# Exploiting Attribute Grammars to Achieve Automatic Tupling

*Jeroen Bransen*

*Atze Dijkstra*

*S. Doaitse Swierstra*

Technical Report UU-CS-2014-013
March 2014

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

# Exploiting Attribute Grammars to Achieve Automatic Tupling

Jeroen Bransen, Atze Dijkstra, and S. Doaitse Swierstra

Utrecht University, {J.Bransen,atze,doaitse}@uu.nl

**Abstract.** Tupling of function results is a well-known technique in functional programming to avoid multiple traversals over the same data. When expressing these programs as attribute grammars, function results are expressed as shared attributes for which tupling is done automatically. In this paper we show how we can get tupling for free by using attribute grammars as an intermediate language. We evaluate the effectiveness of the approach by showing some benchmark results.

**Keywords:** attribute grammars, automatic tupling, program transformation, functional programming

## 1 Introduction

Separation of concerns is important when developing nontrivial programs. In functional programming languages, the ability to partition source code according to some separation of concerns is realized by using functions as components. Each function computes some (distinct) feature of the input and functions are composed to construct more complex functions.

In general, the price we pay for extra clarity is loss of efficiency. Over the years many of such inefficiencies have been addressed, but most of the investigated optimizations involve intraprocedural, not interprocedural (or whole program) optimizations. Clarity by separating concerns (or features) into different functions has as its consequence that programs are not always as efficient as possible. In most cases various interrelated features of the same input data are computed separately. Unfortunately this can lead to redundant computations. It is our intent to address such inefficiencies by using a novel approach based on old *Attribute Grammar* (AG) techniques.

A technique for avoiding these redundant computations is *automatic tupling*: several related functions that compute some feature of the same input data can be grouped into a single function that returns a tuple with the result of each of these functions. For each recursive call in the collection of original functions over a certain substructure of the input the new function is only called once for this substructure. All values for the recursive calls can then be taken from the resulting tuple. The process of automatically turning multiple functions into a tupled function is called automatic tupling, which can lead to superlinear speedups by avoiding repeated computations.

Another approach to this problem is to write the problem solution involving multiple functions over the same data structure as an attribute grammar. In AGs, compositionality can easily be achieved using whole program analysis, an approach which is for example taken in the *Utrecht University Attribute Grammar Compiler* (UUAGC) Swierstra et al. (1998). With this approach it is straightforward to separate concerns in AGs and the tupling is automatically obtained. You might say that AGs are a domain specific language for describing catamorphisms, with tupling for free.

One could argue that we should therefore switch from functional programming to AGs in all applications. This is however not always desirable because AGs describe the class of functions known as *catamorphisms* or folds. Although many functions are catamorphisms, not all are; for example functions that perform structural recursion over more than one argument at the same time. Such functions could theoretically be expressed as folds but not in a way that is attractive to programmers.

In this paper we therefore propose the translation of functional programs to AGs. When using existing methods to translate the AGs back to an efficient functional program this leads to a

$$
\begin{array}{l}
\textbf{data}\ \mathit{Tree}\qquad\ \mid\ \mathit{Leaf}\ \widehat{\mathit{Int}}\ \mid\ \mathit{Bin}\ \mathit{Tree}\ \mathit{Tree}\\
\textbf{data}\ \mathit{Ordering}\mid\ \mathit{EQ}\ \mid\ \mathit{GT}\ \mid\ \mathit{LT}\\[6pt]
\textbf{let}\ \mathit{depth}\ :\ \mathit{Tree}\to\widehat{\mathit{Int}}\\
\quad\ \ \mathit{depth}=\lambda t:\mathit{Tree}.\ \textbf{case}\ t\ \textbf{of}\\
\qquad\qquad\qquad \mathit{Leaf}\ n\ \to\underline{0}\\
\qquad\qquad\qquad \mathit{Bin}\ l\ r\to\underline{\mathit{succ}}\ (\underline{\mathit{max}}\ (\mathit{depth}\ l)\ (\mathit{depth}\ r))\\
\quad\ \ \mathit{dleaves}\ :\ \mathit{Tree}\to\widehat{[\mathit{Int}]}\\
\quad\ \ \mathit{dleaves}=\lambda t:\mathit{Tree}.\ \textbf{case}\ t\ \textbf{of}\\
\qquad\qquad\qquad\ \ \mathit{Leaf}\ n\ \to\underline{\mathit{singleton}}\ n\\
\qquad\qquad\qquad\ \ \mathit{Bin}\ l\ r\to\textbf{case}\ \underline{\mathit{compare}}\ (\mathit{depth}\ l)\ (\mathit{depth}\ r)\ \textbf{of}\\
\qquad\qquad\qquad\qquad \mathit{EQ}\ \to\mathit{dleaves}\ l\ \underline{+\!\!+}\ \mathit{dleaves}\ r\\
\qquad\qquad\qquad\qquad \mathit{GT}\ \to\mathit{dleaves}\ l\\
\qquad\qquad\qquad\qquad \mathit{LT}\ \ \to\mathit{dleaves}\ r\\
\textbf{in}\ \ \mathit{dleaves}
\end{array}
$$

**Fig. 1.** Deepest leaves in a binary tree (straightforward version)

method for automatic tupling. Furthermore, other AG optimization techniques can be applied too, for example that of *ordered AGs* corresponding to a *strictness analysis* at the source level.

Another motivation for this work is that AG definitions usually do not consist completely of AG code but also include expressions in some host language. This mix between AG code and host language code can be very useful for the programmer, but it limits the AG compiler in its optimization possibilities. The two possibilities are to either translate the AG to the host language and let the host language compiler do the optimizations, or to translate the host language fragments to AG and optimize the generated AG as a whole. It is the latter approach we investigate in this paper.

In this paper we illustrate the problem and the solution using an example (Section 2). We make the following contributions:

- We define a functional language $\lambda$ which is the simply typed lambda calculus with algebraic data types and recursive lets (Section 3)
- We define the syntax of *minimal higher order attribute grammars* (MHAGs) (Section 4)
- We give a translation scheme for translating $\lambda$-expressions to MHAGs (Section 5)
- We evaluate the translation with some benchmark tests (Section 6)

Finally, we describe related work (Section 7) and discuss some of the problems and shortcomings of our approach (Section 8). We end the paper with some concluding remarks and future work (Section 9).

## 2 Example

We start by showing an example that illustrate the problem and the result of our approach. The example is written in the language $\lambda$ which is introduced later in the paper. The language is essentially the simply typed lambda calculus with algebraic data types, recursive lets and primitive types and functions. We believe that the reader should have no problem understanding the example.

The straightforward solution is implemented in the way that we feel is the most intuitive and elegant. There are however more efficient but less elegant implementations. In this section we show both the elegant and the (derived) efficient implementation, and in later sections we show how we construct this efficient implementation using the translation to and from AGs.

In the example we construct a list containing the deepest leaves from a binary tree. A straightforward implementation is presented in Figure 1.

