Yet Another Implementation of Attribute Evaluation

Eric Badouel, Bernard Fotsing, Rodrigue Tchougong

Yet Another Implementation of Attribute Evaluation

Éric Badouel — Bernard Fotsing — Rodrigue Tchougong

N° 6315

October 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6315--FR+ENG

Yet Another Implementation of Attribute Evaluation

Éric Badouel[*], Bernard Fotsing[†] , Rodrigue Tchougong[‡]

Thème COM — Systèmes communicants
Projets S4

Rapport de recherche n° 6315 — October 2007 — 24 pages

Abstract: We introduce another item in the already large list of techniques for attribute evaluation. Our algorithm consists in computing an attributes by reduction to normal form using a transducer operating on tree encodings of a cyclic representation of zippers. A zipper is a data structure introduced by Gérard Huet for representing a subtree together with its context, i.e. it is a tree with a focus that points to some node inside it. We mention some potential applications of this representation of attribute grammars as zipper transformers.

Key-words: Attribute Grammars, Attribute Evaluation, Cyclic Data Structures, Zipper

[*] ebadouel@irisa.fr
[†] bfotsing@irisa.fr
[‡] rtchougo@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Une nouvelle implémentation de l’évaluation des attributs

Résumé : Nous introduisons une nouvelle technique pour l’évaluation des attributs d’une grammaire attribuée. Notre algorithme consiste à évaluer un attribut par réduction à une forme normale en utilisant un automate d’arbres opérant sur une représentation arborescente des zippers donnés sous la forme d’une structure de donnée cyclique. Un zipper est une structure de donnée, introduite par Gérard Huet, consistant à représenter un arbre avec son contexte, c’est-à-dire un arbre avec un point focal pointant sur un de ses sous-arbres. Nous mentionnons quelques applications potentielles de cette représentation des grammaires attribuées comme transformations de zippers.

Mots-clés : Grammaires attribuées, Evaluation des attributs, Structures de données cycliques, Zipper

1 Introduction

We introduce another item in the already large list of techniques for attribute evaluation: previous propositions were presented, for instance, in [6, 25, 22, 20, 19, 12, 1, 26]. Our algorithm consists in computing attributes by reductions using a transducer operating on tree representations of zippers. A zipper [16] is a structure representing a subtree together with its context, i.e. it is a tree with a focus that points to some node inside it. More precisely we compute an attribute q in some node u of a tree $t \in T(\Sigma)_s$ as the normal form of the term $q(Z(t,u))$ where $Z(t,u)$ is a tree representation (using an extended signature) of the zipper associated with tree t with a focus on node u. Our approach shares some common features with the previous ones: as in [6, 25] we use an order-algebraic approach based on least fixed-points in order to compute attributes for potentially circular attribute grammars (and on potentially infinite data structures); as in [12] the attribute evaluation is presented as the catamorphism associated with an algebra derived from the semantics rules; and as in [19, 1, 26] we insist on a purely-functional algorithm using lazy evaluation. The presentation by Uustalu and Vene [26] is among these previous works the one that is the most related to our proposition since it also relies on the underlying structure of zipper. An advantage of our presentation is that it leads to a straigthforward encoding into Haskell where the resulting code does not differ much from a litteral description of the semantic rules of the attribute grammar. In the concluding section we mention some potential applications of this representation of attribute grammars as zipper transformers. In this paper we use the notations and definitions for multi-sorted signatures, algebras and transducers introduced in the preliminary sections of the companion paper [2].

2 Attribute grammars

2.1 Decorating structured objects with attributes

A signature can be used to model a family of heteregenous objects while making explicit their structural relationship. The sorts categorize objects where objects sharing some common features are declared of the same sort. The operator $op :: s_1 \times \cdots \times s_n \rightarrow s$ corresponds to an identified manner of combining objects of sorts s_1 to s_n into a compound object of sort s. A term of sort s is an object of that sort whose structural decomposition is made explicit: the term describes completely how the object has been build up starting from the basic objects associated with the constant operators of the signature. Let us consider the example of the assembling of elementary boxes. We have a unique sort *Box* because the only kind of manipulated objects are that of (composite) boxes. We have a constant *elem* :: *Box* representing an elementary box (which we suppose has a unit size: its depth and height is 1) and we have four binary operators $comp_{pos} :: Box \times Box \rightarrow Box$ associated with the four differents manners of assembling two sub-boxes in order to obtained a new box. The parameter $pos \in \{hb, ht, vl, vr\}$ indicates whether we position the two sub-boxes horizontally (with bottom or top alignment), or vertically (with a left or right alignment). A term of sort *Box* completely describe the structure of a composite box. From this representation one can expect to be able to compute various *attributes* of a box like its size, given by its height and depth, or the list of elementary boxes it contains (each one associated with its origin given by its two coordinates). For that purpose we decorate each node of the tree representation of a term with attributes. For each sort we distinguish between its *synthesized* *attributes* and its *inherited attributes*. The values of the synthesized attributes attached to a node u are determined if the values of the other attributes attached to nodes of the subtree rooted at u (including the inherited attributes of u) are known; whereas symmetrically the value of attributes in the context of the subtree rooted at u (i.e. those nodes that does not belong to the subtree rooted at node u) unambiguously determine the values of the inherited attributes at node u. Therefore we can view attributes as providing an interface between the subtree and its context: information flows from the context into the subtree through inherited attributes, and conversely the subtree can send information to its context via its synthesized attributes. The rules governing the computation of the attributes should be syntax directed, i.e. they should be expressed by rules schemas attached to each operator $op :: s_1 \times \cdots \times s_n \rightarrow s$ of the signature. An *input attribute* of operator op is either an inherited attribute of s corresponding to information coming from the context or a synthesized attribute of some s_i corresponding to information synthesized from the corresponding subtree. The synthesized attributes of s and the inherited attributes of the s_i are the *output attributes* of op. The so-called semantic rules allow to compute each output attributes in terms of the input attributes. Back to our example, the size of a box can be computed from the sizes of its subboxes so it is a synthesized attribute. The list of elementary boxes of a compound box is the concatenation of the corresponding lists for its two subcomponents, so it is also a synthesized attribute. But the origin of a box depends only on its context (the boxes that surrounds it), therefore it is an inherited attribute. The semantic rules can be presented as follows.

$$Elem \quad :: \quad \longrightarrow Box_\varepsilon$$

$$\left\{ \begin{array}{lcl} Box_\varepsilon \cdot height & = & 1 \\ Box_\varepsilon \cdot depth & = & 1 \\ Box_\varepsilon \cdot list & = & [(Box_\varepsilon \cdot xCoord, Box_\varepsilon \cdot yCoord)] \end{array} \right.$$

$$hb \quad :: \quad Box_1 \times Box_2 \longrightarrow Box_\varepsilon$$

$$\left\{ \begin{array}{lcl} Box_\varepsilon \cdot height & = & max \quad Box_1 \cdot height \quad Box_2 \cdot height \\ Box_\varepsilon \cdot depth & = & Box_1 \cdot depth \ + \ Box_2 \cdot depth \\ Box_\varepsilon \cdot list & = & Box_1 \cdot list \ +\!\!+ \ Box_2 \cdot list \\ Box_1 \cdot xCoord & = & Box_\varepsilon \cdot xCoord \\ Box_2 \cdot xCoord & = & Box_\varepsilon \cdot xCoord \ + \ Box_1 \cdot depth \\ Box_1 \cdot yCoord & = & Box_\varepsilon \cdot yCoord \ + \ |Box_2 \cdot height - Box_1 \cdot height| \\ Box_2 \cdot yCoord & = & Box_\varepsilon \cdot yCoord \ + \ |Box_1 \cdot height - Box_2 \cdot height| \end{array} \right.$$

Together with the corresponding rules for the related operators ht, vl, and vr. These are actually rule schemas whose purpose is to define the value of each attribute at every node of (the tree representation of) the term. For instance if t is a tree and $u \in Dom(t)$ is such that $t(u)$ then the above equation should be interpreted as

$$\begin{array}{lcl} height(t_u) & = & max\,(height(t_{u \cdot 1}), height(t_{u \cdot 2})) \\ depth(t_u) & = & depth(t_{u \cdot 1}) \ + \ depth(t_{u \cdot 2}) \\ list(t_u) & = & list(t_{u \cdot 1}) \ +\!\!+ \ list(t_{u \cdot 2}) \\ xCoord(t_{u \cdot 1}) & = & xCoord(t_u) \\ xCoord(t_{u \cdot 2}) & = & xCoord(t_u) \ + \ depth(t_{u \cdot 1}) \\ yCoord(t_{u \cdot 1}) & = & yCoord(t_u) \ + \ |height(t_{u \cdot 2}) - height(t_{u \cdot 1})| \\ yCoord(t_{u \cdot 1}) & = & yCoord(t_u) \ + \ |height(t_{u \cdot 1}) - height(t_{u \cdot 2})| \end{array}$$

where t_u stands 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 $t_u(v) = t(u \cdot v)$. If there is no inherited attribute, the computation of attribute is context independant, and the above definition takes the form of a transducer whose states are the synthesized attributes.

