# Notes for pipelines of transformations for ML

F. Bellegarde[*]
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology

$Id : commands.tex, v1.21995/11/0100 : 50 : 15bellExp$

**Abstract**

These notes attempt to survey basic manipulations that can be done to a ML functional program prior to compilation. The paper does not consider transformations for improvement of the code but it only considers manipulations of the source text which can allow the compiler to do a better job. Which manipulation to choose depends on the target language. The survey can be useful for answering the above question and hence for choosing a convenient pipeline of transformations to prepare translation towards a given target language.

Functional program transformation modifies the whole structure of a functional program in order to facilitate the production of a better code. It can also make the program use less memory. Functional programs have a precise mathematical semantic which allows to use source to source transformation techniques in order to provide any semantically correct desired form of program. In this paper, we address a catalog of eager semantic transformations on a functional language like ML. We consider only here fully automatic program transformation. Most of these transformations are usually well known though it can happen that the paper makes some contribution in some of the transformations that are described. The goal of the paper consists mostly in classifying basic transformations and in seeing how they can be combined into diverses transformation strategies. It might happen that the transformation strategy we describe as a composition of basic transformations could be expressed directly in a more specific and efficient algorithm. However the division into basic parts makes the transformation toolkit which is proposed in the paper very flexible and reusable.

[*]The author was supported by a contract with Air Force Materiel Command (F19628-93-C-0069).

A transformation strategy serves a certain purpose. The concern can be the target language of the translation. For example, *monomorphization* is a useful transformation if the target language is a nonpolymorphic language, and *defunctionalization* is useful if the target language is first order, and *lambda-lifting* is useful if the target language is a term rewriting system. The sole concern can be optimization. In this case, transformation strategies such as deforestation or tuppling are useful. Another concern can be the model of the implementation sequential or parallel or distributed. In this paper, we limit ourselves to some transformation strategies of the first category.

Some basic transformations are easily recognized as belonging in a specific strategy but they can also be used as steps in other strategies. For example a type enumeration can be a step for a *monomorphization* but it can also be useful to perform type enumeration as a step of a *defunctionalization* in a polymorphic program. In the first section, we attempt to classify these basic transformations with respect to the area of the structure of the program they modify. Then, we will see how to combine them into safe and effective strategies.

## 1 Classification of basic transformations

Basic transformations exists for every areas of the program structure:

- renamings,
- patterns and case expressions,
- $\lambda$-abstractions and let expressions,
- functions and applications,
- types,
- expressions.

### 1.1 Renaming

Uniqueness of identifiers is a useful prerequisite for all transformation strategies. Two different kinds of renaming can be distinguished:

- *$\alpha$-renaming* ensures scope independence for the identifiers in *lambda*-abstractions and local let declarations.
- *Pattern renaming* ensures scope independence for variables in local patterns.

## 1.2 Pattern and case transformations

The simpler the syntax of the core language of the transformation, the simpler the transformation algorithm. Also limitation of the number of the possible constructions can be very useful as prerequisite for effective transformation strategies. Transformation about the patterns and the case expressions in a functional program are well-known. We class them as follows:

- *Commodity pattern removals:* These transformations removes commodity for programmers like the *wild-card* and the *as* constructions.
- *Pattern limitation to case expressions:* This transformation makes patterns appear only in case expressions. It has been studied by Auguston in [**?**].
- *Conversion to simple patterns:* The simple patterns are interesting because a case expression consisting only of single patterns can be compiled easily into an efficient code. This transformation is described in [**?**]. It is supposed that the patterns occurs only in cases. The algorithm can increase the size of the code by a large amount but it does not change its efficiency. The algorithm can be written so that it creates wild-card patterns but it works only on a program without commodity patterns.
- *Case removal from application arguments:* The transformation removes case expressions as operators in function applications. So

  $$f\ t_1\ \cdots\ \textbf{case}\ e\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n, \cdots\ t_m$$

  is transformed into:

  $$\begin{array}{l} \textbf{case}\ e\ \textbf{of} \\ \qquad p_1 \Rightarrow\ f\ t_1\ \cdots\ e_1\ \cdots\ t_m \\ \qquad \vdots \\ \qquad \mid p_n \Rightarrow f\ t_1\ \cdots\ e_n\ \cdots\ t_m) \end{array}$$

