1(A)+...+a_k*F_k(A)+a_0,
```

where the a_i, b_i and c_i are either integers or constants (names). Products are similar. Thus the set of valid attributes of `A=Set(B)` are written with syntax as follows,

```
F(A)=Set(b_1*F_1(B)+...+b_k*F_k(B)+b_0)+ a_1*F_1(A)+...+a_k*F_k(A)+a_0.
```

In the `combstruct` lexicon the constructor `Set` refers to multisets.

Since `combstruct` verifies that the original grammar is well defined, and since only linear, synthetic attributes are allowed, all attribute grammars in standard form are well defined. In the case of self-referential attributes, the program will ensure that the grammar is well defined and expand the self-references.

Those rules which are not explicitly defined are assigned recursive defaults. For example, together the specification $A = \text{Union}(B, C)$ and the attribute operator F invoke a default rule of $F(A) = \text{Union}(F(A), F(B))$. The default value of an atom is 1 and of an epsilon is 0. Thus, the default attribute is size.

4.2. **Example: Pathlength.** Using the generating function relationships for pathlength, and the existing Maple tools for manipulating generating functions, statistics for pathlength such as average, variance, and other moments are well within reach.

The binary tree grammar and the pathlength attribute grammar are expressed in Maple by

```
> sys := {B = Union(Epsilon, Prod(Node, B, B)), Node = Atom}:
> att:= {ipl(B) = Union(0, Prod(0, ipl(B)+size(B), ipl(B)+size(B)))}:
```

To determine the defining generating function equations, use the command `agfeqns`. The variable z marks the *size*, a default attribute.

```
> eqns:=  agfeqns(sys,att, unlabelled, [[u, ipl]], z);
```

$$\left\{B(z,u) = 1 + z\left(B(zu,u)\right)^2, Node(z,u) = z\right\}$$

The expression for average pathlength can be calculated from the number of trees on n nodes and the sum of all pathlengths on n nodes, or $[z^n]\frac{\partial}{\partial u}B(z,u) \mid_{u=1}$. Maple can solve for both of

[1] A full description of the Maple library `algolib` which includes `combstruct`, and other generating function tools, is available at `http://algo.inria.fr`

these quantities. The call to the function `equivalent`, also in `algolib` determines the asymptotic value of the generating function coefficients.

The function `agfmomentsolve` differentiates a given number n times with respect to each variable, sets the non-size variables to 1 and solves. When $n = 0$ this gives the size generating function. When $n = 1$ it solves for $\frac{\partial}{\partial u}B(z,u)|_{u=1}$ (returned as $B[2](z)$), the cumulative generating function for the attribute marked by u.

```
> agfmomentsolve(eqns, 0);
> num_trees:= equivalent(subs(%, B(z)), z, n);
```

$$\left\{Node(z) = z, B(z) = 1/2\,\frac{1-\sqrt{1-4\,z}}{z}\right\}$$

$$num_trees := \frac{4^n}{\sqrt{\pi}n^{3/2}} + O(\frac{4^n}{n^{5/2}})$$

