Attribute grammars as tree transducers and their descriptional composition

Eric Badouel[1,2], Rodrigue Tchougong[2,3], Célestin Nkuimi-Jugnia[4], Bernard Fotsing[2,5]

1: Inria, Campus Universitaire de Beaulieu, F35042 Rennes Cedex, France
2: LIRIMA, Yaoundé Cameroon
3: University of Ngaoundéré, Cameroon
4: Faculty of Sciences, University of Yaoundé I, Cameroon
5: IUT Victor Fotso, Bandjoun, Cameroon
eric.badouel@inria.fr, rodrigtn@yahoo.fr, nkuimi@yahoo.co.uk, bfotsing@yahoo.fr

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 evaluation can straightforwardly be implemented in a higher-order lazy functional language like Haskell. The resulting evaluation functions are, however, not easily amenable to optimization when we want to compose two attribute grammars. We present an alternative first-order functional interpretation of attribute grammars where the input tree is replaced by an extended cyclic tree each node of which is aware of its context viewed as an additional child tree. These cyclic representations of zippers (trees with their context) are natural generalizations of doubly-linked lists to trees over an arbitrary signature. Then we show that, up to that representation, descriptional composition of attribute grammars reduces to the composition of tree transducers.

1 Introduction

Attribute grammars [22, 29] were introduced to make possible the manipulation of context-sensitive information, like the scope of a variable in a program. This formalism, encountered 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 an input (syntax) tree into some semantic domain. These two problems are related since the result of syntax-directed computation is usually given by the value 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 optimized implementations distinct algorithmic solutions have often been put forward for these two situations. However when 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 the sharing of subexpressions at different places.

The input trees of an attribute grammar are the abstract-syntax trees of an underlying context-free grammar. This set is a regular set of trees which may be identified with the well-sorted terms for some multi-sorted signature whose operators are in bijective correspondance with the set of productions of the grammar. Values of attributes are given by a set of equations described 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 compute the value of each attribute using a topological sort of the dependency graph (whose arcs indicate the dependencies between attribute values). One can alternatively use an order-algebraic approach based on least fixed-points [24, 26] in order to compute attributes for potentially circular attribute grammars (and on potentially infinite input trees).

Attribute grammars use both synthesized and inherited attributes which respectively bear information 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 [8] 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 which 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, 23, 12, 4] leads to efficient implementations in a higher-order lazy functional language like Haskell. The Elegant system developped at Philips [3] and the UUAG system [31] 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 by 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 [17], 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 $\mathcal{Z}_{ipper}(\Sigma)$ whose corresponding trees are associated with Σ-trees or with their contexts. A zipper is given 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 $\mathbf{Z}(\Sigma)$ 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 [1] 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. An attribute grammar on a signature Σ proves to coincide with an $\mathbf{Z}(\Sigma)$-algebra, i.e. an ordinary algebra on an extended signature such that trees sorted over that signature are cyclic representation of zippers.

As already mentionned, attribute grammars [22] were first introduced to model syntax-directed semantics. Latter, Fülöp [13] has defined attributed tree transducers as abstractions of attribute grammars and Maneth [25] showed that attribute grammars and attributed tree transducers have the same expressive power. In this paper attribute grammars are presented as attributed tree transducers in the sense that they translate trees over an input signature Σ into trees over an output signature Σ'. A particular instance is a tree transducer which is an attribute grammar with synthesized attributes only.

Ganzinger and Giegerich [14, 15] introduced *descriptional composition* as a syntactic composition of *attribute coupled grammars*, a variant of attributed tree transducers. The condition imposed on attribute grammars, called *syntactic single use requirement*, or the corresponding condition on attributed tree transducers, called *single use restriction* [20, 21], means that every attribute instance in a tree may be used at most once in the evaluation of some other attributes. We could have considered attribute grammars as graph transformations (at least as transformation on dags, where such a dag is obtained by sharing some common sub-expressions) with conditions, similar to (and slighly less restrictive than) single use requirement. However for simplicity of presentation we will stick to attribute grammars viewed as tree transformations subject to the single use requirement.

Descriptional composition of attribute grammars has been related to various techniques of functional program optimization, see e.g. [10, 11], where a function that produces some data is fused with a function that consumes it so that the intermediate data structure no longer need to be built and destroyed: deforestation [34], lazy composition [32] , pfold/buildp rule [2], pfold/Mbuild rule in the context of monadic programming [30], composition of functions with accumulating parameters [33],*etc*. A tree transducer can be viewed as a (confluent and terminating) rewriting system where the image of a (finite) tree can be obtained by reduction to its normal form. One can then very easily statically compute the composition (fusion) of two tree transducers by replacing each right-hand side of a rewriting rule of the former by its normal form for the latter. Descriptional composition of attribute grammars is much harder to understand (and to implement!). Hence the idea of using our representation of an attribute grammar G from input signature Σ to output signature Σ' as a tree transducer $G^\sharp$ from $\mathbf{Z}(\Sigma)$ into Σ' in order to reduce their descriptional composition as the composition of the corresponding transducers. For that purpose we associate each attribute grammar $G : \Sigma \to \Sigma'$ with a derived attribute grammar $G^\triangle : \Sigma \to \mathbf{Z}(\Sigma')$ obtained by splitting each attribute of the original grammar into two distinct attributes corresponding respectively to the value of the attribute and the value of the context in which it takes place. We show that the resulting tree transducers $\mathbf{Z}(G) = G^{\triangle\sharp} : \mathbf{Z}(\Sigma) \to \mathbf{Z}(\Sigma')$ are such that their composition reflects descriptional composition:

$$\mathbf{Z}(G_2 \copyright G_1) = \mathbf{Z}(G_2) \cdot \mathbf{Z}(G_1)$$

From this construction we derive a very simple proof for the associativity of descriptional composition as well as for its correctnesss, namely that

$$[\![G_2 © G_1]\!] = [\![G_2]\!] \circ [\![G_1]\!]$$

where $[\![G]\!]$ denote the tree transformation induced by attribute grammar G.

The paper contains three main sections. In Section 2 we first recall the basic notations on multi-sorted signatures and their algebras and on attribute grammars. Then we present the higher-order implementation of an attribute grammar viewed as (the presentation of) an algebra, for the underlying signature, where the domain of interpretation associated with each sort is the set of functions from the set of values of their inherited attributes to the set of values of their synthesized attributes. In Section 3 we provide an alternative interpretation of an attribute grammar as a first-order algebra for a signature based on cyclic representations of zippers. Section 4 uses this representation to present descriptional composition of attribute grammars as the composition of associated tree transducers. Then we present a detailed example of descriptional composition using tree transducers before the concluding section.

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 [16, 7] for a more detailed presentation); then we proceed to the definition of an attribute grammar and its associated higher-order interpretation. We conclude this section by introducing the notion of a rooted attribute grammar

2.1 Signature and algebra

Definition 1. *A (multi-sorted) signature $\Sigma = (S, \Omega)$ consists of a finite set S of sorts, and a finite set Ω of operators. Each operator $\omega \in \Omega$ has a type $\tau(\omega) \in S^* \times S$ which we also denote as $\omega : s_1 \dots s_n \to s$. The type can be decomposed into its two components providing respectively the arity $ar(\omega) \in S^*$ and sort $\sigma(\omega) \in S$ of operator ω. We let $\Omega(s_1 \dots s_n, s)$ denote the set of operators of type $s_1 \dots s_n \to s$.*

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

$$\begin{array}{lll} Fork & & : Tree \times Tree \to Tree \\ Leaf & label : & \to Tree \end{array}$$

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. *Let $\Sigma = (S, \Omega)$ 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 $\omega^{\mathcal{A}} : \mathcal{A}_{s_1} \times \dots \times \mathcal{A}_{s_n} \to$*

$\mathcal{A}_s$ associated with each operator $\omega \in \Omega(s_1 \dots s_n, s)$. 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 $a_i \in \mathcal{A}_{s_i}$ one has $f_s\left(\omega^{\mathcal{A}}(a_1, \dots, a_n)\right) = \omega^{\mathcal{B}}\left(f_{s_1}(a_1), \dots, f_{s_n}(a_n)\right)$

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 $t : \mathbb{N}^* \to \Omega$ whose domain $Dom(t)$ is a non-empty prefix-closed language such that for every $u \in Dom(t)$ with $t(u) \in \Omega(s_1 \dots s_n, s)$, 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 3. *An (abstract)* attribute grammar $\mathbb{G} = (\Sigma, Attr, q_0, \mathcal{D}, sem)$ *is a signature $\Sigma = (S, \Omega)$ with a specified sort $a \in S$, called the* ***axiom****. Each grammatical symbol (sort) is associated with a set of attributes in which we distinguish* inherited attributes *from* synthesized attributes*: $Attr(s) = Inh(s) \uplus Syn(s)$. $\mathcal{D} = (\mathcal{D}_q)_{q \in Attr}$ where $\mathcal{D}_q$ is a complete partial order, the so-called domain of evaluation associated with attribute $q \in Attr(s)$. $q_0 \in Syn(a)$ is a specific attribute of the axiom, called* ***exit attribute****, whose value is the result computed by the attribute grammar from a Σ-tree of sort a. 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:*

$$sem(\omega) : \mathcal{D}_s^{\downarrow} \times \mathcal{D}_{s_1}^{\uparrow} \times \dots \times \mathcal{D}_{s_n}^{\uparrow} \to \mathcal{D}_s^{\uparrow} \times \mathcal{D}_{s_1}^{\downarrow} \times \dots \times \mathcal{D}_{s_n}^{\downarrow}$$

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 let an *input attribute* of an operator $\omega \in \Omega(s_1 \dots s_n, s)$ be 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 $sem(\omega)$ associated with operator ω 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*).

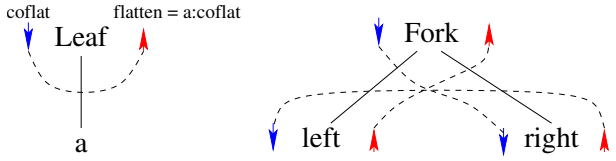

Fig. 1. an attribute grammar for computing the frontier of a binary tree

$$\begin{array}{l} Leaf \quad a \quad :: \quad \longrightarrow Tree_\varepsilon \\ \left\{ Tree_\varepsilon \cdot flatten = a : (Tree_\varepsilon \cdot coflat) \right. \end{array}$$

$$\begin{array}{l} Fork \quad :: \quad Tree_1 \times Tree_2 \longrightarrow Tree_\varepsilon \\ \left\{ \begin{array}{l} Tree_\varepsilon \cdot flatten = Tree_1 \cdot flatten \\ Tree_1 \cdot coflat \ = Tree_2 \cdot flatten \\ Tree_2 \cdot coflat \ = Tree_\varepsilon \cdot coflat \end{array} \right. \end{array}$$

In order to present the semantic rules we need to distinguish the different occurrences of a same grammatical symbol (sort). For that purpose, if $\omega : s_1 \ldots s_n \to s$ is an operator we let $\omega :: (s_1)_1 \times \cdots \times (s_n)_n \to s_\varepsilon$ 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 $\omega :: X_1 \times \cdots \times X_n \to X_\varepsilon$ where X_i is an abreviation for $(s_i)_i$ and $X_\varepsilon = s_\varepsilon$. Then the semantic rules attached to an operator ω are of the form

$$\begin{array}{l} \omega \quad :: X_1 \times \cdots \times X_n \quad \longrightarrow X_\varepsilon \\ \left\{ \begin{array}{l} X_\varepsilon \cdot syn = sem(\omega)_{\varepsilon,syn}\,(X_\lambda \cdot q; (\lambda, q) \in In_\omega) \\ X_i \cdot inh \ = sem(\omega)_{i,inh}\,(X_\lambda \cdot q; (\lambda, q) \in In_\omega) \end{array} \right. \end{array}$$

where $syn \in Syn(s)$ and $inh \in Inh(s_i)$ and

$$\begin{array}{l} In_\omega \ = \{(\varepsilon, q) \mid q \in Inh(s)\} \bigcup \{(i, q) \mid 1 \le i \le n \quad q \in Syn(s_i)\} \\ Out_\omega = \{(\varepsilon, q) \mid q \in Syn(s)\} \bigcup \{(i, q) \mid 1 \le i \le n \quad q \in Inh(s_i)\} \end{array}$$

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. 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

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

Thus each tree is associated with a system of equations whose variables are the occurrences of attributes (where π ranges over $Dom(t)$)

$$V_t = \left\{ v_{t,\pi,q} \mid t(\pi) : s_1 \dots s_n \to s \ ; \ q \in Syn(s) \right\} \\ \cup \left\{ v_{t,\pi\cdot i,q} \mid t(\pi) : s_1 \dots s_n \to s \ ; \ q \in Inh(s_i) \right\}$$

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:

$$\begin{array}{lll} (\!|t|\!)_{\mathbb{G}}(v)(q) & = & v_{t,\varepsilon,q} \quad \text{where} \\ v_{t,\pi,q} & = v(q) & q \in Inh(\sigma(t(\varepsilon))) \\ v_{t,\pi,q} & = sem(\omega)_{\varepsilon,q}\,(v_{t,\pi,in}) & q \in Syn(s) \\ v_{t,\pi\cdot i,q} & = sem(\omega)_{i,q}\,(v_{t,\pi\cdot i,in}) & q \in Inh(s_i) \end{array}$$

where $v_{t,\pi,in} = \left(v_{t,\pi\cdot\lambda,q}\right)_{(\lambda,q)\in In_\omega}$. We assume 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.

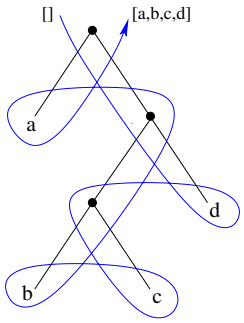

Fig. 2. computing the frontier of a binary tree with an attribute grammar

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 $\mathcal{A}_{\mathbb{G}}$.

Definition 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 $\omega \in \Omega(s_1 \dots s_n, s)$ is the map $\omega^{\mathcal{A}_{\mathbb{G}}}$ given for $f_i : \mathcal{D}_{s_i}^{\downarrow} \to \mathcal{D}_{s_i}^{\uparrow}$, $v \in \mathcal{D}_s^{\downarrow}$ and $q \in Syn(s)$ by:*

$$\begin{array}{lll}
\omega^{\mathcal{A}_{\mathbb{G}}}(f_1, \dots, f_n)(v)(q) & = & v_{\omega,\varepsilon,q} \quad \textit{where} \\
v_{\omega,\varepsilon,q} = v(q) & & q \in Inh(s) \\
v_{\omega,\varepsilon,q} = sem(\omega)_{\varepsilon,q}\left(v_{\omega,in}\right) & & q \in Syn(s) \\
v_{\omega,i,q} = sem(\omega)_{i,q}\left(v_{\omega,in}\right) & & q \in Inh(s_i) \\
v_{\omega,i,q} = f_i(v_i)(q) & & q \in Syn(s_i)
\end{array}$$

where $v_{\omega,in} = \left(v_{\omega,\lambda,q}\right)_{(\lambda,q)\in In_\omega}$ and $v_i(q) = v_{\omega,i,q}$ for $q \in Inh(s_i)$.

This definition is circular [9] since in the "where" clause the inherited attributes $v_i(q') = v_{\omega,i,q'}$ 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 $\langle v_{\omega,\lambda,q}\rangle_{(\lambda,q)\in In_\omega \cup Out_\omega}$ as the least fixed-point of the corresponding transformation.

