Equational Semantics

Loïc Correnson

INRIA - Rocquencourt
email: Loic.Correnson@inria.fr
http://www-rocq.inria.fr/oscar/

Abstract

Attribute grammars are well-designed to construct complex algorithms by composing several ones together. Actually, there exists a powerful transformation called descriptional composition which highly simplifies the composition of two attribute grammars by removing useless intermediate constructions.

However, most of non-linear algorithms can not be expressed with attribute grammars. Thus, many compositions can not be simplified by the descriptional composition. In this paper, we present Equational Semantics, a formalism largely inspired by attribute grammars but where non-linear algorithms can be encoded. More precisely, instead of being restricted to one input static tree as it is the case for attribute grammars, an algorithm encoded with Equational Semantics may use dynamically constructed trees.

This formalism consists in an very poor abstract syntax. We present its semantics and some of its transformations such as partial evaluation and descriptional composition (also called deforestation). In some sense, Equational Semantics is a kind of lambda-calculus dedicated to program transformations.

1. Introduction

For many years, we try to promote our approach for generic programming and software reuse. It consists in composing different basic components together in order to produce more complex ones. Each basic component must be robust and general, so using them in particular cases may be costly because of some translation components or unspecialized algorithms.

Attribute grammars seems to be an interesting model to deal with this kind of generic programming since there is an algorithm, the descriptional composition [5, 6, 12], which simplifies a composition and produces a new and more efficient attribute grammar. However, this descriptional composition may fail: for instance, it may produce multiple definitions for an attribute, or it may introduce a circularity into attribute dependences.

More generally, an attribute grammar can only encode an algorithm which is linear in the number of nodes of its input tree. A syntactic reason for this is the impossibility to dynamically compute over attributes that are not linked to the input tree of the attribute grammar. The key point of our approach consists in removing this impossibility.

Let us consider the following example written with a straightforward notation. It defines an attribute grammar which computes the length of a list and its reversed list (with an accumulator). The first part of the attribute grammar introduces type definitions:

$$
\begin{array}{l}
\texttt{type}\ \mathit{list},\ \mathit{int} \\
\texttt{constructors} \\
\quad \mathit{cons} :\ \mathit{int} * \mathit{list} \ \rightarrow \mathit{list} \\
\quad \mathit{nil} :\ \rightarrow \mathit{list} \\
\texttt{synthesized}(\mathit{list}) =\ \mathit{rev} : \mathit{list}\ \mathit{length} : \mathit{int} \\
\texttt{inherited}(\mathit{list}) =\ \mathit{accu} : \mathit{list}
\end{array}
$$

Then the core of the attribute grammar comes up :

$$
\begin{array}{l}
\mathit{cons}\ x1\ x2 : \\
\quad \mathit{rev} = x2.\mathit{rev} \\
\quad x2.\mathit{accu} = (\mathit{cons}\ x1\ \mathit{accu}) \\
\quad \mathit{length} = (+\ 1\ x2.\mathit{length}) \\
\mathit{nil} : \\
\quad \mathit{rev} = \mathit{accu} \\
\quad \mathit{length} = 0
\end{array}
$$

In this example, there is a functional dependency between the inherited attribute *accu* and the synthesized one *rev* : the expression $x2.\mathit{rev}$ can be seen as a call to some function (or procedure, or visit, or whichever is appropriate) which computes on the sub-tree $x2$ the synthesized attribute *rev* with respect to the value of its inherited attribute *accu*.

Actually, in classical attribute grammars, it is only possible to use these "function calls" on a sub-tree of the (static) input tree of the program. With such a restriction, it is impossible to consider calls on dynamically-constructed trees or multiple calls on one sub-tree with different values for its inherited attributes. This is why an attribute grammar can only encode linear algorithms. Then the key point of our approach consists in introducing local definition, such as :

$$L1 = (\mathit{cons}\ x1\ x2.\mathit{rev})$$

Then, we allow to use expressions like $L1.\mathit{rev}$ and to define a value for $L1.\mathit{accu}$. Thus, it becomes possible to define the reverse of the reverse of a list :

$$
\begin{array}{l}
\texttt{type}\ \mathit{unit} \\
\texttt{constructors} \\
\quad \mathit{reverse} : \mathit{list} \ \rightarrow \mathit{unit} \\
\texttt{synthesized}(\mathit{unit}) =\ r : \mathit{list} \\
\mathit{reverse}\ x1 : \\
\quad r = L1.\mathit{rev} \\
\quad L1.\mathit{accu} = (\mathit{nil}) \\
\quad L1 = x1.\mathit{rev} \\
\quad x1.\mathit{accu} = (\mathit{nil})
\end{array}
$$

Here, to compute the attribute r of the tree $(\mathit{reverse}\ l)$, the list l is reversed, and this dynamically constructed list is also reversed. This algorithm is still linear, but such dynamic constructions allow to encode non-linear algorithm. See section 3 for more examples.

But introducing such syntactic features merely modify the semantics of attribute grammars. Actually, we must completely redefine it. This is why we propose a new formalism, where we only kept the essential of attribute grammars to deal with program transformations, namely the notion of constructors and attributes.

The result is a kind of lambda-calculus, with a notation closed to the one of attribute grammars, especially dedicated to program transformations. We called this formalism Equational Semantic and it is presented in the section 2 of this paper. Section 3 provides examples. Section 4 is a short presentation of how to generate evaluators which compute the attributes of a tree. In section 5, we define what should be a *correct* transformation. Section 6 describes transformations, especially partial evaluation and deforestation.

