Using attribute grammars to derive efficient functional programs*

M.F. Kuiper and S.D. Swierstra

Department of Computer Science, University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
email: kuiper@cs.ruu.nl

Abstract

Two mappings from attribute grammars to lazy functional programs are defined. One of these mappings is an efficient implementation of attribute grammars. The other mapping yields inefficient programs. It is shown how some transformations of functional programs may be better understood by viewing the programs as inefficient implementations of attribute grammars.

1 Introduction

The transformational approach to programming starts with writing very clear and obviously correct programs. These programs are usually not very efficient. The efficiency of the programs is improved by applying successive transformations.

In this article we show that rewrite rules employing tupling[11] and deriving circular programs can be more easily expressed using attribute grammars[9, 10]. The method we decsribe also results in a form of common subexpression elimination.

We define two mappings from attribute grammars to functional programs. One of these mappings, SIM, yields programs that visit the nodes of a certain data structure usually more than once. The other mapping, CIRC, yields programs that visit the nodes of the same structure at most once. So a functional program that is the image of an attribute grammar A under SIM can be transformed into a possibly more efficient program by applying CIRC to A.

*This paper was published in the Conference Papers of CSN'87: Computing Science in the Netherlands. Published by SION in 1987.

Mapping CIRC can also be used to implement attribute grammars. CIRC yields attribute evaluators that visit each node of a structure tree only once and that perform no reevaluations of attribute values.

1.1 An example

The following example has been taken from [1]. The problem is to write a program that takes as input a non-empty binary tree t. Every leaf of t is labeled with an integer value. The output of the program must be a tree t' with the same structure as t but every leaf in t' is labeled with the minimum of the leaf values in t.

A tree is either a leaf with a value n, denoted by *(tip,n)*, or a node with two subtrees, denoted by *(fork,l,r)*. A straightforward functional program (Figure 1) consists of two functions, *tmin* and *replace*. Function *tmin* computes the minimum of the tip values of a tree. Function *replace* replaces in a tree all tip values by a given value. By combining these two functions the problem is solved.

```
tmin (tip,n)    = n
tmin (fork,l,r) = min (tmin l) (tmin r)

replace (tip,n)    min_in = (tip,min_in)
replace (fork,l,r) min_in
                   = fork (replace l min_in) (replace r min_in)

RESULT t = replace t ( tmin t )
```

Figure 1: algorithm 1

In Algorithm 1 the nodes of the tree are visited twice i.e. the algorithm inspects the type of each node twice. Bird [1] uses various rewrite techniques to obtain a solution that visits every node of the tree only once. We will call such a solution a *one touch* solution.

A different way to obtain a one touch solution is to write an attribute grammar for the input trees. The values of the functions *tmin* and *replace* are attached as attribute values to the nodes of the tree (Figure 2). The attribute grammar consists of three productions, numbered 0 to 2. The left hand side and the right hand side of a context free production are separated by an arrow. Attribution rules are written between curly brackets and immediately follow the context free rule. Attribute a of nonterminal L is refered to as $L.a$. If a context free rule contains more than one occurrence of a non-terminal then their uses in the attribution rules are indexed, starting with 0.

0 : L → tip
  { *L.tmin := tip.n ; L.replace := tip(L.min_in)* }.
1 : L → L L
  { *L[0].tmin := min(L[1].tmin,L[2].tmin)*
  ; L[0].replace := fork(L[1].replace,L[2].replace)
  ; L[1].min_in := L[0].min_in
  ; L[2].min_in := L[0].min_in }.
2 : ROOT → L
  { *ROOT.replace := L.replace*
  ; L.min_in := L.tmin }.

Figure 2: an attribute grammar for the problem

This non-circular attribute grammar can be mapped to a set of functions (Figure 3). For every non-terminal *X* in the grammar a function *eval_X* is created. *Eval_X* takes as arguments a structure tree and the inherited attributes of *X*. The result of *eval_X* is a list of the synthesized attributes of *X*. In an attribute grammar it is perfectly possible for an inherited attribute of a non-terminal to depend on some of the synthesized attributes of that same non-terminal. This will result in seemingly cyclic definitions where an argument in a function call depends on the result of that same call. In the example a cyclic definition occurs in the where-part of *eval_ROOT*. With our method such counterintuitive constructs are easy to understand and easy to write.

```
eval_L (tip,n) min_in = (n,(tip,min_in))
eval_L (fork,l,r) min_in
        = (min m1 m2 ,(fork,r1,r2))
        where (m1,r1) = eval_L l min_in
              (m2,r2) = eval_L r min_in

eval_ROOT t = r
          where (m,r) = eval_L t m
```

