INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Structure-directed Genericity in Functional Programming and Attribute Grammars

Etienne DURIS, Didier PARIGOT, Gilles ROUSSEL, Martin JOURDAN

N° 3105

Février 1997

THÈME 2

Rapport de recherche

ISSN 0249-6399

Structure-directed Genericity in Functional Programming and Attribute Grammars

Etienne Duris, Didier Parigot, Gilles Roussel, Martin Jourdan

Thème 2 — Génie logiciel
et calcul symbolique
Projet Oscar

Rapport de recherche n° 3105 — Février 1997 — 15 pages

Abstract: Generic control operators, such as *fold*, have been introduced in functional programming to increase the power and applicability of data-structure-based transformations. This is achieved by making the structure of the data more explicit in program specifications.

We argue that this very important property is one of the original concepts of attribute grammars. In this paper, we present the similarities between the *fold* formalism and attribute grammars. In particular, we show the equivalence of their respective deforestation methods.

Given these results and the fundamental role of deforestation in the concept of *structure-directed genericity*, first devised for attribute grammars with descriptional composition, we show how the *fold* operator with its fusion method allow us to transport this concept in the area of functional programming.

Key-words: Attribute grammars, static analysis, functional programming, structure-directed programming, program transformation, deforestation, genericity.

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Généricité dirigée par la structure dans la programmation fonctionnelle et dans les grammaires attribuées

Résumé : Les opérateurs de contrôle génériques tels que *fold* ont été introduits dans la programmation fonctionnelle pour augmenter la puissance et le champs d'application des transformations basées sur la structure des données. Ceci est obtenu en rendant la structure des données plus explicite dans la spécification des programmes.

Nous considérons que cette caractéristique fondamentale est l'un des concepts de base des grammaires attribuées. Dans cet article, nous présentons les similarités entre le formalisme des *folds* et celui des grammaires attribuées et nous montrons en particulier l'équivalence de leurs méthodes de déforestation respectives.

Étant donné ces résultats et le rôle fondamental de la déforestation dans le concept de *généricité dirigée par la structure*, initialement défini pour les grammaires attribuées avec la composition descriptionnelle, nous montrons comment l'opérateur fold, muni de sa méthode de fusion, permet de transporter ce concept dans le cadre de la programmation fonctionnelle.

Mots-clé : Grammaires attribuées, analyse statique, programmation fonctionnelle, programmation dirigée par la structure, transformation de programmes, déforestation, généricité.

1 Introduction

For the last few years, the functional-programming community has been studying various techniques to optimize programs, in particular to eliminate "useless" intermediate data structures occurring in function composition. A symbolic approach, called *deforestation*, has been proposed [Wad88]; it was mainly based on the "fold and unfold" transformations first introduced in [BD77]. Later, several different formalisms have been introduced to extend the power of these kind of functional-program transformations [MFP91, FSS92, GLJ93, SF93, FSZ94, LS95], and many of these approaches (often called *fusion*) are based on the *structure-directed programming style* (e.g. the *fold* generic control operator), which is attracting a growing interest in the functional-programming community because it allows us to capture patterns of recursion for large classes of types in a uniform way.

On the other hand, the problem of eliminating intermediate data structures occurring in function composition has been studied rather extensively in the context of another well-known structure- or syntax-directed programming paradigm, namely Attribute Grammars (AGs) [Knu68, Paa95]: Ganzinger and Giegerich solved it by introducing their *descriptional composition* (DC) algorithm [GG84].

However, for a couple of years, the most interesting application of DC has not been for optimization but for genericity. Indeed, *structure-directed genericity* [FMY92, LJPR93, BG94] is based on the observation that you can instantiate (reuse) onto type (grammar) T_1 any function (AG) f defined on type T_2 if you write a *coupling* function that transforms every structure of type T_1 into the "equivalent" structure of type T_2. DC revealed itself as the method of choice to perform these instantiations, by eliminating type T_2 from the resulting function specification. Experience with structure-directed genericity in AGs shows that it is quite flexible and well suited to large, complex applications (e.g. compiler construction) [FMY92]. In this paper, hence, we establish the base for transporting this concept to the functional programming paradigm.

