Growing a Domain Specific Language with Split Extensions

Éric Badouel — Marcel Tonga

N° 6314

October 2007

Thème COM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6314--FR+ENG

Growing a Domain Specific Language with Split Extensions

Éric Badouel[*], Marcel Tonga[†]

Thème COM — Systèmes communicants
Projets S4

Rapport de recherche n° 6314 — October 2007 — 18 pages

Abstract: In this paper we consider simple language extensions given by a pair of transducers. When these transducers are viewed as trees transformers, one of this transducer provides an embedding of the original language into its extension while the other, a left-inverse to the embedding, allows every expression of the extended language to be expanded into an expression of the original language. This is of course the easy case and the work presented here should be considered as a work in progress where we have reach only a very preliminary stage. The purpose of this report is to introduce our approach to the extension of domain specific languages, fix the notations and definitions and illustrate the case, of what might be termed the class of the *split extensions*, where the embedding of the language into its extension has a left-inverse. We shall, in the next future, investigate the general case of language extensions where the embedding doesn't necessarily have a left-inverse.

Key-words: Domain Specific Languages, Extensions of a Language, Tree Transducers

[*] ebadouel@irisa.fr
[†] Faculty of Sciences, University of Yaoundé I, tongamarcel@yahoo.fr

Extensions de langages dédiés

Résumé : Dans ce travail nous considérons des extensions simples de langages données par des paires de transducteurs. Lorsque ces transducteurs sont interprétés comme des transformateurs de termes, l'un d'entre eux décrit le plongement du langage de départ dans son extension, tandis que le second, qui est un inverse à gauche de ce plongement, associe à toute expression du langage étendu son expansion dans le langage de départ. Il s'agit de fait du cas simple et cette présentation ne décrit que l'étape préliminaire d'un travail en cours de développement. Le but de ce rapport est de poser les bases de notre approche de la notion d'extension de langages et de l'illustrer dans le cas des extensions, que nous pourrions qualifier de *scindées*, pour lesquelles le plongement admet un inverse à gauche. Dans la suite nous comptons élargir cette présentation à des extensions de langages dans lesquelles le plongement du langage dans son extension n'admet pas nécessairement un inverse à gauche.

Mots-clés : Langages dédiés, extensions de langages, transducteurs d'arbres

1 Introduction

Component-based design is acknowledged as an important approach to improving the productivity in the design of complex software systems, as it allows pre-designed components to be reused in larger systems [16]. Rather than constructing standalone applications one put emphasis on the use of libraries viewed as toolboxes for the development of software line products dedicated to some specific application domain. Using such "components on the shelf" improves productivity in developping software as well as it improves the adaptability of the produced software with respect to changes. Thus intellectual investment is better preserved. In order to avoid redundancies a well designed domain specific library should have generic constituents (using parametrization, inheritence or polymorphism) and then it can been seen as a small programming language in itself. Language oriented programming [4] is an approach to software composition based on domain specific languages (DSL) dedicated to specific aspects of an application domain. A DSL capture the semantics of a specific application domain by packaging reusable domain knowledge into language features. It can be used by an expert of that domain who is provided with familiar notations and concepts rather that confronted to a general purpose programming language.

Many DSL have been designed and used in the past decades like Open GL for 3D graphics, VHDL for hardware description, Lex and Yacc for the generation of lexical and syntatic analysers, Latex, HTML, Postscript for document manipulations, etc. However the systematic study of domain specific languages is a more recent concern. One of the difficulty is that the design and implementation of a programming language, even a simple one, is a difficult task; one has to develop all the tools necessary to support programming and debugging in that language: a compiler for source text analysis, type checking, generation and optimisation of code, handling of errors ... but also related tools for the generation of documentations, the integration of graphic and text editing facilities, the synchronization of multiple partial views, etc. A second difficulty comes from the lack of evolutivity of a language: it is very hard to make a change to the design of a programming language. However some domains of expertise may evolve in time, will we be forced to redesign the associated DSL form start at each time? Third, it might be difficult, if not impossible, to make different DSL collaborate within some application even though many applications do involve different domains of expertise.

We can alleviate these difficulties, as suggested by Hudak [10], by embedding these DSL into a host language. Hudak coined the expression DSEL for *Domain-Specific Embedded Languages*. Each DSEL inherits from the host language all parts that are not specific to the domain. It also inherits the compiler and the various tools used as a support to programming. Finally each DSEL is integrated into a general purpose language, namely its host language; and several DSEL with a common host language can communicate through this common host language. An higher-order strongly typed lazy functional language (Haskell) proved to be an adequate host language. This choice is quite natural since such a language can indeed be viewed as a DSL for denotational semantics!

Recent language workbenches [8] like *Intentional Programming* [20, 21] or the *Meta Programming System* [4] from JetBrains envisage a system where one could systematically scope and design DSL with the ability to compose a language for a particular problem by loading DSL as various plug-ins, where each such plug-in would incorporate meta programming tools allowing to program in the corresponding DSL (browsing, navigating and editing syntax, extracting multiple views or executable code ...).

In a similar manner, we are searching for a DSEL for designing DSEL, i.e. a language that can be used first for specifying DSEL and second for providing means to compose such specifications. It seems reasonable to base this language on the formalism of attribute grammars [15, 19] which is the most tested formalism for designing and implementing programming languages. Moreover one can then benefit from efficient functional implementations of attribute grammars [12, 1] that interpret attribute grammars as executable specifications of DSEL, therefore advocating using this formalism for the prototyping of languages. More specifically we take inspiration from [7] to provide a translation of an attribute grammar into an algebra for the signature given by the abstract syntax of the language. This algebra provides the interface between the DSEL and its host language. Using this algebraic characterization of DSEL together with the splitting operation on algebras introduced in [2] lead us to a notion of modular attributed grammars which allows us to address the issue of modularity and reusability in attribute grammars. This approach extends previous attempts [14, 5, 13] by allowing the imported attribute grammar to be parametrized by the attribute grammar that imports it so that we can have mutual recursivity between the two parts of the specification. Modularity is an important issue as it gives an efficient means to reuse a DSEL in the definition of a larger DSEL by allowing the latter to import the former in its specification. Modularity however is not the unique manner in which a DSEL can be reused. For instance there are situations where we are rather interested in extending some functionalities of a DSEL without of course having to explicitly rewrite the implementation of the DSEL (the constituents of the associated algebra) or even to be aware of the corresponding code. This is precisely the issue that we address in this contribution: we adopt an incremental approach consisting in growing a DSEL by adding new functionalities at the syntactical level and/or by extending its semantical interpretation (by adding new semantical aspects).

2 Multi-sorted and continuous algebras

In this section we recall some basic definitions and results on muti-sorted and continuous algebras that we shall use in this paper. These elements are essentially borrowed from the classical paper [9] with minor variations on notations.

