Attribute Grammars as Recursion Schemes over Cyclic Representations of Zippers

Eric Badouel, Bernard Fotsing, Rodrigue Tchougong

Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 229 (5) (2011) 39–56
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Attribute Grammars as Recursion Schemes over Cyclic Representations of Zippers

Eric Badouel[1]

INRIA Rennes - Bretagne Atlantique,
Campus universitaire de Beaulieu, F-35402 Rennes, France

Bernard Fotsing[2] Rodrigue Tchougong[3]

IRISA, Université de Rennes 1,
Campus universitaire de Beaulieu, F-35402 Rennes, France

Abstract

Evaluation of attributes w.r.t. an attribute grammar can be obtained by inductively computing a function expressing the dependencies of the synthesized attributes on inherited attributes. This higher-order functional approach to attribute grammars leads to a straightforward implementation using a higher-order lazy functional language like Haskell. The resulting evaluation functions are, however, not easily amenable to optimization rules. We present an alternative first-order functional interpretation of attribute grammars where the input tree is replaced with an extended cyclic tree each node of which is aware of its context viewed as an additional child tree. By the way, we demonstrate that these cyclic representations of zippers (trees with their context) are natural generalizations of doubly-linked lists to trees over an arbitrary signature.

Keywords: Attribute Grammar, Attribute Evaluation, Cyclic Data Structure, Zipper

1 Introduction

Attribute grammars [21,24] were introduced to make possible the manipulation of context-sensitive information, like the scope of a variable in a program. This formalism, used mainly in the context of language processing tools, can be used with two purposes: either to decorate an input tree with attributes (thus adding information locally at each node) or to define a syntax-directed computation in order to translate that input tree into some semantic domain. These two problems are related since the result of syntax-directed computation is usually given by the value

[1] Email: `ebadouel@irisa.fr`
[2] Email: `bfotsing@irisa.fr`
[3] Email: `rtchougo@irisa.fr`

doi:10.1016/j.entcs.2011.02.015

of a specific attribute at the root node; which can be extracted once the decoration of the tree has been computed (even though one may not have to compute the whole decoration to obtain the required result). On the other hand the decorated tree can be given by a specific attribute, even though values of these attributes at different nodes can share whole subexpressions. In order to obtain efficient implementations different algorithmic solutions have often been put forward for these two situations. However if a lazy functional language like Haskell is used, one can adopt the same solution in both cases. Indeed, on the one hand, lazy evaluation avoids unecessary computations and, in the other hand, it allows sharing of subexpressions at different places.

The set of input trees is a regular set of trees given as the set of abtract-syntax trees of a context-free grammar or, if we are not interested in the concrete syntax but only on the abstract structure of trees, as the set of terms build from a multi-sorted signature. Values of attributes are defined with a set of recursive definitions given by the so-called semantic rules of the attribute grammar. If the attribute grammar is non circular (there are no cyclic dependencies between attributes) then one can can compute the value of each attribute using a topological sort of the dependency graph (whose arcs indicate the dependencies between attribute values). One can nevertheless use an order-algebraic approach based on least fixed-points [6,23] in order to compute attributes for potentially circular attribute grammars (and on potentially infinite input trees).

General attribute grammars use both synthesized and inherited attributes bearing information respectively from the subtree stemming from the given node and the context of that subtree. Attribute grammars with only synthesized attributes amount to primitive recursive schemes [9] and value of attributes are easily computed by structural recursion on trees. Things are more involved for general attribute grammars due to the manipulation of contextual information. However it was soon recognized that we can resort to attribute grammars with only synthesized attributes whose attributes are functions expressing the dependencies of the synthesized attributes on inherited attributes. Thus attribute grammars reduce to structural induction on trees at the price of using higher-order values. This higher-order functional approach to attribute grammars [19,11,2,22] leads to efficient implementations in a higher-order lazy functional language like Haskell. The Elegant system developped at Philips [1] and the UUAG system [25] from Utrecht university both illustrate this approach.

Unfortunately the resulting evaluation functions are not easily amenable to optimization techniques like short-cut fusion which are based on first-order representations of functions. We thus present an alternative first-order functional interpretation of attribute grammars where the input tree is replaced with an extended cyclic tree where each node is aware of its context viewed as an additional child tree. The price to pay is a preprocessing phase to unfold a tree into its extended cyclic version. By the way, we demonstrate that these cyclic representations of zippers (trees with their context [16]) are natural generalizations of doubly-linked lists to trees over an arbitrary signature. More precisely there are two natural ways of representing lists in order to be able to navigate through them in both directions. Either we represent the list together with its context (zipper) for instance by using a pair of pushdown stacks: one for the current list itself and the other, in reverse order, for its context; or by using at each node a pointer to its preceding node (doubly-linked list). From a given multi-sorted signature Σ we derive an extended signature $\mathcal{Z}_\Sigma$ whose corresponding trees are associated with Σ-trees or with their contexts. A zipper is introduced as a pair made of a tree and its context; thus generalizing the pair of stacks representation of lists to trees over an arbitrary signature. We also introduce a cyclic representation of zippers where each tree (respectively context) is aware of its context (resp. attached subtree) given as an extra argument; this gives rise to a new signature $\mathcal{CZ}_\Sigma$ generalizing the doubly-linked representation of lists. In both cases, we present a corresponding algorithm for attribute evaluation. The first one (related to the zipper representation) is similar to the solution presented by Uustalu and Vene [26] even though we do not make use of the underlying structure of comonad. The second algorithm (related to the cyclic representation of zipper) is new.

The rest of the paper is organized as follows. In section 2 we recall the basic definitions on attribute grammars and present a variant of the higher-order functional interpretation of attribute grammars. In section 3 we introduce the zippers, and the evaluation of attributes based on zipper representation. In section 4 we introduce the cyclic representation of zippers and the unfolding of a tree into its cyclic representation. In section 5 we introduce our first-order interpretation of attribute grammars based on the cyclic representation of trees.

2 Higher-order functional interpretation of an attribute grammar

In order to fix some notations, we first very briefly recall some mathematical definitions on multi-sorted signatures and their algebras (we assume the reader to be familiar with these notions, he may wish to consult [14,4] for a more detailed presentation); then we proceed to the definition of an attribute grammar and its associated interpretation. We conclude this section by introducing the notion of a rooted attribute grammar

2.1 Signature and algebra

Definition 2.1 A (multi-sorted) signature $\Sigma = (\mathcal{S}, \mathcal{O}_p)$ consists of a finite set S of sorts, and a finite set $\mathcal{O}_p$ of operators. Each operator op has an arity $\alpha(op) \in \mathcal{S}^*$ and a sort $\sigma(op) \in \mathcal{S}$. We let notation $op : s_1 \times \cdots \times s_n \to s$ mean that op is an operator of arity $\alpha(op) = s_1 \cdots s_n$ and sort $\sigma(op) = s$. The rank of operator op is the length of its arity: $\rho(op) = |\alpha(op)|$. If $\alpha(op) = \varepsilon$, op is said to be a constant of sort $\sigma(op)$.

