A Generic Framework for Genericity

Loïc CORRENSON, Etienne DURIS,
Didier PARIGOT and Gilles ROUSSEL

INRIA Rocquencourt - France.
Gilles Roussel is with Université de Marne-la-Vallée.

Abstract

Recently, generic programming becomes of a major interest in several programming paradigms. A recurrent idea to achieve genericity is to specify algorithms on their convenient data structure, and to allow these specifications to be instantiated onto a large number of neighboring data structures.

Polytypic programming, shapely types and generic attribute grammars are generic programming methods related to this approach. A framework for generic programming is proposed to embed these methods. It consists in tools for automatic generation of morphisms between data structures, and for program composition.

Thanks to this compositional approach, the complete specialization of generic programs could be advantageously delegated to a general and powerful mechanism of "symbolic composition", which performs deforestation and partial evaluation.

1 Introduction

In several programming paradigms, generic programming is being emerging. Although this concept is not new, genericity currently raises a great interest in the programming and software engineering community. In this area, one of the recent issues is genericity according to the data structure. When an algorithm is specified on a general data structure, the notion of genericity appears with the possibility to reuse it in several contexts, says, onto particular data structures.

A few years ago, Farrow *et al.* [7] devised a generic programming notion for attribute grammars [10, 14]. This genericity is based on the observation that any function f defined by an attribute grammar on a type τ_2 could be instantiated on a type τ_1. The only component needed, if it exists, is a function m that implements a morphism between terms of type τ_1 and terms of type τ_2. The composition of this function m with the function f performs the instantiation process. This approach has been revisited [11, 15, 2] to enable automatic generation of the morphism m from a simple specification of *correspondence relation* between types. In this context, an efficient specialization process consists in applying a transformation that eliminates intermediate data structures occurring in the composition. For attribute grammars, Ganzinger and Giegerich solved this *general* problem by introducing their *descriptional composition* algorithm [8, 1]. Using this technique, the construction of the intermediate representation of type τ_2 is discarded from the composition, and thus the original function f is transposed and actually specialized onto the type τ_1.

This problem of intermediate data structure elimination, usually called *deforestation* [17], has been widely studied in the functional programming community [12, 16, 13]. The comparison between functional and attribute grammar deforestation methods [6, 4] led us to propose a powerful deforestation method for functional programs, the *symbolic composition* [5][1], which is based on descriptional composition.

Since descriptional composition is an important component of the attribute grammar genericity framework, we have compared in [3] attribute grammar genericity with polytypic programming [9]. These methods seems to be complementary, and we propose in this article a general framework for genericity. This framework is based on function composition, automatic morphism generation and a general deforestation method.

The major benefit for this approach is to really separate the application of a generic algorithm from the specialization of this instantiation. The first problem is solved by morphism and composition, while the second is performed by a general and powerful deforestation method. Automatic morphism generation is still an open problem, even if methods already exist.

The paper is organized as follows. Section 2 presents the general framework for genericity through the well-known example of unification algorithm. Section 3 presents two different methods to automatically generate morphisms, while section 4 illustrates the deforestation power for instantiation and specialization purposes.

2 Framework for Genericity

Let us consider the general unification algorithm to illustrate our framework. Recall that it consists in comparing two terms t and t' that contains variables and in computing a substitution σ verifying $(\sigma\ t) = (\sigma\ t')$, if it exists. Although this problem is general, a particular implementation of the general algorithm is required for each type of the terms t and t'.

Submitted to the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP '98).

[1] This method deforests some functional programs for which existing functional methods fail.

To achieve genericity, we propose to specify it on the following well-suited type *term*:

$$\texttt{type}\ term\ c\ a\ =\ var\ a \mid const\ c\ (list\ (term\ c\ a))$$

Type c allows to distinguish the different constructors in the term algebra, while type a allows to distinguish variables. Then, implementing the unification algorithm on this type *term* is natural and it corresponds to classical algorithmic presentations. The advantage of this approach is to ease the implementation and the readability of the generic specification. Indeed, everyone can understand the unification algorithm specified on this type.