Two algebraic data types are defined, *Tree* for binary trees and *Ordering* for the result of comparing two integers. The constructor *Leaf* has a single child of type $\widehat{\mathit{Int}}$ which is a *primitive*

```
data Tree      | Leaf $\widehat{Int}$ | Bin Tree Tree
data Ordering  | EQ | GT | LT
data TreeSyn   | TreeSyn $\widehat{[Int]}$ $\widehat{Int}$
let depth   : TreeSyn → $\widehat{Int}$
    depth   = λts : TreeSyn. case ts of TreeSyn l d → d
    dleaves : TreeSyn → $\widehat{[Int]}$
    dleaves = λts : TreeSyn. case ts of TreeSyn l d → l
    semTree : Tree → TreeSyn
    semTree = λd : Tree. case d of Bin l r → semBin (semTree l) (semTree r)
                                   Leaf n  → semLeaf n
    semLeaf : $\widehat{Int}$ → TreeSyn
    semLeaf = λn : $\widehat{Int}$. TreeSyn ($\underline{singleton}$ n) $\underline{0}$
    semBin  : TreeSyn → TreeSyn → TreeSyn
    semBin  = λl : TreeSyn. λr : TreeSyn.
                let de  : $\widehat{Int}$
                    de  = $\underline{succ}$ ($\underline{max}$ (depth l) (depth r))
                    le  : $\widehat{[Int]}$
                    le  = semOrd $ho_1$ (dleaves l) (dleaves r)
                    $ho_1$ : Ordering
                    $ho_1$ = $\underline{compare}$ (depth l) (depth r)
                in  TreeSyn le de
    semOrd  : Ordering → $\widehat{[Int]}$ → $\widehat{[Int]}$ → $\widehat{[Int]}$
    semOrd  = λc : Ordering. λdll : $\widehat{[Int]}$. λdlr : $\widehat{[Int]}$.
                 case c of EQ → dll $\underline{++}$ dlr
                           GT → dll
                           LT → dlr
in λt : Tree. dleaves (semTree t)
```

**Fig. 2.** Deepest leaves in a binary tree (efficient version)

*type* which cannot be inspected but only be passed as an argument to a primitive function or as a return value.

The program itself consists of a let definition in which two functions are defined. The function *depth* computes the depth of the binary tree in a straightforward way. The underlined functions are *primitive functions* and are thus left abstract. The functions do what their names suggest, so $\underline{max}$ computes the maximum of two integers, $\underline{succ}$ the successor of an integer and $\underline{singleton}$ constructs a singleton list from its argument.

For the actual translation typing information is needed for every part of the program, including the primitive functions. We therefore assume that the types are known for all primitive functions, but we leave them implicit to make the example more concise.

The *dleaves* function computes the list of deepest leaves of the tree, thereby using the *depth* function to compute the depth of its children in the binary case. If the depths are not equal then the deepest leaves of the deepest subtree are returned, otherwise the two lists of deepest leaves are combined into the result list. In the leaf case the value is returned as a singleton list.

The problem with this definition is in the calls to *depth* in *dleaves*. For every call to *dleaves*, which in the worst case is once for each node $x$, the full subtree of $x$ is traversed by *depth*. This makes the complexity of the algorithm quadratic[1], while a linear solution is possible.

We show the result of translating this code to an AG and back in Figure 2. Note that the code shown here is not the direct output of the translation given later in the paper but has been manually prettified to make clear what is going on. The variable and function names have been changed to more meaningful names, and some parts have been inlined and $\beta$-reduced to make the code shorter.

[1] For the sake of the example, we assume that $\underline{++}$ takes constant time.

| | |
|---|---|
| $T, C, P, x, f$ | -- Type, constructor, primitive type, variable |
| | -- and function names respectively |
| $\lambda ::= \overline{d}\ e$ | -- Lambda term |
| $d ::= \textbf{data}\ T\ \overline{c}$ | -- Data type definition |
| $c ::= \mid C\ \overline{\tau}$ | -- Constructor with children |
| $e ::= x$ | -- Variable |
| $\quad\mid\ \lambda x : \tau . e$ | -- Lambda abstraction |
| $\quad\mid\ e\ e$ | -- Application |
| $\quad\mid\ C$ | -- Constructor |
| $\quad\mid\ \textbf{case}\ e\ \textbf{of}\ \overline{a}$ | -- Case distinction |
| $\quad\mid\ \textbf{let}\ \overline{t}\ \textbf{in}\ e$ | -- Recursive let |
| $\quad\mid\ \underline{f : \overline{\tau \to}\ \tau}$ | -- Primitive function |
| $\tau ::= T$ | -- Type of data type |
| $\quad\mid\ \tau \to \tau$ | -- Function type |
| $\quad\mid\ \widehat{P}$ | -- Primitive type |
| $t ::= x\ :\ \tau$ | |
| $\quad x = e$ | -- Let definition |
| $a ::= C\ \overline{x} \to e$ | -- Case alternative |

**Fig. 3.** Functional language $\lambda$

In this version the original *depth* and *dleaves* functions are tupled, and therefore their results are computed together in a single pass over the tree. The data type *TreeSyn* is used to store the result of this pass over the tree and the *depth* and *dleaves* functions are in this case simple projection functions.

The single pass over the tree is performed by *semTree*, and the *semLeaf* and *semBin* functions compute the result for their corresponding constructors, in the case of *Bin* using the results of its children. The functions *semLeaf* and *semBin* together form a *TreeSyn* algebra.

It can be seen that the case for *Leaf* is the trivial tupling of the two results. The case for *Bin* is slightly more complex, yet similar to the original version. The value *de*, for the *depth* case, is identical to the original formulation. However, it is important to notice that its definition is no longer recursive; the calls to *depth* are simple projection functions retrieving the corresponding value from the tuples. The same holds for the computation of the deepest leaves.

For the result of the call to *compare* there is an extra indirection compared to the original formulation. This indirection could be removed by inlining the *semOrd* function which would make the *dleaves* case also similar to the original formulation. The current presentation is however closer to the AG version of this code so we use that for didactic reasons.

## 3 A functional language

The *lambda calculus* forms the basis of all functional programming languages. We now formally introduce the language that we call $\lambda$: the simply typed lambda calculus extended with algebraic data types, recursive let and primitive types and functions.

The syntax of $\lambda$ is already introduced informally in Section 2 and is formally defined in Figure 3. The $\overline{g}$ construct in the figure denotes zero or more occurrences of $g$.

*Simply typed lambda calculus* The bases of our language are expressions from the simply typed lambda calculus. An expression is a variable, an abstraction or an application. Abstractions have explicit type annotations.

*Algebraic data types* We use algebraic data types as base types. At top level there is a list of data type definitions, each introducing a new type with a set of constructors. Each constructor has a name and some child types. The data type definitions can be (mutually) recursive, so the types of the children can refer to any of the data types that are declared at top level.

To the expression language we add constructors for constructing inhabitants of the data types. Furthermore, we add case distinction to pattern match on the values. Each case alternative matches on a constructor and binds the values of the children to the given variables in the body of that alternative.

*Recursive let* We use the recursive let definitions to define a set of mutually recursive functions. As with abstractions, all defined variables must have an explicit type annotation.

