Attribute Grammars as a Functional Programming Paradigm

Thomas Johnsson
Department of Computer Science, Chalmers University of Technology
S-412 96 Göteborg, Sweden

Abstract

The purpose of this paper is twofold. Firstly we show how attributes in an attribute grammar can be simply and efficiently evaluated using a lazy functional language. The class of attribute grammars we can deal with are the most general ones possible: attributes may depend on each other in an arbitrary way, as long as there are no truly circular data dependencies.

Secondly, we describe a methodology based on attribute grammars, where, in a fairly straightforward way, we can develop efficient functional programs where direct, conventional solutions yield less efficient programs. We review two examples from a paper by R. Bird (Using circular programs to eliminate multiple traversals of data, Acta Informatica, 21, 1984) where he transforms simple but inefficient multipass programs into more efficient single pass ones, but which on their own can be very hard to understand. We show how such efficient but tangled programs can have natural formulations as attribute grammars.

We also propose a language construct, called **case rec** (akin to the **case** expression in Standard ML and Lazy ML), that defines an attribute grammar over a data structure in the language. In effect, a **case rec** expression defines a recursion operator that can handle multiple values, both upwards-propagating and downwards-propagating ones.

1 Introduction

Occasionally, using a functional language can be a bit of a pain: where the imperative programming solution is simple and obvious, the corresponding functional program can be both awkward and inefficient.

As a typical example, we want to organize a compiler as follows: The compiler makes multiple passes over the syntax tree, each pass computes some information which is assigned to nodes in the tree. It is straightforward to program this imperatively. However, in the straightforward functional solution, each pass would have to build a a new tree with the additional information in the nodes. Further, one may have to define a set of different tree types, one for each result tree from a pass!

Another example in the same vein is the task of assigning unique numbers to each node in a tree. The imperative program simply traverses the tree to update the nodes, obtaining a unique number by incrementing a global counter. The corresponding functional program has to drag along the unique number in the recursion, and the structure of the program becomes a bit messy.

In this paper we describe how such shortcomings can be overcome, by using a programming paradigm based on attribute grammars. Coupled with a simple and efficient method for attribute evaluation, based on lazy evaluation, we can also obtain efficient programs.

Attribute grammars [Knu68] were originally conceived as a method for specifying semantics of programming languages, but are nowadays mostly regarded as a convenient means of specifying syntax directed translations e.g. in compilers — see e.g. [AUS86]. Not surprisingly, the design of

efficient attribute evaluators has become a very active area of research, as the great number of papers in the field indicates (see [DJL85] for an annotated bibliography). Most efficient attribute evaluation systems determine an evaluation order at evaluation generation time, and impose constraints on how attributes may be written and may depend on each other, to be able to use a particular evaluation scheme; e.g. purely bottom-up, left-to-right-ness, strong non-circularity, etc. Others determine the evaluation order at runtime, but are then usually less efficient. See [DJL86b,DJL86a,DJL85] for a survey on main results, existing systems, and a classified bibliography.

It turns out that attribute evaluation can be done in a particularly simple way with a lazy functional language, without imposing any constraints on attribute dependencies. The difficulties in conventional languages seem to stem from the fact that an imperative program (or a strict functional one) specifies exactly in which order values are computed—consequently, the evaluation order for the attributes will have to be figured out at evaluation-generation time for each attribute grammar. On the other hand, in a lazy functional language implementation, the actual order in which expressions are evaluated is determined at runtime by the data dependencies, on demand, by the lazy evaluation machinery.

This paper is organized as follows. In section 2 we give a brief introduction to attribute grammars. In section 3 we describe the simple method for attribute evaluation. In section 4 we discuss attribute grammars as a convenient means of expressing algorithms traversing data structures (multiple passes in general), and describe how to turn these attribute grammars into functional programs which traverse the data structure only once. We review two examples from [Bir84] (which describe a different methodology for obtaining the same programs), and we compare the two approaches. In section 5 we discuss implementation issues, and some characteristic features of a graph reduction implementation. In section 6 we develop a language construct analogous to an attribute grammar, called **case rec** (in analogy with the **case** expression in LML and SML). Section 7 discusses circular attribute definitions. Section 8 concludes the paper.

We assume that the reader has some familiarity with lazy functional languages such as SASL [Tur76], Miranda [Tur85] or Lazy ML. Program examples in this paper will be given in Lazy ML (LML), a lazy and completely functional variant of ML [GMW79]. Like SML [Mil84], LML has borrowed the concrete data types and pattern matching from HOPE [BMS80]. LML is the source language for a compiler that compiles into efficient machine code that performs graph reduction [Aug84,Joh84].

2 A brief introduction to attribute grammars

An attribute grammar is a context-free grammar augmented with semantic rules. To each nonterminal symbol in the grammar a fixed set of *attributes* is associated. An attribute is either *synthesized* or *inherited*. The semantic rules for each production $X_0 \rightarrow X_1 X_2 \cdots X_i \cdots X_n$ specify the values of of the synthesized attributes of the left hand nonterminal X_0 and the inherited ones for the nonterminals X_i of the right hand side of the production rules. Evaluation of an attribute grammar with respect to a parse tree can be thought of as decorating the nodes in the parse tree with the values of the attributes. Thus, synthesized attribute values propagate upwards in the parse tree, inherited ones downwards. A synthesized attribute a of a nonterminal X will be written $X{\uparrow}a$; similarly, we write $X{\downarrow}a$ for an inherited attribute. Attributes may also depend on possible lexical values (e.g. the numerical value of the lexical symbol *INTCONST*), in which case they act as synthesized attributes assigned by the lexical analyzer. Here they will be written as e.g. $INCONST{\uparrow}lexval$.

Below, we give a (schematic) example of an attribute grammar: expressions *expr* with integer constants *INTCONST* and the single binary operator *PLUS*. We have two synthesized attributes,

called *S1* and *S2*, and one inherited attribute, called *I*. To distinguish between different occurences of the nonterminal *expr* in a production, indexing is used; i.e. $expr_1$ and $expr_2$.

$$
\begin{array}{ll}
expr \rightarrow expr_1\ PLUS\ expr_2 & expr{\uparrow}S1 = E_1[expr{\downarrow}I, \cdots expr_2{\uparrow}S2] \\
& expr{\uparrow}S2 = E_2[expr{\downarrow}I, \cdots expr_2{\uparrow}S2] \\
& expr_1{\downarrow}I = E_3[expr{\downarrow}I, \cdots expr_2{\uparrow}S2] \\
& expr_2{\downarrow}I = E_4[expr{\downarrow}I, \cdots expr_2{\uparrow}S2] \\
\\
expr \rightarrow intconst & expr{\uparrow}S1 = E_5[expr{\downarrow}I,\ expr{\uparrow}S1,\ expr{\uparrow}S2,\ INTCONST{\uparrow}lexval] \\
& expr{\uparrow}S2 = E_6[expr{\downarrow}I,\ expr{\uparrow}S1,\ expr{\uparrow}S2,\ INTCONST{\uparrow}lexval]
\end{array}
$$

Here E_1 through E_6 stand for arbitrary expressions to define the values of the attributes. Since we allow arbitrary attribute dependencies, the expressions E_1 through E_4 may have occurences of all the nine attributes $expr{\downarrow}I$, $expr{\uparrow}S1$, $expr{\uparrow}S2$, $expr_1{\downarrow}I$, $expr_1{\uparrow}S1$, $expr_1{\uparrow}S2$, $expr_2{\downarrow}I$, $expr_2{\uparrow}S1$ and $expr_2{\uparrow}S2$. Similarly, E_5 and E_6 may have occurences of $expr{\downarrow}I$, $expr{\uparrow}S1$, $expr{\uparrow}S2$, and the lexical value $INTCONST{\uparrow}lexval$.