Proposition 1. $(\!|t|\!)_{\mathbb{G}} = t^{\mathcal{A}_{\mathbb{G}}}$

Proof. By continuity it is enough to verify this identity on terms $t \in T(\Sigma)_s$. That verification proceeds by induction on the structure of t. If t is of the form $t = \omega(t_1, \dots, t_n)$ for some operator $\omega \in \Omega(s_1 \dots s_n, s)$, its associated set of variables V_t can be decomposed as $V_t = \left\{v_{t,\varepsilon,q} \mid q \in Att(\sigma(t))\right\} \cup (\bigcup \left\{v_{t,i\cdot\pi,q} \mid v_{t_i,\pi,q} \in V_{t_i}\right\})$ By identification of variables $v_{t,i\cdot\pi,q}$ and $v_{t_i,\pi,q}$ the system of equation E_{t_i} appears as a subsystem of E_t. Thus, assuming by induction that we have

$$v_{t,i,q} \;=\; v_{t_i,\varepsilon,q} \;=\; (\!|t_i|\!)_{\mathcal{A}_{\mathbb{G}}}(v_i)(q) \;=\; t_i^{\mathcal{A}_{\mathbb{G}}}(v_i)(q)$$

we deduce, by applying Bekič principle for the computation of the least fixed-point of a system of equations by substitutions, that for $q \in Syn(s)$ and $v \in \prod_{q\in Inh(s)}(\mathcal{A}_{\mathbb{G}})_{\sigma(q)}$ it comes

$$\begin{array}{lll}
(\!|t|\!)_{\mathbb{G}}(v)(q) & = & v_{t,\varepsilon,q} \quad \text{where} \\
v_{t,\varepsilon,q} = v(q) & & q \in Inh(s) \\
v_{t,\varepsilon,q} = sem(\omega)_{\varepsilon,q}\left(v_{t,in}\right) & & q \in Syn(s) \\
v_{t,i,q} = sem(\omega)_{i,q}\left(v_{t,in}\right) & & q \in Inh(s_i) \\
v_{t,i,q} = t_i^{\mathcal{A}_{\mathbb{G}}}(v_i)(q) & & q \in Syn(s_i) \\
= \; \omega^{\mathcal{A}_{\mathbb{G}}}\left(t_1^{\mathcal{A}_{\mathbb{G}}}, \dots, t_n^{\mathcal{A}_{\mathbb{G}}}\right)(v)(q) & = & t^{\mathcal{A}_{\mathbb{G}}}(v)(q).
\end{array}$$

where $v_{t,in} = \left(v_{t,\lambda,q}\right)_{(\lambda,q)\in In_\omega}$ and $v_i(q) = v_{t,i,q}$ for $q \in Inh(s_i)$. ■

The above semantics of attribute grammars follows the approaches presented in [19, 4] combined with the fixed-point semantics of [26, 24]. 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 [9]. A translation of attribute grammars into catamorphism (evaluation function for an algebra) was originally presented in [12]. However the presentation given above, inspired from [31], is more explicit 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:

$$
\begin{array}{lll}
(\omega(t_1,\ldots,t_n))^{\mathcal{A}_{\mathbb{G}}}(v)(q) & = & v_{\omega,\varepsilon,q} \\
\text{where} & v_{\omega,\varepsilon,q} = v(q) & q \in Inh(s) \\
& v_{\omega,\varepsilon,q} = sem(\omega)_{\varepsilon,q}\,(v_{\omega,in}) & q \in Syn(s) \\
& v_{\omega,i,q} = sem(\omega)_{i,q}\,(v_{\omega,in}) & q \in Inh(s_i) \\
& v_{\omega,i,q} = t_i^{\mathcal{A}_{\mathbb{G}}}(v_i)(q) & q \in Syn(s_i)
\end{array}
$$

where $v_{\omega,in} = (v_{\omega,\lambda,q})_{(\lambda,q)\in In_\omega}$ and $v_i(q) = v_{\omega,i,q}$ for $q \in Inh(s_i)$. It is convenient to implement the evaluation of attributes as a set of functions, one associated with each synthesized attribute:

Remark 1. $t^{\mathcal{A}_{\mathbb{G}}}(v)(q) = build^{\mathbb{G}}_{s,q}(t)(v)$ where maps $build^{\mathbb{G}}_{s,q} : Tree(\Sigma)_s \to (\mathcal{D}^{\downarrow}_s \to \mathcal{D}_q)$ are defined by mutual recursion as:

$$
\begin{array}{lll}
build^{\mathbb{G}}_{s,q}(\omega(t_1,\ldots,t_n))(v) & = & sem(\omega)_{\varepsilon,q}(v_{\omega,in}) \\
\text{where} & v_{\omega,\varepsilon,q} = v(q) & q \in Inh(s) \\
& v_{\omega,\varepsilon,q} = sem(\omega)_{\varepsilon,q}\,(v_{\omega,in}) & q \in Syn(s) \\
& v_{\omega,i,q} = sem(\omega)_{i,q}\,(v_{\omega,in}) & q \in Inh(s_i) \\
& v_{\omega,i,q} = build^{\mathbb{G}}_{s_i,q}(t_i)(v_i) & q \in Syn(s_i)
\end{array}
$$

where $v_{\omega,in} = (v_{\omega,\lambda,q})_{(\lambda,q)\in In_\omega}$ and $v_i(q) = v_{\omega,i,q}$ for $q \in Inh(s_i)$.

For instance the Haskell code corresponding to our example is given in Table 1, which,

Table 1. higher-order implementation of `flatten`

```
flatten :: Tree a -> [a] -> [a]
flatten (Leaf a) coflat = a:coflat
flatten (Fork left right) coflat0 = flatten0
    where flatten0 = flatten1
          coflat1  = flatten2
          coflat2  = coflat0
          flatten1 = flatten left  coflat1
          flatten2 = flatten right coflat2
```

after elimination of the copy rules, reduces to

```
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 grammars and Syntax-directed translations

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 $Root : a \rightarrow \top$ where a is the axiom of the grammar and $\top$ an additional sort, with no attribute, together with the associated semantic rules. A tree of the form $Root(t)$ where $t \in Tree(\Sigma)_a$ represents a rooted tree, *i.e.* a tree with an empty context. The semantic rules associated with this additional operator *Root* are thus given by a function $init : \mathcal{D}_a^{\uparrow} \rightarrow \mathcal{D}_a^{\downarrow}$ providing the initialization of the inherited attributes at the root node. We shall not explicitely add the new operator *Root* to the signature but we rather consider that a rooted attribute grammar is an attribute grammar together with the extra initialization function *init*. The top level function is then given by the expression shown in Figure 3. where $result : \mathcal{D}_a^{\uparrow} \rightarrow \mathcal{D}_{q_0}$ is the projection that returns the value of the at-

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

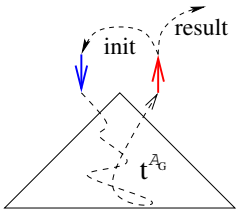

Fig. 3. Tying the knot: the top level function of a rooted attribute grammar

tribute q_0. In our running example the *result* function is the 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 []
```

Definition 5. *A pointed signature is a signature $\Sigma = (S, \Omega)$ together with a specific sort $a \in S$ called its axioms. We let $Tree(\Sigma)$ denote the set of trees whose sort is the axiom; i.e. $Tree(\Sigma) = Tree(\Sigma)_a$. A syntax-directed translation with carrier set a complete partial order $\mathcal{D}$ for a pointed signature Σ is given by a triple $\tau = \langle init, eval, result \rangle$ where $init : \mathcal{S} \rightarrow \mathcal{I}$, $eval : T(\Sigma) \times \mathcal{I} \rightarrow \mathcal{S}$ and $result : \mathcal{S} \rightarrow \mathcal{D}$ are continuous functions (for some complete partial orders $\mathcal{I}$ and $\mathcal{S}$). The induced map $[\![\tau]\!] : Tree(\Sigma) \rightarrow \mathcal{D}$ is given by*

$$
\begin{array}{l}
[\![\tau]\!] : Tree(\Sigma) \longrightarrow \mathcal{D} \\
[\![\tau]\!]\ \ t\ =\ result(val) \qquad \text{where } val\ =\ eval(t, init(val))
\end{array}
$$

The syntax-directed translation associated with a rooted attribute grammar $\mathbb{G}$ is given by $\langle init, eval, result \rangle$ where $eval(t, v) = t^{\mathcal{A}_{\mathbb{G}}}(v)$ and result is the projection associated

with the exit attribute. By abuse of notation we write $[\![\mathbb{G}]\!] : Tree(\Sigma) \to \mathcal{D}$ *the induced evaluation function.*

From now on we shall only consider pointed signatures and rooted attribute grammars and therefore we shall omit these adjectives.

3 First-order interpretation of an attribute grammar

In the previous section we showed that attributes may be evaluated by inductively computing a function expressing the dependencies of the synthesized attributes on inherited attributes. This higher-order functional approach to attribute evaluation can straightforwardly be implemented in Haskell and indeed it is the method used in the tool UUAG developped at Utrecht University [31]. The resulting evaluation functions are, however, not easily amenable to optimization when we want to compose two attribute grammars. When composing two such evaluation functions we would like to be able to eliminate the intermediate data structures used to glue these functions together. We have no general solution for this kind of optimization when higher-order functions are involved. For that purpose, we present in this section an alternative first-order functional interpretation of attribute grammars where the input tree is replaced by an extended cyclic tree each node of which is aware of its context viewed as an additional child tree. These cyclic representations of zippers (trees with their context) are natural generalizations of doubly-linked lists to trees over an arbitrary signature.

3.1 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

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

where $\omega \in \Omega(s_1 \ldots s_n, s)$ is an operator with s_i is the sort of t_i, and C is a context whose hole is of sort s. The trees t_j for $1 \leq j \leq n$ and $j \neq i$ are the siblings of t. Thus trees and contexts are given by the following signature.

Definition 6. *In signature* $\mathcal{Z}_{ipper}(\Sigma)$ *we find two sorts, denoted as* s *and* $\hat{s}$*, associated with each sort* $s \in S$ *in* Σ *and each operator* $\omega : s_1 \ldots s_n \to s$ *in* Σ *is also an operator of* $\mathcal{Z}_{ipper}(\Sigma)$ *with the same arity and sort; but it gives also rise to a family of operators* ω^i *for* $1 \leq i \leq n$ *where* $\omega^i : s_1 \ldots s_{i-1} \cdot \hat{s} \cdot s_{i+1} \ldots s_n \to \hat{s}_i$*. 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* $t \in Tree(\mathcal{Z}_{ipper}(\Sigma))_s$ *together with its context* $c \in Tree(\mathcal{Z}_{ipper}(\Sigma))_{\hat{s}}$*.*

The interpretation of the semantic rule $X_\varepsilon \cdot syn \quad = \quad sem(\omega)_{\varepsilon,syn}\,(X_\lambda \cdot q; (\lambda, q) \in In_\omega)$ associated with $\omega : X_1 \ldots X_n \longrightarrow X_\varepsilon$ is given by the following inductive rule

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

If the subtree t of zipper $c@t$ matches the pattern $\hat{x}_\varepsilon @ \omega(x_1, \ldots, x_n)$, *i.e.* $t = \omega(t_1, \ldots, t_n)$, 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 $X_i \cdot inh \quad = \quad sem(\omega)_{i,inh}\,(X_\lambda \cdot q; (\lambda, q) \in In_\omega)$ is given by

$$\begin{aligned} inh\left(\omega^i\,(x_1, \ldots, x_{i-1}, \hat{x}_\varepsilon, x_{i+1}, \ldots, x_n)\,@\,x_i\right) &= sem(\omega)_{i,inh}(v) \\ \text{where } v(q) &= q\,(\hat{x}_\varepsilon @ \omega(x_1, \ldots, x_n)) && \text{for } q \in Inh(s) \\ v(q) &= q\left(\omega^i\,(x_1, \ldots, x_{i-1}, \hat{x}_\varepsilon, x_{i+1}, \ldots, x_n)\,@\,x_i\right) && \text{for } q \in Syn(s_i) \end{aligned}$$

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

$$\begin{aligned} inh\,(Empty@x) &= init_{inh}\,(q\,(Empty@x)\,; q \in Syn(a)) \\ return(x) &= q_0\,(Empty@x) \end{aligned}$$

The Haskell code corresponding to our running example is given in Table 2. This code

Table 2. first-order implementation of `flatten` using zippers

```
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 (Empty:>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)
```

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 now.

3.2 Zippers as cyclic data structures

Definition 7. *In signature* $\mathbf{Z}(\Sigma)$ *we find two sorts, denoted as* s *and* $\hat{s}$*, associated with each sort* $s \in S$ *in* Σ *and each operator* $\omega : s_1 \dots s_n \to s$ *gives rise to a family of operators* ω_λ *for* $\lambda \in \{\varepsilon\} \cup \{1, \dots, n\}$ *where* $\omega_\varepsilon : \hat{s} \cdot s_1 \dots s_n \to s$ *and* $\omega_i : \hat{s} \cdot s_1 \dots s_n \to \hat{s}_i$*. Finally we have an operator* $Init : a \to \hat{a}$ *where* a *is the axiom of* Σ*. A tree* $t \in Tree(\mathbf{Z}(\Sigma))_s$ *is a representation of a subtree of type* s *and a tree* $c \in Tree(\mathbf{Z}(\Sigma))_{\hat{s}}$ *is a representation of a context of type* s*.*

Notation 1 *We extend* $(\hat{\cdot})$ *into an involution on* $S \cup \hat{S}$ *and we let* $s_\varepsilon = \hat{s}$ *so that* $\omega_\lambda : s_\varepsilon \cdots s_1 \cdots s_n \to \hat{s_\lambda}$ *for* $\lambda \in \{\varepsilon\} \cup \{1, \dots, n\}$*.*

However most of the trees build from this signature $\mathbf{Z}(\Sigma)$ 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 mono-sorted 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 $\mathcal{Z}_{ipper}(\Sigma)$ 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 a,
                      suc ::DStream a}
data CxtDStream a = Init (DStream a)
```

```
                  | CoNode{val':: a ,
                          prev'::CxtDStream a,
                          suc'::DStream a}
```

This is the inductive data structure associated with signature $\mathbf{Z}(\Sigma)$. 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 a,
                    suc ::DList a}
             | Nil (CxtDList a)
data CxtDList a = Init (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 a)
```

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 [28, 27, 18].

3.3 Unfolding of a tree

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 8. *The attribute grammar η_Σ associated with signature $\Sigma = (S, \Omega)$ and axiom $a \in S$ is defined as follows. There is one inherited attribute cxt_s, and one synthesized attribute $tree_s$ associated with each sort $s \in S$; i.e. $Inh = \{cxt_s | s \in \mathcal{S}\}$ and $Syn = \{tree_s | s \in \mathcal{S}\}$. Attribute cxt_s represents the context at the given node of the tree, and $tree_s$ the subtree rooted at that node. These attribute have types $tree_s : s \to s$ and $cxt_s : s \to \hat{s}$. The semantic domains are given by $\mathcal{D}_{tree_s} = Tree(\mathbf{Z}(\Sigma))_s$ and $\mathcal{D}_{cxt_s} = Tree(\mathbf{Z}(\Sigma))_{\hat{s}}$. Exit attribute is $q_0 = cxt_a$. The semantic rules associated with operator $\omega : s_1 \dots s_n \to s$ are given by*

$$\begin{array}{l} \omega : X_1 \dots X_n \to X_\varepsilon \\ \begin{cases} X_\varepsilon \cdot tree_s = \omega_\varepsilon \left(X \cdot cxt_s, X_1 \cdot tree_{s_1}, \dots, X_n \cdot tree_{s_n} \right) \\ X_i \cdot cxt_{s_i} \ = \omega_i \left(X \cdot cxt_s, X_1 \cdot tree_{s_1}, \dots, X_n \cdot tree_{s_n} \right) \end{cases} \end{array}$$

Finally, $init : Tree(\mathbf{Z}(\Sigma))_a \to Tree(\mathbf{Z}(\Sigma))_{\hat{a}}$ is the constructor Init.

