Generic Programming by Program Composition (position paper)

Loïc CORRENSON, Etienne DURIS,
Didier PARIGOT and Gilles ROUSSEL

Abstract

Recently, generic programming becomes of a major interest in several programming paradigms. A recurrent idea to achieve genericity is to abstract computations from their representative data structures. This allows these generic specifications to be instantiated onto a large number of neighboring data structures. Moreover the program can be adapted when the data structures have to evolve.

Polytypic programming, adaptive programming and generic attribute grammars are generic programming methods related to this approach. Their comparison leads us to propose a common framework for generic programming: automatic generation of programs that compute morphisms between data structures, and program composition.

Thanks to this compositional approach, the complete specialization of generic programs could be advantageously delegated to some powerful and general deforestation method.

1 Introduction

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

A few years ago, Farrow *et al.* [7] devised a generic programming notion for attribute grammars [11, 18] that has been recently revisited [12, 1, 19]. This genericity is based on the observation that you can instantiate on a type τ_1 any function f defined on a type τ_2. The only component needed, if it exists, is a *coupling* described by a *grammar coupling* (morphism) that transforms every structure of type τ_1 into its "equivalent" structure of type τ_2. The composition of this coupling function with the function f performs the instantiation process. In this context, the 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]. Using this technique, the construction of the intermediate representation of type τ_2 could be discarded from the composition, and thus the original function f would be transformed and actually specialized onto the type τ_1.

This problem of intermediate data structure elimination, usually called *deforestation* [21], has also been widely studied in the functional programming community [16, 20, 17]. The comparison between functional and attribute grammar deforestation methods [6, 4] led us to discover an extension of the descriptional composition called *symbolic composition* [5]. This method deforests some functional programs for which existing functional methods fail. Since symbolic composition takes an important part in the attribute grammar genericity mechanism, this led us to compare our generic programming method with some functional ones.

In functional programming, the polymorphism provides a kind of genericity. But a polymorphic function can only be reused on types that have exactly the same structure. In order to relax this restriction, the notion of *polytypic programming* [10, 15] has been introduced. A polytypic function is a function that is defined by induction on the structure of user-defined data types [9]. This allows polytypic functions to be adapted to changing structures.

Moreover, in object-oriented programming, there exists another approach of structure-directed genericity called *tree-traversal* [14], that comes from the *adaptive programming* [13] concept. This allows a program that "traverses" a complex data towards its leaves to be generated whatever its structure is. Since the result implements some *walker* algorithm, it could be seen as a kind of morphism between the input data and the list of its components.

Our purpose is to present three different generic methods with a common framework, mainly based on the following concepts:

Generic programming is achieved by functions composition. These functions are divided into two kinds. These of the first kind (classical functions) implement generic algorithms on a convenient data structure. Functions of the second kind (morphisms) implements translations between neighboring data structures.

Morphism specification provides the actual genericity. Since it describes the translation between two data structures, it also gives the way to perform the algorithm instantiation. Most of morphisms involved in the compositions could be automatically derived, thanks to a small meta-language. This meta-language should be independent from (should not include) the original language[1]. Nevertheless, these morphisms could also be hand-written.

Specialization process performs the actual instantiation. The specified compositions have to be implemented. 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 data structures and partial evaluation.

[1]Haskell for polytypic, Scheme for tree-traversal...

The paper is organized as follows. Section 2 briefly presents the considered generic programming methods: polytypic programming, tree-traversal and generic attribute grammars. Section 3 shows how the two first methods can be reformulated according to the compositional approach of attribute grammars. Finally, section 4 describes the advantages and perspectives of this point of view about genericity.

Preliminary remark

In order to highlight cross-fertilizations between these different genericity methods, this paper presents their general principles rather than their technical aspects. The three systems are not presented in their integrity neither with their complete notations. So, by advance, we apologize to their authors for the large simplifications we made in the following presentations.

2 Three Generic Programming Methods

2.1 Polytypic programming

Classically in functional languages, functions can only be specified for one 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 computed by very similar functions. Polytypic programming proposes a way to specify such abstract recursive functions in order to instantiate them over several types.

Polytypic programming is based on the theoretical result that every polymorphic type can be expressed as a combination of sums, products, and elements of several kinds. This property is strongly tied to the notion of initial algebra and functors, from the category theory [16, 9].

