Equational Semantics

Loïc Correnson, Etienne Duris
Didier Parigot, Gilles Roussel

INRIA-Rocquencourt - Domaine de Voluceau
Loic.Correnson@inria.fr

Abstract. In the context of functional programming, semantic methods are commonly used to drive program transformations. However, classical semantic domains often rely on recursive objects which embed the control flow of recursive functions. As a consequence, transformations which have to modify the control flow are difficult to define. We propose in this paper a new semantic domain where the control flow is defined implicitly, and thus can be modified. This new theoretical and practical framework allows to homogeneously define and extend powerful transformations related to partial evaluation and deforestation.
Keywords: semantics, program transformation, partial evaluation, deforestation.

1 Introduction

Many frameworks use some semantic abstraction to transform functional programs. For instance λ-calculus [11, 8], catamorphisms [5], hylomorphisms [7], folds [9]... A more complete bibliography and a large comparison of these different frameworks can be found in [3].

All of them share a similar global structure. Thus, functional programs are abstracted in some mathematical domain. Transformations are performed on the abstraction of the program. Then, the transformed program is obtained by a backward translation from the mathematical domain to functional programs.

For instance, the HYLO system [7] transforms a functional program into hylomorphisms and then perform partial evaluation and sometimes deforestation, thanks to many theorems (acid rain theorem, fusion law... [6]). Then, these new hylomorphisms could be translated back into functional programs.

However, all frameworks we know share a surprising constraint: the abstraction of functional programs always relies on "functional" objects where recursive structures or schemes are strongly preserved and can not be easily modified. For instance, with λ-calculus, the recursive calls are defined *in extenso* in the structure of the λ-terms. With hylomorphisms (and folds), these recursion schemes are exactly pointed out by functors which are used as transformation parameters. Thus, a transformation can not freely restructure these recursive schemes.

We propose in this paper a new abstraction for functional programs, which does not rely on such "functional" objects. In our semantics, the control flow is neither defined nor fixed. A related operational semantics explicitly defines recursive schemes and control flow. This latter semantics can be computed *from* the former and allows to perform backward translations into functional programs.

This paper presents an homogeneous framework to define and extend classical program transformations related to partial evaluation. Especially, the system is quite powerful at transforming control flow and recursive schemes of functional programs.

The paper is structured as follows. Section 2 fixes notations for functional programs and their semantics. In section 3, we introduce our mathematical objects with an example, and then we precisely define them. They are named *Equational Programs* and their *Equational Semantics* is defined in section 4. Obtained results and powerful transformations are then presented in section 5. The end of the paper is about technical bases for all this framework: translation from functional programs is given in section 6, operational semantics is presented in section 7 and backward translation from equational programs into functional ones is presented in section 8.

Notations: we will assume standard definitions for sets and relations. We will also make use of the following notations:

- when a set S is the singleton $\{s\}$, it is also denoted by s without brackets.
- $\mathcal{R} = [\mathcal{R}_1; \mathcal{R}_2; \ldots; \mathcal{R}_n]$ is the relation defined by: $a\mathcal{R}b \Leftrightarrow a\mathcal{R}_1 a_1 \mathcal{R}_2 \ldots \mathcal{R}_n b$.
- $\mathcal{R}^n = [\mathcal{R}; \ldots; \mathcal{R}]$ (with n occurrences of $\mathcal{R}$).
- $\mathcal{R}^*$ is the transitive closure of $\mathcal{R}$.
- $\mathcal{R}_1 + \mathcal{R}_2$ is the relation $\mathcal{R}$ defined by: $a\mathcal{R}b \Leftrightarrow a\mathcal{R}_1 b$ or $a\mathcal{R}_2 b$.

2 Functional Programs

In this section, we fix the functional programs we consider. In few words, it is a standard functional language with higher-order and pattern-matching. Nevertheless, we will only consider well-typed programs, regardless of which kind of type system is used. Programs are defined according to the BNF definition in Figure 1-left-top.

In this definition, the ($\sharp$) symbol means "arity of". Type-constructors of user-defined data types are denoted by c (for instance, *cons*, *nil*, etc.). Primitive values and operations (integers, etc.) are denoted by π. Notice that this grammar is sufficient to define higher-order functions, since partial applications are possible. Expressions like `fun` $x \rightarrow e$ which appear in classical functional programs can be translated into a new function name. For instance, the program

```
let horev x = match x with
  cons a b -> let k = horev b in fun h -> (k (cons a h))
| nil -> fun h -> h
```

is just syntactic sugar for the following one:

$$
\begin{array}{lll}
FP & ::= F \ldots F & \\
F & ::= \texttt{let}\ f\ x_1 \ldots x_n = e & \\
 & \mid\ \texttt{let}\ f\ x_1 \ldots x_n\ = \texttt{fun} & \\
 & \qquad c\ y_1 \ldots y_m \rightarrow e \mid \ \ldots & \\
e & ::= x \mid f \mid (e\ e) & \\
 & \mid\ (c\ e_1 \ldots e_n) & n = \sharp c \\
 & \mid\ (\pi\ e_1 \ldots e_n) & n = \sharp \pi \\
\hline
v & ::= (f_p\ v_1 \ldots v_p) & \\
 & \mid\ (c\ v_1 \ldots v_n) & n = \sharp c \\
 & \mid\ (\pi\ v_1 \ldots v_n) & n = \sharp \pi
\end{array}
$$

- $x \rightarrow_\gamma \gamma(x)$
- $(e_1\ e_2) \rightarrow_\gamma v \Leftrightarrow$
  $e_1 \rightarrow_\gamma (f_p\ v_1 \ldots v_p)$
  $e_2 \rightarrow_\gamma v'$
  $(f_{p+1}\ v_1 \ldots v_p\ v') \rightarrow v$
- $(f_n\ v_1 \ldots v_n) \rightarrow_\gamma v \Leftrightarrow$
  $(\texttt{let}\ f\ x_1 \ldots x_n = e) \in FP$
  $\gamma' = \gamma(x_k : v_k)_{k \leq n}$
  $e \rightarrow_{\gamma'} v$
  or
  $(\texttt{let}\ f\ x_1 \ldots x_{n-1} = \texttt{fun}$
  $c\ y_1 \ldots y_m \rightarrow e_c \mid \ \ldots) \in FP$
  $v_n = (c\ w_1 \ldots w_m)$
  $\gamma' = \gamma(x_k : v_k)_{k<n}(y_i : w_i)_{i \leq m}$
  $e_c \rightarrow_{\gamma'} v$
- $(c\ e_1 \ldots e_n) \rightarrow_\gamma (c\ v_1 \ldots v_n) \Leftrightarrow$
  $(\forall i)\ e_i \rightarrow_\gamma v_i$
- $(\pi\ e_1 \ldots e_n) \rightarrow_\gamma v \Leftrightarrow$
  $(\forall i)\ e_i \rightarrow_\gamma v_i$
  $(\pi\ v_1 \ldots v_n) \triangleright v$

Fig. 1. FP definition and semantics.

```
let horev = fun
  cons a b -> ( (f1 a) (horev b) )
| nil -> f2
```

```
let f1 a k h = (k (cons a h))
let f2 h = h
```

We consider a standard operational semantics for FP, in the call-by-value style. It is defined by a relation $\rightarrow_\gamma$ where γ is an environment which associates variables to values (the empty environment is denoted by ε; the association of x to v in γ is denoted by $\gamma(x : v)$). Values are classically defined in Figure 1-left-bottom.