Definition 1 *An attribute grammar is a structure $\mathbb{G} = (\Sigma, \Delta, Syn, Inh, R)$ where*

- $\Sigma = (\mathcal{S}, O_p)$ *and* $\Delta = \left(\overline{\mathcal{S}}, \overline{O_p}\right)$ *are signatures respectively called the input and output signatures, the elements of which are called input and output symbols.*

- *Syn and Inh are disjoint sets, disjoint also with Σ and Δ, the elements of which, respectively called synthesized attributes and inherited attributes, are operators of arity in Σ and sort in Δ (hence they are unary operators). We let $Attr = Syn \cup Inh$ denote the set of attributes ; and for each symbol $s \in S$ we let $Inh(s) = \{q \in Inh | \alpha(q) = s\}$, $Syn(s) = \{q \in Syn | \alpha(q) = s\}$, and $Attr(s) = Syn(s) \cup Inh(s)$.*

- *For each operator $op :: s_1 \times \cdots \times s_n \to s$ in O_p we define two Δ-sorted sets of variables whose elements represent occurrences of attributes:*

$$\begin{array}{lcl} V_{op,in} & = & \left\{x_{op,\varepsilon,q} \mid q \in Inh(s)\right\} \bigcup \left\{x_{op,i,q} \mid 1 \le i \le n \quad q \in Syn(s_i)\right\} \\ V_{op,out} & = & \left\{x_{op,\varepsilon,q} \mid q \in Syn(s)\right\} \bigcup \left\{x_{op,i,q} \mid 1 \le i \le n \quad q \in Inh(s_i)\right\} \end{array}$$

where the sort of variable $x_{po,\lambda,q}$ for $\lambda \in \{\varepsilon\} \cup \{1, \ldots, n\}$ is given by $\sigma(x_{\lambda,q}) = \sigma(q)$. We let $V_{op} = V_{op,in} \cup V_{op,out}$ stand for the set of variables associated with operator op.

- *R is a family $\langle R_{op}; op \in O_p\rangle$ of maps $R_{op} : V_{op,out} \to T(\Delta, V_{op,in})$ that preserves sorts: $\sigma\left(R_{op}(x_{\lambda,q})\right) = \sigma(q)$. Hence any output variable is assigned a value given by an expression on the output alphabet which depends on the value of input variables. We can view an element of R as a schema of equations*

$$x_{op,\lambda,q} = rhs_{op,\lambda,q}$$

whose right-hand sides are given by $rhs_{op,\lambda,q} = R_{op}(x_{\lambda,q})$.

We let V_t be the Δ-sorted set of variables associated with a closed tree $t \in Tree(\Sigma)_s$ be given as

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

where variable $x_{t,\pi,q}$ has sort $\sigma(x_{t,\pi,q}) = \sigma(q)$. The semantic equations associate a defining expression for every variable V_t except for the variables $x_{t,\varepsilon,q}$ where $q \in Inh(s)$ (i.e. for the inherited attribute of the root). Let the system of equations $E_{\mathbb{G},v} : V_t \to T(\Delta, V_t)$ associated with attribute grammar $\mathbb{G}$ and vector $v \in \prod_{q \in Inh(s)} T(\Delta, V_t)_{\sigma(q)}$ be defined as:

$$\begin{array}{lcl} E_{\mathbb{G},v}(x_{t,\varepsilon,q}) & = & v(q) \quad \text{for } q \in Inh(s) \\ E_{\mathbb{G},v}(x_{t,\pi,q}) & = & rhs_{op,\varepsilon,q}[x_{t,\pi.\cdot\lambda,q'}/x_{op,\lambda,q'}] \\ E_{\mathbb{G},v}(x_{t,\pi\cdot i,q}) & = & rhs_{op,i,q}[x_{t,\pi.\cdot\lambda,q'}/x_{op,\lambda,q'}] \end{array}$$

Let $\mathcal{A}$ be any continuous Δ-algebra, the interpretation of a tree $t \in Tree(\Sigma)_s$ w.r.t. attribute grammar $\mathbb{G}$ and algebra $\mathcal{A}$ is the map

$$([t])_{\mathbb{G},\mathcal{A}} : \prod_{q \in Inh(s)} \mathcal{A}_{\sigma(q)} \longrightarrow \prod_{q \in Syn(s)} \mathcal{A}_{\sigma(q)}$$

given by $([t])_{\mathbb{G},\mathcal{A}}(v)(q) = \mu E^{\mathcal{A}}_{\mathbb{G}}(x_{t,\varepsilon,q})$ for $v \in \prod_{q\in Inh(s)} \mathcal{A}_{\sigma(q)}$ and $q \in Syn(s)$. Otherwise stated

$$\begin{array}{rcl} ([t])_{\mathbb{G},\mathcal{A}}(v)(q) & = & v_{t,\varepsilon,q} \\ \text{where} \quad v_{t,\pi,q} & = & rhs^{\mathcal{A}}_{op,\varepsilon,q}[v_{t,\pi.\cdot\lambda,q'}/x_{op,\lambda,q'}] \\ v_{t,\pi\cdot i,q} & = & rhs^{\mathcal{A}}_{op,i,q}[v_{t,\pi.\cdot\lambda,q'}/x_{op,\lambda,q'}] \end{array}$$

where it is assumed that the vector $v = \langle v_{t,\lambda,q}\rangle$ 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 "where" clauses in Haskell programs.

2.2 Algebra associated with an attribute grammar

The semantic rules of an attribute grammars are syntax-directed in the sense that they are given by rule schemas associated with each operator of the input signature. For that reason we can exhibit a Σ-algebra $\mathcal{B} = \mathbb{G}\mathcal{A}$ derived from the attribute grammar $\mathbb{G}$ and the continuous Δ-algebra $\mathcal{A}$ such that $([t])_{\mathbb{G},\mathcal{A}} = ([\mathcal{B}])(t)$.

Definition 2 *the Σ-algebra $\mathcal{B} = \mathbb{G}\mathcal{A}$ derived from a continuous Δ-algebra $\mathcal{A}$ and an attribute grammar $\mathbb{G} = (\Sigma, \Delta, Syn, Inh, R)$ is such that*

$$\mathcal{B}_s = \prod_{q\in Inh(s)} \mathcal{A}_{\sigma(q)} \longrightarrow \prod_{q\in Syn(s)} \mathcal{A}_{\sigma(q)}$$

and the interpretation of an operator $op :: s_1 \times \cdots \times s_n \to s$ is the map $op^{\mathcal{B}}$ given by:

$$\begin{array}{rcl} op^{\mathcal{B}}(f_1,\ldots,f_n)(v)(q) & = & v_{op,\varepsilon,q} \\ \textit{where} \quad v_{op,\varepsilon,q} & = & v(q) \quad \textit{if } q \in Inh(s) \\ v_{op,i,q} & = & f_i(v_i)(q) \quad \textit{where } q \in Syn(s_i) \textit{ and } v_i(q') = v_{op,i,q'} \textit{ for } q' \in Inh(s_i) \\ v_{op,\varepsilon,q} & = & rhs^{\mathcal{A}}_{op,\varepsilon,q}[v_{op,\lambda,q}/x_{op,\lambda,q}] \quad \textit{if } q \in Syn(s) \\ v_{op,i,q} & = & rhs^{\mathcal{A}}_{op,i,q}[v_{op,\lambda,q}/x_{op,\lambda,q'}] \quad \textit{if } q \in Inh(s_i) \end{array}$$

That definition is circular [4] since in the "where" clause the inherited attributes $v_i(q') = v_{op,i,q'}$ for $q' \in Inh(s_i)$ appear both in the left-hand side and in the right-hand side of the defining equations. Thus it should be interpreted as the characterization of the vector $\langle v_{op,\lambda,q}\rangle_{x_{op,\lambda,q}\in V_{op}}$ as the least fixed-point of the corresponding transformation.