To reach this goal, it is first necessary to identify the fundamental similarities between AGs and structure-directed functional programming. Here, it is sufficient to consider the *fold* formalism introduced by Sheard and Fegaras [SF93], which provides all the features (structure-directed approach and a fusion method) necessary for our purpose.

In section 2, we will show that a fold program can easily be translated into a purely-synthesized AG over the corresponding underlying grammar (see also the relation between Catamorphisms and Attribute Grammars [FJMM91]). The next section will discuss semantics and evaluation methods. The main result is that the functors used to define the semantics of a fold function also define its evaluation algorithm, whereas the semantics of an AG is more declarative.

In section 4, we study the respective deforestation techniques of the fold formalism and AGs. In the context of first-order fold functions and purely-synthesized AGs, we will show that the Normalization Algorithm (NA) [SF93] and DC produce the same results. The fundamental observation is that DC and NA have the same local-transformation property. However, in the same way that an AG is more declarative than a fold, DC is a more symbolic transformation than NA.

$$\begin{aligned}\text{length(x)} = \text{fold}^{list}(&\lambda().\text{Zero},\\ &\lambda(\text{a,r}).\text{Succ(r)}\ )\ \text{x}\end{aligned}$$

Figure 1: Definition of *length* with fold^{list}

```
Nil ->
```
$f_{Nil,s_{\uparrow Nil}}$: $s_{\uparrow Nil} = \text{Zero}$ i.e. $(\lambda().\text{Zero})()$
```
Cons -> a list
```
$f_{Cons,s_{\uparrow Cons}}$: $s_{\uparrow Cons} = \text{Succ}(s_{\uparrow list})$ i.e. $(\lambda(\text{a,r}).\text{Succ(r)})(\text{a},s_{\uparrow list})$

Figure 2: Definition of *length* with an AG

In section 5, we will briefly consider the most general case, with higher-order folds on one hand and AGs with inherited attributes on the other hand: both provide for more powerful deforestation. But using higher-order attributes and semantic rules is not a natural approach with AGs; rather, they use inherited attributes, which enable to specify top-down computations over a recursive structure. We will not discuss this topic in depth, because it is not crucial for structure-directed genericity and because we have only partial results; see [DPRJ96] for more details.

Before concluding, section 6 gives some insights and ideas about the different genericity concepts in the areas of functional programming and attribute grammars. We will first show that the "manual" approach to structure-directed genericity [FMY92] is indeed feasible in functional programming, through the use of fold specifications for functions to reuse and coupling functions. We will also compare polymorphism and structure-directed genericity and discuss their respective evolutions.

2 Notations

In [SF93], *fold* generic control operators are defined over mutually recursive types (algebraic type definitions), which show explicitly the type constructors. Historically, AGs are based on the *Context-Free Grammar* notion, but can also be defined on such algebraic type definitions [CM79, GG84] in which the type constructors play the role of productions.

Over the simple recursive type *list*, the classical function *length* can be defined using the generic control operator *fold* for this type (Fig. 1). The two lambda-expressions in this definition are called *accumulating functions* and represent the computations to perform over each constructor of the type *list*: one for the *Nil* constructor, with no parameter, and the other for the *Cons* constructor, with two parameters. These parameters are provided by the generic definition of fold^{list}, which will be presented in Def. 3.1.

An AG specification over an algebraic type and a given set of attributes is a set of equations over attribute occurrences for each type constructor (production). There are two kinds of attributes: *synthesized* (noted with $_\uparrow$) and *inherited* (noted with $_\downarrow$). For instance, function *length* can be defined by the AG specification in Fig. 2, where the single attribute $s_\uparrow$ is synthesized. The functions $f_{Nil,s_{\uparrow_{Nil}}}$ and $f_{Cons,s_{\uparrow_{Cons}}}$ are the semantic rules for the *Nil* and *Cons* constructors. They exactly correspond to the accumulative functions in the fold formalism and, as shown in Fig. 2, they can easily be expressed with (the same) lambda-expressions. For easier comparison, in the sequel, the semantic rules will be given by lambda-expressions.