Traditionally, in attribute grammar systems the language in which the attribute values are expressed is a conventional imperative one [DJL86a]. However, it should be clear that functional languages are perfect for expressing attribute definitions, as they provide the natural value-oriented view implicit in attribute grammars. Such languages have a very general notion of *value*. Values of expressions can be, among other things, lists (even infinite ones!), trees, functions etc. Thus such languages are perfect for expressing attribute definitions, as the value of an attribute can be a code sequence, a symbol table etc. In contrast, doing the same thing in a conventional language, like Pascal for example, requires much side-effecting — output code sequences, update symbol tables etc.

3 The evaluation method

Normally parser generator systems, like Yacc [Joh75] in Unix, provide at least some simple means of handling values. In Yacc, which constructs a bottom-up LALR(1) parser, a single upwards-propagating (synthesized) attribute can be handled. Our method constitutes transforming the attribute grammar into a new one with a single attribute, a synthesized one. Thus it should be straightforward to put our scheme on top of an already existing parser generator, provided that it can produce a parser in a lazy functional language. A parser generator for LML together with an attribute grammar system based on the method to be described below, is currently being implemented by G. Uddeborg [Udd].

The new single attribute is a function taking the original inherited attributes as arguments and returns a tuple of the original synthesized attributes. We now show how this works by transforming the attribute grammar in the previous section into this form. Thus, the new synthesized attribute, called *fn*, is a function taking the original inherited attribute $expr{\downarrow}I$ as argument and returning the pair of the original attributes $(expr{\uparrow}S1,\ expr{\uparrow}S2)$.

First we turn attribute identifiers $expr{\downarrow}I$ etc. into ordinary identifiers. We do that by simply replacing the attribute operators ↑ and ↓ with a character that may occur in an identifier — we will use underscore "_" for this purpose. Then these slightly rewritten definitions are simply inserted into a structure for handling the administrative task of obtaining the inherited attribute values of the left hand nonterminal symbol *expr*, and the synthesized ones for the nonterminals in the right hans side of the grammar rule. Thus the definitions of the first grammar rule are inserted into

$$
\begin{array}{l}
expr{\uparrow}fn = \lambda expr\_I. \\
\qquad \textbf{let rec}\ (expr_1\_S1,\ expr_1\_S2) = expr_1{\uparrow}fn\ expr_1\_I\ \textbf{and}
\end{array}
$$

$(expr_2\_S1,\ expr_2\_S2) = expr_2{\uparrow}fn\ expr_2\_I$ **and**
{ ... *attribute definitions* ... }
in $(expr\_S1,\ expr\_S2)$.

with $expr_1\_S1$ substituted for $expr_1{\uparrow}S1$, etc, in the attribute definitions. Thus the rewritten attribute grammar for our schematic example now is as follows:

$expr \rightarrow expr_1\ PLUS\ expr_2$
$expr{\uparrow}fn = \lambda expr\_I.$
let rec $(expr_1\_S1,\ expr_1\_S2) = expr_1{\uparrow}fn\ expr_1\_I$ **and**
$(expr_2\_S1,\ expr_2\_S2) = expr_2{\uparrow}fn\ expr_2\_I$ **and**
$expr\_S1 = E_1[expr\_I, \cdots expr_2\_S2]$ **and**
$expr\_S2 = E_2[expr\_I, \cdots expr_2\_S2]$ **and**
$expr_1\_I = E_3[expr\_I, \cdots expr_2\_S2]$ **and**
$expr_2\_I = E_4[expr\_I, \cdots expr_2\_S2]$ **and**
in $(expr\_S1,\ expr\_S2)$

$expr \rightarrow INTCONST$
$expr{\uparrow}fn = \lambda expr\_I.$
let rec $expr\_S1 = E_5[expr\_I,\ expr\_S1,\ expr\_S2]$ **and**
$expr\_S2 = E_6[expr\_I,\ expr\_S1,\ expr\_S2]$
in $(expr\_S1,\ expr\_S2)$

In a shift-reduce parser, e.g. an LR parser [AUS86], during parsing values associated to terminal and non-terminal symbols are kept on stack. When performing a reduction according to the first grammar rule above, the new value stack can be computed from the old one as follows.

reduce_rule_1 $(expr2\_fn\ .\ \_\ .\ expr1\_fn\ .\ restofstack) =$
$(\lambda expr\_I.$
(**let rec** $(expr1\_S1,\ expr1\_S2) = expr1\_fn\ expr1\_I$ **and**
$(expr2\_S1,\ expr2\_S2) = expr2\_fn\ expr_2\_I$ **and**
$expr\_S1 = E_1[expr\_I, \cdots expr2\_S2]$ **and**
$expr\_S2 = E_2[expr\_I, \cdots expr2\_S2]$ **and**
$expr1\_I = E_3[expr\_I, \cdots expr2\_S2]$ **and**
$expr2\_I = E_4[expr\_I, \cdots expr2\_S2]$ **and**
in $(expr\_S1,\ expr\_S2)$
) . *restofstack*

The key to the applicability of the method is the possibility in a lazy functional language to have recursive definitions of non-function values. Definitions are lazy, i.e., none of the defined values are actually computed until they are needed. Definitions where the left hand side is a variable pattern, like

let rec $(x,y) = e \cdots$

are treated as

let rec $A = e$ **and** $x = fst\ A$ **and** $y = snd\ A$ **and** $\cdots$

where *fst* and *snd* are functions returning the first and the second component of a pair, respectively.

Although attribute grammar evaluation according to our method can become reasonably efficient when coupled with an efficient implementation of a lazy functional language such as the

Lazy-ML compiler [Joh84,Aug84] it is probably less efficient than eg. the one in [Kat84] which accepts a restricted set of attribute grammars. Our method, in constrast has the advantage of full generality.

A similar idea has been mentioned in [KL81], where a tree with the attributes built into the nodes is returned as a result of parsing. A related idea is to let each function be a function from (all) inherited attributes to that particular synthesized attribute. This idea has been used in papers on semantics [May81] [CM79], but for implementation purposes this idea is useless except in special cases, since it can lead to large amount of recomputation of attribute values. This recomputation can be avoided if the functions are memoized [Hug85]. The evaluator in [Jou84] can be thought of as a limited form of lazy evaluator, an imperative program constructed by the evaluator-generator.