Proposition 3 $([t])_{\mathbb{G},\mathcal{A}} = ([\mathbb{G}\mathcal{A}])(t)$

Proof. By continuity it is enough to verify this identity on trees $t \in T(\Sigma)_s$. That verification proceeds by induction on the structure of t. If t is of the form $t = op(t_1,\ldots,t_n)$ for some operator $op :: s_1 \times \cdots s_n \to s$, its associated set of variables V_t can be decomposed as

$$V_t = \{x_{t,\varepsilon,q} \mid q \in Att(\sigma(t))\} \cup \left(\bigcup \{x_{t,i\cdot\pi,q} \mid x_{t_i,\pi,q} \in V_{t_i}\}\right)$$

and for $q \in Syn(s)$ and $v \in \prod_{q\in Inh(s)} \mathcal{A}_{\sigma(q)}$ it comes

$$\begin{array}{rcl} ([t])_{\mathbb{G},\mathcal{A}}(v)(q) & = & v_{t,\varepsilon,q} \\ \text{where} \quad v_{t,\varepsilon,q} & = & v(q) \quad \text{if } q \in Inh(s) \\ v_{t,\varepsilon,q} & = & rhs^{\mathcal{A}}_{op,\varepsilon,q}[v_{t,\lambda,q}/x_{op,\lambda,q}] \quad \text{if } q \in Syn(s) \\ v_{t,i,q} & = & rhs^{\mathcal{A}}_{op,i,q}[v_{t,\lambda,q}/x_{op,\lambda,q}] \quad \text{if } q \in Inh(s_i) \\ v_{t,i\cdot\pi,q} & = & v_{t_i,\pi,q} \end{array}$$

By inductive assumption we have

$$v_{t,i,q} \;=\; v_{t\cdot i,\varepsilon,q} \;=\; ([[t_i]])_{\mathbb{G},\mathcal{A}}(v_i)(q) \;=\; t_i^{\mathcal{B}}(v_i)(q)$$

where $\mathcal{B} = \mathbb{G}\mathcal{A}$, $v_i(q') = v_{t,i,q'}$ for $q' \in Inh(s_i)$. By applying Bekič principle for the computation by substitutions of the least fixed-point of a system of equations, it follows that

$$\begin{array}{l} ([t])_{\mathbb{G},\mathcal{A}}(v)(q) \;=\; v_{t,\varepsilon,q} \\ \quad \text{where} \;\; \begin{array}{lcl} v_{t,\varepsilon,q} & = & v(q) \quad \text{if } q \in Inh(s) \\ v_{t,i,q} & = & t_i^{\mathcal{B}}(v_i)(q) \quad \text{if } q \in Syn(s_i) \\ v_{t,\varepsilon,q} & = & rhs^{\mathcal{A}}_{op,\varepsilon,q}[v_{t,\lambda,q}/x_{op,\lambda,q}] \quad \text{if } q \in Syn(s) \\ v_{t,i,q} & = & rhs^{\mathcal{A}}_{op,i,q}[v_{t,\lambda,q}/x_{op,\lambda,q}] \quad \text{if } q \in Inh(s_i) \end{array} \\ = op^{\mathcal{B}}\left(t_1^{\mathcal{B}},\ldots,t_n^{\mathcal{B}}\right)(v)(q) \;=\; t^{\mathcal{B}}(v)(q). \end{array}$$

□

Corollary 4 *If $\varphi : \mathcal{A} \to \mathcal{A}'$ is a morphism of continuous Δ-algebras, $t \in Tree(\Sigma)_s$ a tree of sort s, $v \in \prod_{q\in Inh(s)}$ a vector, and $q \in Syn(s)$*

$$t^{\mathbb{G}\mathcal{A}'}(\varphi v)(q) \;=\; \varphi\left(t^{\mathbb{G}\mathcal{A}}(v)(q)\right)$$

The above semantic of attribute grammars follows the approaches presented in [19, 1], it also draws its inspiration from [25, 6] in the sense that it gives a fixed-point semantics of attribute grammars. We have an almost literal transcription of the above definition into the language Haskell as the mechanism of lazy evaluation escapes the apparent cyclicity of the resulting program as long as the attribute grammar itself is *non-strict* w.r.t. $\mathcal{A}$, where an attribute grammar is said to be non-strict w.r.t. $\mathcal{A}$ if for every tree $t \in Tree(\Sigma)_s$ its interpretation $([\mathbb{G}\mathcal{A}])(t)$ w.r.t. the derived algebra is strict in none of its arguments. We recall that a function $f : A_1 \times \cdots \times A_n \to A$ is strict in its k^{th} argument if $f(x_1,\cdots,x_n) = \bot$ as soon as $x_k = \bot$ whatever the value of the other arguments. It means that in order to get information about the result we need information about its k^{th} argument. Suppose that this information is supplied for each interpretation $op^{\mathcal{A}}$ of operator in Δ w.r.t. algebra $\mathcal{A}$; then we can easily adapt the test of strong non-circularity [9] to derive a polynomial algorithm that checks a sufficient condition for non-strictness. By analogy we could say that an attribute grammar is *strongly non-strict w.r.t. algebra $\mathcal{A}$* when this condition is satisfied.

A translation of attribute grammars into catamorphism (evaluation function for an algebra) was already presented in [12]. However the presentation that we have just given is, in our opinion, more explicit and far more elementary than the one given there and it leads to a straightforward implementation in Haskell. Notice that another advantage of lazy evaluation is that we can define computations of attributes on potentially infinite data structures. For instance we can define semantics rules on streams as long as every approximations of the value of a given attribute can be computed using only a finite prefix of the stream.

Returning to our example of assembling boxes, we first provide an Haskell definition for the data structure of boxes:

```
data Box = Elembox | Comp {pos :: Pos, first, second :: Box}
data Pos = Vert VPos | Hor  HPos
data VPos = Left_ | Right_
data HPos = Top | Bottom
```

Thus a box is either an elementary box (which we suppose has a unit size: its depth and height is 1) or is obtained by composing two sub-boxes. Two boxes can be composed either vertically with a left or right alignment or horizontally with a top or bottom alignment.

The corresponding signature Σ has a unique sort (Box), a constant $elem :: Box$ representing the elementary box and four binary operators
$comp_{pos} :: Box \times Box \rightarrow Box$ associated with the four different manners (with $pos :: Pos$) of assembling two sub-boxes in order to obtained a new box. The related notions of algebras and evaluation morphism can be expressed in Haskell as follows.

```
data AlgBox a = AlgBox {elem  :: a
                       ,comp :: Pos -> a -> a -> a}
evalBox :: AlgBox a  -> Box -> a
evalBox alg Elembox = elem alg
evalBox alg (Comp pos box1 box2) = comp alg pos sembox1 sembox2
     where sembox1 = evalBox alg box1
           sembox2 = evalBox alg box2
```

We consider the Δ algebra $\mathcal{A}$ that interprets max and $+$ as the corresponding operators on integers and $[\,]$ and $++$ as the unit list formation and list concatenation respectively. The attributes are

$$\begin{array}{lll} origin & :: & Box \rightarrow Point \\ list & :: & Box \rightarrow [Point] \\ size & :: & Box \rightarrow Size \end{array}$$

where *origin* is inherited and *list*, and *size* are synthesized. Thus the semantic domain of boxes is given by:

```
data Size = Size {depth_, height_ :: Int} deriving Show
data Point = Point {xcoord, ycoord :: Int} deriving Show
type BoxSem = Point -> (Size,[Point])
```

and the interpretation of the operators are given by

```
elem ::  BoxSem
elem pt = (Size 1 1, [pt])
```

and

```
hb ::  BoxSem -> BoxSem -> BoxSem
hb sem1 sem2 (Point x y) = (Size h d, list)
where h = max h1 h2
      d = d1 + d2
      list = list1 ++ list2
      x1 = x
      x2 = x + d1
      y1 = y + abs (h2 - h1)
      y2 = y + abs (h1 - h2)
      (Size h1 d1, list1) = sem1 (Point x1 y1)
      (Size h2 d2, list2) = sem2 (Point x2 y2)
```

and similarly for the other operators (*ht*, *vl*, and *vr*). We see that the above Haskell code is a direct transcription of the semantic rules of the attribute grammar. Of course the above function can be abreviated as:

```
hb ::  BoxSem -> BoxSem -> BoxSem
hb sem1 sem2 (Point x y) = (Size (max h1 h2)(d1 +
d2),list1++list2)
where (Size h1 d1, list1) = sem1 (Point x (y + abs (h2-h1)))
      (Size h2 d2, list2) = sem2 (Point (x + d1)(y + abs
(h1-h2)))
```