Related Works : There exists a lot of extensions to attribute grammars. A common goal for them is to enlarge the expressiveness of standard attribute grammars. We want to mention here higher-order attribute grammars [14], tree-transducers [8], and dynamic attribute grammars [11]. Since all of them are able to encode $\lambda-$calculus, we will not expose in this article why and how their equivalence holds. We are interested here in showing an extension of an attribute grammar transformation method, the descriptional composition, which applies to non-linear programs thanks to Equational Semantics.

2. Equational Semantics

This section defines notions and vocabulary for the equational semantics formalism.

Terms: Terms are built using *constructors* or *primitives* which take variables or sub-terms as parameters. There is no function call.

Variables: They name or represent terms. A variable can have several forms:

- $x.k$ (k is an integer) represents the k-th sub-term of (the term represented by) the variable x.
- $x.a$ (a is an attribute name) represents the attribute a attached to the variable x.
- $x.L_k$ (k is an integer) represents a local variable associated to the variable x.

The special variable α is used as a root variable.

Attributes: An attribute a represents a computation and the variable $x.a$ represents the result of this computation on the term represented by the variable x.

Equation Systems : The considered equations are of the form $x = t$, where the left-hand-side is restricted to be a variable. A system Σ is a set of equations.

Properties and Program: A program is defined by a set of properties that rely on attributes. For instance, incrementing an integer is represented by the following property about the attribute *inc* :

$$(\forall x) \quad x.inc = (+\ x\ 1)$$

We will only consider properties which depend on the constructor appearing at the head of a term. For instance, the *length* attribute defining the length of a list verifies the two following properties:

$$(\forall x)\left\{\begin{array}{l} x = (cons\ \ \ldots) \Rightarrow \\ \quad x.length = (+\ 1\ x.2.length) \\ x = (nil) \Rightarrow \\ \quad x.length = 0 \end{array}\right.$$

To simplify notations, the universally quantified variable x is denoted by the special variable α. This yields the following specification, which is (a piece of) a program in equational semantics:

$$\begin{cases} \mathit{cons} \rightarrow \\ \quad \alpha.\mathit{length} = (+\ 1\ \alpha.2.\mathit{length}) \\ \mathit{nil} \rightarrow \\ \quad \alpha.\mathit{length} = 0 \end{cases}$$

The complete syntactic definition of a program in equational semantics is given below:

$\mathcal{N}$, *Att*, *Cons* and *Prim* are respectively the sets of integers, attributes, constructors and primitives.

$$\begin{array}{lcll} \mathcal{P} & ::= & (c \rightarrow p^*)^* & \\ p & ::= & x = t & \\ x & ::= & \alpha & \\ & | & x.a & a \in \mathit{Att} \\ & | & x.k & k \in \mathcal{N} \\ & | & x.L_i & i \in \mathcal{N} \\ t & ::= & x & \\ & | & (c\ t^*) & c \in \mathit{Cons} \\ & | & (\pi\ t^*) & \pi \in \mathit{Prim} \end{array}$$

Deduction Rule: A deduction rule φ is a function which takes a system and generates new equations according to it. The basic deduction rules are described below.

$$\varphi_{sub}(\Sigma) = \{x.k = t_k \\ \quad | \ x = (c\ t_1 \ldots t_n) \in \Sigma\}$$

$$\varphi_{subst}(\Sigma) = \{x = t[y := t'] \\ \quad | \ x = t \in \Sigma, y = t' \in \Sigma\}$$

$$\varphi_{prim}(\Sigma) = \{x = t' \\ \quad | \ x = t \in \Sigma, t \triangleright t'\}$$

$$\varphi_{prog}(\mathcal{P})(\Sigma) = \{p[x]\ ,\ \forall p \in A \\ \quad | \ x = (c \ldots) \in \Sigma, (c \rightarrow A) \in \mathcal{P}\}$$

$\triangleright$ is a rewriting rule over terms. The substitution $[x := t]$ replaces the *full* occurrences of variable x by t (i.e. x is not substituted in $x.a$, $x.k$ or $x.L_i$). The substitution $p[x]$ replaces each text occurrence of α in the property p by the variable x.

The deduction rule φ_{sub} is used to have access to sub-terms; for instance, if $x = (c\ t_1\ t_2)$ then the variable $x.1$ represents the sub-term t_1. The deduction rule φ_{subst} substitutes a variable by a term. The deduction rule φ_{prim} handles primitive computations; for instance $x = (+\ 1\ 1)$ gives $x = 2$. The deduction rule φ_{prog} depends on the program $\mathcal{P}$ and applies its *properties* (this notion is defined below). The program transformations described section 6 will be carried out by adding more deduction rules to this basic kernel.

Execution: While a program is defined by a set of properties, its execution[1] is a system of equations. This system is constructed by applying deduction rules to an initial system which represents the input data of the program.

The execution of a program involves the following definitions:

$$\psi ::= \varphi^*$$
$$\psi(\Sigma) = \Sigma \cup \bigcup_{\varphi \in \psi} \varphi(\Sigma)$$
$$\Sigma_\psi = \bigcup_{n \in \mathcal{N}} \psi^n(\Sigma)$$

The result of executing the program $\mathcal{P}$ with the initial system Σ and the set of deduction rules ψ is the system Σ_ψ. The basic kernel of deduction rules is:

$$\psi_{basic} = \{\varphi_{sub},\ \varphi_{subst},\ \varphi_{prim},\ \varphi_{prog}(\mathcal{P})\}$$

The semantics of a program according to the set of deduction-rule ψ is the function which associates, to an input system Σ, the system Σ_ψ.

