Attribute grammars
and automatic complexity analysis

Marni Mishna

Projet Algorithms, INRIA, Rocquencourt, F-78153, La Chesnay, France
Received 5 October 2001; accepted 6 July 2002

Abstract
Attribute grammars provide a concise way to describe traits of a wide family of structures. Structures defined by context-free grammars have been well studied by Delest, Fédou, and more recently by Duchon. One of the principle benefits of this approach is the easy access to multivariate generating function equations from which average and higher moments are easily accessible. This work extends these notions to a wider class of structures and considers the application to algorithm analysis.

Introduction
Many combinatorial problems spawn from questions about certain traits of structures. In particular, enumerative questions about structures formed with operators such as union, Cartesian products, sets and cycles acting upon basic atoms have long been investigated, in particular automatically, by Ph. Flajolet et al. [6]. One of the underlying principles at work is the idea that structures built in a systematic way have the potential to be exploited in a systematic manner. Attribute grammars are a way to describe properties of structures in a systematic way. M.P. Delest et al. [1–3] illustrated how they are well suited to combinatorial problems. Essentially, they provide a concise, structured way to describe properties of combinatorial structures, such as pathlength in trees, and at the same time
give easy access to multivariate generating function equations. The equations can then be probed for information about averages and higher moments, or asymptotics.

This work explores the same strategy over a larger class of structures. Ph. Duchon and I. Dutour have thoroughly investigated the case where the structures are defined by context-free grammars. We will add the decomposable structures that Flajolet et al. have investigated and furthermore, illustrate how to add other grammar operators. All of the generating function consequences are detailed. To illustrate the potential of this system, we give some examples that come from algorithm analysis. Our Maple implementation of attribute grammars on decomposable structures is capable of handling the automatic complexity analysis problems that were treated by the LUO system [5]. Secondly, building upon a classic analysis of Quicksort we treat some related problems that have appeared recently, notably, the average number of key comparisons of multiple quickselect and problems related to bucket trees.

The outline of the article is as follows. The first section furnishes the formal definitions of decomposable structures and attribute grammars. Section two examines the affects on generating functions, summarised by Theorem 3. Next, we introduce the Maple implementation, including a proof of the containment of the LUO system within this framework, which yields some examples within algorithm analysis. Finally, we examine the sorting and searching problems mentioned earlier.

1. Attribute grammars

Attribute grammars were conceived by Knuth [8] as a way to keep track of certain properties of context-free grammars. Each node in the derivation tree has properties which are the “attributes.” M.P. Delest and J.M. Fédou applied this to combinatorial structures defined by context-free grammars, such as Dyck paths. Duchon and Dutour have given a complete characterization of those attributes, well suited to a particular type of manipulation, from which multivariate generating function equations result.

We begin by defining the type of structures we shall consider.

1.1. Grammars and specifications

Decomposable structures are described in detail in [5]. They are built from atoms of weight one (commonly expressed as $Z$), and zero ($\epsilon$). The constructors are disjoint union $\mid$, Cartesian product $\times$, sequence $\ast$, 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: $\text{card(inality)} \leq k$, $\text{card} = k$ and $\text{card} \geq k$, for any non-negative integer $k$.

\footnote{All packages mentioned in this article, including many other generating function tools, are available at http://algo.inria.fr/software.}
Objects are defined by specifications, and may be either labelled or unlabelled. To ensure that the grammar is well defined we require that each specification generate a finite number of objects of a given size.

Structures of this type include many types of trees (e.g., non-planar trees can be defined by \( T = \epsilon \mid Z \cdot \text{MultiSet}(T) \)), permutations (\( P = \text{MultiSet}(\text{Cycle}(Z, \text{card} > 0)) \)), and all structures definable with context-free grammars.

An attribute is a function which assigns a value to a structure. For example, the number of atoms is an attribute. If \( A \) is a structure that is defined in terms of structures \( A_1, \ldots, A_k \), then the attribute value must be a linear function of attribute values of \( A_1, \ldots, A_k \), and \( A \).