```
> agfmomentsolve(eqns, 1):
> tot_pl:= equivalent(subs(%, B[2](z)), z, n);
> avg_pl:= gdev(tot_pl/num_trees, n=infinity,2);
```

$$tot_pl := 4^n + O(\frac{4^n}{\sqrt{n}})$$

$$avg_pl := \sqrt{\pi}n^{3/2} + O(n)$$

The variance is obtained from the next moment, $[x^n]\frac{\partial^2}{\partial u^2}B(z,u)|_{u=1}$. This system is also solvable. The asymptotic value of the coefficient is $(\frac{10}{3} - \pi)n^3 + O(n^{\frac{5}{2}})$.

5. Automatic Complexity Analysis

Historically attribute grammars have had a close connection with algorithm description. This property can be modified such that an attribute describes the number of steps (however that is defined) an algorithm requires when a given structure is input. Generating functions summarize this information and offer a means for automatic average case complexity analysis.

This idea has been explored in depth with respect to the decomposable structures defined here [4, 5, 12]. In fact, the system LUO is an implementation of this concept for a family of algorithms. The main drawback of this system is that only the final univariate generating functions are available.

Attribute grammars give relations for multivariate generating functions and hence for problems it can represent additional information such as variance and higher moments are available. Further, it may be possible to do a distribution analysis. As shall soon be demonstrated, the class of algorithms that attribute grammars can describe contains the LUO class, and further all solutions are obtainable from the attribute grammar generating function equations.

5.1. **LUO.** LUO in its original form, and as implemented in the `combstruct` package, allows the user to describe a class of algorithms using several simple programming primitives: sequence of programs, test on unions, partial program descent and full component iteration. A program $P(a : A)$ takes as input a decomposable structure type A defined in an accompanying grammar. The program is littered with counters. The cost of execution on input A is then calculated as the sum of the counters encountered in a run. The program returns $\tau P_A(z)$ and an asymptotic value for $[z^n]\tau P_A(z)$, the total cost of running $P(a : A)$ on all inputs of type A and size n. The average complexity is then $[z^n]\tau P_A(z)/[z^n]A(z)$.

5.2. **LUO vs Attribute Grammars.** There is a direct correspondence to attribute grammars if the programs are viewed as attributes. The correspondence is summarized in the following theorem.

Theorem 4. *Any algorithm on a decomposable structure expressible in the LUO system can be rewritten as an attribute grammar representing the same complexity problem.*

Proof. Each programming primitive can be mapped to an attribute construction. A program $P(a : A)$ is translated into an attribute $P(A)$ and the complexity function $\tau P(z)$ is equivalent to $A_u(z, u)|_{u=1}$, where the variable z marks size and u marks attribute P. Thus, an equivalent system can be obtained and anything solvable in LUO, will be solvable by the same methods. It remains to create the map from programming primitives to attribute constructors.

A program consists of a sequence of sub-programs. Its total complexity is the sum of the complexities of the sub-programs. Thus, if attribute $P(A)$ is the value of the complexity for program P acting on structure A, clearly the attribute is a sum of the attribute values of the sub-programs. It remains to consider each single primitive.

The input are decomposable structures. The remaining primitives depend heavily on the input structure. If the input is a union of classes, a test on unions primitive distinguishes between the possibilities and sums the value according to the case. This is the same as the attribute for unions.

A partial component descent is encapsulated with the Prod operator for products and a selective attribute in the other cases. The full component iteration, maps to an iterative operator. As in the case of attribute grammars, only sub-programs of substructures can be called within a component descent.

□

5.3. **Example: Differentiation.** The analysis of differentiation illustrates the inclusion of LUO in attribute grammars.

Consider regular algebraic expressions composed of constants 0 and 1 and x using the two binary operations + and * and the unary operation exponentiation, that is, $x \mapsto e^x$. The grammar of this

set of expressions Exp with Atoms $+, *, e$ is easily described:

$$\begin{aligned} Exp &= 0|1|x|P|T|E \\ P &= Exp \cdot + \cdot Exp \\ T &= Exp \cdot * \cdot Exp \\ E &= e \cdot Exp \end{aligned}$$

The following program `Diff`, as it would be entered into LUO, is a program describing differentiation. The cost of a function is the number of atoms generated by the algorithm, thus the counters are one for each atomic structure.

```
function Diff(expr:Exp);
case expr of
  Prod(plus, e1, e2) : Plus(diff(e1), diff(e2));
  Prod(times, e1, e2): Plus(Times(diff(e1), e2),Times(e1, diff(e2)));
  Prod(expo, e1)     : Times(diff(e1), e);
  X                  : One;
  One                : Zero;
  Zero               : Zero;
end;
measure plus, times, expo, Zero, One, X : 1;
```

The attribute grammar version has one non-size attribute `DS`, whose value is the number of atoms when the expression is differentiated.

$$\begin{aligned} DS(Exp) &= 1|1|1|DS(P)|DS(T)|DS(E) \\ DS(P) &= DS(Exp) + 1 + DS(Exp) \\ T &= 3 + DS(Exp) + \text{ size}(Exp) + DS(Exp) + \text{ size}(Exp) \\ E &= 2 + DS(Exp) + \text{ size}(Exp) \end{aligned}$$

`size` refers to the structure size.

The resulting generating function equation set can be manipulated.