Such a semantics can be computed, and with a large amount of technical improvement[2], it can be computed efficiently. We have implemented a prototype, called `EQS`, which performs such computations, and more generally, manipulates and transforms programs in Equational Semantics.

3. Examples

This section intuitively presents how to encode various kind of algorithms with equational semantics. The example of executions come from the ones automatically computed by our implemented prototype `EQS`.

3.1. Attribute Grammars

As an example of encoding attribute grammars, we choose the example of reversing a list with an accumulator. The attribute *accu* is used to accumulate the elements and the final result is returned through the attribute *rev*. In the beginning, *accu* must be set to the empty list (*nil*). This program is specified in equational semantics as follows:

$$\begin{array}{l} cons \rightarrow \\ \quad \alpha.rev = \alpha.2.rev \\ \quad \alpha.2.accu = (cons\ \alpha.1\ \alpha.accu) \\ nil \rightarrow \\ \quad \alpha.rev = \alpha.accu \end{array}$$

Actually, a program which never use local variables looks like an attribute grammar. Now, let us consider the following initial system :

$$\left\{ \begin{array}{l} x = (cons\ 1\ (cons\ 2\ (nil))) \\ x.accu = (nil) \end{array} \right.$$

[1]More precisely, it is the *trace* of an execution of the program.
[2]We do not describe them in this paper

The application of the basic kernel deduction rules yields the following execution. However, the entire execution is too large to be reported here, so we only report some new equations. For each of them, the deduction rule which produced it is noticed inside brackets.

$$
\begin{array}{lr}
x.rev = x.2.rev & [prog] \\
x.2.accu = (cons\ x.1\ x.accu) & [prog] \\
x.1 = 1 & [sub] \\
x.2 = (cons\ 2\ (nil)) & [sub] \\
\hline
x.2.rev = x.2.2.rev & [prog] \\
x.2.2.accu = (cons\ x.2.1\ x.2.accu) & [prog] \\
x.2.1 = 2 & [sub] \\
x.2.2 = (nil) & [sub] \\
\hline
x.2.2.rev = x.2.2.accu & [prog] \\
\hline
x.2.accu = (cons\ 1\ (nil)) & [subst] \\
x.2.2.accu = (cons\ 2\ (cons\ 1\ (nil))) & [subst] \\
(\ldots) & [subst] \\
x.rev = (cons\ 2\ (cons\ 1\ (nil))) & [subst]
\end{array}
$$

To define a function that reverses a list, the constructor *reverse* is introduced. It stands for the call of this function while the attribute r is defined to catch the result of this call.

$$
\begin{array}{l}
reverse \rightarrow \\
\quad \alpha.r = \alpha.1.rev \\
\quad \alpha.1.accu = (nil)
\end{array}
$$

Now, given a list l and the equation $x = (reverse\ l)$, the reversed list is represented by the variable $x.r$.

3.2. Dynamic Trees

In the previous example, the recursion is driven by the constructors *cons* and *nil*. For functions like factorial, the recursion is only driven by a conditional expression. First, as like as in the previous example, a constructor *factorial* and an attribute r are used to represent a call to factorial. Second, for all variable x such that $x = (factorial\ t)$ the new local variable $x.L_1$ represents the result of the comparison $(<\ 1\ t)$ which drives the recursion. The computation is then continued on the constructor *true* or *false* through the attribute *fact*.

$$
\begin{array}{l}
factorial \rightarrow \\
\quad \alpha.r = \alpha.L_1.fact \\
\quad \alpha.L_1.n = \alpha.1 \\
\quad \alpha.L_1 = (<\ 1\ \alpha.1) \\
true \rightarrow \\
\quad \alpha.fact = (*\ \alpha.n\ \alpha.L_2.r) \\
\quad \alpha.L_2 = (factorial\ (-\ \alpha.n\ 1)) \\
false \rightarrow \\
\quad \alpha.fact = 1
\end{array}
$$

To illustrate how conditional recursions work with the local variable $\alpha.L_1$, we present now the execution from the initial system $\{x = (factorial\ 2)\}$:

$x.r = x.L_1.\mathit{fact}$	$[\mathit{prog}]$
$x.L_1.n = x.1$	$[\mathit{prog}]$
$x.L_1 = (<\ 1\ x.1)$	$[\mathit{prog}]$
$x.L_1.n = 2$	$[\mathit{sub}, \mathit{subst}]$
$x.L_1 = (<\ 1\ 2)$	$[\mathit{sub}, \mathit{subst}]$
$x.L_1 = (\mathit{true})$	$[\mathit{prim}]$
$x.L_1.\mathit{fact} = (*\ x.L_1.n\ x.L_1.L_2.r)$	$[\mathit{prog}]$
$x.L_1.L_2 = \quad (\mathit{factorial}\ (-\ x.L_1.n\ 1))$	$[\mathit{prog}]$
$(...)$	$[\ldots]$
$x.L_1.L_2.r = 1$	$[\ldots]$
$x.L_1.\mathit{fact} = (*\ 2\ 1)$	$[\mathit{subst}]$
$x.r = 2$	$[\mathit{prim}, \mathit{subst}]$

3.3. Composition

The example we present here does not belong to the scope of classical attribute grammars. More precisely, it can be encoded with two attribute grammars composed together, but the composition itself can not. Let n be a Peano integer, we build with the attribute *bin* a balanced binary tree of depth n with a first attribute grammar. Then a second one counts the leaves of this constructed tree with the attributes s and h, producing a new Peano integer m. Thus, we have $m = 2^n$. The composition is computed in the attribute r of the constructor *exp*.