4 A functional programming paradigm

Since parse trees are nothing more than objects of a type generated by the grammar, the attribute grammar paradigm for defining values associated to nodes in a tree should be available to us not only when dealing with context free grammars and parse trees, but also for defining functions over any data structure.

In this context it is apt to refer to a paper by R. Bird [Bir84]. He describes a technique for transforming functional programs that repeatedly traverse a data structure, into more efficient ones that traverse the data structure only once. An essential requirement for this to be possible is that the functional language is lazy, and that local recursion (such as provided by **let rec** or **where rec**) is present in the language, to be able to define circular data dependencies. He makes use of fold-unfold transformations [BD77] to transform the straightforward but inefficient program into a program which is more efficient, but which can be very hard to understand on its own. Bird's program derivations are elegant applications of the fold-unfold transformation method coupled with circular programming, but they have the definite disadvantage of being accessible only to fairly sophisticated programmers, the hardest bit probably being to find the right 'eureka' definitions.

Below we will review two examples from [Bir84]. We will show how exactly the same efficient programs can be obtained by simpler and more straightforward means, by first expressing the programs as attribute grammars over data structures, and then by translating the attribute grammars into functional programs according to the idea described in 3. The otherwise tangled programs will thus reveal themselves as attribute grammar formulations of the algorithms. Although the attribute grammar may be specifying an inherently multipass algorithm over the data structure, the resulting evaluator program will make only one pass over the data structure.

The idea of specifying attribute grammars over data structures rather than over context free grammars is not entirely new. A similar idea is the basis of *attribute coupled grammars* [GG84], where an algebraic approach is taken: attribute grammars are viewed as specifying translations from source language terms to target language terms. Similar ideas appear in [Pau82].

4.1 An introductory example

Our task is to replace all the tips of a binary tree with the minimum tip value, with the shape of the new tree the same as the old one. The tree has two constructors, *tip* and *fork*:

type *Tree* = *tip Int* + *fork Tree Tree*

The straightforward solution, given in [Bir84], is as follows.[1]

```
transform t = replace t (tmin t)
where rec
replace (tip n) m = tip m ||
replace (fork L R) m = fork (replace L m) (replace R m) and
tmin (tip n) = n ||
tmin (fork L R) = min (tmin L) (tmin R)
```

The above program makes two passes over the tree, one performed by *tmin* to find the minimun value and one performed by *replace* to make the replacements.

The single pass version of the above program, derived in [Bir84], is as follows.

```
transform t = t1 where rec (t1,m1) = repmin t m1
where rec
repmin (tip n) m = (tip m, n) ||
repmin (fork L R) m = (fork t1 t2, min m1 m2)
                        where (t1,m1) = repmin L m
                          and (t2,m2) = repmin R m
```

The two functions *replace* and *tmin* have been replaced by a single function *repmin* doing the same work as the two. The above program has been obtained firstly by defining

```
repmin t m = (replace t m, tmin t)
```

(the 'eureka' step) from which the *repmin* function is synthesized by a standard application of the fold-unfold transformation method [BD77], and secondly by coupling the two components of the result value of *repmin* to each other using local recursion in *transform*.

We now recast the above tangled version of the program into a form analogous to an attribute grammar. But instead of assigning attributes to nodes in a parse tree for a context free grammar, we will look at the constructors of the tree type. In our example the patterns $T = tip\ n$ and $T = fork\ L\ R$ correspond to production rules in a context free grammar. (Compare this to the **as** pattern construct in SML and LML, e.g. T **as** $fork\ L\ R$, which binds a value, matched by a pattern, to a variable.) The variables T, L and R correspond to nonterminal symbols with attribute values being assigned to them by the semantic rules. The constructor symbols correspond to the terminal symbols. We also need to distinguish a 'start production', here written as $T = t$, with the understanding that this pattern match only at the root of the tree.

We first define a synthesized attribute *min*, being the the minimum tip value for each subtree:

$T = t$:
$T = fork\ L\ R$: $T{\uparrow}min = min\ L{\uparrow}min\ R{\uparrow}min$
$T = tip\ i$: $T{\uparrow}min = i$

Since we do not want the *min* value as a final result from the attribute evaluation, we do not need to define $T{\uparrow}min$ in the 'start production'. Next, we define the inherited attribute *rep*, being the obtained minimum value to replace the previous tip values—they are simply to be passed down the tree to the tips.

[1] In LML the keyword **and** separates definitions in a definition list, and || separates different cases of the same function.

$$
\begin{array}{ll}
T = t: & t{\downarrow}rep = t{\uparrow}min \\
T = fork\ L\ R: & L{\downarrow}rep = T{\downarrow}rep \\
 & R{\downarrow}rep = T{\downarrow}rep \\
T = tip\ i: &
\end{array}
$$

Finally, we define another synthesized attribute *tree*, being the new transformed tree.

$$
\begin{array}{ll}
T = t: & T{\uparrow}tree = t{\uparrow}tree \\
T = fork\ L\ R: & T{\uparrow}tree = fork\ L{\uparrow}tree\ R{\uparrow}tree \\
T = tip\ i: & T{\uparrow}tree = tip\ T{\downarrow}rep
\end{array}
$$

Merging these three attribute definitions into the same attribute grammar, we get the following.

$$
\begin{array}{ll}
T = t: & t{\downarrow}rep = t{\uparrow}min \\
 & T{\uparrow}tree = t{\uparrow}tree \\
T = fork\ L\ R: & T{\uparrow}min = min\ L{\uparrow}min\ R{\uparrow}min \\
 & L{\downarrow}rep = T{\downarrow}rep \\
 & R{\downarrow}rep = T{\downarrow}rep \\
 & T{\uparrow}tree = fork\ L{\uparrow}tree\ R{\uparrow}tree \\
T = tip\ i: & T{\uparrow}min = i \\
 & T{\uparrow}tree = tip\ T{\downarrow}rep
\end{array}
$$

The above attribute grammar can be evaluated in a single pass over the tree, using the technique shown in section 3. For this purpose we now define a function F, which takes as arguments the tree to be traversed and the inherited attribute *rep*, and returns a pair with the synthesized attributes *tree* and *min*:

$$F : Tree \rightarrow (Int \rightarrow Tree \times Int)$$

We further need a function which operates on the top level, taking the tree to be traversed and returning the attribute *tree* being the final answer to the programming problem:

$$Ftop : Tree \rightarrow Tree$$

The two functions are defined as follows.

$$
\begin{array}{l}
Ftop\ t = \textbf{let rec}\ (t\_tree,\ t\_min) = F\ t\ t\_rep \\
\qquad\qquad \textbf{and}\ t\_rep = t\_min \\
\qquad\qquad \textbf{and}\ T\_tree = t\_tree \\
\qquad\quad \textbf{in}\ T\_tree
\end{array}
$$

$$
\begin{array}{l}
F\ (fork\ L\ R)\ T\_rep = \textbf{let rec}\ (L\_tree,\ L\_min) = F\ L\ L\_rep \\
\qquad\qquad\qquad\qquad\quad \textbf{and}\ (R\_tree,\ R\_min) = F\ R\ R\_rep \\
\qquad\qquad\qquad\qquad\quad \textbf{and}\ T\_min = min\ L\_min\ R\_min \\
\qquad\qquad\qquad\qquad\quad \textbf{and}\ L\_rep = T\_rep \\
\qquad\qquad\qquad\qquad\quad \textbf{and}\ R\_rep = T\_rep \\
\qquad\qquad\qquad\qquad\quad \textbf{and}\ T\_tree = fork\ L\_tree\ R\_tree \\
\qquad\qquad\qquad\qquad \textbf{in}\ (T\_tree,\ T\_min)
\end{array}
$$

$\|$

$$F\ (tip\ i)\ T\_rep = \textbf{let rec}\ T\_min = i$$
$$\textbf{and}\ T\_tree = tip\ T\_rep$$
$$\textbf{in}\ (T\_tree,\ T\_min)$$

In this particular example, the definitions are recursive only in the function *Ftop*. After suitable simplification and renaming, the above program is exactly identical to the program derived in [Bir84], with *Ftop* as *transform* and *F* as *repmin*.