We denote by f_p the function f partially applied to exactly p arguments, in order to be consistent with further definitions. For primitives, we will suppose that a rewriting rule $\triangleright$ is available, such that for instance $(+\ 1\ 2) \triangleright (3)$. The operational semantics for FP is then defined in Figure 1-right.

3 Equational Programs

3.1 An intuitive presentation

Consider the following program:

```
let length = fun nil -> 0 | cons a b -> (+ 1 (length b))
```

We can have, as an example, the following execution (here, = gives intermediate steps for $\rightarrow_\varepsilon$):

```
length (cons 5 (cons 6 nil)) =
(+ 1 (length (cons 6 nil))) =
(+ 1 (+ 1 (length nil))) =
(+ 1 (+ 1 0)) = 2
```

Now let us denote each list by a variable x, and the result of the function `length` on x by the variable $x.length$ ($length$ will be called an attribute on x). Intuitively, this implies the following equations:

$$\begin{array}{ll} (\forall x)\ x = (cons\ a\ b) \Rightarrow x.length = (+\ 1\ b.length) & (1) \\ (\forall x)\ x = (nil) \Rightarrow x.length = 0 & (2) \end{array}$$

When the variable x is associated to a term like $(cons\ t_1\ t_2)$, we use by convention the variables $x.1$ and $x.2$ to respectively denote the sub-terms t_1 and t_2. In this context, the previous execution could be represented by the following list of statements, where x, $x.2$, and $x.2.2$ could be thought as variable names:

$$\begin{array}{llll} x & = (cons\ 5\ (cons\ 6\ nil)) & x.2.2.length & = 0 \\ x.length & = (+\ 1\ x.2.length) & x.2.length & = (+\ 1\ 0) \\ x.2 & = (cons\ 6\ nil) & x.length & = (+\ 1\ (+\ 1\ 0)) \\ x.2.length & = (+\ 1\ x.2.2.length) & x.length & = 2 \\ x.2.2 & = (nil) & & \end{array}$$

Note that the two equations (1) and (2) are similar to the functional program, and that the above list of statements satisfies them. When a function uses additional parameters, such a comparison is still possible. For instance, the program

```
let rev h = fun nil -> h | cons a b -> rev (cons a h) b
```

is associated to the following equations where, for any list denoted by the variable x, the parameter `h` is denoted by the variable $x.rev\_h$ and the result of the function `rev` is denoted by the variable $x.rev$. Informally, $x.rev$ stands for the result of $(rev\ x.rev\_h\ x)$.

$$\begin{array}{ll} (\forall x)\ x = (cons\ y\ z) & \Rightarrow x.rev = z.rev \\ (\forall x)\ x = (cons\ y\ z) & \Rightarrow z.rev\_h = (cons\ y\ x.rev\_h) \\ (\forall x)\ x = (nil) & \Rightarrow x.rev = x.rev\_h \end{array}$$

Thus, a set of equations seems to be sufficient to describe the values computed by a functional program on list-like structures. For higher-order functions, we have to manage partial applications, namely, expressions like $(e_1\ e_2)$. In such situations, we propose to associate a fresh variable x to the expression e_1. Then the variable $x.arg$ is associated to e_2, and $x.call$ to the (partial) application $(e_1\ e_2)$.

For instance, consider the function definition `let f a b = (+ a b)`. The function `f` could be partially applied, thus we need the following equations related to *arg* and *call*:

$$\begin{array}{ll}(\forall x)\ x = (f_0) & \Rightarrow x.call = (f_1\ x.arg)\\ (\forall x)\ x = (f_1\ y) & \Rightarrow x.call = (+\ y\ x.arg)\end{array}$$

Here, the constructors f_0 and f_1 are used to represent functional values such as ($\mathtt{f}_0$) and ($\mathtt{f}_1\ v_1$), that is, the different partial applications of the function `f`. The equations defining variable $x.call$ from $x.arg$ are then consistent with the definition of $(e_1\ e_2) \rightarrow_\gamma v$ given in Figure 1.

Thus, it seems possible to represent programs by a set of equations. Of course, we have to formalize such a translation, and to prove a semantic equivalence between the two representations. This is the aim of the paper, and we will start with a definition of what is an *equational program*. The section 4 gives its semantics, and we will present in section 6 a translation from functional programs into equational ones.

3.2 Definitions

The following definitions are mutually recursive and must be considered all together.

Terms: Terms are built using *constructors* or *primitives* with variables or sub-terms as parameters. There is no function call. The set of the variables appearing in a term t is denoted by $\mathcal{V}ars(t)$.

Values: a value v is a term which contains no variable, *i.e.* $\mathcal{V}ars(v) = \emptyset$.

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

- a simple name (an identifier).
- $x.k$ (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 the k-th local variable associated to the variable x.

The form $x.L_k$ is just a way to make a new variable name which is associated to the only variable x and could be used as a fresh local variable to name intermediate or dynamic computations (*e.g:* partial applications).

Attributes: There are two sets of attributes, $\mathcal{R}es$ for attributes which represent the results of a computation, and $\mathcal{P}rm$ for those which represent the parameters of a computation. Then, with $p \in \mathcal{P}rm$ and $r \in \mathcal{R}es$, the variable $x.r$ represents the result of computing the attribute r on x when the attribute p is equal to the term represented by $x.p$.

Statements: A statement is an oriented equation of the form $x = t$, where the left-hand-side is restricted to be a variable. A system Σ is a set of statements.

Equations and Program: A program is defined by a set of equations, which are restricted to be of the following form:

$$(\forall x)\ x = (c\ y_1 \dots y_m) \Rightarrow z = t$$

where the statement $z = t$ may refer to x and $y_1 \dots y_m$. Thus, we will use the shortcut notation $c \to z = t$ where the variable x is replaced by the special identifier α and $y_1 \dots y_m$ by $\alpha.1 \dots \alpha.m$. For instance, the equational program associated with the function `rev` is:

$$\begin{array}{ll} \mathit{cons} & \to \alpha.\mathit{rev} = \alpha.2.\mathit{rev} \\ \mathit{cons} & \to \alpha.2.\mathit{rev\_h} = (\mathit{cons}\ \alpha.1\ \alpha.\mathit{rev\_h}) \\ \mathit{nil} & \to \alpha.\mathit{rev} = \alpha.\mathit{rev\_h} \end{array}$$

These definitions are summarized in figure 2.

$$\begin{array}{lll} \mathcal{P} & ::= (c \to \mathit{stmt})^* & \\ \mathit{stmt} & ::= x = t & \\ x & ::= \alpha \mid x.k \mid x.a \mid x.L_k & \\ t & ::= x & \\ & \quad \mid\ (c\ t_1 \dots t_n) & n = \sharp c \\ & \quad \mid\ (\pi\ t_1 \dots t_n) & n = \sharp \pi \\ \Sigma & ::= \mathit{stmt}^* & \end{array}$$

Fig. 2. Equational Programs

4 Equational Semantics

We define here a semantics for a given equational program $\mathcal{P}$. The intuitive idea consists in computing output-statements Σ_{Out} from input-statements Σ_{In}, by adding new statements such that the equations of the program $\mathcal{P}$ remain satisfied.