The first attribute grammar is:

$$\begin{array}{l} \mathit{succ} \rightarrow \\ \quad \alpha.\mathit{bin} = (\mathit{node}\ \alpha.\mathit{1}.\mathit{bin}\ \alpha.\mathit{1}.\mathit{bin}) \\ \mathit{zero} \rightarrow \\ \quad \alpha.\mathit{bin} = (\mathit{leaf}) \end{array}$$

The second one is:

$$\begin{array}{ll} \mathit{node} \rightarrow & \mathit{leaf} \rightarrow \\ \quad \alpha.s = \alpha.1.s & \quad \alpha.s = (\mathit{succ}\ \alpha.h) \\ \quad \alpha.1.h = \alpha.2.s & \\ \quad \alpha.2.h = \alpha.h & \end{array}$$

The composition is defined by:

$$\begin{array}{l} \mathit{exp} \rightarrow \\ \quad \alpha.r = \alpha.L_3.s \\ \quad \alpha.L_3.h = (\mathit{zero}) \\ \quad \alpha.L_3 = \alpha.1.\mathit{bin} \end{array}$$

Thus, if n and m are Peano integers such that $m = 2^n$, then the initial system $\{x = (\mathit{exp}\ n)\}$ produces the equation $x.r = m$. Both "attribute grammars" are linear algorithms, but the size of the tree produced by the first one is an exponential of the size of the input tree. Thus the composition of these two attribute grammars produces an exponential algorithm. The composition itself can not be encoded with one attribute grammar.

Notice that the previous specification is transformed by our deforestation method into:

$$
\begin{array}{ll}
succ \rightarrow & exp \rightarrow \\
\quad \alpha.s' = \alpha.L_1.s' & \quad \alpha.r = \alpha.1.s' \\
\quad \alpha.L_1 = \alpha.1 & \quad \alpha.1.h' = (zero) \\
\quad \alpha.L_1.h' = \alpha.1.s' & \\
\quad \alpha.1.h' = \alpha.h' & \\
zero \rightarrow & \\
\quad \alpha.s' = (succ\ \alpha.h') &
\end{array}
$$

This result could not be encoded with classical attribute grammar since the visit which computes s' from h' is called twice with two different values for the attribute h' (look at the constructor *succ*). Here, the local variable $\alpha.L_1$ is identical to $\alpha.1$, but $\alpha.L_1.h'$ and $\alpha.1.h'$ represent different values. Notice that the classical descriptional composition composition failed in composing these two attribute grammars.

4. Evaluators

In this section, we show how to construct an *evaluator* for an equational semantics specification. An *evaluator* is a set of recursives *visits* that computes, for any tree t, the values of some attributes associated to t. By definition, the visit-call denoted by $[h_1 \dots h_m \rightarrow s_1 \dots s_n]\ (t)$ computes all the attributes s_i of t if and only if all the attributes h_j of t have been already computed. A visit is defined for each constructor by an ordered list of actions. An action could be either a call to a visit or the evaluation of an equation.

For instance, the following evaluator reverses a list :

$$
\begin{array}{l}
[accu \rightarrow rev] \\
\quad cons \rightarrow \\
\qquad \texttt{eval}\ \alpha.2.accu = (cons\ \alpha.1\ \alpha.accu) \\
\qquad \texttt{visit}\ [accu \rightarrow rev]\ (\alpha.2) \\
\qquad \texttt{eval}\ \alpha.rev = \alpha.2.rev \\
\quad nil \rightarrow \\
\qquad \texttt{eval}\ \alpha.rev = \alpha.accu \\
[\rightarrow r] \\
\quad reverse \rightarrow \\
\qquad \texttt{eval}\ \alpha.1.accu = (nil) \\
\qquad \texttt{visit}\ [accu \rightarrow rev]\ (\alpha.1) \\
\qquad \texttt{eval}\ \alpha.r = \alpha.1.rev
\end{array}
$$

The construction of the evaluators is performed by a fix-point algorithm. The main idea is to compute step by step a pool of available visits. We first define the following operations :

- $Vcons(P, c)$: it finds all the visits that computes attributes on a constructor c. To make these visits, all the visits in pool P are assumed to be available on the sub-terms of c and on its local variables.
- $Vall(P)$: it computes (P', T) where P' is new pool of visits, and T is a table which associates each constructor to its visits. The result of *Vall* is such that for all constructor c, $T(c) = Vcons(P, c)$ and $P' = \bigcup T(c)$
- $Vverify(v, T)$: for the visit $v = [H \rightarrow S]$, it verifies that for each constructor c such that at least one attributes of S is defined on c, there exists a visit $[H' \rightarrow S]$ in $T(c)$ and $H' \subset H$. A visit that verifies this property is called "verified". If it is not the case, then the visit v may be undefined on a constructor and should be eliminated.

