HAL
archives-ouvertes.fr

Modular Design of Domain-Specific Languages using Splittings of Catamorphisms

Eric Badouel, Rodrigue Djeumen Djatcha

To cite this version:

Eric Badouel, Rodrigue Djeumen Djatcha. Modular Design of Domain-Specific Languages using Splittings of Catamorphisms. Bernd Fischer and Tarmo Uustalu. ICTAC 2018 - 15th International Colloquium on the Theoretical Aspects of Computing, Oct 2018, Stellenbosch, South Africa. Springer, 11187, pp.62-79, 2018, LNCS. <10.1007/978-3-030-02508-3_4>. <hal-01919423>

HAL Id: hal-01919423
https://hal.inria.fr/hal-01919423

Submitted on 12 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modular Design of Domain-Specific Languages using Splittings of Catamorphisms *

Eric Badouel[1] and Rodrigue Aimé Djeumen Djatcha[2]

[1] Inria Rennes-Bretagne Atlantique, Irisa, University of Rennes I, France,
eric.badouel@inria.fr
[2] Faculty of Sciences, University of Douala, Cameroon
djeumenr@yahoo.fr

Abstract. Language oriented programming is an approach to software composition based on domain specific languages (DSL) dedicated to specific aspects of an application domain. In order to combine such languages we embed them into a host language (namely Haskell, a strongly typed higher-order lazy functional language). A DSL is then given by an algebraic type, whose operators are the constructors of abstract syntax trees. Such a multi-sorted signature is associated to a polynomial functor. An algebra for this functor tells us how to interpret the programs. Using Bekić's Theorem we define a modular decomposition of algebras that leads to a class of parametric multi-sorted signatures, associated with regular functors, allowing for the modular design of DSLs.

Keywords: Abstract Syntax Trees · Catamorphisms · Bekić's Theorem · Component-based Design · Domain Specific Languages.

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 predesigned components to be reused in larger systems [14]. Instead of constructing standalone applications the focus is on the use of libraries viewed as toolboxes for the development of software product lines dedicated to some specific application domain. Using such "components on the shelf" improves productivity in developing software as well as 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, inheritance or polymorphism) and then it can be seen as a small programming language in itself. Language oriented programming [22,5] is an approach to software composition based on domain specific languages (DSL) dedicated to specific aspects of an application domain. A DSL captures the semantics of a specific application domain by packaging reusable domain knowledge into language features. It can be used by an expert of that

* This work was partially supported by ANR Headwork.

domain who is provided with familiar notations and concepts rather that confronted with a general purpose programming language.

Many DSLs have been designed and used in the past decades, however their systematic study is a more recent concern. 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... and also related tools for the generation of documentation, the integration of graphic and text editing facilities, the synchronization of multiple partial views, etc. Language adaptivity is another concern: it is very hard to make a change to the design of a programming language. However some domains of expertise may evolve in time, calling for frequent redesigns of the associated DSL: will we have to go through the process all over again every time? Finally, it might be difficult, if not impossible, to make different DSLs collaborate within some application even though most applications do involve different domains of expertise.

To alleviate these difficulties Hudak [10] suggested embedding the DSL into a chosen general-purpose host language; and coined the expression *Domain-Specific Embedded Languages* (or DSEL) to qualify them. 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 DSELs can communicate through their common host language. A higher-order strongly-typed lazy functional language like Haskell is an ideal host language since it can be viewed as a DSL for denotational semantics: a language that can be used to describe the semantics of various programming languages and thus also to combine them.

Recent language workbenches [7] like *Intentional Programming* [16,21] or the *Meta Programming System* [5] from JetBrains envisage a system where one could systematically scope and design DSLs with the ability to compose a language for a particular problem by loading DSLs as various plug-ins. Each such plug-in would incorporate meta-programming tools allowing one to program in the corresponding DSL (browsing, navigating and editing syntax, extracting multiple views or executable code). The core of such intensional representations are abstract syntax trees associated to a multi-sorted signature whose operators are the basic contructions of the language. These operators are usually interpreted as closed higher-order functions (i.e., combinators). Following the higher-order interpretation of attribute grammars [11,6,2] we shall assume that these combinators derive from the semantic rules of an attribute grammar built on the multi-sorted signature.

Combining such DSLs requires considering a global grammar such that each DSL is associated with some subgrammar. The global grammar need not be constructed explicitly but we should be able to evaluate its abstract syntax trees by combining the catamorphisms of the corresponding subgrammars.

In this paper we address this problem by introducing the so-called modular grammars. The initial algebra of the polynomial functor associated with the operators of the language coincides with its least fixed-point. This fixed-point can be computed by a method of substitution using Bekić's Theorem [4]. By doing so the system of polynomial functors is transformed into a related system of regular functors. We introduce a splitting operation on algebras producing an algebra for the resulting system of regular functors from an algebra of the original system of polynomial functors. This transformation preserves the interpretation function (catamorphism).

2 Modular Domain Specific Languages

The syntax of a DSL is given by a multi-sorted signature $\Sigma = (S, \Omega)$ consisting of a set of sorts S and a set of operators $\omega \in \Omega$ where each operator has an *arity* in S^* and a *sort* in S. We let $\Omega(s_1 \cdots s_n, s)$ denote the set of operators $\omega \in \Omega$ with arity $s_1 \cdots s_n \in S^*$ and sort $s \in S$. Let us first assume that each sort appears as the sort of some operator. Then the signature can be associated with the endofunctor $F : |\mathbf{Set}|^S \to |\mathbf{Set}|^S$ such that

$$F(X)_s = \coprod_{\omega \in \Omega(s_1 \cdots s_n, s)} X_{s_1} \times \ldots \times X_{s_n}$$

which we may write

$$F(X)_s = \{\omega(x_1, \ldots, x_n) \mid \omega \in \Omega(s_1 \cdots s_n, s),\ (\forall 1 \leq i \leq n)\ x_i \in X_{s_i}\}$$

where $\omega(x_1, \ldots, x_n)$ is used to denote the element $(x_1, \ldots, x_n) \in X_{s_1} \times \ldots \times X_{s_n}$ that lies in the component indexed by ω. It is a polynomial functor (a sum of products) and it has a least fixed-point $F^\dagger$ made of the sorted Σ-trees. We readily show by induction that it is also the initial algebra. Hence there exists a unique morphism of F-algebra $(\![\varphi]\!)_F : F^\dagger \to A$, called a *catamorphism* associated with each F-algebra $\varphi : F(A) \to A$. Note that such an F-algebra is nothing more than a Σ-algebra, namely a carrier set A_s associated with each sort $s \in S$ together with an interpretation function $\omega^\varphi : A_{s_1} \times A_{s_n} \to A_s$ for each $\omega \in \Omega(s_1 \cdots s_n, s)$. And the catamorphism amounts to interpreting the tree in the algebra by replacing each symbol ω by its interpretation ω^φ and evaluating the resulting expression.