Notice that the projection associated with the exit attribute $result : Tree(\mathbf{Z}(\Sigma))_a \to Tree(\mathbf{Z}(\Sigma))_a$ is the identity function. We let $\mathcal{U}_\Sigma$, for "unfolding", denote the algebra associated with attribute grammar η_Σ. By Def. 4 the interpretation of operator ω is thus given for $f_i : Tree(\mathbf{Z}(\Sigma))_{\hat{s}_i} \to Tree(\mathbf{Z}(\Sigma))_{s_i}$, and $cxt \in Tree(\mathbf{Z}(\Sigma))_{\hat{s}}$ by

$$\begin{array}{l} \omega^{\mathcal{U}_\Sigma}(f_1, \dots, f_n)\ cxt \ = \ \omega_\varepsilon \left(cxt, tree_1, \dots, tree_n \right) \\ \quad \text{where} \quad tree_i \ = \ f_i \left(\omega_i \left(cxt, tree_1, \dots, tree_n \right) \right) \end{array}$$

and we let $unfold_\Sigma = [\![\eta_\Sigma]\!]$ denote the corresponding top level function:

$$\begin{array}{l} unfold_\Sigma : Tree(\Sigma) \to Tree(\mathbf{Z}(\Sigma)) \\ unfold_\Sigma(t) \ = \ ctree \quad \text{where} \quad ctree \ = \ t^{\mathcal{U}_\Sigma}(Init(ctree)) \end{array}$$

By Remark 1 the unfolding function can be written as:

$$unfold_\Sigma(t) \ = \ ctree \qquad \text{where} \quad ctree \ = \ build_a(t)(Init(ctree))$$

where $build_s : Tree(\Sigma)_s \to (Tree(\mathbf{Z}(\Sigma))_{\hat{s}} \to Tree(\mathbf{Z}(\Sigma))_s)$ are given my mutual recursion as:

$$\begin{array}{l} build_s(\omega(t_1, \dots, t_n))(cxt) \ = \ \omega_\varepsilon(cxt, ct_1, \dots, ct_n) \\ \quad \text{where} \quad ct_i \ = \ build_{s_i}(t_i)(\omega_i(cxt, ct_1, \dots, ct_n)) \end{array}$$

3.4 A first order algebra associated with an attribute grammar

Definition 9. *The $\mathbf{Z}(\Sigma)$-algebra $\mathcal{A}^\sharp_\mathbb{G}$ associated with attribute grammar $\mathbb{G}$ is given by*

$$\begin{array}{rl} (\mathcal{A}^\sharp_\mathbb{G})_s = \mathcal{D}^\uparrow_s & (\mathcal{A}^\sharp_\mathbb{G})_{\hat{s}} = \mathcal{D}^\downarrow_s \\ \omega_\varepsilon^{\mathcal{A}^\sharp_\mathbb{G}}(v)(syn) = & sem(\omega)_{\varepsilon,syn}(v) \\ \omega_i^{\mathcal{A}^\sharp_\mathbb{G}}(v)(inh) = & sem(\omega)_{i,inh}(v) \\ Init^{\mathcal{A}^\sharp_\mathbb{G}} = & init \end{array}$$

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

Notice that the attribute grammar may be unambiguously recovered from the associated $\mathbf{Z}(\Sigma)$-algebra. Actually the semantic rules of an attribute grammar on Σ are in one to one correspondance with the functions of interpretation of the operators in $\mathbf{Z}(\Sigma)$ in the corresponding $\mathbf{Z}(\Sigma)$-algebra. So these two notions are equivalent: they are two different ways of representing the same pieces of information. Katsumata presents in [35] a description of attribute grammars as algebras in a categorical setting where the signature remains the same but the underlying category is changed.

Proposition 2. *A tree has the same interpretation in an attribute grammar and in the corresponding* $\mathbf{Z}(\Sigma)$*-algebra. More precisely* $(unfold(t))^{\mathcal{A}_{\mathbb{G}}^{\sharp}} = val$ *where* $val = t^{\mathcal{A}_{\mathbb{G}}}(init(val))$.

Proof. Since continuous functions commute with least fixed points we obtain the interpretation of the tree $unfold(t)$ in the algebra $\mathcal{A}_{\mathbb{G}}^{\sharp}$ by replacing in its definition each occurrence of $Init$, ω_ε and ω_i by their respective interpretations $Init^{\mathcal{A}_{\mathbb{G}}^{\sharp}} = init$, $\omega_\varepsilon^{\mathcal{A}_{\mathbb{G}}^{\sharp}}$ and $\omega_i^{\mathcal{A}_{\mathbb{G}}^{\sharp}}$. Thus

$$(unfold(t))^{\mathcal{A}_{\mathbb{G}}^{\sharp}} = val \qquad \text{where} \quad val = build'_a(t)(init(val))$$

where the auxiliary functions $build'_s : Tree(\Sigma)_s \to (\mathcal{D}_s^{\downarrow} \to \mathcal{D}_s^{\uparrow})$, interpretations of the functions $build_s$ in the algebra $\mathcal{A}_{\mathbb{G}}^{\sharp}$, are given by mutual recursion as:

$$\begin{aligned} build'_s(\omega(t_1, \dots, t_n))(v_\varepsilon) &= \omega_\varepsilon^{\mathcal{A}_{\mathbb{G}}^{\sharp}}(v_\varepsilon, v_1, \dots, v_n) \\ \text{where} \quad v_i &= build'_{s_i}(t_i)(\omega_i^{\mathcal{A}_{\mathbb{G}}^{\sharp}}(v_\varepsilon, v_1, \dots, v_n)) \end{aligned}$$

Since $\omega_\varepsilon^{\mathcal{A}_{\mathbb{G}}^{\sharp}}(v)(syn) = sem(\omega)_{\varepsilon,syn}(v)$ and $\omega_i^{\mathcal{A}_{\mathbb{G}}^{\sharp}}(v)(inh) = sem(\omega)_{i,inh}(v)$ we observe that $build'_s(t)(v)(q) = build_{s,q}^{\mathbb{G}}(t)(v)$, with $v = (v_\varepsilon, v_1, \dots, v_n)$. Now this latter expression was introduced in Remark 1 as a reformulation of $t^{\mathcal{A}_{\mathbb{G}}}(v)(q)$. ■

Therefore (see Fig.3) $result\left((unfold(t))^{\mathcal{A}_{\mathbb{G}}^{\sharp}}\right)$ coincides with the value *return(t)* returned by the attribute grammar $\mathbb{G}$ from the given input tree. I.e.

Corollary 1. $[\![G^{\sharp}]\!] \circ [\![\eta_\Sigma]\!] = [\![G]\!]$

where $G^{\sharp} = \langle (), ()^{\mathcal{A}_{\mathbb{G}}^{\sharp}}, result \rangle$ is the corresponding syntax-directed translation (see Def.5). Notice that $G^{\sharp}$ is a rooted attribute grammar with synthesized attributes only. We thus obtain an algorithm for computing that result by simple structural recursion on the unfolding of the input tree. In our running example, we obtain the Haskell code displayed in Table 3. We notice that the resulting Haskell code is an immediate transcription of the semantic rules of the attribute grammar: If we omit the first section of the code describing the definitions of the cyclic data structures and the unfolding operation, which could be automatically generated from the original data structures, the remaining part is a verbatim of the semantics rules depicted in Fig. 1. This code is thus as readable as the one displayed in Table 1, and maybe even more so because we have a neater separation of