Let *unify* be the unification function. In order to use unification in practice, this unification function has to be composed with a morphism which instantiates the algorithm on a particular type. Suppose that unification is needed on a given type τ. This implies that some morphism M from type τ to type *term* exists and can be implemented in a function m. Since the unification gives a substitution which associates values of type *term* to variables, the inverse morphism m^{-1} is needed to translate back the result. The unification on τ is instantiated by:

$$unify\_\tau\ t\ t' = \texttt{let}\ \sigma = unify\ (m\ t)\ (m\ t')\ \texttt{in}\ \lambda x.(m^{-1}\ (\sigma\ x))$$

Then a classical deforestation method can be applied. After the deforestation of function $unify\_\tau$, the intermediate values of type *term* are no longer constructed and the functions m and m^{-1} are no longer used. Let g be the function *unify* deforested with m and m^{-1}, the function $unify\_\tau$ becomes:

$$unify\_\tau\ t\ t' = \texttt{let}\ \sigma' = g\ t\ t'\ \texttt{in}\ \lambda x.(\sigma'\ x)$$

Another approach to this unification problem could be found in polytypic programming with `PolyP` [9].

In the example above, two parts appear. The first one is a coupling process that produces a morphism between two types. This morphism is a function that translates a value into an intermediate representation; it could be composed with any program working on this intermediate representation. The second part applies a deforestation method in order to specialize such compositions into an instantiated algorithm. Consequently, the compositional framework for genericity is based on the three following key points:

Generic programming is achieved by functions composition. These functions are divided into two kinds. These of the first kind (e.g. *unify*) implement generic algorithms on a convenient type. Functions of the second kind (e.g. m and m^{-1}) implement translations between neighboring types.

Morphism specification provides the actual genericity. Since it describes the translation between two types, it also gives the way to perform the algorithm instantiation. Most of morphisms involved in the compositions could be automatically derived, thanks to small specification in simple meta-languages. These meta-languages should *not* be super-languages of the original programming language used, in order to be reusable and quickly developed. Nevertheless, these morphisms could also be hand-written. Moreover, succesive morphisms can be composed.

Specialization process performs the actual instantiation. The specified compositions have to be improved. In fact, each generic function has to be customized with respect to its composition context. Rather than applying an *ad-hoc* method for each particular generic system, it is worthwhile to use a *general* deforestation or fusion method, that symbolically performs at one and the same time composition, elimination of intermediate values and partial evaluation.

Let M be the morphism specification, $\mathcal{C}$ the morphism generation algorithm, *alg* the "generic" algorithm, m the morphism function generated, alg_M the expected instanciated algorithm, and Γ the programming environment. Then the framework for genericity is abstracted by the following figure:

$$\frac{M, \Gamma \overset{\mathcal{C}}{\Rightarrow} m}{alg, M, \Gamma \Rightarrow alg \circ m} \qquad \frac{alg \circ m \overset{deforestation}{\Rightarrow} alg_M}{alg \circ m \Rightarrow alg_M}$$

where $\circ$ is the standard composition in the original language, and *deforestation* is a method like HYLO [13], or Symbolic Composition [5].

3 Morphism Generation

In this section, two case studies of automatic morphism generation are presented. The first one was inspired by polytypic programming [9], but has been totally revisited to match with our generic framework. The second case study was inspired by attribute grammar genericity [11, 15, 2], and is an illustration of cross-fertilization from different paradigms. For clarity, technical details and algorithm sketches are presented in annex A.

3.1 Compositional approach of polytypic programming

Classically, in functional languages, functions are specified for a single given (polymorphic) type. However, most functions could be abstracted from any type. For example, the number of leaves in a tree and the length of a list are specified by very similar functions.