4.1 Substitutions

Two kinds of substitutions are involved here. The first one, denoted by $[x := t]$, replaces each whole occurrence of the variable x with the term t. Thus, we have: $(+\ 1\ x)[x := t] \equiv (+\ 1\ t)$, but $(+\ 1\ x.a)[x := t] \equiv (+\ 1\ x.a)$ since $x.a$ is not a whole occurrence of x.

The second kind of substitution, denoted by $[x]$, replaces the special identifier α by x everywhere it appears, even inside variables. Thus, we have $(+\ 1\ \alpha)[x] \equiv (+\ 1\ x)$, and $(+\ 1\ \alpha.a)[x] \equiv (+\ 1\ x.a)$.

4.2 Derivations

A step is a relation denoted by $\rightarrow_{\mathcal{P}}$, such that $\Sigma \rightarrow_{\mathcal{P}} \Sigma'$ holds if and only if one of the following rules holds, (for some x, c, etc.):

- the *sub-term rule*, which deals with sub-term variables, holds when $x = (c\ t_1...t_n) \in \Sigma$ and $\Sigma' = \Sigma \cup \{x.k = t_k\}$.
- the *substitution rule* holds when $x = t \in \Sigma$, $y = t' \in \Sigma$ and $\Sigma' = \Sigma \cup \{x = t[y := t']\}$.
- primitive operations are handled by the *primitive rule* which holds when $x = t \in \Sigma$, $t \triangleright t'$ and $\Sigma' = \Sigma \cup \{x = t'\}$.
- Finally, the *instantiation rule* deals with applying an equation of the program $\mathcal{P}$. This rule holds when $x = (c\, t_1...t_n) \in \Sigma$ and when there exists an equation of the form $c \rightarrow y = t$ in $\mathcal{P}$, and when $\Sigma' = \Sigma \cup \{y[x] = t[x]\}$. In this special case, the fact $\Sigma \rightarrow_{\mathcal{P}} \Sigma'$ is also denoted by $\Sigma \overset{y=t,x}{\rightarrow} \Sigma'$ when the instantiated equation should be pointed out.

Remark that if the *instantiation rule* is not used, the relation $\rightarrow_{\mathcal{P}}$ could be replaced by $\rightarrow_{\emptyset}$. Now, we define the relation $\Rightarrow_{\mathcal{P}}$ by:

$$\Sigma \Rightarrow_{\mathcal{P}} \Sigma' \;\Leftrightarrow\; \Sigma \rightarrow^*_{\mathcal{P}} \Sigma'' \text{ and } \Sigma' \subset \Sigma''$$

Intuitively, $\Sigma \Rightarrow_{\mathcal{P}} \Sigma'$ means that there exists a derivation from Σ which produces at least the equations Σ'. The relation $\Rightarrow_{\mathcal{P}}$ benefits from many properties. For instance, the following theorem holds[1]:

Theorem 41 *$\Rightarrow_{\mathcal{P}}$ is monotonic, that is:*
if $(\forall i)\ \Sigma_i \Rightarrow_{\mathcal{P}} \Sigma'_i$, then $(\bigcup_i \Sigma_i) \Rightarrow_{\mathcal{P}} (\bigcup_i \Sigma'_i)$.
As a direct consequence, $\Rightarrow_{\mathcal{P}}$ is confluent.
We have also: $\Sigma \Rightarrow_{\mathcal{P}} \Sigma'$ if and only if $(\forall x)\ \Sigma[x] \Rightarrow_{\mathcal{P}} \Sigma'[x]$.

4.3 Semantics

We are interested in using equational programs to perform program transformations. So we need a semantics which only consider what *are* the values computed by a program, not *how* they are computed. Consider the system $\Sigma_{In} = \{\alpha = v\}$, where v is a value (*ie.* a term with no variable). Any derivation from Σ_{In} is a trace of an execution of the program $\mathcal{P}$ on the value v. The resulting values are statements of the form $\alpha.r = v_r$ where $r \in \mathcal{R}es$. An interesting semantics associated to $\mathcal{P}$ should not consider any complete derivation, but only the values v and v_r.

More precisely, the semantics of an equational program $\mathcal{P}$ is defined according to a pair (P, R), where P is sequence of parameter attributes, and R a sequence of result ones. If $P = \{p_1 \dots p_n\} \subset \mathcal{P}rm$ and $R = \{r_1 \dots r_m\} \subset \mathcal{R}es$, the semantics of $\mathcal{P}$ according to (P, R) is denoted by $[\![\mathcal{P}]\!]_{P,R}$ and is the relation between tuples of values defined by:

[1] Note that $\Sigma[x] = \{y[x] = t[x] \mid y = t \in \Sigma\}$.

$$(v, v_1 \dots v_n)[\![\mathcal{P}]\!]_{P,R}(w_1 \dots w_m) \quad \Leftrightarrow \quad \begin{cases} \alpha = v \\ \alpha.p_1 = v_1 \\ \dots \\ \alpha.p_n = v_n \end{cases} \Rightarrow_{\mathcal{P}} \begin{cases} \alpha.r_1 = w_1 \\ \dots \\ \alpha.r_m = w_m \end{cases}$$

Thus two programs $\mathcal{P}$ and $\mathcal{P}'$ are equivalent if and only if their semantics are equal (using the standard equality on relations) for all pairs (P, R). With such a definition, if $\mathcal{P}$ and $\mathcal{P}'$ are equivalent, they may use completely different derivations, but they must compute the same values.

5 Results

As presented in Section 3, there is a translation from functional programs to equational ones, which is correct with respect to both functional and equational semantics. Moreover, the inverse translation exists, and is also correct. The first one is defined in section 6, and the second one is defined in sections 7 and 8.

The translation from an equational program $\mathcal{P}$ into a functional program *FP* computes the functions that could be defined according to the equations in $\mathcal{P}$. Thus, transformations are no more restricted by any fixed recursion scheme. Moreover, it is possible to freely add new equations to $\mathcal{P}$ if they are consistent with its semantics. These additions may not directly participate to any "function" recursion. The translation from $\mathcal{P}$ into *FP* will decide which equations of $\mathcal{P}$ have to be taken into account to define functions.

Actually, all of the transformations presented here, and many others, have been implemented in a completely systematic transformation system. Our implementation takes a functional program, translates it in an equational one, transforms it, and then produces back a new functional program.

5.1 Partial Evaluation

Thanks to the theorem 41, partial evaluation is easy to define. Consider the following relation: $\{\alpha = (c\ \alpha.1 ... \alpha.n)\} \Rightarrow_{\mathcal{P}} \Sigma$.