As an example, let us consider the polymorphic types *list* α and *tree* α:

$$\begin{aligned} &\mathit{list}\ \alpha = \mathit{nil} \mid \mathit{cons}\ (\alpha\ ,\ \mathit{list}\ \alpha) \\ &\mathit{tree}\ \alpha = \mathit{leaf}\ (\alpha) \mid \mathit{node}\ (\mathit{tree}\ \alpha\ ,\ \mathit{tree}\ \alpha) \end{aligned}$$

These types can be automatically abstracted by their functors:

$$\begin{aligned} \mathcal{F}_{list} &= \texttt{Empty} + (\texttt{Par} \times \texttt{Rec}) \\ \mathcal{F}_{tree} &= \texttt{Par} + (\texttt{Rec} \times \texttt{Rec}) \end{aligned}$$

where `Rec` denotes the recursive type itself and `Par` denotes the polymorphic parameter α. More generally, in the same way any term (value) can be abstracted by the following constructors:

$+\ c\ x$	for the alternative $(c\ x)$ of a sum (e.g. the $\mathit{cons}(h,t)$ or *nil* alternative of a list).
$\times\ x\ x'$	for the product (x, x').
`Par` x	for interesting parameters.
`Rec` x	for recursive components.
`Const` x	for hidden parameters.
`Empty`	for empty values

For instance the list $[a;b;c]$ is abstracted into $+\ (cons)\ (a\ ,\ [b;c])$, since it is a *cons* alternative of a list, with a as the head element, and $[b;c]$ as the tail list.

Writing the generic program

We will specify the polytypic function *length* with respect to the previously listed constructors. Intuitively, since a sum (+) stands for possible alternatives, the length of a sum is the length of the considered alternative. In the same way, computing the length of a product (×) of terms consists in adding their lengths. Finally, the length of a parameter is "1" if it has to be count and "0" otherwise. These intuitive remarks lead to the following polytypic function *length* (*add* stands for the integer addition):

```
polytypic $length\ x$ =
  match (out $x$) with
   $+\ c\ y$      $\rightarrow (length\ y)$
   $\times\ y\ y'$     $\rightarrow (add\ (length\ y)\ (length\ y'))$
   Rec $y$      $\rightarrow (length\ y)$
   Par $y$      $\rightarrow 1$
   Const $y$    $\rightarrow 0$
   Empty      $\rightarrow 0$
```

Instance specification

Of course, one has to notice that this program needs an operator to abstract the input data structure in terms of sums and products. This is done by the operator `out` of the polytypic language. It is an initial morphism that must be defined for every possible type of its argument. In the polytypic frameworks [9], a type-checker exists and detects all these types, allowing the operator `out` to be automatically defined.

Instantiation & Specialization process

For an implementation of the polytypic function *length* for an argument of type *tree* α, the expression *length* $(x : tree\ \alpha)$ will be customized into $length_{tree\alpha}\ x$. Then, the operator `out` has to be defined on the type *tree* α as follows:

```
out $(leaf\ n) = +\ (leaf)\ (n)$
out $n$ = Par $n$            if $n$ is of type $\alpha$
out $(node\ x\ y) = +\ (node)\ (x,y)$
out $(x,y) = \times\ x\ y$
out $x$ = Rec $x$            if $x$ is of type $tree\ \alpha$
```

To complete the instantiation implementation, all abstract constructors $(+, \times,$ Par$\ldots)$ have to be eliminated. In [9] this is performed by applying an *ad-hoc*[2] partial evaluation dedicated to the constructors introduced by the operator `out`. This finally leads to the following specialized function:

[2]By *ad-hoc*, we mean that no *real* deforestation method has been pointed out.

```
let $length_{tree\alpha}$ $x$ =
  match $x$ with
   $node\ a\ b$   $\rightarrow$ ($add$ ($length_{tree\alpha}$ $a$) ($length_{tree\alpha}$ $b$))
   $leaf\ n$      $\rightarrow$ 1
```

For a more complete presentation of polytypic programming the reader is invited to refer at [9].

2.2 Tree-Traversal

The concept of *adaptive programming* [13] extends the classical object-oriented paradigm. It gives more flexibility between functions and data types by loosely coupling them through navigation specifications. This section deals with an implementation of this concept called *tree-traversal* [14].

Writing the generic program

An illustrative example for this tree-traversal kind of genericity is the problem of highlighting a particular word w in a text document. The basic specification one would intuitively write for this problem is:

```
at word {
    if (this.val=w) then this.highlight();
}
```

where ‘this’ denotes the current word and w the word to highlight. Since ‘document’ contains ‘paragraphs’, which themselves may contain ‘lines’ containing ‘words’, the previous specification is incomplete.