*Primitive functions* The addition of primitive functions is not strictly necessary. However, to make the example concise and to be able to leave out irrelevant details we add primitive functions and types. Type annotations are also needed for primitive functions, but in our example we leave out type annotations because we believe the reader can easily infer them from the name of the function. Furthermore, we restrict ourselves to first-order functions only, so the $\tau$'s in the primitive function calls may only be $T$ or $\widehat{P}$.

## 4 MHAGs

Attribute Grammars (AGs) Knuth (1968) are well-suited for the implementation of the semantics of programming languages. In most AG implementations the AG is translated to some host language that is then used for evaluation. Semantic rules are often specified in this same host language, for example in UUAGC Swierstra et al. (1998) with host language Haskell or JastAdd Ekman and Hedin (2007) with host language Java.

An AG defines the decoration of abstract syntax trees (AST) with attributes. An AG consists of a set of algebraic data types defining the shape of the AST, and a set of attribute definitions defining how to decorate the AST. An attribute is either *inherited*, meaning that the value depends on the context of the corresponding node, or *synthesized*, meaning that the value depends on the subtree rooted at the corresponding node. The attribute definitions specify for each production how to compute its synthesized attributes and the inherited attributes of its children out of its inherited attributes and the synthesized attributes of its children.

### 4.1 Higher order AGs

In *Higher order* AGs (HAGs) Vogt et al. (1989) the attribute values can be trees themselves and thus can be decorated with attributes. This extension to traditional AGs has numerous applications such as looking up a variable in an environment.

In most higher order AG systems the right-hand side of the rules are expressed as functions taking zero or more attribute values as input and returning the attribute value. These functions are usually written in a host language like Haskell and are a powerful tool for writing a compiler. This comes with a price: analyses and optimizations of such AGs are harder.

### 4.2 Minimal higher order AGs

In this paper we use what we call *Minimal Higher order Attribute Grammars* (MHAGs). The key difference with conventional HAGs is that the right-hand side of the semantic rules is specified in the AG itself and not in a host language. As a result, an MHAG compiler has more optimization possibilities. We discuss the MHAG syntax as shown in Figure 4.

*Data types* The tree structures in an MHAG are instances of algebraic data types. The only difference with the data types in $\lambda$ is that children in MHAGs are named. This name is used to refer to a child and its attributes.

*Attribute declarations* The attribute declarations specify for each nonterminal the inherited and synthesized attributes of this nonterminal. Attributes have a name and type, and an inherited attribute may have the same name as a synthesized attribute, since no confusion can arise.

$$
\begin{array}{lll}
N, P, x, y & & \text{-- Nonterminal, production, child and} \\
 & & \text{-- attribute names respectively} \\
t & ::= N & \text{-- Type} \\
 & \mid\ \widehat{N} & \text{-- Primitive type} \\
\psi & ::= \overline{D}\ \overline{A}\ \overline{S} & \text{-- Full MHAG definition} \\
D & ::= \textbf{data}\ N\ \overline{p} & \text{-- Data type for nonterminal } N \\
p & ::= \mid\ P\ \overline{c} & \text{-- Production } P \text{ with children } \overline{s} \\
c & ::= x :: t & \text{-- Child } x \text{ of type } t \\
A & ::= \textbf{attr}\ N\ \overline{i}\ \overline{s} & \text{-- Attribute declarations for } N \\
i & ::= \textbf{inh}\ y :: t & \text{-- Inherited attribute } y \text{ of type } t \\
s & ::= \textbf{syn}\ y :: t & \text{-- Synthesized attribute } y \text{ of type } t \\
S & ::= \textbf{sem}\ N\ \overline{o} & \text{-- Semantics for nonterminal } N \\
o & ::= \mid\ P\ \overline{r} & \text{-- Rules for production } P \\
r & ::= x.y \quad = q & \text{-- Inherited attribute } y \text{ of child } x \\
 & \mid\ \textbf{lhs}.y \ = q & \text{-- Synthesized attribute } y \\
 & \mid\ \textbf{loc}.y \ :: t & \text{-- Local attribute } y \text{ of} \\
 & \quad \textbf{loc}.y \ = q & \text{-- type } t \\
 & \mid\ \textbf{inst}.x :: N & \text{-- Declaration and instantiation of} \\
 & \quad \textbf{inst}.x = q & \text{-- higher order child } x \text{ of type } N \\
q & ::= u & \text{-- Value of attribute } u \\
 & \mid\ P\ \overline{u} & \text{-- Constructor } P \text{ with its arguments} \\
 & \mid\ \underline{f : \overline{t \to}\ t}\ \overline{u} & \text{-- Primitive function call} \\
u & ::= @\textbf{lhs}.y & \text{-- Inherited attribute } y \\
 & \mid\ @x.y & \text{-- Synthesized attribute } y \text{ of child } x \\
 & \mid\ @x.\textbf{self} & \text{-- Reference to child } x \\
 & \mid\ @\textbf{loc}.y & \text{-- Local attribute } y
\end{array}
$$

**Fig. 4.** Syntax of minimal higher order attribute grammars

*Semantic rules* The semantic rules specify how the attribute values can be computed. For each production $P$ of nonterminal $N$ there are two types of attribute values to be defined: the synthesized attributes of $N$ and the inherited attributes of the children of $P$. The right-hand side of the rule can either be a reference to another attribute, or the instantiation of constructor $P$.

The keyword **lhs** means *left-hand side* and refers to the parent node in the tree. It is important to notice that **lhs** can be used in two different ways. When defining an attribute value $\textbf{lhs}.y = ...$ we refer to the synthesized attribute $y$ of the current node and pass that to the parent node. If we use the attribute reference $@\textbf{lhs}.y$ in the body of a semantic rule we refer to the inherited attribute $y$ of the current node, for which the value is defined by the parent node.

*Local attributes* Local attributes are a simple extension to the above to make the code more concise. A local attribute is an attribute that is both instantiated and used at a single production only, and can be simulated with an extra child per production called **loc** in which all inherited attribute values are returned directly as synthesized values.

*Self rules* Like local attributes, self rules are an extension to MHAGs to make the example somewhat simpler. A self rule can be implemented by means of a synthesized attribute that returns a copy of the constructor and all of its children. For efficiency, the self rules can be implemented in the host language in terms of a copy of or a reference to the child.

*Higher order children* Higher order children are declared and instantiated with the **inst** construct. The type of a higher order child must be a nonterminal name from the MHAG such that it can be decorated. The right-hand side of the instantiation is again either a reference to another attribute value or the instantiation of a constructor.

*Primitive functions* As we already explained we have added primitive functions and types to both $\lambda$ and MHAGs in this paper to make the example more concise and clear. A primitive function must be a first-order function with a type annotation, but in the example we leave out the type annotation. All its arguments must be given as attribute references, so no partial application is supported for primitive functions.

Note that HAGs can be embedded in MHAGs with the following method using primitive functions. Each semantic rule is assigned a unique name that will be the name of the primitive function. For each referenced attribute this primitive function gets a parameter and finally in the semantic rule this primitive function is called with the corresponding attribute references.

In this paper we assume whole program analysis in which all computations in MHAG scope and no primitive functions are used. However, for an actual MHAG implementations and for a concise example in this paper some primitive functions might be used, for which no optimization is performed.