Table 3. first-order implementation of flatten using cyclic representation of zippers

```
data Tree  a  = Leaf a | Fork (Tree a) (Tree a)
data ZTree a  = Leaf_ a (ZCxt a) | Fork_  (ZCxt a) (ZTree a) (ZTree a)
data ZCxt  a  = Init_T (ZTree a) | Fork1 (ZCxt a)  (ZTree a) (ZTree a)
                                 | Fork2 (ZCxt a)  (ZTree a) (ZTree a)

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

flatten :: ZTree a -> [a]
flatten (Leaf_ a cxt) = a:(coflat cxt)
flatten (Fork_ cxt left right) = flatten left

coflat :: ZCxt a -> [a]
coflat (Init_T tree) = []
coflat (Fork1 cxt left right) = flatten right
coflat (Fork2 cxt left right) = coflat cxt

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

the semantic rules from the rest of the code. Moreover, this solution is a simple inductive definition whereas recursion was needed in the previous case. If we compare now this solution to the first-order implementation based on zippers (Table 2) we can see two advantages. First, in this previous implementation the programmer has to explicitely build the shapes of the subtrees and contexts in the right-hand side of the rules. This is an undesirable overhead and a potential source of programming errors. The second advantage that we perceive is even more meaningful. Both first-order implementations can be viewed as rewriting systems with rules of the form $q(\omega(x_1, \ldots, x_n)) \to E$ where $q \in Q$ is a function name symbol, called a *state*, viewed as a unary symbol; $x_1, \ldots, x_n$ are variables and E is an expression in which states and variables in $\{x_1, \ldots, x_n\}$ may occur. We can use an output signature for representing these expressions as trees using operators of that signature together with states viewed as (sorted) unary symbols, and the variables. Then, the set of rules define a tree transformation from the input signature Σ to the output signature Σ'. In the solution based on cyclic zippers a state symbol always directly applies to a variable within any right-hand side expression E. We say, in such a case, that the resulting transformation is a *tree transducer* $\mathbf{T} : \Sigma \to_{tt} \Sigma'$. The resulting evaluation function $[\![\mathbf{T}]\!] : Tree(\Sigma) \to Tree(\Sigma')$ is the continuous extension of a map taking a finite tree (term) to its normal form for the transducer, which is a confluent and terminating rewrite system. We can then easily compose two tree transducers $\mathbf{T}_1 : \Sigma_1 \to_{tt} \Sigma_2$ and $\mathbf{T}_2 : \Sigma_2 \to_{tt} \Sigma_3$ to obtain a new tree transducer $\mathbf{T} = \mathbf{T}_2 \cdot \mathbf{T}_1 : \Sigma_1 \to_{tt} \Sigma_3$ such that $[\![T]\!] = [\![T_2]\!] \circ [\![T_1]\!]$. The combined transducer avoids the construction of the intermediate Σ_2-trees. The solution based on zippers gives rise to tree rewriting systems where the above properies do not hold in general and therefore it cannot be used for the fusion of the associated tree transformations.

4 Descriptional composition of attributed tree transducers

In the previous two sections we have respectively presented an higher-order and then a first order implementations of attribute grammars. This changing of point of view allows to transfer the intricacy of attribute computations from the algorithmic side to the data structure side: we have replaced a complex cyclic computation on a simple inductive data structure by a simple primitive recursive scheme defined on cyclic data structures. In this section we use this transformation to present the descriptional composition of attribute grammars [14, 15] in terms of composition of transducers with the same benefit: we obtain a very simple operation of composition of attribute grammars at the price of a preprocessing phase used to encode attribute grammars as tree transducers.

4.1 The signature of derived operators

In order to use attribute grammars for translating trees over an input signature $\Sigma = (S, \Omega)$ into trees over an output signature $\Sigma' = (S', \Omega')$ we let the domain $\mathcal{D}_q$ associated with an attribute q be the set of trees $Tree(\Sigma')_{s'}$ over the output signature for a corresponding sort $s' \in S'$; attributes can then be viewed as unary operators. Each component of the semantic rules $sem(\omega)$ associated with an operator $\omega \in \Omega$ is then an expression that represents a tree over the output signature with arguments, associated with the respective input attributes, which are also trees over that same signature. This expression can, in general, combine constructors of the output signature $\omega' \in \Omega'$ with predefined functions like `if ... then ... else ...` whose semantics is given by confluent and terminating rewriting systems. We term this transformation *attributed tree transducer* when no predefined functions are used, i.e. semantics functions just consist in plugging the arguments in a given context (to be defined below). We shall restrict to attributed tree transducers, the given constructions can be extended without difficulty to the general case.

Definition 10. *The set of operators of signature $\Sigma^@ = (S, \Omega^@)$ derived from a multi-sorted signature $\Sigma = (S, \Omega)$ is the least set such that*

- *A unary operator $I_s \in \Omega^@(s, s)$ is associated with each sort $s \in S$; it is interpreted as a "hole" of type s*
- *If $\omega \in \Omega(s_1 \dots s_n, s)$ and for all $1 \leq i \leq n$, $t_i \in \Omega^@(u_i, s_i)$, then $\omega(t_1, \dots, t_n) \in \Omega^@(u_1 \dots u_n, s)$.*

*An operator in $\Omega^@$ is called a **derived operator**.*

Definition 11. *The inductive extension $\mathcal{A}^@$ of a Σ-algebra $\mathcal{A}$ is the $\Sigma^@$-algebra with the same carrier sets and such that*

- *$I_s^{\mathcal{A}^@} : \mathcal{A}_s \to \mathcal{A}_s$ is the identity map $id_{\mathcal{A}_s}$*
- *$(\omega(t_1, \dots, t_n))^{\mathcal{A}^@} = \omega^{\mathcal{A}} \circ (t_1^{\mathcal{A}^@} \times \dots \times t_n^{\mathcal{A}^@})$*

In particular the extension of the free interpretation, abbreviated as $(\cdot)^@$, is given by:

- $I_s^@ : Tree(\Sigma)_s \to Tree(\Sigma)_s$ is the identity map $id_{Tree(\Sigma)_s}$

– $(\omega(t_1, \ldots, t_n))^{@} = \omega \circ (t_1^{@} \times \cdots \times t_n^{@})$

In restriction to closed trees $t \in \coprod_s Tree^c(\Sigma^{@})_s$, i.e. trees with no hole, the $\Sigma^{@}$-algebra $(\cdot)^{@}$ can be seen as a (sort preserving) map $(\cdot)^{@} : \coprod_s Tree^c(\Sigma^{@})_s \to \coprod_s Tree(\Sigma)_s$ that provides the expansion of a (closed) tree build up from derived operators into the corresponding tree over the original signature. By induction and continuity it comes that for every closed tree $t \in Tree^c(\Sigma^{@})_s$ and Σ-algebra $\mathcal{A}$: $t^{\mathcal{A}^{@}} = \left(t^{@}\right)^{\mathcal{A}}$. It is a classical result of universal algebra that the extensions of Σ-algebras are precisely those $\Sigma^{@}$-algebras that satisfy the set of equations E given on closed trees by: $t =_E u \quad \Leftrightarrow \quad t^{@} = u^{@}$ i.e. two trees build up from derived operators should have the same interpretation if they represent the same Σ-tree. In particular $(\cdot)^{@}$ is the free $(\Sigma^{@}, E)$-algebra.

Definition 12. *The inductive extension of an attribute grammar* $\mathbb{G} = (\Sigma, Attr, q, \mathcal{D}, sem)$ *is the attribute grammar* $\mathbb{G}^{@} = (\Sigma^{@}, Attr, q, \mathcal{D}, sem^{@})$ *where*

$$sem^{@}(I_s) : \mathcal{D}_s^{\downarrow} \times \mathcal{D}_s^{\uparrow} \to \mathcal{D}_s^{\uparrow} \times \mathcal{D}_s^{\downarrow}$$

is the function that twists its arguments: $sem^{@}(I_s)(v, v') = (v', v)$ *and*

$$\begin{array}{l} sem^{@}(\omega(t_1, \ldots, t_n))(v_0, v'_{1,1}, \ldots, v'_{1,k_1}, \ldots, v'_{n,1}, \ldots, v'_{n,k_n}) = \\ \qquad (v'_0, v_{1,1}, \ldots, v_{1,k_1}, \ldots, v_{n,1}, \ldots, v_{n,k_n}) \\ \qquad \textit{where } (v'_0, v_1, \ldots, v_n) \quad = sem(\omega)(v_0, v'_1, \ldots, v'_n) \\ \qquad\qquad (v'_j, v_{j,1}, \ldots, v_{j,k_j}) = sem^{@}(t_j)(v_j, v'_{j,1}, \ldots, v'_{j,k_j}) \end{array}$$

For instance, comming back to our running example, the semantic rules of the derived operator

$$op = Cons(Tree, Cons(Cons((Leaf\ b), Tree), Leaf\ d))$$

depicted in Fig. 4 are:

$$\begin{array}{l} op :: Tree_1 \times Tree_2 \to Tree \\ \begin{cases} Tree \cdot flatten = Tree_1 \cdot flatten \\ Tree_1 \cdot coflat = b : (Tree_2 \cdot flatten) \\ Tree_2 \cdot coflat = d : (Tree \cdot coflat) \end{cases} \end{array}$$

An attribute grammar is said to be **non circular** when there is non circular dependencies in the above definition of the semantic rules associated with the derived operators an thus no fixed point computation is required for the evaluation of the attributes. Using the respective definitions of the algebra associated with an attribute grammar (Def 4), of the inductive extension of Σ-algebra (Def 11) and of the inductive extension of the semantic rules of an attribute grammar (Def 12) it comes that:

Proposition 3. $\mathcal{A}_{\mathbb{G}^{@}} = (\mathcal{A}_{\mathbb{G}})^{@}$

4.2 Attributed tree transducers

Using the notion of derived operator introduced in the previous section, we can now make the definition of an attributed tree transducer more precise.

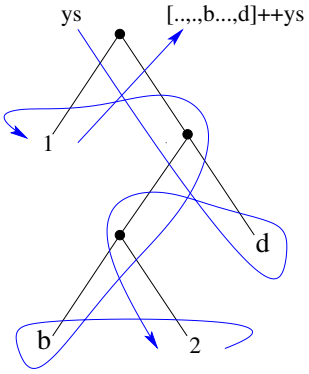

Fig. 4. semantic rules for a derived operator

Definition 13. *An attributed tree transducer $\mathbb{T}$ from input signature $\Sigma = (S, \Omega)$ to output signature $\Sigma' = (S', \Omega')$, in notation $\mathbb{T} : \Sigma \rightarrow_{att} \Sigma'$, is a* ***non circular*** *attribute grammar with underlying signature Σ (see Def. 3) such that each carrier set $\mathcal{D}_q = Tree(\Sigma')_{\sigma(q)}$ is the set of trees on the output signature where $\sigma : Attr \rightarrow S'$ is a map associating a sort to each attribute. We assume that map σ takes the exit attribute to the axiom $a' = \sigma(q_0)$ of the output signature. The semantics rules $sem(\omega)$ attached to an operator $\omega \in \Omega(s_1 \dots s_n, s)$ are given by associating a derived operator $\omega_{\lambda,q} \in \Omega'^{@}(s'_1 \dots s'_k, s')$ [1] and a map $\varphi^{\omega}_{\lambda,q} : \{1, \cdots, k\} \rightarrow In_\omega$ with each output attribute $(\lambda, q) \in Out_\omega$, such that (i) $\sigma(q) = s'$ and (ii) $\varphi^{\omega}_{\lambda,q}(k) = (\lambda', q')$ entails that $q' \in Attr(s_{\lambda'})$ and $\sigma(q') = s'_k$. More precisely for each $(\lambda, q) \in Out_\omega$ $x_\varepsilon \in \mathcal{D}^{\downarrow}_s$, and $x_i \in \mathcal{D}^{\uparrow}_{s_i}$, we let $sem(\omega)_{(\lambda,q)}(x_\varepsilon, x_1, \dots, x_n) = \omega_{\lambda,q}((x_{\lambda_1})_{q_1}, \cdots, (x_{\lambda_k})_{q_k})$ where $\varphi^{\omega}_{\lambda,q}(k) = (\lambda_k, q_k)$. The function init is also described by derived operators. More precisely, we suppose given for each $q \in Inh(a)$ a derived operator $in_q \in \Omega'^{@}(u_q, \sigma(q))$ and a map $\varphi_q : \{1, \dots, n_q\} \rightarrow Syn(a)$ where $n_q = |u_q|$ so that the function init can be expressed as $init(x)_q = in_q(x_{\varphi_q(1)}, \cdots, x_{\varphi_q(n_q)})$ where $x \in \mathcal{D}^{\uparrow}_a = \prod_{q \in Syn(a)} Tree(\Sigma')_{\sigma(q)}$.*

We restrict to non circular grammars in order to ensure that the inductive extension $\mathbb{T}^{@}$ of an attributed tree transducer $\mathbb{T}$ is also an attributed tree transducer: in case of cyclic dependencies the semantics rules associated with derived operators would be represented by infinite trees (on the output signature). We let $[\![\mathbb{T}]\!] : Tree(\Sigma) \rightarrow Tree(\Sigma')$ be the syntax-directed translation associated with attributed tree transducer $\mathbb{T}$ (See Def. 5). [2] The *descriptional composition* is an operation that given two attributed tree transducers $\mathbb{T}_1 : \Sigma_1 \rightarrow_{att} \Sigma_2$ and $\mathbb{T}_2 : \Sigma_2 \rightarrow_{att} \Sigma_3$ produces a new attributed tree transducer $\mathbb{T}_2 © \mathbb{T}_1 : \Sigma_1 \rightarrow_{att} \Sigma_3$ such that $[\![\mathbb{T}_2 © \mathbb{T}_1]\!] = [\![\mathbb{T}_2]\!] \circ [\![\mathbb{T}_1]\!]$. Such an operation was introduced in [14, 14] under some condition (single used requirement) that we shall

[1] of course the sequence $s'_1 \dots s'_k$, and in particular its length k, depends on ω, λ and q. We will write $k = k^{\omega}_{\lambda,q}$

[2] We recall that if Σ is a pointed signature, $Tree(\Sigma)$ stands for the set of Σ-trees the sort of whose root is the axiom.

present below. Our purpose now is to show that the same effect can be obtained by the composition of (ordinary) tree transducers.

4.3 Tree transducers and their composition

A tree transducer is an attributed tree transducer with only synthesized attributes, which are called *states*. We let $Q(s)$ denote the set of states (synthesized attributes) associated with sort s and we let $Q(s, s')$ denote the set $\{q \in Q(s) \mid \sigma(q) = s'\}$ whose elements can be viewed as unary (sorted) operators. Thus Q can be viewed as a $S \times S'$-sorted set. We thus can present tree transducers as it follows.

Definition 14. *A tree transducer* $\mathbf{T}$ *from input signature* $\Sigma = (S, \Omega)$ *to output signature* $\Sigma' = (S', \Omega')$*, in notation* $\mathbf{T} : \Sigma \to_{tt} \Sigma'$*, is a* Σ*-algebra with carrier sets* $\mathbf{T}_s = \prod_{q \in Q(s,s')} Tree(\Sigma')_{s'}$ *where* Q *is a* $S \times S'$*-sorted set whose elements are called states. We let* $Q(s) = \coprod_{s' \in S'} Q(s, s')$ *denote the set of states associated with an element of sort* $s \in S$*. There is a specific state* $q_0 \in Q(a, a')$*, where* a *and* a' *are the respective axioms of* Σ *and* Σ'*, called the* ***initial state*** *of the transducer. The interpretation of an operator* $\omega \in \Omega(s_1 \dots s_n, s)$ *is the map* $\omega^{\mathbf{T}} : \mathbf{T}_{s_1} \times \cdots \times \mathbf{T}_{s_n} \to \mathbf{T}_s$ *given by* $\omega^{\mathbf{T}}(x_1, \dots, x_n)_q = \omega_q\left((x_{i_1})_{q_1}, \dots, (x_{i_m})_{q_m}\right)$ *where* $\omega_q \in \Omega'^{@}(s'_1, \dots, s'_m, s')$ *is a derived operator associated with* $q \in Q(s, s')$ *that comes along with a map* $\varphi^{\omega}_q : \{1, \dots, m\} \to \coprod_{1 \le j \le n} Q(s_j)$ *so that* $\varphi^{\omega}_q(k) = (i_k, q_k)$ *(i.e.* $q_k \in Q(s_{i_k})$*). The map result* $: \mathbf{T}_a \to Tree(\Sigma')_{a'}$ *is the projection associated with the initial state.*

Since there is no inherited attribute the definition of the associated syntax-directed translation $[\![\mathbf{T}]\!] : Tree(\Sigma) \to Tree(\Sigma')$ is no longer circular and is simply given as: $[\![\mathbf{T}]\!](t) = result(t^{\mathbf{T}})$ i.e. we extract the required result directly from the evaluation of the tree in the algebra. As we already noticed, states can be viewed as sorted unary operators, then we can form compound expressions using operators from both input and output signature and states as long as the resulting expression is well-sorted. Then the transducer can be interpreted as a rewrite system with rules $q(\omega(x_1, \dots, x_n)) \to \omega_q\left(q_1(x_{i_1}), \dots, q_m(x_{i_m})\right)$. This rewrite system is confluent and terminating and the normal form of expression $q_0(t)$ for t a finite tree is $[\![\mathbf{T}]\!](t)$. By continuity the value of $[\![\mathbf{T}]\!](t)$ for a possibly infinite tree t is the least upper bound of the normal forms of the expressions $q_0(t')$ for t' a finite approximant of t. In this form we see immediately how to compose two tree transducers $\mathbf{T}_1 : \Sigma_1 \to_{tt} \Sigma_2$ and $\mathbf{T}_2 : \Sigma_2 \to_{tt} \Sigma_3$: we have to specify how to compute the normal form of an expression of the form $q_0^{(2)}(q_0^{(1)}(t))$. The sequential composition of the respective tree transformations consists in computing first the normal form t_1 of expression $q_0^{(1)}(t)$ for $\mathbf{T}_1$ and then the normal form of $q_0^{(2)}(t_1)$ for $\mathbf{T}_2$. The fusion of these transformations, by contrast, use a lazy approach: we never apply a rule for $\mathbf{T}_1$ when some rule for $\mathbf{T}_2$ can be applied. Notice that if t is a term with (sorted) variables in X and q a state such that expression $q(t)$ is well-sorted, then the normal form of this expression is a term built on the output signature with variables of the form $q(x)$: states flow from the root to the leaves during reduction. Thus, there is no applicable reduction for $\mathbf{T}_2$ in some expression exactly when the successor of any state in Q_2, within this expression, is a state in Q_1. This lazy composition is therefore implemented by the rewrite system obtained by replacing each right-hand side of a rule for $\mathbf{T}_1$ by its normal form by $\mathbf{T}_2$. More precisely we proceed as follows. According to Def. 11, we define the inductive extension

$\mathbf{T}^{@} : \Sigma^{@} \to_{tt} \Sigma'$ of a tree transducer $\mathbf{T} : \Sigma \to_{tt} \Sigma'$ such that the right-hand side of the rule associated in $\mathbf{T}^{@}$ with a derived operator and a state is given by the normal form of the corresponding expression using $\mathbf{T}$. Then we can assume, without loss of generality, that each of the derived operators used in the right-hand side of a rule of a tree transducer contains exactly one operator of the output signature, the tree transducer is said to be **uniform**. Indeed, when composing $\mathbf{T}_1 : \Sigma_1 \to_{tt} \Sigma_2$ and $\mathbf{T}_2 : \Sigma_2 \to_{tt} \Sigma_3$ we can replace the intermediate signature Σ_2 by the (finite) subset of its inductive extension $\Sigma_2^{@}$ given by the derived operators appearing in the right-hand side of the rules of $\mathbf{T}_1$ and, accordingly, we replace $\mathbf{T}_2$ by the restriction of its inductive extension $\mathbf{T}_2^{@} : \Sigma_2^{@} \to_{att} \Sigma_3$ to this subset of derived operators. Now the composition of uniform tree transducers is given as follows.