Instance specification

Highlighting the word w requires the programmer to write by hand many search methods to reach all words through the document structure. The tree-traversal technique uses the `traverse` statement to automatically generate these methods.

```
traverse
    from document to word
        at word {
            if (this.val=w) then this.highlight();
        }
```

Instantiation process

From such an instantiation specification, the system produces one method for each class that contains (even indirectly) words. These search methods are:

```
document::search(w) {
    for paragraph in this.paragraphs()
        { paragraph.search(w); }
}
paragraph::search(w) {
    for line in this.lines()
        { line.search (w); }
}
line::search(w) {
    for word in this.words()
        { word.search(w); }
}
word::search(w) {
    if (this.val=w) then this.highlight();
}
```

Of course, tree-traversal yields automatic generation of such methods that are boring to write by hand. But this approach is much more useful when data structures evolve. For instance, suppose that the previous document structure has to be modified in order to contain 'things' that are 'paragraphs' *or* 'tables' (that also contain words). Then, the search methods will be automatically updated while the instantiation specification remains unchanged. In fact, the more a program contains tree-traversal specifications, the more tolerant it is for data type structure evolution.

For a more complete presentation of tree-traversal method the reader is invited to refer at [14].

2.3 Attribute Grammar

Attribute grammars [11, 18] are declarative and structure-directed specifications. They specify on each type constructor *what* is to be computed instead of *how* it is computed. More precisely, programs are specified by oriented equations over the data structure.

For example, to compute the length of a list, the idea is to define an attribute (i.e., a value) *length* for every list x, denoted $x.length$. Since every list is build either by *nil* or *cons* constructors, the attribute grammar defining *length* attribute is the following:

$$\begin{array}{l} cons\ h\ t \rightarrow \\ \quad this.length = (add\ \ t.length\ 1) \\ nil \rightarrow \\ \quad this.length = 0 \end{array}$$

where *this* is the considered list. As a less trivial example, let us consider a document that consists of a list of chapters, each chapter being a list of sections, etc. The aim is to compute the list of the words in the document (to count them, or to search a word in a document...).

Writing the generic attribute grammar

A well suited type for abstracting a structure that contains elements is the type *bag* (a set with possible repetitions).

$$bag = union\ (bag, bag)\ \mid single\ (element)\ \mid empty$$

The list of elements of a *bag* is build by accumulation. So, the attribute grammar *bag.list* computing this list needs two attributes *outlist* and *inlist* (the later is initialized with the list *nil*):

```
single a →
   this.outlist = (cons a this.inlist)
empty →
   this.outlist = this.inlist
union a b →
   this.outlist = b.outlist
   b.inlist = a.outlist
   a.inlist = this.inlist
```

Instance specification

Suppose now that the complete document structure is defined with respect to several sub-types and their constructors: *doc* and *end_doc* for type *document*, *chap* and *end_chap* for type *chapter*, *sect* and *end_sect* for type *section*, etc.

$$\begin{aligned}
&document = doc\ (chapter, document) \mid end\_doc\\
&chapter = chap\ (section, chapter) \mid end\_chap\\
&section = sect\ (paragraph, section) \mid end\_sect\\
&paragraph = par\ (word, paragraph) \mid end\_par
\end{aligned}$$

To instantiate word-listing for documents, the idea is to generate a coupling function that transforms a document into a *bag* structure.

Intuitively, let *Cor* be the property (or relation) that associates each document, chapter, section, and paragraph with a *bag*, and each word with a *bag*'s element. Then *Cor* completely defines the morphism between *bag* and document structures. This correspondence relation *Cor* can be specified as follows:

$$\begin{aligned}
Cor(bag) &= \{document, chapter, section, paragraph\}\\
Cor(element) &= \{word\}
\end{aligned}$$

Instantiation

It is possible to write by hand an attribute grammar that computes a morphism consistent with the relation *Cor*: for each document, chapter, section and paragraph the attribute grammar *document.coupling* can be defined to represent their equivalent *bag*-structure.

```
doc c d →
   this.coupling = (union c.coupling d.coupling)
chap s c →
   this.coupling = (union s.coupling c.coupling)
sect p s →
   this.coupling = (union p.coupling s.coupling)
par w p →
   this.coupling = (union (single w) p.coupling)
```

Rather than write by hand this coupling attribute grammar, there exists algorithms to infer it automatically. In [12, 2], we formalize the correspondence and *grammar coupling* notions. We study some conditions for this coupling to be well-defined (a function). In addition, given an output grammar (e.g., *bag*), an input grammar (e.g., *document*) and a valid coupling between them (e.g., $\mathcal{C}or$), we propose an algorithm which constructs the *coupling attribute grammar* under certain conditions.