As an example we consider the signature with only one sort $Tree$ and whose operators are as follows:

$$\begin{array}{ll} Fork & : Tree \times Tree \to Tree \\ Leaf \;\; label : & \qquad\qquad\quad \to Tree \end{array}$$

Thus we have a set of constants indexed by a set of labels together with a binary operator. It corresponds to the following Haskell datatype definition.

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

Definition 2.2 Let $\Sigma = (\mathcal{S}, \mathcal{O}_p)$ be a signature, a Σ-algebra $\mathcal{A}$ consists of a domain of interpretation, a set $\mathcal{A}_s$, for each sort $s \in \mathcal{S}$, and a function $op^{\mathcal{A}} : \mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n} \to \mathcal{A}_s$ associated with each operator $op : s_1 \times \cdots \times s_n \to s$. A morphism of algebras $f : \mathcal{A} \to \mathcal{B}$ is a family of maps $f_s : \mathcal{A}_s \to \mathcal{B}_s$ such that, for every $a_i \in \mathcal{A}_{s_i}$

$$f_s\left(op^{\mathcal{A}}(a_1, \ldots, a_n)\right) = op^{\mathcal{B}}\left(f_{s_1}(a_1), \ldots, f_{s_n}(a_n)\right)$$

An algebra is said to be continuous when the domains of interpretations are complete partial orders and the interpretations of operators are continuous functions. We let $T(\Sigma)_s$ denote the set of Σ terms of type s, and $Tree(\Sigma)_s$ the set of finite or infinite trees of sort s build upon the signature Σ together with their approximants. These sets are the respective carrier sets of the free Σ-algebra and the free continuous Σ-algebra. We identify terms with finite trees, and we interpret a tree as a partial map $t : \mathbb{N}^* \to \Sigma$ whose domain $Dom(t)$ is a non-empty prefix-closed language such that for every $u \in Dom(t)$ with $t(u) = op : s_1 \times \cdots \times s_n \to s$, and $i \in \mathbb{N}$, one has $u \cdot i \in Dom(t) \Leftrightarrow 1 \leq i \leq n$ and $\sigma(t(u.i)) = s_i$. Moreover we let t_u stand for the subtree of t rooted at u given by $Dom(t_u) = \{v \in \mathbb{N}^* \mid u \cdot v \in Dom(t)\}$ and $t_u(v) = t(u \cdot v)$.

2.2 Attribute grammar

The nodes of a Σ-tree can be decorated by attributes whose values are computed according to semantic rules.

Definition 2.3 An (abstract) *attribute grammar* $\mathbb{G} = (\Sigma, Attr, \mathcal{D}, sem)$ is a signature $\Sigma = (\mathcal{S}, \mathcal{O}_p)$ each grammatical symbol (sort) of which is associated with a set of attributes in which we distinguish *inherited attributes* from *synthesized attributes*: $Attr(s) = Inh(s) \uplus Syn(s)$. The domain of evaluation of an attribute $q \in Attr(s)$ is a complete partial order $\mathcal{D}_q$. We let

$$\mathcal{D}_s^{\downarrow} = \prod_{q \in Inh(s)} \mathcal{D}_q \text{ and } \mathcal{D}_s^{\uparrow} = \prod_{q \in Syn(s)} \mathcal{D}_q$$

denote respectively the domains of interpretation for the inherited and the synthesized attributes of a node of sort s. Moreover a set of rules (the so-called *semantic rules*) are associated with each operator of the signature. These rules give the functional dependencies between the values of attributes and are given by the function:

$$sem(op) : \mathcal{D}_s^{\downarrow} \times \mathcal{D}_{s_1}^{\uparrow} \times \cdots \times \mathcal{D}_{s_n}^{\uparrow} \longrightarrow \mathcal{D}_s^{\uparrow} \times \mathcal{D}_{s_1}^{\downarrow} \times \cdots \times \mathcal{D}_{s_n}^{\downarrow}$$

A node $u \in Dom(t)$ is said to be an occurrence of grammatical symbol $s = \sigma(t(u))$, then it has the same attributes as s. The rationale of the distinction made between inherited and synthesized attributes is the following. Synthesized attributes in a given node of a tree represent information comming from the subtree rooted at that node. Conversely, inherited attributes represent information comming from outside this subtree (*i.e.* from its context). For this reason, we define as *input attribute* of operator $op : s_1 \cdots s_n \to s$ either an inherited attribute of s (whose value comes from the context) or a synthesized attribute of some of the s_i for $1 \leq i \leq n$ (whose value comes from the respective subtree). The remaining attributes, *i.e.* the synthesized attributes of s and the inherited attributes of the s_i for $1 \leq i \leq n$ are called *ouput attributes* or *defined attributes*. The set of semantic rules $sem(op)$ associated with production op does actually contain exactly one definition for each output attribute in term of the input attributes.

Let us consider, as an illustration, the following attribute grammar for computing the frontier of a binary tree (the list of labels of its leaves from left to right) given by synthesized attribute *flatten* using an accumulating parameter (inherited attribute *coflat*).

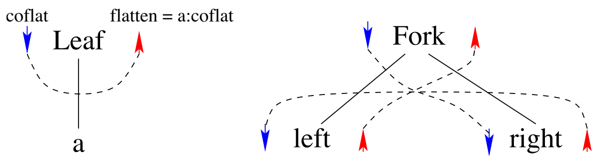

Fig. 1. an attribute grammar for computing the frontier of a binary tree

$$Leaf \quad a \quad :: \quad \longrightarrow Tree_\varepsilon$$
$$\left\{ Tree_\varepsilon \cdot flatten = a : (Tree_\varepsilon \cdot coflat) \right.$$

$$Fork \quad :: \quad Tree_1 \times Tree_2 \longrightarrow Tree_\varepsilon$$
$$\begin{cases} Tree_\varepsilon \cdot flatten = Tree_1 \cdot flatten \\ Tree_1 \cdot coflat \ = Tree_2 \cdot flatten \\ Tree_2 \cdot coflat \ = Tree_\varepsilon \cdot coflat \end{cases}$$

In order to present the semantic rules we need to distinguish the different occurrences of a same grammatical symbol (sort). For this purpose if $op : s_1 \times \cdots \times s_n \to s$ is an operator we let $op :: (s_1)_1 \times \cdots \times (s_n)_n \to s_\varepsilon$ for the extended notation where each occurrence of sort is tagged with its position, and by a slight abuse of notation we shall often write $op :: X_1 \times \cdots \times X_n \to X_\varepsilon$ where X_i is an abreviation for $(s_i)_i$ and $X_\varepsilon = s_\varepsilon$. Then the semantic rules attached to an operator op are of the form

$$op \quad :: X_1 \times \cdots \times X_n \quad \longrightarrow \ X_\varepsilon$$
$$\begin{cases} X_\varepsilon \cdot syn = sem(op)_{\varepsilon,syn}\,(X_\lambda \cdot q; (\lambda, q) \in In_{op}) & \text{where } syn \in Syn(s) \\ X_i \cdot inh \ = sem(op)_{i,inh}\,(X_\lambda \cdot q; (\lambda, q) \in In_{op}) & \text{where } 1 \leq i \leq n \text{ and } inh \in Inh(s_i) \end{cases}$$

where

$$\begin{aligned} In_{op} &= \{(\varepsilon, q) \mid q \in Inh(s)\} \bigcup \{(i, q) \mid 1 \leq i \leq n \quad q \in Syn(s_i)\} \\ Out_{op} &= \{(\varepsilon, q) \mid q \in Syn(s)\} \bigcup \{(i, q) \mid 1 \leq i \leq n \quad q \in Inh(s_i)\} \end{aligned}$$

represent the sets of occurrences of input attributes and output attributes respectively.