Given a set of specifications for decomposable structures \( G \), an attribute grammar is a set of attributes and their definitions for the structures in \( G \). In order for an attribute grammar to be well-defined we require that every attribute is a well-defined function, that is, every structure has a unique attribute value, which can be calculated. Essentially these correspond to attributes which can be recursively defined.

Duchon introduced the notion of rank. We translate his notion in the following way. An attribute which is a constant function is of rank one. Otherwise, the attribute of an arbitrary specification is one more than the largest rank of a structure referenced, not including itself. The rank is of interest in describing the complexity of generation schemes.

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.

Selective attributes are largely probabilistic, and hence are not thoroughly investigated here.

### 1.2. Example: Pathlength

A non-planar tree \( T \) admits the following decomposition:

\[
T = Z \cdot \text{Set}(T).
\]

That is, it is a node, and an unordered, possibly empty, set of subtrees. The size of \( T \), regarded as the number of atoms, is an attribute. It can be defined recursively for a given \( t \in T \) as \( \text{size}(t) = 1 + \sum_{t_i \text{ in the set of subtrees size}(t_i)} \). This fits within our requirement of linear in attributes of the descendants.

The depth of a node is the distance to the root. The sum of depths over all nodes in the tree is the (internal) pathlength (ipl) of the tree. This can be defined as an attribute. Notice that given a root node \( Z \) in a tree \( t \in T \), and one of its child subtrees \( t_1 \), the distance from every node in \( t \) is one more than its distance to the root of \( t_1 \). Thus the contribution of the nodes of \( t_1 \) to the total pathlength in \( t \) is the pathlength of \( t_1 \) plus one for every node, i.e., plus the size of \( t_1 \). That is, \( \text{ipl}(t) = \sum_{t_1 \text{ in the set of subtrees ipl}(t_1)} + \text{size}(t_1) \). Shortly, we will develop a better notation for such a description.
1.3. Linear attributes

We can formalize several notions which were introduced earlier. Given a structure defined by a rule $A = \Phi(B_1, \ldots, B_k)$, an attribute of this structure $F(A)$ 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 linear attributes or constant functions. An attribute grammar which consists only of linear attributes is aptly named a **linear attribute grammar**. In the lexicon of Delest and Duchon, the grammar is $Q$-computable.

2. Attribute specifications

Next we give a formal way to define the functions.

Consider the explicit definition of linear attributes. Let $AG = (G, F)$ be an attribute grammar, where $G$ is a grammar of decomposable structures and $F$ contains the specifications for the attributes $\{F_1, \ldots, F_n\}$. The structure specification gives the form for the attribute definition. 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 $B$. In the attribute specification, read the “|” as “in the case of” to determine the attribute value.

For example, given the grammar specification $C = A \mid B$, we express the attribute described with the following phrase: The attribute value $F(c)$ of an element $c$ in $C$ is $4$ if $c$ is originally in $A$ or $3G(b) + 1$ if the element is from $B$. (Where $G$ is another attribute in the attribute grammar) by the attribute specification

$$F(C) = 4 \mid 3G(B) + 1.$$  

In the general case, this is written

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

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 $B$.

So, with this grammar specification, we translate the word description of an attribute $F$: Given $c = (a, b) \in A \cdot B = C$, the value of the attribute $F(c)$ is $3F(a) + 4G(b)$, where $G$ is another attribute in the attribute grammar, by the attribute specification

$$F(C) = 3F(A) + 4G(B)$$

which will be notationally equivalent to $F(C) = \cdot(3F(A), 4G(B))$. 
Thus, the most general attribute specification of $F_i(C)$ is

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

Similarly, this will also be written

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

with a dot to facilitate notation readability.