3 Functors and Attribute Evaluators

After establishing the similarity of notations and expressiveness of *fold* functions and AGs, we now deal with their semantics and evaluation.

The semantics of a function expressed with *fold* is given by the *fold* operator definition, which uses the notion of *functor* [SF93]. The following equations define the *fold* operator for the simple type *list*.

Definition 3.1 (Fold operator for type *list*) *The* fold^{list} operator *is defined over each constructor of type* list *by:*

$$\begin{array}{ll} fold^{list}(f_n, f_c)\ Nil & = f_n() \\ fold^{list}(f_n, f_c)\ (Cons(a,l)) & = f_c(a, fold^{list}(f_n, f_c) l) \end{array}$$

With this definition, the *length* function in Fig. 1 can be evaluated (computed), since the parameters of f_c (i.e. a and r) are identified. As can be seen in Def. 3.1, for the $Cons(a,l)$ constructor, the *accumulative result variable* r is recursively defined over l. This fold^{list} operator is defined with *functors* which are statically and automatically defined over the *list* type constructors. The means to compute accumulating functions over the structure is given by these functors. They thus give a sense to the evaluation of functions expressed with *fold*. Note that the recursion scheme of the computation is strictly identical to that of the type definition: this is the essence of structure-directed programming.

The semantics of an AG is the solution of the system of equations associated with the set of semantic rules with attribute occurrences over a particular input structure (a tree in the CFG sense), and this, whatever method is used to solve this system of equations. In other words, from a given specification (AG), different techniques can be used to generate an attribute evaluator, leading possibly to different evaluations methods but always to the same semantics.

In the case in which an AG represents a *fold* function, the evaluator specified by the functors is a correct but particular attribute evaluator. Indeed, functor definitions depend only on the given type(s). They are statically derived from the constructors of this type and are valid for all accumulating functions of all *fold* programs defined on this type. Furthermore, the class of AGs which corresponds to functions expressed with folds is a well-known (actually, the simplest) class of AGs, called *purely-synthesized*, and noted S^1—the "1" refers to the fact that each non-terminal carries a single attribute (see also [FJMM91]).

$$\begin{array}{lrl} \text{append(x,y)} = \text{fold}^{list}(f_n, f_c)\ \text{x} & & \\ \quad \text{where} & f_n = & \lambda().\text{y} \\ & f_c = & \lambda(\text{a,r}).\text{Cons(a,r)} \end{array}$$

Figure 3: Definition of *append(x,y)* with fold^{list}

```
Nil ->
```
$f_{Nil,v_{\uparrow_{Nil}}}$: $\text{v}_{\uparrow_{Nil}} = (\lambda().\text{y})()$
```
Cons -> a list
```
$f_{Cons,v_{\uparrow_{Cons}}}$: $\text{v}_{\uparrow_{Cons}} = (\lambda(\text{a,r}).\text{Cons(a,r)})(\text{a,v}_{\uparrow_{list}})$

Figure 4: Definition of *append(x,y)* with an AG

Since	$\text{length(x)} = \text{fold}^{list}(\lambda\ ().\text{Zero},\lambda\ (\text{a,r}).\text{Succ(r)}\ )\ \text{x}$
and	$\text{fold}^{list}\ (f_n, f_c)\ (\text{Cons(a,l)}) = f_c(\text{a,fold}^{list}(f_n, f_c)\ \text{l})$
then	$\text{length(Cons(a,l))} = \text{Succ(length(l))}$

Figure 5: *Application to a Construction* over *length*

4 Deforestation

Since the expressiveness and the evaluation methods of folds and purely synthesized AGs are very similar, the aim of this section is to compare the deforestation methods associated with both formalisms: the Normalization Algorithm for first-order folds and Descriptional Composition for S^1 AGs. In the following, we quickly describe each method, not to give a formal treatment of them, but in order to show that DC is based on a similar but more *symbolic* Promotion Theorem, in the sense that it is independent of the evaluation method (functor definition). We will illustrate their effects over the simple example of *length(append)* [SF93]. Fig. 3 shows the fold function for append, and Fig. 4 shows the corresponding AG.