The semantic functions are actually rule schemes whose purpose is to define the value of each attribute at every node of (the tree representation of) the term. For instance if t is a tree and $u \in Dom(t)$ is such that $t(u) = Fork$ then the above equations should be interpreted as

$$\begin{aligned} flatten(t_u) &= flatten(t_{u \cdot 1}) \\ coflat(t_{u \cdot 1}) &= flatten(t_{u \cdot 2}) \\ coflat(t_{u \cdot 2}) &= coflat(t_u) \end{aligned}$$

Thus each tree is associated with a system of equations whose variables are the occurrences of attributes

$$\begin{aligned} V_t = &\{v_{t,\pi,q} | \pi \in Dom(t) \quad t(\pi) :: s_1 \times \cdots s_n \to s \quad ; q \in Att(s)\} \\ \cup &\{v_{t,\pi \cdot i,q} | \pi \in Dom(t) \quad t(\pi) :: s_1 \times \cdots s_n \to s \quad ; q \in Att(s_i)\} \end{aligned}$$

whose resolution provides the interpretation of tree $t \in Tree(\Sigma)_s$ w.r.t. to attribute grammar $\mathbb{G}$ as the map $(\![t]\!)_{\mathbb{G}} : \mathcal{D}_s^{\downarrow} \to \mathcal{D}_s^{\uparrow}$ given by:

$$\begin{aligned} (\![t]\!)_{\mathbb{G}}(v)(q) &= v_{t,\varepsilon,q} \\ \text{where } v_{t,\pi,q} &= sem(t(\pi))_{\varepsilon,q} \left(v_{t,\pi \cdot \lambda,q'}/(\lambda, q') \in In_{t(\pi)}\right) \quad \text{for } q \in Syn(\sigma(t(\pi))) \\ v_{t,\pi \cdot i,q} &= sem(t(\pi))_{i,q} \left(v_{t,\pi \cdot \lambda,q'}/(\lambda, q') \in In_{t(\pi)}\right) \quad \text{for } q \in Inh(\sigma(t(\pi \cdot i))) \\ v_{t,\varepsilon,q} &= v(q) \qquad \text{for } q \in Inh(\sigma(t(\varepsilon))) \end{aligned}$$

where it is assumed that the vector $\langle v_{t,\lambda,q} \rangle$ appearing in the "where" clause is the least solution of the corresponding system of equations. We shall make this assumption each time a "where" clause occurs in a definition; this conforms to the interpretation of Haskell programs. Figure 2 displays the flow of computations of attribute occurrences that produces the frontier of a binary tree assuming the initial value of the accumulating parameter (value of attribute *coflat* at the root node) is the empty list.

2.3 Algebra associated with an attribute grammar

The semantic rules of an attribute grammar are syntax-directed in the sense that they are given by rule schemes associated with each operator of the signature. For this reason we can exhibit a Σ-algebra $\mathcal{A}_{\mathbb{G}}$ derived from the attribute grammar $\mathbb{G}$ such that $(\![t]\!)_{\mathbb{G}} = t^{\mathcal{A}_{\mathbb{G}}}$, *i.e.* the interpretation of a tree as defined in the previous section is given by the evaluation morphism (catamorphism) associated with algebra $\mathcal{A}_{\mathbb{G}}$.

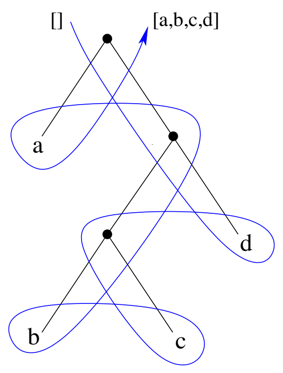

Fig. 2. computing the frontier of a binary tree with an attribute grammar

Definition 2.4 The Σ-algebra $\mathcal{A}_{\mathbb{G}}$ derived from attribute grammar $\mathbb{G}$ is such that $(\mathcal{A}_{\mathbb{G}})_s = \mathcal{D}_s^{\downarrow} \to \mathcal{D}_s^{\uparrow}$, and the interpretation of an operator $op : s_1 \times \cdots \times s_n \to s$ is the map $op^{\mathcal{A}_{\mathbb{G}}}$ given by:

$$\begin{aligned}
op^{\mathcal{A}_{\mathbb{G}}}(f_1, \ldots, f_n)(v)(q) \; &= \; v_{op,\varepsilon,q} \\
\text{where } v_{op,\varepsilon,q} &= v(q) \quad \text{if } q \in Inh(s) \\
v_{op,i,q} &= f_i(v_i)(q) \quad \text{where } q \in Syn(s_i) \text{ and } v_i(q') = v_{op,i,q'} \text{ for } \; q' \in Inh(s_i) \\
v_{op,\varepsilon,q} &= sem(op)_{\varepsilon,q}\left(v_{op,\lambda,q'}/(\lambda,q') \in In_{op}\right) \quad \text{if } q \in Syn(s) \\
v_{op,i,q} &= sem(op)_{i,q}\left(v_{op,\lambda,q'}/(\lambda,q') \in In_{op}\right) \quad \text{if } q \in Inh(s_i)
\end{aligned}$$

This definition is circular [5] since in the "where" clause the inherited attributes $v_i(q') = v_{op,i,q'}$ for $q' \in Inh(s_i)$ appear both in the left-hand side and in the right-hand side of the defining equations. Thus it should be interpreted as the characterization of the vector $\langle v_{op,\lambda,q} \rangle_{(\lambda,q) \in In_{op} \cup Out_{op}}$ as the least fixed-point of the corresponding transformation.