Figure 3: a one touch solution

In this article we formally show how to use the mappings from attribute grammars to functional programs. Attribute grammars are defined in the second section. Then, in section 3, it is shown how to implement attribute evaluators that do not perform an explicit tree walk on a structure tree. In section 4 the mappings from section 3 are used to rewrite functional programs. Section 5 contains a comparison with related work.

2 Attribute grammars

In this section attribute grammars are defined. The definitions are taken, almost literally, from [14].

2.1 Definitions

A context free grammar $G = (T, N, P, Z)$ consists of a set of terminal symbols T, a set of non-terminal symbols N, a set of productions P and a start symbol $Z \in N$.

When evaluating attributes we are not interested in the concrete syntax. Semantic analysis takes place using the abstract syntax. A *structure tree* obeys the abstract syntax. We assume that G describes the abstract syntax. To every node in a structure tree corresponds a production from G.

Definition 2.1 *An* attribute grammar *is a 4-tuple* $AG = (G, A, R, B)$. $G = (T, N, P, Z)$ *is a context free grammar.* $A = \bigcup_{X \in T \cup N} A(X)$ *is a finite set of attributes,* $R = \bigcup_{p \in P} R(p)$ *is a finite set of attribution rules and* $B = \bigcup_{p \in P} B(p)$ *is a finite set of conditions.* $A(X) \cap A(Y) \neq \emptyset$ *implies* $X = Y$. *For each occurrence of non-terminal* X *in the structure tree corresponding to a sentence of* $L(G)$*, exactly one rule is applicable for the computation of each attribute* $a \in A(X)$.

Elements of R have the form

$$X.a := f(\ldots, Y.b, \ldots).$$

In this attribution rule, f is the name of a function, X and Y are non-terminals and $X.a$ and $Y.b$ denote attributes. We assume that the functions used in the attribution rules are strict.

Definition 2.2 *For each* $p : X_0 \to X_1 \ldots X_n \in P$ *the set of* defining occurrences *of attributes is* $AF(p) = \{X_i.a | X_i.a := f(\ldots) \in R(p)\}$. *An attribute* $X.a$ *is called synthesized if there exists a production* $p : X \to \chi$ *and* $X.a$ *is in* $AF(p)$*; it is inherited if there exists a production* $q : Y \to \mu X \nu$ *and* $X.a \in AF(q)$.

AS(X) is the set of synthesized attributes of X. AI(X) is the set of inherited attributes of X.

Definition 2.3 *An attribute grammar is* complete *if the following statements hold for all X in the vocabulary of G:*

- *For all* $p : X \to \chi \in P, AS(X) \subseteq AF(p)$
- *For all* $q : Y \to \mu X \nu \in P, AI(X) \subseteq AF(q)$
- $AS(X) \cup AI(X) = A(X)$
- $AS(X) \cap AI(X) = \emptyset$

Further, if Z *is the root of the grammar then* AI(Z) *is empty.*

Definition 2.4 *An attribute grammar is* well defined *(WAG) if, for each structure tree corresponding to a sentence of L(G), all attributes are effectively computable. A sentence of L(G) is* correctly attributed *if, in addition, all conditions yield* true.

Definition 2.5 *For each* $p : X_0 \to X_1 \ldots X_n \in P$ *the set of* strict attribute dependencies *is given by*

$$DDP(p) = \{(X_i.a, X_j.b) | X_j.b := f(\ldots X_i.a \ldots) \in R(p)\}$$

The grammar is locally acyclic *if the graph of DDP(p) is acyclic for each* $p \in P$.

Definition 2.6 *Let S be the attributed structure tree corresponding to a sentence in L(G), and let* $K_0 \ldots K_n$ *be the nodes corresponding to an application of* $p : X_0 \to X_1 \ldots X_n$. *The set* $DT(S) = \{K_i.a \to K_j.b | X_j.b := f(\ldots X_i.a \ldots) \in R(p)\}$, *where we consider all applications of productions in S, is called the* dependency relation over the tree S. *The* dependency graph of S, *DG(S), is the graph of the relation* $DT(S)$.