Then, $\Sigma[x]$ is a set of statements that can be deduced from any system containing an equation of the form $x = (c\ \dots)$. Then it is possible to prove that adding the statements $\{c \to x = t | x = t \in \Sigma\}$ to $\mathcal{P}$ is consistent with its semantics. Now, computing the operational semantics for $\mathcal{P}$ will automatically get benefit from these new equations. The final result obtained is a partial evaluation of $\mathcal{P}$. For this method of partial evaluation, the only problem of termination comes from functional programs that infinitely loop.

5.2 Approximative dependences

The operational semantics of an equational program $\mathcal{P}$ points out some dependences between parameter attributes and result ones. Thus, in the relation

$[\![\mathcal{P}]\!]_{P,R}$, the attributes of R depend on those of P. Fortunately, it is easy to compute an approximation of these dependences, denoted $\mathcal{D}ep$. We expect that if the result attribute r may depend on the parameter attribute p, then $p \in \mathcal{D}ep(r)$. This approximation is computed by looking for every equation which may participate to the computation of the attribute r, and by collecting every parameter attributes involved. This analysis will be very useful for further transformations.

5.3 Specialization

Specialization of a function `f` is, for example, a new function `g` such that `g x = (f K x)` where `K` is a constant (a value). Sometimes, introducing such a function `g` allows to perform simplifications. In equational programs, a specialization is defined in two parts. Let p be a parameter attribute and K a constant, a new attribute r' is defined for every result attribute r such that $p \in \mathcal{D}ep(r)$. As the first step, the definition of $\Sigma \rightarrow_{\mathcal{P}} \Sigma'$ is extended by the rule where $\Sigma' = \Sigma \cup \{x.r = x.r'\}$ if $x.p = K \in \Sigma$. As the second step, we add new equations for each constructor c, namely:

$$\begin{array}{ll} c \rightarrow \alpha.r' = \alpha.L_m.r & \\ c \rightarrow \alpha.L_m.p' = \alpha.p & (\forall p' \in \mathcal{D}ep(r) - \{p\}) \\ c \rightarrow \alpha.L_m.p = K & \end{array}$$

The local variable $\alpha.L_m$ is supposed to be fresh. The specialization is then automatically performed by partial evaluation.

5.4 Deforestation

Deforestation is an extension of the specialization dealing with function compositions. In functional terms, it consists in defining a function `h` such that `h x = f (g x)`. There are well known methods to simplify function compositions, but they are not powerful enough, especially in the presence of additional parameters. In most cases, the problems come from the difficulty to change the recursion scheme of a function in the context of standard semantics for functional programs. In the context of equational programs, recursive schemes are *computed* from equations, so the problem is simpler. Actually, the composition of two attributes can be defined in a way similar to specialization, by introducing new attributes and by extending the relation $\rightarrow_{\mathcal{P}}$. The deforestation works well, even through parameters, as illustrated by the following examples.

The definition of deforestation depends on the kind – result or parameter – of the involved attributes. More details about these transformations could be found in [2].

Result-deforestation: Suppose that r and s are two result attributes, with $\mathcal{D}ep(r) = \{p_1 \dots p_n\}$ and $\mathcal{D}ep(s) = \{q_1 \dots q_m\}$. When the attribute s is computed on the result given by r, we say that there is a composition of r and s. It would be also defined by a new result attribute r' and from the new parameter attributes $q'_1 \dots q'_m$ to avoid any name-clash. Thus, we can extend the relation $\Sigma \rightarrow_{\mathcal{P}} \Sigma'$ by a new rule where $\Sigma' = \Sigma \cup \Sigma_+$ if $x = y.r \in \Sigma$. Σ_+ is defined in the sequel. The new attributes involve new equations, for each constructor c involved by the computation of r:

$$\begin{array}{l} c \to \alpha.r' = \alpha.L_{loc}.s \\ c \to \alpha.L_{loc}.q_j = \alpha.q'_j \\ c \to \alpha.L_{loc} = \alpha.r \end{array} \qquad \Sigma_+ = \begin{cases} x.s = y.r' \\ y.q'_j = x.q_j \end{cases}$$

The local variable $\alpha.L_{loc}$ is supposed to be fresh.

Parameter-deforestation: To deforest through parameters, the solution is similar. Suppose the result attribute r is computed on a parameter attribute p, with $\mathcal{D}ep(r) = \{p_1 \dots p_n\}$. Then the new result attributes are $r'_1 \dots r'_n$, and the new parameter attribute is p'. Then, we extend the relation $\Sigma \to_{\mathcal{P}} \Sigma'$ by a new rule where $\Sigma' = \Sigma \cup \Sigma_+$ if $x = y.p \in \Sigma$. Σ_+ is defined in the sequel. The new attributes involve new equations, for each constructor c where a variable $x.p$ is involved:

$$\Sigma_+ = \begin{cases} x.r = y.p' \\ y.r'_i = x.p_i \end{cases} \qquad \begin{array}{l} c \to x.p' = x.L_{loc}.r \\ c \to x.L_{loc}.p_i = x.r'_i \\ c \to x.L_{loc} = x.p \end{array}$$

The local variable $x.L_{loc}$ is supposed to be fresh.

5.5 Examples

All these examples come from the implementation of our system. It is available on the web[2].

Reversed flatten: the function `f` given in Figure 3-left takes a binary tree, flattens its leaves, and then reverses the obtained list. After four steps of deforestation, the program in figure 3-right is obtained. One can observe that it is a variant of the function `flat` where the tree is flattened in the reversed direction. So, our analysis and deforestation methods are able to completely modify the control flow of a recursive function.

Inefficient composition: Figure 4-left presents the function `append` which appends two lists, and the function `f` which appends three lists. Actually, the expression `(append (append x y) z)` should be translated into `(append x (append y z))` to avoid one duplication of each list `x` and `y`. Deforestation performs the transformation automatically as shown in Figure 4-right.

Removing continuations: As a last example, we transform the reverse function written with a continuation, given in Figure 5-left. The data deforested is the continuation. The result in Figure 5-right is equal to the standard function `rev` with accumulator. This result shows the power of dealing with a system which does not include function calls. In equational semantics, functional values are encoded like other values, and thus, they could be treated in a same way. Here , the elimination of the continuation is performed by the *standard* deforestation for equational programs.