To implement such generic functions, it is necessary to specify them on a type that can represent the structure of any value of any (polymorphic) type. We then propose the following type *Poly*:

$$\begin{array}{lcl}\texttt{type}\ Poly\ c\ o\ a & = & Sum\ c\ (Poly\ c\ o\ a) \\ & \mid & Prod\ (Poly\ c\ o\ a)\ (Poly\ c\ o\ a) \\ & \mid & Par\ a \\ & \mid & Obj\ o\end{array}$$

The type *Poly* is parameterized with three type variables. Let τ be a given type. Intuitively, c is a type that identify the constructors of τ, o is a type that identify other types that could appear in τ (e.g. boolean, integer, etc.), and a is the polymorphic variable of type τ. Thus, the classical type *list*

$$\texttt{type}\ list\ a = nil \mid cons\ a\ (list\ a)$$

can be represented by the type ($Poly\ c\_list\ o\_list\ a$) where:

$$\texttt{type}\ c\_list = list\_cons \mid list\_nil$$
$$\texttt{type}\ o\_list = list\_empty$$

An interesting point is that type *Poly* is defined in the *original* functional language and does not require any special notation. The translation from type *list* to type *Poly* (resp. the backward translation) is performed by the function *out_list* (resp. *inn_list*):

```
out_list x = match x with
  cons a b → (Sum list_cons (Prod (Par a) (out_list b)))
  nil → (Sum list_nil (Obj list_empty))
inn_list p = match p with
  Sum list_cons (Prod (Par a) r) → (cons a (inn_list r))
  Sum list_nil (Obj list_empty) → (nil)
  _ → raise "not a list"
```

These two morphisms *out_τ* and *inn_τ* can be *automatically* generated from every type τ. Mutually recursive types and type compositions can also be automatically abstracted by morphism composition. Details can be found in annex A.

Then a function that is independent from any type, but that depends on the data structure of its variable can be specified on type *Poly*. For instance, consider the functions *size* and *flatten*; they respectively calculate the number of *Par* occurrences in a value of type *Poly*, and their list:

```
size x = match x with
    Sum c y      → (size y)
    Prod y y'    → (size y) + (size y')
    Par y        → 1
    Obj z        → 0
flatten x h = match x with
    Sum c y      → (flatten y h)
    Prod y y'    → (flatten y (flatten y' h))
    Par y        → (cons y h)
    Obj z        → h
```

Then, functions that compute the size and the leaves list on type *tree* (binary trees) and type *list* are obtained by the following compositions:

```
type tree a = leaf a | node (tree a) (tree a)
size_list t = (size (out_list t))
flatten_list t h = (flatten (out_list t) h)
size_tree t = (size (out_tree t))
flatten_tree t h = (flatten (out_tree t) h)
```

Applying deforestation will eliminate the construction of the intermediate value of type *Poly*, and will lead to the expected functions. For instance, for the type *tree*, deforestation leads to:

```
size_tree t = match t with
  node a b → (size_tree a) + (size_tree b)
  leaf n → 1
flatten_tree t h = match t with
  node a b → (flatten_tree a (flatten_tree b h))
  leaf n → (cons n h)
```

The main advantage of type *Poly* is that a bijective morphism can be automatically derived from any (polymorphic) type. But this type is not the most natural to implement many algorithms. Actually, many algorithms need semantic information on values that are not necessary explicited by the structure of the type. For instance, it is difficult for the unification algorithm, to determine what and where are variables in the type *Poly*.

3.2 Correspondence relation

Previous section shows one way to generate a bijective morphism between any type τ and type *Poly*. This section proposes another method to generate morphism between data-structures. The aim is to infer a – not necessarily bijective – morphism between two arbitrary types. Consider the following type representing a binary tree with one or two integers at each node:

```
type clumsytree =
        node elements clumsytree clumsytree | nothing
type elements = one int | two int int
```

Now, consider the type τ:

```
type τ α = leaf
  | node1 α (τ α) (τ α)
  | node2 α α (τ α) (τ α)
```

A morphism between type *clumsytree* and type τ can be implemented with the following function:

```
let couplage t = match t with
  node (one a1) t1 t2 →
    (node1 a1 (couplage t1) (couplage t2))
  node (two a1 a2) t1 t2 →
    (node2 a1 a2 (couplage t1) (couplage t2))
  nothing → leaf
```

Two properties are verified by this morphism: type *clumsytree* is associated to $(\tau\ \alpha)$, and type *int* is associated to α. It is possible to denote these two properties by the following relation:

$$Cor(tree\ \alpha) = \{clumsytree\}$$
$$Cor(\alpha) = \{int\}$$

Such a relation is called a *correspondence relation* between types *clumsytree* and τ.