Sorts that are used (they appear in arities of some operator) but not defined (they do not coincide with the sort of any operator) are called the *parameters* of the signature. When parameters exists the corresponding functor is no longer an endofunctor but has the form $F : |\mathbf{Set}|^{p+n} \to |\mathbf{Set}|^n$ where we have assumed an enumeration of the sorts with parameters coming first. Since $|\mathbf{Set}|^{p+n} \cong |\mathbf{Set}|^p \times |\mathbf{Set}|^n$, functor F can be viewed as a parametric endofunctor $F : |\mathbf{Set}|^p \to (|\mathbf{Set}|^n \to |\mathbf{Set}|^n)$, and we can apply the results of the above discussion to each of the endofunctors $F\zeta$ for $\zeta \in |\mathbf{Set}|^p$. We readily verify that the fixed-point construction gives rise to a functor (the so-called *type functor* such that $F^\dagger\zeta =$

$(F\zeta)^\dagger$) and the isomorphim $F\zeta\left(F^\dagger\zeta\right) \cong F^\dagger\zeta$ is natural in ζ. We let $in_{F,\zeta}$: $F\zeta\left(F^\dagger\zeta\right) \to F^\dagger\zeta$ and $out_{F,\zeta} : F^\dagger\zeta \to F\zeta\left(F^\dagger\zeta\right)$ stand for the inverse bijections associated with this isomorphism. Again a Σ-algebra is nothing more than a map $\varphi : F\zeta\xi \to \xi$ where $\zeta \in |\mathbf{Set}|^p$ and $\xi \in |\mathbf{Set}|^n$. The catamorphism $(\!|\varphi|\!)_{F,\zeta}$: $F^\dagger\zeta \to \xi$ associated with φ and ζ is characterized by the identity:

$$(\!|\varphi|\!)_{F,\zeta} \circ in_{F,\zeta} \;=\; \varphi \circ F\zeta\,(\!|\varphi|\!)_{F,\zeta}$$

Haskell functions are however interpreted in the category $\mathcal{H} = \text{DCPO}_\perp$ of pointed dcpos and continuous functions. Thus we should replace the category of sets and functions in the above discussion by $\mathcal{H}$. However (see [15,1]) the category of pointed dcpos and continuous functions does not have coproducts and thus the above functorial interpretation of a signature does not seem to be possible. The trick used by Haskell to represent its data types is to resort to the subcategory $\mathcal{C} = \text{DCPO}_{\perp!}$ of pointed dcpos and **strict** continuous functions. Finite products in $\mathcal{C}$ are given by the cartesian products and the finite coproduct of two dcpos is their *coalesced sum* $A \oplus B$ obtained from their disjoint union by identifying their respective least elements: $\perp_{A\oplus B} = \perp_A = \perp_B$. The lifting operator $(-)_\perp$ consists in adding a new least element to a given dcpo: $A_\perp = A \uplus \{\perp\}$. Finally, we let the sum of pointed dcpos be given by $\sum_{1\leq i\leq n} A_i \;=\; (A_1)_\perp \oplus \cdots \oplus (A_n)_\perp$ or equivalently by $\sum_{1\leq i\leq n} A_i \;=\; (A_1 \uplus \cdots \uplus A_n)_\perp$. When this sum has only two operands it will be written with an infix notation: $A+B \;=\; (A \uplus B)_\perp$. However, we should pay attention to the fact that this binary operation is not associative and that the corresponding n-ary operation cannot be presented as an iterated application of the binary one: we rather have a family of operators indexed by non-negative integers. The unary sum coincides with the lifting operator and the nullary sum gives $1 = ()_\perp = \{\perp, ()\}$. With these notations the following data type definition in Haskell

```
data Tree a = Node a (Forest a)
data Forest a  = Leaf | Cons  (Tree a) (Forest a)
```

is associated with the (parametric) polynomial functor $F : \mathcal{C}^3 \to \mathcal{C}^2$ such that $F(A,T,F) \;=\; ((A \times F)_\perp, 1 + (T \times F))$. Now, by observing that $\mathcal{C}(A_\perp, B) \cong \mathcal{H}(A,B)$ we deduce that an F-algebra $\varphi : F\zeta\alpha \to \alpha$ boils down to a continuous Σ-algebra in the sense that all the carrier sets are pointed dcpos and the interpretation functions are continuous functions. Hence the constituents of an algebra can be expressed by Haskell functions as intended.

All mentioned results holds more generally for *locally continuous* functors and in particular for the class of *regular functors* which is the least family of functors from $\mathcal{C}^n$ to $\mathcal{C}^m$ that contains the projections and is closed by sum, product, composition and the formation of type functors.

In the remaining parts of this section we introduce an example that will help us to explain our approach to modularity of domain specific languages embedded in Haskell.

2.1 DSL associated with an algebra

Let us consider a toy language for assembling elementary boxes. 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 has a unique sort (Box), a constant standing for an elementary box and four binary operators associated with the various ways of assembling two sub-boxes in order to obtained a new box. The related notions of algebra and evaluation morphism can be expressed in Haskell as follows.

```
data AlgBox a = AlgBox {elembox :: a,  comp :: Pos -> a -> a -> a}
eval :: AlgBox a  -> Box -> a
eval (AlgBox elembox comp) = f where
  f Elembox = elembox
  f (Comp pos box1 box2) = comp (f box1) (f box2)
```

Now we need to make explicit the semantic aspects attached to a box: these are methods to extract useful information from a box. For instance we might be interested in representing a box by the list of origins of its elementary boxes, which of course depends on 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_ :: Double} deriving Show
data Point = Point {xcoord, ycoord :: Double} deriving Show
class SemBox a where
  list :: a -> Point -> [Point]
  size :: a -> 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*. One needs to specify the computations of the attributes *size* and *list* of a given box. For that purpose we use an attribute grammar that provides the required algebra following the higher-order functional approach to attribute grammars introduced in [11,6,2].

```
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 (\ 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 define 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
```

We can also define their extensions on non-empty lists of boxes

```
hb*, ht*, vl*, vr* :: [SBox] -> SBox
hb* = foldl hb
ht* = foldl ht
vl* = foldl vl
vr* = foldl vr
```

For instance the following expression

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

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

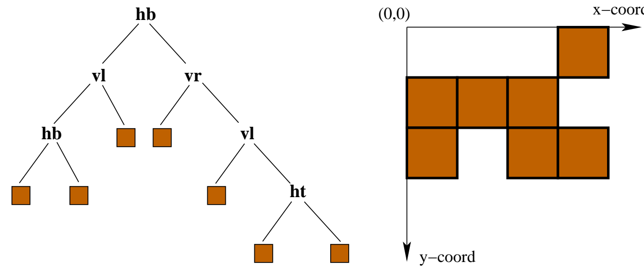

Fig. 1. A language of boxes

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

and the expression *list box* (*Point* 0 0) returns the corresponding list of located elementary boxes when the 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

$$\begin{array}{ll} ebox & :: SBox \\ cmp & :: Pos \rightarrow SBox \rightarrow SBox \rightarrow SBox \\ size & :: SBox \rightarrow Size \\ list & :: SBox \rightarrow Point \rightarrow [Point] \end{array}$$

(together with the derived operators: *hb*, *ht*, *vl*, and *vr* and their inductive extensions). That set of functions constitutes the interface of this embedded tiny language with its host language (Haskell).

Note that this language contains both the interpretation functions of the algebra (*ebox* and *cmp*) and the methods of the considered semantic domain (*size* and *list*). The description of the datatype *SBox* is not exported by the module dedicated to the language of boxes but only the functions that allows to build such boxes (*ebox* and *cmp*) or to use them (*size* and *list*).