## 4.3 Semantics

We informally give the semantics of MHAGs by describing a translation to $\lambda$. The translation that we give in this section relies on lazy evaluation. There exist more efficient translations for the wide class of ordered AGs, for example the one from Bransen et al. (2012). We illustrate the translation with Figure 2.

Translating MHAG data types to $\lambda$ data types is done in a straightforward way by erasing the names of the children. For each nonterminal an additional data type is generated for storing the values of all synthesized attributes of the nonterminal. This data type has a single constructor and this constructor has a child for each synthesized attribute. In the example *TreeSyn* is such a data type.

Also for each synthesized attribute a projection function is generated for retrieving the corresponding value from the data type containing the synthesized values. In the example *depth* and *dleaves* are such functions.

For each nonterminal and for each production we generate a semantic function. We call the semantic function for a nonterminal a *nonterminal semantic function* and a semantic function for a production a *production semantic function*.

The nonterminal semantic functions take a tree and the inherited attributes, and return the synthesized values. This is implemented by a pattern match on the value and calling the semantic function corresponding to that production. The children of the production are wrapped in a call to the corresponding nonterminal semantic function. In the example *semTree* is such a function. Note that the children of a primitive type are not wrapped but are directly passed to the production semantic function.

The production semantic functions take an argument for each child, which is a function from inherited to synthesized attributes, and its own inherited attributes. The body of the function is a recursive let in which both the inherited attributes of the children and the synthesized attributes of the production itself are computed. In the example, *semLeaf*, *semBin* and *semOrd* are such functions.

For each child, the child function is called once with the corresponding inherited attribute values. This is not visible in the example as the children do not have inherited attributes.

Note that this does not work for **self** rules, but that instead of wrapping the children in a call to the nonterminal semantic function in the nonterminal semantic function itself, we can pass the unwrapped version to the production semantic function and do the wrapping there. Then both the child itself as well as its synthesized attributes are in scope. However, when no **self** rules are involved we feel that the above presentation is easier.

For the instantiation of a higher order child the corresponding semantic function is called and the inherited attributes are passed in the same way as with regular children. However, the value of the higher order child itself can depend on inherited attributes of the production itself and synthesized attributes of its children.

## 4.4 Deepest leaves example

In Figure 5 we show the MHAG code that is the intermediate representation of the example. The first two data types are present in the original as well as in the final $\lambda$ code. The *Ordering′*

$$
\begin{array}{llll}
\mathbf{data}\ \mathit{Tree} & \mid \mathit{Leaf}\ n :: \widehat{\mathit{Int}} \mid \mathit{Bin}\ l :: \mathit{Tree}\ r :: \mathit{Tree} & & \\
\mathbf{data}\ \mathit{Ordering} & \mid \mathit{EQ} \mid \mathit{GT} \mid \mathit{LT} & & \\
\mathbf{data}\ \mathit{Ordering}' & \mid \mathit{EQ}' \mid \mathit{GT}' \mid \mathit{LT}' & & \\
\mathbf{data}\ \mathit{Ordering}'' & \mid \mathit{EQ}''\ \mathit{dl} :: \widehat{[\mathit{Int}]} \mid \mathit{GT}''\ \mathit{dl} :: \widehat{[\mathit{Int}]} \mid \mathit{LT}''\ \mathit{dl} :: \widehat{[\mathit{Int}]} & & \\
\mathbf{attr}\ \mathit{Tree} & \mathbf{syn}\ \mathit{depth} :: \widehat{\mathit{Int}}\ \ \mathbf{syn}\ \mathit{dleaves} :: \widehat{[\mathit{Int}]} & & \\
\mathbf{attr}\ \mathit{Ordering} & \mathbf{syn}\ \mathit{cmpres} :: \mathit{Ordering}' & & \\
\mathbf{attr}\ \mathit{Ordering}' & \mathbf{inh}\ \mathit{arg} :: \widehat{[\mathit{Int}]}\ \ \mathbf{syn}\ \mathit{res} :: \mathit{Ordering}'' & & \\
\mathbf{attr}\ \mathit{Ordering}'' & \mathbf{inh}\ \mathit{arg} :: \widehat{[\mathit{Int}]}\ \ \mathbf{syn}\ \mathit{res} :: \widehat{[\mathit{Int}]} & & \\
\mathbf{sem}\ \mathit{Tree} & \mid \mathit{Bin} & \mathbf{lhs}.\mathit{dleaves} & = @\mathit{ho}_3.\mathit{res} \\
 & & \mathbf{inst}.\mathit{ho}_1 & :: \mathit{Ordering} \\
 & & \mathbf{inst}.\mathit{ho}_1 & = \underline{\mathit{compare}}\ @l.\mathit{depth}\ @r.\mathit{depth} \\
 & & \mathbf{inst}.\mathit{ho}_2 & :: \mathit{Ordering}' \\
 & & \mathbf{inst}.\mathit{ho}_2 & = @\mathit{ho}_1.\mathit{cmpres} \\
 & & \mathit{ho}_2.\mathit{arg} & = @l.\mathit{dleaves} \\
 & & \mathbf{inst}.\mathit{ho}_3 & :: \mathit{Ordering}'' \\
 & & \mathbf{inst}.\mathit{ho}_3 & = @\mathit{ho}_2.\mathit{res} \\
 & & \mathit{ho}_3.\mathit{arg} & = @r.\mathit{dleaves} \\
 & & \mathbf{lhs}.\mathit{depth} & = \underline{\mathit{succ}}\ @\mathbf{loc}.a_1 \\
 & & \mathbf{loc}.a_1 & :: \widehat{\mathit{Int}} \\
 & & \mathbf{loc}.a_1 & = \underline{\mathit{max}}\ @l.\mathit{depth}\ @r.\mathit{depth} \\
 & \mid \mathit{Leaf} & \mathbf{lhs}.\mathit{dleaves} & = \underline{\mathit{singleton}}\ @n.\mathbf{self} \\
 & & \mathbf{lhs}.\mathit{depth} & = \underline{0} \\
\mathbf{sem}\ \mathit{Ordering} & \mid \mathit{EQ} & \mathbf{lhs}.\mathit{res} & = \mathit{EQ}' \\
 & \mid \mathit{GT} & \mathbf{lhs}.\mathit{res} & = \mathit{GT}' \\
 & \mid \mathit{LT} & \mathbf{lhs}.\mathit{res} & = \mathit{LT}' \\
\mathbf{sem}\ \mathit{Ordering}' & \mid \mathit{EQ}' & \mathbf{lhs}.\mathit{res} & = \mathit{EQ}''\ @\mathbf{lhs}.\mathit{arg} \\
 & \mid \mathit{GT}' & \mathbf{lhs}.\mathit{res} & = \mathit{GT}''\ @\mathbf{lhs}.\mathit{arg} \\
 & \mid \mathit{LT}' & \mathbf{lhs}.\mathit{res} & = \mathit{LT}''\ @\mathbf{lhs}.\mathit{arg} \\
\mathbf{sem}\ \mathit{Ordering}'' & \mid \mathit{EQ}'' & \mathbf{lhs}.\mathit{res} & = @\mathit{dl}.\mathbf{self}\ \underline{+\!\!+}\ @\mathbf{lhs}.\mathit{arg} \\
 & \mid \mathit{GT}'' & \mathbf{lhs}.\mathit{res} & = @\mathit{dl}.\mathbf{self} \\
 & \mid \mathit{LT}'' & \mathbf{lhs}.\mathit{res} & = @\mathbf{lhs}.\mathit{arg}
\end{array}
$$

**Fig. 5.** MHAG code corresponding to deepest leaves example

and *Ordering''* data types are intermediate data types that are generated as the result of the translation process for storing intermediate values.