```
> gram:= {Exp = Union( zero, one, x,  Prod(plus, Exp, Exp),
              Prod(times, Exp, Exp), Prod(e, Exp))
         zero = Atom, x= Atom, one= Atom,
         plus = Atom, mult = Atom, e = Atom }:

> ag:= {DS(Exp) = Union( 1, 1, 1, Prod(1, DS(Exp), DS(Exp)),
            Prod(3, DS(Exp)+size(Exp), DS(Exp)+size(Exp)),
            Prod(2, DS(Exp)+size(Exp)))}:
> eqns:= agfeqns(gram, ag, unlabelled,[[u, DS]], z);
```

$$eqns := \{\mathrm{one}(z,\,u) = z\,u,\ \mathrm{p}(z,\,u) = z\,u,\ \mathrm{m}(z,\,u) = z\,u, \mathrm{Exp}(z,\,u) = u\,\mathrm{zero}(z,\,1) + u\,\mathrm{one}(z,\,1) + u\,\mathrm{x}(z,\,1) + u\,\mathrm{p}(z,\,1)\,\mathrm{Exp}(z,\,u)^2 + u^3\,\mathrm{m}(z,\,1)\,\mathrm{Exp}(z\,u,\,u)^2 + u^2\,\mathrm{e}(z,\,1)\,\mathrm{Exp}(z\,u,\,u),\ \mathrm{e}(z,\,u) = z\,u,\ \mathrm{x}(z,\,u) = z\,u,\ \mathrm{zero}(z,\,u) = z\,u\}$$

```
> num:= agfmomentsolve(eqns, 0):
> avg:= agfmomentsolve(eqns, 1):
```

Next, the generating functions for the expression and the cumulative size are isolated, and the asymptotic value of the coefficients is extracted:

```
> num_exp:= subs(num, Exp(z)):
  avg_exp:=subs(avg, Exp[2](z)):
> num_coef:=equivalent(num_exp, z, n):
  avg_coef:=equivalent(avg_exp, z, n):
  gdev(avg_coef/num_coef, n=infinity, 2); evalf(%);
```

$$8\,\frac{\left(\frac{13}{46} - \frac{3}{46}\sqrt{6} - \frac{17}{2}\left(-\frac{1}{23} + \frac{2}{23}\sqrt{6}\right)^2\right)\sqrt{\pi}\,n^{(3/2)}}{\left(\frac{2}{23} - \frac{4}{23}\sqrt{6} - 46\left(-\frac{1}{23} + \frac{2}{23}\sqrt{6}\right)^2\right)\sqrt{-\frac{2}{23} + \frac{4}{23}\sqrt{6} + 46\left(-\frac{1}{23} + \frac{2}{23}\sqrt{6}\right)^2}} + \mathrm{O}(n)$$

$$.8042175440\,n^{(3/2)} + \mathrm{O}(n)$$

The variance is also computable (unlike in LUO):

```
> agfmomentsolve(eqns, 2):
  facmom:=equivalent(subs(%, Exp[2,2](z)),z,n):
  evalf(gdev((facmom+avg_coef)/num_coef-(avg_coef/num_coef)^2, n=infinity, 2));
```

$$0.0394740311\,n^3 + O(1.0\,n^{5/2})$$

5.4. **Comparison.** LUO remains the more natural way to express some programming primitives such as direct structure descent and declarations with more than one input. It is not as straightforward with attribute grammars to develop a corresponding shorthand notation, consequently the grammars must be manipulated by the user. These manipulations are always possible, since similar manipulations are done internally within the implementation of LUO. However, for other types expressions the attribute description can be much simpler, and the general method simpler.

This implementation of attribute grammars considers each attribute/structure combination, even though in the LUO model a given function may only have one relevant structure input type. This can lead to multivariate functions which need to be manually reduced to be solved and manipulated.

The principle, and significant, advantage to the use of attribute grammars is the additional information. Variance and further moments can be extracted from the equations, unlike the LUO counterparts.