2.2 Extension of a domain specific language

Now, imagine that we seek to extend this language to allow an elementary box to contain an image

```
data Image = Image {image :: a -> Point -> Maybe Color,
                    bb:: a -> Size}
```

represented as a function that returns the color of the point whose coordinates relative to the upper left corner of the image are given as arguments. This function returns the undefined value $Nothing$ (interpreted as "transparency") if the coordinates exceed the bounding box of the image. However, the image itself may contain some transparent parts. In addition we may wish to allow sub-boxes to be centered when composed (horizontally and vertically, see Fig. 2).

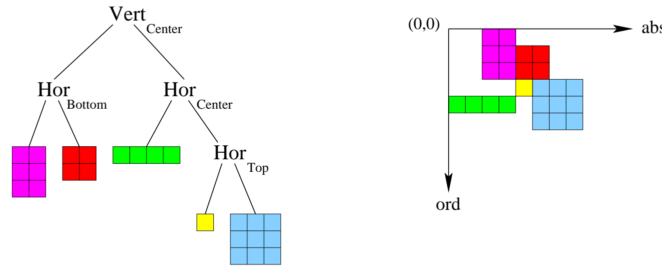

Fig. 2. A richer language of boxes

The definition of the language can be adapted as follows:

```
data Box = Elembox  {image _ :: Image}
         | Comp {pos :: Pos, first, second :: Box}
data Pos = Vert VPos | Hor  HPos
data VPos = Left_ | Center_ | Right_
data HPos = Top | Center | Bottom
class SemBox a where
  list :: a -> Point -> [(Point,Image)]
  size :: a -> Size
```

The interface of a DSL is given by its algebra. An algebra consists of the choice of a carrier set for each sort (the semantic domains of interpretation) and a function of interpretation for each operator. Note that the precise definitions of the carrier sets are not made visible. They are represented as abstract data types (given by the two functions *list* and *size* for the basic version of our example). If we want to reuse this DSL without modifying the existing code we should kept the carrier sets unchanged. We may associate new methods with the carrier sets of the algebra. But we are limited in this if the carrier sets cannot simultaneously be extended. As far as the type *SBox* is concerned, it is clear that any such function should be definable directly in terms of *list* and *size*; so these are just derived methods. Still, we may envisage adding new operators. For instance we may add the two operators vc = *Vert (Vpos Center_)* and hc = *Hor (Hpos Center)* to allow for extra ways of combining boxes. Then we should be able to extend the interpretation functions (*elembox* and *comp*) for handling these new operators while preserving the existing code. This problem has been referred to as the "expression problem" by Philip Walder:

The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the datatype, without recompiling existing code, and while retaining static type safety.

An elegant solution to this problem has been proposed by Wouter Swierstra in [18] using a method akin to an implementation of the visitor pattern. Nonetheless this method is no longer applicable if we are forced to reshape the carrier sets of the algebra, which is indeed the case for the extension considered here. We may even face more drastic changes imposed by the introduction of new operators. For instance the semantic representation of a box as a list of elementary boxes (containing an image of a given size) will not allow us to add a frame (of a given width and color) around a box using a function:

$$frame :: SBox \to Double \to Color \to SBox$$

The only reasonable choice to interpret the corresponding boxes seems to be the following:

```
class SemBox a where
  at   :: a -> Point -> Image
  size :: a -> Size
```

where *box 'at' pt* provides the image formed by anchoring the box at the given point. As in the preceding case, we have no other choice than to completely overwrite the interpretation functions *elembox* and *comp*.

3 Decomposition of catamorphisms

3.1 Modular grammar

The above example and discussion make it clear that a modular approach to DSLs requires that a basic module is dedicated to a specific set of sorts. Its interface is given by an algebra presented both by a set of interpretation functions for the operators and by methods that allow using objects of the carrier sets of the algebra. To be more precise let $\mathcal{L}$ be a language with signature $\Sigma = (S, \Omega)$ and $F : \mathcal{C}^{p+n} \to \mathcal{C}^n$ be its associated polynomial functor. Suppose that $n = n_1 + n_2$ and that the sorts S_2 corresponding to indices in n_2 are those defined by a particular module $\mathcal{L}_2$ of $\mathcal{L}$. Note that $S = S_0 \uplus S_1 \uplus S_2$ where S_0, such that $|S_0| = p$ are the parameters of the grammar and S_1, such that $|S_1| = n_1$, are the sorts defined by $\mathcal{L}$ outside the considered module. The signature of $\mathcal{L}_2$ is $\Sigma_2 = (S, \Omega_2)$ where Ω_2 is the set of operators in Ω whose sorts belong to S_2. Its associated polynomial functor is the composition of F with the second projection $\pi_2^{(n_1,n_2)} : \mathcal{C}^n \to \mathcal{C}^{n_2}$, namely $F_2 = \pi_2^{(n_1,n_2)} \circ F : \mathcal{C}^{p+n} \to \mathcal{C}^{n_2}$. Note that the parameters of Σ_2 are the elements of $S_0 \cup S_1$. Thus the sorts defined outside the module are extra parameters for this module. Of course a module would normally be given on a smaller set of sorts $S'' \subseteq S$ because it is usually defined prior to the language that uses it and we cannot anticipate all the potential usages of

a module. Nonetheless, and for ease of presentation we assume as above that $S'' = S$. Indeed any signature can be viewed as a signature over a larger set of sorts where the additional sorts play the role of extra parameters, even though the interpretation functions will not use these arguments.

In order to implement language $\mathcal{L}$, assuming that its submodule $\mathcal{L}_2$ already exists, we have to define the interpretation functions for the operators in $\Omega \setminus \Omega_2$, namely to provide an algebra for the functor $F_1 = \pi_1^{(n_1,n_2)} \circ F : \mathcal{C}^{p+n} \to \mathcal{C}^{n_1}$. The parameters of this polynomial functor are the elements of $S_0 \cup S_2$. However we should distinguish between the parameters of the overall language $\mathcal{L}$ whose carrier sets $\zeta \in |\mathcal{C}|^p$ can be arbitrarily chosen (parametric polymorphism) from the sorts of S_2 whose value should lie in $F_2^\dagger \zeta \alpha_1$ if $\alpha_1 \in |\mathcal{C}|^{n_1}$ corresponds to the carrier sets for sorts in S_1. Hence the data that is needed to reconstruct the overall language from its submodule is an algebra for the residual functor F/F_2 defined in the following categorical version of Bekić's Theorem [4].