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

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

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

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

The range of values of $F_i(A)$ is the set formed by considering all structures of type $\mathcal{A}$ and summing the value of the linear function over them. For example, the specification for the pathlength of trees described by Eq. (1) would be

$$\text{ipl}(T) = \text{Set}(\text{ipl}(T) + \text{size}(T)) - 1. \quad (2)$$

With this notation the general specification of a structure is

$$\mathcal{A} = \Phi_1(B_1^{(1)}, \ldots, B_{k_1}^{(1)}) \cdots \Phi_n(B_1^{(n)}, \ldots, B_{k_n}^{(n)}), \quad (3)$$

and an attribute of this specification has the general form:

$$F_i(A) = \bigcup_{m=1}^{n} \Phi_m \left( \sum_{j} \alpha_{ij}^{(1,m)} F_j(B_1^{(m)}), \ldots, \sum_{j} \alpha_{ij}^{(k_m,m)} F_j(B_{k_m}^{(m)}) \right) + \gamma_i. \quad (4)$$

In this general expression $\Phi_m$ is a non-union constructor ($\cdot, \text{Set}$, etc.), $k_m$ depends on the type of operator (for iterative $k_m$ is always 1) and indexed, lower case, Greek letters indicate integer constants. The sums with index $j$ are over all attributes $F_j$ in the attribute grammar.
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 attribute of specification (2) could be specified instead by the rule

\[
\text{ipl}(T) = \text{Set}(\text{ipl}(T)) + \text{size}(T) - 1,
\]

where the reference to size outside of an operator refers to the whole tree.

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 rewritten 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 \( \mathcal{A} \) be some decomposable structure and \( \mathcal{F} = \{ \mathcal{F}_i \}_{i=1}^n \) be a set of attributes defined for \( \mathcal{A} \). Since the attribute grammar is well defined they all have values on \( \mathcal{A} \). Thus, there exists some total ordering of evaluation of \( \mathcal{F} \), say \( \mathcal{F}_1', \mathcal{F}_2', \ldots, \mathcal{F}_n' \). The first in this total ordering must be either constant or depend on values of descendants. In either case all references to \( \mathcal{F}_1'(A) \) 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 \( \mathcal{A} \) are expressed in terms of descendants. \( \blacksquare \)

3. Generating functions

Generating functions conveniently encode information about combinatorial structures. If the specification is sufficiently structured, it will do much of the dirty work, such as encode attribute information within multivariate generating functions. By ‘sufficiently structured’ we mean that our attribute specifications mirror the specification of the structure and that we know a priori the translation from structure specification to generating function.

The set of translations for decomposable structures has long been known.

**Theorem 2** (Folk theorem of combinatorial analysis). *Given a specification for a class \( \mathcal{C} \), a set of equations for the corresponding generating functions is obtained automatically by the translation rules listed in Table 1. (The function \( \phi \) is Euler’s totient function.)*

Here we use the convention that the name of the class doubles as the name of the generating function. In the unlabelled case, \( A(z) = \sum_{n=0}^\infty a_n z^n \) where \( a_n \) is the number of structures of size \( n \). The labelled case is written \( A(z) = \sum_{n=0}^\infty a_n z^n / n! \).
Table 1
Generating functions of combinatorial operations

<table>
<thead>
<tr>
<th>Specification</th>
<th>C(z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C = A \mid B</td>
<td>A(z) + B(z) either</td>
</tr>
<tr>
<td>C = A \cdot B</td>
<td>A(z)B(z) either</td>
</tr>
<tr>
<td>C = A*</td>
<td>\frac{1}{1 - A(z)} either</td>
</tr>
<tr>
<td>C = \text{Set}(A)</td>
<td>\exp(A(z)) labelled</td>
</tr>
<tr>
<td>C = \text{Cycle}(A)</td>
<td>\ln\left(\frac{1}{1 - A(z)}\right) labelled</td>
</tr>
<tr>
<td>C = \text{Set}(A)</td>
<td>\exp\left(\sum_{k&gt;0} (-1)^{k-1} \frac{A(z^k)}{k}\right) unlabelled</td>
</tr>
<tr>
<td>C = \text{Cycle}(A)</td>
<td>\sum_{k&gt;0} \frac{\phi(k)}{k} \ln\left(\frac{1}{1 - A(z^k)}\right) unlabelled</td>
</tr>
<tr>
<td>C = \text{MultiSet}(A)</td>
<td>\exp\left(\sum_{k&gt;0} A(z^k)\right) unlabelled</td>
</tr>
</tbody>
</table>

For specification \( \Phi(B_1, \ldots, B_k) \) let \( G_\Phi(B_1, \ldots, B_k) \) 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 \( \{F_i\}_{i=1}^n \) is \( A(z_1, z_2, \ldots, z_n) = \sum_{a \in A} z_1^{a_1} z_2^{a_2} \cdots z_n^{a_n} \), and in a labelled universe each term is divided by \( |a|! \). By convention, the first parameter \( z_1 \) tracks 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.