4.2 A further example

In the previous example the number of traversals of the tree was reduced by a small constant factor (from 2 to 1) by going from the straightforward solution to the transformed version or the attribute grammar version — a marginal improvement at best. However, this technique sometimes has the power of improving on the complexity of the algorithm, as the next example, also from [Bir84], will show. Again we are to transform a binary tree to another with the same shape, but this time we require that the tip values of the new tree are the tip values of the old tree sorted in ascending order. The direct solution to this problem can be formulated as follows.

$$transform\ t = replace\ t\ (sort\ (tips\ t))$$
$$\textbf{where rec}$$
$$replace\ (tip\ n)\ [m] = tip\ m\ \|$$
$$replace\ (fork\ L\ R)\ x = fork\ (replace\ L\ (take\ (size\ L)\ x))$$
$$(replace\ R\ (drop\ (size\ L)\ x))\ \textbf{and}$$
$$tips\ (tip\ n) = [n]\ \|$$
$$tips\ (fork\ L\ R) = tips\ L\ @\ tips\ R\ \textbf{and}$$
$$size\ (tip\ n) = 1\ \|$$
$$size\ (fork\ L\ R) = size\ L + size\ R$$

The tree is traversed a first time with the function *tips* to obtain a list of the tip values of the tree. The list is then sorted, and the tree is traversed a second time using the function *replace* to obtain the new tree. At each interior node we take appropiate chunks of the sorted list (the functions *take* and *drop* takes the k first and all but the k first, respectively, elements of a list) and pass them further down the tree.

The program has a worst time behaviour of $\mathcal{O}(n^2)$, n being the number of tips in the tree. There are three separate reasons for this, which each cause the program to have $\mathcal{O}(n^2)$ time complexity: (1) repeated calculation of sizes at each internal node, (2) because of the use of the functions *take* and *drop*, and (3) because concatenation is used to collect the list of tip values.

Bird now transforms away each of these inefficiencies. The inefficiency in the use of concatenation in *tips* is dealt with by defining

$$ntips\ t\ x = tips\ t\ @\ x$$

which does away with the need for concatenation entirely, since we then have

$$ntips\ (fork\ L\ R)\ x = ntips\ L\ (ntips\ R\ x).$$

Then, with a 'little foresight', the rest is taken care of with the following definition.

$$repnd\ t\ x\ y = [replace\ t\ (take\ (size\ t)\ x),\ drop\ (size\ t)\ x,\ ntips\ t\ y]$$

Using this definition, and three laws for the functions *take* and *drop* the following program is finally arrived at (we here omit the rather lengthy derivation, for further details see [Bir84]).

```
transform t = t1 where rec (t1, x, y) = repnd t (sort y) []
where rec
repnd (tip n) x y = ( tip (hd x), tl x, n.y ) ||
repnd (fork L R) x y = (fork t1 t2, x2, y1)
                       where rec (t1, x1, y1) = repnd L x y2
                             and (t2, x2, y2) = repnd R x1 y
```

We now derive the same program using an attribute grammar formulation. The efficiency improvement embodied in the definition of *ntips* can be formulated in terms of attributes as follows. Collect the list of tips by having attributes visiting the tips in reverse order, at each tip the current tip value is prepended to the hitherto obtained list of tips. Start at the root of the tree with the empty list []. This requires two attributes, one synthesized and one inherited, appropriately called *stips* and *itips*, whose definition for the three cases is given below.

$T = t$:	$t{\downarrow}itips = []$	The top case.
$T = tip\ n$:	$T{\uparrow}stips = n.T{\downarrow}itips$	
$T = fork\ L\ R$:	$R{\downarrow}itips = T{\downarrow}itips$	prepend the ones on the left ...
	$L{\downarrow}itips = R{\uparrow}stips$	... and then the ones on the right
	$T{\uparrow}stips = L{\uparrow}stips$	

To distribute the list of sorted tips, we again traverse the tree in the same manner as above, but this time in the order from left to right. At each tip the head of the sorted list is taken, and rest passed along. For this we require two more attributes, which we then call *ssorted* and *isorted*. Below we also show the construction of the new tree, carried by the attribute *tree*.

$T = t$:	$t{\downarrow}isorted = sort\ t{\uparrow}stips$	Top case: sort tips and pass down
	$T{\uparrow}tree = t{\uparrow}tree$	
$T = tip\ n$:	$T{\uparrow}ssorted = tl\ T{\downarrow}isorted$	Consume one element in the sorted list
	$T{\uparrow}tree = tip(hd\ T{\downarrow}isorted)$	
$T = fork\ L\ R$:	$L{\downarrow}isorted = T{\downarrow}isorted$	first distribute to the left ...
	$R{\downarrow}isorted = L{\uparrow}ssorted$	... and then to the right ...
	$T{\uparrow}ssorted = R{\uparrow}ssorted$	... and pass up what's left over.
	$T{\uparrow}tree = fork\ L{\uparrow}tree\ R{\uparrow}tree$	

The above attribute grammar (the two sets of definitions merged into one) is translated into two evaluation functions *Ftop* and *F*, in the same manner as in the previous example. Thus, the function *Ftop* takes the tree to be traversed and the synthesized attribute value *tree*. The function *F* similarly takes the tree to be traversed and the inherited attributes *itips* and *isorted*, and returns a triple with the synthesized attributes *stips*, *ssorted* and *tree*.

```
Ftop t =
    let rec (t_stips, t_ssorted, t_tree) = F t t_itips t_isorted
      and t_itips = []
      and t_isorted = sort t_stips
      and T_tree = t_tree
```

```
        in T_tree
and
F (tip n) T_itips T_isorted =
        let rec T_stips = n.T_itips
            and T_ssorted = tl T_isorted
            and T_tree = tip(hd T_isorted)
        in (T_stips, T_ssorted, T_tree)
||
F (fork L R) T_itips T_isorted =
        let rec (L_stips, L_ssorted, L_tree) = F L L_itips L_isorted
            and (R_stips, R_ssorted, R_tree) = F R R_itips R_isorted
            and R_itips = T_itips
            and L_itips = R_stips
            and T_stips = L_stips
            and L_isorted = T_isorted
            and R_isorted = L_ssorted
            and T_ssorted = R_ssorted
            and T_tree = fork L_tree R_tree
        in (T_stips, T_ssorted, T_tree)
```

simplifying the definitions as much as possible (which might be done by the compiler), we get:

```
Ftop t =
            let rec (t_stips, t_ssorted, t_tree) = F t [] (sort t_stips)
            in t_tree
and
F (tip n) T_itips T_isorted =
            (n.T_itips, tl T_isorted, tip(hd T_isorted))
||
F (fork L R) T_itips T_isorted =
            let rec (L_stips, L_ssorted, L_tree) = F L R_stips T_isorted
                and (R_stips, R_ssorted, R_tree) = F R T_itips L_ssorted
            in (L_stips, R_ssorted, fork L_tree R_tree)
```

The above program is essentially the same as derived by Bird, except for variable and function names. The function F corresponds to the function *repnd*, and *Ftop* to *transform*.