The *Tree* data type has, as expected, two synthesized attributes: *depth* and *dleaves*. The *Ordering* data type has a synthesized attribute *cmpres* and returns a value of type *Ordering'*. This *Ordering'* has an inherited attribute *arg* (the function argument) and returns an *Ordering''*, which again takes an inherited attribute *arg* and finally returns a synthesized attribute *res* of type $\widehat{[\mathit{Int}]}$. This pattern can be thought of as a function taking two $\widehat{[\mathit{Int}]}$ arguments and returning an $\widehat{[\mathit{Int}]}$. In the resulting $\lambda$ code as presented in Figure 2 the semantic functions for *Ordering*, *Ordering'* and *Ordering''* are combined and called *semOrd*.

This indirection is a result of the way the translation scheme is formulated but can be easily avoided. We have decided to keep the example results close to the actual translation scheme as it is presented later to help the reader in understanding the described approach.

The semantic rules for the *Leaf* case are straightforward and directly follow the structure of the original functions. In the *Bin* case, the computation of *depth* is also a direct mapping from the original. The small difference is that in MHAGs the right-hand side of an expression can only be exactly one primitive function call, so in order to compose functions the intermediate value has to be given a name explicitly ($\mathbf{loc}.a_1$ in this case).

For the computation of *dleaves* the result of the $\underline{\mathit{compare}}$ function is instantiated as a higher order child $\mathit{ho}_1$. The synthesized attribute *cmpres* represents the function that, given the deepest leaves of the left and the right child, returns the deepest leaves. This value is instantiated as a higher order child $ho_2$ and the deepest leaves of the left child are passed as an argument. The result is again instantiated as a higher order child $ho_3$ and the deepest leaves of the right child are passed as the second argument. The synthesized *dleaves* value is finally the result of $ho_3$.

## 5 Translation

The translation from $\lambda$ to MHAG consists of several translation steps, which are described in the following sections. We start with an overview of the translation process in Section 5.1. We then identify a set of top level declarations (Section 5.2) and describe how the code can be translated to a form that is suitable for the translation (Section 5.3). In Section 5.4 we transform the $\lambda$ expressions further to obtain even better sharing behavior. We use a form of lambda lifting (Section 5.5) as a necessary step to make sure that all variables are in scope after translating to MHAGs. Finally in Section 5.6 we formally define the actual translation.

### 5.1 Overview

Before going into details we give an intuitive description of the construction used to translate $\lambda$ expressions to MHAGs. We assume that in all cases the initial $\lambda$ expression is well-typed. We illustrate the overview with the MHAG code for the example in Figure 5.

Functions are represented by trees which have one inherited attribute for the argument and one synthesized attribute for the result. In the example, $Ordering'$ and $Ordering''$ are such trees. Note that *depth* and *dleaves* are special functions that are handled in a different way which is explained in Section 5.3.

We introduce a new nonterminal for every function type, and a new constructor is added to the corresponding nonterminal for each lambda abstraction. In the example $EQ'$, $GT'$, $LT'$, $EQ''$, $GT''$ and $LT''$ are such constructors. Note that the lambda abstractions corresponding to these constructors are not visible in the original $\lambda$ code but are introduced in the step described in Section 5.5.

For each free variable in the body of the abstraction we add a child to the corresponding constructor to store the value of the corresponding variable. In the example *dl* is such a child, which is introduced for all constructors of $Ordering''$.

For each case abstraction a synthesized attribute is added to the corresponding nonterminal. In the example *depth*, *dleaves* and *cmpres* are such synthesized attributes.

Evaluation proceeds as follows. For a variable there is an attribute in scope containing the value of the attribute. For an application, the function is first evaluated and its result, which is a value of a nonterminal corresponding to the function type, is instantiated as higher order child. The result of the evaluation of the argument is then passed as the inherited value of the new higher order child and the synthesized attribute of the higher order child is returned as return value. In the example the higher order children $ho_2$ and $ho_3$ are introduced to represent function application.

For abstractions the corresponding constructor is returned, for example in the semantic rules of *Ordering*. For case distinction, the argument is instantiated as a higher order child and the result is the corresponding synthesized argument of this child. In the example this is the case for $ho_1$ and its synthesized attribute *cmpres*.

### 5.2 Top level declarations

We define top level declarations as the set of variables that are bound by a recursive let not inside an abstraction or case distinction. In other words, a top level declaration is a let bound variable such that all its free variables are also top level declarations. In our example, the top level declarations are *depth* and *dleaves*.

We let top level declarations be available everywhere, and thus depend on the MHAG semantics to *tie the knot*. Although this is not an essential step in the translation process, this can avoid explosion of the size of the resulting MHAG because otherwise all top level declarations are passed around explicitly at every function call.

### 5.3 Recognizing folds

We define *folds* to be expressions of the form

$$\lambda t : \tau.\ \textbf{case}\ t\ \textbf{of}\ as$$

such that the set of free variables of the alternatives $as$ is empty. Intuitively, such a computation can be written in MHAGs using a synthesized attribute, and thus can be efficiently used to achieve better sharing behavior.

The success of the overall approach depends on the number of functions that can be efficiently represented in MHAGs. It is therefore important that the functions in the input are written as folds or can automatically be transformed into folds.

Automatically transforming the input into this form is a separate problem that we do not solve in the present paper. However, with some simple rules like the following one can already achieve good results. When $u \not\equiv t$ the $\lambda u : \tau$ can be moved inside the alternatives:

$$\begin{array}{lcl} \lambda u : \tau.\ \textbf{case}\ t\ \textbf{of} & & \textbf{case}\ t\ \textbf{of} \\ \quad p_1 \rightarrow e_1 & \rightsquigarrow & \quad p_1 \rightarrow \lambda u : \tau.\ e_1 \\ \quad p_n \rightarrow e_n & & \quad p_n \rightarrow \lambda u : \tau.\ e_n \end{array}$$

### 5.4 Identifying recursive calls

A call of the form $f\ a$ where $a$ is a variable and $f$ is a fold should have special treatment. This is where the better sharing behavior is obtained. Intuitively, usually a function is represented by some tree structure in the MHAG code. For evaluating the application the function is instantiated as a higher order child and its argument is passed as an inherited attribute.

In the case that the function is a fold, its argument is a tree itself, and thus a copy of this tree needs to be passed as inherited attribute. However, this copy of the tree is then immediately instantiated as a higher order child to retrieve the synthesized attribute for the case distinction. But when the argument is a child of the current node, the synthesized attribute for the case distinction is already available. Furthermore, when multiple computations use the same synthesized attribute, for example when the same function is called multiple times on the same argument, the result is now shared.