2.1 Signatures and algebras

Definition 1 *A signature $\Sigma = (\mathcal{S}, O_p)$ consists of a finite set S of sorts, and a finite set O_p of operators. Each operator op has an arity $\alpha(op) \in \mathcal{S}^*$ and a sort $\sigma(op) \in \mathcal{S}$. We let notation $op :: s_1 \times \cdots \times s_n \rightarrow s$ mean that op is an operator of arity $\alpha(op) = s_1 \cdots s_n$ and sort $\sigma(op) = s$. The rank of operator op is the length of its arity: $\rho(op) = |\alpha(op)|$. If $\alpha(op) = \varepsilon$, op is said to be a constant of sort $\sigma(op)$.*

Definition 2 *Let $\Sigma = (\mathcal{S}, O_p)$ be a signature, a Σ-algebra $\mathcal{A}$ consists of a domain of interpretation, a set $\mathcal{A}_s$, for each sort $s \in \mathcal{S}$, and a function $op^{\mathcal{A}} : \mathcal{A}_{s_1} \times \cdots \times \mathcal{A}_{s_n} \rightarrow \mathcal{A}_s$ associated with each operator $op :: s_1 \times \cdots \times s_n \rightarrow s$. A morphism of algebras $f : \mathcal{A} \rightarrow \mathcal{B}$ is a family of maps $f_s : \mathcal{A}_s \rightarrow \mathcal{B}_s$ such that*

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

for every $a_i \in \mathcal{A}_{s_i}$.

There exists an initial Σ-algebra whose elements are the (closed) terms defined as follows. The sets $T(\Sigma)_s$ of terms of sort s are mutually defined by induction as the least vector of languages (ordered componentwise by set theoretic inclusion) such that:

$$(op :: s_1 \times \cdots \times s_n \to s \quad \wedge \quad t_i \in T(\Sigma,X)_{s_i} \quad 1 \leq i \leq n) \Rightarrow op(t_1,\ldots,t_n) \in T(\Sigma,X)_s$$

(where a term is represented by a word over the alphabet whose symbols are the operators of the signature together with three additional symbols: the left and right parenthesis and the comma). We let $([\mathcal{A}]) : T(\Sigma) \to \mathcal{A}$ denote the canonical morphism from the term algebra to algebra $\mathcal{A}$, and $t^{\mathcal{A}} = ([\mathcal{A}])_s(t)$ the interpretation of a closed term $t \in T(\Sigma)_s$ in algebra $\mathcal{A}$.

2.2 Equations

In order to state equations we need expressions with variables: If X is a $\mathcal{S}$-sorted set of variables (i.e. each element $x \in X$ is associated with a sort $\sigma(x) \in \mathcal{S}$) we let terms with variables in X be defined as the (closed) terms over the extended signature $\Sigma + X$ obtained by considering variables as additional constant operators: $T(\Sigma,X)_s = T(\Sigma + X)_s$; i.e. they are inductively defined as follows

- a variable is a term: $x \in T(\Sigma,X)_{\sigma(x)}$
- if $op :: s_1 \times \cdots \times s_n \to s$ and $t_i \in T(\Sigma,X)_{s_i}$ for $1 \leq i \leq n$, then $op(t_1,\ldots,t_n) \in T(\Sigma,X)_s$

An algebra for this extended signature is given by an algebra for signature Σ together with a vector $\nu \in \prod_{x\in X} \mathcal{A}_{\sigma(x)}$, called a *valuation*, and we let

$$t^{\mathcal{A}} : \prod_{x\in X} \mathcal{A}_{\sigma(x)} \longrightarrow \mathcal{A}_{\sigma(t)}$$

the interpretation of term $t \in T(\Sigma,X)_s$ (viewed as a derived operator) in algebra A be given as $t^{\mathcal{A}}(\nu) = t^{(\mathcal{A},\nu)}$; i.e. $t^{\mathcal{A}}(\nu)$ is the interpretation of the closed term t (with respect to the extended signature) in algebra $(\mathcal{A},\nu)$. The universal property of $([\mathcal{A},\nu]) = \lambda t \cdot t^{\mathcal{A}}(\nu)$ gives the following equivalent inductive definition:

- $x^{\mathcal{A}}(\nu) = \nu(x)$, i.e. $x^{\mathcal{A}}$ is the projection that gives the value at entry x,
- $(op(t_1,\ldots,t_n))^{\mathcal{A}}(\nu) = op^{\mathcal{A}}\left(t_1^{\mathcal{A}}(\nu),\ldots,t_n^{\mathcal{A}}(\nu)\right)$

We shall identify a term $t \in T(\Sigma,X)_s$ with its interpretation $t^{T(\Sigma)}$ in the free algebra in the same manner as an operator op was identified to a term constructor; it is in this sense that term t is viewed as a derived operator on terms. Now, the value $t^{\mathcal{A}}(\nu)$ depends only on the entries of valuation ν associated with variables $Var(t) = \{x_1,\ldots,x_n\}$ occurring in term $t \in T(\Sigma,X)_s$ thus we let $t^{\mathcal{A}}[a_i/x_i]$ represent the value $t^{\mathcal{A}}(\nu)$ for any valuation ν s.t. $\nu(x_i) = a_i$:

- $x_i^{\mathcal{A}}[a_i/x_i] = a_i$,
- $(op(t_1,\ldots,t_n))^{\mathcal{A}}[a_i/x_i] = op^{\mathcal{A}}\left(t_1^{\mathcal{A}}[a_i/x_i],\ldots,t_n^{\mathcal{A}}[a_i/x_i]\right)$

When $\mathcal{A}$ is the term algebra, the above definition gives the *substitution operation* on terms.

A system of equations is a map $E : X \to T(\Sigma, X)$ that is compatible with sorts: $\sigma(x) = \sigma(E(x))$. A solution of E in an algebra $\mathcal{A}$ is any valuation $\nu \in \prod_{x \in X} \mathcal{A}_{\sigma(x)}$ such that

$$\forall x \in X \qquad \nu(x) = E(x)^{\mathcal{A}}(\nu)$$

i.e. it is a fixed point of the transformation

$$E^{\mathcal{A}} : \prod_{x \in X} \mathcal{A}_{\sigma(x)} \longrightarrow \prod_{x \in X} \mathcal{A}_{\sigma(x)}$$

given by $E^{\mathcal{A}}(\nu)(x) = E(x)^{\mathcal{A}}(\nu)$.