The aim is now to automatically derive the function *couplage* from a given correspondence relation. In [11] we propose such an inference algorithm for attribute grammars. It is easy to translate it into functional programming, as described in annex B.

The basic idea of this algorithm is to associate a sub-term of type *clumsytree* to a sub-term of type τ. Thus, the algorithm yields the following associations:

- the term $(node\ (one\ a_1)\ t_1\ t_2)$ must be associated to a term of type τ, composed with a tuple of type (α, τ, τ). So it is associated to the term $(node1\ a_1'\ t_1'\ t_2')$.
- the term $(node\ (two\ a_1\ a_2)\ t_1\ t_2)$ must be associated to a term of type τ, composed with a tuple of type $(\alpha, \alpha, \tau, \tau)$. So it is associated to the term $(node2\ a_1'\ a_2'\ t_1'\ t_2')$.
- the term (*nothing*) corresponds to a term of type τ composed with nothing else. So it is associated to (*leaf*).

Then, from such an association, it is very easy to generate the expected function *couplage*. Sometimes, more complex associations have to be defined.

In [11], we show that it is not always possible to find the associations, and we characterize these situations. A typical example is:

$$Cor(a') = \{a\} \qquad \texttt{type}\ a = c_1\ n$$
$$Cor(b') = \{b\} \qquad \texttt{type}\ n = c_2\ n\ n \mid c_3\ b$$
$$\texttt{type}\ b = \ldots$$

The problem is due to the fact that a can "derive" into an infinity of n which are not associated to anything by the correspondence relation. Moreover, in [11], we try to associate a term with a constructor instead of with a subterm. The general problem of "parsing" the leaves of a term like $(node\ (one\ a_1)\ t_1\ t_2)$ with the constructors of a given type remains opened.

Once again, this method fails when semantic information on types have to be taken into account in order to generate the morphism. Moreover, the morphism is often not bijective.

Even if none of the two previously exposed methods are powerful enough to infer almost expected morphisms, they could yet be considered as tools to construct complex morphisms by composition of several simple ones.

4 Deforestation

This section illustrates the power of deforestation in order to specialize instanciations of a "generic" program. After giving notations for the unification example, we show some critical steps of the complete deforestation process, for our deforestation method, namely the symbolic composition [5].

Unification example: recall that a well suited type for this problem is:

$$\texttt{type}\ term\ c\ a\ =\ var\ a\ |\ const\ c\ (list\ (term\ c\ a))$$

Suppose now that the unification algorithm is standardly written for this simple type:

$$unify\ :\ (term\ c\ a) \rightarrow (term\ c\ a) \rightarrow (a \rightarrow (term\ c\ a))$$

The expression $(unify\ t\ t')$ returns the substitution s if t and t' are equals modulo s. The substitution s is given as a function from variables to terms. If the substitution does not exist, the function raises the exception *No_unif*. The kernel of unification algorithm is implemented by the function *uni*:

```
uni s t t' = match (t, t') with
  (var x , var x') → if x = x' then s else (link s x t')
  ( _ , var x') → (link s x' t)
  (const c lt , const c' lt') →
    if c = c' then
      foldr (λ(a, a').λr.(uni r a a')) s (zip lt lt')
    else
      raise No_unif
```

where $(link\ s\ x\ t)$ adds the substitution $x = t$ to s if possible, and raises the exception *No_unif* otherwise. We then have:

$$unify\ t\ t' = (uni\ empty\ t\ t')$$

where *empty* is the empty substitution.