Proposition 2.5 $([t])_{\mathbb{G}} = t^{\mathcal{A}_{\mathbb{G}}}$

Proof. Straightforward, details are provided in [3]. □

The above semantics of attribute grammars follows the approaches presented in [19,2], it also draws its inspiration from [23,6] in the sense that it gives a fixed-point semantics of attribute grammars. We have an almost literal transcription of the above definition into the language Haskell as the mechanism of lazy evaluation escapes the apparent cyclicity of the resulting program [5]. A translation of attribute grammars into catamorphism (evaluation function for an algebra) was already presented in [11]. However the presentation that we have just given is, in our opinion, more explicit and far more elementary than the one given there and it leads to a straightforward implementation in Haskell. Notice that another advantage of lazy evaluation is that we can define computations of attributes on potentially infinite data structures. For instance we can define semantics rules on streams as long as every approximations of the value of a given attribute can be computed using only a finite prefix of the stream.

The evaluation of a tree w.r.t. the algebra induced by the attribute grammar is then given by the inductive definition:

$$\begin{aligned}
(op(t_1,\ldots,t_n)^{\mathcal{A}_{\mathbb{G}}}(v)(q) &= v_{op,\varepsilon,q} \\
\text{where } v_{op,\varepsilon,q} &= v(q) \quad \text{if } q \in \mathit{Inh}(s) \\
v_{op,i,q} &= t_i^{\mathcal{A}_{\mathbb{G}}}(v_i)(q) \ \text{ where } q \in \mathit{Syn}(s_i) \text{ and } v_i(q') = v_{op,i,q'} \text{ for } \ q' \in \mathit{Inh}(s_i) \\
v_{op,\varepsilon,q} &= \mathit{sem}(op)_{\varepsilon,q}\left(v_{op,\lambda,q'}/(\lambda,q') \in \mathit{In}_{op}\right) \quad \text{if } q \in \mathit{Syn}(s) \\
v_{op,i,q} &= \mathit{sem}(op)_{i,q}\left(v_{op,\lambda,q'}/(\lambda,q') \in \mathit{In}_{op}\right) \quad \text{if } q \in \mathit{Inh}(s_i)
\end{aligned}$$

Returning to our example, we derive the following Haskell code:

```
flatten :: Tree a -> [a] -> [a]
flatten (Leaf a) coflat = a:coflat
flatten (Fork left right) coflat = flatten left (flatten right coflat)
```

2.4 *Rooted attribute grammar*

From now on we consider that each signature has a specific sort a, called its *axioms.* It is often convenient to consider a top level function that uses an attribute grammar to evaluate a tree after an appropriate initialization of the inherited attributes of the root node (these attributes are parameters of the corresponding system of equations). This can be done by extending the attribute grammar with an additional operator $\mathit{Root} : a \to \top$ where a is the axiom of the grammar and $\top$ an additional sort, together with the associated semantic rules. A tree of the form $\mathit{Root}(t)$ where $t \in \mathit{Tree}(\Sigma)_a$ represents a rooted tree, *i.e.* a tree with an empty context. We assume that the additional sort $\top$ has no inherited attribute and one synthesized attribute corresponding to the end result. The semantic rules associated with this additional operator root are then given by a pair of functions: $\mathit{init} : \mathcal{D}_a^{\uparrow} \to \mathcal{D}_a^{\downarrow}$ providing the initialization of the inherited attributes at the root node, and $\mathit{result} : \mathcal{D}_a^{\uparrow} \to \mathcal{D}$ where $\mathcal{D} = \mathcal{D}_{\top}^{\uparrow}$ is the domain of values for the end result. We shall not explicitely add this new operator to the signature but consider that a rooted attribute grammar is an attribute grammar together with these two functions. The top level function is then the evaluation function associated with (the now implicit) operator *Root* in the algebra induced by the extended attribute grammar, it is therefore given by the expression shown in Figure 3. In our running example the *result* function is the

$$\begin{aligned}
&\mathit{return} : T(\Sigma)_a \longrightarrow \mathcal{D} \\
&\mathit{return}\ \ t \ = \ \mathit{result}(\mathit{val}) \\
&\quad \text{where } \mathit{val} \ = \ t^{\mathcal{A}_{\mathbb{G}}}(\mathit{init}(\mathit{val}))
\end{aligned}$$

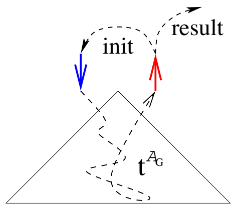

Fig. 3. Tying the knot: the top level function of a rooted attribute grammar

identity and the *init* function is the constant function returning the empty list (the accumulating parameter associated with the inherited attribute *coflat* is initialized to the empty list regardless of the value of the synthesized attribute *flatten* at the root node), hence it is given by the following function:

```
return :: Tree a  -> [a]
return tree = flatten tree []
```

3 Attribute grammar as a zipper transformer

In order to account for context-dependent information we manipulate a subtree together with its corresponding context. We restrict attention to trees whose sort is the axiom. A zipper (of sort s) is given by a pair made of a tree of sort s together with a context for that tree. The representation of a context in the zipper comes from the following observation: either the context of the considered subtree t is empty or it is of the form

$$op^i(t_1, \ldots, t_{i-1}, C, t_{i+1}, \ldots, t_n) \overset{def}{=} C[op(t_1, \cdots, t_{i-1}, [\,], t_{i+1}, \cdots, t_n)]$$

where $op : s_1 \times \cdots \times s_n \to s$ is an operator such that s_i is the sort of t_i, and C is a context whose hole is of sort s. The trees t_j for $1 \le j \le n$ and $j \ne i$ are the siblings of t. Thus trees and contexts are given by the following signature.

Definition 3.1 In the signature $\mathcal{Z}_\Sigma$ we find two sorts denoted s and $\hat{s}$, associated with each sort $s \in \mathcal{S}$ in Σ and each operator $op :: s_1 \times \cdots \times s_n \to s$ in Σ is also an operator of $\mathcal{Z}_\Sigma$ with the same arity and sort; but it gives also rise to a family of operators op^i for $1 \le i \le n$ where

$$op^i : s_1 \times \cdots \times s_{i-1} \times \hat{s} \times s_{i+1} \times \cdots \times s_n \to \hat{s}_i$$

We finally have a constant operator *Empty* of sort $\hat{a}$ representing the empty context. A zipper $c@t$ of sort s is a pair made of a subtree $t \in Tree(\mathcal{Z}_\Sigma)_s$ together with its context $c \in Tree(\mathcal{Z}_\Sigma)_{\hat{s}}$.