2.3 Continuous algebras

Continuous algebras are introduced in order to ensure the existence of solutions for such systems. There exists several notions of continuity; the one that we choose here is the preservation of least upper bounds of directed sets. A *non empty* subset D of an ordered set is *directed* if any finite subset $F \subseteq D$ has a least upper bound in D. A *complete partial order* (CPO) is an ordered set whose directed subsets have least upper bounds. In fact one can restrict ourselves to totally ordered subsets (chains): an ordered set is a CPO if and only if any of its chains has a least upper bound. This result which uses Zorn lemma has been established by Iwamura [11] (see also [18]). A CPO is pointed if it has a least element, usually denoted $\perp$. A continuous map between two CPO is a map which commutes with least upper bounds of directed sets: $f(\bigvee D) = \bigvee f(D)$ for any directed set D. In particular a continuous map is monotonic: $a \leq b \Rightarrow f(a) \leq f(b)$. Since a directed set is a non empty set it may be the case that a continuous map does not satisfy $f(\perp) = \perp$; if it does it is said to be a *strict* continuous map. Now any continuous map f from a pointed CPO into itself has a least fixed point given by

$$\mu f = \bigvee_{n \in \omega} f^n(\perp)$$

Indeed the us consider the ω-chain f^ω obtained by iteration of f from the least element:

$$\perp \leq f(\perp) \leq \cdots \leq f^n(\perp) \leq \cdot$$

Its image by the continuous map f is the same chain without its first element $\perp$; and thus $f(\mu f) = \mu f$ by continuity. And if $\overline{f}$ is any fixed point of f we deduced by monotony of f and minimality of $\perp$ that every element of the chain f^ω, and therefore also its least upper bound μf, is less than $\overline{f}$. This reasoning shows that we could have alternatively considered ω-chain complete posets and ω-continuous maps.

A Σ-algebra $\mathcal{A}$ is continuous if each domain of interpretation $\mathcal{A}_s$ is a pointed CPO, and each function of interpretation $op^{\mathcal{A}}$ is a continuous function (a cartesian product of CPO is a CPO with the pointwise order relation). We readily verify that the interpretation $t^{\mathcal{A}}$ of a term $t \in T(\Sigma, X)_s$ is a continuous map as well as the transformation $E^{\mathcal{A}}$ associated with a system of equations. Thus the system of equation E admits a least solution given by $\mu E^{\mathcal{A}}$.

2.4 The initial continuous algebra

The initial continuous Σ-algebra is the continuous extension of the algebra of terms. The elements of this algebra are certain classes of trees that we now defined. If L is a set, an L-labelled tree $t : Dom(t) \to L$ is a partial function whose domain $Dom(t) \subseteq \mathbb{N}^*$ is a prefix-closed sublanguage of the free monoid generated by the set of positive integers. L-labelled trees is a CPO with the following order relation

$$t \leq u \quad \text{iff} \quad Dom(t) \subseteq Dom(u) \text{ and } \forall \pi \in Dom(t) \quad t(\pi) = u(\pi)$$

Indeed we immediately see that a set of trees T has a least upper bound if and only if all its elements are pairwise coherent, where two trees t and u are said to be coherent when

$$\forall \pi \in Dom(t) \cap Dom(u) \quad t(\pi) = u(\pi)$$

and its least upper bound is such that $Dom(\bigvee T) = \cup_{t \in T} Dom(t)$ and for all $\pi \in Dom(\bigvee T)$, $(\bigvee T)(\pi) = t(\pi)$ for any $t \in T$ such that $\pi \in Dom(t)$. In particular, any directed set of trees has a least upper bound because any two of its elements, having a least upper bound, are coherent. This CPO has a least element $\bot$, the tree with empty domain.

If $\Sigma = (\mathcal{S}, O_p)$ is a signature, a Σ-tree is a O_p-labelled tree t such that for all $\pi \in Dom(t)$, if $t(\pi) = op :: s_1 \times \cdots \times s_n \to s$ then $\sigma(t(\pi \cdot i)) = s_i$ for all $i \in \{1, \cdots, n\}$ such that $\pi \cdot i \in Dom(t)$, and $\pi \cdot i \in Dom(t)$ implies that $i \in \{1, \cdots, n\}$. A Σ-tree t is said to be completely defined if its domain is non-empty and

$$\forall \pi \in Dom(t) \quad \rho(t(\pi)) = n \quad \Rightarrow \quad (\pi \cdot i \in Dom(t) \text{ iff } 1 \leq i \leq n)$$

And it is said to be finite when its domain is a finite language.

The set of Σ-trees forms a Σ-algebra: a tree t is of sort s if it is the tree with empty domain $\bot$ or it satisfies $\sigma(t(\varepsilon)) = s$; if $op :: s_1 \times \cdots \times s_n \to s$ is an operator of Σ, and $t_i \in Tree(\Sigma)_{s_i}$ then $t = op(t_1, \cdots, t_n)$ is the Σ-tree of sort s such that $Dom(t) = \{\varepsilon\} \cup_{1 \leq i \leq n} \{i \cdot \pi | \pi \in Dom(t_i)\}$, and $t(\varepsilon) = op$, and $t(i \cdot \pi) = t_i(\pi)$. This algebra is continuous because least upper bounds, when they exists, are given by the set-theoretic union of the corresponding domains. The canonical morphism

$$j = ([Tree(\Sigma)]) : T(\Sigma) \to Tree(\Sigma)$$

is an injective map that identifies Σ-terms with the finite and completely defined Σ-trees.

The finite, but not necessarily completely defined, Σ-trees are given as follows. We consider the extended signature $\Sigma_\bot = \Sigma \cup \{\bot_s;\ s \in \mathcal{S}\}$ obtained by adding to signature Σ one constant operator $\bot_s$ of sort s for each $s \in \mathcal{S}$. Associating these constants with the tree with empty domain $\bot$, that actually belongs to $Tree(\Sigma)_s$ for any $s \in \mathcal{S}$ makes Σ-trees into a $\Sigma_\bot$-algebra. We let finite trees be identified with the images of $\Sigma_\bot$-terms by the induced canonical morphism.

An element e of a CPO is said to be finite (or compact) if for any directed set D such that $e \leq \bigvee D$ it is the case that there exists some $d \in D$ such that $e \leq d$. A CPO is *algebraic* if it has a countable set of compact elements and each element is the least upper bound of (the directed set of) the compact elements below it. It is immediate to see that the CPO of Σ-trees is algebraic with the finite trees (as defined above) as its compact elements. From this property if follows that for any continuous Σ-algebra $\mathcal{A}$ there exists a unique morphism of continuous Σ-algebras (where morphisms of continuous algebras are morphisms of algebra whose maps are *strict* continuous functions), that we shall also denote as $([\mathcal{A}]) : Tree(\Sigma) \to \mathcal{A}$. This map is defined inductively on finite trees as

$$\begin{array}{lcll} ([\mathcal{A}])_s(\bot) & = & \bot_{\mathcal{A}_s} & \text{the least element of CPO } \mathcal{A}_s \\ ([\mathcal{A}])_s(op(t_1, \cdots, t_n)) & = & op(([\mathcal{A}])_{s_1}(t_1), \ldots, ([\mathcal{A}])_{s_n}(t_n)) & \end{array}$$

And given for an arbitrary tree u as

$$([\mathcal{A}])_s(u) = \bigvee \{([\mathcal{A}])_s(t) \mid t \in T(\Sigma_\perp) \quad t \leq u\}$$

Hence the Σ-algebra of Σ-trees is the initial continuous Σ-algebra. As for terms we let $t^{\mathcal{A}} = ([\mathcal{A}])_s(t)$ denote the interpretation of a tree $t \in Tree(\Sigma)_s$ in algebra $\mathcal{A}$. These notations are not conflicting since the canonical morphism $([\mathcal{A}]) : Tree(\Sigma) \to \mathcal{A}$ is an extension of $([\mathcal{A}]) : T(\Sigma) \to \mathcal{A}$; indeed the compostion of the former with the embedding $j = ([Tree(\Sigma)]) : T(\Sigma) \to Tree(\Sigma)$ is a morphism of Σ-algebras from $T(\Sigma)$ to $\mathcal{A}$ hence coincides with the latter.