The following theorem gives another characterization of well-defined attribute grammars. A proof can be found in [14].

Theorem 2.1 *An attribute grammar is* well-defined *iff it is complete and the graph DG(S) is a-cyclic for each structure tree S corresponding to a sentence of L(G).*

3 Functional implementations of attribute grammars

Attribute grammars are used to specify the semantics of programming languages. They specify the computation of attribute values attached to nodes in a structure tree. An attribute grammar can be transformed into a compiler[6]. A compiler based on attribute grammars usually consists of two parts: the first part parses the input and builds a structure tree; the second part, the attribute evaluator, decorates the structure tree i.e. it evaluates attributes that are attached to the nodes of the tree. Traditional implementations of attribute grammars perform a tree walk on the structure tree. Nodes in the structure tree are visited. During each visit to a node a subset of the attributes attached to the node is evaluated.

An alternative way to structure a compiler based on attribute grammars is to let the first part of the compiler construct the dependency graph of the structure tree of the input program. The second part of the compiler will reduce the constructed graph. Nodes in the graph correspond with attribute occurrences. A node that corresponds to an attribute a is labelled with the semantic function defining the value of a. If attribute a directly depends on attribute b there will be an arc from the node corresponding with a to the node corresponding with b.

An attribute evaluation scheme that explicitly constructs the dependency graph and then reduces this graph will be called a 2-phase attribute evaluation scheme. The first phase builds the graph. The second phase reduces the graph.

In this approach attribute values are viewed as terms. A term is either a basic value or a function applied to a list of terms. The basic values in the terms are the basic values in the attribute grammar, like integers and characters. The function symbols in the terms are the names of the semantic functions in the attribute grammar. An attribute evaluator must compute the synthesized attributes of the root of a structure tree. The dependency graph is a representation of these attributes.

We will, from now on, abstract from the use of attribute grammars in compiler generation. We consider attribute grammars as describing computations of values attached to nodes in a labelled tree.

3.1 A circular implementation of attribute grammars

The 2-phase attribute evaluation scheme can be implemented in a functional language with lazy evaluation and local definitions. In this article SASL [13] will be used. We will define the mapping CIRC that maps an attribute grammar into a functional program. CIRC constructs a SASL program that takes as input a structure tree corresponding to the underlying context free grammar of the attribute grammar. Trees are represented in SASL as lists. Every node consists of a marker and other lists representing the subtrees of the node. The marker in a node determines the applied production rule.

The pattern matching facility of SASL is used to distinguish between different productions with the same left hand side non-terminal. We will anly use simple forms of patterns. In our programs patterns are denotations of finite lists; the first element is the marker, which is a constant; the other elements are identifiers. The use of patterns in the function definitions is not essential.

The different productions of a non-terminal can also be distinguished in the body of the functions by using conditional expressions.

Assume that an attribute grammar AG=(G,A,R,B) is given, and B=$\emptyset$. Assume, without loss of generality, that for all X in N

$$\mathrm{AI}(X) = \{X.inh_0, \ldots, X.inh_{k_X - 1}\}$$

and

$$\mathrm{AS}(X) = \{X.s_0, \ldots, X.s_{l_X - 1}\}.$$

So X has k_X inherited and l_X synthesized attributes.

A non-terminal N_0 is translated into a SASL function $eval\_N_0$. The first argument of $eval\_N_0$ is a labelled tree. Production p : $N_0 \rightarrow N_1 \ldots N_n$ is translated into a definition for $eval\_N_0$:

$$eval\_N_0 \ (p, L_1, \ldots, L_n) \ inh_0^0 \ldots inh_{k_{N_0}}^0 = (s_0^0, \ldots, s_{l_{N_0}-1}^0)$$
$$\textbf{where } \mathrm{BODY(p)}$$