When verifying that this attribute grammar is strongly non-strict w.r.t. the algebra $\mathcal{A}$ we deduce that the synthesized attribute *size* does not depend on the inherited attribute *origin*. We can make the type of semantic boxes reflects more precisely this fact by letting

```
data SBox = SBox{list_ ::  Point -> [Point]
                ,size_ ::  Size}
```

we can also group the similar functions *hb*, *ht*, *vl*, and *vr* into one parametric function *comp*. We then end up with the Haskell code presented in Table 2.2 below.

```
lang ::  AlgBox SBox
lang = AlgBox elembox comp where
 -- elembox ::  SBox
 elembox = SBox (\ pt -> [pt])(Size 1 1)
 -- comp ::  Pos -> SBox -> SBox -> SBox
 comp pos box1 box2 = SBox list' size' where
   list' pt = (list box1 (pi1 pt))++(list box2 (pi2 pt))
   size' = case pos of
   Vert _ -> Size (max d1 d2)(h1 + h2)
   Hor _  -> Size (d1 + d2)(max h1 h2)
   pi1 (Point x y) = case pos of
       Vert Left_  -> Point x y
       Vert Right_ -> Point (x + (max (d2-d1) 0)) y
       Hor Top     -> Point x y
       Hor Bottom  -> Point x (y + (max (h2-h1) 0))
   pi2 (Point x y) = case pos of
       Vert Left_  -> Point x (y+h1)
       Vert Right_ -> Point (x + (max (d1-d2) 0)) (y+h1)
       Hor Top     -> Point (x+d1) y
       Hor Bottom  -> Point x (y + (max (h1-h2) 0))
   Size d1 h1 = size box1
   Size d2 h2 = size box2
```

Table 1: algebra associated with the attribute grammar for boxes composition

2.3 Rooted attribute grammars

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 together with associated semantic rules. For the *Box* example we can consider an additional sort *RBox*, for "box at root", with one synthesized attribute $return :: RBox \rightarrow [Point]$ and no inherited attribute. We add also an operator $Root :: Box \rightarrow RBox$ indicating that a given box is not a subbox of a larger one, but is the box at the root of the expression. Writing the following semantics rules

$$
\begin{array}{l}
Root :: Box_1 \longrightarrow RBox_\varepsilon \\
\left\{ \begin{array}{lcl}
Box_1 \cdot xCoord & = & 0 \\
Box_1 \cdot yCoord & = & 0 \\
RBox_\varepsilon \cdot return & = & Box_1 \cdot list
\end{array} \right.
\end{array}
$$

says that the intended end result is the *list* attribute of the top level box when it is positionned at the origin of the coordinate axes (*Point* 0 0).

Definition 5 *Signature Σ is said to be pointed if it has a specific sort a called its axiom; Then we let $\Sigma_\top$ represents the extended signature obtain by adding an additional sort $\top$, "at top", together with an additional operator $Root :: a \longrightarrow \top$. An attribute grammar $\mathbb{G} = (\Sigma_\bot, \Delta, Syn, Inh, R)$ whose input alphabet is the extended signature $\Sigma_\top$ is said to be rooted if sort $\top$ has one synthesized attribute $return :: top \longrightarrow a'$ and no inherited attribute. We let $init_{inh} = rhs_{Root,1,inh}$ denote the function that gives the value of the inherited attribute inh at the root node, and $result = rhs_{Root,\varepsilon,result}$ the functions that give the end result computed from the synthesized attributes at the root node.*

Thus a rooted attribute grammar has semantics rules presented as follows.

$$
\begin{array}{l}
op :: X_1 \times \cdots \times X_n \longrightarrow X_\varepsilon \\
\left\{ \begin{array}{lcll}
X_\varepsilon \cdot q & = & rhs_{op,\varepsilon,q} \left[X_\lambda \cdot q / x_{op,\lambda,q}\right] & \quad q \in Syn(s) \\
X_i \cdot q & = & rhs_{op,i,q} \left[X_\lambda \cdot q / x_{op,\lambda,q}\right] & \quad q \in Inh(s_i)
\end{array} \right.
\end{array}
$$

together with

$$
\begin{array}{l}
Root :: a_1 \longrightarrow \top_\varepsilon \\
\left\{ \begin{array}{lcll}
a_1 \cdot inh & = & init_{inh} \left(a_1 \cdot syn \,;\, syn \in Syn(a)\right) & \quad inh \in Inh(a) \\
\top_\varepsilon \cdot return & = & result \left(a_1 \cdot syn \,;\, syn \in Syn(a)\right) &
\end{array} \right.
\end{array}
$$

However we shall usually give the semantic rules attached to operator *Root* a special form by leaving the sort name $\top$ implicit and renaming operator *Root* by the name of the defined function, and replacing $\top_\varepsilon$ by the type of the computed result (the sort of attribute *return*). Thus we shall use presentation of the form:

$$
\begin{array}{l}
\langle \text{name of function} \rangle :: a \longrightarrow a' \\
\left\{ \begin{array}{ll}
a \cdot inh \,=\, init_{inh} \left(a \cdot syn \,;\, syn \in Syn(a)\right) & \quad inh \in Inh(a) \\
return \;\; result \left(a \cdot syn \,;\, syn \in Syn(a)\right) &
\end{array} \right.
\end{array}
$$

For instance the top level function corresponding to the translation of a box into the list of origins of its elementary subboxes could be written as:

$$
\begin{array}{l}
Root :: Box \longrightarrow [Point] \\
\left\{ \begin{array}{l}
Box \cdot xCoord \,=\, 0 \\
Box \cdot yCoord \,=\, 0 \\
return \;\; Box \cdot list
\end{array} \right.
\end{array}
$$

A rooted attribute grammar transforms a tree $t \in T(\Sigma)_a$ over the input signature into the tree $return(\top(t)) \in T(\Delta)_{a'}$ over the output signature.

3 Attribute grammars as zipper transformers

As for transducers, we would like to be able to interpret the semantics equations of the attribute grammar as rewriting rules so that the computation of attributes is simulated by reduction to normal forms. The main difference is that the value of an attribute q at a given node u of a tree t, even a synthesized attribute, may (and generally) depend of the context of the subtree t_u rooted at that node. We have said that the values of the synthesized attributes of a node u are determined by the values of the other attributes from the subtree t_u which means that if the values of these attributes, say $\langle q_i \rangle_{i \in I}$, are known then the value of the synthesized attribute q can be inferred. But the value of an inherited attribute of u (which belongs that that family $\langle q_i \rangle_{i \in I}$) depends, by definition, on values of attributes from the context of t_u. Therefore, in general, even the value of synthesized attribute of a node u *indirectly* depends on information coming from the context of the subtree. Attribute grammar were actually introduced for the very purpose of enabling the designer of a language to manipulate context-dependant information like the scope of a variable in a program for instance.

3.1 Rewriting a zipper

In order to account for context-dependant information we manipulate a subtree together with its corresponding context. We assume a particular sort a , called axiom of the grammar, and 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

$$C[op(t_1, \cdots, t_{i-1}, [], t_{i+1}, \cdots, t_n)]$$

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

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

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

Interpretation of the semantic rule $x_{op,\varepsilon,q} = rhs_{op,\varepsilon,q}$ for $op : s_1 \times \cdots \times s_n \to s$ and $q \in Syn(s)$ is then given by the following rewriting rule

$$\begin{aligned} q\left(\hat{x}_\varepsilon @ op(x_1,\ldots,x_n)\right) \longrightarrow rhs_{op,\varepsilon,q} \, [\;\; & q\left(\hat{x}_\varepsilon @ op(x_1,\ldots,x_n)\right) / x_{op,\varepsilon,q} \, ; \\ & q\left(op^i\left(x_1,\ldots,x_{i-1},\hat{x}_\varepsilon,x_{i+1},\ldots,x_n\right) @ x_i\right) / x_{op,i,q} \,] \end{aligned}$$

whose interpretation is as follows. If the subtree t of zipper $c@t$ matches the pattern $\hat{x}_\varepsilon @ op(x_1,\ldots,x_n)$, i.e. $t = op(t_1,\ldots,t_n)$, then the expression $q(c@t)$, standing for the value of the synthesized attribute q of subtree t in 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 $x_{op,i,q} = rhs_{op,i,q}$ for $op : s_1 \times \cdots \times s_n \rightarrow s$, and $q \in Inh(s_i)$ is given by the rewritng rule:

$$\begin{array}{l} q\left(op^i\,(x_1,\ldots,x_{i-1},\hat{x}_\varepsilon,x_{i+1},\ldots,x_n)\,@x_i\right) \longrightarrow \\ rhs_{op,\varepsilon,q}\left[\,q\,(\hat{x}_\varepsilon @op(x_1,\ldots,x_n))\,/x_{op,\varepsilon,q}\ ;\ q\left(op^i\,(x_1,\ldots,x_{i-1},\hat{x}_\varepsilon,x_{i+1},\ldots,x_n)\,@x_i\right)/x_{op,i,q}\,\right] \end{array}$$