If $E : X \to T(\Sigma, X)$ is a system of equations, we let μE denote its least solution in the initial continuous algebra. This solution is in fact unique in case the system is *ideal* [6]: i.e. if for any variable x the term $E(x)$ is a not reduced to a variable. We can alternatively interpret the solution of a system of equations in a given continuous algebra or solve (by computing the least solution) the interpreted system of equations, leading to identical results:

$$\forall x \in X \qquad \left(\mu E^{\mathcal{A}}\right)(x) = (\mu E(x))^{\mathcal{A}}$$

3 Embedding a domain specific language

We show how to embed a Domain Specific Language into Haskell by considering a toy language for assembling elementary boxes that we shall use as our running example. The following is an Haskell definition of a data structure for such 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
$comp_{pos} :: Box \times Box \to Box$ 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
```

Now we need to explicit the semantical aspects attached to a box: these are methods by which information can be extracted from the abstract representation of a box. For instance we can be interested by representing a box by the list of origins of its elementary boxes, which of course depends of its own origin. Another property is the size of the box given by its height and depth. Thus a semantical domain for boxes would be an element of the following class:

```
data Size = Size {depth_, height_ :: Int} deriving Show
data Point = Point {xcoord, ycoord :: Int} deriving Show
class SemBox sem where
  list ::  sem -> Point -> [Point]
  size ::  sem -> Size
```

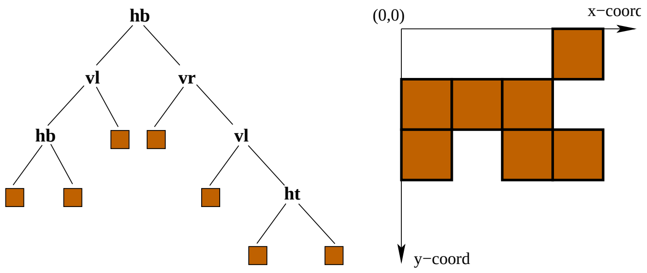
An implementation of the language of boxes is given by an algebra whose domain of interpretation for boxes is an element of the class SemBox. Of course one has to specify the computations of the attributes *size* and *list* of a given box. This can be specified with an attribute grammar, which in the case of this example provides the following algebra.

```
data SBox = SBox{list_ ::  Point -> [Point]
                ,size_ ::  Size}
instance SemBox SBox where
  list = list_
  size = size_
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
```

Using the algebra *lang* we can defined derived operators

```
ebox ::  SBox
ebox = elembox lang
hb, ht, vl, vr ::  SBox -> SBox -> SBox
hb = cmp (Hor Bottom)
ht = cmp (Hor Top)
vl = cmp (Vert Left_)
vr = cmp (Vert Right_)
cmp = comp lang
```

Figure 1: A language of boxes



that allows us to write expressions like

```
box ::  SBox
box = hb (vl (hb ebox ebox) ebox)
             (vr ebox (vl ebox (ht ebox ebox)))
```

This expression is a description of the compound box displayed in Fig. 1. The shape of this expression follows exactly the shape of the corresponding data structure of type *Box* but it is an Haskell function whose type is an element of *SBox* ; thus the expression *size box* returns the size of that box

```
size box = Size{depth_=4, height_=3}
```

and the expression *list box* (*Point* 0 0) returns the corresponding list of located elementary boxes when that box is positioned at the origin.