[2] `http://www-rocq.inria.fr/~correnso/agdoc/`

`let flat x h = match x with`
`  node a b -> flat a (flat b h)`
`| leaf n -> cons n h`
`let flatten x = flat x nil`
 	
`let f x = reverse (flatten x)`
	`let f t_27 = ((fpfun_1 t_27) nil)`
	
	`let fpfun_1 t_42 t_43 =`
	`  match t_42 with`
	`  | node t_44 t_45 ->`
	`    ((fpfun_1 t_45)`
	`      ((fpfun_1 t_44) t_43))`
	`  | leaf t_51 ->`
	`    ((cons t_51) t_43)`

Fig. 3. flatten and reverse

`let append x y = match x with`
`  cons a b -> cons a (append b y)`
`| nil -> y`
`let f x y z =`
`  (append (append x y) z)`
	`let fpfun_2 t_38 t_39 =`
	`  match t_38 with`
	`  | nil -> t_39`
	`  | cons t_41 t_42 ->`
	`    ((cons t_41)`
	`      ((fpfun_2 t_42) t_39))`
	
	`let f t_16 t_17 t_15 =`
	`  ((fpfun_2 t_16)`
	`    ((append t_17) t_15))`

Fig. 4. composition with append

`let revho x = match x with`
`  cons a b ->`
`    let k = (revho b) in`
`    (fun h -> k (cons a h))`
`| nil -> (fun h -> h)`
 	
`let reverse x = ((revho x) nil)`
	`let fpfun_1 t_11 t_12 =`
	`  match t_11 with`
	`  | nil -> t_12`
	`  | cons t_14 t_15 ->`
	`    ((fpfun_1 t_15)`
	`      ((cons t_14) t_12))`
	
	`let reverse t_3 =`
	`  ((fpfun_1 t_3) nil)`

Fig. 5. reverse with higher order

6 Translation

In this section, we will see how to translate a functional program into an equational program. This translation works by a simple encoding of the operational semantics of functional programs given in section 2.

Constructors and primitives are directly translated from FP to equational notations. But since there is no function in equational programs, we have to define a new constructor for each partially applied function. Thus, the functional value $(f_p\ v_1 ... v_p)$ is also a value in equational semantics by considering f_p as a classical constructor.

The main concept driving the translation consists in the management of functions and applications, and relies on the following theorem :

Theorem 61 *If $(v_i)_{i \leq 3}$ are values, then:*

$$(v_1\ v_2) \rightarrow_\varepsilon v_3 \Leftrightarrow (v_1, v_2)\ [\![\mathcal{P}]\!]_{arg,call}\ (v_3)$$

where $[\![\mathcal{P}]\!]$ is the semantics of the equational program $\mathcal{P}$ translated from FP.

For each function definition, the function fun$[\![\ ]\!]$ defined in figure 6-left computes a piece of the expected equational program $\mathcal{P}$. Pieces of equational programs are denoted by Π, and the notation $\Pi = c \rightarrow \Sigma$ means that $\Pi = \{c \rightarrow x = t \mid x = t \in \Sigma\}$. The expression $[\![\ e\ ]\!]\ x, \gamma$, defined in figure 6-right, computes a set of statements such that:

Theorem 62

$$\begin{array}{l} e \rightarrow_\gamma v \Leftrightarrow \Sigma \Rightarrow_{\mathcal{P}} \{x = v\} \\ \text{with :}\quad \Sigma = [\![\ e\ ]\!]\ x, \gamma \end{array}$$

For instance, the definition of the function `rev` is translated into:

$$\begin{array}{rl|rl} rev_0 & \rightarrow \alpha.call = (rev_1\ \alpha.arg) & cons & \rightarrow \alpha.rev\_call = \alpha.L_2.call \\ rev_1 & \rightarrow \alpha.call = \alpha.L_1.rev\_call & " & \rightarrow \alpha.L_2.arg = \alpha.2 \\ " & \rightarrow \alpha.L_1.rev\_h = \alpha.1 & " & \rightarrow \alpha.L_2 = \alpha.L_3.call \\ " & \rightarrow \alpha.L_1 = \alpha.arg & " & \rightarrow \alpha.L_3.arg = (cons\ \alpha.L_4\ \alpha.L_5) \\ nil & \rightarrow \alpha.rev\_call = \alpha.rev\_h & " & \rightarrow \alpha.L_3 = (rev_0) \\ & & " & \rightarrow \alpha.L_4 = \alpha.1 \\ & & " & \rightarrow \alpha.L_5 = \alpha.rev\_h \end{array}$$

After partial evaluation and renaming of local variables, the following equational program is obtained:

$$\begin{array}{rl|rl} rev_0 & \rightarrow \alpha.call = (rev_1\ \alpha.arg) & cons & \rightarrow \alpha.rev\_call = \alpha.2.rev\_call \\ rev_1 & \rightarrow \alpha.call = \alpha.L_1.rev\_call & " & \rightarrow \alpha.2.rev\_h = (cons\ \alpha.1\ \alpha.rev\_h) \\ " & \rightarrow \alpha.L_1.rev\_h = \alpha.1 & nil & \rightarrow \alpha.rev\_call = \alpha.rev\_h \\ " & \rightarrow \alpha.L_1 = \alpha.arg & & \end{array}$$

- $\mathrm{fun}[\![\texttt{let}\ f\ x_1 ... x_n = e]\!] =$
  $\Pi \cup \{f_n \rightarrow \Sigma\}$
  where :
  $\Pi = \bigcup_{0 \leq k < n} \mathit{Closure}(f, k)$
  $\gamma = (x_k : \alpha.k)_{1 \leq k < n}(x_n : \alpha.arg)$
  $\Sigma = [\![e]\!]\ \alpha.call, \gamma$
- $\mathrm{fun}[\![\texttt{let}\ \ f\ x_1 ... x_n = \texttt{fun}$
  $c\ y_1 ... y_m \rightarrow e_c \mid ...]\!] =$
  $\Pi \cup \Pi' \cup (\bigcup \Pi_c)$
  where :
  $\Pi = \bigcup_{0 \leq k < n} \mathit{Closure}(f, k)$
  $\gamma = (x_k : \alpha.k)_{1 \leq k \leq n}$
  $m = \mathit{new\_local\_nbr}()$
  $\Pi' = f_n \rightarrow \{$
  $\alpha.call = \alpha.L_m.f\_call$
  $\alpha.L_m.f\_x_k = \alpha.k \quad {}_{1 \leq k \leq n}$
  $\alpha.L_m = \alpha.arg$
  $\}$
  $\gamma_0 = (x_k : \alpha.f\_x_k)_{1 \leq k \leq n}$
  $(\forall c)\ \gamma_c = \gamma_0(y_j : \alpha.j)_{j \leq m}$
  $(\forall c)\ \Pi_c = c \rightarrow [\![e_c]\!]\ \alpha.f\_call, \gamma_c$
- $\mathit{Closure}(f, k) = f_k \rightarrow \{$
  $\alpha.call = (f_{k+1}\ \alpha.1 ... \alpha.k\ \alpha.arg)$
  $\}$

- $[\![\ x\ ]\!]\ y, \gamma = \{y = \gamma(x)\}$
- $[\![\ f\ ]\!]\ y, \gamma = \{y = (f_0)\}$
- $[\![\ (e_1\ e_2)\ ]\!]\ y, \gamma =$
  $\bigcup_{i \leq 2} \Sigma_i$
  where :
  $m = \mathit{new\_local\_nbr}()$
  $\Sigma_0 = \{y = \alpha.L_m.call\}$
  $\Sigma_1 = [\![e_1]\!]\ \alpha.L_m, \gamma$
  $\Sigma_2 = [\![e_2]\!]\ \alpha.L_m.arg, \gamma$
- $[\![\ (c\ e_1 \ldots e_n)\ ]\!]\ y, \gamma =$
  $\{y = (c\ y.L_{m_1} \ldots y.L_{m_n})\} \cup (\bigcup \Sigma_i)$
  where :
  $(\forall i)\ m_i = \mathit{new\_local\_nbr}()$
  $(\forall i)\ \Sigma_i = [\![e_i]\!]\ y.L_{m_i}, \gamma$
- $[\![\ (\pi\ e_1 \ldots e_n)\ ]\!]\ y, \gamma =$
  $\{y = (\pi\ y.L_{m_1} \ldots y.L_{m_n})\} \cup (\bigcup \Sigma_i)$
  where :
  $(\forall i)\ m_i = \mathit{new\_local\_nbr}()$
  $(\forall i)\ \Sigma_i = [\![e_i]\!]\ y.L_{m_i}, \gamma$

Fig. 6. Translation: function and expressions

7 Strategies

In contrast with Section 4, this section concerns the recursive structure of the derivations. The aim is to construct canonical derivations for the relation $[\![\mathcal{P}]\!]_{P,R}$.