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 \( z = (z_1, \ldots, z_k) \), then denote \( z^\alpha = (z_1^{\alpha_1} \cdots z_k^{\alpha_k}, \ldots, z_1^{\alpha_k} \cdots z_k^{\alpha_k}) \) and \( z^\gamma = (z_1^{\gamma_1} \cdots z_k^{\gamma_k}) \).

**Theorem 3.** Given a grammar specification for a combinatorial structure \( A \) in the form of equation (3) and a corresponding set of attributes, \( F = \{F_i\}_{i=1}^n \) each defined by equations as in (4), then the multivariate generating function

\[
A(z) = \sum_{a \in A} z_1^{a_1} \prod_{i=2}^n z_i^{F_i(a)}
\]

satisfies

\[
A(z) = \sum_m z^\gamma(m) G_\Phi(m) \left( B_1^{(m)} \left( z_1^{\alpha_1(m)} \right), \ldots, B_k^{(m)} \left( z_k^{\alpha_k(m)} \right) \right),
\]

where \( G_\Phi(m) \) 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 \( A(z) = z^r G_{\text{Set}}(B(z^a)) \).

\[
A(z) = \sum_{a \in A} z_1^{F_1(a)} z_2^{F_2(a)} \cdots z_k^{F_k(a)} = \sum_{a \in A} \prod_{i=1}^k z_i^{\gamma_i + \sum_{b \in B} \sum_{j=1}^k a_{ij} F_j(b)}
\]

\[
= z^r \prod_{b \in B} \left(1 + \prod_{i=1}^k z_i^{\sum_{j=1}^k a_{ij} F_j(b)}\right)
\]

\[
= z^r \prod_{(e_1, e_2, \ldots, e_k)} \left(1 + \prod_{j=1}^k \left( \prod_{i=1}^k z_i^{\alpha_{ij}} \right)^{e_j} \right)^{N_B(e_1, e_2, \ldots, e_k)}
\]

\[
= z^r \exp \left( \sum_{(e_1, \ldots, e_k)} N_B(e_1, \ldots, e_k) \ln \left(1 + \prod_{j=1}^k \left( \prod_{i=1}^k z_i^{\alpha_{ij}} \right)^{e_j} \right) \right)
\]

\[
= z^r \exp \left( \sum_{k \geq 1} (-1)^{k+1} \frac{B(z^{ka})}{k} \right) = z^r G_{\text{Set}}(B(z^a)).
\]

Here \( N_B(e_1, \ldots, e_k) \) 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 non-planar trees with path length are consequences of the definitions. By applying the attribute grammar version of the folk theorem we translate the grammar defined by Eqs. (1) and (5) into the equation \( T(z, u) = z \exp(T(zu, u)) \). 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 \( P = \text{Set}(\text{Cycle}(Z, \text{card} > 0)) \) (labelled) is determined by the attribute grammar, \{num(\(P\) = \text{Set}(1))\} which yields \( P(z, u) = z \exp(u \ln((1 - z)^{-1})) \).

4. Maple implementation

The main Maple package for combinatorial structure manipulation (combstruct) provides a template for the definition of attribute grammars.
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 combstruct structure specification is written

\[ C = \text{Union}(A, B) \]

and the attribute \( \mathcal{F}(C) = 4 \mid 3G(B) + 1 \), for example, is written

\[ \mathcal{F}(C) = \text{Union}(4, 3 \times G(B) + 1) . \]

The object/attribute pair of \( C = A \cdot B \) and \( \mathcal{F}(C) = 3\mathcal{F}(A) + 4G(B) \) is similarly encoded

\[ C = \text{Prod}(A, B) \]

\[ \mathcal{F}(C) = \text{Prod}(3 \times \mathcal{F}(A), 4 \times G(B)) . \]

The iterative constructors are similar.