Theorem 1. *Let a locally continuous functor $F : \mathcal{C}^{p+n} \to \mathcal{C}^n$ with $n = n_1 + n_2$ be decomposed on the form $F = \langle F_1, F_2 \rangle$ where $F_1 = \pi_1^{(n_1,n_2)} \circ F : \mathcal{C}^{p+n} \to \mathcal{C}^{n_1}$ and $F_2 = \pi_2^{(n_1,n_2)} \circ F : \mathcal{C}^{p+n} \to \mathcal{C}^{n_2}$ where functors $\pi_1^{(n_1,n_2)} : \mathcal{C}^n \to \mathcal{C}^{n_1}$ and $\pi_2^{(n_1,n_2)} : \mathcal{C}^n \to \mathcal{C}^{n_2}$ are the two canonical projections. Then*

$$F^\dagger \,\zeta \;=\; H\,\zeta \times K\,\zeta$$

where

$$\begin{array}{lll} F/F_2 & = F_1 \circ \left\langle id_{p+n_1}, F_2^\dagger \right\rangle & : \mathcal{C}^{p+n_1} \to \mathcal{C}^{n_1} \\ H & = (F/F_2)^\dagger & : \mathcal{C}^p \to \mathcal{C}^{n_1} \\ F_2' & = F_2 \circ (\langle id_p, H \rangle \times id_{n_2}) & : \mathcal{C}^{p+n_2} \to \mathcal{C}^{n_2} \\ K & = F_2'^\dagger & : \mathcal{C}^p \to \mathcal{C}^{n_2} \end{array}$$

and $id_\ell : \mathcal{C}^\ell \to \mathcal{C}^\ell$ stands for the identity functor of $\mathcal{C}^\ell$.

Bekić's Theorem corresponds to the classical method of resolution by substitution. Indeed let $\mathbf{y}$, $\mathbf{x_1}$ and $\mathbf{x_2}$ be variables ranging respectively over $|\mathcal{C}|^p$, $|\mathcal{C}|^{n_1}$ and $|\mathcal{C}|^{n_2}$. Variable $\mathbf{x_1}$ of system F becomes a parameter for its subsystem F_2. By solving the latter we obtain a parametric solution $F_2^\dagger : \mathcal{C}^{p+n_1} \to \mathcal{C}^{n_2}$. We substitute this solution for variable $\mathbf{x_2}$ in the system F_1 thus leading to a new system $F/F_2 = F_1 \circ \left\langle id_{p+n_1}, F_2^\dagger \right\rangle : \mathcal{C}^{p+n_1} \to \mathcal{C}^{n_1}$ in which variable $\mathbf{x_2}$ no longer appears. Solving this new system provides us with the $\mathbf{x_1}$ component of the solution of the original system thus given by $H = (F/F_2)^\dagger : \mathcal{C}^p \to \mathcal{C}^{n_1}$. We can substitute that value into F_2 in order to derive the system $F_2' = F_2 \circ (\langle id_p, H \rangle \times id_{n_2}) : \mathcal{C}^{p+n_2} \to \mathcal{C}^{n_2}$ whose resolution gives the $\mathbf{x_2}$ component of the solution of the original system. The following lemma says that the $\mathbf{x_2}$ component of the solution of the original system can alternatively be obtained by substituting the $\mathbf{x_1}$ component of the solution of the original system (given by H) in the parametric solution $F_2^\dagger : \mathcal{C}^{p+n_1} \to \mathcal{C}^{n_2}$. The condition expressed by this lemma appears in several axiomatizations of parametric fixed-point operators [17], and in particular in the theory of traced monoidal categories [12].

Lemma 1. $K\ \zeta\ \simeq\ F_2^{\dagger}\ \zeta\ (H\ \zeta)$

Proof. First notice that $F_2'\zeta\,(K\zeta) = F_2\zeta\,(H\zeta)\,(K\zeta)$. The initial F_2', ζ-algebra

$$in_{F_2',\zeta} : F_2\zeta\,(H\zeta)\,(K\zeta) \to K\zeta$$

is thus an F_2-algebra with parameters $\zeta \times H\zeta$. We let

$$\iota_1 = (\![in_{F_2',\zeta}]\!)_{F_2,\zeta\times H\zeta}\ :\ F_2^{\dagger}\zeta\,(H\ \zeta) \to K\zeta$$

be the corresponding catamorphism which, by definition, satisfies

$$\iota_1 \circ in_{F_2,\zeta\times H\zeta}\ =\ in_{F_2',\zeta} \circ F_2\zeta\,(H\zeta)\,\iota_1$$

Symmetrically, since $F_2\zeta(H\zeta)\left(F_2^{\dagger}\zeta\,(H\zeta)\right) = F_2'\zeta\left(F_2^{\dagger}\zeta\,(H\zeta)\right)$, we deduce that the initial $F_2, \zeta \times H\zeta$-algebra $in_{F_2,\zeta\times H\zeta}\ :\ F_2\zeta(H\zeta)\left(F_2^{\dagger}\zeta\,(H\zeta)\right) \to F_2^{\dagger}\zeta\,(H\zeta)$ is an F_2', ζ-algebra. Let $\iota_2 = (\![in_{F_2,\zeta\times H\zeta}]\!)_{F_2',\zeta}\ :\ K\zeta \to F_2^{\dagger}\zeta\,(H\zeta)$ denote the corresponding catamorphism which, by definition, satisfies $\iota_2 \circ in_{F_2',\zeta} = in_{F_2,\zeta\times H\zeta} \circ F_2\zeta(H\zeta)\iota_2$. On the one hand it follows

$$\begin{aligned}
\iota_1 \circ \iota_2 \circ in_{F_2',\zeta} &= \iota_1 \circ in_{F_2,\zeta\times H\zeta} \circ F_2\zeta(H\zeta)\iota_2\\
&= in_{F_2',\zeta} \circ F_2\zeta\,(H\zeta)\,\iota_1 \circ F_2\zeta(H\zeta)\iota_2\\
&= in_{F_2',\zeta} \circ F_2\zeta\,(H\zeta)\,(\iota_1 \circ \iota_2)\\
&= in_{F_2',\zeta} \circ F_2'\zeta\,(\iota_1 \circ \iota_2)
\end{aligned}$$

and thus $\iota_1 \circ \iota_2\ =\ (\![in_{F_2',\zeta}]\!)_{F_2',\zeta}\ =\ id_{K\zeta}$. On the other hand

$$\begin{aligned}
\iota_2 \circ \iota_1 \circ in_{F_2,\zeta\times H\zeta} &= \iota_2 \circ in_{F_2',\zeta} \circ F_2\zeta\,(H\zeta)\,\iota_1\\
&= in_{F_2,\zeta\times H\zeta} \circ F_2\zeta(H\zeta)\iota_2 \circ F_2\zeta\,(H\zeta)\,\iota_1\\
&= n_{F_2,\zeta\times H\zeta} \circ F_2\zeta(H\zeta)\,(\iota_2 \circ \iota_1)
\end{aligned}$$

and thus $\iota_2 \circ \iota_1\ =\ (\![in_{F_2,\zeta\times H\zeta}]\!)_{F_2,\zeta\times H\zeta}\ =\ id_{F^{\dagger}\zeta(H\zeta)}$. The pair of morphisms $\iota_1\ :\ F_2^{\dagger}\zeta\,(H\ \zeta) \to K\zeta$ and $\iota_2\ :\ K\zeta \to F_2^{\dagger}\zeta\,(H\zeta)$ thus constitutes the required isomorphism $K\ \zeta\ \simeq\ F_2^{\dagger}\ \zeta\ (H\ \zeta)$. □

Corollary 1. $F^{\dagger} = (F/F_2)^{\dagger} \rtimes F_2^{\dagger}$

where operation $\rtimes$ is given by

Definition 1. *The semidirect product (or cascaded composition) of functors* $H\ :\ \mathcal{C}^p \to \mathcal{C}^n$ *and* $T\ :\ \mathcal{C}^{p+n} \to \mathcal{C}^m$ *is given by*

$$H \rtimes T = \langle H, T \circ \langle id_p, H\rangle\rangle\ :\ \mathcal{C}^p \to \mathcal{C}^{n+m}$$

A module should be able to import other modules. This means that we should be able to apply a hierarchical decomposition of a signature. However, because of the presence of the type functor $F_2^{\dagger}$, we shall no longer stay within the frame of

polynomial functors. Nonetheless, if we start from polynomial functors all constructions involved in Beckić's Theorem remain in the family of regular functors. We thus model a *modular grammar* as a combination of a polynomial functor, that describes the operators whose sorts are locally defined, and a regular functor associated with the imported definitions.