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

$$\begin{array}{rcl} inh\,(Nil@Root(x)) & \longrightarrow & init_{inh}[q\,(Nil@Root(x))\,/x_{Root,1,q} \quad q \in Syn(a)] \\ result\,(Nil@Root(x)) & \longrightarrow & return[q\,(Nil@Root(x))\,/x_{Root,1,q} \quad q \in Syn(a)] \end{array}$$

Table 3.1 gives the Haskell code for our running example using the above approach. Each attribute is implemented by a function defined by structural induction on its first parameter. It has two parameters corresponding to the subtree and its context; the first of which is the subtree if the attribute is synthesized and the context if the attribute is inherited. By completeness and because the attribute grammar is rooted, we do have, in such a definition, one clause associated with each compatible constructor. For instance the clause

```
origin Nil box = Point 0 0
```

shows that the inherited attribute *origin* is initialized to the value *Point* 0 0) at the root node (i.e. when the context is empty). The function *init* is, in this example, a constant function; but in more complex situations it could have depended upon the value of the synthesized attributes of the root. The clause

```
result box = list box Nil
```

says that the end result is given by the value of the synthesized attribute *list* at the root of the tree.

3.2 A Simplified implementation of attribute evaluation

The rewriting system given in the previous section provides a satisfactory operational semantics for attribute evaluation; moreover it leads to a simple encoding of the attribute grammar into a lazy functional language like Haskell. However this presentation may be further simplified. Maybe the opportunities for improvement are more patent on the associated Haskell code. Let us consider the following excerpt from the definition of the inherited attribute *origin*

```
origin (First pos cxt box2) box1 =
   case pos of
      ......................................................
   where Point xcoord ycoord = origin cxt (Comp pos box1 box2)
         Size d1 h1 = size box1 (First pos cxt box2)
         Size d2 h2 = size box2 (Snd pos box1 cxt)
```

which shows that if we have a tree of the form $box = cxt\,[Comp_{pos}\,box_1\,box_2]$ the origin of the subbox box_1 in context $First_{pos}\,cxt\,box_2$ depends on the origin determined by context cxt for the surrounding box $Comp_{pos}\,box_1\,box_2$, and the size of the two subboxes box_1 and box_2 in their respective contexts $First_{pos}\,cxt\,box_2$ and $Snd_{pos}\,box_1\,cxt$. When the various parameters of the function *origin* are given as

```
data Box = Elembox | Comp{pos::Pos, first,second::Box}
data Cxt = First Pos Cxt Box | Snd Pos Box Cxt | Nil
data Pos = Hor HPos | Vert VPos
data VPos = Left_ | Right_
data HPos = Top | Bottom
data Point = Point{xcoord, ycoord ::Int} deriving Show
data Size = Size{depth, height ::Int} deriving Show

size ::  Box -> Cxt -> Size
size Elembox cxt = Size 1 1
size (Comp pos box1 box2) cxt = case pos of
     Vert _ -> Size (max d1 d2)(h1 + h2)
     Hor _  -> Size (d1 + d2)(max h1 h2)
        where Size d1 h1 = size box1 (First pos cxt box2)
              Size d2 h2 = size box2 (Snd pos box1 cxt)

list ::  Box -> Cxt -> [Point]
list Elembox cxt = [(origin cxt Elembox)]
list (Comp pos box1 box2) cxt =
   (list box1 (First pos cxt box2)) ++ (list box2 (Snd pos box1 cxt))

origin ::  Cxt -> Box -> Point
origin Nil box = Point 0 0
origin (First pos cxt box2) box1 =
   case pos of
      Vert Left_     -> Point xcoord ycoord
      Vert Right_    -> Point (xcoord + (max (d2-d1) 0)) ycoord
      Hor Top        -> Point xcoord ycoord
      Hor Bottom     -> Point xcoord (ycoord + (max (h2-h1) 0))
   where Point xcoord ycoord = origin cxt (Comp pos box1 box2)
         Size d1 h1 = size box1 (First pos cxt box2)
         Size d2 h2 = size box2 (Snd pos box1 cxt)
origin (Snd pos box1 cxt) box2 =
   case pos of
      Vert Left_     -> Point xcoord (ycoord+h1)
      Vert Right_    -> Point (xcoord + (max (d1-d2) 0))(ycoord+h1)
      Hor Top        -> Point (xcoord+d1) ycoord
      Hor Bottom     -> Point (xcoord+d1)(ycoord + (max (h1-h2) 0))
   where Point xcoord ycoord = origin cxt (Comp pos box1 box2)
         Size d1 h1 = size box1 (First pos cxt box2)
         Size d2 h2 = size box2 (Snd pos box1 cxt)

result box = list box Nil
```

Table 2: evaluation of attributes

```
origin (First pos cxt box2) box1
```

we already know that context *cxt* corresponds to the subtree $Comp_{pos}\ box_1\ box_2$ and that box_1 and box_2 have respective contexts $First_{pos}\ cxt\ box_2$ and $Snd_{pos}\ box_1\ cxt$. It is therefore useless to force the programmer to make these extra parameters explicit since they can automatically be inferred; moreover this unecessary constraint on programming introduces possible risk of errors. Leaving these extra parameters implicit however induces a change on the arity of the functions associated with attributes. They now become unary functions: An inherited attribute has a context parameter and a synthesized attribute has a subtree parameter; however a context should know about the subtree it is attached to and a subtree should know its context. This in turn induces a change on the interpretation of the various constructors. For instance the context constructor $First_{pos}$ needs an additional parameter to supply the subtree the considered context is attached to. The above piece of code could then write as

```
origin (First pos cxt box1 box2) =
   case pos of
      .................................................
   where Point xcoord ycoord = origin cxt
         Size d1 h1 = size box1
         Size d2 h2 = size box2
```

We end up with the Haskell code given in Table 3.2.

```
data Box = Elembox cxt | Comp Pos Cxt Box Box
data Cxt = First Pos Cxt Box Box | Snd Pos Cxt Box Box | Nil Box
data Pos = Hor HPos | Vert VPos
data VPos = Left_ | Right_
data HPos = Top | Bottom
data Point = Point{xcoord, ycoord ::Int} deriving Show
data Size = Size{depth, height ::Int} deriving Show

size ::  Box -> Size
size (Elembox cxt) = Size 1 1
size (Comp pos cxt box1 box2) = case pos of
     Vert _ -> Size (max d1 d2)(h1 + h2)
     Hor _  -> Size (d1 + d2)(max h1 h2)
        where Size d1 h1 = size box1
              Size d2 h2 = size box2

list ::  Box -> [Point]
list (Elembox cxt) = [(origin cxt)]
list (Comp pos cxt box1 box2) = (list box1)++(list box2)

origin ::  Cxt -> Point
origin (Nil box) = Point 0 0
origin (First pos cxt box1 box2) =
   case pos of
      Vert Left_     -> Point xcoord ycoord
      Vert Right_    -> Point (xcoord + (max (d2-d1) 0)) ycoord
      Hor Top        -> Point xcoord ycoord
      Hor Bottom     -> Point xcoord (ycoord + (max (h2-h1) 0))
   where Point xcoord ycoord = origin cxt
         Size d1 h1 = size box1
         Size d2 h2 = size box2
origin (Snd pos cxt box1 box2) =
   case pos of
      Vert Left_     -> Point xcoord (ycoord+h1)
      Vert Right_    -> Point (xcoord + (max (d1-d2) 0))(ycoord+h1)
      Hor Top        -> Point (xcoord+d1) ycoord
      Hor Bottom     -> Point (xcoord+d1)(ycoord + (max (h1-h2) 0))
   where Point xcoord ycoord = origin cxt
         Size d1 h1 = size box1
         Size d2 h2 = size box2
```

Table 3: evaluation of attributes: second version

3.3 Zippers as cyclic data structures

If we compare the codes given in respectively Table 3.1 and in Table 3.2, we observe that we have transformed a program that evaluate attributes by visiting the nodes of a tree into a simple inductive algorithm operating on a tree representation of a complex cyclic data structure. We have thus transfer the complexity from the algorithmic side onto the data structure side by replacing a complex algorithm defined on a simple inductive data structure into a simple algorithm operating on a complex cyclic data structure. We will establish shortly the equivalence between the two approaches. But before doing so we explicit, in this section, the nature of these cyclic data structures and, in the next section, we explain how one can transform a zipper into such a cyclic data structure. The transformation of the attribute evaluation algorithm amounts to replace the signature $\mathcal{Z}_\Sigma$ given in Definition 6 by the following signature $\mathcal{Z}(\Sigma)$.