As we have seen, a definite disadvantage of the Darlington–Burstall method is that it requires great sophistication and cunning to find the right eureka definition, on the part of the user — this second example is a good example of that. Our method is more straightforward in that respect. On the other hand, once one has found the right eureka definition(s), the transformations provide more or less their own correctness proof. Thus, our attribute paradigm can also be thought of as a systematic method for obtaining the heureka definitions.

5 Implementation through graph reduction

We now show how a graph reduction implementation of a lazy functional language behaves when executing programs like the ones in the previous section.

Figure 1 illustrates graph reduction à la G-machine [Joh84] of the expression *Ftop*(*fork*(*tip* 2)(*tip* 5)), with *Ftop* defined as in section 4.1. Figure 1(a) shows the graph for the initial expression.

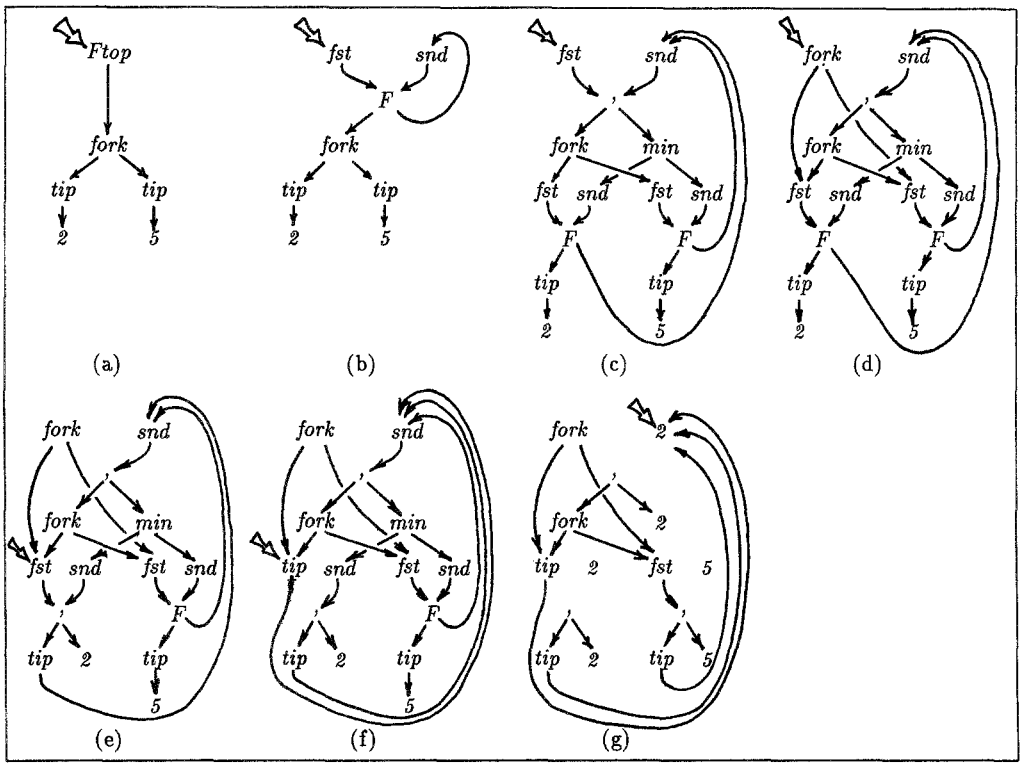

Figure 1: Graph reduction of $Ftop(fork(tip\ 2)(tip\ 5))$ from the introductory example.

When reducing a function application where the body of the function contains recursive data definitions, as in e.g. the function *Ftop*, the compiled G-machine code for the function builds cyclic graphs corresponding to the circular data depedencies. Thus, in our example the G-code for *Ftop* rewrites the graph to that of (b). The reduction then continues with an application of *fst*. *fst* calls eval for its argument thus causing the reduction of the application of F, which reduces the application of F to pair-form, (c). *fst* then calls eval for the first component of the pair, but in this case it is allready on constructor form (i.e. *fork*), so eval immediately returns. Finally, *fst* copies the *fork* constructor node onto the *fst* application, (d). Having reduced the entire expression to *fork* form, the original call to eval is now completed.

Let us further assume that the value of the left component of *fork* is requested: then *fst* calls eval to reduce its argument (the application $F(tip\ 2)\cdots$) to pair-form, calls eval for the first component of the resulting pair and finally copies the *tip* node onto the *fst* application (figures (e) and (f)).

Let us finally assume that the integer value of the *tip* is requested. Now the evaluation of the minimum tip value takes place, resulting in the graph shown in figure (g).

Thus the lazy evaluator has taken over the job normally done by the special purpose attribute evaluation machinery. Normally in other attribute grammar systems the order in which the attributes are evaluated are determined at evaluator-generation time. In our scheme this order is implicitly determined by the lazy evaluator at runtime. The order is entirely determined by the data dependencies, and may vary depending on the order in which the values of the various attributes are demanded.

Similar behaviour will be exhibited when the attribute evaluation machinery is incorporated into the parser, shown in section 3. No explicit parse tree is built. What will happen instead is that a tree-structured graph will be built, representing a function from the inherited attributes to the synthesized ones. This 'function tree' will have the same structure as the parse tree. When this function is applied to the initial inherited attributes, and evaluation demanded to obtain the values of the synthesed attributes of the root, graph reduction will proceed in a manner similar to figure 1. As graph reduction proceeds, the function tree gradually disappears.

6 A language construct

In section 4 we demonstrated that the attribute grammar paradigm is useful for constructing functional programs. So far, we have appealed to the readers intuition on some issues— for instance, one attribute grammar rule was simply dubbed the 'start production' or the 'top case', and the attribute grammar was translated into functions with this in mind. It is time to formalize our hitherto informal notation. In this section we define an LML language construct, called **case rec**, (in analogy with the **case** expression in SML and LML).

But why would we want a language construct for something which is so simple to translate into ordinary LML anyway? Firstly, providing a specific notation for a programming paradigm makes it easier to learn and use (c.f. abstract data types for modular programming, the if-then-else and while for structured programming, etc). Secondly, the mere existence of a certain language feature attracts the programmers attention to a solution method that would perhaps not otherwise occur to him—a particularly important consideration in our case, since the programs involved are by themselves counter-intuitive. Also, for a programmer trying to understand someone else's program, a program like the ones we have derived may appear completely incomprehensible unless he knows how the program has been derived.