Now, to perform genericity, we have to specify for every type τ a morphism from τ to *term*. Since the unification returns a substitution, it is important to work with a bijective morphism. The function *all_unify* instantiates *unify* with such a morphism defined by the functions $in : term \rightarrow tree$ and $out : tree \rightarrow term$.

```
(all_unify in out) t t' =
  let s = (unify (out t) (out t')) in λx.(in (s x))
```

To instantiate the unification algorithm on trees where leaves are the variables, the two following morphisms are used:

```
tree_to_term t = match t with
  node a b →
    (const 1 [tree_to_term a ; (tree_to_term b)])
  leaf n → (var n)
term_to_tree t = match t with
  const 1 [a; b] →
    (node (term_to_tree a) (term_to_tree b))
  var n → (leaf n)
  _ → raise "not a tree !"
```

Then, unification on trees is specified by:

$$unify\_tree\ t\ t' = (all\_unify\ term\_to\_tree\ tree\_to\_term)\ t\ t'$$

The aim is now to transform this instantiation specification – the only one the programmer has to write – into a more specialized function.

Applying deforestation: the first step consists in specializing the definition of *unify_tree*:

```
unify_tree t t' =
  let s = (uni empty (tree_to_term t) (tree_to_term t'))
  in λx.(term_to_tree (s x))
```

Next, the composition of *uni* and *tree_to_term* is deforested into the function uni_1. In this function, many simplifications have been performed, especially the partial evaluation of the function *foldr*:

```
uni1 s t t' = match (t, t') with
  (leaf x , leaf x') →
    if x = x' then s else (link s x (tree_to_term t'))
  ( _ , leaf x') → (link s x' (tree_to_term t))
  (node a b , node a' b') → (uni1 (uni1 s b b') a a')
```

Now the composition of *link* and *tree_to_term* is deforested into the function $link_1$. Thus, the function uni_1 is updated into uni_2, and this leads to:

```
unify_tree t t' =
  let s = (uni2 empty t t') in λx.(term_to_tree (s x))

uni2 s t t' = match (t, t') with
  (leaf x , leaf x') →
    if x = x' then s else (link1 s x t')
  ( _ , leaf x') → (link1 s x' t))
  (node a b , node a' b') → (uni2 (uni2 s b b') a a')
```

At this point, the substitution s still associates variables to terms, and not variables to trees. But further deforestation is possible, and the function uni_3 will pre-calculate the composition of s with *term_to_tree*. At the end of the entire process, substitutions are physically constructed (in a list for instance). Then the substitution s is discarded and replaced by s_1, computed by the function uni_4. Consequently, the empty substitution is replaced by $empty_1$. This leads to:

```
unify_tree t t' =
  let s1 = (uni4 empty1 t t') in λz.(s1 z)
```

$$
\begin{array}{l}
uni_4\ s_1\ t\ t' = \mathtt{match}\ (t, t')\ \mathtt{with} \\
\quad (leaf\ x\ ,\ leaf\ x') \rightarrow \\
\quad\quad \mathtt{if}\ x = x'\ \mathtt{then}\ s\ \mathtt{else}\ (link_2\ s_1\ x\ t') \\
\quad (\ \_\ ,\ leaf\ x') \rightarrow (link_2\ s_1\ x'\ t)) \\
\quad (node\ a\ b\ ,\ node\ a'\ b') \rightarrow (uni_4\ (uni_4\ s\ b\ b')\ a\ a')
\end{array}
$$

This deforestation process seems to be complex, but it only consists in multiple application of *few* rules. Moreover these simple rules are expressed independently from any functional language. We are using an attribute grammar based formalism enriched by dynamic constructions in order to take into account composition and partial evaluation into one single transformation, called *symbolic composition*. Thus, deforestation is the key tool to achieve genericity by composition, since a unique framework is available independently from any programming language.

5 Conclusion

This article presents a general concept of generic programming, which is independent of any programming language. Instead of bringing different approaches into conflict, it expects large cross-fertilizations between different generic methods. Each of them has advantages and limitations, and offers different – and complementary – kinds of genericity.

In order to ease the cross-fertilizations it seems worthwhile to separate morphism specification from algorithm instantiation and specialization. Then, symbolic composition – or other deforestation method – is the basic tool which enables the specialization of an algorithm to be performed over new structures via morphisms specifications. As soon as a deforestation method is available, many ways to achieve genericity can be developed quickly, easily and efficiently.