```
list box (Point 0 0)= [Point{xcoord=0,ycoord=1},
Point{xcoord=1,ycoord=1}, Point{xcoord=0,ycoord=2},
Point{xcoord=3,ycoord=0}, Point{xcoord=2,ycoord=1},
Point{xcoord=2,ycoord=2}, Point{xcoord=3,ycoord=2}]
```

Therefore we have interpreted some static data structure as an active object on which one may operate using the corresponding methods

$$\begin{array}{lll} \mathit{ebox} & :: & \mathit{SBox} \\ \mathit{cmp} & :: & \mathit{Pos} \longrightarrow \mathit{SBox} \longrightarrow \mathit{SBox} \longrightarrow \mathit{SBox} \\ \mathit{size} & :: & \mathit{SBox} \longrightarrow \mathit{Size} \\ \mathit{list} & :: & \mathit{SBox} \longrightarrow \mathit{Point} \longrightarrow [\mathit{Point}] \end{array}$$

(together with the derived operators: *hb*, *ht*, *vl*, and *vr*). That set of functions constitutes the interface of this embedded tiny language with its host language (Haskell). Our intention is to provide means first to specify such languages and second to compose them or make them grow.

4 Transducers

Tree transducers constitute a natural and simple way to relate abstract syntaxes of different DSLs. The notion of transducers given below is an adaptation of deterministic top-down tree transducers (defined on ranked alphabets) in the context of (sorted) signatures. Such a transducer can be viewed as a deterministic and terminating rewriting system allowing to translate terms on an input signature Σ into terms on an output signature Δ and in the converse direction it associates each Δ-algebra with a related Σ-algebra.

4.1 Transducer as a tree transformer

Definition 3 *A transducer $\mathbb{T} = (\Sigma, \Delta, Q, R)$ consists of an input signature $\Sigma = (\mathcal{S}, O_p)$, an output signature $\Delta = \left(\overline{\mathcal{S}}, \overline{O_p}\right)$, a finite set Q of states and a finite set R of rules. Each state $q \in Q$ is an unary operator with arity $\alpha(q)$ in $\mathcal{S}$ and sort $\sigma(q)$ in $\overline{\mathcal{S}}$. Thus Q is a signature on $\mathcal{S} \cup \overline{\mathcal{S}}$. The set R contains exactly one rule $r_{op,q}$ for each pair consisting of an operator $op \in O_p$ and a state q such that $\alpha(q) = \sigma(op)$. This rule is of the form*

$$q\,(op(x_1,\ldots,x_n)) \longrightarrow t[q_j(x_{i_j})/y_j]$$

where $op :: s_1 \times \cdots \times s_n \to s$, $q :: s \to \overline{s}$, $\sigma(x_i) = s_i$, $t \in T(\Delta, \{y_1,\ldots,y_m\})_{\overline{s}}$, $\sigma(y_j) = \overline{s_{i_j}}$, and $q_j :: s_{i_j} \to \overline{s_{i_j}}$; i.e. the respective left-hand side and right-hand side of rule $r_{op,q}$ are well-sorted terms on signatures $\Sigma \cup Q$ and $\Delta \cup Q$ respectively. A transducer with a unique state is called a morphism of signatures ; thus a morphism of signatures simply associates every operator of the input signature with a derived operator in the output signature.

The right-hand side of the rule associated with operator op and state q is of the form $t[q_j(x_{i_j})/y_j]$ where $t \in T(\Delta, \{y_1,\ldots,y_m\})_{\overline{s}}$; the only purpose of these formal variables y_j is to be subtituted by the expression $q_j(x_{i_j})$. We can freely rename these variables. Since the rôle of variable y_j in this rule is to select the value of state q_j attached to the $(i_j)^{th}$ operand of operator op we shall renamed it by x_{op,i_j,q_j}. More precisely we associate each operator $op :: s_1 \times \cdots \times s_n \to s$ with a set of variables

$$V_{op} = \left\{x_{op,i,q} \quad | \quad 1 \le i \le n \quad \alpha(q) = s_i\right\}$$

where $\sigma(x_{op,i,q}) = \sigma(q)$. Then if $rhs_{op,q} \in T(\Delta, V_{op})$ is the term obtained from t by renaming variable y_j by x_{op,i_j,q_j}, the rule associated with operator op and state q reads as

$$q\,(op(x_1,\ldots,x_n)) \longrightarrow rhs_{op,q}[q_j(x_{i_j})/x_{op,i_j,q_j}]$$

The rewriting system is confluent because there is no overlapping left-hand sides and it is terminating (the length of a derivation is bounded by the size of the initial term) hence for any closed term $t \in T(\Sigma)_s$ and state $q :: s \to \overline{s}$, the term $q(t)$ has a normal form $\mathbf{nf}(q,t)$ which is an element of $T(\Delta)_{\overline{s}}$. Thus each state $q :: s \to \overline{s}$ can be interpreted as a term transformer $\mathbb{T}(q) : T(\Sigma)_s \to T(\Delta)_{\overline{s}}$ where $\mathbb{T}(q)(t) = \mathbf{nf}(q,t)$. More generally one can define a transformation on non necessarily closed terms:

$$\mathbb{T}(q) : T(\Sigma, X)_{\alpha(q)} \longrightarrow T(\Delta, Q \times X)_{\sigma(q)}$$

inductively by:

$$\begin{aligned}&\mathbb{T}(q)(x) = (q,x)\\&\mathbb{T}(q)(op(t_1,\ldots,t_n)) = rhs_{op,q}[\mathbb{T}(q_j)(t_{i_j})/x_{op,i_j,q_j}]\end{aligned}$$

Notice that the variables $(q,x) \in Q \times X$ that appear in the result are such that $\alpha(q) = \sigma(x)$; and the image of a closed term $t \in T(\Sigma)_s$ is the normal form of $q(t)$, a closed term in $T(\Delta)_{\overline{s}}$.

Lemma 4 $\mathbb{T}(q)\,(t[\nu]) = \mathbb{T}(q)(t)[\mathbb{T}\nu]$ *where* $\mathbb{T}\nu(q,x) = \mathbb{T}(q)\,(\nu(x))$

Proof. If $t = x$ is a variable, then $\mathbb{T}(q)(x)[\mathbb{T}\nu] = \mathbb{T}\nu(q,x) = \mathbb{T}(q)(\nu(x)) = \mathbb{T}(q)(x[\nu])$. If $t = op(t_1,\ldots,t_n)$ then

$$\begin{array}{rcl} \mathbb{T}(q)(op(t_1,\ldots,t_n))[\mathbb{T}\nu] & = & rhs_{op,q}[\mathbb{T}(q_j)(t_{i_j})/x_{op,i_j,q_j}][\mathbb{T}\nu] \\ & = & rhs_{op,q}[\mathbb{T}(q_j)(t_{i_j})[\mathbb{T}\nu]/x_{op,i_j,q_j}] \\ & = & rhs_{op,q}[\mathbb{T}(q_j)(t_{i_j}[\nu])/x_{op,i_j,q_j}] \\ & & \text{(By inductive assumption)} \\ & = & \mathbb{T}(q)(op(t_1[\nu],\ldots,t_n[\nu])) \quad = \quad \mathbb{T}(q)(t[\nu]) \end{array}$$

□

We can extend the above constructions from terms to trees allowing thus transducers to be interpreted as tree transformers. For that purpose we consider the extended transducers $\mathbb{T}_\perp = (\Sigma_\perp, \Delta_\perp, Q, R_\perp)$ where $R_\perp$ is obtained from R by adding the rules $q(\perp_s) \longrightarrow \perp_{\bar{s}}$ where $q :: s \to \bar{s}$. Then

$$\mathbb{T}_\perp(q) : T(\Sigma_\perp, X)_{\alpha(q)} \longrightarrow T(\Delta_\perp, Q \times X)_{\sigma(q)}$$

associates any finite tree in $Tree(\Sigma + X)$ with its normal form, a finite tree in $Tree(\Delta + Q \times X)$ (we recall $T(\Sigma,X) = T(\Sigma + X)$). Then

$$\mathbb{T}(q) : Tree(\Sigma, X)_{\alpha(q)} \longrightarrow Tree(\Delta, Q \times X)_{\sigma(q)}$$

is given by

$$\mathbb{T}(q)(u) = \bigvee \left\{ \mathbb{T}_\perp(q)(t) \mid t \in T(\Sigma_\perp, X)_{\alpha(q)} \quad t \leq u \right\}$$

Thus the image of a tree is given as the least upper bound of the normal forms of its finite approximates by the rewrite system associated with the extended transducer $\mathbb{T}_\perp$.

4.2 Composition of transducers

In order to model the combined effect of two transducers $\mathbb{T}_1 = (\Sigma_1, \Sigma_2, Q_1, R_1)$ and $\mathbb{T}_2 = (\Sigma_2, \Sigma_3, Q_2, R_2)$, let us consider a term $t_1 \in T(\Sigma_1)_{s_1}$ of sort s_1. It can be transformed by $\mathbb{T}_1(q_1)$ for some state $q_1 :: s_1 \to s_2$ into a term $t_2 = \mathbb{T}_1(q_1)(t_1) \in T(\Sigma_2)$ of sort s_2. Now a transformation $\mathbb{T}_2(q_2)$ with some state $q_2 :: s_2 \to s_3$ may be applied to derive a term $t_3 = \mathbb{T}_2(q_2)(t_2) \in T(\Sigma_3)$ of sort s_3. Thus if some transducer $\mathbb{T}$ is supposed to represent the composition of $\mathbb{T}_1 = (\Sigma_1, \Sigma_2, Q_1, R_1)$ and $\mathbb{T}_2 = (\Sigma_2, \Sigma_3, Q_2, R_2)$ it set of states should be given by the pairs $(q_1, q_2) \in Q_1 \times Q_2$ such that $\sigma(q_1) = \alpha(q_2)$ with arity and sort given by $\alpha(q_1, q_2) = \alpha(q_1)$ and $\sigma(q_1, q_2) = \sigma(q_2)$. Moreover the overall transformation $\mathbb{T}(q_1, q_2) : T(\Sigma_1)_{\alpha(q_1)} \to T(\Sigma_2)_{\sigma(q_2)}$ should realize the composition of the respective term transformations:

$$\mathbb{T}(q_1, q_2)(t) = \mathbb{T}_2(q_2)\,(\mathbb{T}_1(q_1)(t))$$

Each transformation $\mathbb{T}_i(q_i)(t_i)$ consists in computing the normal form of the term $q_i(t_i)$ w.r.t. the rewriting system associated with transducer $\mathbb{T}_i$. Thus in order to characterize the term rewriting associated with the composite transducer $\mathbb{T}$ we are led to consider the normal form of term $q_2(q_1(t_1))$ with respect to the rules coming respectively from both transducers $\mathbb{T}_1$ and $\mathbb{T}_2$. Assuming t_1 of the form $t_1 = op(u_1, \ldots, u_n)$ there exists initially only one redex whose reduction leads to term $q_2\left(rhs^{\mathbb{T}_1}_{op,q_1}[q'_1(u_i)/x_{op,i,q'_1}]\right)$.

In order to compute the normal form $t_2 = \mathbb{T}_1(q_1)(t_1) \in T(\Sigma_2)$ we should proceed by reducing the innermost redexes $q'_1(u_i)$. However the composite rewriting system is confluent, since it has no overlapping left-hand sides, and we can equivalently chose to reduce the outermost redex first until we reach the normal form of $q_2\left(rhs^{\mathbb{T}_1}_{op,q_1}\right)$ with respect to $\mathbb{T}_2$, thus reaching term $\mathbb{T}_2(q_2)(rhs^{\mathbb{T}_1}_{op,q_1}[q'_2(q'_1(u_i))/(q'_2,x_{op,i,q'_1})]$ and then we are led to reduce the subterms $q'_2(q'_1(u_i))$ which have the same shape as the term we started from and thus we can apply recursively the same reduction strategy to them. An advantage is that the computation of $\mathbb{T}_2(q_2)(rhs^{\mathbb{T}_1}_{op,q_1})$ does not depend of the argument t_1 but, on the contrary, is a static information that depends only on the definition of both transducers. Computing this expression beforehand can save subsequent computations if subexpressions of the form $q_2(q_1(op(x_n,\ldots,x_n)))$ occur in many places during the reduction of $q_2(q_1(t_1))$.

Definition 5 *The composition of a two transducers* $\mathbb{T}_1 = (\Sigma_1,\Sigma_2,Q_1,R_1)$ *and* $\mathbb{T}_2 = (\Sigma_2,\Sigma_3,Q_2,R_2)$ *is the transducer* $\mathbb{T} = \mathbb{T}_2 \circ \mathbb{T}_1 = (\Sigma_1,\Sigma_3,Q,R)$ *where*

- $Q = \{(q_1,q_2) \in Q_1 \times Q_2 \mid \sigma(q_1) = \alpha(q_2)\}$ *with* $\alpha(q_1,q_2) = \alpha(q_1)$ *and* $\sigma(q_1,q_2) = \sigma(q_2)$*;*
- $rhs^{\mathbb{T}}_{op,(q_1,q_2)} = \mathbb{T}_2(q_2)(rhs^{\mathbb{T}_1}_{op,q_1})[x_{op,i,(q'_1,q'_2)}/(q'_2,x_{op,i,q'_1})]$

The following result shows that the term transformation induced by the composite transducer is indeed the composition of the transformations associated with each transducers.

Proposition 6 *For any closed term* $t \in T(\Sigma_1)$

$$\mathbb{T}(q_1,q_2)(t) = \mathbb{T}_2(q_2)\,(\mathbb{T}_1(q_1)(t))$$

Proof. We prove by induction the slightly more general statement expressing the fact that for any term $t \in T(\Sigma_1,X)$ the following holds:

$$\mathbb{T}(q_1,q_2)(t) = \mathbb{T}_2(q_2)\,(\mathbb{T}_1(q_1)(t))\,[\nu]$$

where ν is the valuation such that $\nu(q_2,(q_1,x)) = ((q_1,q_2),x)$. If t is a variable x then we have $\mathbb{T}(q_1,q_2)(x) = ((q_1,q_2),x)$ and $\mathbb{T}_2(q_2)\,(\mathbb{T}_1(q_1)(x)) = (q_2,(q_1,x))$, hence $\mathbb{T}(q_1,q_2)(x) = \mathbb{T}_2(q_2)\,(\mathbb{T}_1(q_1)(x))\,[\nu]$ as required.

For the induction step let us assume that $t = op(t_1,\ldots,t_n)$ then

$$\begin{array}{rcl}
\mathbb{T}(q_1,q_2)(t) & = & rhs^{\mathbb{T}}_{op,(q_1,q_2)}[\mathbb{T}(q'_1,q'_2)(t_i)/x_{op,i,(q'_1,q'_2)}] \\
 & = & \mathbb{T}_2(q_2)(rhs^{\mathbb{T}_1}_{op,q_1})[\mathbb{T}(q'_1,q'_2)(t_i)/(q'_2,x_{op,i,q'_1})] \\
 & = & \mathbb{T}_2(q_2)(rhs^{\mathbb{T}_1}_{op,q_1})[\mathbb{T}_2(q'_2)(\mathbb{T}_1(q'_1)(t_i))[\nu]/(q'_2,x_{op,i,q'_1})] \\
 & & \text{(By inductive assumption)} \\
 & = & \mathbb{T}_2(q_2)(rhs^{\mathbb{T}_1}_{op,q_1}[\mathbb{T}_1(q'_1)(t_i)/x_{op,i,q'_1}])[\nu] \quad \text{By Lemma 4} \\
 & = & \mathbb{T}_2(q_2)\,(\mathbb{T}_1(q_1)(t))\,[\nu]
\end{array}$$

□

4.3 Transducer as an algebra transformer

We have seen that a transducer can be interpreted as a term (or tree) transformer. In the converse direction we can associate a Δ-algebra $\mathcal{A}$ with a Σ-algebra $\mathbb{T}\mathcal{A}$ such that the interpretation of a term $t \in T(\Sigma)_s$ w.r.t. $\mathbb{T}\mathcal{A}$ is the vector indexed by the set of states of arity s such that the component associated with state q is the interpretation of the normal form of $q(t)$ w.r.t. $\mathcal{A}$; i.e.

$$\alpha(q) = \sigma(t) \quad \Rightarrow \quad t^{\mathbb{T}\mathcal{A}}(q) = (\mathbb{T}(q)(t))^{\mathcal{A}}$$

Definition 7 *the Σ-algebra $\mathcal{B} = \mathbb{T}\mathcal{A}$ derived from a Δ-algebra $\mathcal{A}$ and a transducer $\mathbb{T} = (\Sigma, \Delta, Q, R)$ is such that $\mathcal{B}_s = \prod \{ \mathcal{A}_{\sigma(q)} | q \in Q \ s.t. \ \alpha(q) = s \}$ and the interpretation of an operator $op :: s_1 \times \cdots \times s_n \to s$ is the map $op^{\mathcal{B}} = \langle op_q^{\mathcal{B}} \ ; \ q \in Q \ s.t. \ \alpha(q) = s \rangle$ where $op_q^{\mathcal{B}} : \mathcal{B}_{s_1} \times \cdots \times \mathcal{B}_{s_n} \to \mathcal{A}_{\sigma(q)}$ is given by $op_q^{\mathcal{B}}(v_1, \ldots, v_n) = rhs_{op,q}^{\mathcal{A}}[v_i(q)/x_{op,i,q}]$.*

Indeed from $op^{\mathcal{B}}(v_1, \ldots, v_n)(q) = op_q^{\mathcal{B}}(v_1, \ldots, v_n) = t^{\mathcal{A}}[v_{i_j}(q_j)/y_j]$ it follows, by induction of the structure of term t, that $t^{\mathcal{B}}(q) = (\mathbb{T}(q)(t))^{\mathcal{A}}$.

The family of maps $q^{\mathcal{A}} : T(\Sigma)_{\alpha(q)} \to \mathcal{A}_{\sigma(q)}$ given by $q^{\mathcal{A}}(t) = \mathbf{nf}(q,t)^{\mathcal{A}}$ satisfies

$$q^{\mathcal{A}}(op(t_1, \ldots, t_n)) = rhs_{op,q}^{\mathcal{A}}[q_j^{\mathcal{A}}(t_{i_j})/x_{op,i_j,q_j}] \qquad \text{for} \quad t_i \in T(\Sigma)_{s_i}$$

If we consider the set of rules of the transducer as a system of equations whose unknown are states (i.e. we interpret such a transducer as a primitive recursive scheme), then an algebra $\mathcal{A}$ is an interpretation of this primitive recursive scheme and the family of maps $q^{\mathcal{A}} : T(\Sigma)_{\alpha(q)} \to \mathcal{A}_{\sigma(q)}$ is thus a solution of this primitive recursive scheme in interpretation $\mathcal{A}$. It can be shown [3] that any primitive recursive scheme has a unique solution in all interpretations. This result follows from the following fact

Proposition 8 ([3]) *A family $\overline{q} : T(\Sigma)_{\alpha(q)} \to \mathcal{A}_{\sigma(q)}$ is a solution of the interpreted primitive recursion scheme if, and only if, the family of maps $\theta_s : T(\Sigma)_s \to \mathcal{B}_s$ given by $\theta_s(t)(q) = \overline{q}(t)$ is a morphism of Σ-algebras from the free algebra $T(\Sigma)$ to the derived algebra $\mathcal{B} = \mathbb{T}\mathcal{A}$.*

Definition 9 *A transducer $\mathbb{T}_2 = (\Delta, \Sigma, Q_2, R_2)$ is a left-inverse to transducer $\mathbb{T}_1 = (\Sigma, \Delta, Q_1, R_1)$ if for any $q :: s \to \overline{s}$ in Q_1 there exists $\overline{q} :: \overline{s} \to s$ in Q_2 such that*

$$\mathbb{T}_2(\overline{q})(rhs_{op,q}^{\mathbb{T}_1})[x_i/(\overline{q}, x_{op,i,q})] = op(x_1, \ldots, x_n)$$

for any operator op and state q such that $\alpha(q) = \sigma(op)$.

Proposition 10 *Let $\mathbb{T}_2 = (\Delta, \Sigma, Q_2, R_2)$ be a left-inverse to $\mathbb{T}_1 = (\Sigma, \Delta, Q_1, R_1)$, then $\mathbb{T}_2(\overline{q})(\mathbb{T}_1(q)(t)) = t$ for any closed term $t \in T(\Sigma)_s$ and state q such that $\alpha(q) = s$ and $\mathbb{T}_1(\mathbb{T}_2\mathcal{A}) \cong \mathcal{A}$ for any Δ-algebra $\mathcal{A}$.*

5 Simple extensions of the language of boxes

We illustrate the use of an algebra derived from a transducer by describing a simple extension of the language of boxes given in Section 3 where one would like to be able to assemble arbitrary (non empty) lists of boxes rather than to combine them only pairwise. The signature associated with this extended language is given as follows.