The interpretation of the semantic rule

$$X_\varepsilon \cdot syn \;=\; sem(op)_{\varepsilon,syn}\,(X_\lambda \cdot q; (\lambda, q) \in In_{op})$$

associated with $op \;::\; X_1 \times \cdots \times X_n \longrightarrow X_\varepsilon$ is given by the following inductive rule

$$\begin{aligned} & syn\,(\hat{x}_\varepsilon @ op(x_1, \ldots, x_n)) = \\ & sem(op)_{\varepsilon,syn}\;(q\,(\hat{x}_\varepsilon @ op(x_1, \ldots, x_n))\,/q \in Inh(s)\;; \\ & \qquad\qquad q\left(op^i\,(x_1, \ldots, x_{i-1}, \hat{x}_\varepsilon, x_{i+1}, \ldots, x_n)\,@x_i\right)/q \in Syn(s_i)\;) \end{aligned}$$

whose interpretation is as follows. If the subtree t of zipper $c@t$ matches the pattern $\hat{x}_\varepsilon @ op(x_1, \ldots, x_n)$, *i.e.* $t = op(t_1, \ldots, t_n)$, then the expression $syn(c@t)$, standing for the value of the synthesized attribute syn of subtree t within context c, is given by the expression in the right-hand side where variables $\hat{x}_\varepsilon$ and x_i are replaced respectively by the context c and the subtrees t_i given by pattern matching.

Similarly, the interpretation of the semantic rule

$$X_i \cdot inh \;=\; sem(op)_{i,inh}\,(X_\lambda \cdot q; (\lambda, q) \in In_{op})$$

is given by

$$\begin{aligned} & \mathit{inh}\left(\mathit{op}^i\left(x_1,\ldots,x_{i-1},\hat{x}_\varepsilon,x_{i+1},\ldots,x_n\right)@x_i\right) = \\ & \mathit{sem}(\mathit{op})_{i,\mathit{inh}}\left(q\left(\hat{x}_\varepsilon @\mathit{op}(x_1,\ldots,x_n)\right)/q \in \mathit{Inh}(s)\ ;\right. \\ & \qquad\qquad \left. q\left(\mathit{op}^i\left(x_1,\ldots,x_{i-1},\hat{x}_\varepsilon,x_{i+1},\ldots,x_n\right)@x_i\right)/q \in \mathit{Syn}(s_i)\ \right) \end{aligned}$$

In this case, an inherited attribute appears as an attribute of the context (which is tested against a pattern) relative to a given subtree. If the attribute grammar is rooted the semantic rules associated with the (implicit) operator *Root* are translated as:

$$\mathit{inh}\left(\mathit{Empty}@x\right) = \mathit{init}\left(q\left(\mathit{Empty}@x\right)/q \in \mathit{Syn}(a)\right)_{\mathit{inh}}$$

$$\mathit{return}(x) = \mathit{result}\left(q\left(\mathit{Empty}@x\right)/q \in \mathit{Syn}(a)\right)$$

The Haskell code corresponding to our running example is given as follows.

```
data Tree a = Leaf a | Fork (Tree a)(Tree a)
data Cxt a  = Empty  | LCxt (Cxt a)(Tree a) | RCxt (Tree a)(Cxt a)
data Zipper a = Cxt a :> Tree a

flatten :: Zipper a -> [a]
flatten (cxt :> tree@(Leaf a))  =  a :(coflat (cxt :> tree))
flatten (cxt :> (Fork left right)) = flatten ((LCxt cxt right):>left)

coflat :: Zipper a -> [a]
coflat (CoRoot:>tree)  = []
coflat ((LCxt cxt right):>left) = flatten ((RCxt left cxt):>right)
coflat ((RCxt left cxt):>right) = coflat  (cxt:>(Fork left right))

return :: Tree a -> [a]
return tree = flatten (Empty:>tree)
```

This code is an immediate transcription of the semantic rules, where a synthesized attribute is defined inductively on the structure of the tree component and an inherited attribute is defined inductively on the structure of the component giving the context. However we have in the right-hand side of each rule to update accordingly the various parameters. For instance the rule

```
coflat ((LCxt cxt right):>left) = flatten ((RCxt left cxt):>right)
```

states that the inherited attribute *coflat* when applied to a context of the form `LCxt cxt right` and a subtree `left` is given by the synthesized attribute *flatten* for subtree `right` in the corresponding context, namely `RCxt left cxt`. We can make this extra parameter implicit if each subtree is aware of its context, given as an extra parameter, and symmetrically each context is aware of the subtree to which it is applied. We achieve this result using cyclic representations of zippers which we define in the next section.

4 Zippers as cyclic data structures

First we introduce an extended signature such that the corresponding trees are encodings of cyclic representations of zippers. In a second part we describe the process of unfolding a rooted tree (*i.e.* with an initial empty context) into its cyclic representation.

4.1 Cyclic data structures

Definition 4.1 In the signature $\mathcal{CZ}_\Sigma$ we find two sorts denoted s and $\hat{s}$, associated with each sort $s \in \mathcal{S}$ in Σ and each operator $op : s_1 \times \cdots \times s_n \to s$ gives rise to a family of operators op_λ for $\lambda \in \{\varepsilon\} \cup \{1, \ldots, n\}$ where $op_\varepsilon :: \hat{s} \times s_1 \times \cdots \times s_n \to s$ and $op_i :: \hat{s} \times s_1 \times \cdots \times s_n \to \hat{s}_i$. Finally we have an operator $CoRoot : a \to \hat{a}$ where a is the axiom of Σ. A tree $t \in Tree(\mathcal{CZ}_\Sigma)_s$ is a representation of a subtree of type s and a tree $c \in Tree(\mathcal{CZ}_\Sigma)_{\hat{s}}$ is a representation of a context of type s.

However most of the trees build from this signature $\mathcal{CZ}_\Sigma$ are not valid representations of subtrees or contexts. Let us illustrate this phenomenon with the example of doubly-linked streams. If A is an alphabet, a stream is a tree on the monosorted signature Σ (with sort $\mathcal{S} = \{st\}$) containing one unary operator $a :: st \to st$ for each letter $a \in A$. The tree $a(st)$ stands for the stream whose root node is labelled a and such that the remaining stream obtained by removing this root node is st. The signature $\mathcal{Z}_\Sigma$ provides the associated structure of zipper

```
data Stream a = Cons{val::a, suc::Stream a}
data StreamCxt a = Snoc{val::a, pred::StreamCxt a} | Empty
data StreamZipper a = (StreamCxt a):>(Stream a)
```

The structure of zipper allows to navigate streams non destructively:

```
left, right :: StreamZipper a -> StreamZipper a
right (cxt:>(Cons a str)) = (Snoc a cxt):>str
left  ((Snoc a cxt):>str) =  cxt:>(Cons a str)
```

In order to navigate a stream in both direction we can alternatively add to each node a pointer to the preceding node, leading us to the structure of a doubly-linked stream:

```
data DStream a = Node{val:: a, prev::CxtDStream, suc ::DStream a}
data CxtDStream a = CoRoot (DStream a)
                  | CoNode{val':: a , prev'::CxtDStream a, suc'::DStream a}
```

which is the inductive data structure associated with signature $\mathcal{CZ}_\Sigma$. If we were to consider doubly-linked lists rather than doubly-linked streams then we would just have to add one unary constructor associated with the constant operator associated with the empty list:

```
data DList a = Node{val:: a, prev::CxtDList, suc ::DList a}
             | Nil (CxtDList)
data CxtDList a = CoRoot (DList a)
```

```
                 | CoNode{val':: a , prev'::CxtDList a, suc'::DList a}
```