Definition 2. *A modular grammar $\mathbb{G} = (F, D)$ is a pair that consists of a polynomial functor $F : \mathcal{C}^{p+n+m} \to \mathcal{C}^n$ and a regular functor $D : \mathcal{C}^{p+n} \to \mathcal{C}^m$. The signature $\Sigma = (S, \Omega)$ associated with F concretizes the sorts and operators of the grammar where $S = S_p \uplus S_d \uplus S_i$ with $|S_p| = p$, $|S_d| = n$, and $|S_i| = m$. Sorts in S_p are the parameters of $\mathbb{G}$. A sort is said to be defined (respectively imported) by $\mathbb{G}$ if it belongs to S_d (resp. S_i). The regular functor represents the imported definitions of the grammar. The functor associated with the modular grammar is the (regular) functor*

$$F_{\mathbb{G}} \; = \; F \circ \langle id_{p+n}, D \rangle \; : \; \mathcal{C}^{p+n} \to \mathcal{C}^n$$

We let $F(\mathbb{G}) = F$ and $D(\mathbb{G}) = D$ denote the respective components of modular grammar $\mathbb{G}$.

The following proposition states that the family of modular grammars is closed by the operation of decomposition of a system into a subsystem and the corresponding residual system as described in Bekić's Theorem.

Proposition 1. *Let $\mathbb{G} = (F, D)$ be a modular grammar with polynomial functor $F : \mathcal{C}^{p+n+m} \to \mathcal{C}^n$ and regular functor $D : \mathcal{C}^{p+n} \to \mathcal{C}^m$. If $n = n_1 + n_2$ then $\pi_2^{(n_1,n_2)} \circ F_{\mathbb{G}} = F_{\mathbb{G}_2}$ and $F_{\mathbb{G}/\mathbb{G}_2} = F_{\mathbb{G}}/F_{\mathbb{G}_2}$ where the second projection $\mathbb{G}_2 = \pi_2^{(n_1,n_2)}(\mathbb{G})$ of modular grammar $\mathbb{G}$ is given by*

$$\begin{array}{lcll} F(\mathbb{G}_2) & = & \pi_2^{(n_1,n_2)} \circ F(\mathbb{G}) & : \mathcal{C}^{(p+n_1)+n_2+m} \to \mathcal{C}^{n_2} \\ D(\mathbb{G}_2) & = & D(\mathbb{G}) & : \mathcal{C}^{(p+n_1)+n_2} \to \mathcal{C}^{m} \end{array}$$

and the residual operation is defined as

$$\begin{array}{lcll} F(\mathbb{G}/\mathbb{G}_2) & = & \pi_1^{(n_1,n_2)} \circ F(\mathbb{G}) & : \mathcal{C}^{p+n_1+(n_2+m)} \to \mathcal{C}^{n_1} \\ D(\mathbb{G}/\mathbb{G}_2) & = & F_{\mathbb{G}_2}^{\dagger} \rtimes D(\mathbb{G}) & : \mathcal{C}^{p+n_1} \to \mathcal{C}^{n_2+m} \end{array}$$

The situation is depicted in Figure 3 where we note that the sorts defined by the residual grammar $\mathbb{G}/\mathbb{G}_2$ (its outputs) are additional parameters for the subgrammar $\mathbb{G}_2$, whereas the outputs of $\mathbb{G}_2$ are additional imported sorts for $\mathbb{G}/\mathbb{G}_2$

Proof. The identity $\pi_2^{(n_1,n_2)} \circ F_{\mathbb{G}} = F_{\pi_2^{(n_1,n_2)}(\mathbb{G})}$ is immediate.

$$\begin{array}{l} F_{\mathbb{G}}/F_{\mathbb{G}_2} = \pi_1^{(n_1,n_2)} \circ F_{\mathbb{G}} \circ \langle id_{p+n_1}, F_{\mathbb{G}_2}^{\dagger} \rangle \\ = \pi_1^{(n_1,n_2)} \circ F(\mathbb{G}) \circ \langle id_{p+n_1+n_2}, D(\mathbb{G}) \rangle \langle id_{p+n_1}, F_{\mathbb{G}_2}^{\dagger} \rangle \end{array}$$

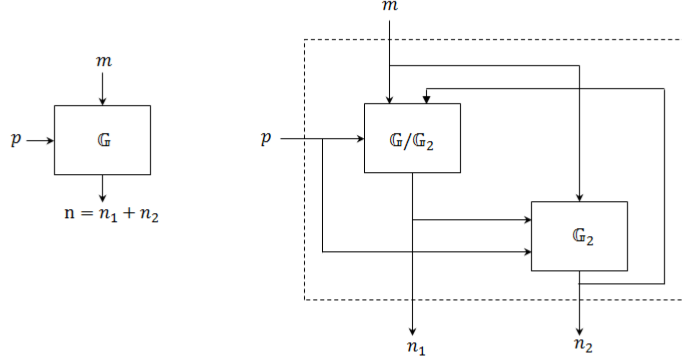

Fig. 3. Decomposition of modular grammars

and

$$\begin{aligned}&F_{\mathbb{G}/\mathbb{G}_2} = F(\mathbb{G}/\mathbb{G}_2) \circ \langle id_{p+n_1}, D(\mathbb{G}/\mathbb{G}_2)\rangle \\ &= \pi_1^{(n_1,n_2)} \circ F(\mathbb{G}) \circ \langle id_{p+n_1}, F^{\dagger}_{\mathbb{G}_2} \rtimes D(\mathbb{G})\rangle \\ &= \pi_1^{(n_1,n_2)} \circ F(\mathbb{G}) \circ \langle id_{p+n_1}, \langle F^{\dagger}_{\mathbb{G}_2}, D(\mathbb{G}) \circ \langle id_{p+n_1}, F^{\dagger}_{\mathbb{G}_2}\rangle\rangle\rangle\end{aligned}$$

In order to prove $F_{\mathbb{G}}/F_{\mathbb{G}_2} = F_{\mathbb{G}/\mathbb{G}_2}$ if suffices to show that

$$\langle id_{p+n_1+n_2}, D(\mathbb{G})\rangle\langle id_{p+n_1}, F^{\dagger}_{\mathbb{G}_2}\rangle = \langle id_{p+n_1}, \langle F^{\dagger}_{\mathbb{G}_2}, D(\mathbb{G}) \circ \langle id_{p+n_1}, F^{\dagger}_{\mathbb{G}_2}\rangle\rangle\rangle$$