Normalization Algorithm

The Normalization Algorithm consists of three parts:

- *Generalization* consists in associating some terms with variables, and in replacing such a term by its associated variable each time it is encountered during derivations in NA.
- *Application to a Construction* is the application of the fold operator definition to a constructor and its parameters. Fig. 5 presents an example of this step over the *length* function.

fold^{list}(λ().length(y),
λ(r$_1$,r$_2$).Succ(r$_2$)) x

Figure 6: Result of NA: *length(append(x,y))*

- *Fold Promotion* is the fundamental step of NA. It is based on the Fold Promotion Theorem[1] [SF93]. This theorem states that the composition of a function g with a fold function is a new fold function. The new accumulating functions depend on function g and on the accumulating functions of the original fold. For instance, this theorem gives for type *list*:

$$\frac{\begin{array}{lcl}\phi_n() & = & g(f_n()) \\ \phi_c(a, g(r)) & = & g(f_c(a,r))\end{array}}{g(fold^{list}(f_n, f_c)x) = fold^{list}(\phi_n, \phi_c)x} \tag{1}$$

The Fold Promotion Theorem ensures the validity of the definition of the resulting fold function when the construction of the ϕ functions is performed locally on each constructor, i.e. each accumulating function in the result depends only on function g and on the accumulating functions of the original fold.

In our previous work [Dur94] comparing Wadler's first deforestation technique [Wad88] and DC, we realized that Wadler's algorithm is a more global transformation than DC (i.e. it considers the whole program at once). In contrast, we will show below that DC has the same local-transformation property as the Fold Promotion Theorem.

It is important to notice that the role of the Fold Promotion Theorem is to define the ϕ_i accumulating functions of the resulting fold, on which the *Application to a Construction* and *Generalization* steps can be applied. More precisely, function g is moved inside the accumulating functions of the resulting fold and hence is directly applied over the original accumulating functions (i.e. their constructors). But the real "deforestation" process (i.e. elimination of structure constructors) is only performed by the *Application to a Construction* and *Generalization* steps in these new ϕ_i functions.

The result of NA over *length(append)*, which contains no *Cons* constructor anymore, is given in Fig. 6.

Descriptional Composition

The aim of Descriptional Composition is to construct, for a given composition of two attribute grammars $\Omega(G_\Omega) \to G_\Delta$ and $\Delta(G_\Delta) \to G_\Theta$, a new attribute grammar $(\Delta \circ \Omega)(G_\Omega) \to G_\Theta$ which has the same semantics as the successive application of Ω and Δ. This new AG will not, though, create the structure corresponding to the intermediate result of type G_Δ.

[1] This theorem is an instance of a famous law in category theory, often called *fixed point fusion* [MFP91].

```
Nil ->
```
$f_{Nil,vs_{\uparrow_{Nil}}}$: $\mathrm{vs}_{\uparrow_{Nil}} = (\lambda().\mathrm{length(y)})\ ()$
```
Cons -> a list
```
$f_{Cons,vs_{\uparrow_{Cons}}}$: $\mathrm{vs}_{\uparrow_{Cons}} = (\lambda(\mathrm{a,r}).\mathrm{Succ(r)})\ (\mathrm{a,vs}_{\uparrow_{list}})$

Figure 7: The DC *(length ∘ append)(x,y)* of *(length(append(x,y)))*

Rather than the formal definition of DC [GG84], let's give its basic idea: the notion of *semantic rule projection*. Intuitively, $(\Delta \circ \Omega)$ is constructed from Ω by replacing each semantic rule which computes a term of G_Δ by a projection of the semantic rules in Δ over this term. More precisely, if a given semantic rule for the constructor $C_i^{G_\Omega}$ in Ω computes a term t having the type of $C_j^{G_\Delta}$ in G_Δ, then this semantic rule and its attributes are replaced by the projection of the semantic rules and attributes for the constructor $C_j^{G_\Delta}$ in Δ to Ω. This projection follows the structure of the original "constructor" semantic rule. DC is a purely syntactic transformation and does not take into account the semantics of the projected semantic rules.