These two data structures are isomorphic, and by identifying them we obtain a more traditional representation of doubly-linked lists as:

```
data DList a = Node{val:: a, prev,suc ::DList a} | Nil (DList)
```

An implicit assumption is that if `suc xs` is defined then `prev (suc xs)= xs`, and if `prev xs` is defined then `suc (prev xs)= xs`, similarly `prev xs = Nil ys` or `suc xs = Nil ys` entails ys=xs. These conditions are met in the following doubly-linked representation of the list $[1, 2, 3]$

```
dlist = node1
        where node1 = Node 1 (Nil node1) node2
              node2 = Node 2 node1 node3
              node3 = Node 3 node2 (Nil node3)
```

An abstract data type is often presented by a multi-sorted signature together with equational constraints stated in terms of the *constructors* of the signature. They thus constrain the class of valid interpretations to belong to the corresponding equational variety of algebras. The abstract data type is then identified with the initial object of that category; namely, the quotient of the initial algebra by the induced congruence. In the present case, equations are stated in terms of the *selectors* of the signature. They limit the class of valid generators and the abstract data type can be identified with a subcoalgebra of the terminal coalgebra. Elements of this abstract representation can be represented by graphs whose tree unfolding satisfies the equations in the following sense. The set of equational contraints determines a binary relation on the set of nodes of a tree. The tree satisfies the equational contraints if two subtrees rooted at related nodes are the same. The carrier of the abstract data type is then given as the set of trees satisfying the equational constraints; and each such element can be seen as a tree representation of the graph whose nodes are the isomorphic class of its subtrees. Due to this graphical representation we use the expression of *cyclic data structures* to stand for abstract data types defined from a multi-sorted signature and a base of cycles given by a set of equations using the selectors of the signature. It could be interesting to investigate more deeply such a coalgebraic presentation of cyclic data structures [15,20,7,13,18].

In order to generate only doubly-linked lists that are well-formed (*i.e.* that satisfy the above identities) we will exclusively generate them using some stream coalgebra. Such a coalgebra allows to generate streams:

```
data StrCoalg b a = StrCoalg{out::b->a, next::b->b}

streamGen :: StrCoalg b a -> b -> Stream a
streamGen (StrCoalg out next) = build
  where build gen = Cons (out gen)(build (next gen))
```

For instance one can generate the stream of prime numbers using the sieve of Eratosthenes as follows:

```
sieve = StrCoalg head next
  where next xs = filter (\n -> not((n ‘mod‘ (head xs))==0))(tail xs)

primes = streamGen sieve [2..]
```

In order to generate a well-formed double-linked stream from a stream coalgebra we only have to adapt the above definition of the stream generation function by adding a new parameter for handling the context:

```
dStreamGen :: StrCoalg b a -> b -> DStream a
dStreamGen (StrCoalg out next) gen = dstr
  where dstr = build gen (CoRoot dstr)
        build gen cxts = Node (out gen) cxts dstr’
          where dstr’ = build (next gen) (CoNode (out gen) cxts dstr’)

dprimes = dStreamGen sieve [2..]
```

Then we derive a function translating a stream into a corresponding double-linked stream:

```
stream2dStream :: Stream a -> DStream a
stream2dStream = dStreamGen (StrCoalg val suc)
```

Or equivalently by expanding the definition of function *dStreamGen*:

```
stream2dStream str  = dstr
  where dstr = build str (CoRoot dstr)
        build (Cons val suc) cxts = Node val cxts dstr’
           where dstr’ = build suc (CoNode val cxts dstr’)
```

Now one can navigate a doubly-linked stream:

```
right (Node a cxts dstr) = dstr
left  (Node _ (CoNode b cxts dstr) _) = Node b cxts dstr

first :: Int -> DStream a -> [a]
first 0 str = []
first (n+1) (Node a cxts dstr)= a:(first n dstr)

test = first 5 ((left.right.right.left.right.right.right.right) dprimes)
> test
[11,13,17,19,23]
```

4.2 Unfolding of a tree

In this section we generalize on the previous example of doubly-linked streams to present a translation of trees into zippers. For that purpose we introduce an attribute grammar canonically associated with a given signature.

Definition 4.2 The rooted attribute grammar η_Σ associated with signature $\Sigma = (\mathcal{S}, \mathcal{O}_p)$ and axiom $a \in \mathcal{S}$ is defined as follows. It has one inherited attribute associated with each sort $Inh = \{cxt_s | s \in \mathcal{S}\}$ representing the context at the given node of the tree, and one synthesized attribute $Syn = \{trees_s | s \in \mathcal{S}\}$ representing the subtree rooted at that node, with arities and sorts given by $trees_s : s \to s$ and $cxt_s : s \to \hat{s}$. The semantic domains are given by $\mathcal{D}_{trees_s} = Tree(\mathcal{CZ}_\Sigma)_s$ and $\mathcal{D}_{cxt_s} = Tree(\mathcal{CZ}_\Sigma)_{\hat{s}}$. The semantic rules associated with operator $op : s_1 \times \cdots \times s_n \to s$ are given by

$$op :: X_1 \times \cdots \times X_n \to X_\varepsilon$$

$$\begin{cases} X_\varepsilon \cdot trees_s &= op_\varepsilon\,(X \cdot cxt_s, X_1 \cdot trees_{s_1}, \ldots, X_n \cdot trees_{s_n}) \\ X_i \cdot cxt_{s_i} &= op_i\,(X \cdot cxt_s, X_1 \cdot trees_{s_1}, \ldots, X_n \cdot trees_{s_n}) \end{cases}$$

and the auxiliary functions $result : Tree(\mathcal{CZ}_\Sigma)_a \to Tree(\mathcal{CZ}_\Sigma)_a$ and $init : Tree(\mathcal{CZ}_\Sigma)_a \to Tree(\mathcal{CZ}_\Sigma)_{\hat{a}}$ are respectively the identity and the operator $CoRoot$.