```
data Box2 = Elembox2 | Comp2 Pos LBox2
data LBox2 = Unit2 Box2 | Cons2 Box2 LBox2
data AlgBox2 box lbox = AlgBox2{
```

```
                            elembox2 ::  box,
                            comp2 ::  Pos -> lbox -> box,
                            unit2 ::  box -> lbox,
                            cons2 ::  box -> lbox -> lbox }
```

This signature Δ has two different sorts (Box2 and LBox2), a constant $elem2 :: Box2$ representing the elementary box and four binary operators $comp2_{pos} :: LBox2 \rightarrow Box2$ associated with the four different manners (with $pos :: Pos$) of assembling a list of sub-boxes in order to obtained a new box, and two operators $unit2 :: Box2 \rightarrow LBox2$ and $cons2 :: Box2 \rightarrow LBox2 \rightarrow LBox2$ used to form non-empty lists of boxes. We have a morphism of signature $j : \Sigma \rightarrow \Delta$ (i.e. a transducer with a unique state that we shall also denote j) given by

$$\begin{array}{lcl} j(elembox) & \rightarrow & elembox2 \\ j(comp_{pos}(x,y)) & \rightarrow & comp2_{pos}(cons2(j(x), unit2(j(y)))) \end{array}$$

which allows to translate an expression of the original language into an equivalent expression in the extended language. If $\mathcal{A}$ is a Δ-algebra, the derived algebra $j\mathcal{A}$ is such that $(j\mathcal{A})_{Box} = \mathcal{A}_{Box2}$ and

$$\begin{array}{lcl} elembox^{j\mathcal{A}} & = & elembox2^{\mathcal{A}} \\ comp^{j\mathcal{A}}_{pos}(x,y) & = & comp2^{\mathcal{A}}_{pos}(cons2^{\mathcal{A}}(x, unit2^{\mathcal{A}}(y))) \end{array}$$

An extension of language $lang$, which is an Σ-algebra, is a Δ-algebra $lang2$ such that $lang = j\ lang2$. In order to construct such an extension we define a transducer $\mathbb{T}$ in the opposite direction, i.e. from Δ to Σ. The states of this transducer are

$$\begin{array}{lcll} q & :: & Box2 \rightarrow Box & \\ p_{pos} & :: & LBox2 \rightarrow Box & \text{for } pos \in Pos \end{array}$$

with the following rules

$$\begin{array}{lcl} q(elembox2) & \rightarrow & elembox \\ q(comp2_{pos}(x)) & \rightarrow & p_{pos}(x) \\ p_{pos}(unit2(x)) & \rightarrow & q(x) \\ p_{pos}(cons2(x,y)) & \rightarrow & comp_{pos}(q(x), p_{pos}(y)) \end{array}$$

If $\mathcal{B}$ is a Σ-algebra, the derived algebra $\mathbb{T}\mathcal{B}$ is such that $(\mathbb{T}\mathcal{B})_{Box2} = \mathcal{B}_{Box}$, $(\mathbb{T}\mathcal{B})_{LBox2} = \prod_{pos \in Pos} \mathcal{B}_{Box}$, and

$$\begin{array}{lcl} elembox2^{\mathbb{T}\mathcal{B}} & = & elembox^{\mathcal{B}} \\ comp2^{\mathbb{T}\mathcal{B}}_{pos}(v) & = & v(pos) \\ unit2^{\mathbb{T}\mathcal{B}}(x) & = & \langle x \rangle_{pos \in Pos} \\ cons2^{\mathbb{T}\mathcal{B}}(x, \langle y_{pos} \rangle_{pos \in Pos}) & = & \langle comp^{\mathcal{B}}_{pos}(x, y_{pos}) \rangle_{pos \in Pos} \end{array}$$

We obtain the following Haskell code.