Let Γ be the programming environment containing the type description of *bag* and document. Let $\mathcal{C}$ be the algorithm that infer the coupling attribute grammar *document.coupling*. The instantiation can be formalized as:

$$\boxed{\mathcal{C}or, \Gamma \overset{\mathcal{C}}{\Rightarrow} document.coupling}$$

Specialization process using deforestation method

Then it is easy to compose the attribute grammar *document.coupling* with the *bag.list* one. The symbolic composition [5] transforms these two attribute grammars into a new single one, denoted *document.list* which no more constructs the intermediate *bag*-structure:

```
doc c d →
   this.outlist = d.outlist
   d.inlist = c.outlist
   c.inlist = this.inlist
chap s c →
   ...
sect p s →
   ...
par w p
   this.outlist = (cons w p.inlist)
```

Let SC be the symbolic composition. The specialization process is defined by:

$$\boxed{document.coupling \circ bag.list \overset{\text{SC}}{\Rightarrow} document.list}$$

The important point to notice here is that morphism generation and deforestation application are totally separated from each other.

3 Compositional Approach

This section presents a different approach of polytypic and tree-traversal programming.

3.1 Polytypic programming

In the original polytypic approach, special constructors ($+$, $\times$,`Rec`, `Par`, etc.) are introduced to manipulate both the data and their structure. Let us define the following type:

$$\begin{array}{rcl} \mathcal{F} & = & \mathit{Sum}\ (\mathit{Object}, \mathcal{F}) \\ & | & \mathit{Prod}\ (\mathcal{F}, \mathcal{F}) \\ & | & \mathit{Par}\ (\mathit{Object}) \\ & | & \mathit{Rec}\ (\mathcal{F}) \\ & | & \mathit{Obj}\ (\mathit{Object}) \end{array}$$

The important point is that $\mathcal{F}$ is a type defined in the *original* functional language, without using any special constructor. The only technical point to solve is the definition of the *Object* type, that can represent heterogeneously any value of any type[3].

The type $\mathcal{F}$ allows the *length* function to be specified in the original language:

```
let length x = match x with
    Sum c y    → (length y)
    Prod y y'  → (add (length y) (length y'))
    Rec y      → (length y)
    Par y      → 1
    Obj z      → 0
```

Suppose now that the function *length* is needed on type *tree* α. This instantiation is performed thanks to an explicit morphism from type *tree* α to type $\mathcal{F}$, easy to write as the *out_tree*α function:

```
let out_treeα t = match x with
    node a b  → (Sum (node) (out_treeα a) (out_treeα b))
    leaf y    → (Sum (leaf) (Par y))
```

Notice that *out_tree*α is no more an operator like the `out` one in polytypic language, but just a function written in the original functional language. Then the function *length* on type *tree* α is obtained by the following composition:

$$\mathit{length\_tree}\alpha\ t = (\mathit{length}\ (\mathit{out\_tree}\alpha\ t))$$

Applying deforestation will eliminate the construction of the $\mathcal{F}$ term, and will lead to the expected function *length* on type *tree* α.

Here, function *out_tree*α has been hand-written. In the same way as the polytypic operator `out` is inferred, this function can be automatically generated.

[3]Many solutions exist: sub-typing, cast, or explicit constructors for each type involved...

System rephrasing

Considering the previous example, we can rephrase the polytypic system into:

$$\boxed{\frac{\mathcal{T} \overset{\mathcal{C}}{\Rightarrow} out\_\mathcal{T}}{f, \mathcal{T} \Rightarrow f \circ out\_\mathcal{T}} \qquad f \circ out\_\mathcal{T} \overset{deforestation}{\Rightarrow} f\_\mathcal{T}}$$

where $\circ$ is the standard composition in the original language. Thus a general deforestation method can be applied, rather than an *ad-hoc* method which would be restricted to polytypic functions. So, a method like HYLO system [17], or symbolic composition [5], can advantageously take charge of the specialization process in polytypic approach. Of course, some interesting polytypic features still remain to be converted or generalized.

This above presentation comes from a more detailed comparison between generic attribute grammars and polytypic programming that is presented in [3].