These two expressions are equal because they give rise to the same results when composed with the three projections from $\mathcal{C}^{(p+n_1)+n_2+m}$ to $\mathcal{C}^{p+n_1}$, $\mathcal{C}^{n_2}$, and $\mathcal{C}^m$ respectively:

$$\begin{aligned}\pi_1^{p+n_1,n_2,m} \circ E &= id_{p+n_1} \\ \pi_2^{p+n_1,n_2,m} \circ E &= F^{\dagger}_{\mathbb{G}_2} \\ \pi_3^{p+n_1,n_2,m} \circ E &= D(\mathbb{G}) \circ \langle id_{p+n_1}, F^{\dagger}_{\mathbb{G}_2}\rangle\end{aligned}$$

□

By Corollary 1 it follows that

Corollary 2. $F^{\dagger}_{\mathbb{G}} = \left(F^{\dagger}_{\mathbb{G}/\mathbb{G}_2}\right) \rtimes F^{\dagger}_{\mathbb{G}_2}$

3.2 Decomposition of algebras

Using Bekić's Theorem we now define a decomposition of algebras.

Definition 3. *We let $F : \mathcal{C}^{p+n} \to \mathcal{C}^n$ be a locally continuous functor with $n = n_1 + n_2$. Let moreover $\Phi : F\zeta\alpha_1\alpha_2 \to \alpha_1 \times \alpha_2$ be an $F\zeta$-algebra ($\zeta \in |\mathcal{C}|^p$) on the domain $\alpha = \alpha_1 \times \alpha_2$ ($\alpha_1 \in |\mathcal{C}|^{n_1}$, and $\alpha_2 \in |\mathcal{C}|^{n_2}$). Φ can be decomposed into*

$$\begin{aligned}\varphi_1 &= \pi_1^{(n_1,n_2)}(\Phi) : F_1\ \zeta\ \alpha_1\ \alpha_2 \to \alpha_1 \\ \varphi_2 &= \pi_2^{(n_1,n_2)}(\Phi) : F_2\ \zeta\ \alpha_1\ \alpha_2 \to \alpha_2\end{aligned}$$

The (n_1, n_2)*-splitting of* Φ *is the pair consisting of the* $(F/F_2)\,\zeta$*-algebra of domain* α_1

$$\pi_{F/F_2}\Phi \triangleq \varphi_1 \circ \left(F_1\ \zeta\ \alpha_1\ \ (\!|\varphi_2|\!)_{F_2,\zeta\times\alpha_1}\right) :\ F_1\ \zeta\ \alpha_1\ \left(F_2^\dagger\ \zeta\ \alpha_1\right) \to \alpha_1$$

together with the $F_2\,(\zeta \times \alpha_1)$*-algebra of domain* α_2

$$\pi_{F_2}\Phi \triangleq \varphi_2 : F_2\ \zeta\ \alpha_1\ \alpha_2 \to \alpha_2$$

The operation of decomposition of algebras is thus given as:

$$\begin{array}{l} Split^{(n.m)}\ :\ Alg_{F,\zeta}(\alpha_1 \times \alpha_2) \to \left(Alg_{F/F_2,\zeta}(\alpha_1)\right) \times (Alg_{F_2,\zeta\times\alpha_1}(\alpha_2)) \\ Split^{(n_1,n_2)}\ \Phi\ =\ \left(\pi_{F/F_2}\Phi, \pi_{F_2}\Phi\right) \end{array}$$

Thus an algebra $\Phi = \varphi_1 \times \varphi_2 : F\zeta\alpha_1\alpha_2 \to \alpha_1 \times \alpha_2$ is decomposed into an algebra $\pi_{F_2}\Phi = \varphi_2 : F_2\ \zeta\ \alpha_1\ \alpha_2 \to \alpha_2$ for the "subsystem" F_2 together with an algebra $\pi_{F/F_2}\Phi : F/F_2\zeta\alpha_1 \to \alpha_1$ for the "residual system" F/F_2. The following result shows that the catamorphism (evaluation function) associated with the algebra Φ for the overall system can be reconstructed from the catamorphisms associated respectively with $\pi_{F_2}\Phi$ and $\pi_{F/F_2}\Phi$ using a semidirect product operation which we first introduce.

In Definition 1 we defined the semidirect product of two functors $H\ :\ \mathcal{C}^p \to \mathcal{C}^n$ and $T\ :\ \mathcal{C}^{p+n} \to \mathcal{C}^m$ as

$$H \rtimes T = \langle H, T \circ \langle id_p, H\rangle\rangle\ :\ \mathcal{C}^p \to \mathcal{C}^{n+m}$$

By functoriality of the product and composition we deduce a related operation of semidirect product of natural transformations $\eta : H \dot{\to} H'$ and $\tau : T \dot{\to} T'$ where $H, H'\ :\ \mathcal{C}^p \to \mathcal{C}^n$ and $T, T'\ :\ \mathcal{C}^{p+n} \to \mathcal{C}^m$ given by

$$(\eta \rtimes \tau)_\zeta = \eta_\zeta \times (\tau_{\zeta,H'\zeta} \circ T\zeta\eta_\zeta) = \eta_\zeta \times (T'\zeta\eta_\zeta \circ \tau_{\zeta,H\zeta})$$

Considering the special case where the target functors H' and T' are constant functors leads us to the following definition

Definition 4. *The semidirect composition of two maps* $f : H\zeta \to \alpha$ *and* $g : T\zeta\alpha \to \beta$ *where* $H\ :\ \mathcal{C}^p \to \mathcal{C}^n$ *and* $T\ :\ \mathcal{C}^{p+n} \to \mathcal{C}^m$ *is the map* $f \rtimes g : (H \rtimes T)\,\zeta \to \alpha \times \beta$ *given by* $(f \rtimes g) = f \times (g \circ T\zeta f)$*.*

Using this operation we can now state

Theorem 2. *Up to the isomorphisms* $F^\dagger\zeta = H\zeta \times K\zeta$ *and* $K\zeta = F_2^\dagger\zeta\,(H\zeta)$ *one has*

$$(\!|\Phi|\!)_{F,\zeta} = \left(\left[\pi_{F/F_2}\Phi\right]\right)_{F/F_2,\zeta} \rtimes (\!|\pi_{F_2}\Phi|\!)_{F_2,\zeta\times\alpha_1}$$

Lemma 2. *Up to the isomorphism* $F^\dagger\zeta = H\zeta \times K\zeta$ *the initial algebra* $in_{F,\zeta} : F\zeta\left(F^\dagger\zeta\right) \to F^\dagger\zeta$ *decomposes to the form* $in_{F,\zeta} = in_{H,\zeta} \times in_{K,\zeta}$ *where* $in_{H,\zeta} : F_1\zeta(H\zeta)(K\zeta) \to H\zeta$ *and* $in_{K,\zeta} : F_2\zeta(H\zeta)(K\zeta) \to K\zeta$ *are respectively given by* $in_{H,\zeta} = in_{F/F_2,\zeta} \circ (F_1\zeta(H\zeta)\iota_2)$ *and* $in_{K,\zeta} = in_{F_2',\zeta}$*.*