Definition 7 *In signature $\mathcal{Z}(\Sigma)$ we find two sorts denoted s, and $\hat{s}$ associated with each sort $s \in \mathcal{S}$ in Σ and each operator $op :: s_1 \times \cdots \times s_n \rightarrow s$ gives rise to a family of operators op_λ for $\lambda \in \{\varepsilon\} \cup \{1, \ldots, n\}$ where $op_\varepsilon :: \hat{s} \times s_1 \times \cdots \times s_n \rightarrow s$ and $op_i :: \hat{s} \times s_1 \times \cdots \times s_n \rightarrow \hat{s}_i$. If Σ is a pointed signature with axiom $a \in \mathcal{S}$, we shall further impose in $\mathcal{Z}(\Sigma_\top)$ that $\hat{\top} = ()$ is the unit type and we let $Root = Root_\varepsilon$ and $Nil = Root_1$ which therefore have arities and sorts given by: $Nil :: a \rightarrow \hat{a}$, and $Root :: a \rightarrow \top$. Observe that $\mathcal{Z}(\Sigma_\top) = (\mathcal{Z}'(\Sigma))_\top$ where $\mathcal{Z}'(\Sigma)$ is the pointed signature with axiom a and whose operators op_ε, op_i, and Nil allow to define contexts and subtrees: A closed term $t \in T(\mathcal{Z}'(\Sigma))_s$ is a representation of a subtree of type s and a closed term $c \in T(\mathcal{Z}'(\Sigma))_{\hat{s}}$ is a representation of a context of type s.*

However most of the closed terms builds from this signature $\mathcal{Z}'(\Sigma)$ are not valid representations of subtrees or contexts. Let us illustrate this phenomenon with the example of double-linked streams. If A is an alphabet, a stream is a tree on the monosorted signature Σ (with sort $\mathcal{S} = \{st\}$) containing one unary operator $a :: st \rightarrow st$ 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.

```
data Stream a = Cons{val::a, suc::Stream a}
```

The signature $\mathcal{Z}_\Sigma$ provides the associated structure of zipper

```
data Stream a = Cons{val::a, suc::Stream a}
data StreamCxt a = Snoc{val::a, pred::StreamCxt a} | Nil
data StreamZipper = (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 double-linked stream:

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

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

```
data DList a = Node{val::  a, prev::CxtDList, suc ::DList a}
             | Nil (CxtDList)
data CxtDList a = Root (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 double-linked lists:

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

An implicit assumption is that if *suc xs* is defined then $prev\,(suc\,xs) = xs$, and if *prev xs* is defined then $suc\,(prev\,xs) = xs$, similarly $prev\,xs = Nil\,ys$ or $suc\,xs = Nil\,ys$ entails $ys = xs$. These conditions are met in the following double-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 [3, 10]. However these equations are usually 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, 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 identify 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 determine 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 isomorphic. 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; relatively few studies have been conducted on cyclic data structures apart from e.g. [15, 24, 7, 14, 18] that could serve as starting points.

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

```
data StrCoalg b a = StrCoalgout::b->a, next::b->b

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

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

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

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

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

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

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

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

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

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

```
stream2dStream str = dstr
  where dstr = build str (Root dstr)
        build (Str val suc) cxts = Node val cxts dstr'
          where dstr' = build suc (CoNode val cxts dstr')
```

Now one can navigate a double-linked stream:

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

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

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

3.4 Generic attribute grammars

In this section we generalize on the previous example of double-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 rooted attribute grammar $\mathbb{G}_\Sigma = (\Sigma_\top, \mathcal{Z}(\Sigma_\top), Syn, Inh, \mathbb{G}_\Sigma(R))$ associated with pointed signature $\Sigma = (\mathcal{S}, O_p)$ with axiom $a \in \mathcal{S}$ is defined as follows. Its input alphabet is signature $\Sigma_\top$, and its output alphabet is signature $\mathcal{Z}(\Sigma_\perp)$. It has one inherited attribute associated with each sort $Inh = \{cxt_s | s \in \mathcal{S}\}$ representing the context at the given node of the tree, and one synthesized attribute $Syn = \{tree_s | s \in \mathcal{S}\}$ representing the subtree rooted at that node, with arities and sorts given by $tree_s :: s \longrightarrow s$*

and $cxt_s :: s \longrightarrow \hat{s}$, plus a synthesized attribute $return :: \top \longrightarrow \top$. The semantics rules associated with operator $op :: s_1 \times \cdots \times s_n \longrightarrow s$ are given by

$$op :: X_1 \times \cdots \times X_n \to X$$
$$\begin{cases} X \cdot tree_s &= op_\varepsilon(X \cdot cxt_s, X_1 \cdot tree_{s_1}, \ldots, X_n \cdot tree_{s_n}) \\ X_i \cdot cxt_{s_i} &= op_i(X \cdot cxt_s, X_1 \cdot tree_{s_1}, \ldots, X_n \cdot tree_{s_n}) \end{cases}$$

and the top level function is given by:

$$eval :: X \longrightarrow \top$$
$$\begin{cases} a \cdot cxt_a = Nil(a \cdot tree_a)) \\ return\ Root(a \cdot tree_a) \end{cases}$$

We let $\mathcal{U} = \mathbb{G}_\Sigma(T(\mathcal{Z}(\Sigma_\top)))$ for "unfolding" denote the algebra induced by attribute grammar $\mathbb{G}_\Sigma$ from the free algebra $T(\mathcal{Z}(\Sigma_\perp))$ and we let

$$unfold\, t = t' \quad \text{where} \quad Root(t') = (Root(t))^{\mathcal{U}}$$

denote the unfolding of a tree $t \in Tree(\Sigma)_a$ into its cyclic representation in $Tree(\mathcal{Z}(\Sigma))_a$. This algebra is given by function

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

associated with operator $op :: s_1 \times \cdots \times s_n \longrightarrow s$ together with the top level function:

$$\begin{aligned} Root^{\mathcal{U}}\ build &= Root(tree) \\ \text{where} \quad tree &= build\ (Nil(tree)) \end{aligned}$$

And the unfolding function is given by

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

For instance we can add the following to the code in Table 3.2 in order to translate a box into its cyclic representation.

```
data Bx = Elm | Cmp Pos Bx Bx

unfold ::  Bx -> Box
unfold bx = box
   where box = build bx (Nil box)
         build Elm cxt = Elembox cxt
         build (Cmp pos bx1 bx2) cxt = Comp pos cxt box1 box2
                  where box1 = build bx1 (First pos cxt box1 box2)
                        box2 = build bx2 (Snd pos cxt box1 box2)
```

In the example of double-linked streams we introduced this unfolding function as a special case of the function generating a double-linked list from a stream coalgebra. The same can be done mre generallt by using an adaptation of the above unfolding function as follows:

```
data Trunk a = Elm | Cmp Pos a a
data Bx = In{out::Trunk Bx}
data BxCoalg a = BxCoalg{next::a->Trunk a}

punfold ::  BxCoalg a -> a -> Box
punfold coalg gen = box
   where box = build gen (Nil box)
         build gen cxt = case next coalg gen of
          Elm  -> Elembox cxt
          Cmp pos gen1 gen2 -> Comp pos cxt box1 box2
                  where box1 = build gen1 (First pos cxt box1 box2)
                        box2 = build gen2 (Snd pos cxt box1 box2)

unfold = punfold (BxCoalg out)
```

More generally this parametric unfolding will be given as:

$$
\begin{array}{l}
\quad punfold\ coalg\ gen\ =\ tree\\
\qquad \text{where}\quad tree\ =\ build_a\ gen\ (Root\, tree)\\
\qquad\qquad\quad build_s\ gen\ cxt\ =\ case\ coalg\ gen\ of\\
\qquad\qquad\qquad op(gen_1,\ldots,gen_n)\ \rightarrow\ op_\varepsilon\,(cxt,tree_1,\ldots,tree_n)\\
\qquad\qquad\qquad\quad \text{where}\, tree_i\ =\ build_{s_i}\ gen_i\ op_i\,(cxt,tree_1,\ldots,tree_n)
\end{array}
$$

3.5 Transducer associated with an attribute grammar

The attribute grammar associated with a pointed signature given in the preceding section is generic in the sense that it leaves the semantic functions uninterpreted by representing them by the operators of signature $\mathcal{Z}(\Sigma_\top)$. The resulting evaluation function provides the unfolding of a tree, i.e. the tree representation of the associated cyclic data structure. But we can also consider the evaluation $t^{\mathcal{E}}$ of a tree with respect to the algebra $\mathcal{E} = \mathbb{G}_\Sigma \mathcal{S}$ for some $\mathcal{Z}(\Sigma_\top)$-algebra $\mathcal{S}$. Such an algebra $\mathcal{S}$ consists of the semantic domains $\mathcal{S}_s$, $\mathcal{S}_{\hat{s}}$, and $\mathcal{S}_\top$ that we interpret as the domains of values for respectively the synthesized attributes of sort s, the inherited attributes of sort s and the produced end result; together with the semantic functions $op_\varepsilon^{\mathcal{S}}$, $op_i^{\mathcal{S}}$, and function $init = Nil^{\mathcal{S}}$ giving the initialization of the inherited attributes at the root node, and function $return = Root^{\mathcal{S}}$ for extracting the value of the end result. Altogether the algebra $\mathcal{S}$ are the semantic functions that can be derived from the transducer associated with rooted attribute grammar which we now define.

Definition 9 *The transducer $\mathbb{T}_\mathbb{G} = (\mathcal{Z}(\Sigma_\top), \Delta, Q, \mathcal{Z}(R))$ associated with a rooted attribute grammar $\mathbb{G} = (\Sigma_\top, \Delta, Syn, Inh, R)$ is defined as follows.*

- *the set of states $Q = Inh \cup Syn$ is made of the inherited and synthesized attributes with $q :: s_1 \rightarrow s_2$ if q is a synthesized attribute of s_1 of sort s_2 (including the synthesized attribute return of $\top$) and $q :: \hat{s}_1 \rightarrow s_2$ if q is an inherited attribute of s_1 of sort s_2.*

- *For each operator $op :: s_1 \times \cdots \times s_n \rightarrow s$ in Σ and each $q \in Syn(s)$ we have rule*

$$q\,(op_\varepsilon(x_\varepsilon, x_1, \ldots, x_n)) \longrightarrow rhs_{op,\varepsilon,q}[q(x_\lambda)/x_{op,\lambda,q}]$$

  For each operator $op :: s_1 \times \cdots \times s_n \rightarrow s$ in Σ and each $q \in Inh(s_i)$ we have rule

$$q\,(op_i(x_\varepsilon, x_1, \ldots, x_n)) \longrightarrow rhs_{op,i,q}[q(x_\lambda)/x_{op,\lambda,q}]$$

  Recall that variable $x_{op,\lambda,q}$ can occur in the right-hand side of a rule if either $\lambda = \varepsilon$ and $q \in Inh(s)$ or $\lambda = i \in \{1,\ldots,n\}$ and $q \in Syn(s_i)$.

- *Associated with operator Root* $:: a \to \top$ *we similarly have the two rules*

$$\begin{array}{lcll} inh(Nil(x)) & \longrightarrow & init_{inh}[q(x);\ q \in Syn(a)] & \textit{for } inh \in Inh(a) \\ return(Root(x)) & \longrightarrow & result[q(x);\ q \in Syn(a)] & \end{array}$$

We recall that $init_{inh} = rhs_{Root,1,inh}$ *and* $result = rhs_{Root,\varepsilon,result}$ *and we have assumed that* $\hat{\top} = (\,)$ *is the unit type and we have let* $Root = Root_\varepsilon$ *and* $Nil = Root_1$ *which therefore have arities and sorts given by:* $Nil :: a \to \hat{a}$, *and* $Root :: a \to \top$.

Proposition 10 *For any* Δ*-algebra* $\mathcal{A}$

$$\mathbb{G}\mathcal{A} \ = \ \mathbb{G}_\Sigma(\mathbb{T}_\Sigma \mathcal{A})$$

Proof. The carriers of the induced algebra $\mathcal{B} = \mathbb{G}\mathcal{A}$ are $\mathcal{B}_\top = \mathcal{A}_{a'}$, and

$$\mathcal{B}_s \ = \ \prod_{q \in Inh(s)} \mathcal{A}_{\sigma(q)} \longrightarrow \prod_{q \in Syn(s)} \mathcal{A}_{\sigma(q)}$$

and its functions of interpretation are given by

$$\begin{array}{rcl} op^{\mathcal{B}}(f_1,\ldots,f_n)(v)(q) & = & rhs^{\mathcal{A}}_{op,\varepsilon,q}[v(q)/x_{op,\varepsilon,q};\ f_i(v_i)(q)/x_{op,\varepsilon,q}] \\ \text{where} \quad v_i(q) & = & rhs^{\mathcal{A}}_{op,i,q}[v(q)/x_{op,\varepsilon,q};\ f_i(v_i)(q)/x_{op,\varepsilon,q}] \end{array}$$

and

$$\begin{array}{rcl} Root^{\mathcal{B}}\ build & = & result^{\mathcal{A}}(syn) \\ \text{where}\ syn & = & build\left(init^{\mathcal{A}}(syn)\right) \end{array}$$

Let $\mathcal{S} = \mathbb{T}_{\mathbb{G}}\mathcal{A}$ be the $\mathcal{Z}(\Sigma_\top)$-algebra induced by Δ-algebra $\mathcal{A}$ and transducer $\mathbb{T}_{\mathbb{G}}$. We have

$$\mathcal{S}_s = \prod_{q \in Syn(s)} \mathcal{A}_{\sigma(q)} \qquad \mathcal{S}_{\hat{s}} = \prod_{q \in Inh(s)} \mathcal{A}_{\sigma(q)} \qquad \mathcal{S}_\top = \mathcal{A}_{a'}$$

and

$$\begin{array}{rcl} op^{\mathcal{S}}_\varepsilon(inh_\varepsilon, syn_1,\ldots,syn_n)(q) & = & rhs^{\mathcal{A}}_{op,\varepsilon,q}[inh_\varepsilon(q)/x_{op,\varepsilon,q};\ syn_i(q)/x_{op,\varepsilon,q}] \\ op^{\mathcal{S}}_i(inh_\varepsilon, syn_1,\ldots,syn_n)(q) & = & rhs^{\mathcal{A}}_{op,i,q}[inh_\varepsilon(q)/x_{op,\varepsilon,q};\ syn_i(q)/x_{op,\varepsilon,q}] \end{array}$$

together with

$$Root^{\mathcal{S}}(syn) \ = \ result^{\mathcal{S}}(syn) \qquad Nil^{\mathcal{S}}(syn)(inh) \ = \ init^{\mathcal{S}}_{inh}(syn)$$

The carriers of the algebra $\mathcal{E} = \mathbb{G}_\Sigma \mathcal{S}$ are given by

$$\mathcal{E}_s \ = \ \prod_{q \in Inh(x)} \mathcal{A}_{\sigma(q)} \longrightarrow \prod_{q \in Syn(x)} \mathcal{A}_{\sigma(q)}$$

hence

$$\begin{array}{l} \mathcal{E}_s \ = \ \mathcal{S}_{\hat{s}} \to \mathcal{S}_s \ = \ \prod_{q \in Inh(s)} \mathcal{A}_{\sigma(q)} \longrightarrow \prod_{q \in Syn(s)} \mathcal{A}_{\sigma(q)} \ = \ \mathcal{B}_s \\ \mathcal{E}_\top \ = \ \mathcal{S}_\top \ = \ \mathcal{A}_{a'} \ = \ \mathcal{B}_\top \end{array}$$

Concerning the functions of interpretation we observe thet

$$\begin{array}{rcl} op^{\mathcal{E}}(f_1,\ldots,f_n)(cxt) & = & op^{\mathcal{S}}_\varepsilon(cxt, tree1,\ldots,tree_n) \\ \text{where} \quad tree_i & = & f_i(op^{\mathcal{S}}_i(cxt, tree1,\ldots,tree_n)) \end{array}$$

i.e.

$$\begin{array}{rcl} op^{\mathcal{E}}(f_1,\ldots,f_n)(cxt)(q) & = & rhs^{\mathcal{A}}_{op,\varepsilon,q}[cxt(q)/x_{op,\varepsilon,q};\ f_i(v_i)(q)/x_{op,\varepsilon,q}] \\ \text{where} \quad v_i(q) & = & op^{\mathcal{S}}_i(cxt, tree1,\ldots,tree_n))(q) \\ & = & rhs^{\mathcal{A}}_{op,i,q}[cxt(q)/x_{op,\varepsilon,q};\ f_i(v_i)(q)/x_{op,\varepsilon,q}] \end{array}$$

and

$$Root^{\mathcal{E}}\ build \;=\; Root^{\mathcal{S}}\ tree \quad \text{where}\ tree = build(Nil^{\mathcal{S}}(tree))$$

hence

$$Root^{\mathcal{E}}\ build \;=\; result^{\mathcal{A}}(syn) \quad \text{where}\ syn = build(init^{\mathcal{A}}(syn))$$

and therefore $\mathcal{E} = \mathcal{B}$. □

The following result shows that evaluating a rooted expression by an attribute grammar can be obtained by evaluation of its unfolding by the associated transducers; thus showing the equivalence of the programs given respectively in Table 3.1 and Table 3.2.