  This transformation increases the size of the code.
- *Case and if application:* This transformation pushes the applied terms of a **case** or of a functional **if** expression inside the **case**. For example;

  $$\begin{array}{l} \textbf{case}\ e\ \textbf{of} \\ \qquad p_1 \Rightarrow\ e_1 \\ \qquad \vdots \\ \qquad \mid p_n \Rightarrow e_n)\ u_1\ u_2\ \cdots\ u_m \end{array}$$

becomes

**case** $e$ **of**

$p_1 \Rightarrow e_1\ u_1\ u_2\ \cdots\ u_m$

$\vdots$

$\mid p_n \Rightarrow e_n\ u_1\ u_2\ \cdots\ u_m$

- *Overlapping pattern removal:* Patterns in case expressions are supposed to be checked from top to bottom. This allows to have overlapping patterns. However an earlier pattern is not allowed to overlap completely a later pattern since then, the later pattern could never match. This error can be detected syntactically. We suppose here that the program is syntactically correct. The goal of this transformation is to remove overlapping patterns from the program. It is necessary for a translation into a term rewrite system which does not take account of order in its rewriting.

  When a list of patterns $p_1, \cdots, p_n$ is made of simple patterns, overlapping of patterns is either a repetition of the same pattern or an overlap with a variable. In a syntactically correct program, this can happen only when there is a pattern reduced to a variable. Moreover, the pattern reduced to a variable must be $p_n$. The case expression is then:

  **case** $e$ **of**

  $c_1(x_{1,1}, \cdots, x_{1,m_1}) \Rightarrow e_1$

  $\vdots$

  $\mid c_{n-1}(x_{n-1,1}, \cdots, x_{n-1,m_n} \Rightarrow e_{n-1}$

  $\mid x => e_n$

  Suppose the constructors belong to the data type $T$ of constructor set $C$, the complement $\{c'_1, c'_2, \cdots, c'_m\}$ in $C$ of the set $\{c_1, \cdots, c_{n-1}\}$ allows to generate the equivalent and non overlapping case expression:

  **case** $e$ **of**

  $c_1(x_{1,1}, \cdots, x_{1,m_1}) \Rightarrow e_1$

  $\vdots$

  $\mid c_{n-1}(x_{n-1,1}, \cdots, x_{n-1,m_n}) \Rightarrow e_{n-1}$

  $\mid c'_1(x_{new_{1,1}}, \cdots, x_{new_{1,m'_1}}) \Rightarrow e_n$

  $\vdots$

  $\mid c'_m(x_{new_{1,1}}, \cdots, x_{new_{m-1,m'_m}}) \Rightarrow e_n$

When patterns are non overlapping in a syntactically correct program, the following particular cases can be noticed:

- when a tuple is a pattern, it has to be the unique pattern,
- when the simple patterns are exhaustive and when the last pattern is a variable pattern, this last pattern will never match in a well-typed program, therefore, it corresponds to dead code that can be removed.

This transformation increases the size of the code but it does not affect its efficiency.

- *Trivial case expressions:* It is important to free the program from trivial cases. The identity case **case** $e$ **of** $y \Rightarrow e_1$ can be simplified into $e_1[e/y]$. Other simplifications can be made like:

$$\begin{array}{l} \textbf{case } e \textbf{ of} \\ \qquad p_1 \Rightarrow p_1 \\ \qquad \vdots \\ \qquad \mid p_n \Rightarrow p_n \end{array}$$

  can be transformed into $e$ or

$$\begin{array}{l} \textbf{case } e \textbf{ of} \\ \qquad p_1 \Rightarrow e_1 \\ \qquad \vdots \\ \qquad \mid e' \Rightarrow e_i \\ \qquad \vdots \\ \qquad \mid p_n \Rightarrow e_n \end{array}$$

  where $e'$ is a renaming $\alpha(e)$ of $e$ can be transformed into $\alpha^{-1}(e_1)$.

- *Variable limited case test expressions:* Case test expressions need to be removed for translating the program into a term rewrite system. A construction

$$\begin{array}{l} \textbf{case } \ e \textbf{ of} \\ \qquad p_1 \Rightarrow e_1 \\ \qquad \vdots \\ \qquad \mid p_n \Rightarrow e_n \end{array}$$

can be transformed into a function application $f_{new}(e, x_1 \cdots x_m)$ where the variables $x_1, \cdots, x_m$ are the global variables occurring inside the case expressions and where $f_{new}$ is defined as:

$$f_{new}(y_{new}, x_1, \cdots, x_m) = \textbf{case}\ y_{new}\ \ \textbf{of} \begin{array}{l} \\ p_1 \Rightarrow e_1 \\ \vdots \\ \mid p_n \Rightarrow e_n \end{array}$$

This transformation creates a new mutually recursive function. A simple unfolding of $f_{new}$ gives back the original program.