### 5.5 Lambda lifting in case alternatives

Case expressions are translated by instantiating the expression as a higher order child and adding the semantics of the alternatives to the data type using a synthesized attribute. The result is that the semantic functions of the alternatives are in a different context, and thus the variables that are bound by the environment of the case are not in scope anymore.

The solution for this is to perform lambda lifting of the alternatives. First all free variables of the alternatives together are gathered. Then a lambda abstraction is introduced in each alternative for each free variable. Finally, the variables that contain the values in the context of the case are applied to the result of the case expression.

As an example, consider the following expression, where $v_1 : \tau_1$ and $v_2 : \tau_2$ are bound in the context of the case:

$$\begin{array}{l} \textbf{case}\ e\ \textbf{of}\ C_1 \ldots \rightarrow \ldots v_1 \ldots \\ \qquad\qquad\ C_2 \ldots \rightarrow \ldots v_2 \ldots v_1 \ldots \end{array}$$

This example is translated to the following, operationally equivalent, code:

$$\begin{array}{l} (\textbf{case}\ e\ \textbf{of}\ C_1 \ldots \rightarrow \lambda v_1 : \tau_1.\ \lambda v_2 : \tau_2.\ \ldots\ v_1 \ldots \\ \qquad\qquad\quad C_2 \ldots \rightarrow \lambda v_1 : \tau_1.\ \lambda v_2 : \tau_2.\ \ldots\ v_2 \ldots v_1 \ldots)\ v_1\ v_2 \end{array}$$

For recursive calls of the form $f\ \ a$ as identified in Section 5.4 where $a$ is a free variable in the alternative, a specialized rule is defined. Instead of abstracting over the free variable itself, we abstract over $f\ \ a$. This specialized rule will make sure that whenever a folding function is called on a substructure, this fold will not be performed again. If we would not have this specialized rule it could be the case that a copy of the substructure needs to be passed to one of the case alternatives after which it is instantiated again as a higher-order child.

### 5.6 Translation rules

In Figure 6 we show the translation rules. The translation scheme is of the form $\Gamma; \Delta \vdash^{P}_{N} e \rightsquigarrow \varphi; \psi$ and consists of seven elements; $\Gamma$ is the environment in which $\lambda$ variables are mapped to attribute references, $\Delta$ is an application context containing an ordered list of attribute references that still need to be applied, $P$ and $N$ are the current production and nonterminal respectively, $e$ is the $\lambda$ expression that is being translated, $\varphi$ is the resulting attribute reference and finally $\psi$ is the resulting MHAG.

Some details in the rules have been omitted. In the translation process a nonterminal is introduced for every function type in the $\lambda$ expression. The function *ntp* is used in the translation rules to get the nonterminal name for the given expression. However, as the expression might contain variables that are bound outside this sub expression, this should actually also depend on some environment containing all bound variables and types. As this environment would need to be passed around in all rules but is not an essential part of the translation itself, we have left it out.

In the generated MHAG code type signatures for the **inst** and **loc** cases are left out. Again, the type information can in all cases be inferred, but is left out to keep translation rules simpler.

Only the MHAG code for generating semantic terms is shown. For the generated MHAG code multiple **sem** blocks might be generated for a single production. We assume a post-processing step in which these **sem** blocks are all combined into a complete MHAG definition and leave the generation of the data types corresponding to the $\lambda$ data types and the attribute declarations implicit.

*Variables* In case of a variable and an empty application context the rule VAR.E is used, which performs a lookup of the variable in the environment.

In case of a nonempty application context, the rule VAR.A is used. In this case the variable represents a function and can thus be instantiated as a higher order child. The argument of the function, which is the first value from the application context, is passed as an inherited attribute *arg*. The result of the application is returned in the synthesized attribute *res*.

*Lambda abstraction* For a lambda abstraction and a nonempty application context the rule LAM.A is used to bind the abstracted variable to the variable reference from the application context. This is essentially $\beta$-reduction, however it is important to note that sharing is retained.

For an empty application context, the rule LAM.E constructs a new production $P_{\alpha}$ representing the abstraction. The function *fv* returns all free variables of the expression. The set of free variables returned by *fv* never contains *top level declarations* as defined in Section 5.2. For this an environment containing all top level declarations needs to be available, which again we leave implicit.

For each of the free variables a child is introduced to store the value of the variable, and the environment for the translation of the rest of the expression contains exactly these variables. The return value is the instantiation of the new production with the corresponding values of the variables as children.

*Application* The rule APP translates the arguments of the application and adds the resulting attribute reference to the application context. In the translation of $e_1$ this might result in instantiating $e_1$ as a higher order child and pass the argument value as an inherited attribute.

$$\frac{}{\Gamma; \epsilon \vdash^{P}_{N} x \rightsquigarrow \Gamma\ (x); \epsilon}\ \textsc{var.e} \qquad \frac{[x \mapsto y]\ \Gamma; \Delta \vdash^{P}_{N} e \rightsquigarrow \varphi; \psi}{\Gamma; y\ \Delta \vdash^{P}_{N} \lambda x : \tau . e \rightsquigarrow \varphi; \psi}\ \textsc{lam.a}$$

$$\frac{\begin{array}{c} \Gamma; \epsilon \vdash^{P}_{N} e_2 \rightsquigarrow \varphi'; \psi' \\ \Gamma; \varphi'\ \Delta \vdash^{P}_{N} e_1 \rightsquigarrow \varphi; \psi \end{array}}{\Gamma; \Delta \vdash^{P}_{N} e_1\ e_2 \rightsquigarrow \varphi; \psi\ \psi'}\ \textsc{app} \qquad \frac{\begin{array}{c} \psi'' = \boxed{\mathbf{data}\ N' \mid P_\alpha\ x_1 \ldots x_n} \\ \Gamma'; \epsilon \vdash^{P_\alpha}_{N'} e \rightsquigarrow \varphi'; \psi' \\ x_1 \ldots x_n = \mathit{fv}\ (e) \\ \Gamma' = [x \mapsto @\mathbf{loc}.\alpha \ldots, x_n \mapsto @x_n.\mathbf{self}] \\ N' = \mathit{ntp}\ (e) \\ \alpha, \beta \text{ fresh} \\ \psi = \boxed{\begin{array}{l} \mathbf{sem}\ N \mid P \\ \quad \mathbf{loc}.\beta = P_\alpha\ \Gamma\ (x_1) \ldots \Gamma\ (x_n) \end{array}} \end{array}}{\Gamma; \epsilon \vdash^{P}_{N} \lambda x : \tau . e \rightsquigarrow \varphi; \psi\ \psi'\ \psi''}\ \textsc{lam.e}$$