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 \( \mathcal{F} \) invoke a default rule of \( \mathcal{F}(A) = \text{Union}(\mathcal{F}(A), \mathcal{F}(B)) \). 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

\[
> \text{sys:} = \{ B = \text{Union}(\text{Epsilon}, \text{Prod}(\text{Node}, B, B)), \text{Node} = \text{Atom} \}:
\]

\[
> \text{att:} = \{ \text{ipl}(B) = \text{Union}(0, \text{Prod}(0, \text{ipl}(B) + \text{size}(B), \\
& \text{ipl}(B) + \text{size}(B))) \}:
\]

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

\[
> \text{eqns:} = \text{agfeqns}(\text{sys}, \text{att}, \text{unlabelled}, [[u, \text{ipl}]], z);
\]

\[
\{ B(z, u) = 1 + z(B(zu, u))^2, \text{Node}(z, u) = z \}.
\]

---

4 In the combstruct lexicon the constructor Set refers to multisets.
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 \( [z^n]\frac{\partial}{\partial u}B(z,u)\)\(_{|u=1}\). Maple can solve for both of 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 \( u \).

\[
\begin{align*}
> \text{agfmomentsolve}(\text{eqns}, 0); \\
> \text{num_trees} := \text{equivalent}(\text{subs}(%B(z)), z, n); \\
& \left\{ \text{Node}(z) = z, B(z) = \frac{1}{2} \left( 1 - \sqrt{1 - 4z} \right) \right\} \\
& \text{num_trees} := \frac{4^n}{\sqrt{\pi} n^{3/2}} + O\left( \frac{4^n}{n^{5/2}} \right)
\end{align*}
\]

\[
\begin{align*}
> \text{agfmomentsolve}(\text{eqns}, 1); \\
> \text{tot_pl} := \text{equivalent}(\text{subs}(%B[2](z)), z, n); \\
> \text{avg_pl} := \text{gdev}(\text{tot_pl}/\text{num_trees}, n=\text{infinity}, 2); \\
& \text{tot_pl} := 4^n + O\left( \frac{4^n}{\sqrt{n}} \right), \quad \text{avg_pl} := \sqrt{\pi} n^{3/2} + O(n).
\end{align*}
\]

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 \( (10/3 - \pi) n^3 + O(n^{5/2}) \).

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 pro-
gram $P(a : A)$ 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. $\blacksquare$

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, $x \mapsto e^x$.
If we represent such an expression by its pre-fix expression tree, the grammar of this set
of expressions Exp with Atoms $+, *, e$ is easily described. We can describe the action
of differentiation on the expression tree and determine its size after differentiation. This post-differentiation size, $D$, gives us an indication of the complexity of the action of differentiation.

\[
\begin{align*}
\text{Exp} &= 0 | 1 | x | P | T | E, & \quad D(\text{Exp}) &= 1 | 1 | 1 | D(P) | D(T) | D(E), \\
P &= \text{Exp} \cdot + \cdot \text{Exp}, & \quad D(P) &= D(\text{Exp}) + 1 + D(\text{Exp}), \\
T &= \text{Exp} \cdot * \cdot \text{Exp}, & \quad D(T) &= 3 + D(\text{Exp}) + \text{size}(\text{Exp}) + D(\text{Exp}) + \text{size}(\text{Exp}), \\
E &= e \cdot \text{Exp}, & \quad D(E) &= 2 + D(\text{Exp}) + \text{size}(\text{Exp}).
\end{align*}
\]

The attribute size refers to the structure size.

The resulting generating function equation set can be manipulated, and using the same Maple tools described in the previous example. We can determine the asymptotic average length of an expression of length $n$ after differentiation $0.8042175440 n^{3/2} + O(n)$. The asymptotic variance is also computable, $0.0394740311 n^3 + O(n^{5/2})$.

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.

6. Examples: Analyses of sorting and selecting

The primary benefit of this point of view is that a variety of problems can be considered simultaneously. Having produced the grammar, there are automated techniques to extract additional information. Of course, this framework serves little use if no examples of interesting grammars can be found.