- *Pattern limitation to functions:* When the simple patterns are exhaustive and non overlapping i.e. when there are no variable as a pattern, a transformation can push the patterns into the left-hand side of the function definition. It generates an equation for each pattern:

$$f(x_1, \cdots, x_m) \ = \textbf{case}\ x_i\ \ \textbf{of} \begin{array}{l} \\ p_1 \Rightarrow e_1 \\ \vdots \\ \mid p_n \Rightarrow e_n \end{array}$$

becomes

$$\begin{array}{l} f(x_1, \cdots, x_{i-1}, p_1, x_{i+1}, \cdots, x_m) = e_1 \\ \vdots \\ f(x_1, \cdots, x_{i-1}, p_n, x_{i+1}, \cdots, x_m) = e_n \end{array}$$

We get nested patterns by furthermore transformations of the constructions:

$$f(pa_1, \cdots, pa_i, \cdots, pa_m) \ = \textbf{case}\ x_i\ \ \textbf{of} \begin{array}{l} \\ p_1 \Rightarrow e_1 \\ \vdots \\ \mid p_n \Rightarrow e_n, \end{array}$$

where $x_i$ occurs in the pattern $pa_i$, into:

$$\begin{array}{l} f(pa_1, \cdots, pa_1[p_1/x_i], \cdots, pa_m) = e_1 \\ \vdots \\ f(pa_1, \cdots, pa_i[p_n/x_i], \cdots, pa_m) = e_n \end{array}$$

- *Case from conditional:* The conditional is a non strict feature in an eager language. For example, there is no *if* construction in a (non conditional) term rewriting system. It is easy to remove *if* constructions by transforming an expression

$$\textbf{if } b \textbf{ then } e_1 \textbf{ else } e_2$$

  into an application $f_{new}\ b\ x_1\ \cdots\ x_n$ where $x_1, \cdots, x_n$ are the global variables in $e_1$ and $e_2$, and

$$\begin{aligned} f_{new}\ \textbf{true}\ x_1\ \cdots\ x_n &= e_1 \\ f_{new}\ \textbf{false}\ x_1\ \cdots\ x_n &= e_2. \end{aligned}$$

- *Conditional from case:* It is the inverse transformation of the conditional removal. If the language has patterns limited to case expressions, the transformation consists in looking for of pairs of patterns **true**, **false** in case expressions.

- *Case generation by selectors in tuple removal:* An expression $e$ which contains occurrences of a same expression $e_1$ under selectors **first** and **second** in a tuple can be compiled in a better code if transformed into:

$$\begin{array}{l} \textbf{case } e_1 \textbf{ of} \\ \qquad (u, v) \Rightarrow e[uo_1 \leftarrow u, \cdots, uo_n \leftarrow u, vo_1 \leftarrow v, \cdots, vo_n \leftarrow v] \end{array}$$

  where $uo_1, \cdots, uo_n$ are the occurrences of **first** $e_1$ and $vo_1, \cdots, vo_n$ are the occurrences of **second** $e_1$.

- *Trivial conditional:* This consists to cancel the trivial:

$$\textbf{if true then } e_1 \textbf{ else } e_2$$

  into $e_1$, and

$$\textbf{if false then } e_1 \textbf{ else } e_2$$

  into $e_2$.

- *Constant propagation:* It can be worthwhile to propagate informations about values that are given to variables by the way of constant equality tests:

$$\textbf{if } x = e \textbf{ then } e_1 \textbf{else } e_2$$

where $e$ is a constant can be transformed into:

$$\textbf{if } x = e \textbf{ then } e_1[e/x]\textbf{else } e_2$$

or inequality tests:

$$\textbf{if } x \neq e \textbf{ then } e_1\textbf{else } e_2$$

where $e$ is a constant can be transformed into:

$$\textbf{if } x \neq e \textbf{ then } e_1\textbf{else } e_2[e/x]$$

The same can be done for constant patterns in case expression where $p_i \Rightarrow e_i$ from a case expression with test variable $x$ and where $p_i$ is a constant can be replaced by $p_i \Rightarrow e_i[p_i/x]$.