```
Quicksort(L:List)
 // return a sorted copy of L
   if |L|<=1 return L;
   otherwise
           Partition(L, L1, L2);
           return( Quicksort(L1), L[1], Quicksort(L2));
end;
Partition(L:List, L1:List, L2:List)
 // sets L1 to the list of elements of L smaller than L[1]
 // sets L2 to the list of elements of L larger than L[1]
 // if |L|=0 then sets L1, L2 to L
   if |L| = 0 then
         L1 = L; L2 = L;
   L[0]=0; L[|L|+1]=∞;
   i=2; j=|L|;
   l=L[1];
   repeat
           repeat i= i+1 until L[i] > l;
           repeat j= j-1 until L[j] < l;
           if i < j exchange(L[i], L[j]);
   until j < i;
   L1=L[2..j]; L2=L[i..|L|];
end;
```

FIGURE 2. Quicksort Algorithm

6. QUICKSORT

Quicksort is an extremely efficient method of sorting lists indexed by unique keys. There exists substantial analysis on its complexity and of the complexity of several variants, summarized nicely by P. Hennequin [9, 10]. This example illustrates how the type of problem it represents can be modelled with decomposable structures. It also illustrates the incorporation of a new constructor. Figure 2 gives the pseudocode for the Quicksort algorithm.

The following assumptions are made:

- distribution of the input is uniformly random
- the complexity is the total number of comparisons.

Under these conditions, a random instance of the algorithm corresponds to taking as input a random permutation. A run can be visualized with a binary search tree; the permutation is inserted in order into the binary tree in order and then each parent in the tree corresponds to a pivot. The left side of figure 3 illustrates this process on [521634]. In this view, the distance from the root gives

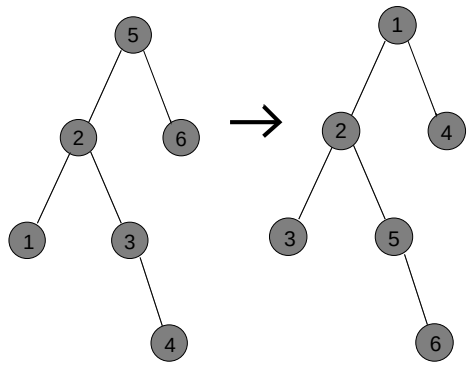

FIGURE 3. The binary search tree and increasing tree associated with [521634]

the number of times that number has been compared to others. That is, the internal pathlength gives the number of comparisons in total for the run represented by a given tree. Now binary trees are a decomposable structure as they have been defined thus far but they are not sufficient as a class to represent all instances. For example, the inputs 231 and 213 yield the same binary search tree. To remedy this problem, retain the shape of the search tree but label it instead with the order in which the nodes are filled. This eliminates the problem and creates a bijection. To reclaim the original permutation, parse the tree in infix order. If i is encountered at step j, there is a j at position i in the permutation. This resulting structure is a binary tree where the label at each node is less than the label of any ancestor, an *increasing tree*, or *heap*. To describe this with decomposable structures a new operator is required.

6.1. **Min Label.** Theorem 3 suggests that given an operator on decomposable structures, if the corresponding generating functions transformations are known, the linear attribute grammar generating function equations are determinable as well. D. Greene introduced a minimum label, or box operator in his thesis on decomposable structures [8]. In a labelled product one designates a particular component of the product as one to receive the smallest label. A sample construction looks like:

$$A = B \cdot Min(C) \cdot D \cdot E.$$

A structure of type A is labelled with its smallest label occurring in the second component. Greene also determined the generating function relationship:

$$A(z) = \int_0^z B(x) \frac{\partial C(x)}{\partial x} D(x) E(x) dx.$$

Having this, the minimum label operator (and similarly a maximum label operator) can be incorporated into the grammar vocabulary. Increasing binary trees can then be described as

$$H = \epsilon \mid \min(Z) \cdot H \cdot H,$$

since a label on a node of subtree must be less than its root.

6.2. **Analysis.** Given the structure, the analysis comes as a result of determining the average pathlength on the structures.

Define $H = \epsilon \mid \min(Z) \cdot H \cdot H$, and $\mathrm{ipl}(H) = 0 \mid \mathrm{ipl}(IT) + \mathrm{size}(IT) + \mathrm{ipl}(IT) + \mathrm{size}(IT)$. This yields the system:

$$H(z,u) = 1 + \int \left(\frac{\partial}{\partial z} z\right) (H(zu,u))^2 \, dz$$