With such basic components, the fix-point algorithm is defined as follows:

$$\begin{array}{lcl} P_0 & = & \{[\rightarrow a] \mid a \in Att\} \\ P_{n+1} & = & F(P_n) \end{array}$$

where F is defined by:

$$\begin{array}{ll} F(P) = \{v \mid & v \in P',\ \ Vverify(v,T), \\ & (T,P') = Vall(P)\} \end{array}$$

When the fix-point is reached, the remaining visits correctly compute the values of the attributes. As an example, here is the first iteration to compute the visits to reverse a list:

$$P_0 = \{[\rightarrow rev],\ [\rightarrow r]\}$$

The computations of *Vcons* lead to:

$$\begin{array}{l} Vcons(P_0, cons) = \\ \quad [\rightarrow rev] \\ \qquad \texttt{visit}\ [\rightarrow rev]\ (\alpha.2) \\ \qquad \texttt{eval}\ \alpha.rev = \alpha.2.rev \\ Vcons(P_0, nil) = \\ \quad [accu \rightarrow rev] \\ \qquad \texttt{eval}\ \alpha.rev = \alpha.accu \\ Vcons(P_0, reverse) = \\ \quad [\rightarrow r] \\ \qquad \texttt{visit}\ [\rightarrow rev]\ (\alpha.1) \\ \qquad \texttt{eval}\ \alpha.r = \alpha.1.rev \end{array}$$

Thus, after the first computation of *Vall* the visit $[\rightarrow rev]$ must be removed since it is not “verified” for *cons*. However, the new visit $[accu \rightarrow rev]$ is “verified” by *cons* and *nil*. Of course, since the fix point has not been reached, the evaluators found are not correct. Thus we have:

$$P_1 = \{[accu \rightarrow rev],\ [\rightarrow r]\}$$

Then, the second step produces the right evaluators and the fix point is reached.

Of course, this simple algorithm have to be improved to be efficient. The critical point is the computation of *Vcons* which seams to be highly exponential. However, a large amount of the constructed visits are identical (modulo permutation), and it is possible to compute them together. In practice, with our implemented prototype `EQS`, the complexity of the entire algorithm remains reasonable.

5. Safe Transformations

Intuitively, a transformation is correct if the transformed program produces the same results as the original one. In section 2 we define the execution of a program according to an input system Σ_i.

However, this execution is a system which contains many intermediate computations mixed with the expected result. Thus, we have to define which equations of the execution belong to the output system. For instance, consider the input system:

$$\Sigma_i \begin{cases} \alpha = (cons\ 1\ (cons\ 2\ (nil))) \\ \alpha.accu = (nil) \end{cases}$$

If we suppose that the interesting attributes are *rev* and *length*, the interesting output system is:

$$\Sigma_o \begin{cases} \alpha.rev = (cons\ 2\ (cons\ 1\ (nil))) \\ \alpha.length = 2 \end{cases}$$

Let R be a given set of the interesting attributes. The output system of an execution is the set of equations of the form: $\alpha.a = t$, where t is a term with no variable, and $a \in R$.

With such a definition, a program transformation is safe (or correct) if and only if, for all input system, the output systems of the original program and of the transformed one are equal. Thus, additional computations may exist and internal computations may change, but the final results have to remain identical.

6. Transformations

6.1. Partial Evaluation

Applying deduction rules and collecting the new equations produced stands for a kind of partial evaluation. For instance, suppose that we have the following program:

$$\begin{array}{l} test \rightarrow \\ \quad \alpha.r = (+\ \alpha.1\ \alpha.L_9.result) \\ \quad \alpha.L_9 = (factorial\ 3) \end{array}$$

Then from the initial system $x = (test\ x.1)$ it is possible to obtain the following equation:

$$x.r = (+\ x.1\ 6)$$

This equation can be generalized on the variable x since we only use the fact that $x = (test \ldots)$. Thus, a new property on the constructor *test* can be added, and finally we obtain the new program:

$$\begin{array}{l} test \rightarrow \\ \quad \alpha.r = (+\ \alpha.1\ \alpha.L_9.r) \\ \quad \alpha.r = (+\ \alpha.1\ 6) \\ \quad \alpha.L_9 = (factorial\ 3) \end{array}$$

Now, there exists two properties associated to the variable $\alpha.r$ for the constructor *test*. The two properties are correct according to section 5. The proof of such a correction comes from two ideas. Firstly, the property $\alpha.r = (+\ \alpha.1\ 6)$ only comes from the original program. Secondly, adding this new equation does not modify the execution of the original program, but some equations will be deduced with less applications of ψ.

Actually, partial evaluation is the real kernel of the other transformations we define in this paper.

6.2. Reduction

In a program, there are often several properties for a unique variable. In the previous example *test*, there are two properties for the variable $\alpha.r$ (the original and the generated one). In this case, it is interesting to eliminate the first one which involves too much other equations to be computed. To get benefit from a program transformation, many properties must be eliminated.

It is not always possible to eliminate a property. More precisely, an elimination will be safe if and only if it never produces undefined variables during an execution.

In most cases, many solutions exist and we have to choose an efficient one. Reaching optimality is a very difficult problem. However there are simple and intuitive heuristics (which were implemented in our prototype) to obtain reasonable results. In the previous example *test* the reduction leads to:

$$\begin{array}{l} \mathit{test} \rightarrow \\ \quad \alpha.r = (+\ \alpha.1\ 6) \end{array}$$

6.3. Specialization

With functional notations, this transformation is defined as follows: suppose that f is a function of n parameters $x_1 \ldots x_n$, the specialization of f when the parameter x_1 is equal to the constant K is the new function h defined by:

$$(h\ x_2 \ldots x_n) = (f\ K\ x_2 \ldots x_n)$$

This is the first step of the transformation, where a new definition is introduced. The second step of the transformation consists in recognizing where f can be replaced by h. More precisely, it consists in the following term-replacement everywhere in the program:

$$(f\ K\ t_1 \ldots t_{n-1}) \Rightarrow (h\ t_1 \ldots t_{n-1})$$

These two steps can be translated into equational semantics in a systematic way. For the first step, a new attribute is introduced for the computation of h and new attributes are introduced for its parameters. Additional properties are automatically generated in order to link the new attributes to the old ones. For the second step, a new deduction rule is added to the basic kernel, which simply translates the old attributes into the new ones whenever it is possible.