## 1.3 Lambda-abstraction and let expression transformations

- Trivial let: A let expression like $l$et $x = e$ $i$n $x$ becomes $e$.
- *$\beta$-reduction:* Applications of $\lambda$-abstractions can safely be $\beta$-reduced in a well-typed program. Safely, here means that the transformation is terminating. However it has also to induce a better, not a worse generated code. For that it is required to limit $beta$-reductions to variable parameters or to expression parameters corresponding to linear bound variables (i.e. bound variables which occurs once in the body of the $\lambda$-abstraction.
- *Lambda-naming:* It gives a name to a $\lambda$-term ( e.g. $\lambda x.\lambda y.e$ becomes **let** $f = \lambda x.\lambda y.e$ **in** $f$) to prepare removal of a $\lambda$-abstraction.
- *Let-lifting:* It lifts a local let expression at the top level. It supposes that patterns are limited to case expressions e.g. the let constructions cannot use patterns. The transformation abstracts the free variables in let expressions following an algorithm described by Johnson [4] and then it can lift safely the let expressions at the top level.
- *Lambda-pulling:* The transformation pulled all the abstracted variables of a $\lambda$-abstraction named $f$ as arguments for $f$. For example $f = \lambda x.\lambda y.e$ becomes $f\ x\ y = e$. *Lambda-pulling* composed with *Lambda-naming* removes *syntactic* $\lambda$-abstractions from the program.
- *Lambda-pushing:* which conversely transforms a function definition with arguments as an abstraction. For example $f\ x\ y = e$ becomes $f = \lambda x.\lambda y.e$.

- *Eta-abstraction:* It is also called *eta-expansion.* The transformation expands $\lambda$-abstractions which returns functions , e.g $\lambda$-abstractions of type-order greater than 0, by supplying the variables that are missing to get a type-order 0. For example $\lambda x.e$ of type-order 2, becomes $(\lambda u.\lambda v.\lambda x.e)\ u\ v$.

## 1.4 Functions

Functions applications can be *folded*, *unfolded*, *specialized* with respect to some arguments, or *partially* evaluated.

- *Unfolding:* Unfolding is a basic technique in program transformation. A function application can be unfolded under certain conditions which ensures that the unfolding transformation terminates (*safety condition*), and that it does not imply generation of less efficient code (*effectiveness condition*). Unfolding is *rewriting.* It can be seen also as a *partial* evaluation with another point of view.

  For *effectiveness*, the conditions are the following:

  1. unfolding a variable parameter is *effective*,
  2. unfolding an expression parameter (not reduced to a variable) is *effective* if and only if the unfolding does not duplicate the expression. This implies that the parameter corresponds to a right linear argument.

  For *safety* a transformation strategy which uses unfolding must ensure that it always unfolds finitely.

  Obviously unfolding nonrecursive function is always safe. However, for code expansion concerns, only nonrecursive functions that are applied once can be safely and effectively unfolded.

  When the program is translated into a term rewriting system, the safety condition is the *termination property* of the term rewriting system. Though this property is not decidable, powerful orderings like recursive path orderings can ensure that some term rewriting systems are terminating. In this case, unfolding becomes safe. The term rewriting systems we can obtain from functional program which have their function declarations has a set of mutually recursive functions $\{f_1, \cdots, f_n\}$ declared as

$$f_1\ tc_{1,1}\ \cdots\ tc_{1,m_1} = t_1$$

$$\vdots$$
$$f_n\ tc_{n,1}\ \cdots\ tc_{n,m_n} = t_n$$

where $tc$ is a constructor term i.e. a term built only from constructors and variables. As such, it is a constructor-based term rewriting system. Recent work [?] on an ordering which is suitable for constructor-based term rewriting systems can be interesting for this respect.

- *Evaluation:* We dare not to say *partial evaluation* though it is exactly what the transformation we are proposing here is, but the name *partial evaluation* [?] has a broader meaning and a larger effect. Let us consider an eager semantic terminating program in a term rewriting system form. As in [?], we call *inductive* the parameters which are not reduced to a variable, in other words their top symbol is a constructor. An inductive parameter position is *subterm decreasing*, if the recursive calls arguments in the set of mutually recursive constructor-based declarations are strict subterms of the inductive argument. For example, in the constructor based definition of append:

$$append\ []\ y = y$$
$$append\ (x :: xs)\ y = x :: (append\ xs\ y)$$

  the position 1 is a subterm decreasing inductive position. An application which contains a constant arguments in inductive and subterm decreasing parameter positions can be safely unfolded. For example, the application $append\ [2; 3; 4]\ y$ can be evaluated by unfolding into $2 :: (3 :: (4 :: y))$. This is far too restrictive. This idea is extended for:

  1. a set of constant arguments in inductive parameter positions if the corresponding set of constructor-based arguments is decreasing according to a multiset extension or a lexicographical extension of the subterm ordering, and
  2. a set of arguments that matches the constructor terms of the inductive parameters.