Definition 15. *The composition $\mathbf{T} : \Sigma_1 \to_{tt} \Sigma_3$ of two uniform tree transducers $\mathbf{T}_1 : \Sigma_1 \to_{tt} \Sigma_2$ and $\mathbf{T}_2 : \Sigma_2 \to_{tt} \Sigma_3$ is the (uniform) transducer whose states $\langle q, q' \rangle \in Q(s_1, s_3)$ are pairs of states of the respective transducers: $q \in Q_1(s_1, s_2)$ and $q' \in Q_1(s_2, s_3)$ for some $s_2 \in S_2$. Rule*

$$\langle q, q' \rangle(\omega(x_1, \ldots, x_n)) \to \omega''(\langle q_{j_1}, q'_1 \rangle(x_{i_{(j_1)}}), \ldots, \langle q_{j_{m'}}, q'_{m'} \rangle(x_{i_{(j_{m'})}})$$

is associated with each each pair of so-called composable rules $q(\omega(x_1, \ldots, x_n)) \to \omega'(q_1(x_{i_1}), \ldots, q_m(x_{i_m}))$ and $q'(\omega'(y_1, \ldots, y_m)) \to \omega''(q'_1(y_{j_1}), \ldots, q'_{m'}(y_{j_{m'}}))$ taken from $\mathbf{T}_1$ and $\mathbf{T}_2$ respectively.

We thus obtain an operation of composition of tree transducers such that $[\![\mathbf{T}_2 \cdot \mathbf{T}_1]\!] = [\![\mathbf{T}_2]\!] \circ [\![\mathbf{T}_1]\!]$. See e.g. [7] for a detailed justification.

4.4 Descriptional composition as tree transducers composition

We saw in Section 3.4 how to associate a first-order algebra to an attribute grammar. When the attribute grammar is in fact an attributed tree transducer the resulting algebra is a tree transducer. The definition can be rephrased as it follows in this particular case.

Definition 16. *The set of states of the tree transducer $\mathbb{T}^{\sharp} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ associated with an attributed tree transducer $\mathbb{T} : \Sigma \to_{att} \Sigma'$ is given by $Q(s, s') = \{q \in Syn(s) \mid \sigma(q) = s'\}$ and $Q(\hat{s}, s') = \{q \in Inh(s) \mid \sigma(q) = s'\}$. The initial state of $\mathbb{T}^{\sharp}$ is the exit attribute $q_0 \in Q(a, a')$ of $\mathbb{T}$. For each operator $\omega \in \Omega(s_1 \cdots s_n, s)$ we have $(\omega_\lambda)_q = \omega_{\lambda,q}$ with $\varphi_q^{\omega_\lambda} = \varphi_{\lambda,q}^{\omega}$. Finally, $Init_q = in_q$ and $\varphi_q^{Init} = \varphi_q$ for every $q \in Syn(a)$.*

Remark 2. We let $1_\Sigma : \Sigma \to_\Sigma \Sigma$ denote the tree transducer whose states are associated bijectively with sorts: $Q = \{q_s \mid s \in S\}$ with $\sigma(q_s) = s$, whose initial state is q_a and with rules $q_s(\omega(t_1, \ldots, t_n) \to \omega(q_{s_1}(t_1), \ldots, q_{s_n}(t_n))$ for $\omega \in \Omega(s_1 \ldots s_n, s)$. Then $[\![1_\Sigma]\!]$ is the identity map $id_{Tree(\Sigma)}$ and $\eta_\Sigma^{\sharp} = 1_{\mathbf{Z}(\Sigma)}$.

Corollary 1 states that

Corollary 2. $[\![\mathbb{T}^{\sharp}]\!] \circ [\![\eta_\Sigma]\!] = [\![\mathbb{T}]\!]$

However, the tree transducers $\mathbb{T}_1^\sharp : \mathbf{Z}(\Sigma_1) \to_{tt} \Sigma_2$ and $\mathbb{T}_2^\sharp : \mathbf{Z}(\Sigma_2) \to_{tt} \Sigma_3$ associated respectively with composable attributed tree transducers $\mathbb{T}_1 : \Sigma_1 \to_{att} \Sigma_2$ and $\mathbb{T}_2 : \Sigma_2 \to_{att} \Sigma_3$ cannot be composed because the former produces trees whereas the latter expects zippers. Therefore we should enrich the semantic rules of the original attributed tree transducer so that they define not only the values of the attributes (which are trees over the output signature) but also describe the contexts of these trees within the final result, i.e. the value of the implicit attribute *return*. For that purpose we replace an attributed tree transducer $\mathbb{T} : \Sigma \to_{att} \Sigma'$ by a derived attributed tree transducer $\mathbb{T}^\triangle : \Sigma \to_{att} \mathbf{Z}(\Sigma')$ obtained by splitting each attribute q into two distinct attributes $q^\uparrow$ and $q^\downarrow$ corresponding respectively to the value of the attribute and the value of its context in the final result. Instead, we define an operation taking a tree transducer $\mathbf{T} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ to $\mathbf{T}^{\Uparrow} : \mathbf{Z}(\Sigma) \to_{tt} \mathbf{Z}(\Sigma')$, from which we derive the splitting operation on attributed tree transducers as: $\mathbb{T}^\triangle = \mathbb{T}^{\sharp\Uparrow\sharp}$ using the bijective correspondance $(\cdot)^\sharp : (\Sigma \to_{att} \Sigma') \xrightarrow{\approx} (\mathbf{Z}(\Sigma) \to_{tt} \Sigma')$. For this construction to be possible, we have to ensure that the value of an attribute contributes at a unique occurrence in the final result because if it where not the case we would have different possible contexts for it. For that purpose we impose that each input attribute of an operator is used at most once in its semantic rules.

Definition 17. *A attribute tree transducer $\mathbb{T} : \Sigma \to_{att} \Sigma'$ is said to satisfy the **single used requirement** when for each operator $\omega \in \Omega$ the maps $\varphi^\omega_{\lambda,q}$ for $(\lambda, q) \in Out_\omega$ are jointly monic, i.e. there exists an injective map $\varphi^\omega : \coprod_{(\lambda,q)\in Out_\omega} \left\{1, \ldots, k^\omega_{\lambda,q}\right\} \to In_\omega$ so that $\varphi^\omega_{\lambda,q}$ is the restriction of φ^ω to $\left\{1, \ldots, k^\omega_{\lambda,q}\right\}$. $\varphi^\omega\langle(\lambda, q), j\rangle = (\lambda', q')$ witnesses that the value of attribute (λ', q') is used at position j in the derived operator $\omega_{\lambda,q}$. The same condition must be enforced for the derived operators in_q for $q \in Inh(a)$; i.e. there exists an injective map $\varphi : \coprod_{q\in Inh(a)} \left\{1, \ldots, n_q\right\} \to Syn(a) \setminus \{q_0\}$ whose corresponding restrictions are the maps φ_q for $q \in Inh(a)$. We say that a tree transducer $\mathbf{T} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ satisfies the single used requirement when its associated attributed tree transducer $\mathbb{T} = \mathbf{T}^\sharp$ does.*

Since the value of the exit attribute is used by the environement (it is the end result computed by the attribute tree transducer and returned to the environement) the single used requirement excludes that this value be also used in the initialization of the inherited attributes. With this condition, which was indeed introduced in [14, 15] for the definition of the descriptional composition of attribute grammars, we are in a situation where the splitting operation of the semantic rules is made possible (Definition 18 below). However we first make an observation: in view of Definition 12 and Proposition 3 and using the same reasoning as for tree transducers, we shall assume that the attributed tree transducers are uniform, i.e. that each of the derived operators $\omega_{\lambda,q}$ used in the semantic rules of an attributed tree transducer contains exactly one operator of the output signature. We recall, see Definition 12, that the semantic of a derived operator I_s, a so-called **copy rule**, just propagate the values of inherited attributes downward and the values of synthesized attributes upward.

Definition 18. *The splitting of a uniform tree transducer $\mathbf{T} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ satisfying the single used requirement is the tree transducer $\mathbf{T}^{\Uparrow} : \mathbf{Z}(\Sigma) \to_{tt} \mathbf{Z}(\Sigma')$ where states of the original transducer have been duplicated, i.e. $Q^{\Uparrow} = Q^\uparrow \cup Q^\downarrow$ where*

$Q^{\uparrow} = \left\{q^{\uparrow} \mid q \in Q\right\}$ *and* $Q^{\downarrow} = \left\{q^{\downarrow} \mid q \in Q\right\}$ *are two disjoint copies of* Q*; more precisely* $Q^{\Uparrow}(\mathbf{s}, s') = Q^{\uparrow}(\mathbf{s}, s') \cup Q^{\downarrow}(\hat{\mathbf{s}}, s')$ *for* $\mathbf{s} \in S \cup \hat{S}$ *and* $s' \in S'$*. Rules of* $\mathbf{T}$ *consists of*

$$\begin{aligned} q^{\uparrow}(\omega_{\lambda}(x_{\varepsilon}, x_1, \ldots, x_n)) &\to (\omega_{\lambda,q})_{\varepsilon}(q^{\downarrow}(x_{\lambda}), q_1^{\uparrow}(x_{\lambda_1}), \ldots, q_k^{\uparrow}(x_{\lambda_k})) \\ q_i^{\downarrow}((\omega_{\lambda_i}(x_{\varepsilon}, x_1, \ldots, x_n)) &\to (\omega_{\lambda,q})_i(q^{\downarrow}(x_{\lambda}), q_1^{\uparrow}(x_{\lambda_1}), \ldots, q_k^{\uparrow}(x_{\lambda_k})) \;\; 1 \le i \le k \end{aligned}$$

for each rule of the form $q(\omega_{\lambda}(x_{\varepsilon}, x_1, \ldots, x_n) \to \omega_{\lambda,q}\,(q_1(x_{\lambda_1}), \ldots, q_k(x_{\lambda_k}))$ *in* $\mathbf{T}$*. Similarly for each* $q \in Inh(a)$ *the rule* $q(Init(x)) \to in_q(q_1(x), \ldots, q_{n_q}(x))$ *in* $\mathbf{T}$ *is associated with*

$$\begin{aligned} q^{\uparrow}(Init(x)) &\to (in_q)_{\varepsilon}(q^{\downarrow}(x), q_1^{\uparrow}(x), \ldots, q_{n_q}^{\uparrow}(x)) \\ q_i^{\downarrow}(Init(x)) &\to (in_q)_i(q^{\downarrow}(x), q_1^{\uparrow}(x), \ldots, q_{n_q}^{\uparrow}(x)) \end{aligned}$$

Finally we add rule $q_0^{\downarrow}(Init(x)) \to Init(q_0^{\uparrow}(x))$*.*

A tree transducer is said to be *deterministic* (respectively *complete*) when there exist, for every pair of compatible operator $\omega : s_1 \cdots s_n \to s'$ and state $q : s' \to s''$, at most (resp. at least) one rule whose left-hand side is $q(\omega(x_1, \ldots, x_n))$. The single use requirement states precisely that the above rewriting system is deterministic: there is no multiple definitions of attributes. However this system is not complete in general except if we assume the stronger condition that every input occurrence of attribute of an operator ω occurs at least in one of the right-hand side of the semantic rules attached to (output occurrences of attribute of) this operator. This condition is however far too restrictive and we prefer to relax the requirement that the set of semantic rules is complete: some occurrence of output attribute may be defined by no semantic rule. This is not necessarily a problem: we want to ensure that the tree transformation $[\![\mathbb{T}]\!] : Tree(\Sigma) \to Tree(\Sigma')$ associated with an attributed tree transducer $\mathbb{T} : \Sigma \to_{att} \Sigma'$ is defined for every input tree (it is not a partial function) and that the image of a finite Σ-tree is a finite Σ'-tree. These properties hold initially if we start from non circular attribute tree transducers (with complete set of semantic rules) and they are preserved by descriptional composition (because of the identity $[\![\mathbb{T}_2 © \mathbb{T}_1]\!] = [\![\mathbb{T}_2]\!] \circ [\![\mathbb{T}_1]\!]$ to be established shortly).

Proposition 4. $[\![\eta_{\Sigma'}]\!] \circ [\![\mathbb{T}]\!] = [\![\mathbb{T}^{\triangle}]\!]$ *for* $\mathbb{T} : \Sigma \to_{att} \Sigma'$

Proof. Using the interpretation of a tree w.r.t. an attribute grammar (Section 2.2) we obtain:

$$\begin{aligned} t' = [\![\mathbb{T}]\!](t) = x_{\varepsilon,q_0} \text{ where } x_{\pi\cdot\lambda,q} &= \omega_{\lambda,q}(x_{\pi\cdot\lambda_1,q_1}, \cdots, x_{\pi\cdot\lambda_k,q_k}) \\ x_{\varepsilon,q'} &= in_{q'}(x_{\lambda_1,q_1}, \cdots, x_{\lambda_{n_{q'}},q_{n_{q'}}}) \end{aligned}$$

where $\pi \in Dom(t)$, $\omega = t(\pi)$ and $X_{\lambda} \cdot q = \omega_{\lambda,q}\,(X_{\lambda_1} \cdot q_1, \ldots, X_{\lambda_k} \cdot q_k)$ is the semantic rule associated with $(\lambda, q) \in Out_{\omega}$, and $q' \in Inh(a)$ with semantic rule $init(x)_{q'} = in_{q'}(x_{q_1}, \ldots, x_{q_{n_{q'}}})$. We let $E_{\mathbb{T}}$ denote the set of equations appearing in the corresponding `where` clause. We have similar systems of equations associated with attributed tree transducers $\eta_{\Sigma'}$ and $\mathbb{T}^{\triangle} = \mathbb{T}^{\sharp\Uparrow\sharp}$. In particular

$$t'' = [\![\eta_{\Sigma'}]\!](t') = x'_{\varepsilon,\uparrow} \quad \text{where} \quad E_{\eta_{\Sigma'}}$$

where $E_{\eta_{\Sigma'}}$ contains the equations

$$\begin{aligned} x'_{\pi,\uparrow} &= \omega'_{\varepsilon}(x'_{\pi,\downarrow}, x'_{\pi\cdot 1,\uparrow}, \ldots, x'_{\pi\cdot n,\uparrow}) \\ x'_{\pi\cdot i,\downarrow} &= \omega'_{i}(x'_{\pi,\downarrow}, x'_{\pi\cdot 1,\uparrow}, \ldots, x'_{\pi\cdot n,\uparrow}) \end{aligned}$$

for $\pi \in Dom(t')$ and $t'(\pi) = \omega' \in \Omega'(s'_1, \ldots s'_n, s')$ together with rule $x'_{\varepsilon,\downarrow} = Init(x'_{\varepsilon,\uparrow})$ where $\uparrow$ (respectively $\downarrow$) represents the synthesized attribute $tree_s$ (resp. the inherited attribute cxt_s). If we let variable $x_{\pi,q\uparrow}$ stands for the occurrence of attribute $\uparrow$ for the value of attribute q at node of t at address π we obtain precisely

$$t'' = [\![(\eta_{\Sigma'})]\!]([\![(\mathbb{T})]\!](t)) = x_{\varepsilon,q_0\uparrow} \quad \text{where} \quad E_{\mathbb{T}^\triangle}$$

■

The tree transducer $\mathbf{Z}(\mathbb{T}) : \mathbf{Z}(\Sigma) \to_{tt} \mathbf{Z}(\Sigma')$ associated with attributed tree transducer $\mathbb{T} : \Sigma \to_{att} \Sigma'$ is then given by:

Definition 19. $\mathbf{Z}(\mathbb{T}) = \mathbb{T}^{\sharp\Uparrow} = \mathbb{T}^{\triangle\sharp}$

We let ι denote the embedding where $\mathbb{T} = \iota(\mathbf{T})$ is tree transducer $\mathbf{T} : \Sigma \to_{tt} \Sigma'$ viewed as a attribute tree transducer with synthesized attributes only (the states of $\mathbf{T}$). Letting $\iota_\Sigma = \iota(1_\Sigma) : \Sigma \to_{att} \Sigma$, we notice

Remark 3 (See Remark 2). $\iota_\Sigma^\triangle = \eta_\Sigma$ and thus $\mathbf{Z}(1_\Sigma) = 1_{\mathbf{Z}(\Sigma)}$

By Corollary 2 we deduce

Corollary 3. $[\![(\eta_{\Sigma'})]\!] \circ [\![(\mathbb{T})]\!] = [\![(\mathbf{Z}(\mathbb{T}))]\!] \circ [\![(\eta_\Sigma)]\!]$

Thus transducer $\mathbf{Z}(\mathbb{T})$ realizes, up to unfoldings, the same tree transformation as the original attributed tree transducer $\mathbb{T}$; moreover $\mathbf{Z}(\mathbb{T}_1)$ and $\mathbf{Z}(\mathbb{T}_2)$ can be composed as soon as $\mathbb{T}_1$ and $\mathbb{T}_2$ are composable, and $[\![(\mathbf{Z}(\mathbb{T}_2) \cdot \mathbf{Z}(\mathbb{T}_1))]\!] \circ [\![(\eta_{\Sigma_1})]\!] = [\![(\mathbf{Z}(\mathbb{T}_2))]\!] \circ [\![(\mathbf{Z}(\mathbb{T}_1))]\!] \circ [\![(\eta_{\Sigma_1})]\!] = [\![(\eta_{\Sigma_3})]\!] \circ [\![(\mathbb{T}_2)]\!] \circ [\![(\mathbb{T}_1)]\!]$. Therefore descriptional composition of attributed tree transducers can be obtained (on zippers representations) as ordinary tree transducers compositions. If we want to collect the final result as a tree for the underlying signature we need to erase all contextual information which can be obtained using the following tree transducer:

Remark 4. The tree transformation induced by tree transducer $\varepsilon_\Sigma = \iota_\Sigma^\sharp : \mathbf{Z}(\Sigma) \to \Sigma$ is by Corollary 2 a right-inverse to the unfolding: $[\![(\varepsilon_\Sigma)]\!] \circ [\![(\eta_\Sigma)]\!] = [\![(\iota_\Sigma^\sharp)]\!] \circ [\![(\eta_\Sigma)]\!] = [\![(\iota_\Sigma)]\!] = id_{Tree(\Sigma)}$. The rules for ε_Σ are of the form $q_s(\omega_\varepsilon(x_\varepsilon, x_1, \cdots x_n)) \to \omega(q_{s_1}(x_1), \ldots, q_{s_n}(x_n))$. Thus this transformation just consists in forgetting all information about the contexts.

Corollary 4. $[\![(\mathbb{T})]\!] = [\![(\varepsilon_{\Sigma'})]\!] \circ [\![(\mathbf{Z}(\mathbb{T}))]\!] \circ [\![(\eta_\Sigma)]\!]$

Proposition 5. *Operator $(\cdot)^\Uparrow$ that lifts a tree transducer $\mathbf{T} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ into the tree transducer $\mathbf{T}^\Uparrow : \mathbf{Z}(\Sigma) \to_{tt} \mathbf{Z}(\Sigma')$ satisfies the following properties:*

1. $\varepsilon_\Sigma^\Uparrow = 1_{\mathbf{Z}(\Sigma)}$
2. $\varepsilon_{\Sigma'} \cdot \mathbf{T}^\Uparrow = \mathbf{T}$
3. $\left(\mathbf{T}_2 \cdot \mathbf{T}_1^\Uparrow\right)^\Uparrow = \mathbf{T}_2^\Uparrow \cdot \mathbf{T}_1^\Uparrow$

Proof. The rules for ε_Σ are $q_s(\omega_\varepsilon(x_\varepsilon, x_1, \ldots, x_n)) \to \omega(q_{s_1}(x_1), \ldots, q_{s_n}(x_n))$ hence the derived rules for $\varepsilon_\Sigma^\Uparrow$ are given by:

$$q_s^\uparrow(\omega_\varepsilon(x_\varepsilon, x_1, \ldots, x_n)) \to \omega_\varepsilon(q_s^\downarrow(x_\varepsilon), q_{s_1}^\uparrow(x_1), \ldots, q_{s_n}^\uparrow(x_n))$$
$$q_{s_i}^\downarrow(\omega_i(x_\varepsilon, x_1, \ldots, x_n)) \to \omega_i(q_s^\downarrow(x_\varepsilon), q_{s_1}^\uparrow(x_1), \ldots, q_{s_n}^\uparrow(x_n))$$

whence $\varepsilon_{\Sigma}^{\Uparrow} = 1_{\mathbf{Z}(\Sigma)}$. The only composable rules from $\mathbf{T}^{\Uparrow}$ and $\varepsilon_{\Sigma'}$ respectively are the pairs consisting of

$$q^{\uparrow}(\omega_{\lambda}(x_{\varepsilon}, x_1, \ldots, x_n)) \to (\omega_{\lambda,q})_{\varepsilon}\left(q^{\downarrow}(x_{\lambda}), q_1^{\uparrow}(x_{\lambda_1}), \ldots, q_k^{\uparrow}(x_{\lambda_k})\right)$$

and $q_s((\omega_{\lambda,q})_{\varepsilon}(x_{\varepsilon}, x_1, \ldots, x_k)) \to \omega_{\lambda,q}(q_{s_1}(x_1), \ldots, q_{s_k}(x_k))$ with the appropriate sorts s, $s_1, \ldots, s_k$. The corresponding composite rule is

$$\langle q^{\uparrow}, q_s\rangle(\omega_{\lambda}(x_{\varepsilon}, x_1, \ldots, x_n)) \to \omega_{\lambda,q}(\langle q_1^{\uparrow}, q_{s_1}\rangle(x_{\lambda_1}), \ldots, \langle q_k^{\uparrow}, q_{s_k}\rangle(x_{\lambda_k}))$$