$$\frac{\begin{array}{c} \alpha \text{ fresh} \\ \psi = \boxed{\begin{array}{l} \mathbf{sem}\ N \mid P \\ \quad \mathbf{inst}.\alpha = \varphi' \\ \quad \alpha.\mathit{arg} = y \end{array}} \\ \Gamma; \Delta \vdash^{P}_{N} x \rightsquigarrow \varphi'; \psi' \end{array}}{\Gamma; y\ \Delta \vdash^{P}_{N} x \rightsquigarrow @\alpha.\mathit{res}; \psi\ \psi'}\ \textsc{var.a} \qquad \frac{\begin{array}{c} \alpha, \beta \text{ fresh} \\ \psi = \boxed{\begin{array}{l} \mathbf{sem}\ N \mid P \\ \quad \mathbf{inst}.\alpha = \varphi' \\ \quad \alpha.\mathit{arg} = y \end{array}} \\ \Gamma; \Delta \vdash^{P}_{N} \mathbf{case}\ e\ \mathbf{of}\ \mathit{as} \rightsquigarrow \varphi'; \psi' \end{array}}{\Gamma; y\ \Delta \vdash^{P}_{N} \mathbf{case}\ e\ \mathbf{of}\ \mathit{as} \rightsquigarrow @\alpha.\mathit{res}; \psi\ \psi'}\ \textsc{case.a}$$

$$\frac{\begin{array}{c} \alpha, \beta \text{ fresh} \\ (P_1\, a_1^1 \ldots a_1^k \to e_1) \ldots (P_n\ a_n^1 \ldots a_n^m \to e_n) = \mathit{as} \\ \psi'' = \boxed{\begin{array}{l} \mathbf{sem}\ N' \mid P_i \\ \quad \mathbf{lhs}.\beta = \varphi_i \end{array}} \\ \Gamma'_i; \epsilon \vdash^{P_i}_{N'} e_i \rightsquigarrow \varphi_i; \psi_i \\ \Gamma'_i = [a_i^1 \mapsto @\mathit{ch}_1.\mathbf{self}, \ldots, a_i^k \mapsto @\mathit{ch}_k.\mathbf{self}]\ \Gamma \\ \Gamma; \epsilon \vdash^{P}_{N} e \rightsquigarrow \varphi'; \psi' \\ N' = \mathit{ntp}\ (e) \\ \psi = \boxed{\begin{array}{l} \mathbf{sem}\ N \mid P \\ \quad \mathbf{inst}.\alpha = \varphi' \end{array}} \end{array}}{\Gamma; \epsilon \vdash^{P}_{N} \mathbf{case}\ e\ \mathbf{of}\ \mathit{as} \rightsquigarrow @\alpha.\beta; \psi\ \psi'\ \psi''\ \psi_1 \ldots \psi_n}\ \textsc{case.e}$$

$$\frac{\begin{array}{c} \alpha \text{ fresh} \\ \psi = \boxed{\begin{array}{l} \mathbf{sem}\ N \mid P \\ \quad \mathbf{loc}.\alpha = C\ y_1 \ldots y_n \end{array}} \end{array}}{\Gamma; y_1 \ldots y_n \vdash^{P}_{N} C \rightsquigarrow @\mathbf{loc}.\alpha; \psi}\ \textsc{con} \qquad \frac{\begin{array}{c} \alpha \text{ fresh} \\ \psi = \boxed{\begin{array}{l} \mathbf{sem}\ N \mid P \\ \quad \mathbf{loc}.\alpha = \underline{f}\ y_1 \ldots y_n \end{array}} \end{array}}{\Gamma; y_1 \ldots y_n \vdash^{P}_{N} \underline{f} \rightsquigarrow @\mathbf{loc}.\alpha; \psi}\ \textsc{prim}$$

$$\frac{\begin{array}{c} f \text{ is a fold} \\ \beta = \mathit{fsa}\ (f) \\ \Gamma; \epsilon \vdash^{P}_{N} a \rightsquigarrow @\alpha.\mathbf{self}; \psi \end{array}}{\Gamma; \epsilon \vdash^{P}_{N} f\ a \rightsquigarrow @\alpha.\beta; \psi}\ \textsc{fold} \qquad \frac{\begin{array}{c} (f_1 = e_1) \ldots (f_n = e_n) = \mathit{fs} \\ \Gamma'; \Delta \vdash^{P}_{N} e \rightsquigarrow \varphi; \psi \\ \Gamma' = [f_1 \mapsto \varphi_1, \ldots, f_n \mapsto \varphi_n]\ \Gamma \\ \Gamma'; \epsilon \vdash^{P}_{N} e_i \rightsquigarrow \varphi_i; \psi_i \end{array}}{\Gamma; \Delta \vdash^{P}_{N} \mathbf{let}\ \mathit{fs}\ \mathbf{in}\ e \rightsquigarrow \varphi; \psi\ \psi_1 \ldots \psi_n}\ \textsc{let}$$

**Fig. 6.** Translation from $\lambda$ to MHAG

*Recursive let* For translating a set of recursive let definitions, the rule LET translates each definition and adds all resulting attribute references to the environment which is passed to all definitions as well as the body of the let. The resulting attribute reference of the body is used as the result of the translation of the let.

*Case distinction* For case distinction there are two rules. The rule CASE.A is used in case of a nonempty application context. This rule is similar to the rule VAR.A and passes the attributes from the application context one by one as an inherited attribute to the result of the translation of the case itself.

The rule CASE.E is the most complicated one. A case distinction is implemented by introducing a new synthesized attribute $\beta$ for the result. The result of the expression itself is instantiated as a higher order child $\alpha$ and the semantics of the alternatives are added to the corresponding productions.

*Constructor* For a constructor the rule CON instantiates this constructor with the application context. The type correctness of the $\lambda$ expression guarantees that the number of arguments in the application context is exactly the number of children for this constructor.

*Primitive functions* Primitive function calls are translated by the rule PRIM to MHAG primitive function calls. When the input $\lambda$ expression is type correct, we know that the number of arguments that are applied to the function are exactly the number of arguments that the primitive function takes. Hence, we take the full application context as the arguments for the primitive function.

*Special rules for folds* This set of rules is an embedding of $\lambda$ expressions into MHAGs. However, to obtain better sharing behavior the rule FOLD is a specialized rule for the recursive positions as identified in Section 5.4. As folds are implemented by case distinctions and case distinctions are translated to a synthesized attribute, the result of a fold call is the value of the corresponding synthesized attribute. The function *fsa* is a lookup function for finding the name of the synthesized attribute corresponding to the fold.

## 6 Implementation

We have created a toy implementation of the translation using AGs. Using this implementation we have been able to verify that the approach works as expected for some examples. Furthermore, we have run some simple benchmarks to evaluate the effectiveness of the approach and the possible overhead that is introduced when no tupling can be done.

| | Original implementation | After translation |
|---|---|---|
| Deepest leaves | 834.1 ms | 4.063 ms |
| Prolog interpreter | 937.0 ms | 1337.0 ms |

**Table 1.** Benchmark results

In Table 1 we show the results of two benchmarks. To run the benchmarks, the $\lambda$ code is translated to Haskell using a straightforward translation. The Haskell code is compiled with GHC version 7.4.1 with the `-O2` flag and timings are collected with Criterion version 0.6.1.1 with 100 samples per benchmark.

The deepest leaves benchmark is the example from this paper and is run on a tree with 10,000 leaves. It can be seen that the resulting code is about 200 times faster than the original code.