- *Function specialization:* It can be interesting to specialize a function with respect to variable only parameters. The variable only criteria has been defined by Chin in [3].

  **Definition 1** *Given a set of mutually recursive functions* $\{f_1, \cdots, f_n\}$ *declared as*

$$f_1\ x_{1,1}\ \cdots\ x_{1,m_1} = t_1$$

$$\vdots$$
$$f_n\ x_{n,1}\ \cdots\ x_{n,m_n} = t_n$$

*the $k^{th}$ parameter $x_{i,k}$ of a function $f_i$ is variable only if and only if each recursive call is in the form $f_i\ e_1\ \cdots\ e_{m_i}$ in the body of the function declarations of $\{f_1, \cdots, f_n\}$ and has its $j^{th}$ argument as a* global variable *i.e. variables from the parameters $x_{i,j}$.*

This criteria ensures the safety of the unfolding which generates the definition of the specialized function by guaranteeing that the recursive call can be folded. Other parameter positions that can be specialized safely are the positions where the recursive calls have constant arguments. Let us call such positions *constant recursive position.* Effective arguments in application that are interesting to be specialized are the following:

- Spécialization of constant arguments in variable only positions or constant recursive positions permits some constant propagation among function applications. For example, consider the above declaration of *append.* The application *append* $x$ $[2;3;4]$ is specialized with respect to its second constant argument into $app_2\ x$ where:

$$app_2\ [] = [2;3;4]$$
$$app_2\ (x :: xs) = x :: (app_2\ xs).$$

- Spécialization of higher-order arguments in variable only positions or in constant recursive positions allows to decrease the type order of the functions in the program.

The variable-only specialization algorithm is driven by call sites. A dictionary of the specialized functions is maintained[1].

For each function application $f\ t_1\ t_2 \cdots t_m\ t_1\ t_2 \cdots t_n$ with higher-order variable only arguments $t_1, t_2, \cdots, t_n$ (with one at least not reduced to a variable) triggers the following transformation which:

- 1. computes the set $V$ of variables in $t_1, t_2, \cdots, t_n$,

[1]The dictionary must be transmitted to a first order algebraic transformation system like ASTRE so that, it can use specializations of laws on combinators or theorem for free.

2. *specialization:* if it does not exists yet in the dictionary, creates an equation of a new specialized function $f_{spec}$ of arity $m + size(V)$ by unfolding the right-hand side of the definition of *f*. This unfolding is realized by substituting the variable only higher-order argument to the variables that corresponds to the higher-order arguments in the right-hand side of the definition. The specialized function is a new entry in the dictionary. The call is updated with the new specialized function.

3. *folding:* if it exists in the dictionary, fold to update the call.

- *termination*

  **Proof:** Consider all the applications that needs to be specialized. There exists a finite number of higher-order variable only arguments of functions that could be specialized by the instances given by these applications. A specialization does not create another instance because of the variable only condition. Therefore the transformation must terminate. □

After specialization of a program of order 0, it remains higher-order applications with nonvariable only arguments, and nonconstant recursive arguments. It remains also higher order constructor applications.

*Function declaration cleaning:* After some transformations, for example after a *specialization*, useless function declarations can be removed from the program. A useless function is a function which is not called by the program expression. The calling relation is the transitive closure of the relation between functions defined as $f$ calls $g$ if and only if an application of $g$ occurs in the body of $f$. The main functions are the functions that occurs in the body of the program expression. The cleaning consists in keeping in the declaration list only the function declarations which are strongly connected to a main function.