We let $\mathcal{U}$, for "unfolding", denote the algebra induced by attribute grammar η_Σ The interpretation of operator op is thus given by

$$\begin{aligned} op^{\mathcal{U}}(f_1, \ldots, f_n)\; cxt &= op_\varepsilon\,(cxt, tree_1, \ldots, tree_n) \\ \text{where}\quad tree_i &= f_i\,(op_i\,(cxt, tree_1, \ldots, tree_n)) \end{aligned}$$

and we let unfold denote the corresponding top level function:

$$unfold :: Tree(\Sigma)_a \to Tree(\mathcal{CZ}_\Sigma)_a$$

$$unfold(tree) \;=\; ctree \quad \text{where}\; ctree \;=\; t^{\mathcal{U}}(CoRoot(ctree))$$

Thus the unfolding function can be written as:

$$\begin{aligned} &unfold\;\; tree \;=\; ctree \\ &\quad \text{where}\; ctree \;=\; build_a\;\; tree\;\; (CoRoot\; ctree) \\ &\qquad build_s\,(op(t_1, \ldots, t_n))\;\; cxt \;=\; op_\varepsilon\,(cxt, tree_1, \ldots, tree_n) \\ &\qquad\quad \text{where}\; tree_i \;=\; build_{s_i}\;\; t_i\;\; op_i\,(cxt, tree_1, \ldots, tree_n) \end{aligned}$$

In our example of binary trees the unfolding can be given as follows:

```
data Tree  a  = Leaf a | Fork (Tree a) (Tree a)
data ZTree a = ZLeaf a (ZCxt a) | ZFork  (ZCxt a) (ZTree a) (ZTree a)
data ZCxt  a = CoRoot (ZTree a) | ZLeft  (ZCxt a) (ZTree a) (ZTree a)
                                | ZRight (ZCxt a) (ZTree a) (ZTree a)


unfold :: Tree a -> ZTree a
unfold tree = ctree
 where ctree = build tree (CoRoot ctree)
       build (Leaf a) cxt = ZLeaf a cxt
       build (Fork left right) cxt = ZFork cxt cleft cright
```

```
         where cleft  = build left  (ZLeft  cxt cleft cright)
               cright = build right (ZRight cxt cleft cright)
```

In the example of doubly-linked lists we introduced this unfolding function as a special case of the function generating a doubly-linked list from a list coalgebra. The same can be done by using an adaptation of the above unfolding function:

```
data Trunk  a b = Leaf a | Fork b b
data Tree a = In{out:: Trunk a (Tree a)}
data TreeCoalg c a = TreeCoalg{next::c-> Trunk a c}

punfold :: TreeCoalg c a -> c -> ZTree a
punfold coalg gen = ctree
 where ctree = build gen (CoRoot ctree)
       build gen cxt = case next coalg gen of
         Leaf a -> ZLeaf a cxt
         Fork left right -> ZFork cxt cleft cright
          where cleft  = build left  (ZLeft  cxt cleft cright)
                cright = build right (ZRight cxt cleft cright)

unfold = punfold (TreeCoalg out)
```

More generally this parametric unfolding will be given as:

$$\begin{array}{l} \mathit{punfold}\ \ \mathit{coalg}\ \ \mathit{gen}\ =\ \mathit{tree} \\ \quad \text{where}\ \mathit{tree}\ =\ \mathit{build}_a\ \ \mathit{gen}\ \ (\mathit{CoRoot\,tree}) \\ \qquad\qquad \mathit{build}_s\ \ \mathit{gen}\ \ \mathit{cxt}\ =\ \mathit{case}\ \ \mathit{coalg}\ \ \mathit{gen}\ \ \mathit{of} \\ \qquad\qquad\quad \mathit{op}(\mathit{gen}_1,\ldots,\mathit{gen}_n)\ \rightarrow\ \mathit{op}_\varepsilon\,(\mathit{cxt},\mathit{tree}_1,\ldots,\mathit{tree}_n) \\ \qquad\qquad\qquad\quad \text{where}\ \mathit{tree}_i\ =\ \mathit{build}_{s_i}\ \ \mathit{gen}_i\ \ \mathit{op}_i\,(\mathit{cxt},\mathit{tree}_1,\ldots,\mathit{tree}_n) \end{array}$$

5 First-order functional interpretation of an attribute grammar

We associate a rooted attribute grammar $\mathbb{G}$ with a $\mathcal{CZ}_\Sigma$-algebra $\mathcal{A}^\sharp_\mathbb{G}$ where

$$\left(\mathcal{A}^\sharp_\mathbb{G}\right)_s = \mathcal{D}^\uparrow_s \text{ and } \left(\mathcal{A}^\sharp_\mathbb{G}\right)_{\hat{s}} = \mathcal{D}^\downarrow_s$$

and

$$\begin{aligned} op_\varepsilon^{\mathcal{A}^\sharp_\mathbb{G}}(v)(syn) &= sem(op)_{\varepsilon,syn}(v) \\ op_i^{\mathcal{A}^\sharp_\mathbb{G}}(v)(inh) &= sem(op)_{i,inh}(v) \\ CoRoot^{\mathcal{A}^\sharp_\mathbb{G}} &= init \end{aligned}$$

where $op : s_1 \times \cdots \times s_n \to s$, $v \in \mathcal{D}_s^{\downarrow} \times \mathcal{D}_{s_1}^{\uparrow} \times \cdots \times \mathcal{D}_{s_n}^{\uparrow}$, $syn \in Syn(s)$, and $inh \in Inh(s_i)$.

Proposition 5.1 $(unfold \quad tree)^{\mathcal{A}_{\mathbb{G}}^{\sharp}} = val \quad where\ val = t^{\mathcal{A}_{\mathbb{G}}}(init(val))$

Proof. Straightforward, details may be found in [3]. □

Therefore $result\left((unfold \quad tree)^{\mathcal{A}_{\mathbb{G}}^{\sharp}}\right)$ coincides with the value *return tree* returned by the rooted attribute grammar $\mathbb{G}$ from the given input tree. It gives an algorithm for computing that value by simple structural recursion on the unfolding of the input tree. In our running example, we obtain the following Haskell code.

```
data Tree  a  = Leaf a | Fork (Tree a) (Tree a)
data ZTree a  = ZLeaf a (ZCxt a) | ZFork  (ZCxt a) (ZTree a) (ZTree a)
data ZCxt  a  = CoRoot (ZTree a) | ZLeft  (ZCxt a) (ZTree a) (ZTree a)
                                 | ZRight (ZCxt a) (ZTree a) (ZTree a)


unfold :: Tree a -> ZTree a
unfold tree = ctree ....

flatten :: ZTree a -> [a]
flatten (ZLeaf a cxt) = a:(coflat cxt)
flatten (ZFork cxt left right) = flatten left

coflat :: ZCxt a -> [a]
coflat (CoRoot tree) = []
coflat (ZLeft  cxt left right) = flatten right
coflat (ZRight cxt left right) = coflat cxt

return :: Tree a -> [a]
return tree = flatten (unfold tree)
```

6 Conclusion

We have presented a new interpretation of attribute grammars (with both inherited and synthesized attributes), based on Huet's zipper datatype. More precisely we introduced a signature $\mathcal{CZ}_\Sigma$ for representing zippers as cyclic data structures, together with an unfolding operation transforming a Σ-tree into their cyclic representation (a $\mathcal{CZ}_\Sigma$-tree). An attribute grammar on signature Σ can immediately be identified with an $\mathcal{CZ}_\Sigma$-algebra so that attribute values can be computed by structural induction on the unfolding of the input tree.