However, we can make use of a classic analysis of Quicksort, and a new grammar operator to approach several problems of algorithm analysis which have appeared recently.
In [7] Greene presents a method to analyse the average number of comparisons made on an input of \( n \) distinct keys to Quicksort by use of a morphism between an instance of Quicksort and increasing binary trees on \( n \) nodes. An increasing tree is simply a labelled tree with the restriction that the label on a node is strictly less than the label of its parent. These trees are easily described using grammar notation and as the number of comparisons in the sorting instance corresponds with the pathlength of the tree, we can provide an analysis of the number of comparisons.

The work in this example is done largely by the discovery of the morphism. Since there are many variants and neighbours of Quicksort, the hope is that they can be analysed with equal ease by use of a similar morphism.

The examples considered here are Quickselect, and bucket trees, each of which has been recently examined from a probabilistic point of view. We offer a succinct analysis and an illustration of how the certain characteristics can be calculated in a manner making use of a similar morphism as Quicksort.

6.1. Quicksort

We begin with a brief recollection of the original morphism. The Quicksort algorithm takes a list of \( n \) unsorted keys, selects a pivot key \( k \), and partitions the list into two: one of keys less than \( k \), and one of elements whose key is greater. The algorithm then recurses on the two sublists until the entire list is sorted. The comparisons arise in the partition stage. We assume that each key is compared once with the pivot, hence there are \( n - 1 \) comparisons make on the initial pivot with a call to Quicksort on \( n \) keys.

To analyse all possible instances, we use the following model. We consider an instance in terms of the relative orderings of the keys. Hence, an instance is simply one of the \( n! \) permutations of \( 1 \ldots n \). We assume a uniformly distributed choice of pivot, and so shall take the first in the list as a pivot point. Further, we assume that the ranks within the resulting sublists after partitioning around pivot \( k \) are uniformly distributed permutations of \( 1 \ldots k - 1 \) and \( 1 \ldots n - k \).

To illustrate how an instance of Quicksort becomes an increasing tree, we shall use an example. Let the order of the keys be 5723614. We take the pivot from the head of the list, hence 5. The two sublists after partitioning are then 2314 and 76. The next pivots on the recursive steps are 2, yielding 1 and 34 as sublists. Visually this corresponds to the binary tree:
However, this tree does not correspond uniquely to the instance. For example, 5762134 would have the same tree. To remedy this, we take the shape, which encodes some information and we label the nodes with their position in the permutation. Thus, a preorder traversal gives in order the position of 1, 2, ..., n in the ordinal permutation. The new tree under the morphism is:

Notice, in the original permutation 1 is in the 6th position, 2 in the 3rd, etc. Furthermore, since we place a node at the root before its child, the resulting structure is an increasing tree. Finally, the depth of a node corresponds to the number of times it was compared with a pivot, hence the sum of all the depths, or the internal path length, gives the total number of comparisons.

To model increasing trees we need to add a new operator to our tool box.

6.2. Min label: The box operator

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 [7]. In a labelled product one designates a particular component of the product as one to receive the smallest label. A sample construction looks like:

\[ A = B \cdot \text{Min}(C) \cdot D \cdot E. \]

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

\[ T = \epsilon | \text{Min}(Z) \cdot T \cdot T, \]
since a label on a node of subtree must be less than its root. We define pathlength as before:

\[ \text{ipl}(T) = 0 \left| \cdot (\text{size}(T) + \text{ipl}(T) + \text{size}(T) + \text{ipl}(T)) \right. \]  

(6)

These combinatorial equations give the following functional equations:

\[ T(z, u) = z \int_0^1 \left( \frac{\partial}{\partial z} z \right) T^2(zu, u) \, dz + 1 = \int_0^1 T^2(zu, u) \, dz + 1. \]

The average number of comparisons on input of size \( n \) is

\[ \left[ z^n \right] \frac{\partial}{\partial u} T(z, u) \bigg|_{u=1} \left[ z^n \right] T(z, 1). \]