Corollary 11 *For every Δ-algebra $\mathcal{A}$, and rooted expression $t \in Tree(\Sigma_\top)_\top$ (i.e. of the form $t = Root(t_0)$ where $t_0 \in Tree(\Sigma)_a$) one has:*

$$t^{\mathbb{G}\mathcal{A}} \;=\; \left(t^{\mathcal{U}}\right)^{\mathbb{T}_{\mathbb{G}}\mathcal{A}}$$

Proof. The canonical morphism $(\ )^{\mathbb{T}_{\mathbb{G}}\mathcal{A}} : Tree(\Sigma_\top) \to \mathbb{T}_{\mathbb{G}}\mathcal{A}$ is a continuous morphism of $\mathcal{Z}(\Sigma_\top)$-algebras and the result follows from Corollary 4 since $\mathbb{G}\mathcal{A} = \mathbb{G}_\Sigma(\mathbb{T}_{\mathbb{G}}\mathcal{A})$, and $\mathcal{U} = \mathbb{G}_\Sigma(Tree(\Sigma_\top))$, and a rooted tree has no inherited attribute and a unique synthesized attribute. □

4 Conclusion

We have presented a transformation of an attribute grammar, viewed as a tree transformer, into a tree transducer having an extended input signature for describing cyclic representations of zippers for its input signature. In this concluding section we mention some potential applications of that transformation.

4.1 Composition of attribute grammars

It is much easier to compose (top-down) tree transducers than to compute the syntactic composition of attribute grammars. The latter operation introduced by Ganzinger and Giegerich, as the co-called *attributed coupled grammars* [13], has already been related to the functional programming deforestation technique in [8, 11]. We would like to recover the syntactic composition of attribute grammars through the composition of the associated (top-down) tree transducers acting on cyclic representations of zippers.

4.2 Attribute grammars for the specification of reactive processes

We are also interested in using the formalism of attribute grammars for the specification of reactive processes that lazily produce data structures throught their synthesized attributes while consumming input data structure from their inherited attributes. Generally this kind of reactive processes, e.g. the so-called *Kahn Networks* [23], are specified in terms of a system of equations on streams. But as already noticed by Uustalu and Vene [26] zippers can be seen as generalized streams and therefore the representation of an attribute grammar as a zipper transformer can be used for that purpose. Moreover as noted by Caspi and Pouzet [5] while Haskell easily allow to write data-flow programs by expressing streams as abstract data type, this implementation can be inefficient as long as some kind of deforestation technique is not involved. But it happens, as established by Jürgensen and Vogler, that syntactic composition of top-down tree transducers is short cut fusion [21]. Thus if we succeed to relate syntactic composition of top-down tree transducers to the syntactic composition of attribute grammars through our encoding we would be in a position where the composition of attribute grammars can be presented as a synchronous composition of reactive processes acting on generalized streams (the zippers).

4.3 Combinators for the edition of structure documents

We can notice that the Haskell code presented in Table 3.2 is almost an immediate transcription of the semantic rules of the attribute grammar. Still the programmer need to be aware of the underlying cyclic representation of zippers and this is an undesirable overhead a potential source of programming errors. We would like to be able to encapsulate these aspects into a structure of monad (or a structure of arrows) so that all these considerations would be totally transparent to the programmer. Therefore, we are looking for a set of functional combinators (similar to the functional monadic parser combinators [17]) that would provide a Domain Specific Language embedded in Haskell for the encoding of attribute evaluators . Using these combinators the programmer will specify his attribute grammar (mainly he will write the semantic rules) but by doing so he will actually build an Haskell program for the corresponding evaluator of attributes or even maybe for an associated interactive editor.

References

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

[2] E. Badouel and M. Tonga. Growing a Domain Specific Language with Split Extensions. INRIA Research Report, October 2007.

[3] J.A. Bergstra, J. Heering, P. Klint, Eds. Algebraic Specification. ACM Press/Addison-Wesley, 1989.

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

[5] P. Caspi, M. Pouzet. Synchronous Kahn Networks. International Conference on Functional Programming, pp. 226-238, 1996.

[6] L.M. Chirica, D.F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical System Theory*, 13(1):1-27, 1979.

[7] C. Clack, S. Clayman, D. Parrott. Dynamic Cyclic Data Structures in Lazy Functional Languages. citeseer.ist.psu.edu/clack95dynamic.html, Dept. of Computer Science, University College London 1995.

[8] L. Correnson, E. Duris, D. Parigot, G. Roussel. Declarative Program Transformation: A Deforestation Case-Study. In *Principles and Pratice of Declarative Programming*, 1998, 360-377.

[9] B. Courcelle, P. Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes, I *Theoretical Computer Science* 17: 163-191, and II *Theoretical Computer Science* 17:235-257, 1982.

[10] A. van Deursen, J. Heering, P. Klint, Eds. Language Prototyping: An Algebraic Specification Approach, volume 5 of AMAST Series in Computing. World Scientific Publishing Co., 1996.

[11] E. Duris, D. Parigot, G. Roussel, M. Jourdan. Structured-directed Genericity in Functional Programming and Attribute Grammars. INRIA Research Report 3105, 1997.

[12] M. Fokkinga, J. Jeuring, L. Meertens, E. Meijer. A translation from Attribute Grammars to Catamorphisms. *The Squiggolist*, 2(1):20-26, 1991.

[13] H. Ganzinger, R. Giegerich. Attribute coupled grammars; In ACM SIGPLAN'84 Symp. on Compiler Construction, pp. 157-170, Montréeal, June 1984. Appeared in ACM SIGPLAN Notices 1986.

[14] N. Ghani, M. Hamana, T. Uustalu, V. Vene. Representing Cyclic Structures as Nested Datatypes. Proceedings of 7th Trends in Functional Programming, 2006, pp. 173-188.

[15] The Haskell Wiki. Tying The Knot, How to build a cyclic data structure. http://www.haskell.org/hawiki/TyingTheKnot .

[16] G. Huet. The Zipper. *Journal of Functional Programming* 7(5), Sept 1997, pp. 549-554.

[17] G. Hutton, E. Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming* 8(4), 1998.

[18] P. Johann, N. Ghani. Initial Algebra Semantics is Enough! Typed Lambda Calculus and Applications, June 2007.

[19] T. Johnsson. Attribute grammars as functional programming paradigm. In G. Kahn, ed, *Proc. of 3rd Int. Conf. on Functional Programming and Computer Architecture*, FPCA'87, vol. 274 of Lecture Notes in Computer Science, 154-173, Springer-Verlag, 1987.

[20] M. Jourdan. Strongly non-circular attribute grammars and their recursive evaluation. SIGPLAN Notices, 19(6), 81-93. *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, 1984.

[21] C. Jürgensen, H. Vogler. Syntactic composition of top-down tree transducers is short cut fusion. Mathematical Structure in Computer Science 14(2), 2004, 215-282.

[22] T. Katayama, Translation of attribute grammars into procedures. *ACM Transactions in Programming Languages and Systems*, 6, 345-369, 1984.

[23] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing* 74, pp. 993-998, North-Holland, 1977.

[24] N. Klarlund, M. I. Schartzbach. Graph Types. Proceedings of the 20th ACM SIGPLAN-SIGACT *Symposium on Principles of Programming Languages*, POPL'93, pp. 196-205, 1993.

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

[26] Tarmo Uustalu, Varmo Vene (2005). Comonadic functional attribute evaluation. In *Sixth Symposium on Trends in Functional Programming*, Tallinn, Estonia.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399