Proof. The initial algebra is an isomorphism and the converse also holds true (any algebra which is an isomorphism is initial) when we have uniqueness of fixed-point (up to isomorphism) which is indeed the case here.

$$in_{H,\zeta} = in_{F/F_2,\zeta} \circ (F_1\zeta(H\zeta)\iota_2) : F_1\zeta(H\zeta)(K\zeta) \to H\zeta$$

and $in_{K,\zeta} = in_{F_2',\zeta} : F_2\zeta(H\zeta)(K\zeta) \to K\zeta$ are isomorphisms and thus

$$in_{H,\zeta} \times in_{K,\zeta} : F\zeta(H\zeta)(K\zeta) \to H\zeta \times K\zeta$$

is the initial algebra of functor F. □

Corollary 3. *Up to the isomorphism $F^\dagger\zeta = H\zeta \times K\zeta$, the two parts $f : H\zeta \to \alpha_1$ and $g : K\zeta \to \alpha_2$ of catamorphism $(\![\Phi]\!)_{F,\zeta} = f \times g$ are characterized by $f \circ in_{H,\zeta} = \varphi_1 \circ F\zeta fg$ and $g \circ in_{K,\zeta} = \varphi_2 \circ F_2\zeta fg$.*

Lemma 3. *For any morphism $f : H\zeta \to \alpha_1$ one has*

$$(\![\varphi_2 \circ F_2\zeta f\alpha_2]\!)_{F_2',\zeta} = (\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ \left(F_2^\dagger\zeta f\right) \circ \iota_2 \ : \ K\zeta \to \alpha_2$$

and that morphism $g(f)$ satisfies $g(f) \circ in_{K,\zeta} = \varphi_2 \circ (F_2\,\zeta\,f\,g(f))$.

Proof. By definition $F_2^\dagger\zeta f = \left(\!\left[in_{F_2,\zeta\times\alpha_1} \circ \left(F_2\,\zeta\,f\,\left(F_2^\dagger\zeta\alpha_1\right)\right)\right]\!\right)_{F_2,\zeta\times H\zeta}$ and that morphism satisfies

$$F_2^\dagger\zeta f \circ in_{F_2,\zeta\times H\zeta} = in_{F_2,\zeta\times\alpha_1} \circ \left(F_2\,\zeta\,f\,\left(F_2^\dagger\zeta\alpha_1\right)\right) \circ F_2\zeta\,(H\zeta)\left(F_2^\dagger\zeta f\right)$$

It follows that

$$\begin{aligned}
&(\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ \left(F_2^\dagger\zeta f\right) \circ \iota_2 \circ in_{F_2',\zeta}\\
&= (\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ \left(F_2^\dagger\zeta f\right) \circ in_{F_2,\zeta\times H\zeta} \circ F_2\zeta(H\zeta)\iota_2\\
&= (\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ in_{F_2,\zeta\times\alpha_1} \circ \left(F_2\,\zeta\,f\,\left(F_2^\dagger\zeta\alpha_1\right)\right) \circ F_2\zeta\,(H\zeta)\left(F_2^\dagger\zeta f\right) \circ\\
&\quad F_2\zeta(H\zeta)\iota_2\\
&= \varphi_2 \circ F_2\zeta\alpha_1\,(\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ \left(F_2\,\zeta\,f\,\left(F_2^\dagger\zeta\alpha_1\right)\right) \circ F_2\zeta\,(H\zeta)\left(F_2^\dagger\zeta f \circ \iota_2\right)\\
&= \varphi_2 \circ F_2\zeta f\alpha_2 \circ F_2\zeta\,(H\zeta)\,(\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ F_2\zeta\,(H\zeta)\left(F_2^\dagger\zeta f \circ \iota_2\right)\\
&= (\varphi_2 \circ F_2\zeta f\alpha_2) \circ F_2\zeta\,(H\zeta)\left((\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ F_2^\dagger\zeta f \circ \iota_2\right)
\end{aligned}$$

and thus $(\![\varphi_2 \circ F_2\zeta f\alpha_2]\!)_{F_2',\zeta} = (\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ \left(F_2^\dagger\zeta f\right) \circ \iota_2$. If we let $g(f) \triangleq (\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1} \circ \left(F_2^\dagger\zeta f\right) \circ \iota_2$ denote this morphism, we deduce $g(f) \circ in_{K,\zeta} = \varphi_2 \circ F_2\zeta f\alpha_2 \circ F_2\zeta\,(H\zeta)\,g(f) = \varphi_2 \circ F_2\,\zeta\,f\,g(f)$ because $in_{K,\zeta} = in_{F_2',\zeta}$. □

Lemma 4. *If $f : H\zeta \to \alpha_1$ and $g : K\zeta \to \alpha_2$ are, up the isomorphism $F^\dagger\zeta = H\zeta \times K\zeta$, the two parts of catamorphism $(\![\Phi]\!)_{F,\zeta} = f \times g$ then $f = \left(\!\left[\varphi_1 \circ F_1\zeta\alpha_1\,(\![\varphi_2]\!)_{F_2,\zeta\times\alpha_1}\right]\!\right)_{F/F_2,\zeta}$ and $g = (\![\varphi_2 \circ F_2\zeta f\alpha_2]\!)_{F_2',\zeta}$.*

Proof. By Corollary 3 the two parts $f : H\zeta \to \alpha_1$ and $g : K\zeta \to \alpha_2$ of the catamorphism $(\!|\Phi|\!)_{F,\zeta} = f \times g$ are characterized by $f \circ in_{H,\zeta} = \varphi_1 \circ F\zeta f g$ and $g \circ in_{K,\zeta} = \varphi_2 \circ F_2\zeta f g$. Set $f' = \left(\!\left[\varphi_1 \circ F_1\zeta\alpha_1 (\!|\varphi_2|\!)_{F_2,\zeta\times\alpha_1}\right]\!\right)_{F/F_2,\zeta}$ and $g' = g(f') = (\!|\varphi_2 \circ F_2\zeta f'\alpha_2|\!)_{F_2',\zeta}$. By the preceding lemma $g' \circ in_{K,\zeta} = \varphi_2 \circ F_2\,\zeta\, f'\, g'$, moreover

$$
\begin{aligned}
& f' \circ in_{H,\zeta} \\
&= f' \circ in_{F/F_2,\zeta} \circ F_1\zeta(H\zeta)\iota_2 \\
&= \varphi_1 \circ F_1\zeta\alpha_1 (\!|\varphi_2|\!)_{F_2,\zeta\times\alpha_1} \circ F_1\zeta f' \left(F_2^\dagger\zeta f'\right) \circ F_1\zeta(H\zeta)\iota_2 \\
&= \varphi_1 \circ F_1\zeta\alpha_1 (\!|\varphi_2|\!)_{F_2,\zeta\times\alpha_1} \circ F_1\zeta\alpha_1 \left(F_2^\dagger\zeta f'\right) \circ F_1\zeta f' \left(F_2^\dagger\zeta(H\zeta)\right) \circ \\
&\quad F_1\zeta(H\zeta)\iota_2 \\
&= \varphi_1 \circ F_1\zeta\alpha_1 (\!|\varphi_2|\!)_{F_2,\zeta\times\alpha_1} \circ F_1\zeta\alpha_1 \left(F_2^\dagger\zeta f'\right) \circ F_1\zeta\alpha_1\iota_2 \circ F_1\zeta f'(K\zeta) \\
&= \varphi_1 \circ F_1\zeta\alpha_1 \left((\!|\varphi_2|\!)_{F_2,\zeta\times\alpha_1} \circ F_2^\dagger\zeta f' \circ \iota_2\right) \circ F_1\zeta f'(K\zeta) \\
&= \varphi_1 \circ F_1\zeta\alpha_1 g' \circ F_1\zeta f'(K\zeta) \\
&= \varphi_1 \circ F_1\zeta f' g'
\end{aligned}
$$

From which it follows that $f' = f$ and $g' = g$. □

Theorem 2 follows from Lemma 3 and Lemma 4.