For instance, consider the example of mapping the function factorial to a list. Let *mapf* be the new attribute that computes this specialization of *map*. Since the attribute *map* is defined on the constructors *cons* and *nil*, the properties verified by *mapf* must be reported on these two constructors. The additional program corresponding to the **first step** is then:

$$\begin{array}{l} (\forall c \in \{\mathit{cons}, \mathit{nil}\}) \\ c \rightarrow \\ \quad \alpha.\mathit{mapf} = \alpha.L_m.\mathit{map} \\ \quad \alpha.L_m.f = (\mathit{fact\_ho}) \\ \quad \alpha.L_m = \alpha \end{array}$$

The local variable $\alpha.L_m$ must be fresh for each additional program, that is, not already used. The **second step** automatically produces the new following deduction rule:

$$\begin{array}{l} \varphi_{spe}(\Sigma) = \{x.\mathit{map} = x.\mathit{mapf} \mid \\ \quad x.f = (\mathit{fact\_ho}) \in \Sigma\} \end{array}$$

At this point, the specialization of the attribute *map* in the special case where f is equal to (*fact_ho*) is done and safe. The interesting point is now that partial evaluation and reduction will get benefit from the introduction of these new attributes, properties and deduction rules. For instance, let us describe how simplifications occur for the constructors *cons*. We only report some equations produced by partial evaluation and related to this specialization :

$x = (cons\ x.1\ x.2)$	
$x.mapf = x.L_m.map$	$[prog]$
$x.L_m = (cons\ x.1\ x.2)$	$[prog, subst]$
$x.L_m.f = (fact\_ho)$	$[prog]$
$x.L_m.map =$ $(cons\ x.L_m.L_4.call\ x.L_m.2.map)$	$[prog]$
$x.L_m.L_4.arg = x.1$	$[prog, \ldots]$
$x.L_m.L_4 = (fact\_ho)$	$[\ldots]$
$x.L_m.L_4.call = x.L_m.L_4.L_3.r$	$[prog, \ldots]$
$x.L_m.L_4.L_3 = (factorial\ x.1)$	$[\ldots]$
$x.L_m.2.f = (fact\_ho)$	$[prog, \ldots]$
$x.L_m.2.map = x.L_m.2.mapf$	$[spe]$

The two last blocks show how the constant *fact_ho* is propagated, and how the *map* attribute is transformed into *mapf*. After generalization and reduction, the following properties are generated for the constructors *cons* and *nil* :

$$
\begin{array}{l}
cons \rightarrow \\
\quad \alpha.mapf = (cons\ \alpha.L_{10}.r\ \alpha.2.mapf) \\
\quad \alpha.L_{10} = (factorial\ x.1) \\
nil \rightarrow \\
\quad \alpha.mapf = (nil)
\end{array}
$$

The new local variable $\alpha.L_{10}$ has been introduced to rename (safely) the local variable $\alpha.L_m.L_4.L_3$.

6.4. Deforestation

In functional terms, this transformation occurs when functions are composed. Basically, the problem involves two functions : f with parameters $x_1 \ldots x_n$ and g with parameters $y_1 \ldots y_m$. If f and g are composed, for instance through the first parameter of f, a new function h is defined :

$$(h\ y_1 \ldots y_m\ x_2 \ldots x_n) = (f\ (g\ y_1 \ldots y_m)\ x_2 \ldots x_n)$$

This is the first step of the transformation, where a new definition is introduced. The second step of the transformation consists in recognizing when f is composed with g and then in replacing such a composition by a call to h. More precisely, it consists in the following term-replacement everywhere in the program :

$$(f\ (g\ s_1 \ldots s_m)\ t_1 \ldots t_{n-1}) \Rightarrow (h\ s_1 \ldots s_m\ t_1 \ldots t_{n-1})$$

From an equational semantics point of view, this transformation is performed in two steps as like as for specialization. In the first step, we introduce a new attribute for h and new attributes for its $(m+n-1)$ parameters. New properties (a new program) are also automatically generated to link the new attributes to the old ones. For the second step, a new deduction rule is added to the execution kernel, which simply translates the old attributes into the new ones.

As a preliminary remark, a composition is detected in equational semantics when the variable $x.b$ is used while the equation or property $x = y.a$ holds. In such a case, the composed attributes are a and b.

However, there are actually two kinds of deforestation. In the first kind, named *upward deforestation*, the attribute a is the result of a computation. In the second kind, named *downward deforestation*, the attribute a is a parameter of a computation.

We choose an example which involves these two kinds of deforestation: the reversion of the reversion of a list. For this purpose, the following program is specified:

$$
\begin{array}{l}
foo \rightarrow \\
\quad \alpha.r = \alpha.L_{11}.rev \\
\quad \alpha.L_{11}.accu = (nil) \\
\quad \alpha.L_{11} = \alpha.1.rev \\
\quad \alpha.1.accu = (nil)
\end{array}
$$

We present now the two steps of the two kinds of the deforestation transformations.