We thus obtain a tree transducer isomorphic to $\mathbf{T}$. In order to prove $\left(\mathbf{T}_2 \cdot \mathbf{T}_1^{\Uparrow}\right)^{\Uparrow} = \mathbf{T}_2^{\Uparrow} \cdot \mathbf{T}_1^{\Uparrow}$, it is enough to establish that the image of $(\cdot)^{\Uparrow}$ is closed by tree transducer composition. Indeed if for composable $\mathbf{T}_1^{\Uparrow}$ and $\mathbf{T}_2^{\Uparrow}$ we know there exists some tree transducer $\mathbf{T}$ such that $\mathbf{T}_2^{\Uparrow} \cdot \mathbf{T}_1^{\Uparrow} = \mathbf{T}^{\Uparrow}$ then we necessarily have $\mathbf{T} = \varepsilon_{\Sigma_3} \cdot \mathbf{T}^{\Uparrow} = \varepsilon_{\Sigma_3} \cdot \left(\mathbf{T}_2^{\Uparrow} \cdot \mathbf{T}_1^{\Uparrow}\right) = \left(\varepsilon_{\Sigma_3} \cdot \mathbf{T}_2^{\Uparrow}\right) \cdot \mathbf{T}_1^{\Uparrow} = \mathbf{T}_2 \cdot \mathbf{T}_1^{\Uparrow}$. A tree transducer is in the image of $(\cdot)^{\Uparrow}$ if we have an involution $\overline{(.)} : Q \to Q$ on its set of states such that $q \in Q(\mathbf{s}, s)$ for $\mathbf{s} \in S \cup \hat{S}$ and $s \in S'$ implies $\overline{q} \in Q(\hat{\mathbf{s}}, \hat{s})$ and the following conditions hold.

1. For $\omega \in \Omega(s_1 \ldots s_n, s)$ and $q \in Q(\hat{s_{\lambda}}, s')$ (we recall from Notation 1 that we set $s_{\varepsilon} = \hat{s}$) the rule with left-hand side $q(\omega_{\lambda}(x_{\varepsilon}, x_1, \ldots, x_n))$ has a right-hand side of the form $\omega'_{\varepsilon}(\overline{q}(x_{\lambda}), q_1(x_{\lambda_1}), \ldots, q_m(x_{\lambda_m}))$. The following related rules should also appear:
$$\overline{q_i}(\omega_{\lambda_i}(x_{\varepsilon}, x_1, \ldots, x_n))) \to \omega'_i(\overline{q}(x_{\lambda}), q_1(x_{\lambda_1}), \ldots, q_m(x_{\lambda_m}))$$
2. Similarly, the rules associated with constructor *Init* and a state $q \in Q(\hat{a}, s)$ are of the form
$$\begin{aligned} q(\mathit{Init}(x)) &\to \omega'_{\varepsilon}(\overline{q}(x), q_1(x), \ldots, q_k(x)) \\ \overline{q_i}(\mathit{Init}(x)) &\to \omega'_i(\overline{q}(x), q_1(x), \ldots, q_k(x)) \end{aligned}$$
3. For q_0 the initial state we have rule $\overline{q_0}(\mathit{Init}(x)) \to \mathit{Init}(q_0(x))$.

The set of states Q for the composite transducer $\mathbf{T}_2^{\Uparrow} \cdot \mathbf{T}_1^{\Uparrow}$ is such that set $Q(\mathbf{s}_1, \mathbf{s}_3)$ is made of the pairs $\langle q_1, q_2\rangle$ such that $q_1 \in Q_1(\mathbf{s}_1, \mathbf{s}_2)$ and $q_2 \in Q_2(\mathbf{s}_2, \mathbf{s}_3)$ for some $\mathbf{s}_2 \in S_2 \cup \hat{S_2}$. We let the involution on Q be defined componentwise by the respective involutions on Q_1 and Q_2. The initial state is the pair made of the initial states of $\mathbf{T}_1^{\Uparrow}$ and $\mathbf{T}_2^{\Uparrow}$ respectively. The last condition is then satisfied for the composite transducer. Since condition (2) has the same form as condition (1) we only have to check the latter. There are two cases to be considered according whether the intermediate sort $\mathbf{s}_2$ is in S_2 or in $\hat{S_2}$. In the first case we consider the composition of a rule of the form $q(\omega_{\lambda}(x_{\varepsilon}, x_1, \ldots, x_m)) \to \omega'_{\varepsilon}(\overline{q}(x_{\lambda}), q_1(x_{\lambda_1}), \ldots, q_m(x_{\lambda_m}))$ with a rule $q'(\omega'_{\varepsilon}(y_{\varepsilon}, y_1, \ldots, y_{m'})) \to \omega''_{\varepsilon}(\overline{q'}(y_{\varepsilon}), q'_1(y_{\lambda'_1}), \ldots, q_{m'}(y_{\lambda'_{m'}}))$. The composite rule is $\langle q, q'\rangle(\omega_{\lambda}(x_{\varepsilon}, x_1, \ldots, x_m)) \to \omega''_{\varepsilon}(\langle \overline{q}, \overline{q'}\rangle(x_{\lambda}), \langle q_{\lambda'_1}, q'_1\rangle(x_{\lambda_{\lambda'_1}}), \ldots, \langle q_{\lambda'_{m'}}, q'_{m'}\rangle(x_{\lambda_{\lambda'_{m'}}}))$ whose first argument has the required shape. Now we have the associated rules

$$\begin{aligned} \overline{q_{\lambda'_i}}(\omega_{\lambda_{\lambda'_i}}(x_{\varepsilon}, x_1, \ldots, x_m)) &\to \omega'_{\lambda'_i}(\overline{q}(x_{\lambda}), q_1(x_{\lambda_1}), \ldots, q_m(x_{\lambda_m})) \\ \overline{q'_i}(\omega'_{\lambda'_i}(y_{\varepsilon}, y_1, \ldots, y_m)) &\to \omega''_i(\overline{q'}(y_{\varepsilon}), q'_1(y_{\lambda'_1}), \ldots, q_{m'}(y_{\lambda'_{m'}})) \end{aligned}$$

whose composition

$$\langle \overline{q_{\lambda'_i}}, \overline{q'_i}\rangle(\omega_{\lambda_{\lambda'_i}}(x_{\varepsilon}, x_1, \ldots, x_m)) \to \omega''_i(\langle \overline{q}, \overline{q'}\rangle(x_{\lambda}), \langle q_{\lambda'_1}, q'_1\rangle(x_{\lambda_{\lambda'_1}}), \ldots, \langle q_{\lambda'_{m'}}, q'_{m'}\rangle(x_{\lambda_{\lambda'_{m'}}}))$$

has also the required form. In the second case we consider a rule

$$\overline{q_i}(\omega_{\lambda_i}(x_\varepsilon, x_1, \ldots, x_m)) \to \omega'_i(\overline{q}(x_\lambda), q_1(x_{\lambda_1}), \ldots, q_m(x_{\lambda_m}))$$

associated with $q(\omega_\lambda(x_\varepsilon, x_1, \ldots, x_m)) \to \omega'_\varepsilon(\overline{q}(x_\lambda), q_1(x_{\lambda_1}), \ldots, q_m(x_{\lambda_m}))$ and we compose it with $q'(\omega'_i(y_\varepsilon, y_1, \ldots, y_m)) \to \omega''_\varepsilon(\overline{q'}(y_i), q'_1(y_{\lambda'_1}), \ldots, q_{m'}(y_{\lambda'_{m'}}))$ The resulting composite rule is

$$\langle \overline{q_i}, q' \rangle(\omega_{\lambda_i}(x_\varepsilon, x_1, \ldots, x_m)) \to \omega''_\varepsilon(\langle q_i, \overline{q'} \rangle(x_{\lambda_i}), \langle q_{\lambda'_1}, q'_1 \rangle(x_{\lambda_{\lambda'_1}}), \ldots, \langle q_{\lambda'_{m'}}, q'_{m'} \rangle(x_{\lambda_{\lambda'_{m'}}}))$$

Again the first argument has the required shape. And by composing rules

$$\begin{aligned} \overline{q_{\lambda'_j}}(\omega_{\lambda_i}(x_\varepsilon, x_1, \ldots, x_m)) &\to \omega'_{\lambda'_j}(\overline{q}(x_\lambda), q_1(x_{\lambda_1}), \ldots, q_m(x_{\lambda_m})) \\ \overline{q'_j}(\omega'_{\lambda'_j}(y_\varepsilon, y_1, \ldots, y_m)) &\to \omega''_j(\overline{q'}(y_i), q'_1(y_{\lambda'_1}), \ldots, q_{m'}(y_{\lambda'_{m'}})) \end{aligned}$$

we obtain

$$\langle \overline{q_{\lambda'_j}}, \overline{q'_j} \rangle(\omega_{\lambda_i}(x_\varepsilon, x_1, \ldots, x_m)) \to \omega''_j(\langle q_i, \overline{q'} \rangle(x_{\lambda_i}), \langle q_{\lambda'_1}, q'_1 \rangle(x_{\lambda_{\lambda'_1}}), \ldots, \langle q_{\lambda'_{m'}}, q'_{m'} \rangle(x_{\lambda_{\lambda'_{m'}}}))$$

as expected. ■

Corollary 5. $\varepsilon_{\mathbf{Z}(\Sigma)} \cdot \mathbf{Z}(\eta_\Sigma) = 1_{\mathbf{Z}(\Sigma)}$

Proof. Prop.5.(2) states $\varepsilon_{\Sigma'} \cdot \mathbf{T}^{\Uparrow} = \mathbf{T}$ for $\mathbf{T} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$, then $\varepsilon_{\mathbf{Z}(\Sigma)} \cdot \mathbf{Z}(\eta_\Sigma) = \varepsilon_{\mathbf{Z}(\Sigma)} \cdot \eta_\Sigma^{\sharp\Uparrow} = \varepsilon_{\mathbf{Z}(\Sigma)} \cdot 1_{\mathbf{Z}(\Sigma)}^{\Uparrow} = 1_{\mathbf{Z}(\Sigma)}$. ■ The following proposition states that if an attributed tree transducer $\mathbb{T} : \Sigma \to_{att} \Sigma'$ contains no inherited attributes, i.e. is of the form $\mathbb{T} = \iota(\mathbf{T})$ for some $\mathbf{T} : \Sigma \to_{tt} \Sigma'$, then $\mathbb{T}^\sharp : \mathbf{Z}(Z) \to_{tt} \Sigma'$ should coincide with tree transducer $\mathbf{T}$ when contextual information are erased (by composition with ε_Σ).

Proposition 6. $\mathbf{T} \cdot \varepsilon_\Sigma = \iota(\mathbf{T})^\sharp$ *for* $\mathbf{T} : \Sigma \to_{tt} \Sigma'$

Proof. $q(\omega(x_1, \ldots, x_n)) \to \omega'(q_1(x_{i_1}), \ldots, q_m(x_{i_m}))$, a rule of $\mathbf{T}$, is composable with the rule $q_s(\omega_\varepsilon(x_\varepsilon, x_1, \cdots x_n)) \to \omega(q_{s_1}(x_1), \ldots, q_{s_n}(x_n))$ in ε_Σ (and only with that one) and the composite rule is

$$\langle q_s, q \rangle(\omega_\varepsilon(x_\varepsilon, x_1, \cdots x_n)) \to \omega'(\langle q_{s_{i_1}}, q_1 \rangle(x_{i_1}), \ldots, \langle q_{s_{i_m}}, q_m \rangle(x_{i_m}))$$

where the first state component is redundant. We thus obtain a tree transducer with rules

$$q(\omega_\varepsilon(x_\varepsilon, x_1, \cdots x_n)) \to \omega'(q_1(x_{i_1}), \ldots, q_m(x_{i_m}))$$

which is precisely the definition of $\iota(\mathbf{T})^\sharp$ ■

Corollary 6. $\mathbf{T} \cdot \varepsilon_\Sigma = \varepsilon_{\Sigma'} \cdot \mathbf{Z}(\iota(\mathbf{T}))$ *for* $\mathbf{T} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$

Proof. $\mathbf{T} \cdot \varepsilon_\Sigma = \iota(\mathbf{T})^\sharp = \varepsilon_{\Sigma'} \cdot \iota(\mathbf{T})^{\sharp\Uparrow}$ by Prop.5.(2), i.e. $\mathbf{T} \cdot \varepsilon_\Sigma = \varepsilon_{\Sigma'} \cdot \mathbf{Z}(\iota(\mathbf{T}))$. ■

Definition 20. *The descriptional composition of attributed tree transducers* $\mathbb{T}_1 : \Sigma_1 \to_{att} \Sigma_2$ *and* $\mathbb{T}_2 : \Sigma_2 \to_{att} \Sigma_3$ *is the attributed tree transducer* $\mathbb{T}_2 © \mathbb{T}_1 : \Sigma_1 \to_{att} \Sigma_3$ *given by* $(\mathbb{T}_2 © \mathbb{T}_1)^\sharp = \mathbb{T}_2^\sharp \bullet \mathbb{T}_1^\sharp$ *where the binary composition* $\bullet : (\mathbf{Z}(\Sigma_2) \to_{tt} \Sigma_3) \times (\mathbf{Z}(\Sigma_1) \to_{tt} \Sigma_2) \to (\mathbf{Z}(\Sigma_1) \to_{tt} \Sigma_3)$ *is given by* $\mathbf{T}_2 \bullet \mathbf{T}_1 = \mathbf{T}_2 \cdot \mathbf{T}_1^{\Uparrow}$.

Theorem 2. *The descriptional composition satisfies the following identities for* $\mathbb{T} : \Sigma \to_{att} \Sigma'$ *and* $\mathbb{T}_i : \Sigma_i \to_{att} \Sigma_{i+1}$*:*

1. $\mathbb{T} = \iota(\mathbb{T}^\sharp)\copyright\eta_\Sigma$ *and* $\mathbb{T}^\sharp = \varepsilon_{\Sigma'} \cdot \mathbf{Z}(\mathbb{T})$
2. $\eta_{\Sigma'}\copyright\mathbb{T} = \mathbf{Z}(\mathbb{T})\copyright\eta_\Sigma$
3. $[\![\mathbb{T}_2\copyright\mathbb{T}_1]\!] = [\![\mathbb{T}_2]\!] \circ [\![\mathbb{T}_1]\!]$
4. $\mathbf{Z}(\mathbb{T}_2\copyright\mathbb{T}_1) = \mathbf{Z}(\mathbb{T}_2) \cdot \mathbf{Z}(\mathbb{T}_1)$
5. $\mathbb{T}\copyright 1_\Sigma = 1_{\Sigma'}\copyright\mathbb{T} = \mathbb{T}$
6. $\mathbb{T}_3\copyright(\mathbb{T}_2\copyright\mathbb{T}_1) = (\mathbb{T}_3\copyright\mathbb{T}_2)\copyright\mathbb{T}_1$

Proof. By Corollary 5 and Proposition 6 it comes $\left(\iota(\mathbb{T}^\sharp)\copyright\eta_\Sigma\right)^\sharp = \left(\iota(\mathbb{T}^\sharp)\copyright\eta_\Sigma\right)^\sharp = \iota(\mathbb{T}^\sharp)^\sharp \cdot \mathbf{Z}(\eta_\Sigma) = \left(\mathbb{T}^\sharp \cdot \varepsilon_{\mathbf{Z}(\Sigma)}\right)\cdot\mathbf{Z}(\eta_\Sigma) = \mathbb{T}^\sharp\cdot(\varepsilon_{\mathbf{Z}(\Sigma)} \cdot \mathbf{Z}(\eta_\Sigma)) = \mathbb{T}^\sharp\cdot 1_{\mathbf{Z}(\Sigma)} = \mathbb{T}^\sharp$. Proposition 5.(1) states that $\varepsilon_{\Sigma'}\cdot\mathbf{T}^{\Uparrow} = \mathbf{T}$ for $\mathbf{T} : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ and thus $\mathbb{T}^\sharp = \varepsilon_{\Sigma'}\cdot\mathbb{T}^{\sharp\Uparrow} = \varepsilon_{\Sigma'}\cdot\mathbf{Z}(\mathbb{T})$ for $\mathbb{T} : \Sigma \to_{att} \Sigma'$. $(\eta_{\Sigma'}\copyright\mathbb{T})^\sharp = \eta^\sharp_{\Sigma'}\cdot\mathbf{Z}(\mathbb{T}) = \mathbf{Z}(\mathbb{T})$, using identity $\mathbb{T} = \mathbb{T}^\sharp\copyright\eta_\Sigma$ we deduce $\eta_{\Sigma'}\copyright\mathbb{T} = \mathbf{Z}(\mathbb{T})\copyright\eta_\Sigma$. By Corollary 3 and Corollary 2 we deduce $[\![\mathbb{T}_2\copyright\mathbb{T}_1]\!] = [\![\mathbb{T}_2^\sharp\bullet\mathbb{T}_1^\sharp]\!]\circ[\![\eta_{\Sigma_1}]\!] = [\![\mathbb{T}_2^\sharp\cdot\mathbf{Z}(\mathbb{T}_1)]\!]\circ [\![\eta_{\Sigma_1}]\!] = [\![\mathbb{T}_2^\sharp]\!] \circ [\![\mathbf{Z}(\mathbb{T}_1)]\!] \circ [\![\eta_{\Sigma_1}]\!] = [\![\mathbb{T}_2^\sharp]\!] \circ [\![\eta_{\Sigma_2}]\!] \circ [\![\mathbb{T}_1]\!] = [\![\mathbb{T}_2]\!] \circ [\![\mathbb{T}_1]\!]$. By Prop. 5.3 $\mathbf{Z}(\mathbb{T}_2\copyright\mathbb{T}_1) = (\mathbb{T}_2\copyright\mathbb{T}_1)^{\sharp\Uparrow} = \left(\mathbb{T}_2^\sharp \bullet \mathbb{T}_1^\sharp\right)^{\Uparrow} = \mathbb{T}_2^{\sharp\Uparrow}\cdot\mathbb{T}_1^{\sharp\Uparrow} = \mathbf{Z}(\mathbb{T}_2)\cdot\mathbf{Z}(\mathbb{T}_1)$. The unit laws are established as it follows: $(\mathbb{T}\copyright 1_\Sigma)^\sharp = \mathbb{T}^\sharp\cdot\mathbf{Z}(1_\Sigma) = \mathbb{T}^\sharp\cdot 1_{\mathbf{Z}(\Sigma)} = \mathbb{T}^\sharp$ and $(1_{\Sigma'}\copyright\mathbb{T})^\sharp = 1^\sharp_{\Sigma'}\cdot\mathbf{Z}(\mathbb{T}) = \varepsilon_{\Sigma'}\cdot\mathbf{Z}(\mathbb{T}) = \mathbb{T}^\sharp$. The associativity law follows from the associativity of tree transducers composition: $(\mathbb{T}_3\copyright(\mathbb{T}_2\copyright\mathbb{T}_1))^\sharp = \mathbb{T}_3^\sharp \bullet (\mathbb{T}_2\copyright\mathbb{T}_1)^\sharp = \mathbb{T}_3^\sharp\cdot\mathbf{Z}(\mathbb{T}_2\copyright\mathbb{T}_1) = \mathbb{T}_3^\sharp\cdot(\mathbf{Z}(\mathbb{T}_2)\cdot\mathbf{Z}(\mathbb{T}_1)) = \left(\mathbb{T}_3^\sharp\cdot\mathbf{Z}(\mathbb{T}_2)\right)\cdot\mathbf{Z}(\mathbb{T}_1) = (\mathbb{T}_3\copyright\mathbb{T}_2)^\sharp\cdot\mathbf{Z}(\mathbb{T}_1) = ((\mathbb{T}_3\copyright\mathbb{T}_2)\copyright\mathbb{T}_1)^\sharp$. ■

Thus attributed tree transducers with descriptional composition form a category, $\mathbf{Z}(\cdot)$ is a functor from this category to the category of tree transducers, and $\eta : 1 \dot{\to} \mathbf{Z}(\cdot)$ is a natural transformation where $\eta_\Sigma : \Sigma \to_{att} \mathbf{Z}(\Sigma)$ is weakly universal in the sense that for every attributed tree transducer $\mathbb{T} : \Sigma \to_{att} \Sigma'$ there exists a tree transducer $\mathbb{T}^\sharp : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ such that $\mathbb{T}^\sharp\copyright\eta_\Sigma = \mathbb{T}$. However since there are many trees over signature $\mathbf{Z}(\Sigma)$ that are not unfoldings of trees in Σ many other tree transducers $\mathbf{T}$ such that $\iota(\mathbf{T})\copyright\eta_\Sigma = \mathbb{T}$ exist. For instance if we select one particular rule in $\mathbb{T}^\sharp$ of the form

$$q(\omega_\varepsilon(x_\varepsilon, x_1, \ldots, x_n)) \to \omega'(q_1(x_{\lambda_1}), \ldots, q_k(x_{\lambda_k}))$$

for some $\omega \in \Omega(s_1 \ldots s_n, s)$ and $q \in Q(s, s')$, and replace it by the following rules

$$\begin{aligned} &q(\omega_\varepsilon(x_\varepsilon, x_1, \ldots, x_n)) \to \overline{q}(x_\varepsilon)\\ &\overline{q}(\omega'_i(x_\varepsilon, x_1, \ldots, x_m)) \to \underline{q}(x_i)\\ &\underline{q}(\omega_\varepsilon(x_\varepsilon, x_1, \ldots, x_n)) \to \omega'(q_1(x_{\lambda_1}), \ldots, q_k(x_{\lambda_k})) \end{aligned}$$

where $\overline{q}$ and $\underline{q}$ are new states and we have one rule of the second kind for every pair $\langle\omega', i\rangle$ such that $\omega' \in \Omega(s'_1 \ldots s'_m, s')$ with $s'_i = s$. Since any $t = \omega_\varepsilon(\omega'_i(u_\varepsilon, u_1, \ldots, u_m), t_1, \ldots, t_n)$, subtree of some unfolded Σ-tree, satisfies $t = u_i$, we deduce that the new transducer behaves as $\mathbb{T}^\sharp$ on the image of η_Σ (at the price of extra copy rules).

5 An example of descriptional composition: reversing the flattening of a binary tree

We consider our running example of an attribute grammar for flattening binary trees. In order to make an attributed tree transducer $\mathbb{T}_1$ out of it, we introduce an explicit

signature for lists and replace the predefined Haskell list constructors in the semantic rules by the corresponding expressions using these constructors. Thus we consider the signature Σ' having a constant operator Nil for the empty list and a unary operator $C(a)$ for each symbol a for appending this element a at the begining of the list given in argument. The tree transducer $\mathbb{T}_1^\sharp : \mathbf{Z}(\Sigma) \to_{tt} \Sigma'$ has the following rules (see Fig.1) where we shorten names `flatten` and `coflat` into fl and cf respectively.

$$\begin{array}{ll} fl(Leaf(a)_\varepsilon(x_\varepsilon)) & \to C(a)(cf(x_\varepsilon))) \\ fl(Fork_\varepsilon(x_\varepsilon, x_1, x_2)) & \to I(fl(x_1)) \\ cf(Fork_1(x_\varepsilon, x_1, x_2)) & \to I(fl(x_2)) \\ cf(Fork_2(x_\varepsilon, x_1, x_2)) & \to I(cf(x_\varepsilon)) \\ cf(Init(x)) & \to Nil \end{array}$$

The resulting tree transducer $\mathbf{Z}(\mathbb{T}_1) : \mathbf{Z}(\Sigma) \to_{tt} \mathbf{Z}(\Sigma')$ is given by the following set of rules:

$$\begin{array}{ll} fl^\uparrow(Leaf(a)_\varepsilon(x_\varepsilon)) & \to C(a)_\varepsilon(fl^\downarrow(x_\varepsilon), cf^\uparrow(x_\varepsilon)) \\ cf^\downarrow(Leaf(a)_\varepsilon(x_\varepsilon)) & \to C(a)_1(fl^\downarrow(x_\varepsilon), cf^\uparrow(x_\varepsilon)) \\ fl^\uparrow(Fork_\varepsilon(x_\varepsilon, x_1, x_2)) & \to I_\varepsilon(fl^\downarrow(x_\varepsilon), fl^\uparrow(x_1)) \\ fl^\downarrow(Fork_1(x_\varepsilon, x_1, x_2)) & \to I_1(fl^\downarrow(x_\varepsilon), fl^\uparrow(x_1)) \\ cf^\uparrow(Fork_1(x_\varepsilon, x_1, x_2)) & \to I_\varepsilon(cf^\downarrow(x_1), fl^\uparrow(x_2)) \\ fl^\downarrow(Fork_2(x_\varepsilon, x_1, x_2)) & \to I_1(cf^\downarrow(x_1), fl^\uparrow(x_2)) \\ cf^\uparrow(Fork_2(x_\varepsilon, x_1, x_2)) & \to I_\varepsilon(cf^\downarrow(x_2), cf^\uparrow(x_\varepsilon)) \\ cf^\downarrow(Fork_\varepsilon(x_\varepsilon, x_1, x_2)) & \to I_1(cf^\downarrow(x_2), cf^\uparrow(x_\varepsilon)) \\ cf^\uparrow(Init(x)) & \to Nil_\varepsilon(cf^\downarrow(x)) \\ fl^\downarrow(Init(x)) & \to Init(fl^\uparrow(x)) \end{array}$$

Now $(I_s)_\varepsilon : \hat{s} \times s \to s$ and $(I_s)_1 : \hat{s} \times s \to \hat{s}$ are the respective first and second projections. By grouping together the rules associated with a same attribute we obtain the following set of Haskell functions (defined by primitive recursion):