We want to design an LML language construct which could we regarded as syntactic sugar for the translation of it into evaluation functions, in the manner we described in the previous sections. Thus, we also want to be faithful to the polymorphic typing scheme of LML (and SML).

The first attempt looks as follows:

```
transform t =
    case rec t in
    T = t:          T↑tree = t↑tree and
                    t↓rep = t↑min
    ||
    T = fork L R:   T↑min = min L↑min R↑min and
                    T↑tree = fork L↑tree R↑tree and
                    L↓rep = T↓rep and
                    R↓rep = T↓rep
    ||
    T = tip i:      T↑min = i and
                    T↑tree = tip T↓rep
    end
```

The intended value of the above **case rec** expression is the value of the $T{\uparrow}tree$ of the root.

The above simple-minded construction have some obvious problems.

- What exactly should be the value of the **case rec** expression in general? We could just stipulate that at the root there must be only a single synthesized attribute and no inherited

one, in which case the value of the **case rec** expression obviously is the value of that attribute. This however seems to be an unnecessary restriction. More generally, we could let the value of the **case rec** expression be a function from the inherited attributes to a tuple with the synthesized ones, just as the resulting evaluation function. But in that case, the programmer needs to know in what order to apply the initial inherited attributes to the **case rec** expression, and in what order the synthesized attributes comes in the tuple being the value of the **case rec** expression.

- Somehow it must be made clear that the pattern $T = t$ above is intended to match only at the root of the scrutinized data structure.
- In the same **case rec** expression we may want to traverse data of different types; for instance,

  type *Expr* = *ID id* + *APL* (*List*(*Expr*))
  ⋮
  case rec *e* **in**
  $T = ID\ i$: ⋯
  $T = APL\ el$: ⋯
  $T = e.el$: ⋯
  $T = []$: ⋯
  end

  We have patterns and expressions both of types *Expr* and *list*(*Expr*). To get a well-typed program the above **case rec** expression has to be translated into (at least) two functions, one traversing objects of type *Expr*, one travering data of type *list*(*Expr*). An example of this will be given in section 6.3.
- Different sets of attributes may be of interest at different places. This is exemplified in our *transform* program above, where we have different sets of attributes associated to the left hand variables T in the 'productions'. Again this will cause a typing problem if we attempt to translate the **case rec** expression to a single evaluation function.
- We may also want different sets of attributes, or no attributes at all, to be associated to the variables in the right hand side patterns—so far we have no means of expressing our intentions in that respect. For instance, in our example we want to associate all the three attributes *rep*, *min* and *tree* to the variables L and R in the pattern *fork L R*, whereas we are not interested in any attributes for n in *tip n*.

To deal with these shortcomings, we propose that the programmers intentions should be made explicit by annotating the **case rec** expression with *attribute sorts*. By an attribute sort we mean an enumeration of the inherited and synthesized attributes that we associate to a variable. In our example, in the case $T =$ *fork L R* we thus have the attribute sort $rep \rightarrow min\ tree$ associated to the variables T, L and R, and we would like to so annotate them. Similarly, for the case $T =$ *tip* n we have the same attribute sort associated to T, but no attributes associated to n. In the case $T = t$ we would annotate t with the same sort as above, whereas T would receive the annotation $\rightarrow tree$, since T here has the single synthesized attribute *tree*. But in order for the notation not to be too cumbersome, we give names to the attribute sorts, and annotate with the names of the attribute sorts instead. In general an attribute sort declaration will thus look like

$$A{=}i_1 \cdots i_m \rightarrow s_1 \cdots s_n$$

where A stands for the name of the attribute sort, $i_1 \cdots i_m$ for the names of the inherited attributes, and $s_1 \cdots s_n$ for the names of the synthesized attributes. For our example, we thus need two attribute sort declarations, one for the general case,

$a = rep \rightarrow tree\ min$

and one for the top case,

$atop = \rightarrow tree.$

To indicate what attributes we are interested in for the top case, we put the corresponding attribute sort name (*atop* in this case) in conjunction with the expression to be scrutinized (t in this case). Doing this roughly corresponds to indicating the starting nonterminal in a context free grammar.

We have now arrived at the final form of the proposed **case rec** construct. Below we give the *transform* program anew, with attribute sort declarations and annotations added.

```
transform t =
    case rec t::atop
        sort atop = → tree
        and a = rep → tree min
    in
    T::atop = (t::a):          T↓tree = t↓tree and
                               t↓rep = t↑min
    ||
    T::a = fork (L::a) (R::a): T↑min = min L↑min R↑min and
                               T↑tree = fork L↑tree R↑tree and
                               L↓rep = T↓rep and
                               R↓rep = T↓rep
    ||
    T::a = tip i:              T↑min = i and
                               T↑tree = tip T↓rep
    end
```

6.1 Semantics of the case rec expression

So far, the semantics of the **case rec** expression has been described on terms of what it is translated into. In this section we give a semantics which is independent on the translation (albeit very similar in spirit).

The **case rec** expression has the following general form.

case rec e::A **sort** S **in** D **end**

where

e is the scrutinized expression,

A is the name of one of the attribute sorts declared in S,

S is the list of attribute sort declarations, with the syntax

S ::= s **and** s **and** $\cdots$
s ::= $id^* \rightarrow id^+$

i.e., there must be at least one synthesized attribute, and

D is the list of cases, with the syntax

$D ::= d \parallel d \parallel \cdots$
$d ::= id :: id = pattern\text{: } decl$

where *pattern* is as ordinarily found in a **case** expression, except that variable may be annotated with the name of an attribute sort, *id*::*sortname*, and *decl* is a declaration as ordinarily found in a **let** or **let rec** expression in LML.

In the **case** expression and in functions defined by cases in LML the patterns are to be tested for a match against the value of the scrutinized expression in sequential order from the first pattern to the last one. We want this to hold also for the **case rec** expression, but here we also consider the left hand variable and its annotation to be a part of the pattern. Thus to match we must have both that the scrutinized expression matches the pattern in the ordinary sense, and that the sort name annotating the left hand variable in the pattern is the same as the variable A annotating the scrutinized expression (which is checked before we try to match the pattern).

We define the value of the **case rec** expression as follows. First, find a pattern matching the scrutinized expression, as explained above. Assume that $V::A = pattern\text{: } decl$ is the first such match. Assume further that *pattern* has N annotated variables $V_j::a_j$. A and a_j are defined in S as

$a_j = i_{j1} \cdots i_{jm_j} \rightarrow s_{j1} \cdots s_{jn_j}$
$A = i_1 \cdots i_m \rightarrow s_1 \cdots s_n$

Then the value of the **case rec** expression is the same as the value of

$\lambda V{\downarrow}i_1. \cdots \lambda V{\downarrow}i_m.$
let rec $(V_1{\uparrow}s_{11}, \cdots, V_1{\uparrow}s_{1n_1}) =$ **case rec** $V_1::a_1$ **sort** S **in** D **end**
$V_1{\downarrow}i_{11} \cdots V_1{\downarrow}i_{1m_1}$ **and**
$\vdots$
$(V_N{\uparrow}s_{N1}, \cdots, V_N{\uparrow}s_{Nn_N}) =$ **case rec** $V_N::a_N$ **sort** S **in** D **end**
$V_N{\downarrow}i_{N1} \cdots V_N{\downarrow}i_{Nm_N}$ **and**
decl
in $(V{\uparrow}s_1, \cdots, V{\uparrow}s_n)$.