BODY(p) is the translation of R(p), the attribution rules for p. For every attribution rule , defining a synthesized attribute of N_0,

$$N_0.s_j \ := \ f(\ldots)$$

in R(p), BODY(p) contains a SASL definition

$$s_j^0 = f(\ldots).$$

For every attribution rule, defining an inherited attribute of N_j $(1 \leq j \leq n)$,

$$N_j.inh_i \ := \ f(\ldots)$$

in R(p), BODY(p) contains a SASL definition

$$inh_i^j = f(\ldots).$$

Occurrences of $N_j.s_i$ and $N_0.inh_l$ in $f(\ldots)$ are replaced by s_i^j and inh_l^0 respectively. For every N_j, $1 \leq j \leq n$, BODY(p) contains a definition

$$(s_0^j, \ldots, s_{l_{N_j}-1}^j) = eval\_N_j \ L_j \ inh_0^j \ldots inh_{k_{N_j}-1}^j$$

Theorem 3.1 *Let AG be a WAG, and let S be a structure tree obeying the context free grammar of AG. The execution of CIRC(AG) with input S terminates.*

Proof: The SASL program CIRC(AG) contains two kinds of functions: the eval functions and the semantic functions.

First note that the eval functions never cause non-termination. They split their first argument, a finite structure tree, in smaller parts and pass these to the eval-functions in their body.

The semantic functions are strict by definition. They do not terminate if they are called with a non-terminating argument or if they cause infinite recursion. If the latter happens then AG contains an error. So, to show that the execution of CIRC(AG) terminates, it must be shown that the semantic functions are always called with well defined arguments.

With the call of a function in BODY(p) corresponds a piece of the dependency graph DG(S). Suppose that BODY(p) is evaluated during the execution of CIRC(AG) S. If BODY(p) contains the definition

$$a = f(\ldots, b, \ldots, c, \ldots)$$

then DG(S) contains nodes corresponding with a,b and c (say α, β and γ); furthermore DG(S) contains arrows from β to α and from γ to α.

So if the computation of CIRC(AG) S leads to a infinite sequence of function calls then DG(S) must contain a cycle. This contradicts the assumption that AG is WAG. $\Box$

The case B $\neq \emptyset$ is an easy extension of the case B $= \emptyset$; the result of an eval function is extended with a boolean value. This boolean value indicates whether all conditions in the tree passed to this function yielded true.

4 Using attribute grammars to derive functional programs

Mapping CIRC can be used to implement attribute grammars. In this section we will define another mapping, SIM, from attribute grammars to functional programs. SIM can also be applied to all well defined attribute grammars. SIM is however too inefficient to act as a realistic implementation of attribute grammars. SIM and CIRC can be used in the derivation of efficient functional programs. A functional program that is the image of AG under SIM is usually inefficient: nodes in the structure may be visited more than once and attributes may be evaluated more than once. A more efficient program, equivalent with SIM(AG) can be derived by applying CIRC to AG. The strategy in transforming a functional program F is: first find an attribute grammar AG such that F=SIM(AG) and then apply CIRC to AG. Program F'=CIRC(AG) is equivalent with F

SIM maps every synthesized attribute to a function. For every synthesized attribute $N.s$ of AG, SIM(AG) contains a function *eval_N.s*. *Eval_N.s* takes as arguments a structure tree and all the inherited attributes of N_0. If

- s is a synthesized attribute of non-terminal N_0 which depends on v other attributes,

- $p \ : \ N_0 \rightarrow N_1 \dots N_n \in P$ and
- SF(p) contains $N_0.s := h(\dots)$

then SIM(AG) contains the definition

$$\begin{aligned} &eval\_N.s\ (p, L_1, \dots, L_n)\ a_0 \dots a_{v-1} = h(\dots) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where } BODY'(N.s) \end{aligned}$$

For every definition of an inherited attribute

$$N_j.inh_\alpha := g_\alpha(\dots)$$

in SF(p), $BODY'(N.s)$ contains a definition

$$inh^j_\alpha = g_\alpha(\dots).$$

For every synthesized attribute $N_j.s_\beta$, $1 \leq j \leq n$, $BODY'(N.s)$ contains a definition

$$s^j_\beta = eval\_N_j.s_\beta\ L_j\ inh^j_0 \dots inh^j_{k_j-1}.$$

Figure 1 contains an example of an image of SIM. This is a rather typical example. The images of SIM contain a lot of functions each working on the entire structure tree. As can be seen in the example the structure tree is visited more than once. The example does not show that during the execution of SIM(AG) attribute values might be computed more than once.