## 1.5 Types

- *Explicit tuples:* For example, the transformation of an expression into a first-order term requires to put explicit constructors in place of tuples with an explicit arity. For that, the transformation introduces possibly infinite families of constructors $Tuple_2, Tuple_3, \cdots$. In ML, where the constructors are not currified, this transformation introduces a *Tuple* constructor under the constructors of arity greater than one. It is a source of obscurity for later program transformation when such a pattern is simplified, since then, the recursive call is placed by the transformation in a mutually recursive set of functions. For example, the naive declaration of the function *reverse* is:

$$\begin{aligned} reverse\ x = \textbf{case}\ x\ \ &\textbf{of} \\ &[] \Rightarrow [] \\ &(C\ (x, xs)) \Rightarrow append\ (reverse\ xs)\ [x] \end{aligned}$$

But with simple patterns, it becomes:

$$\begin{aligned} reverse\ x = \textbf{case}\ x\ \ &\textbf{of} \\ &[] \Rightarrow [] \\ &(C\ x) \Rightarrow f_{new}(x) \\ f_{new}\ x = \textbf{case}\ x\ \ &\textbf{of} \\ &(Tuple_2\ (x, xs)) = append\ (reverse\ xs)\ [x] \end{aligned}$$

In the first form, it is easy to see that the *continuation* of *reverse* is *append*, but it is less easy with the second form.

- *Enumeration:* This transformation enumerates for each declared polymorphic function the diverse type instances coming from the function applications. It also annotates the function declaration by these types. For example, given the polymorphic declaration $id\ x = x$ where $id : \alpha \leftarrow \beta$, the application $id\ 3$ updates the enumeration set of *id* with: $\{int \leftarrow int\}$, and the application $id\ sqr\ x$ allows to update the enumeration set of *id* as: $\{int \leftarrow int, (int \leftarrow int) \leftarrow (int \leftarrow int)\}$. Once enumeration is made, later transformations must update enumeration sets.

- *Expansion:* This transformation expands a polymorphic declaration with type annotations into different copies according to the types in the enumeration set. For the example above, it will make two copies of *id*: $id_1$ for the type instance $\{int \leftarrow int\}$ and $id_2$ for the type instance $(int \leftarrow int) \leftarrow (int \leftarrow int)$. It also updates the applications.

## 2 Strategies

A strategy is supposed to put the program on a form which is appropriate to its translation into a target language.

## 2.1 Monomorphization

The target program is nonpolymorphic, in other words its typing does not use type variables. The transformation can simply be composed of an enumeration transformation followed by an expansion transformation.

## 2.2 Lambda-lifting

Lambda-lifting puts the local let expressions and the local $\lambda$-abstractions at the top level. We suppose here that patterns occur only in case expressions. This transformation is done by: *Lambda-naming* followed by an abstraction of the free variables in let expressions (following mostly an algorithm described by Johnson [4]), then *Let-lifting:* the transformation pushes all the let expressions at the global level, and finally *Lambda-pulling:* the transformation pulled all the abstracted variables of a named $\lambda$-abstraction to the left-hand sides. At this point all the *syntactic* $\lambda$-abstractions are removed.

## 2.3 Defunctionalization

This transformation is used if the target language is first-order. When the language is polymorphic, the analysis of the program is made at application sites but if the language is nonpolymorphic, the higher-order function definitions can be transformed directly. The transformation is concerned by functions that returns functions, and functions that take functions as arguments.

- *Functional results:* A program without functional results is said to be *fully applied. Full-application* is called *full_-parametrization* in [3]. It is a variant of *eta-expansion* in a polymorphic language. In a polymorphic language, an argument can be functional or not, depending of the polymorphic instance. For example in *id* 2, the application of *id* returns an integer but in *id f* 6, the application *id f* returns a function. Therefore the program needs to be *monormophized* with respect to an enumeration of the application sites which returns functions. An expansion of polymorphic functions that returns functions in their applications need to be made. For the above example, it introduces a copy *id'* of the function *id* which takes 2 arguments: a functional argument and an integer. The application *id f* 6 is renamed $id'$ $f$ 6. In the transformed program, in each application of a function *f*, *f* has a number of arguments which is equal to its arity.

We suppose here that the transformation is applied on *lambda-lifted* programs. Full application allows the transformation of functions returning functions that are not encapsulated in a constructor or a tuple. In a nonpolymorphic program, full application is quite simple, it is enough to have function applications (which are not in parameter position) with a number of arguments equal to their arity. i.e.

$$f = \mathbf{add}\ 2 \text{ becomes } f\ x = add\ 2\ x$$

In a polymorphic program, as we show above, it is slightly more complicated, because the arity of a polymorphic function does not reflect its type order, e.g. the occurrence of *id* in the term $id\ g\ x$ has type instance $(\alpha \leftarrow \beta) \leftarrow \alpha \leftarrow \beta$. The subterm $id\ g$ returns a function. Here, the number of arguments of the application is greater than the arity of *id*. A way out, in this case, is a transformation we can call a *monomorphization by need.* It specializes higher-order functionals according to their instances in the program. For example, the term $id\ g\ x$ becomes $id_{new}\ g\ x$ where $id_{new}\ f\ y = f\ y$. Afterwards, the number of arguments of the application of $id_{new}$ is equal to its arity. The transformation is composed of:

1. *uncurrying:* Functions definitions are uncurried by looking for each right-hand sides equation of a function definition of type-order greater than 0. In such a case, there will be $\eta$-expansion (also called $\eta$-abstraction) followed by $\lambda$-pulling.
2. *case application:* The variables introduced by the $\eta$-expansion are pulled inside the cases so the type-order of function in applications which are not arguments is greater or equal to its arity.
3. *monomorphization by need:* It concerns only the functions which have applications where the number of arguments is greater than their arities.

   The two last steps has to be mutually recursive until full application is reached.

A term is then:

```
t::=v t1...tn |c t1...tn | f t1...tn
    | case t of p1 ->t1 | ... | pn -> tn
p::= v | c p1...pn
```

Moreover, after *full-application* transformation, function applications (except functional arguments) are of type-order 0.

*Argument for termination:*

> **Proof:** From a *monormophized* application other applications in the copy body have to be *monormophized.* However, the process is finite for the following reasons: Consider the set A of applications that have their type-order greater than their arity. We have in the program only a finite number of polymorphic variables susceptible to be *monormophized* and they can have only the instances asked by the set A. □

*Remark* Some functions that have been *monormophized* can be cleaned from the program.

- *Functional arguments:* Functional arguments can be removed by application of Reynolds' method. Reynolds' idea gives a general and complete method to generate a program where all the arguments of constant function applications are of type-order 0 given a *fully-applied* program. The target program is first-order when the program expression is of type-order 0. The transformation is easier if the source program is given in a *lambda-lifted* form.

  The transformation encodes families of functions which are argument of higher-order functions into elements of a (possibly recursive) sum declared by a data type. It defines an *apply* function taking account of each encoding corresponding to the sum. Following Reynolds, we call the transformation a *defunctionalization* [1].

  The functional arguments to remove can be:

  - *Higher-order arguments of function applications:* The higher-order arguments (that are not variables) in position $i$ of applications of $f$ must be encoded as different elements of a sum data type. In this encoding, a closed term is represented by a constant constructor, and a non closed term of set of variables $v_1, \cdots, v_n$ is represented by a constructor with variables type arguments which corresponds to the data-types of the $v_i^{\prime}s$ (it is a type variable for first order variables).

    An application of an encoded function (i.e a higher order global variable in a right-hand side of a definition), must be replaced by an application of a function *apply* to the encoding on its arguments. The body of the apply function is a case statement on the patterns which correspond to the different constructors of the sum type which encodes the function. Each constructor corresponds to a functional term. This functional term is unfolded in the arm of the case. It may happens that this unfolding uncovers some new applications of an encoded argument. Then it allows to define new occurrences of *apply* applications. This happens with higher-order functional arguments.

Each data-type of encoded functions creates different *apply* functions. Because of the polymorphism, it may happen that an argument $i$ has different types instances in different applications of $f$. Then, similar data-types must be constructed for the different instances. This will generate different *apply* functions as well and also different copies of the declaration of $f$. So the algorithm needs to know the types of the higher-order application arguments and needs to detect them. It also needs to look at the types of the arguments of function applications in order to create data-types. Afterwards it creates the new applications and the copies of the declaration that are driven by different applications. Then it creates the declaration of the *apply* functions by unfolding the higher-order term which is encoded. For example:

$$\begin{array}{lll} mp\ Z\ f\ x & = \textbf{case}\ x & \textbf{of} \\ & & [] \Rightarrow (f\ 0) :: [] \\ & & (x :: xs) \Rightarrow (f\ x) :: mp\ Z\ (Z\ f) xs \end{array}$$

$$\begin{array}{ll} db\ f\ x & = f\ (f\ x) \\ inc\ x & = x + 1 \\ mp\ db\ inc\ [2;3;4] & \end{array}$$

becomes:

$$\begin{array}{l} T_1 = D \\ T_2 = I \mid C\ \textbf{of}\ T_1 \times T_2 \end{array}$$