```
data ZList a = Nil_ (CxtL a) | Cons_ a (CxtL a)(ZList a)
data CxtL a  = Init_L (ZList a) | Cons1 a (CxtL a)(ZList a)
data ZTree a = Leaf_ a (CxtT a) | Fork_ (CxtT a)(ZTree a)(ZTree a)
data CxtT a  = Init_T (ZTree a) | Fork1 (CxtT a)(ZTree a)(ZTree a)
                                | Fork2 (CxtT a)(ZTree a)(ZTree a)

flatten :: ZTree a -> ZList a
flatten = fl^

fl^ :: ZTree a -> ZList a
fl^ (Leaf_ a x_) = Cons_ a (fl_ x_) (cf^ x_)
fl^ (Fork_ x_ x1 x2) = fl^ x1

fl_ :: CxtT a -> CxtL a
fl_ (Init_T x) = Init_L (fl^ x)
fl_ (Fork1 x_ x1 x2) = fl_ x_
fl_ (Fork2 x_ x1 x2) = cf_ x1

cf^ :: CxtT a -> ZList a
```

```
cf^ (Init_T x) = Nil_ (cf_ x)
cf^ (Fork1 x_ x1 x2) = fl^ x2
cf^ (Fork2 x_ x1 x2) = cf^ x_

cf_ :: ZTree a -> CxtL a
cf_ (Leaf_ a x_) = Cons1 a (fl_ x_) (cf^ x_)
cf_ (Fork_ x_ x1 x2) = cf_ x2
```

We may compose this flattening operation on trees with a list reversing function in order to get the list of leaves in the reverse order, i.e. from right-to-left. We can invert a list, using an accumulating parameter, using for instance the following Haskell function

```
reverse xs = rev xs []
rev [] ys = ys
rev (a:xs) ys = a:(rev xs ys)
```

This function is the higher-order semantics associated with the following attributed tree transducer

$$
\begin{array}{l}
Nil :: \quad \rightarrow Tree_{\varepsilon} \\
\qquad \left\{ Tree_{\varepsilon} \cdot rev = Tree_{\varepsilon} \cdot cr \right. \\
C(a) :: Tree_1 \rightarrow Tree_{\varepsilon} \\
\qquad \begin{cases} Tree_{\varepsilon} \cdot rev = Tree_1 \cdot rev \\ Tree_1 \cdot cr \ = Cons(a)(Tree_{\varepsilon} \cdot cr) \end{cases} \\
Root :: Tree \rightarrow \\
\qquad \left\{ Tree \cdot cr = Nil \right.
\end{array}
$$

The corresponding tree transducer $\mathbf{T}_2 = \mathbb{T}_2^{\sharp} : \mathbf{Z}(\Sigma') \rightarrow_{tt} \Sigma'$ has the following rules:

$$
\begin{array}{ll}
rev(Nil_{\varepsilon}(x_{\varepsilon})) & \rightarrow I(cr(x_{\varepsilon})) \\
rev(C(a)_{\varepsilon}(x_{\varepsilon}, x_1)) & \rightarrow I(rev(x_1)) \\
cr(C(a)_1(x_{\varepsilon}, x_1)) & \rightarrow C(a)(cr(x_{\varepsilon})) \\
cr(Init(x)) & \rightarrow Nil
\end{array}
$$

from which we deduce the rules for $\mathbf{Z}(\mathbb{T}_2) : \mathbf{Z}(\Sigma') \rightarrow \mathbf{Z}(\Sigma')$:

$$
\begin{array}{ll}
rev^{\uparrow}(Nil_{\varepsilon}(x_{\varepsilon})) & \rightarrow I_{\varepsilon}(rev^{\downarrow}(x_{\varepsilon}), cr^{\uparrow}(x_{\varepsilon})) = cr^{\uparrow}(x_{\varepsilon}) \\
cr^{\downarrow}(Nil_{\varepsilon}(x_{\varepsilon})) & \rightarrow I_1(rev^{\downarrow}(x_{\varepsilon}), cr^{\uparrow}(x_{\varepsilon})) = rev^{\downarrow}(x_{\varepsilon}) \\
rev^{\uparrow}(C(a)_{\varepsilon}(x_{\varepsilon}, x_1)) & \rightarrow I_{\varepsilon}(rev^{\downarrow}(x_{\varepsilon}), rev^{\uparrow}(x_1)) = rev^{\uparrow}(x_1) \\
rev^{\downarrow}(C(a)_1(x_{\varepsilon}, x_1)) & \rightarrow I_1(rev^{\downarrow}(x_{\varepsilon}), rev^{\uparrow}(x_1)) = rev^{\downarrow}(x_{\varepsilon}) \\
cr^{\uparrow}(C(a)_1)(x_{\varepsilon}, x_1)) & \rightarrow C(a)_{\varepsilon}(cr^{\downarrow}(x_1), cr^{\uparrow}(x_{\varepsilon})) \\
cr^{\downarrow}(C(a)_{\varepsilon})(x_{\varepsilon}, x_1)) & \rightarrow C(a)_1(cr^{\downarrow}(x_1), cr^{\uparrow}(x_{\varepsilon})) \\
cr^{\uparrow}(Init(x)) & \rightarrow Nil_{\varepsilon}(cr^{\downarrow}(x)) \\
rev^{\downarrow}(Init(x)) & \rightarrow Init(rev^{\uparrow}(x))
\end{array}
$$

We thus derive the following Haskell functions:

```
rev :: ZList a -> ZList a
rev = rev^
```

```
rev^ :: ZList a -> ZList a
rev^ (Nil_ x_) = cr^ x_
rev^ (Cons_ a x_ x1) = rev^ x1

rev_ :: CxtL a -> CxtL a
rev_ (Init_L x) = Init_L (rev^ x)
rev_ (Cons1 a x_ x1) = rev_  x_

cr^ :: CxtL a -> ZList a
cr^ (Init_L x) = Nil_ (cr_ x)
cr^ (Cons1 a x_ x1) = Cons_ a (cr_ x1)(cr^ x_)

cr_ :: ZLista a -> CxtL a
cr_ (Nil_ x_) = rev_ x_
cr_ (Cons_ a x_ x1) = Cons1 a (cr_  x1)(cr^ x_)
```

Now we can compose transducers $\mathbf{Z}(\mathbb{T}_1)$ and $\mathbf{Z}(\mathbb{T}_2)$ in order to obtain the list of leaves of a binary tree from right-to-left. Since they are tree transducers the rule of the composite transducer are obtained by replacing each right-hand side of rules in $\mathbf{Z}(\mathbb{T}_1)$ by its normal form for $\mathbf{Z}(\mathbb{T}_2)$. For instance the composite rule $\langle fl^{\uparrow}, rev^{\uparrow}\rangle(Leaf(a)_{\varepsilon}(x_{\varepsilon}))) \rightarrow \langle cf^{\uparrow}, rev^{\uparrow}\rangle(x_{\varepsilon})$ follows from the derivation

$$rev^{\uparrow}(fl^{\uparrow}(Leaf(a)_{\varepsilon}(x_{\varepsilon}))) \rightarrow rev^{\uparrow}(C(a)_{\varepsilon}(fl^{\downarrow}(x_{\varepsilon}), cf^{\uparrow}(x_{\varepsilon}))) \rightarrow rev^{\uparrow}(cf^{\uparrow}(x_{\varepsilon}))$$

The states of the composite transducer are the pairs of composable states from the respective transducers. We compute the functions associated with these states as the composition of the corresponding pair of functions. We finally obtain the following implementation for the composite transducer (where the convention for name's formation follows the functional composition rather than the state's name formation: e.g. name `r^f^` reflects composition $rev^{\uparrow} \circ fl^{\uparrow}$ which corresponds to state $\langle fl^{\uparrow}, rev^{\uparrow}\rangle$).