If the domain of interpretation of attributes are trees over an output (or semantic) signature, and if the semantic rules are accordingly given by expressions build on this output signature, then an attribute grammar can be interpreted as a tree transformer. In this context, our result amounts to transform an attribute grammar into a (deterministic, top-down) tree transducer with input signature $\mathcal{CZ}_\Sigma$. It is much easier to compose (top-down) tree transducers than to compute the syntactic composition of attribute grammars. The latter operation introduced by Ganzinger and Giegerich, as the co-called *attributed coupled grammars* [12], has already been related to the functional programming deforestation technique in [8,10]. We would like to recover the syntactic composition of attribute grammars through the composition of the associated (top-down) tree transducers acting on cyclic representations of zippers.

We can notice that the Haskell code we ended with is almost an immediate transcription of the semantic rules of the attribute grammar. Still the programmer need to be aware of the underlying cyclic representation of zippers and this is an undesirable overhead and a potential source of programming errors. We would like to be able to encapsulate these aspects into a structure of monad (or a structure of arrows) so that all these considerations would be totally transparent to the programmer. we are thus looking for a set of functional combinators (similar to the functional monadic parser combinators [17]) providing an attribute grammar designer with a Domain Specific Language embedded in Haskell. Using these combinators it would specify an attribute grammar (mainly by writing down semantic rules) but by doing so he would actually build an Haskell program for the corresponding evaluator of attributes or for related tools.

References

[1] Augusteijn, A., “Functional Programming, Program Transformations and Compiler Construction”, PhD thesis, Technische Universiteit Eindhoven, 1993.

[2] Backhouse, K. S., *A functional semantics of attribute grammars*, in: J.-P. Katoen and P. Stevens, eds., “Proc. of 8th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems, TACAS 2002 (Grenoble, Apr. 2002),” Lecture Notes in Computer Science **2280**, Springer, 2002, pp. 142–157.

[3] Badouel, E., B. Fotsing and R. Tchougong, *Yet another implementation of attribute evaluation,* Research Report No. 6315, INRIA, 2007. Available at `https://hal.inria.fr/inria-00175810`.

[4] Badouel, E. and M. Tonga, *Growing a domain specific language with split extensions,* Research Report No. 6314, INRIA, 2007. Available at `https://hal.inria.fr/inria-00175805`.

[5] Bird, R., *Using circular programs to eliminate multiple traversals of data*, Acta Inform. **21** (1984), 239–250.

[6] Chirica, L. M. and D. F. Martin, *An order-algebraic definition of Knuthian semantics*, Math. Syst. Theory **13** (1979), pp. 1–27.

[7] Clack, C., S. Clayman and D. Parrott, *Dynamic cyclic data structures in lazy functional languages,* draft, 1995. Available at `http://www.cs.ucl.ac.uk/staff/C.Clack/papers/NotSubmitted/graph.ps`.

[8] Correnson, L., E. Duris, D. Parigot, and G. Roussel, *Declarative program transformation: a deforestation case-study*, in: G. Nadathur, ed., “Proc. of Int. Conf. on Principles and Practice of Declarative Programming, PPDP ’99 (Paris, Sept./Oct. 1999),” Lecture Notes in Computer Science **1702**, Springer, 1999, pp. 360–377.

[9] Courcelle, B. and P. Franchi-Zannettacci, *Attribute grammars and recursive program schemes, I*, Theor. Comput. Sci. **17** (1982), pp. 163–191; *Attribute grammars and recursive program schemes, II*, Theor. Comput. Sci. **17** (1982), pp. 235–257.

[10] Duris, E., D. Parigot, G. Roussel, and M. Jourdan, *Structured-directed genericity in functional programming and attribute grammars,* Research Report No. 3105, INRIA, 1997.

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

[12] Ganzinger, H. and R. Giegerich, *Attribute coupled grammars*, in: "Proc. of 1984 SIGPLAN Symp. on Compiler Construction (Montréal, June 1984)", ACM Press, 1984, pp. 157–170.

[13] Ghani, N., M. Hamana, T. Uustalu and V. Vene, *Representing cyclic structures as nested datatypes*, in: H. Nilsson, ed., "Proc. of 7th Symp. on Trends in Functional Programming (Nottingham, Apr. 2006),", Univ. of Nottingham, 2006, pp. 173–188.

[14] Goguen, J., J. W. Thatcher, E. G. Wagner and J. B. Wright, *Initial algebra semantics and continuous algebras*, J. of ACM **24**(1) (1977), pp. 68–95.

[15] The Haskell Wiki, *Tying the knot: how to build a cyclic data structure,* wiki note, 2002. Available at `http://haskell.org/wikisnapshot/TyingTheKnot.html`.

[16] Huet, G., *The zipper*, J. of Funct. Programming **7**(5) (1997), pp. 549–554.

[17] Hutton, G. and E. Meijer, *Monadic parsing in Haskell*, J. of Funct. Program. **8**(4) (1998), pp. 437–444.

[18] Johann, P. and N. Ghani, *Initial algebra semantics is enough!*, in: S. Ronchi Della Rocca, ed., "Proc. of 8th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2007 (Paris, June 2007)," Lecture Notes in Computer Science **4583**, Springer, 2007, pp. 207–222.

[19] Johnsson, T., *Attribute grammars as functional programming paradigm*, in: G. Kahn, ed., "Proc. of 3rd Int. Conf. on Functional Programming and Computer Architecture, FPCA '87 (Portland, OR, Spept. 1987)," Lecture Notes in Computer Science **274**, Springer, 1987, pp. 154–173.

[20] Klarlund, N. and M. I. Schwartzbach, *Graph types*, in: "Conf. Record of 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '93 (Charleston, SC, Jan. 1993)," ACM Press, 1993, pp. 196–205.

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

[22] Kuiper, M. and S. D. Swierstra, *Using attribute grammars to derive efficient functional programs*, Technical Report RUU-CS-86-16, Universiteit Utrecht, 1986. Available at `http://www.cs.uu.nl/research/techreps/RUU-CS-86-16.html`.

[23] Mayoh, B., *Attribute grammars and mathematical semantics*, SIAM J. of Comput. **10** (1981), pp. 503–518.

[24] Paakki, J., *Attribute grammar paradigms: a high-level methodology in language implementation*, ACM Comput. Surv. **27**(2) (1995), pp. 196–255.

[25] Swierstra, S. D., P. R. Azero Alcocer and J. Saraiva, *Designing and implementing combinator languages*, in: S. D. Swierstra, P. R. Henriques and J. N. Oliveira, eds., "Revised Lectures from 3rd Int. School on Advanced Functional Programming, AFP '98 (Braga, Sept. 1998)," Lecture Notes in Computer Science **1608**, Springer, 1999, pp. 150–206. See also UUAGG (Utrecht University Attribute Grammar Compiler), `http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem`.

[26] Uustalu, T. and V. Vene, *Comonadic functional attribute evaluation*, in: M. van Eekelen, ed., "Trends in Functional Programming 6," Intellect, 2007, pp. 145–162.