Besides, there exists certainly other generic programming and specializing methods that should be considered. From our point of view, it will be interesting to carry out some *unified* way to specify morphisms and to exhibit families of automatic or semi-automatic methods to generate these morphisms.

References

[1] John Boyland and Susan L. Graham. Composing tree attributions. In *21st ACM Symp. on Principles of Programming Languages*, pages 375–388, Portland, Oregon, January 1994. ACM Press.

[2] Loïc Correnson. Généricité dans les grammaires attribuées. Rapport de stage d'option, École Polytechnique, 1996.

[3] Loïc Correnson. Programmation polytypique avec les grammaires attribuées. Rapport de DEA, Université de Paris VII, September 1997.

[4] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Attribute grammars and functional programming deforestation. In *Fourth International Static Analysis Symposium – Poster Session*, Paris, France, September 1997.

[5] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Symbolic composition. Technical Report 3348, INRIA, January 1998.

[6] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Attribute grammars and folds: Generic control operators. Rapport de recherche 2957, INRIA, August 1996.

[7] Rodney Farrow, Thomas J. Marlowe, and Daniel M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *19th ACM Symp. on Principles of Programming Languages*, pages 223–234, Albuquerque, NM, January 1992. ACM press.

[8] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN '84 Symp. on Compiler Construction*, pages 157–170, Montréal, June 1984. Published as *ACM SIGPLAN Notices*, 19(6).

[9] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *24th ACM Symp. on Principles of Programming Languages*, 1997.

[10] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[11] Carole Le Bellec, Martin Jourdan, Didier Parigot, and Gilles Roussel. Specification and Implementation of Grammar Coupling Using Attribute Grammars. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP '93)*, volume 714 of *Lect. Notes in Comp. Sci.*, pages 123–136, Tallinn, August 1993. Springer-Verlag.

[12] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conf. on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lect. Notes in Comp. Sci.*, pages 124–144, Cambridge, September 1991. Springer-Verlag.

[13] Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *In Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997.

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

[15] Gilles Roussel. *Algorithmes de base pour la modularité et la réutilisabilité des grammaires attribuées.* PhD thesis, Département d'Informatique, Université de Paris 6, March 1994.

[16] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Conf. on Functional Programming and Computer Architecture (FPCA'93)*, pages 233–242, Copenhagen, Denmark, June 1993. ACM Press.

[17] Philip Wadler. Deforestation: transforming programs to eliminate trees. In *Theoretical Computer Science*, volume 73, pages 231–248, 1990. (Special issue of selected papers from 2'nd European Symposium on Programming).

Annex A

Morphism generation for *Poly*

The following global-naming conventions are assumed:

G_{sum} is the type that represents the constructors of every type τ. Thus, the constructor c of type τ is denoted by the constructor τ_c of type G_{sum}.

G_{obj} is the type that represents values of any simple type appearing in some type τ. For instance, if τ contains integers and booleans:

$$\texttt{type}\ G_{obj} = \ldots \tau\_int\ int \mid \tau\_bool\ bool \ldots$$

out_τ is the function that implements a morphism between type τ and type *Poly*; inn_τ is the reciprocal function.

Notice that all the morphisms have to be infered before the generation of types G_{sum} and G_{obj}. But since type *Poly* is a polymorphic type, this is not a problem for separate compilation. Types are defined according to the following grammar:

τ	::=	$c_1\ \rho_1 \mid \ldots \mid c_n\ \rho_n$	constructors
ρ	::=	α	the polymorphic variable
	$\mid$	ι	basic type
	$\mid$	$(\tau\ \rho)$	type composition
	$\mid$	$\rho * \rho$	type product

Generation of out_τ: The algorithm is defined by inference rules. Their general scheme is:

$$\frac{conditions}{\tau \vdash x, \rho \Rightarrow t}$$

where τ is the current type, x is a term of type ρ, and t is the term x translated from type ρ to type *Poly*.