To grasp the intuition of this method, we propose to study its effect on our example *length(append)*, presented in Fig. 7—for a complete definition, the reader is referred to the original paper [GG84, DPRJ96]. Consider the definition of each function as AGs (Figs. 2 and 4). The semantic rule $f_{Cons,v_{\uparrow_{Cons}}}$ in *append* creates a *Cons*. So, the semantic rule $f_{Cons,s_{\uparrow_{Cons}}}$ of *length* for the *Cons* production is projected onto $f_{Cons,v_{\uparrow_{Cons}}}$, and a new attribute vs is created, leading to the semantic rule $f_{Cons,vs_{\uparrow_{Cons}}}$ in the *Cons* production of $(length \circ append)$ (Fig. 7).

Like the difference between fold evaluation and AG evaluation (the functor definition depends only on the type whereas all attribute evaluation methods depend on the form of semantic rules), the difference between NA and DC lies in the knowledge of the evaluation method. DC doesn't require this knowledge at all. In fact, the evaluation method chosen for the result of DC may be different from the evaluation methods of the input AGs.

To interpret the DC method in terms of NA notions, the projection of the semantic rules of DC directly yields the deforested version of the ϕ_i's, whereas those given by the Fold Promotion Theorem must be further deforested by the *Application to a Construction* and *Generalization* steps of NA. Thus, it is possible to view the DC correctness theorem [GG84] as a more symbolic Fold Promotion Theorem, in the sense that the correctness proof for DC is independent of the attribute evaluation method,[2] unlike the Fold Promotion Theorem which is strongly based on the functor definitions. Thus, DC is a true source-to-source transformation, independent of any evaluation method: it is a *symbolic composition* without any *Application to a Construction* step.

The *Application to a Construction* step in NA is strongly tied to the evaluation method, i.e. the functor. We view this step as a kind of partial evaluation, which is generalized by the

[2] Beware however of DC class-closure problems.

$$\text{reverse(x)} = \text{fold}^{list}(\lambda().\text{Nil}, \quad \lambda(\text{a,r}).\text{append(r,Cons(a,Nil))}\ )\ \text{x}$$

Figure 8: The *reverse* fold function

`Nil ->`
$f_{Nil,s_{\uparrow_{Nil}}}$: $s_{\uparrow_{Nil}} = (\lambda().\text{Nil})\ ()$
`Cons -> a list`
$f_{Cons,s_{\uparrow_{Cons}}}$: $s_{\uparrow_{Cons}} = (\lambda(\text{a,r}).\text{append(r,Cons(a,Nil))})\ (\text{a},s_{\uparrow_{list}})$

Figure 9: The AG corresponding to the *reverse* fold function

Generalization step. If this *Application to a Construction* step is never applied to a bound lambda-term (not a specialization of the input term), then it seems that the first-order fold normalization is equivalent to the descriptional composition of the corresponding S^1 AG. In the same way that DC is some kind of symbolic composition, NA can thus be viewed as some kind of generalized partial evaluation.

5 Higher-order fold and Inherited Attributes

In this section we present rapidly more complex fold program examples, which require to be transformed into second-order fold functions for normalization and which, in the AG context, can be expressed with inherited attributes. As a running example, we will use the deforestation of (*reverse* ∘ *reverse*) over the *list* type [SF93], which gives the identity (or *copy*) function. It is impossible to apply NA on every fold program, just as DC doesn't work on every AG: essentially, the constructors used to build the result value must be directly visible in the accumulating functions. For example, the *reverse* fold (Fig. 8) is not *potentially normalizable*, just as DC can't be applied over the corresponding AG (Fig. 9), for the same reason. To solve this problem, Sheard and Fegaras [SF93, LS95] introduced the *Second-order Fold Promotion Theorem*.