```
type SLBox = Pos -> SBox
lang2 ::  AlgBox2 SBox SLBox
lang2 = AlgBox2 elembox2 comp2 unit2 cons2 where
  -- elembox2 ::  SBox
  elembox2 = elembox lang
  -- comp2 ::  Pos -> SLBox -> SBox
```

```
comp2 pos lbox = lbox pos
-- unit2 :: SBox -> SLBox
unit2 box pos = box
-- cons2 :: SBox -> SLBox -> SLBox
cons2 box lbox pos = comp lang pos box (lbox pos)
```

Langage $lang2$ is an extension of $lang$, i.e. $lang = j\ lang2$, because transducer j is a right-inverse to transducer $\mathbb{T}$ as can checked by performing the following computations:

$$\begin{array}{lll} q(j(elembox)) & \rightarrow & q(elembox2) \rightarrow elembox \\ q(j(comp_{pos}(x,y))) & \rightarrow & q(comp2_{pos}(cons2(j(x),unit2(j(y))))) \\ & \rightarrow & p_{pos}(cons2(j(x),unit2(j(y)))) \\ & \rightarrow & comp_{pos}(q(j(x)),p_{pos}(unit2(j(y)))) \\ & \rightarrow & comp_{pos}(q(j(x)),q(j(y))) \end{array}$$

One can then introduce the following derived combinators providing the interface for the extended language.

```
hb*, ht*, vl*, vr* ::  [SBox] -> SBox
hb* = comp_ (Hor Bottom)
ht* = comp_ (Hor Top)
vl* = comp_ (Vert Left_)
vr* = comp_ (Vert Right_)
comp_ :: Pos -> [SBox] -> SBox
comp_ pos boxes = fold (unit2 lang2)(cons2 lang2) boxes pos
fold :: (a->b)->(a->b->b)->([a]->b)
fold unit cons = f where
  f [x] = unit x
  f (x:xs) = cons x (f xs)
```

For instance the following is an expression in this language

```
box2 = ht* [vl* [ebox, ht* [ebox,ebox],ebox],
            hb* [ebox, ht* [vl* [ebox,ebox],ebox]],
            vl* [ebox, hb* [ebox,ebox]] ]
```

It defines a box such that

```
size box2 = Size{depth_=7, height_=3}
```

and

```
list box2 (Point 0 0)= [Point{xcoord=0,ycoord=0},
Point{xcoord=0,ycoord=1}, Point{xcoord=1,ycoord=1},
Point{xcoord=0,ycoord=2}, Point{xcoord=2,ycoord=1},
Point{xcoord=3,ycoord=0}, Point{xcoord=3,ycoord=1},
Point{xcoord=4,ycoord=0}, Point{xcoord=5,ycoord=0},
Point{xcoord=5,ycoord=1}, Point{xcoord=6,ycoord=1}]
```

6 Conclusion

In this paper we have considered simple language extensions given by a pair of transducers. When these transducers are viewed as trees transformers, one of this transducer provides an embedding of the original language into its extension while the other, a left-inverse to the embedding, allows every expression of the extended language to be expanded into an expression of the original language. This is of course the easy case and the work presented here should be considered as a work in progress where we have reach only a very preliminary stage. The purpose of this report was to present our approach to the extension of domain specific languages, fix the notations and definitions and illustrate the case, of what might be termed the class of the *split extensions*, where the embedding of the language into its extension has a left-inverse. We shall now investigate the general case of language extensions where the embedding doesn't necessarily have a left-inverse.

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 R. Djeumen. Modular Grammars and Splitting of Catamorphisms. INRIA Research Report, October 2007.

[3] 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.

[4] Sergey Dmitriev. Language Oriented Programming: The Next Paradigm. http://www.onboard.jetbrains.com/articles/04/10/lop/index.html

[5] G.D.P. Dueck, G.V. Cormack. Modular Attribute Grammars. *The Computer Journal*, vol. 33, No 2, 164-172, 1990.

[6] C.C. Elgot. Monadic Computation and iterative algebraic theories. *Proc. Logic Colloquium'73*, Bristol, England, North-Holland, Amsterdam, 1975, pp. 175-230.

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

[8] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages. http://www.martinfowler.com/articles/languageWorkbench.html

[9] J. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright. Initial Algebra Semantics and Continuous Algebras. *Journal of The Association for Computing Machinery*, vol. 24, No 1, January 1977, pp. 68-95.

[10] P. Hudak. Building domain-specific embedded languages. ACM Computing Surveys, 28(4), December 1996.

[11] Iwamura. A lemma on directed sets. *Zenkoku Shijo Sugaku Danwakai*, 262: 107-111.

[12] 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.

[13] M. Jourdan, C. Le Bellec, D. Parigot, G. Roussel. Specification and Implementation of Grammar Couplings using Attribute Grammars. Proccedings of *Programming Language Implementation and Logic Programming* (PLILP'93). Vol. 714 of Springer-Verlag Lecture Notes in Computer Science, pp; 123-136, 1993.

[14] U. Kastens, W.-M. Waite. Modularity and reusability in attribute grammars. Acta Informatica, 31: 601-627, 1994.

[15] Donald E. Knuth. Semantics of context free languages. *Mathematical System Theory*, 2(2): 127-145, June 1968.

[16] C.W. Krueger. Software reuse. ACM Computing Surveys, 24(2):131-183, June 1992.

[17] D. Leijen, E. Meijer. Domain specific embedded compilers. In *Proceedings of the scond USENIX Conference on Domain-Specific Languages*. USENIX Association, pages 109-122, October 3-5, 1999.

[18] Markowsky. Chain-complete Posets and Directed Sets with Applications. *Algebra Univ.* 6: 185-200.

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

[20] Charles Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In B. Randell (Ed.) The future of Software, Proceeding of the joint International Computer Limited. University of Newcastle seminar (Also : Technical Report MSR-TR-95-52, Microsoft Research, redmond), 1995.

[21] Eric Van Wyk, Oege de Moor, Ganesh Sittampalam, Ivan Sanabria Piretti, Kevin Backhouse, Paul Kwiatkowski. Intentional Programming: A Host of Language Features. Oxford University Computing Laboratory, PRG-RR-01-21, 2001.

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