Here attribute identifiers $V{\uparrow}a$ and $V{\downarrow}a$ have the same status as ordinary identifiers, but $V{\uparrow}a$ etc should only be allowed to occur if V is an annotated variable for which we want attribute values, and that a is a valid attribute for V.

6.2 Translation of the case rec expression

Each **case rec** expression is translated into a set of mutually recursive evaluation functions, one for each attribute sort declaration.

For an annotated expression to match a pattern our semantics required that the sort annotating the left hand variable in the pattern is the same as the sort requested, i.e., as annotated to the scrutinized expression, and in attempting to match we thus need only try those patterns that has the correct annotation. We thus translate **case rec** $e::A \cdots$ **end** into the function application *FE_A* e, where the function *FE_A* tries to match $e{\uparrow}A$ against the patterns $V_k::a_k = pattern_k$. In the test for match the function *FE_A* need only include those cases where the identifier a_k is the same as A, and this goes for all the evaluation functions. The function *FE_A* thus becomes

FE_A $pattern'_{k_1} =$ *translation of* $decl_{k_1} \parallel$
FE_A $pattern'_{k_2} =$ *translation of* $decl_{k_2} \parallel$

where k_1, k_2 etc are the places where the attribute sort name a_{k_i} are the same as A, *pattern'* is the same as *pattern* but with annotations removed. The translation of $decl_k$ is a function, and can be a λ-expression analogous to the one in 6.1. For each case $V::A$ = *pattern*: *decl*, assuming the the pattern has N annotated variables $V_j::a_j$ and where the a_js have been declared as $a_j{=}i_{j1}\cdots i_{jm_j} \rightarrow s_{j1}\cdots s_{jm_j}$ and A is declared as $A{=}i_1\cdots i_m \rightarrow s_1\cdots s_n$, the corresponding case for the evaluation can be written as

FE_A pattern' V_i$_1$ $\cdots$ *V_i*$_m$ =
let rec (*V*$_1$*_s*$_{11}$, $\cdots$, *V*$_1$*_s*$_{1n_1}$) = *FE_a*$_1$ *V*$_1$ *V*$_1$*_i*$_{11}$ $\cdots$ *V*$_1$*_i*$_{1m_1}$ **and**
⋮
(*V*$_N$*_s*$_{N1}$, $\cdots$, *V*$_N$*_s*$_{Nn_N}$) = *FE_a*$_N$ *V*$_N$ *V*$_N$*_i*$_{N1}$ $\cdots$ *V*$_N$*_i*$_{Nm_N}$ **and**
decl'
in (*V_s*$_1$, $\cdots$, *V_s*$_n$)

where *decl'* is the same as *decl* but with attribute operators $V{\uparrow}a$ translated into identifiers *V_a* (the variables for the inherited attributes have been moved to the left hand side of the equal-sign, instead of being λ-bound in the right hand side, but the meaning is the same). The entire **case rec** expression is translated into

FE_A e
where rec declarations of evaluation functions.

Our *transform* example from section 4.1 will according to the above scheme be translated into the program given below.

```
transform t =
        FE_atop t
        where rec
        FE_atop t =
                let rec (t_tree, t_min) = FE_a t t_rep and
                        T_tree = t_tree and
                        t_rep = t_min
                in T_tree
        and
        FE_a (fork L R) T_rep =
                let rec (L_tree, L_min) = FE_a L L_rep and
                        (R_tree, R_min) = FE_a R R_rep and
                        T_min = min L_min R_min and
                        T_tree = fork L_tree R_tree and
                        L_rep = T_rep and
                        R_rep = T_rep
                in (T_tree, T_min)
        ||
        FE_a (tip i) T_rep =
                let rec T_min = i and
                        T_tree = tip T_rep
                in (T_tree, T_min)
```

6.3 Further examples

As a further example of the use of the **case rec** expression, we offer the following compiler-oriented one: assigning unique names to identifiers, so that identifiers bound in a **let rec** allways

have different names. The tree to be traversed has the following type:

$$Expr = ID\ Id + AP\ Expr\ Expr + LETREC\ (List(Id \times Expr))\ Expr$$

The attribute sort e is for traversal of *Expr*, and d for traversal of $List(Id \times Expr)$. Note that even if we had wanted wanted the same attributes to be defined over the two types, it had still been necessary to have two sort declarations. Note also that sorts may have the same name as variables without causing any confusion between the two (except perhaps in the mind of the programmer).

$rename\ e =$
case rec $e::etop$
  sort $etop = \rightarrow ren$
  and $e = map\ iu \rightarrow ren\ su$
  and $d = map\ iu \rightarrow ren\ su\ newmap$
in

$E::etop = e::e$:	$E{\uparrow}ren = e{\uparrow}ren$	**and**
	$e{\downarrow}map = []$ **and** $e{\downarrow}su = 0$	‖
$E::e = AP\ e1::e\ e2::e$:	$E{\uparrow}ren = AP\ e1{\uparrow}ren\ e2{\uparrow}ren$	**and**
	$e1{\downarrow}map = E{\downarrow}map$ **and** $e2{\downarrow}map = E{\downarrow}map$	**and**
	$e1{\downarrow}iu = E{\downarrow}iu$ **and** $e2{\downarrow}iu = e1{\uparrow}su$ **and** $E{\uparrow}su = e2{\uparrow}su$	‖
$E::e = ID\ id$:	$E{\uparrow}ren = ID\ newid$	**and**
	$newid = lookup\ id\ E{\downarrow}map$	**and**
	$E{\uparrow}su = E{\downarrow}iu$	‖
$E::e = LETREC\ d::d\ e::e$:	$E{\uparrow}ren = LETREC\ d{\uparrow}ren\ e{\uparrow}ren$	**and**
	$extendedmap = E{\downarrow}map \mathbin{@} d{\uparrow}newmap$	**and**
	$e{\downarrow}map = extendedmap$ **and** $d{\downarrow}map = extendedmap$	**and**
	$d{\downarrow}iu = E{\downarrow}iu$ **and** $e{\downarrow}iu = d{\uparrow}su$ **and** $E{\uparrow}su = e{\uparrow}su$	‖
$D::d = (id,\ e::e)\ .\ d::d$:	$D{\uparrow}ren = (newid,\ e{\uparrow}ren).d{\uparrow}ren$	**and**
	$newid = itoname\ D{\downarrow}iu$	**and**
	$e{\downarrow}map = E{\downarrow}map$ **and** $d{\downarrow}map = E{\downarrow}map$	**and**
	$D{\uparrow}newmap = (id,\ newid).d{\uparrow}newmap$	**and**
	$e{\downarrow}iu = D{\downarrow}iu + 1$ **and** $d{\downarrow}iu = e{\uparrow}su$ **and** $D{\uparrow}su = d{\uparrow}su$	‖
$D::d = []$:	$D{\uparrow}ren = []$	**and**
	$D{\uparrow}su = D{\downarrow}iu$	**and**
	$D{\uparrow}newmap = []$	

end

The notion of attribute sorts is quite a general and powerful device. It can be used to specify that different computations are to be done in different contexts. As an example of this, the following **case rec** expression has the value *true* if the scrutinized list l has an even number of number of elements, *false* otherwise.