Since AGs naturally allow the use of inherited attributes, the prefered AG form for *reverse* is not the AG of Fig. 9 corresponding to the fold function of Fig. 8; rather, it makes use of an inherited attribute, as shown in Fig. 10. DC can then be directly applied on (the composition of) this natural AG (with itself).

In [DPRJ96], we go a little further in the comparison between the second-order (fold) and inherited (AG) approaches. In spite of their apparent dissimilarities, we notice that there exist some relations between these methods. For instance, the purely-synthesized higher-order AG derived from a given AG with inherited attributes by Knuth's transformation [CM79] is similar to the corresponding second-order fold program.

With $Id = \lambda(x).x$ being the *identity* function:
`Nil ->`
$f_{Nil,s_{\uparrow_{Nil}}}$: $s_{\uparrow_{Nil}} = Id\ (h_{\downarrow_{Nil}})$
`Cons -> a list`
$f_{Cons,s_{\uparrow_{Cons}}}$: $s_{\uparrow_{Cons}} = Id\ (s_{\uparrow_{list}})$
$f_{Cons,h_{\downarrow_{list}}}$: $h_{\downarrow_{list}} = (\lambda(a,h).\mathrm{Cons}(a,h))\ (a,h_{\downarrow_{Cons}})$

Figure 10: The natural AG for *reverse*

6 Structure-directed Genericity

In this section we study the notion of genericity, which in this paper is the possibility to reuse algorithms specified on a certain data structure on other structures. Although not a new concept by far, genericity raises a lot of interest in the programming and software engineering community, because it opens a way to lower software costs. This is one of the main motivations of our comparison between AGs and functional programming, in which genericity schemes look quite different.

In functional programming, the scheme which received the most attention is polymorphism, made popular by type inference algorithms implemented in ML systems. In this approach, there is a strict separation between the (top-level) structure and the (opaque) data attached to its leaves. A polymorphic function may only examine the top part. Then, such a function can be reused immediately, without any additional "action" from the user, on any other structure that has exactly the same top part: the type inference algorithm eliminates the instantiation specification/operation. This ease of use makes polymorphism quite attractive.

However, it also sets the limits of the approach: the structure on which a polymorphic function works is absolutely fixed, at least on the top part which it is allowed to examine, and hence you can't reuse it on a datum that is similar but not quite isomorphic. As an example, the polymorphic function which computes the length of a list (see Fig. 1) can't be reused to compute the number of leaves of a tree, although both structures can be considered as (the same kind of) containers for a collection of data.

At the other end of the spectrum stands *structure-directed genericity*, devised a few years ago in the context of AGs by Farrow *et al.* and ourselves [FMY92, LJPR93, Rou94, BG94]. It is based on the observation that you can instantiate (reuse) onto type T_1 any function f defined on type T_2 if you write a *coupling* function that transforms every structure of type T_1 into the "equivalent" structure of type T_2. For instance, to reuse the list-length function to count the number of leaves of a tree, you write a function which returns the list of leaves of a tree—see Fig. 11 for the definition of this function as an AG—and compose it with the list length function. Thanks to DC, the result of the composition, i.e. the function which counts the number of leaves of a tree, works directly on the tree without constructing the intermediate list—see Fig. 12. It is clear that, with this approach, instantiation is much less easy than with polymorphism, because you have to write the coupling function by hand, but on the other hand you have much more flexibility in defining the "equivalence" of two structures.

Structure-directed genericity is popular in AGs because it's easy to write structure translations with AGs (and because most grammars are so complex that superficial polymorphism is insufficient). However, it is not at all restricted to this paradigm. In particular, the comparison between folds and AGs in the previous section suggests to use folds to apply it to structure-directed functional programming. For instance, to count the leaves of a tree with a fold, use the fold in Fig. 13 to construct the list of leaves and fuse it with the list-length fold in Fig. 1. Again, the instantiated function (the fold resulting from the fusion) works directly on the tree without constructing the intermediate list, and hence is no less efficient than a polymorphic function.