We will say that a system Σ defines a variable x if and only if there exists a value v such that $x = v \in \Sigma$. By extension, we will say that Σ defines a set of attributes A on a variable x if and only if Σ defines the variable x and all the variables $x.a$ where $a \in A$.

7.1 An example

Consider the following program $\mathcal{P}$:

$$\begin{array}{lll} cons & \rightarrow \alpha.r = \alpha.2.r & (eqn_1) \\ cons & \rightarrow \alpha.2.p = (cons\ \alpha.1\ \alpha.p) & (eqn_2) \\ nil & \rightarrow \alpha.r = \alpha.p & (eqn_3) \end{array}$$

These three equations have been denoted by eqn_i to simplify notations. Actually, for any list l the following statement holds:

$$(l, nil)\ [\![\mathcal{P}]\!]_{p,r}\ (l')$$

where l' is the list l reversed. Recall that such a statement means that there exists a derivation $\Sigma_{in} \Rightarrow_{\mathcal{P}} \Sigma_{out}$ where Σ_{in} defines p on α and Σ_{out} defines r on α. In this section, we want to find the structure of a possible derivation for $\Sigma_{in} \Rightarrow_{\mathcal{P}} \Sigma_{out}$.

We need here new notations. In Section 4.2, we have denoted by $\overset{y=t,x}{\rightarrow}$ the instantiation of the equation $y = t$ on the variable x. In the same way, we will denote by $\overset{p,r,x}{\rightarrow}$ a derivation which allow to define $x.r$ from $x.p$. More precisely, if system Σ defines p on x and $\Sigma \overset{p,r,x}{\rightarrow} \Sigma'$, then Σ' defines r on x.

So, let us start with a derivation $\Sigma_{in} \rightarrow^*_{\mathcal{P}} \Sigma'$. From the definition of $(\rightarrow_{\mathcal{P}})$, we can observe that its $(\rightarrow_{\emptyset})$ part is not able to introduce a variable $\alpha.r$ on the left side of an equation. Thus at least one equation of $\mathcal{P}$ has been instantiated on α. This requires that either $\alpha = (cons\ v_1\ v_2)$ or $\alpha = (nil)$. Then, it is possible to inductively construct $\overset{p,r,\alpha}{\rightarrow}$ as follows:

- Either $\alpha = (nil)$: it is possible to use the derivation $(\overset{eqn_3,\alpha}{\rightarrow})$, thus applying the equation associated to nil in the program $\mathcal{P}$. Then $(\Rightarrow_{\emptyset})$ will perform all the sub-term, substitution and primitive derivations to produce a system Σ' which defines r on α, without using any other equation of $\mathcal{P}$. More generally, using Theorem 41 leads to the expected derivation for $x = (nil)$:

$$\overset{p,r,x}{\rightarrow} = d^x_{nil} = [\overset{eqn_3,x}{\rightarrow}; \Rightarrow_{\emptyset}]$$

- Or $\alpha = (cons\ v_1\ v_2)$: The strategy consists here in recursively applying the derivation $\overset{p,r,\alpha.2}{\rightarrow}$. As a first step, we get a system which defines p on $\alpha.2$ thanks to the derivation $\overset{eqn_2,\alpha}{\rightarrow}$, followed by $\Rightarrow_{\emptyset}$ to perform substitution, sub-term and primitive derivations. Then we use the derivation $\overset{p,r,\alpha.2}{\rightarrow}$ to get a system which defines r on $\alpha.2$, and we end the process by applying $\overset{eqn_1,\alpha}{\rightarrow}$ followed by $\Rightarrow_{\emptyset}$ to get a system which defines r on α. More generally, using Theorem 41 leads to:

$$\overset{p,r,x}{\rightarrow} = d^x_{cons} = [\overset{eqn_2,x}{\rightarrow}; \Rightarrow_{\emptyset}; \overset{p,r,x.2}{\rightarrow}; \overset{eqn_1,x}{\rightarrow}; \Rightarrow_{\emptyset}]$$

Thus, a possible definition of the expected derivation is the inductive definition:

$$\overset{p,r,x}{\rightarrow} = d^x_{cons} + d^x_{nil}$$

With this definition, for every system Σ_1 which defines p on x, and Σ_2 which defines r on x, the following statement holds[3]:

$$\Sigma_1 \Rightarrow_{\mathcal{P}} \Sigma_2 \Leftrightarrow \Sigma_1 \overset{p,r,x}{\rightarrow} \Sigma_2$$

[3] The proof for such a statement is made by induction on the length of the derivation $\Rightarrow_{\mathcal{P}}$, and then by case-analysis on the equations of $\mathcal{P}$ that have been applied to x. The proof largely makes use of Theorems 41.

As a short cut, we summarize all these properties by the following notations, which define an operational semantics for $\mathcal{P}$:

$$[\![\mathcal{P}]\!]^{op} = \{\ (p,r,cons,seq_1)\ (p,r,nil,seq_2)\ \} \quad \text{where} \begin{cases} seq_1 = [eqn_2;\ p,r,\alpha.2;\ eqn_1] \\ seq_2 = [eqn_3] \end{cases}$$

7.2 Operational Semantics

In this section we refine the definitions above. An elementary derivation step s is either an equation $y = t$ of the program $\mathcal{P}$, or a triplet (P, R, y). A sequence seq is a concatenation $[s_1; \ldots; s_n]$ of steps. A strategy $\mathcal{S}$ is a set of tuples (P, R, c, s), where $P \subset \mathcal{P}rm$ and $R \subset \mathcal{R}es$ are two sets of attributes, c is a constructor, and s is a sequence.