With these notations:

$$\frac{}{\tau \vdash x, \alpha \Rightarrow (Par\ x)} \qquad \frac{}{\tau \vdash x, (\tau\ \alpha) \Rightarrow (out\_\tau\ x)}$$

$$\frac{\tau \vdash x_i, \rho_i \Rightarrow t_i \qquad i = 1,2}{\tau \vdash x, (\rho_1 * \rho_2) \Rightarrow (\texttt{let}\ (x_1, x_2) = x\ \texttt{in}\ (Prod\ t_1\ t_2))}$$

$$\frac{}{\tau \vdash x, \iota \Rightarrow (Obj\ (\tau\_\iota\ x))}$$

$$\frac{\tau \vdash y, \rho \Rightarrow t}{\tau \vdash x, (\tau'\ \rho) \Rightarrow (shift\ (\lambda y.t)\ (out\_\tau'\ x))}$$

The function *shift* is needed when type composition occurs. Actually, to compute the morphism from $(\tau\ \rho)$ to *Poly*, a solution is: first, compute the morphism from τ to *Poly*; second, compose this first morphism with the one from $(\tau\ \alpha)$ to $(\tau\ \rho)$. The second morphism is simply computed by the function *shift*, where f is supposed to be a morphism from ρ to *Poly*.

```
shift f x = match x with
  (Par u) → (f u)
  (Prod a b) → (Prod (shift f a) (shift f b))
  (Sum c u) → (Sum c (shift f u))
  (Obj o) → (Obj o)
```

The function out_τ is derived with:

$$\frac{\tau = \ldots \mid c_k\ \rho_1 \ldots \rho_n \mid \ldots \qquad \tau \vdash x_i, \rho_i \Rightarrow t_i}{\begin{array}{l} out\_\tau\ x = \texttt{match}\ x\ \texttt{with} \\ \quad (c_k\ x_1 \ldots x_n) \rightarrow \\ \quad\quad (Sum\ (\tau\_c_k)\ (Prod\ t_1\ (Prod\ t_2 \ldots\ t_n))) \\ \quad \ldots \end{array}}$$

Generation of inn_τ: Here, the inference rule notation is:

$$\frac{conditions}{\tau, \varphi \vdash \rho \Rightarrow t, t'}$$

where τ is the current type and φ the function to apply where *Par* values are expected – useful for type compositions. For any type ρ, the algorithm generates the pattern t of type *Poly* that represents a term of type ρ, and its backward translation t' of type τ.

$$\frac{}{\tau, \varphi \vdash \alpha \Rightarrow t, (\varphi\ t)} \qquad \frac{}{\tau, \varphi \vdash (\tau\ \alpha) \Rightarrow t, (out\_\tau\ t)}$$

$$\frac{}{\tau, \varphi \vdash \iota \Rightarrow (Obj\ (\tau\_\iota\ u)), u}$$

$$\frac{\tau \vdash \rho_i \Rightarrow t_i, t'_i \qquad i = 1,2}{\tau, \varphi \vdash (\rho_1 * \rho_2) \Rightarrow (Prod\ t_1\ t_2), (t'_1, t'_2)}$$

$$\frac{\tau, \varphi \vdash \rho \Rightarrow t, t' \quad \psi \text{ new name} \quad \left\{ \begin{array}{l} \psi\ y = \texttt{match}\ y\ \texttt{with} \\ \quad t \rightarrow t' \\ \quad \_ \rightarrow \texttt{raise}\ \text{“}\texttt{error}\text{''} \end{array} \right.}{\tau, \varphi \vdash (\tau'\ \rho) \Rightarrow x, (unshift\_\tau'\ \psi\ x)}$$

Then:

$$\frac{\tau = \ldots \mid c_i\ \rho^i_1 \ldots \rho^i_n \mid \ldots \qquad \tau, \varphi \vdash \rho^i_k \Rightarrow p^i_k, t^i_k}{\begin{array}{l} unshift\_\tau\ \varphi\ x = \texttt{match}\ \ x\ \texttt{with} \\ \quad (Sum\ (\tau\_c_i)\ (Prod\ p^i_1 \ldots p^i_n)) \rightarrow (c_i\ t^i_1 \ldots t^i_n) \\ \quad \ldots \end{array}}$$