AG ttl-aux is
  `Leave -> a`
    $s_{\uparrow Leave} = (\lambda(a,h).\mathrm{Cons}(a,h))\ (a,\ h_{\downarrow Leave})$
  `Node -> ltree rtree`
    $s_{\uparrow Node} = Id\ (s_{\uparrow ltree})$
    $h_{\downarrow rtree} = Id\ (h_{\downarrow Node})$
    $h_{\downarrow ltree} = Id\ (s_{\uparrow rtree})$
function tree-to-list(t) = ttl-aux(t, Nil)

Figure 11: An AG which constructs the list of leaves of a tree

AG tlgt-aux is
  `Leave -> a`
    $s_{\uparrow Leave} = (\lambda(a,h).\mathrm{Succ}(h))\ (a,\ h_{\downarrow Leave})$
  `Node -> ltree rtree`
    $s_{\uparrow Node} = Id\ (s_{\uparrow ltree})$
    $h_{\downarrow rtree} = Id\ (h_{\downarrow Node})$
    $h_{\downarrow ltree} = Id\ (s_{\uparrow rtree})$
function tree-length(t) = tlgt-aux(t, Zero)

Figure 12: The automatically-generated AG which counts the number of leaves of a tree

$$\text{tree-to-list}(x) = \text{fold}^{tree}(\lambda(a).\mathrm{Cons}(a,\mathrm{Nil}),\ \lambda(l,r).\mathrm{append}(l,r)\ )\ x$$

Figure 13: The fold which constructs the list of leaves of a tree

Now, none of these approaches is entirely satisfactory. On one hand, polymorphism is too restrictively applicable to be a real-life software engineering tool. On the other hand, the all-manual approach is, well, manual and tedious, even if structure-directed programming helps a lot.

So, some people tried to do better from both sides. Trying to relax the restrictions of polymorphism while keeping the automatic-instantiation property lead to work on polytypic programming [JJ96, Mee96] and shapely types [JC94]. Trying to automate structure-directed genericity lead to our own work on automatic generation of coupling AGs [LJPR93, Rou94, Cor96] and to a completely different approach to genericity in AGs, devised by Kastens and Waite and based on some notion of inheritance [KW94].

Interestingly enough, neither polytypic programming nor our automatic generation of coupling AGs is able to handle the above problem as it is formulated, i.e. reusing (without modification) the list-length function to compute the number of leaves of a tree. In [JJ96], a polytypic *size* function is given, which can compute both the length of a list and the number of leaves of a tree, but it is different from the classical list-length function. And, following [Cor96] and its notion of normal form of a grammar w.r.t. a set of key non-terminals, it is easy to show that lists and trees are in different classes, so that there is no way to go from a tree to a list by our automatically-generated coupling AGs, which are purely-synthesized structure-preserving transformations (however, a list is a degenerated form of tree, with "less" structure, so we can handle the list-to-tree transformation).

This doesn't mean that polytypic programming and our automatic generation of coupling AGs are inferior to shapely types and Kastens and Waite's constructs, respectively, which do better on our particular example. To the contrary, we believe that the former approaches, which give more importance to the type structure, are more precise than the latter in their notion of correspondence between two types, and hence will be able to handle more complex cases.

7 Conclusion

In the program transformation area, there is an emergence and a recently growing interest in the structure-directed style of functional programming. This lead us to compare some of the methods and approaches related to this style with those of attribute grammars, of which this notion is the fundamental basis.

First, we have shown that generic control operators such as fold are equivalent in expressiveness to a restricted class of attribute grammars (purely synthesized). For the first order fold, the Normalization Algorithm, which performs fusion of functions, has the same effect as the Descriptional Composition of the corresponding attribute grammars. Actually, these methods are slightly different, because we can view DC as a more symbolic transformation than NA, which is more akin to generalized partial evaluation. But the most important point is that they have the same local-transformation property (over a type constructor), which is possible because the type structure is explicit in the programs.