case rec l:*even* **sort** $even = \rightarrow r$ **and** $odd = \rightarrow r$ **in**
  $L::even = []$: $L{\uparrow}r = true$
‖ $L::odd = []$: $L{\uparrow}r = false$
‖ $L::even = \_\ .\ l::odd$: $L{\uparrow}r = l{\uparrow}r$
‖ $L::odd = \_\ .\ l::even$: $L{\uparrow}r = l{\uparrow}r$
end

7 On circular attribute definitions

A classical problem with attribute grammars is to determine whether a particular attribute grammar is *well-formed* or *non-circular*, i.e., that for every possible parse tree, there are no circular data dependencies. This test has been proved to be intrinsically exponential [JOR75]. The usual assumption then is that for an attribute definition $X\uparrow a = expression$, the values of the attribute occurences in *expression* must be computed before the value of $X\uparrow a$ can be computed. This is not true with is lazy functional language: for instance,

$$X\uparrow a = 1.X\uparrow a$$

is a circular attribute definition in the traditional sense, but is perfectly well-defined in lazy functional language, the value of $X\uparrow a$ being the infinite list of ones. Furthermore, not only can an attribute value be totally defined or totally undefined, now we can also have partially defined values, like $1.\perp$ (i.e., computation of this value terminates with a cons and the first element defined, but the tail of the list is undefined). So, with a lazy functional language, the problem of welldefinedness of an attribute grammar is a much more diversified one. It is a subject for further research to determine to what extent existing algorithms for circularity-check can be usefully adapted to attribute grammar systems with lazy functional languages.

So the only practical road open to us seems to be to detect circularities at run-time; fortunately, though, this can be done at very little extra cost. In the G-machine [Joh84] this can be done by changing the tag of the apply node which is later going to be updated anyway with the value of the function application.

8 Concluding remarks

In this paper we have described a particular way of taking advantage of the normal-order semantics, to obtain non-obvious solutions to programming problems. As yet, we have not implemented the **case rec** expression in LML, but the style of circular programming implied has been used in programming the LML compiler itself (which is almost entirely written in LML). We first used this circular style when programming the lambda-lifting part—the functional program in [Joh85] is of this kind. It was only afterwards the we discovered that attribute grammars provided a clear explanation of what was going on! This technique was subsequently used also in the G-machine code to target code generator, for propagating information backwards and forwards in the G-code stream. In our opinion we are barely beginning to learn how to use the full power of lazy evaluation in functional programming, and how to construct functional programs which might be just as efficient as conventional imperative solutions.

Conventional wisdom has it that for conventional languages "semantic evaluation methods based on attribute grammars are currently not efficient enough compared with ad-hoc algorithms used in the usual hand-written compilers" (quote from [Kat84]), For functional languages the reverse may well be true—where a conventional functional program would make multiple passes over the abstract syntax tree and perhaps in the process build intermediate structures to store intermediate information between passes, a functional attribute evaluator makes only one pass over the parse tree — intermediate values are "stored" in closures representing the data dependencies. An efficient implementation of a lazy functional language such as the Lazy ML-compiler [Joh84,Aug84] is optimized to handle closures, the central mechanism of lazy evaluation. Thus I would not find it surprising if an attribute evaluator implemented in this manner could still compete successfully with a more conventional attribute evaluator implemented in a conventional language.

Acknowledgements: Lennart Augustsson, John Hughes, Bengt Nordström, Göran Uddeborg,

and Phil Wadler provided comments and suggestions for improvements on earlier versions of this paper.

References

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[AUS86] A. V. Aho, J. D. Ullman, and R. Sethi. *Compilers: Principles, Techniques, Tools.* Addison-Wesley Publishing Company, Reading, Mass., 1986.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[Bir84] R. S. Bird. Using circular programs to eliminate multiple traverals of data. *Acta Informatica*, 21:239–250, 1984.

[BMS80] R. M. Burstall, D. B. McQueen, and D. T. Sannella. Hope: an experimental applicative language. In *Proceedings of the 1980 ACM Symposium on Lisp and Functional Programming*, pages 136–143, Stanford, CA, August 1980.

[CM79] L. M. Chirica and D. F. Martin. An order-algebraic definition of knuthian semantics. *Math. Systems Theory*, 13:1–27, 1979.

[DJL85] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *A Survey on Attribute Grammars, Part III: Classified Bibliography.* Rapport de Recherche 417, INRIA, Rocquencourt, France, June 1985.

[DJL86a] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *A Survey on Attribute Grammars, Part II: Review of Existing Systems.* Rapport de Recherche 510, INRIA, Rocquencourt, France, March 1986.

[DJL86b] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *A Survey on Attribute Grammars, Part I: Main Results on Attribute Grammars.* Rapport de Recherche 485, INRIA, Rocquencourt, France, January 1986.

[GG84] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 157–170, Montreal, 1984.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh* LCF. Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[Hug85] R. J. M. Hughes. Lazy memo-functions. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Joh75] S. C. Johnson. *Yacc—Yet Another Compiler Compiler.* Technical Report 32, Bell labs, 1975. Also in UNIX Programmer's Manual, Volume 2B.

[Joh84] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[Joh85] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[JOR75] Mehdi Jazayeri, William F. Ogden, and William C. Rounds. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Communications of the ACM*, 18:697–706, 1975.

[Jou84] M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In *Proceedings of 6th Int. Symp. on Programming, LNCS 167*, pages 167–178, Springer-Verlag, April 1984.

[Kat84] Takuya Katayama. Translation of attribute grammars into procedures. *ACM Trans. on Programming Languages and Systems*, 6(3):345–369, July 1984.

[KL81] R. M. Keller and G. Lindstrom. *Applications of feedback in functional programming.* Technical Report, University of Utah, Salt Lake City, April 1981.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2:127–145, 1968.

[May81] B. Mayoh. Attribute grammars and mathematical semantics. *SIAM J. of Computing*, 10(3):503–518, August 1981.

[Mil84] R. Milner. Standard ML proposal. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[Pau82] L. Paulson. A semantics-directed compiler generator. In *Proc. 9th POPL*, 1982.

[Tur76] D. A. Turner. SASL *Language Manual.* Technical report, University of St. Andrews, 1976.

[Tur85] D. A. Turner. Miranda: A non-strict language with polymorphic types. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 1–16, Nancy, France, 1985.

[Udd] G. Uddeborg. *A Functional Parser Generator.* In preparation.