The Prolog interpreter is a real-world example from the Nofib benchmark (Partain, 1993). Our $\lambda$ implementation of this interpreter has been kept as close to the original code as possible, with the exception of polymorphic functions that had to be re-implemented for each different type. In this benchmark we use the Prolog interpreter to append a (Prolog) list with one element to the end of a list with 1000 elements.

In this case no extra sharing can be achieved. However, the translation will introduce extra data types which could lead to extra overhead. The benchmark shows that in this case the resulting code is about 1.5 times slower. Considering the fact that this is only a toy implementation for which no work has been done to optimize the resulting code, we believe the overhead is minor.

## 7 Related work

Automatic tupling is a well-studied subject so there is a lot of related work that could be included in this section. It would be hard to give a complete overview of all work related to automatic tupling and it is not our intention to do so. However, we try to highlight some of the most important ones.

One of the earlier works in this category is that of Burstall and Darlington (1977) describing the so called *fold/unfold transformation*. They rely on the automatic construction of *eureka* tuples, which is further explored in Chin (1993). The problem of this approach is that it is hard to avoid infinite unfolding.

A classic example of tupling is from Bird (1984) where two traversals over a tree are merged into one. The essence is that the second traversal depends on the first one, which makes the tupling less straightforward. Our approach works fine in such cases too.

The relevance of attribute grammars in general is described in Swierstra (2005). The example used is a typical case where attribute grammars are useful, and the technique we have presented in this paper can be used to handle other parts of a program that cannot intuitively be expressed as an attribute grammar.

In Hu et al. (1997) a solution is described for finding the appropriate tuples of calls for automatic tupling. Their work forms the basis of the present paper, where we assume that the programs are already in a suitable form.

For functional-logic programs the fold/unfold transformation for tupling is explored in Moreno (2002). This leads to a practical approach for optimizing functional-logic programs.

From Kuiper and Swierstra (1987) we know that we can derive efficient programs from attribute grammars, the inverse of the technique described in the present paper. Combining that approach with our current approach leads to a transformation from a functional program to another functional program with better sharing behavior. A similar approach is described in Johnsson (1987).

How to obtain an incremental evaluation algorithm for ordered AGs is described in Yeh and Kastens (1988). The class of ordered AGs is a large subclass of AGs and we believe that most AGs resulting from our translation belong to this class. Thus, combining this with our automatic translation could result in automatic generation of incremental evaluation of functional programs.

## 8 Discussion

The given approach uses whole program analysis to optimize the code and obtain better sharing behavior. The downside of whole program analysis is that a form of compositionality is lost. When functions are optimized separately and then composed, the extra sharing behavior is not obtained. However, this is also the case for the $\lambda$ language so our approach does not make matters worse.

The $\lambda$ language that we defined in this paper is a simply typed language and thus does not support polymorphism. In order for this work to be applied in real applications support for polymorphic functions must be added. Furthermore, depending on the exact application, more advanced features like Haskell's type classes need to be incorporated. We believe that this not lead to fundamental problems. Incorporating these features in HAGs has already been investigated Middelkoop et al. (2012) and the same approach could be used for MHAGs.

The language $\lambda$ with support for polymorphism is similar to the core language of the *Glasgow Haskell Compiler* (GHC). Combining our approach with GHC would make it possible to apply our optimization to any Haskell program.

Our approach leads to better sharing in the cases described and leaves the complexity of other cases unchanged. However, due to extra indirections and tupling the approach could lead to a constant overhead for cases where no better sharing can be achieved. The preliminary benchmark results show that the overhead is small compared to the overall running time. The approach could also lead to fewer pattern matches and thus lead to a constant improvement even when no tupling is performed. Furthermore, using the approach described in Bransen et al. (2012) only attribute values that are required need to be evaluated, which can also remove the potential overhead.

The given translation from $\lambda$ to $\lambda$ should be *semantics preserving*. We have not defined the semantics of $\lambda$ and MHAGs, nor have we proved that the translation does preserve semantics. It is however not hard to see that the translation is *complete* with respect to the type system, so for every well-typed $\lambda$ expression the translation will construct another well-typed $\lambda$ expression. Only for the rule LET this is not obvious because the environment passed to each definition depends on the result of the translation of these definitions. Note however that we can introduce a fresh local attribute for each definition that we store in the environment and then assign the result of each translation to this fresh variable name.

## 9 Conclusion and future work

In this paper we described a novel technique for translating expressions from a functional language to attribute grammars. We have combined this translation with existing techniques for translating attribute grammars to a functional language, and we have showed that this leads to an approach for automatic tupling. Using an example we have showed that this can give superlinear speedups.

We plan to add polymorphism to the translation to support a wider class of functional programs. Furthermore, we would like to implement the translation in the context of the Utrecht Haskell Compiler Dijkstra et al. (2009) to be able to experiment and benchmark. We also plan to investigate what other existing attribute grammar optimization techniques can be combined with our approach.

As a concluding remark, we believe that the approach described in the current paper lifts attribute grammar techniques to the functional programming world, thereby giving rise to new optimization possibilities.

## Bibliography

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

Bransen, J., Middelkoop, A., Dijkstra, A., and Swierstra, S. D. (2012). The Kennedy-Warren algorithm revisited: ordering Attribute Grammars. In Russo, C. and Zhou, N.-F., editors, *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin / Heidelberg.

Burstall, R. M. and Darlington, J. (1977). A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67.

Chin, W.-N. (1993). Towards an automated tupling strategy. In *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '93, pages 119–132, New York, NY, USA. ACM.

Dijkstra, A., Fokker, J., and Swierstra, S. D. (2009). The architecture of the Utrecht Haskell Compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA. ACM.

Ekman, T. and Hedin, G. (2007). The JastAdd extensible Java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA. ACM.

Hu, Z., Iwasaki, H., Takeichi, M., and Takano, A. (1997). Tupling calculation eliminates multiple data traversals. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 164–175, New York, NY, USA. ACM.

Johnsson, T. (1987). Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, volume 274 of LNCS*, pages 154–173. Springer-Verlag.

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

Kuiper, M. F. and Swierstra, S. D. (1987). Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN'87*.

Middelkoop, A., Bransen, J., Dijkstra, A., and Swierstra, S. D. (2012). Merging idiomatic haskell with attribute grammars. To be published.

Moreno, G. (2002). Automatic tupling for functional-logic programs. Technical report, Dep. Informática, UCLM, Albacete.

Partain, W. (1993). The nofib benchmark suite of haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, UK. Springer-Verlag.

Swierstra, S. D., Alcocer, P. R. A., and Saraiva, J. (1998). Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, pages 150–206.

Swierstra, W. (2005). Why Attribute Grammars Matter. *The Monad.Reader*, Issue Four.

Vogt, H. H., Swierstra, S. D., and Kuiper, M. F. (1989). Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, volume 24 of *PLDI '89*, pages 131–145, New York, NY, USA. ACM.

Yeh, D. and Kastens, U. (1988). Improvements of an incremental evaluation algorithm for ordered attribute grammars. *SIGPLAN Not.*, 23(12):45–50.