From the outset, DC can handle inherited attributes, which allow to express more complicated programs. On the other hand, NA can be applied to these programs, possibly after a transformation to higher order. Since each of these approaches (higher order folds and inherited attributes) are specific to each domain (functional programming and attribute grammars), their comparison is more difficult. In spite of obvious similarities and minor differences, this subject requires a more extensive study.

Still, our main motivation for this work holds in the important role of DC for structure-directed genericity in AGs. Indeed, DC is the basic tool which enables to instantiate an AG (algorithm) over a new structure (via a specification of the structure coupling by another AG).

According to our present work and other recent ones, it seems that both functional programming and attribute grammars (and probably other programming paradigms or styles) tend to reach a concept of genericity which is more or less the same, striking the right balance between the manual approach by structure coupling and function composition and the automatic approach by basic polymorphism. This common goal for different communities with different approaches could lead to fruitful cross-fertilizations; this is our strong belief and the thrust of our future work.

References

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BG94] John Boyland and Susan L. Graham. Composing tree attributions. In *21st ACM Symp. on Principles of Programming Languages*, pages 375–388, Portland, Oregon, January 1994. ACM Press.

[CM79] Laurian M. Chirica and David F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13(1):1–27, 1979. See also: report TRCS78-2, Dept. of Elec. Eng. and Computer Science, University of California, Santa Barbara, CA (October 1978).

[Cor96] Loïc Correnson. Généricité dans les grammaires attribuées. Rapport de stage d'option, École Polytechnique, 1996.

[DPRJ96] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-2957.ps.gz`.

[Dur94] Etienne Duris. Transformation de grammaires attribuées pour des mises à jour destructives. Rapport de DEA, Université d'Orléans, September 1994.

[FJMM91] M. M. Fokkinga, J. Jeuring, L. Meertens, and E. Meijer. A translation from attribute grammars to catamorphisms. *The Squiggolist*, 2(1):20–26, 1991.

[FMY92] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[FSS92] Leonidas Fegaras, Tim Sheard, and David Stemple. Uniform traversal combinators: Definition, use and properties. In *11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *Lect. Notes in Comp. Sci.*, pages 148–162, Saratoga Springs, New York, June 1992. Springer-Verlag.

[FSZ94] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*, pages 21–32, Orlando, Florida, June 1994.

[GG84] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, June 1984. Published as *ACM SIGPLAN Notices*, 19(6).

[GLJ93] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen, Denmark, June 1993. ACM Press.

[JC94] C. Barry Jay and J. R. B Cockett. Shapely types and shape polymorphism. In Donald Sannella, editor, *European Symp. on Programming (ESOP '94)*, volume 788 of *Lect. Notes in Comp. Sci.*, pages 302–316. Springer-Verlag, April 1994.

[JJ96] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lect. Notes in Comp. Sci.*, pages 68–114. Springer-Verlag, 1996.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[KW94] Uwe Kastens and William M. Waite. Modularity and Reusability in Attribute Grammars. *Acta Informatica*, 31:601–627, 1994.

[LJPR93] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lect. Notes in Comp. Sci.*, pages 123–136, Tallinn, August 1993. Springer-Verlag.

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 314–323, La Jolla, CA, USA, 1995. ACM Press.

[Mee96] Lambert Meertens. Calculate polytypically. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 1–16, Aachen, September 1996. Springer-Verlag.

[MFP91] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lect. Notes in Comp. Sci.*, pages 124–144, Cambridge, September 1991. Springer-Verlag.

[Paa95] Jukka Paakki. Attribute grammar paradigms — A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[Rou94] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées*. PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 233–242, Copenhagen, Denmark, June 1993. ACM Press.

[Wad88] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lect. Notes in Comp. Sci.*, pages 344–358, Nancy, March 1988. Springer-Verlag.

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

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399