which is solved:

$$H(z) = \frac{1}{1-z}$$

and has

$$\frac{[z^n]H_u(z,u)|_{u=1}}{[z^n]H(z)} = 2H_n - 3 + \frac{H_n}{n}$$

where H_n is the nth harmonic number.

This technique can be used to handle variants of Quicksort, relatives of Quicksort such as quick-select, and other problems on increasing trees.

7. Random Generation

Since the attribute grammar structure is so similar to the structures themselves, it is worthwhile to consider other information that the structures yield. Dutour and Fédou, described random generation procedures for a single attribute with an enumerable range value for CFGs [3]. The algorithm modifies the random generation algorithms for structures by adding an extra loop for possible values. Their algorithm determines, with a uniform distribution, a random element of a given size n and attribute value k in time $O(kn(k+n))$. The generalization to the full family of decomposable structures follows by a similar method as well by writing of the grammar in a standard form [7]. These standard forms use a Theta, or pointing operator and a Delta, or Diagonal operator, which readily admit random generation schemes with specified values for attributes.

8. Conclusions

Attribute grammars provide a succinct way of describing recursive properties of decomposable structures. The structure yields information as readily as the form of the objects themselves.

The implementation described here takes the definition of decomposible structures and presents a means to define properties. It remains to fully integrate the min operator and selective attributes into the combstruct package. The tools that do exist could make the first part of tools for further parameter analysis, such as distribution. Random generation for object grammars is implemented in the qALGO package of Dutour. A similar random generation package could be implemented for decomposible structures.

From the aspect of attribute grammar research, some theory has been developed on the idea of coupling grammars. This simulates repeated application of a function. This, for example, would allow a simple analysis of repeated differentiation, and other composed functions. This requires a system where the attributes may be more than constants, but rather structures.

References

[1] F. Bergeron, P. Flajolet, and B. Salvy. Varieties of increasing trees. In J.-C. Raoult, editor, *CAAP'92*, volume 581 of *Lecture Notes in Computer Science*, pages 24–48, 1992. Proceedings of the 17th Colloquium on Trees in Algebra and Programming, Rennes, France, February 1992.

[2] M. P. Delest and J.-M. Fédou. Attribute grammars are useful for combinatorics. *Theoretical Computer Science*, 98:65–76, 1992.

[3] I. Dutour and J.-M. Fédou. Object grammars and random generation. *Discrete Mathematics and Theoretical Computer Science*, 2:49–63, 1998.

[4] P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-Upsilon-Omega: The 1989 Cookbook. Research Report 1073, Institut National de Recherche en Informatique et en Automatique, August 1989. 116 pages.

[5] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science, Series A*, 79(1):37–109, February 1991.

[6] P. Flajolet and R. Sedgewick. The average case analysis of algorithms: Counting and generating functions. Research Report 1888, Institut National de Recherche en Informatique et en Automatique, April 1993. 115 pages.

[7] Philippe Flajolet, Paul Zimmerman, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1-2):1–35, 1994.

[8] D. Green. *Formal Languages and their Uses.* Ph. D. thesis, Stanford University, 1985.

[9] P. Hennequin. Combinatorial analysis of quicksort algorithm. *RAIRO Inform. Théor. Appl.*, 23(3):317–333, 1989.

[10] P. Hennequin. *Analyse en moyenne d'algorithmes, tri rapide et arbres de recherche.* Ph. D. thesis, École Polytechnique Palaiseau, 1991.

[11] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[12] P. Zimmerman. *Series génératrices et analyse automatique d'algorithmes.* Ph. D. thesis, École Polytechnique Palaiseau, 1991.

marni.mishna@math.uqam.ca

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
http://www.inria.fr
ISSN 0249-6399