To apply our method of transforming functional programs it is necessary to find, given a functional program F, an attribute grammar AG such that F=SIM(AG). We first need some terminology. In this section we use the word function to denote a function defined in some functional program F. If a function definition has the form

$f\ \langle pattern\rangle\langle other\text{-}arguments\rangle = \langle expression\ \rangle$

then we say that the function f is defined with pattern $\langle pattern\rangle$ and that $\langle pattern\rangle$ is used in the definition of f. Note that there usually will be more than one definition of f, with different patterns. The function *tmin* from Algorithm 1 is defined with patterns *(fork,l,r)* and *(tip,n)*. Two functions are called *pattern equivalent* if they are defined with the same set of patterns. Likewise two patterns p_0 and p_1 are called *function equivalent* if the set of functions defined with p_0 and the set of functions defined with p_1 are equal. Clearly these two relations are equivalence relations. We let N^F_g denote the set of functions that are pattern equivalent with function g; N^P_r denotes the set of patterns that are function equivalent with r. We will use N_x for either N^F_x or N^P_x if it can be deduced from the context which set is meant.

To construct AG from F, assuming F=SIM(AG), we must first of all define the nonterminals of the grammar. Nonterminals are represented in F in two different ways; in the patterns and in the functions. The patterns in the program correspond to context free productions in the grammar. The left hand sides of these productions are nonterminals. The nonterminals are also represented by the functions implementing their synthesized attributes (the *eval_N.s* from the definition of SIM). There is a close connection between these two representations. In F the functions implementing the synthesized attributes of nonterminal X are defined with patterns whose productions have as lefthand side that same nonterminal X. This connection is the key to inverting SIM.

Lemma 4.1 *Let F be a functional program and AG be an attribute grammar. If F=SIM(AG) then there exists a bijection φ from $\{N_f^F|$ f is a function in F $\}$ to $\{N_p^P|$ p is a pattern used in F $\}$ such that $\varphi(N_f^F) = N_p^P$ implies that f is defined with p.*

Proof: Let N be the set of nonterminals from AG. Then every f in F corresponds to a synthesized attribute of a nonterminal in N. Let ρ_F be the mapping that assigns to a function the nonterminal it corresponds with. Every pattern used in a function definition in F is associated with a context free production from AG. Let ρ_P be the mapping that assigns to a pattern p the lefthand side nonterminal of the production that is associated with p. Mapping φ defined by

$$\varphi(N_f) = N_p \Longleftrightarrow \rho_F(f) = \rho_P(p)$$

is the required bijection. □

For the definition of SIM^{-1} we need some further terminology. A function definition is in normal form if it has the form

⟨*function-identifier*⟩ ⟨*pattern*⟩ ⟨*argument-identifier*⟩∗
=⟨*result-identifier*⟩
where
(⟨*identifier*⟩ = ⟨*function-identifier*⟩ {⟨*pattern*⟩} ⟨*identifier*⟩∗ *)*∗

A function is called a *pattern function* if all its definitions contain a pattern. A function is called a *semantic function* if none of its definitions contains a pattern. We let *patf* (*semf*) stand for a pattern function (semantic function). A subpattern is a L_i occurring in pattern $(m, L_0, \ldots, L_{K-1})$. A *terminal pattern* is a pattern whose subpatterns are passed to semantic functions only. A *nonterminal pattern* is a pattern whose subpatterns are passed to pattern functions only. Subpatterns of a terminal pattern stand for synthesized attributes of terminals, in nonterminal patterns they stand for sub trees of a structure tree.

Theorem 4.1 *Let F be a set of function definitions in normal form. F is the image of an attribute grammar AG under SIM iff*

1. *there exists a bijection φ from $\{N_f|$ f is a function in F $\}$ to $\{N_p|$ p is a pattern used in F $\}$ such that $\varphi(N_f) = N_p$ implies that f is defined with p.*

2. *if* $N_f = N_g$ *then* f *and* g *have the same number of arguments.*

3. *global functions are not partially parameterized.*

4. *a function that is called twice with the same pattern in the same where-clause is called with exactly the same identifiers as arguments*

5. *a pattern is either a terminal pattern or a nonterminal pattern*

6. *each subpattern is passed as argument to global functions of the same equivalence class only.*