Upward Deforestation: The composed attributes are *rev* and *rev*. We denote by r_2 the attribute for the result of the composition, and by a_1 and a_2 the two attributes needed for the two accumulators of *rev* and *rev*. The **first step** defining these new attributes corresponds to the following program:

$$
\begin{array}{l}
[\text{for c} = cons \text{ and c} = nil] \\
c \rightarrow \\
\quad \alpha.r_2 = \alpha.L_p.rev \\
\quad \alpha.L_p.accu = \alpha.a_1 \\
\quad \alpha.L_p = \alpha.L_q.rev \\
\quad \alpha.L_q.accu = \alpha.a_2 \\
\quad \alpha.L_q = \alpha
\end{array}
$$

where L_p and L_q are fresh. This requirement is important to safely add these properties to the original program. The **second step** produces automatically the new following deduction rule which detects where r_2 could replace a composition:

$$
\begin{array}{l}
\varphi_{defo\_up}(\Sigma) = \{ \\
\quad x.rev = x.L_m.r_2 \\
\quad x.L_m.a_1 = x.accu \\
\quad x.L_m.a_2 = y.accu \\
\quad x.L_m = y \\
\quad | \quad x = y.rev \in \Sigma \\
\}
\end{array}
$$

where L_m is a fresh variable for each application of the deduction rule. The deforestation definition is done and safe. Now, partial evaluation and reduction will perform the expected simplifications. For instance, for the constructor *foo*, the following equations are deduced:

$x = (foo\ x.1)$	
$x.r = x.L_{11}.rev$	[*prog*]
$x.L_{11} = x.2.rev$	[*prog*]
$x.L_{11}.accu = (nil)$	[*prog*]
$x.2.accu = (nil)$	[*prog*]
$x.L_{11}.rev = x.L_{11}.L_m.r_2$	[*defo_up*]
$x.L_{11}.L_m.a_1 = x.L_{11}.accu$	[*defo_up*]
$x.L_{11}.L_m.a_2 = x.2.accu$	[*defo_up*]
$x.L_{11}.L_m = x.2$	[*defo_up*]

After generalization and reduction, the following properties are obtained :

$$
\begin{array}{l}
\textit{foo} \rightarrow \\
\quad \alpha.r = \alpha.1.r_2 \\
\quad \alpha.1.a_1 = (\textit{nil}) \\
\quad \alpha.1.a_2 = (\textit{nil})
\end{array}
$$

In the same way, for the constructors *cons* and *nil* we obtain :

$$
\begin{array}{l}
\textit{cons} \rightarrow \\
\quad \alpha.r_2 = \alpha.2.r_2 \\
\quad \alpha.2.a_1 = \alpha.a_1 \\
\quad \alpha.2.a_2 = (\textit{cons}\ \alpha.1\ \alpha.a_2) \\
\textit{nil} \rightarrow \\
\quad \alpha.r_2 = \alpha.L_{12}.\textit{rev} \\
\quad \alpha.L_{12}.\textit{accu} = \alpha.a_1 \\
\quad \alpha.L_{12} = \alpha.a_2
\end{array}
$$

Downward Deforestation: After the deforestation above, the second kind of deforestation appears on the constructor *nil*. The composed attributes are a_2 and *rev*, where a_2 is a parameter-attribute instead of a result-attribute. Such a deforestation through accumulative parameters is known to be difficult [2], but is naturally handled in equational semantics.

Let r_3 be the new attribute introduced for the result of the composition, and a_3 the new attribute introduced for the related accumulative parameter. The **first step** still consists in the automatic generation of the program which defines these attributes : everywhere the attribute a_2 is computed, the attribute r_3 must be equal to *rev* on a_2 with *accu* being equal to a_3. In the example, a_2 is computed on $\alpha.2$ for the constructor *cons*, and on $\alpha.1$ for the constructor *foo*. So the first step corresponds to the following additional program :

$$
\begin{array}{l}
\textit{cons} \rightarrow \\
\quad \alpha.2.r_3 = \alpha.L_m.\textit{rev} \\
\quad \alpha.L_m.\textit{accu} = \alpha.2.a_3 \\
\quad \alpha.L_m = \alpha.2.a_2 \\
\textit{foo} \rightarrow \\
\quad \alpha.1.r_3 = \alpha.L_p.\textit{rev} \\
\quad \alpha.L_p.\textit{accu} = \alpha.1.a_3 \\
\quad \alpha.L_p = \alpha.1.a_2
\end{array}
$$

where L_m and L_p are fresh local variables. The **second step** of the transformation is the automatic generation of the following deduction rule which detects where r_3 could replace a composition :

$$
\begin{array}{l}
\varphi_{\textit{defo\_down}}(\Sigma) = \{ \\
\quad x.\textit{rev} = y.r_3 \\
\quad y.a_3 = x.\textit{accu} \\
\quad \mid x = y.a_2 \\
\}
\end{array}
$$

Multiple applications of this deduction rule on the same variable y is not allowed. This technical point is not explained here since it is too specific to this kind of deforestation. After partial evaluation and reduction, the following program is obtained :

$$
\begin{array}{l}
\mathit{foo} \rightarrow \\
\quad \alpha.r = \alpha.1.r_2 \\
\quad \alpha.1.r_3 = \alpha.1.a_3 \\
\quad \alpha.1.a_1 = (\mathit{nil}) \\
\mathit{cons} \rightarrow \\
\quad \alpha.r_2 = \alpha.2.r_2 \\
\quad \alpha.2.r_3 = \alpha.r_3 \\
\quad \alpha.a_3 = (\mathit{cons}\ \alpha.1\ \alpha.2.a_3) \\
\quad \alpha.2.a_1 = \alpha.a_1 \\
\mathit{nil} \rightarrow \\
\quad \alpha.r_2 = \alpha.r_3 \\
\quad \alpha.a_3 = \alpha.a_1
\end{array}
$$

Notice that the deforestation really succeed since only one list is constructed. Moreover, the result is a copy of the first list, as it is expected to: in fact r_2 is always equal to r_3, and a_3 appends a_1 (initialized to *nil*) to the end of the list.