Clearly, \( [z^n]T(z) = 1 \). Using Maple we can determine

\[ \left. \frac{\partial}{\partial u} T(z, u) \right|_{u=1} = -2 \frac{z}{(z-1)^2} - 2 \frac{\ln(z-1)}{(z-1)^2}. \]

Using the identity

\[ [z^n] \frac{1}{(1-z)^{m+1}} \log(1-z)^{-1} = \binom{n+m}{n} (H_{n+m} - H_m), \]  

(7)

where \( H_n = \sum_{k=1}^{n} 1/k \), we have the classic result that the average number of comparisons on input of size \( n \) is

\[ 2H_n - 3 + \frac{H_{n}}{n}. \]

We can also solve for higher moments by differentiating again, as we did in the earlier example of pathlength.

6.3. Quickselect

Suppose now instead of requiring the list of size \( n \) be sorted, that one only needs the value of the \( k \)th largest element. For example, the \( k = \lceil n/2 \rceil \), or the smallest value (\( k = 1 \)). Hoare modified Quicksort into the Find algorithm which, given a certain position \( k \), finds the value of the key which is the \( k \)th largest. More recently, this has been modified into an algorithm to select the values at several positions, \( 1 \leq k_1 < k_2 < \ldots < k_t \leq n \). For example, given \( k_1 = 1, k_2 = 2, \ldots, k_n = n \) the algorithm sorts the list.

We shall consider the case with one key, and explain how the others could be considered. A similar analyses appears by A. Panholzer and H. Prodinger [11]. H.M. Mahmoud and T.S. Smythe [10] considered this problem as well, and using probabilistic methods, determined a distribution. However, in the framework of attribute grammars, we determine exact values and remain on a very intuitive level.
6.3.1. Grammar

The details of the problem are as follows. We use the same partitioning stage as Quick sort. However, we only recurse on the sublist containing the desired element. If that element should be the pivot, we stop. We model this in the same way, except now, to encode the fact that we do not recurse on one side, we substitute that part of the tree with an unsorted list. The rest of the morphism is the same. We desire the average number of comparisons. To count this we ignore the calculations which determine which partitioned side contains the key. There are several justifications for this. Thus, the number of times each node is compared remains the distance from the root. By modifying our earlier definition of pathlength we can obtain the number of comparisons in a given instance.

\[
T = \text{Min}(Z) \cdot Z^* \cdot Z^* \mid \text{Min}(Z) \cdot Z^* \cdot T \mid \text{Min}(Z) \cdot T \cdot Z^* ,
\]

\[
\text{ipl}(T) = \cdot \left(0 + \text{size}(Z^*) + \text{size}(Z^*)\right)
\]

\[
\cdot \left(\text{size}(Z^*) + \text{size}(T) + \text{ipl}(T)\right)
\]

\[
\cdot \left(\text{size}(Z^*) + \text{size}(T) + \text{ipl}(T)\right) .
\] (8)

As before, the average number of comparisons is

\[
\left[\frac{z^n}{u} \frac{\partial}{\partial u} T(z, u)\right]_{u=1},
\]

and using Maple we can try to solve for these quantities exactly.

The combinatorial equations (8) yield the following generating function equations:

\[
T(z, u) = \int_0^\frac{z}{1} 2T(zu, u) \frac{dz}{1 - zu} + \int_0^\frac{z}{1} \frac{dz}{(1 - zu)^2} , \quad T(z) = z/(1 - z)^2 .
\]

Notice that \([z^n]T(z) = n\). Further, with Maple we can solve:

\[
\left[\frac{z^n}{u} \frac{\partial}{\partial u} T(z, u)\right]_{u=1} = \left[\frac{z^n}{u} \frac{\partial}{\partial u} T(z, u)\right]_{u=1} = \frac{2z - 8 \ln((1 - z)^{-1}) - 6}{(1 - z)^2} + \frac{6}{(z - 1)^3}
\]

\[
= -4n + 3(n + 2)(n + 1) - 8(n + 1)(H_{n+1} - 1) - 6,
\]

giving the average number of comparisons to be

\[
\left(\frac{-4n + 3(n + 2)(n + 1) - 8(n + 1)(H_{n+1} - 1) - 6}{n}\right).
\]