3.2 Tree Traversal

Compositional approach

Recall the example presented in section 2.2, and consider the following document D:

$$D \rightarrow \begin{cases} P_1 \rightarrow \{L_1 \rightarrow \{w_1\,;w_2\}\} \\ P_2 \rightarrow \{L_2 \rightarrow \{w_3\,;w_4\}\,;L_3 \rightarrow w_5\} \end{cases}$$

D consists of two paragraphs P_1 and P_2. P_1 consists of the line L_1 that consists of the words w_1 and w_2, etc. The traversal methods 'search' will traverse the document D in the following order:

$$D\,;P_1\,;L_1\,;w_1\,;w_2\,;P_2\,;L_2\,;w_3\,;w_4\,;L_3\,;w_5$$

Let us denote by T the type of such an heterogeneous list. Thus the traversal can be divided into two programs. Given any document, the first one produces a list of type T. The second one performs the highlighting of w in any list of type T. Their composition yields the highlighting of w in a document.

Now the real instantiation has to be performed, and the construction of the intermediate list has to be eliminated. We do not know what kind of deforestation method already exists in object-oriented programming. Nevertheless, we are sure that it is possible to define some new specialization method in this area, based on the symbolic composition principles.

4 Compositional Genericity

In this section, an abstract generic programming system is defined independently from any language. It is based on two strongly separated parts. The first one is a coupling algorithm that produces a morphism between two data structures. 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 deforestation in order to specialize such compositions. Let s be the morphism specification, $\mathcal{C}$ the coupling algorithm, f the "generic" function, g the morphism function, h the expected specialized function, and Γ the programming environment defining types or classes, etc. The abstract generic-system is defined as follows:

$$\boxed{\frac{s, \Gamma \overset{\mathcal{C}}{\Rightarrow} g}{f, s, \Gamma \Rightarrow f \circ g} \qquad \frac{f \circ g \overset{deforestation}{\Rightarrow} h}{f \circ g \Rightarrow h}}$$

The specialization process of the polytypic method is very closed to those of the generic attribute grammar. More precisely, they are akin to deforestation. Referring previous comparisons [4, 5] between functional and attribute grammar deforestation, it is worthwhile to use a general algorithm rather than *ad-hoc* ones. In a compositional approach, generic programming will freely benefit from deforestation improvements. In object-oriented programming, it is less clear that the specialization process could be performed by a deforestation-like transformation. As a future work, it seems interesting to define a new specialization of object oriented programs based on deforestation principles.

At this point, it is important to identify the instantiation process separately from the specialization process. Furthermore, both the generic program and the morphism generated during the instantiation ought to be written with the *original* language, since any general deforestation method will be freely and easily applicable.

Moreover, remark that the presented generic systems provide methods to automatically infer morphisms. So it is worthwhile to specify these morphisms with some small independent meta-language rather than with super-languages that include the original ones.

Then, it would be nice to transpose each of these methods on every other paradigms. For instance, the polytypic `out` operator definition suggests new methods to produce coupling attribute grammars. In the same way, the correspondence relation suggests extensions for the fixed definition of `out` operator.

Another interesting problem is to look for a unified formalism that describes all these morphism specifications. This would ease the transposition of specific methods for morphism generation onto other programming paradigms.

5 Conclusion

This article tends to extract a general concept of generic programming from different methods. Instead of bringing them into conflict, we expect large cross-fertilizations. Each method 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 instantiation and specialization. Then, symbolic composition – or other deforestation method – is the basic tool which enables the specialization of an algorithm 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 efficiency.

Besides, there certainly exists many other generic programming and specializing methods that should be included in this comparison. This is our strong belief and the thrust of our future works. In our point of view, future research will have 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] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lect. Notes in Comp. Sci.*, pages 68–114. Springer-Verlag, 1996.

[11] 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).

[12] 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.

[13] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[14] Cristina Videira Lopes and Karl J. Lieberherr. AP/S++: A CASE-study of a MOP for purposes of software evolution. Technical Report NU-CCS-95-?, Xerox PARC and Northeastern University, November 1995. ftp://ftp.ccs.neu.edu/pub/people/lieber/reflection-adaptive.ps.

[15] Lambert Meertens. Calculate polytypically. In Herbert Kuchen and S. Doaitse Swierstra, editors, *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140 of *Lect. Notes in Comp. Sci.*, pages 1–16, Aachen, September 1996. Springer-Verlag.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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).