From a strategy $\mathcal{S}$, the derivation $\overset{P,R,x}{\rightarrow_{\mathcal{S}}}$ is defined recursively by:

$$\begin{array}{lll} \Sigma \overset{P,R,x}{\rightarrow_{\mathcal{S}}} \Sigma' & \Leftrightarrow & x = (c\ldots) \in \Sigma \\ & & \text{and } (P', R, c, seq) \in \mathcal{S} \\ & & \text{with } P' \subset P \\ & & \text{and } \Sigma \overset{seq,x}{\rightarrow_{\mathcal{S}}} \Sigma' \end{array} \qquad \begin{array}{lll} \Sigma \overset{seq,x}{\rightarrow_{\mathcal{S}}} \Sigma' & \Leftrightarrow & seq = [s_1; \ldots; s_n] \\ & & \text{and } \Sigma \overset{s_1,x}{\rightarrow_{\mathcal{S}}} \ldots \overset{s_n,x}{\rightarrow_{\mathcal{S}}} \Sigma' \\ \Sigma \overset{y=t,x}{\rightarrow_{\mathcal{S}}} \Sigma' & \Leftrightarrow & \Sigma \overset{y=t,x}{\rightarrow} \Sigma' \\ \Sigma \overset{(P,R,y),x}{\rightarrow_{\mathcal{S}}} \Sigma' & \Leftrightarrow & \Sigma \overset{P,R,y[x]}{\rightarrow_{\mathcal{S}}} \Sigma' \end{array}$$

A strategy $\mathcal{S}$ is an operational semantics for $\mathcal{P}$ if and only if, for every system Σ_P defining P on x, and every system Σ_R defining R on x, the following holds:

$$\Sigma_P \Rightarrow_{\mathcal{P}} \Sigma_R \Leftrightarrow \Sigma_P \overset{P,R,x}{\rightarrow_{\mathcal{S}}} \Sigma_R$$

We denote such a situation by $[\![\mathcal{P}]\!]^{op} = \mathcal{S}$, and we have the following property:

$$(v, v_1 \ldots v_n)[\![\mathcal{P}]\!]^{op}_{P,R}(w_1 \ldots w_m) \quad \Leftrightarrow \quad \begin{cases} \alpha = v \\ \alpha.p_1 = v_1 \\ \ldots \\ \alpha.p_n = v_n \end{cases} \overset{P,R,\alpha}{\rightarrow_{\mathcal{S}}} \begin{cases} \alpha.r_1 = w_1 \\ \ldots \\ \alpha.r_m = w_m \end{cases}$$

Notice the difference with the definition of $[\![\mathcal{P}]\!]$:

$$(v, v_1 \ldots v_n)[\![\mathcal{P}]\!]_{P,R}(w_1 \ldots w_m) \quad \Leftrightarrow \quad \begin{cases} \alpha = v \\ \alpha.p_1 = v_1 \\ \ldots \\ \alpha.p_n = v_n \end{cases} \Rightarrow_{\mathcal{P}} \begin{cases} \alpha.r_1 = w_1 \\ \ldots \\ \alpha.r_m = w_m \end{cases}$$

While $[\![\mathcal{P}]\!]$ gives no information about the structure of the derivation $\Rightarrow_{\mathcal{P}}$ involved in this semantics, $[\![\mathcal{P}]\!]^{op}$ exhibits a derivation with a fixed scheme of recursion.

7.3 Construction of $[\![\mathcal{P}]\!]^{op}$

This section presents the algorithm which find such an operational semantics for a given equational program $\mathcal{P}$. Actually, we want to translate $[\![\mathcal{P}]\!]$ into $[\![\mathcal{P}]\!]^{op}$. The kernel of the algorithm is a fixpoint computation of a well suited strategy, denoted by $\mathcal{S}_\infty$. We introduce two notations:

$$\begin{array}{l} \Sigma \Rightarrow_{n,\mathcal{P}} \Sigma' \Leftrightarrow (\exists \Sigma'' \supset \Sigma') \quad \Sigma \rightarrow^n_{\mathcal{P}} \Sigma'' \\ \mathcal{H}(n,\mathcal{S}) \Leftrightarrow \quad \forall (P,R), \quad \Sigma_P \Rightarrow_{n,\mathcal{P}} \Sigma_R \Leftrightarrow \Sigma_P \overset{P,R,x}{\rightarrow_{\mathcal{S}}} \Sigma_R \end{array}$$

The fixpoint computation involves a function *Next*, and an order-relation ($\sqsubset$) such that the following theorem holds:

Theorem 71

$$\begin{array}{l} (\forall n) \quad \mathcal{H}(n,\mathcal{S}) \Rightarrow \mathcal{H}(n+1, \mathcal{S} \cup \mathit{Next}(\mathcal{S})) \\ \mathcal{S} \sqsubset \mathcal{S}' \Rightarrow \mathit{Next}(\mathcal{S}) \sqsubset \mathit{Next}(\mathcal{S}') \end{array}$$

The relation ($\sqsubset$) is defined in the next section and allow to define a lattice-structure on the strategies. Then it is easy to prove that the greatest fixpoint of *Next* exists and the following holds:

Theorem 72 $(\forall n) \quad \mathcal{H}(n, \mathrm{gfix}\ \mathit{Next})$ *and* $[\![\mathcal{P}]\!]^{op} = \mathrm{gfix}\ \mathit{Next}$.

The function *Next* is long and complex to define precisely. The next section provides guidelines to understand its complete definition.

7.4 The function *Next*

In spite of a long definition, the construction of the *Next* function is intuitive, and consists in exploring what a strategy could consist of. The extended definition is reported in figure 7, and is only the formalization of what we presented in the introducing example. This section just provides guidelines (in small font) to understand the role of the different components involved in these definitions.

The *Next* function computes strategies independently for each constructor through the function *Pool*. Actually, we must ensure that a derivation $\overset{P,R,x}{\rightarrow_{\mathcal{S}}}$ will be available for all the constructors involved to compute the attributes in R. This set of constructors is denoted by τ_R. The predicate *implemented*$(P,R,\mathcal{S})$ tests if a strategy is available for the pair (P,R) in $\mathcal{S}$ for each constructor in τ_R.

The function $\mathit{Pool}(c,\mathcal{S})$ computes all available strategies when $\alpha = (c\ v_1 \ldots v_n)$ and when the recursive derivations are taken from $\mathcal{S}$. By a fixpoint algorithm, this function computes a set of tuples (D, seq, P, R), where D is a set of variables, *seq* a sequence of steps, P and R two sets of attributes. The invariant property maintained at each iteration of the fixpoint computation is the following. For each tuple (D, seq, P, R), *seq* is a derivation such that:
Let Σ be a system which defines P on α, and where $\alpha = (c \ldots)$. Suppose that applying the sequence *seq* on Σ leads to the system Σ', that is: $\Sigma \overset{seq,\alpha}{\rightarrow_{\mathcal{S}}} \Sigma'$. Then, this system Σ' defines R on α, and it defines also all the variables in the set D.

Each iteration of this fixpoint algorithm is computed by the function $\mathit{Infer}(c, \mathcal{S})(E)$ which adds new steps to the sequences seq inside tuples $(D, \mathit{seq}, P, R) \in E$, maintaining the invariant property above.

The other functions compute auxiliary results. Thus $\mathit{need}(s)$ and $\mathit{prod}(s)$ respectively computes the variables that are *need*ed to be defined before applying the step s, and those which are *prod*uced after this step.