And finally:

```
parshift x = match x with
  (Par x) → x | _ → raise "error"
inn_τ x = (unshift_τ parshift x)
```

Example

To illustrate how type composition is processed, let us consider the following example:

```
type flower a = rose int a (tree (flower a)))
```

It leads to:

```
type G_obj = flower_int int | ...
type G_cons = flower_rose | tree_node | tree_leaf | ...
out_flower x = match x with
  rose a b c → (Sum (flower_rose) (Prod
        (flower_int a)
        (Prod (Par b) (shift out_flower (out_tree x))))
inn_flower x = unshift_flower parshift x
unshift_flower f x = match x with
  Sum flower_rose (Prod (flower_int a) (Prod p r)) →
    (rose a (f p) (unshift_tree inn_flower r))
```

Annex B

Correspondence relations

Let *Cor* be a correspondence relation from type τ_1 to type τ_2. We define the following objects:

Components: each type consists of several components. Each constructor, each argument of a constructor and each occurrence of a (sub-)type are components. The correspondence relation links components from type τ_1 to components from type τ_2.

Key component: a component is a *key* one if it is linked by the correspondence relation.

Neutral component: a component is a *neutral* one if it contains a *key* or *neutral* sub-component.

Dead component: a component that contains neither *key* nor *neutral* is *dead*.

Then, from a correspondence relation, it is possible to tag each component of a type. This defines the *Tag* annotation. For instance, with the previous exemple (c, i stands for the i-th argument of constructor c):

$$\begin{array}{l} Tag(nothing) = key\ (tree_{12}\ \alpha) \\ Tag(one) = neutral\ (elements) \\ Tag(one, 1) = key\ (\alpha) \\ Tag(node) = key\ (tree_{12}\ \alpha) \\ Tag(node, 1) = neutral\ (elements) \\ Tag(node, 2) = key\ (tree_{12}\ \alpha) \\ etc. \end{array}$$

To generate the morphism, it is necessary to look for *closed*-terms. A closed term is a finite term, whose root and leaves are tagged by *key*, and that contains only *neutral* internal constructors (*dead* components are discarded). For instance, ($node\ (two\ a_1\ a_2)\ t_1\ t_2$) is closed. But ($node\ e_1\ t_1\ t_2$) is not closed, since e_1 is tagged by *neutral*. And ($node\ (one\ a_1)\ (nothing)\ t_2$) is nor closed, since the constructor *nothing* replaces an internal *key* component.

It is easy to generate the closed-terms by a transitive closure (or fix-point) algorithm. The idea is to recursively replace a *neutral* leaf of a non-closed term by any sub-term of type τ. Of course, there exist conditions to insure the termination of the algorithm. See [11] for more details.

Now, the notion of signature is needed. The signature of a term is the list of its root and leaves *key*-tags. For instance, with the previous example, the closed-terms and their signature are:

$$\begin{array}{l} node\ (one\ a_1)\ t_1\ t_2\ : \alpha\ \tau\ \tau \rightarrow \tau \\ node\ (two\ a_1\ a_2)\ t_1\ t_2\ : \alpha\ \alpha\ \tau\ \tau \rightarrow \tau \\ nothing\ : () \rightarrow \tau \end{array}$$

Signature is extended to constructors of type τ_2. Thus, the signature of the constructor $node_1$ is the same as of the closed term ($node\ (one\ a_1)\ t_1\ t_2$) one. Then, each closed-term of type τ_1 is associated with one constructor of type τ_2 that has the same signature. With the previous example, the following association is obtained:

$$\begin{array}{l} node\ (one\ a_1)\ t_1\ t_2 \Rightarrow node_1 \\ node\ (two\ a_1\ a_2)\ t_1\ t_2 \Rightarrow node_2 \\ nothing \Rightarrow leaf \end{array}$$

From this association, the couplage function is easily inferred. Of course, many improvements could be done about correspondence relations. In the last step of the algorithm, the problem to solve is how to associate a closed tree of type τ_1 to some value of type τ_2. The solution proposed here is quite simple by associating signatures to constructors. Commutativity, associativity, parsing may be taking into account to find more complex associations.