4 Conclusion

In this paper we relied on a modular decomposition of a (multi-sorted) signature based on a hiearchical decomposition of its set of sorts in order to reconstruct a language, specified by an algebra, by composition of the algebras associated with its sublanguages. As mentioned in the introduction the global laguage would normally be left implicit. Our result represents it as a cascaded composition of its constituent sublanguages. This representation preserves catamorphisms. One can then adopt an incremental approach consisting of growing a DSL by an operation of composition of modular grammars derived from Bekić's Theorem. This approach differs from the solution of the "expression problem" proposed by Swierstra in [18] which allows adding new operators for a fixed sort (or a fixed set of sorts) and thus stays confined to a given module in our context.

We intend to apply the work presented in this paper to Guarded Attribute Grammars [3]. It is a declarative model that describes the different ways of performing a task by recursively decomposing it into more elementary subtasks. This is formalized by the productions of an abstract context-free grammar (i.e. a multi-sorted signature). The actual way a task is decomposed depends on the choices made by the person to whom the task is assigned and on the data attached to the task (inherited attributes whose values are refined over time). Productions of the grammar are associated with guards that filter the rules applicable in a given configuration. The evaluation of these guards is done incrementally which means that a rule is allowed as soon as its guard is satisfied.

This allows the workspaces of different users to operate concurrently and in reactive mode. The local grammar of a user specifies how he can behave in order to solve the pending tasks in his workspace. It defines a DSL that captures the user's domain of expertise (his role). The lazy composition of roles is compatible with the choice of Haskell as host language. Still, it remains to take side effects into account, in particular for modelling user interactions. We might use the approach proposed in [18] to represent the set of involved input-output actions as a datatype in order to isolate the input-output side effects from the hiearchical description of the system that would be specified, using the method presented in this paper, with ordinary Haskell functions (without side effects).

As we have seen above, the splitting of algebras is an approach to modular attribute grammars. This approach is orthogonal to, and thus can be combined with, alternative approaches of modularity in attribute grammars [13] such as the descriptional composition [8,9] or the composition by aspects [19,20].

Acknowledgement

We are very grateful to the reviewers for the relevance of their comments which greatly helped us to improve the presentation of this work.

References

1. Abramsky, S., Jung, A.: In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science (Vol. 3), chap. Domain Theory, pp. 1–168. Oxford University Press, Oxford, UK (1994).
2. Backhouse, K.: A functional semantics of attribute grammars. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS. pp. 142–157 (2002).
3. Badouel, E., Hélouët, L., Morvan, C., Kouamou, G., Nsaibirni, R.F.J.: Active workspaces: Distributed collaborative systems based on guarded attribute grammars. SIGAPP Applied Computing Review **15**(3), 6–34 (2015). https://doi.org/10.1145/2695664.2695698
4. Bekic, H.: Definable operation in general algebras, and the theory of automata and flowcharts. In: Jones, C.B. (ed.) Programming Languages and Their Definition - Hans Bekic (1936-1982). Lecture Notes in Computer Science, vol. 177, pp. 30–55. Springer (1984). https://doi.org/10.1007/BFb0048939
5. Dmitriev, S.: Language oriented programming: The next paradigm, `http://www.onboard.jetbrains.com/articles/04/10/lop/index.html`
6. Fokkinga, M.M., Jeuring, J., Meertens, L., Meijer, E.: A Translation from Attribute Grammars to Catamorphisms (March 1991), appeared in: The Squigollist, Vol 2, Nr 1, 1991, pages 20–26
7. Fowler, M.: Language workbenches: The killer-app for domain specific languages, `http://www.martinfowler.com/articles/languageWorkbench.html`
8. Ganzinger, H., Giegerich, R.: Attribute coupled grammars. In: Deusen, M.S.V., Graham, S.L. (eds.) SIGPLAN Symposium on Compiler Construction. pp. 157–170. ACM (1984).
https://doi.org/10.1145/502874.502890

9. Giegerich, R.: Composition and evaluation of attribute coupled grammars. Acta Informatica **25**(4), 355–423 (1988). https://doi.org/10.1007/BF02737108
10. Hudak, P.: Building domain-specific embedded languages. ACM Computing Survey **28**(4) (1996). https://doi.org/10.1145/242224.242477, article No. 196
11. Johnsson, T.: Attribute grammars as functional programming paradigm. In: Kahn, G. (ed.) Functional Programming and Computer Architecture, FPCA'87. Lecture Notes in Computer Science, vol. 274, pp. 154–173 (1987)
12. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Mathematical Proceedings of the Cambridge Philosophical Society **119**(3), 447468 (1996). https://doi.org/10.1017/S0305004100074338
13. Kastens, U., Waite, W.M.: Modularity and reusability in attribute grammars. Acta Informatica **31**(7), 601–627 (July 1994). https://doi.org/10.1007/BF01177548
14. Krueger, C.W.: Software reuse. ACM Computing Surveys **24**(2), 131–183 (June 1992). https://doi.org/10.1145/130844.130856
15. Plotkin, G.: Post-graduate lectures notes in advanced domain theory (1981), (incorporating the "Pisa Notes"). Dept. of Computer Science, Univ. of Edinburg
16. Simonyi, C.: The death of computer languages, the birth of intentional programming (1995), 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)
17. Simpson, A.K., Plotkin, G.D.: Complete axioms for categorical fixed-point operators. In: IEEE Symposium on Logic in Computer Science. pp. 30–41 (2000). https://doi.org/10.1109/LICS.2000.855753
18. Swierstra, W.: Data types à la carte. Journal of Functional Programming **18**(4), 423–436 (2008). https://doi.org/10.1017/S0956796808006758.
19. Van Wyk, E.: Aspects as modular language extensions. Electronic Notes in Theoretical Computer Science **82**(3), 555–574 (2003). https://doi.org/10.1016/S1571-0661(05)82628-3
20. Van Wyk, E.: Implementing aspect-oriented programming constructs as modular language extensions. Science of Computer Programming **68**(1), 38–61 (2007). https://doi.org/10.1016/j.scico.2005.06.006
21. Van Wyk, E., de Moor, O., Sittampalam, G., Piretti, I.S., Backhouse, K., Kwiatkowski, P.: Intensional programming: A host of language features. Tech. Rep. PRG-RR-01-21, Oxford University Computing Laboratory (2001)
22. Ward, M.P.: Language-oriented programming. Software - Concepts and Tools **15**(4), 147–161 (1994)