The *Next* function has a greatest fixpoint, with the following order $\sqsubset$ on strategies:

$$\begin{array}{l} \mathcal{S} \sqsubset \mathcal{S}' \Leftrightarrow (\forall P, P', R) \\ \left(\begin{array}{c} \mathit{implemented}(P', R, \mathcal{S}') \\ \text{and } \mathit{implemented}(P, R, \mathcal{S}) \end{array} \right) \ \Rightarrow P \subset P' \end{array}$$

Since the function *Next* is monotonic, its greater fixpoint is $\mathit{Next}^m(\mathcal{S}_0)$ for some integer m, with $\mathcal{S}_0 = \{(\emptyset, R, c, []) \mid c \in \tau_r,\ R \subset \mathcal{R}es\}$.

The algorithm provided here is well-suited to make proofs, but is completely inefficient in practice. Essentially this complexity comes from the permutations allowed by the confluence theorem 41, and from the large amount of possible pairs (P, R) to consider. Our implementation improves this algorithm in order to take into account these permutations, and to control and limit the number of pairs (P, R) to be considered.

$\bullet \mathit{Next}(\mathcal{S}) = \{ \quad (P, R, c, \mathit{seq}) \mid$	$\bullet \mathit{Add}(s, (D, \mathit{seq}, P, R)) = ($
$\quad (\exists P' \subset P)\ (\_, \mathit{seq}, P', R) \in \mathit{Pool}(c, \mathcal{S})$	$\quad D \cup \mathit{prod}(s), [\mathit{seq}; s],$
$\quad$ and $\forall c' \in \tau_R,$	$\quad P \cup (\mathit{need}(s) \cap \{\alpha.p \mid p \in \mathcal{P}rm\}),$
$\quad (\exists P' \subset P)\ (\_, \_, P', R) \in \mathit{Pool}(c', \mathcal{S})$	$\quad R \cup (\mathit{prod}(s) \cap \{\alpha.r \mid r \in \mathcal{R}es\}) \quad)$
$\quad \} \quad \}$	$\bullet \mathit{defined}(s, (D, \_, \_, \_)) \Leftrightarrow$
$\bullet \tau_R = \{c \mid \forall r \in R\ \exists t\ c \to \alpha.r = t \in \mathcal{P}\}$	$\quad \mathit{needs}(s) \subset D \cup \{\alpha.p \mid p \in \mathcal{P}rm\}$
$\bullet \mathit{implemented}(P, R, \mathcal{S}) \Leftrightarrow \forall c \in \tau_R$	$\bullet \mathit{callable}((x = t), \mathcal{S}) \Leftrightarrow \text{true}$
$\quad (\exists P' \subset P) \quad (P', R, c, \_) \in \mathcal{S}$	$\bullet \mathit{callable}((P, R, x), \mathcal{S}) \Leftrightarrow$
$\bullet \mathit{Pool}(c, \mathcal{S}) = \text{fix } (\mathit{Infer}(c, \mathcal{S}))\ E_0$	$\quad \mathit{implemented}(P, R, \mathcal{S})$
$\quad$ with $E_0 = \{(D_c, [], \emptyset, \emptyset)\}$	$\bullet \mathit{need}(x = t) = \mathcal{V}ars(t)$
$\quad$ and $D_c = \{\alpha; \alpha.1; \ldots; \alpha.n\},\ n = \sharp c$	$\bullet \mathit{need}(P, R, x) = \{x\} \cup \{x.p \mid p \in P\}$
$\bullet \mathit{Infer}(c, \mathcal{S})(E) = E \cup \{\mathit{Add}(s, e) \mid e \in E$	$\bullet \mathit{prod}(x = t) = \{x\}$
$\quad$ and $\mathit{defined}(s, e)$ and $\mathit{callable}(s, \mathcal{S})\}$	$\bullet \mathit{prod}(P, R, x) = \{x.r \mid r \in R\}$

Fig. 7. The function *Next*

8 Backward Translation

This section gives guidelines about the translation from the operational semantics of an equational program (*i.e.* a strategy) into a new functional program.

Though consisting in many steps, the translation is not complex. During the first step, the strategy $[\![\mathcal{P}]\!]^{op}$ is reduced to a new strategy denoted by $\mathcal{S}$ such that for each triplet (P, R, c) there is at most one sequence seq such that

$(P, R, c, seq) \in \mathcal{S}$. Selecting which sequence will be optimal is a difficult problem, but simple heuristics are sufficient to choose interesting sub-optimal ones. Actually, we choose sequences with few constructors (to save space), few non-evaluated expressions (like $y = x.r$ with $x = (c\ \ldots)$) and few compositions (to make deforestation)[4].

The second step defines the functions to be created in order to implement the strategies. For each pair (P, R), since there is only one sequence *seq* per constructor c in $\mathcal{S}$, the relation $\stackrel{P,R,x}{\to}_{\mathcal{S}}$ can be implemented by a pattern-matching function. This function has one parameter per attribute in P, and returns a tuple[5] with one value per attribute in R. Then, for each sequence *seq*, a piece of code is generated.

To implement a sequence *seq*, the idea consists in associating each variable in *seq* to a fresh local variable of `f`, and then to inline the variables that are used only once. For instance, the strategy of the section 7.1 is implemented in the following way:

```
let f x1 = fun
| cons y1 y2 -> f (cons y1 x1) y2
| nil -> x1
```

From this basic scheme, there exists many variations. Thus, a constructor c for which a unique pair (P, R) is defined should be interpreted as a function-closure construtor. Then, no pattern-matching is needed, and a pure functional expression is generated. Special treatment is also performed for constructors which correspond to tuples. See the results in section 5 to find examples.

9 Conclusion

Equational programs and semantics have been dedicated to perform program transformations in the context of functional programming. However, this framework does not rely on functional definitions, such as functors, morphisms or λ-calculus. Thus, the control flow of a program is not embedded in any fixed recursion scheme. Since the control flow is reconstructed *after* applying program transformations, it can be completely transformed. This provides significant improvements to many program transformations, especially to partial evaluation and deforestation.

Another interest of this approach is that equational semantics is not restricted to functional programs and could be used to modelize other programming paradigms. The key idea of such a semantics is to separate, as far as possible, *what* is computed from *how* it is computed. Such an idea should be used largely to improve existing transformation methods.

[4] Actualy, we need an approximation for the complexity of each sequence. We are sure that related abstract interpretations and static analysis may be used to improve this step. Future works will investigate this possibility.

[5] It is easy to add tuples to functional programs as syntactic sugar.

This work comes from various formalisms and programming paradigms. For many years, we have been collecting the best of existing techniques, such as attribute grammar deforestation [1], folds and hylomorphism fusion [4], type-directed or calculational deforestation [8, 10, 9, 11]. But these formalisms were too much different from each others to be compared and to produce nice cross-fertilization. This is why we try now to refund them in a new theoretical and implementable framework. Following this driving idea, the notion of equational programs raised naturally and equational semantics was not far away.

References

1. Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Symbolic composition. Technical Report 3348, INRIA, January 1998.
2. Loic Correnson. Equational Semantics. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 205–222, Amsterdam, The Netherlands, March 1999. INRIA rocquencourt.
3. Etienne Duris. *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle*. PhD thesis, Université d'Orléans, 1998.
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. 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.
6. 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.
7. Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A calculational fusion system HYLO. In *Proc. IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, Le Bischenberg, France, February 1997.
8. Tim Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*. ACM press, 1997.
9. 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.
10. 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.
11. 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.