Proof: $\Longrightarrow$ From the definition of SIM and lemma 4.1.
$\Longleftarrow$ Suppose the conditions hold. Define AG by:

nonterminals the set of nonterminals is $N = \{N_f^F \mid f \text{ is a function in F }\}$

terminals the set of terminals $T = \{T^{pat} | pat \text{ is a terminal pattern in F}\}$.

productions let $pat = (m, L_0, \ldots, L_{K-1})$ be a pattern used in F. If *pat* is a nonterminal pattern then AG contains a production

$$m : N_{pat}^P \rightarrow N_{L_0} \; ... \; N_{L_{K-1}}$$

where N_{L_j} is defined as the equivalence class of the function to which L_j is passed as an argument. If *pat* is a terminal pattern then AG contains a production

$$m : N_{pat}^P \rightarrow T^{pat}$$

synthesized attributes Nonterminal N_f has a synthesized attribute $N_f.g$ for all $g \in N_f$. Terminal T^{pat} has a synthesized attribute $T^{pat}.n$ for each subpattern n of terminal pattern *pat*.

inherited attributes Nonterminal N_f has k inherited attributes $N_f.inh_i$,$0 \leq i < k$, if a definition of f contains k *argument-identifiers*.

attribution rules Let

$$f \; p \; (m, L_0, \ldots, L_{K-1}) \; a_0 \ldots a_{v-1} = r \textbf{ where } \textit{where-clause}$$

be a definition from F. TR(x) is the translation of identifier x occurring in the *where-clause*:

$$\text{TR}(x) = \begin{cases} \textit{semf} \ldots \text{TR}(y) \ldots & \text{if } x = \textit{semf} \ldots y \ldots \text{ is the defining occurrence of } x \\ N_{patf}.\textit{patf} & \text{if } x = \textit{patf}\, L_j \ldots \text{ is the defining occurrence of } x \\ N_f.inh_j & \text{if } x \text{ is an argument identifier} \end{cases}$$

If x is the *result-identifier* of f and the *where-clause* contains the definition $x = semf \ldots y \ldots$ then the attribution rules contain $N_f[0].f :=$ $semf \ldots TR(y) \ldots$. For every definition $x = patf\, L_j\ arg_0 \ldots arg_{M-1}$ the attribution rules contain $N_{patf}[l].inh_i := TR(a_i)$ $(0 \leq i < M)$ where l is the the number of elements from N_{L_j} in the sequence $(m, L_0, \ldots, L_{K-1})$, $L_0, \ldots, L_{j-1}$. if x is the *result-identifier* of f then the attribution rules will also contain $N_f[0].f := N_{patf}.patf$.

From the conditions it follows that the above is the definition of an attribute grammar. It is straightforward to check that, after renaming some identifiers, F=SIM(AG). □

In the remainder of this section we will demonstrate our technique with a simple example. The problem[4] is to find the deepest nodes of a tree. A tree may have many leaves at the same depth so the result is a list of leaves. The program we derive is lazy: only lists that are needed to construct the answer are computed. The first and inefficient solution consists of two functions, *depth* and *front*.

$$
\begin{array}{lll}
depth\ (tip,n) & = & 0 \\
depth\ (fork,l,r) & = & 1 + max\ (depth\ l)\ (depth\ r)
\end{array}
$$

$$
\begin{array}{lll}
front\ (tip,n) & = & (n,) \\
front\ (fork,l,r) & = & depth\ l > depth\ r \rightarrow front\ l \\
 & & depth\ l < depth\ r \rightarrow front\ r \\
 & & depth\ l = depth\ r \rightarrow front\ l + front\ r
\end{array}
$$

The functions *depth* and *front* are pattern equivalent, and the patterns (tip, n) and $(fork, l, r)$ are function equivalent so the grammar has only one nonterminal. We will call this nonterminal L. Since (tip, n) is a terminal pattern the grammar has one terminal $T^{(tip,n)}$. We will call this terminal *tip*. The grammar has two context free productions, one for pattern (tip, n) and one for pattern $(fork, l, r)$:

$$
\begin{array}{l}
0 : L \rightarrow tip \\
1 : L \rightarrow L\ L
\end{array}
$$