```
r^f^ :: ZTree a -> ZList a
r^f^ (Leaf_ a x_) = r^c^ x_
r^f^ (Fork_ x_ x1 x2) = r^f^ x1

r^c^ :: CxtT a -> ZList a
r^c^ (Init_T x) = c^c_  x
r^c^ (Fork1 x_ x1 x2) = r^f^ x2
r^c^ (Fork2 x_ x1 x2) = r^c^ x_

c^c_ :: ZTree a -> ZList a
c^c_ (Leaf_ a x_) = Cons_ a (c_c^  x_)(c^f_  x_)
c^c_ (Fork_ x_ x1 x2) = c^c_  x2

c_c^ :: CxtT a -> CxtL a
c_c^ (Init_T x) = r_c_  x
c_c^ (Fork1 x_ x1 x2) = c_f^ x2
c_c^ (Fork2 x_ x1 x2) = c_c^ x_

c^f_ :: CxtT a -> ZList a
```

```
c^f_ (Init_T x) = Nil_  (c_f^ x)
c^f_ (Fork1 x_ x1 x2) = c^f_  x_
c^f_ (Fork2 x_ x1 x2) = c^c_  x1

r_c_ :: ZTree a -> CxtL a
r_c_ (Leaf_ a x_) = r_f_  x_
r_c_ (Fork_ x_ x1 x2) = r_c_  x2

c_f^ :: ZTree a -> CxtL a
c_f^ (Leaf_ a x_) = Cons1 a (c_c^  x_)(c^f_  x_)
c_f^ (Fork_ x_ x1 x2) = c_f^ x1

r_f_ :: CxtT a  -> CxtL a
r_f_ (Init_T x) = Init_L (r^f^x)
r_f_ (Fork1 x_ x1 x2) = r_f_  x_
r_f_ (Fork2 x_ x1 x2) = r_c_  x1
```

This transducer $\mathbb{T}$ is the splitting of a tree transducer $\mathbb{T}^{\nabla} : \mathbf{Z}(\Sigma) \rightarrow_{tt} \Sigma'$ whose rules are obtained by forgetting all information about the contexts: $\mathbb{T}^{\nabla} = \varepsilon_{\Sigma'} \cdot \mathbb{T}$. This gives us the following rules:

```
rf :: ZTree a -> List a
rf (Leaf_ a x_) = rc x_
rf (Fork_ x_ x1 x2) = rf x1

rc :: CxtT a -> List a
rc (Init_T x) = cc  x
rc (Fork1 x_ x1 x2) = rf x2
rc (Fork2 x_ x1 x2) = rc x_

cc :: ZTree a -> List a
cc (Leaf_ a x_) = Cons a (cf  x_)
cc (Fork_ x_ x1 x2) = cc  x2

cf :: CxtT a -> List a
cf (Init_T x) = Nil
cf (Fork1 x_ x1 x2) = cf  x_
cf (Fork2 x_ x1 x2) = cc  x1
```

We have $\mathbb{T}^{\nabla\triangle} = \mathbb{T}$ with the following correspondances

```
(rf)^ = r^f^   (rf)_ = r_f_   (rc)^ = r^c^   (rc)_ = r_c_
(cc)^ = c^c_   (cc)_ = c_c^   (cf)^ = c^f_   (cf)_ = c_f^
```

From the definition of $\mathbb{T}^{\nabla}$ we extract the semantic rules of the descriptional composition $\mathbb{T}_2 © \mathbb{T}_1 = \mathbb{T}^{\nabla\sharp}$ (see Fig. 5):

$$
\begin{array}{l}
Leaf(a) :: \quad \rightarrow Tree \\
\quad \left\{ \begin{array}{l} Tree \cdot rf = Tree \cdot rc \\ Tree \cdot cc = a : (Tree \cdot cf) \end{array} \right. \\
Fork :: Tree_1 \times Tree_2 \rightarrow Tree \\
\quad \left\{ \begin{array}{l} Tree_\varepsilon \cdot rf = Tree_1 \cdot rf \\ Tree_\varepsilon \cdot cc = Tree_2 \cdot cc \\ Tree_1 \cdot cf = Tree_\varepsilon \cdot cf \\ Tree_1 \cdot rc = Tree_2 \cdot rf \\ Tree_2 \cdot cf = Tree_1 \cdot cc \\ Tree_2 \cdot rc = Tree_\varepsilon \cdot rc \end{array} \right. \\
Root :: Tree \rightarrow \\
\quad \left\{ \begin{array}{l} Tree \cdot cf = [\,] \\ Tree \cdot rc = Tree \cdot cc \end{array} \right.
\end{array}
$$

Figure 6 (left) shows the flow of computation involved by this attribute grammar for

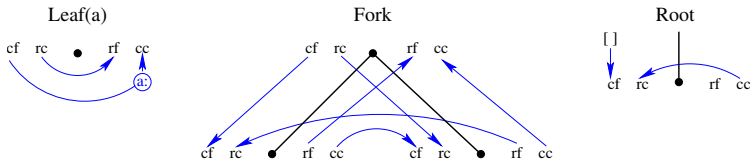

Fig. 5. attribute grammar obtained by descriptional composition

a given binary tree. As illustrated in this example descriptional composition tends to create more copy rules that necessary. However by a static analysis of the semantic rules one can determine the set of pairs of attributes whose value will be the same whatever the input tree: this "largest congruence" for the semantic rules can be computed very efficiently by a fixed-point calculation. We can then form the attribute grammar derived from this congruence. In our example the attributes rc, rf and cc are congruent and, factoring out by this congruence leads to the reduced attribute grammar:

$$
\begin{array}{l}
Leaf(a) :: \quad \rightarrow Tree \\
\quad \left\{ Tree \cdot syn = a : (Tree \cdot syn) \right. \\
Fork :: Tree_1 \times Tree_2 \rightarrow Tree_\varepsilon \\
\quad \left\{ \begin{array}{l} Tree_\varepsilon \cdot syn = Tree_2 \cdot syn \\ Tree_1 \cdot inh = Tree_\varepsilon \cdot inh \\ Tree_2 \cdot inh = Tree_1 \cdot syn \end{array} \right. \\
Root :: Tree \rightarrow \\
\quad \left\{ Tree \cdot inh = [\,] \right.
\end{array}
$$

for which the result is computed by collecting the leaves along a depth-first tranversal of the tree, see Fig. 6 (right).

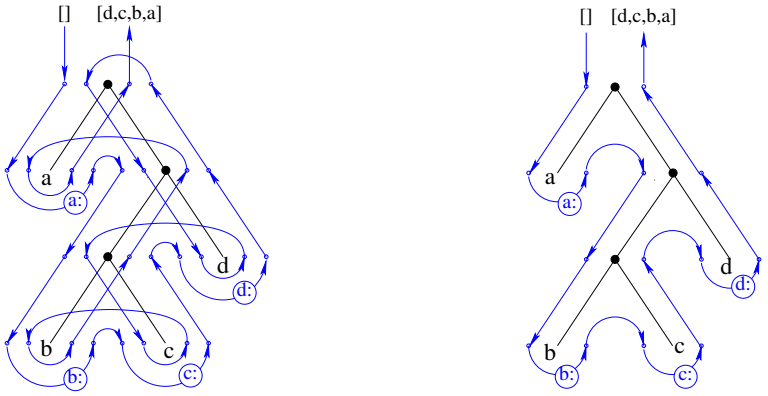

Fig. 6. computing the frontier of the binary tree in reverse order (left) and after reduction of the attribute grammar (right)

6 Conclusion

We have presented a first-order implementation of attribute grammars based on cyclic representations of zippers. We have then used this encoding for revisiting descriptional composition of attribute grammars presented here as tree transducers composition. To put it simply, we have replaced the descriptional composition, a fairly involved operation on attribute grammars, by two more basic operations: the composition of tree transducers and the splitting of attributes (the operation $(\cdot)^{\triangle}$). By doing so we have gained a much better understanding of the "coupling of attributes" that takes place during descriptional composition: it boilds down to the composition of states in tree transducers (a usual composition of functions) up to the symmetry between a subtree and its context.

Besides its pedagogical interest, our construction also has a theoretical value since the correctness of the descriptional composition and the associativity of this operation both follow immediately from our presentation whereas these two results are very tedious to establish in the original formalism: even if they are often mentionned in the litterature we have knowledge of no published direct proofs for these two properties.

This presentation has also a practical interest: since tree transducers composition is a very simple operation we can consider having replaced the *binary* operation of descriptional composition by a *unary* operation of splitting (of attributes). Therefore we no longer have to compute the coupling of attributes for all those attribute grammars which are composable with a given one. Rather, we split the attributes of this attribute grammar in order to put it on a form where it can be composed with any other appropriate attribute grammar by direct application of rewriting rules. This splitting operation is thus a "universal" descriptional composition, and indeed by identity $\mathbb{T}^{\triangle} = \eta_{\Sigma'} \mathbb{C} \mathbb{T}$ it amounts to coupling the attributes of $\mathbb{T}$ with those of the universal attribute grammar $\eta_{\Sigma'}$. Hence we have replaced the computation of all the appropriate couplings by the computation of just one of them and, in fact, the simplest one.

We intend to adapt an editor of attribute grammars that we are currently developping so that the specification of the attribute grammar given by a user using a specific syntax

or a graphical interface be automatically transformed into a transducer for the associated zippers. A library of such attribute grammar specifications can then be build for further reuse. When we form a new specification by reusing components, the corresponding composition is automatically computed and the obtained specification is then reduced by elimination of useless copy rules.

It is also easier to consider extensions of this formalism when attribute grammars are presented as tree transducers. For instance we have mentionned that, in their general form, attributed tree transducers should combine in their semantic rules the operators of the output signature with predefined operators like the conditional. Since the semantics of a tree transducer is defined by a confluent and terminating rewrite system it can be combined without difficulty with any other confluent and terminating rewrite system that specifies the semantics of these additional operators. All the material that we have presented thus extends immediately to that context. Another interesting extension is to relax the single use requirement by observing that it should be possible to make multiple use of an attribute whose sort has no inherited attribute: its value is independent of the context in which it is used. We thus distinguish *semantic sorts*, sorts that should not have inherited attribute, from *syntactic sorts*, which may have such inherited attributes. We then let that $\mathbb{T}$ is an attributed tree transducer from $(\Sigma_1, S_1^{(0)})$ to $(\Sigma_2, S_2^{(0)})$, where $S_i^{(0)} \subseteq S_i$ are subsets of sorts, when a sort in $S_1^{(0)}$ has no inherited attribute and sorts in $S_2^{(0)}$ may not satisfy the single use requirement. Then we have that if $\mathbb{T} : (\Sigma_1, S_1^{(0)}) \rightarrow_{att} (\Sigma_2, S_2^{(0)})$ we also have $\mathbb{T} : (\Sigma_1, S_1^{(0')}) \rightarrow_{att} (\Sigma_2, S_2^{(0')})$ for any $S_1^{(0)'} \subseteq S_1^{(0)}$ and $S_2^{(0)} \subseteq S_2^{(0)'}$, and two transducers $\mathbb{T}_1 : (\Sigma_1, S_1^{(0)}) \rightarrow_{att} (\Sigma_2, S_2^{(0)})$ and $\mathbb{T}_2 : (\Sigma_2, S_2^{(0)}) \rightarrow_{att} (\Sigma_3, S_3^{(0)})$ can be composed to form a new transducer $\mathbb{T} = \mathbb{T}_2 \copyright \mathbb{T}_1 : (\Sigma_1, S_1^{(0)}) \rightarrow_{att} (\Sigma_3, S_3^{(0)})$. The universal attributed tree transducer $\eta_{\Sigma, S^{(0)}} : (\Sigma, S^{(0)}) \rightarrow_{att} \mathbf{Z}(\Sigma, S^{(0)})$ and the related notion of zipper should be adapted so that a sort in $S^{(0)}$ has no context, thus it has only a synthesized attribute in $\eta_{\Sigma, S^{(0)}}$. We have not yet worked out the details but are confident about this extension. Still, there is one complication. The single use requirement ensures that values of attributes need not to be stored since they will be used only once. Since attribute of semantic sorts can be used in several places, we should take care to compute their values only once. For that purpose, it is convenient to replace trees by graphs, or at least by directed acyclic graphs, in order to allow the sharing of subtrees whose root has a semantic sort. A document, now represented by a graph, is well-sorted with respect to $(\Sigma, S^{(0)})$ when its elements have sorts compatible with signature Σ and only those nodes with a semantic sort can be shared.

Acknowledgement

We would like to thank Tarmo Uustalu for his constructive remarks on an Inria research report [5] presenting a preliminary version of the first part of this work. This preliminary work was also presented at the Workshop on *Mathematical Structure in Functional Programming* 2008 [6] where we benefited from detailed comments from the reviewers that help us to improve our presentation.

Bibliography

[1] Comonadic functional attribute evaluation. In *Sixth Symposium on Trends in Functional Programming*, 2005.

[2] Jo ao Paulo Fernandes, Alberto Pardo, and Jo ao Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 95–106, New York, NY, USA, 2007. ACM.

[3] Lex Augusteijn. *Functional Programming, Program Transformations and Compiler Construction*. PhD thesis, Departement of Computing Sciences, Eindhoven University of Technology, The Netherlands, Eindhoven, September 1993.

[4] Kevin Backhouse. A functional semantics of attribute grammars. In *In International Conference on Tools and Algorithms for Construction and Analysis of Systems, Lecture Notes in Computer Science*, pages 303–326. Springer-Verlag, 2002.

[5] Eric Badouel, Bernard Fotsing, and Rodrigue Tchougong. Yet another implementation of attribute evaluation. Research Report 6315, INRIA, 10 2007.

[6] Eric Badouel, Bernard Fotsing, and Rodrigue Tchougong. Attribute grammars as recursion schemes over cyclic representations of zippers. In *Mathematically Structured Functional Programming 2008*, pages 37–54. published in Electronic Notes in Theoretical Computer Science, Iceland, 2008.

[7] Eric Badouel and Marcel Tonga. Growing a domain specific language with split extensions. Research Report 6314, INRIA, 10 2007.

[8] P. Franchi-Zannettaci B.Courcelle. Attribute grammars and recursive program schemes i & ii. *Theoretical Computer Science*, 17:163–257, 1982.

[9] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Inf.*, 21:239–250, 1984.

[10] LoÃ¯c Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative program transformation: A deforestation case-study. In *Principles and Pratice of Declarative Programming*, pages 360–377, 1998.

[11] G. Roussel M. Jourdan E. Duris, D. Parigot. Structured-directed genericity in functional programming and attribute grammars. Research Report 3105, INRIA, 1997.

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

[13] Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.

[14] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN 84' Symp. on Compiler Construction*, pages 157–170, Montreal, june 1984. ACM Press.

[15] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Inf.*, 25(4):355–423, 1988.

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

[17] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[18] Patricia Johann and Neil Ghani. Initial algebra semantics is enough. In *Proceedings, Typed Lambda Calculus and Applications*, pages 207–222. Springer-Verlag, 2007.

[19] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173, 1987.

[20] A. K˙ Benefits of tree transducers for optimizing functional programs.

[21] A. K˙ Tree transducer composition as deforestation method for functional programs. Technical report.

[22] Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.

[23] M. F. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. Technical Report RUU-CS-86-16, Department of Information and Computing Sciences, Utrecht University, 1986.

[24] D. F. Martin L. M. Chirica. An order-algebraic definition of knuthian semantics. *Mathematical System Theory*, 13:1–27, 1979.

[25] Sebastian Maneth. The generating power of total deterministic tree transducers. *Information and Computation*, 147(2):111–144, 1998.

[26] B. Mayoh. Attribute grammars and mathematical semantics. *SIAM Journal of Computing*, 10:503–518, 1981.

[27] T. Uustalu V. Vene N. Ghani, M. Hamana. Representing cyclic structures as nested datatypes. In *Proceedings of 7th Trends in Functional Programming*, pages 173–188, 2006.

[28] M.I. Schwartzbach N. Klarlund. Graph types. In *Symposium on Principles of Programming Languages*, SIGPLAN-SIGACT, pages 196–205. ACM, 1993.

[29] Jukka Paakki. Attribute grammar paradigmsâa high-level methodology in language implementation. *ACM Comput. Surv.*, 27:196–255, June 1995.

[30] Alberto Pardo, Jo ao Paulo Fernandes, and Jo ao Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 81–90, New York, NY, USA, 2009. ACM.

[31] S. Doaitse Swierstra, Joao Saraiva, and Pablo Azero. Designing and implementing combinator languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

[32] Janis Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher-Order and Symbolic Computation*, 17:129–163, 2004.

[33] Janis Voigtländer and Armin Kühnemann. Composition of functions with accumulating parameters. *Journal of Functional Programming*, 14(3):317–363, 2004.

[34] P. Wadler. *Deforestation: Transforming Programs to Eliminate Trees*, pages 344–358. Berlin: Springer-Verlag, 1988.

[35] S. y. Katsumata. Attribute grammars and categorical semantics. In *Proceedings of ICALP 2008*, volume 5126 of *Lecture Notes in Computer Science*, pages 271–282. Springer-Verlag, 2008.