6.5. Elimination of Identity

Consider the properties about r_2 and a_3 on the constructor *nil*. They are both equalities. The elimination of identity try to prove whether these equalities are verified for all constructors or not. The transformation is performed in two steps. First, the equality is automatically proved or refuted by induction. Second, for the proved equalities, a new deduction rule is automatically defined.

In the example below, the induction proof on the constructor *cons* consists in assuming the properties on variable $\alpha.2$, and prove them on variable α. The proof will be automatically performed by partial evaluation. Assuming the induction hypothesis on $\alpha.2$ corresponds to the following system:

$$
\begin{array}{l}
x = (\mathit{cons}\ x.1\ x.2) \\
x.2.r_2 = x.2.r_3 \\
x.2.a_3 = x.2.a_1
\end{array}
$$

The partial evaluation produces the following execution:

$$
\begin{array}{lr}
x.r_2 = x.2.r_2 & [\mathit{prog}] \\
x.2.r_3 = x.r_3 & [\mathit{prog}] \\
x.a_3 = (\mathit{cons}\ x.1\ x.2.a_3) & [\mathit{prog}] \\
x.2.a_1 = x.a_1 & [\mathit{prog}] \\
\hline
x.r_2 = x.2.r_3 & [\mathit{subst}] \\
x.a_3 = (\mathit{cons}\ x.1\ x.2.a_1) & [\mathit{subst}] \\
\hline
x.r_2 = x.r_3 & [\mathit{subst}] \\
x.a_3 = (\mathit{cons}\ x.1\ x.a_1) & [\mathit{subst}]
\end{array}
$$

The inductive hypothesis is verified for the equality $\alpha.r_2 = \alpha.r_3$, but the other equality is not verified. So, for the second step of the transformation there is only one new deduction rule defined:

$$\varphi_{id}(\Sigma) = \{x.r_2 = x.r_3 \mid x \textit{ appears in } \Sigma\}$$

After partial evaluation, we obtain the following program:

$$
\begin{array}{l}
\textit{foo} \rightarrow \\
\quad \alpha.r = \alpha.1.a_3 \\
\quad \alpha.a_1 = (\textit{nil}) \\
\textit{cons} \rightarrow \\
\quad \alpha.a_3 = (\textit{cons}\ \alpha.1\ \alpha.2.a_3) \\
\quad \alpha.2.a_1 = \alpha.a_1 \\
\textit{nil} \rightarrow \\
\quad \alpha.a_3 = \alpha.a_1
\end{array}
$$

Now, we have succeed in proving automatically that reverse composed with itself is equal to the function copy, which duplicates its input list.

7. Conclusion

This work comes from a large comparative study of various existing methods to perform deforestation and partial evaluation in various programming paradigm. Historically, we compared [3, 2, 1] the deforestation of attribute grammars [5, 6, 12], the Wadler deforestation [16, 13, 7] in functional programming, many works about folds [4, 9] and hylomorphisms [10, 15]. In each of these formalisms, there were many interesting ideas. But they were sometimes restricted to one particular class of algorithms but sometimes more powerful than another method on the same class. However, attribute grammars seems to provide a kind of declarative notation able to gather all of them in an homogeneous way.

Actually, we think that the key of our approach is to define a program only by the set of the properties it verifies. Functions, procedures, data types, control statements of real programming languages are here considered as syntactic sugar to define properties as equations. In this context, Equational Semantics is a minimal but powerful framework to manipulate these properties and translate them back into a more efficient program.

Bibliography

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

[2] Etienne Duris. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle*. PhD thesis, Université d'Orléans, 1998.

[3] Etienne Duris, Didier Parigot, Gilles Roussel, and Martin Jourdan. Structure-directed genericity in functional programming and attribute grammars. Rapport de Recherche 3105, INRIA, February 1997.

[4] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*, pages 21–32, Orlando, Florida, June 1994.

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

[6] Robert Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

[7] G. W. Hamilton. Higher order deforestation. 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 122–136, Aachen, September 1996. Springer-Verlag.

[8] A. Kühnemann. *Berechnungsstärken von Teilklassen primitiv-rekursiver Programmschemata.* PhD thesis, Technical University of Dresden, 1997. Shaker Verlag, Aachen.

[9] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cata's from recursive definitions. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 314–323, La Jolla, CA, USA, 1995. ACM Press.

[10] 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.

[11] Didier Parigot, Gilles Roussel, Martin Jourdan, and Etienne Duris. Dynamic Attribute Grammars. 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 122–136, Aachen, September 1996. Springer-Verlag.

[12] 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.

[13] Morten Heine Sørensen. A grammar-based data-flow analysis to stop deforestation. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 1994.

[14] S. Doaitse Swierstra and Harald H. Vogt. Higher Order Attribute Grammars. In Henk Alblas and Bořivoj Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lect. Notes in Comp. Sci.*, pages 256–296, New York–Heidelberg–Berlin, June 1991. Springer-Verlag.

[15] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. on Func. Prog. Languages and Computer Architecture*, pages 306–313, La Jolla, CA, USA, 1995. ACM Press.

[16] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lect. Notes in Comp. Sci.*, pages 344–358, Nancy, March 1988. Springer-Verlag.