L has two synthesized attributes $L.depth$ and $L.front$, but no inherited attributes since *depth* and *front* have no other arguments besides the patterns. The functions are not in the normal form required by the theorem but can be brought in this form by introducing some new identifiers. As an example a normal form for *depth* is

$$
\begin{array}{ll}
depth\ (fork,l,r) = r & \\
\qquad\qquad\qquad \textbf{where}\ r = 1 + max\ y\ z &
\end{array}
$$

$$y = depth\ l$$
$$z = depth\ r$$

Following the construction in the proof of theorem 4.1, substituting L for N_{depth}, we get $\mathrm{TR}(y) = L[1].depth$ and $\mathrm{TR}(z) = L[2].depth$. The attribution rules attached to production 1 will contain

$$L[0].depth := 1 + \mathrm{TR}(y)\ \mathrm{TR}(z)$$

The corresponding attribute grammar can be easily constructed:

0 : L → tip {*L.depth := 0; L.front := list(n)* }.

```
1 : L → L L {L[0].depth := 1 + max(L[1].depth,L[2].depth)
            ;L[0].front := L[1].depth > L[2].depth → L[1].front
                           L[1].depth < L[2].depth → L[2].front
                           L[1].depth = L[2].depth →
                                       append(L[1].front,L[2].front)}.
```

Now we can apply CIRC to derive the efficient solution:

```
eval_L (tip,n)     =(0,n)
eval_L (fork,l,r) =(1 + max l_depth r_depth,
                     l_depth > r_depth → l_front
                     l_depth < r_depth → r_front
                     l_depth = r_depth → l_front + r_front )
                     where (l_depth,l_front) = eval_L l
                           (r_depth,r_front) = eval_L r
```

5 Related work and conclusions

Other researchers have also described methods to translate attribute grammars into functions or procedures. Jourdan[5] gives a mapping from attribute grammars to functions. His target language is a non-lazy functional language. His translation yields a correct implementation for the class of absolutely non-circular attribute grammars[8]. Katayama[7] translates attribute grammars into Pascal procedures. In his scheme attributes may be evaluated more than once, although he claims otherwise. An overview of these and other evaluation techniques is given in[3].

Deransart and Maluszynski[2] use attribute grammars to analyse logic programs. They derive conditions under which a Prolog program allows a non-standard, but efficient, evaluation strategy.

Takeichi[12] obtains one touch algorithms by introducing higher order functions.

The main conclusion of this article must be that attribute grammars can be used to derive efficient functional programs. Whether the mapping CIRC is a feasible implementation of attribute grammars largely depends on the efficiency of functional language implementations and is beyond the scope of this article.

References

[1] R.S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[2] P. Deransart and J. Maluszynski. *Relating Logic Programs and Attribute Grammars.* Technical Report 393, INRIA, April 1985.

[3] J. Engelfriet. Attribute grammars: attribute evaluation methods. In B. Lorho, editor, *Methods and Tools For Compiler Construction*, pages 103–138, Cambridge University Press, 1984.

[4] J. Hughes. Lazy memo-functions. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 129–146, Springer, 1985.

[5] M. Jourdan. *An Efficient Recursive Evaluator for Strongly Non-circular Attribute Grammars.* Technical Report 235, INRIA, October 1983.

[6] U. Kastens, B. Hutt, and E. Zimmerman. *GAG: A Practical Compiler Generator.* Springer, 1982.

[7] T. Katayama. Translation of attribute grammars into procedures. *TOPLAS*, 6(3):345–369, July 1984.

[8] K. Kennedy and S.Warren. Automatic generation of efficient evaluators for attribute grammars. In *Proceedings of third conference on POPL*, pages 32–49, ACM, 1976.

[9] D.E. Knuth. Semantics of context-free languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[10] D.E. Knuth. Semantics of context-free languages (correction). *Math. Syst. Theory*, 5(1):95–96, 1971.

[11] A. Pettorossi. *Methodologies for Transformations and Memoing in Applicative Languages.* PhD thesis, University of Edinburgh, October 1984.

[12] Masato Takeichi. Partial parametrization eliminates multiple traversals of data structures. *Acta Informatica*, 24:57–77, 1987.

[13] D.A. Turner. A new implementation technique for applicative languages. *Software-practice and experience*, 9:31–49, 1979.

[14] W.M. Waite and G.Goos. *Compiler Construction*. Springer, 1984.