Another piece of information that we can extract using attributes is the number of recursive calls made, nrc. In this instance it is equal to the number of internal nodes and is given by the following grammar:

\[
\text{nrc}(T) = 1 \mid \cdot \left(1 + 0 + \text{nrc}(T)\right) \mid \cdot \left(1 + 0 + \text{nrc}(T)\right).
\]
This translates to the following functional equation:
\[
T(z, u) = \frac{2uT(z, u)}{1-zu} \cdot \frac{dz}{(1-z)^2} + \int \frac{dz}{(1-z)^2}.
\]

To find the average number of passes, we require:
\[
[z^n] \frac{\partial}{\partial u} T(z, u) \bigg|_{u=1} = [z^n] - \frac{2 \ln((1-z)^{-1})}{(z-1)^2} = 2(n+1)(H_{n+1} - 1) - n,
\]
which implies the average number of passes for input of size \( n \) is
\[
\frac{(2(n+1)(H_{n+1} - 1) - n)}{n}.
\]

By further differentiation we could find expressions for the variance and higher moments.

A modification of this grammar can give the multiple case, for any fixed number of keys. It possible that the grand average of grand averages may be reachable with this approach as well.

6.4. Bucket trees

In a 1995 paper [9], H.M. Mahmoud and R.T. Smythe introduced bucket recursive trees. They are a generalisation of the increasing trees we introduced earlier. Nodes in this tree structure are replaced by buckets which can hold up to \( b \) labels. When \( b = 1 \) we have the case of the increasing trees. A bucket tree can be seen to be a result of the following generation process. Labels are considered in order. A label can go, with equal probability, into any existing bucket. If that bucket is filled, it makes its own bucket. Thus, the root bucket, that is the bucket located at the root, forcibly contains labels 1, 2, up to \( b \).

He notes that these trees can be used to model a variety of possible recruiting situations as it has an intrinsic preference to saturation over spreading.

To model this structure with a grammar we begin with the initial example of the increasing trees and suitably modify it to allow buckets. If we model a bucket with an ordered list of labels, where the order in the list corresponds with the order of arrival, each bucket has a unique representation. Thus a ‘leaf’ is a bucket of size 1 to \( b \). In the process described, only buckets of size \( b \) can have children. Thus a bucket of size less than \( b \) is necessarily a leaf.

6.4.1. Grammar

Let \( \beta_n \) denote a bucket of size \( n \). A bucket of size one is clearly an atom: \( \beta_1 = Z \). To create an ordered bucket of size two we write \( \beta_2 = \theta(Z) \cdot Z \), since the labels are ordered. To create an ordered bucket of size three, we take an ordered bucket or size two, and multiply with an additional element under the restriction that the smallest label occur in the first component: \( \beta_3 = \theta(\beta_2) \cdot Z \). In general, \( \beta_n = \theta(\beta_{n-1}) \cdot Z \). Denoting by \( \beta_c \), the
union $\bigcup_{j=1, \ldots, n} \beta_j$, we can use the above descriptions to write a grammar for these trees, $T_{\text{bucket}}$:

$$T_{\text{bucket}} = \beta \cdot b + \text{Min}(\beta) \cdot \text{Set}(T_{\text{bucket}}).$$

(9)

Thus, for any fixed $b$ we can obtain the generating function equation that $T(x)$ satisfies. For some $b$ we can determine the generating function.

Furthermore, we can describe attributes such as internal pathlength:

$$\text{ipl}(T_{\text{bucket}}) = \text{size}(\beta) + b + \text{MultiSet}(\text{ipl}(T_{\text{bucket}}) + \text{size}(T_{\text{bucket}})).$$

7. Future directions

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 decomposable 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 constitute a start towards 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 decomposable structures.

There exist many sorting and selection algorithms that can be treated in a similar way. The framework set up here is very amenable to small modifications to a given problem.

Acknowledgments

The author gratefully acknowledges the financial support offered by NSERC through the PGS-A program in addition to the kind invitation of Projet ALGO, at INRIA, France where this work was completed. In particular, B. Salvy and F. Chyzak offered important insights and helpful, rigorous commentary.

References