$$
\begin{array}{lll}
mps\ Z\ f\ x & = \textbf{case} & x\ \textbf{of} \\
 & & [] \Rightarrow apply_{mps_2}\ f\ 0) :: [] \\
 & & (x :: xs) \Rightarrow (apply_{mps_2}\ f\ x) :: \\
 & & mps\ Z\ (C\ (Z, f))\ xs \\
apply_{mps_2}\ f\ x & = \textbf{case} & f\ \textbf{of} \\
 & & I \Rightarrow inc\ x \\
 & & C(Z, g) \Rightarrow (apply_{mps_1}\ Z\ g\ x) \\
apply_{mps_1}\ Z\ g\ x & = \textbf{case} & Z\ \textbf{of} \\
 & & D \Rightarrow dbs\ g\ x
\end{array}
$$

$$
\begin{array}{ll}
dbs\ f\ x & = apply_{mps_2}\ f\ (apply_{mps_2}\ f\ x) \\
inc\ x & = x + 1 \\
mps\ D\ I\ [2, 3, 4] &
\end{array}
$$

The problem with the above transformation is that it multiplies the data-types and the *apply* functions. Some creation of new data-types could be avoided by reuse of existing standard types when isomorphic. For example, when a data-type is isomorphic to the data-type Nat, i.e. $T = C0 \mid C\ \textbf{of}\ T$, the algorithm could use constructors of the Nat type: 0 and **successor**, if they exist as such in the program. Another important case is isomorphism with lists: $T\ \textbf{of}\ \alpha = C0 \mid C\ \textbf{of}\ \alpha \times T$. This kind of improvement of the method can be made afterwards as shown in [**?**].

A constructor term is obviously an encoding of a closure and an application of *apply* forces the evaluation of a closure. The transformation does not improve the efficiency of the program since it can be seen easily that the reduction paths are not changed by the encodings. However it removes the overhead that comes from the use of higher-order functions in the program.

Notice that a certain amount of *monomorphization* is done by the transformation.

– *Functional arguments in constructor applications:* A constructor can have functional arguments if it belongs to a polymorphic data-type: e.g. *List* **of** $\alpha$ which has *List* **of** $(\alpha \to \beta)$ as an instance. A constructor can also be directly specified in a data-type as having a functional argument (arrow type). Moreover, there can be functional elements in a product.

Functions which are argument of a constructor application are encoded in a data-type in the same way than functions in argument of a functional application. But here the constructor with a functional type must be also updated accordingly. For example look at the constructor *Store* in the following:

$$(\alpha, \beta)\ store = Store\ \textbf{of}\ \alpha \rightarrow \beta$$

with

$$\begin{array}{ll} initstore & = Store\ init \\ init\ x & = 0 \\ fetch\ x\ (Store\ f) & = f\ x \\ update\ (Store\ f)\ x\ a & = Store\ (assoc\ [(x, a)]\ f) \end{array}$$

and

$$\begin{array}{lll} assoc\ l\ f\ y = & \textbf{case}\ l & \textbf{of} \\ & & [] \Rightarrow f\ y \\ & & (x, a) :: xs \Rightarrow if\ x = y \\ & & then\ a\ else\ assoc\ xs\ f\ y \end{array}$$

in

$$update\ initstore\ "x"\ 2$$

which is transformed into:

$$\alpha\ store = Store\ \textbf{of}\ \alpha$$

with

$$T = C0 \mid C\ \textbf{of}\ (string \times\ int \times\ T)$$

and

$$\begin{array}{ll} initstore & = Store\ C0 \\ init\ x & = 0 \\ fetch\ x\ (Store\ f) & = apply\_T\ f\ x \\ update\ (Store\ f)\ x\ a & = Store\ C(x, a, f) \end{array}$$

and

$$\begin{array}{l} apply_T\ f\ y = \textbf{case}\ f\ \textbf{of} \\ \quad C0 \Rightarrow 0 \\ \quad T(x, a, f) \Rightarrow \\ \quad\quad assoc\ [(x, a)]\ f\ y \end{array}$$

and

$$\begin{array}{l} assoc\ l\ f\ y = \textbf{case}\ l\ \textbf{of} \\ \quad [] \Rightarrow apply_T\ f\ y \\ \quad x :: xs \Rightarrow if\ x = y \\ \quad then\ a\ else\ assoc\ xs\ f\ y \end{array}$$

in

$$fetch\ z\ (update\ initstore\ "x"\ 2)$$

Note that it is possible to unfold the application of *assoc* in $apply_T$. The result of this unfolding is the following:

```
$apply_T\ f\ y =$ case $f$ of
        $C0 \Rightarrow 0$
        $T(x, a, f) \Rightarrow$
            $if\ x = y\ then\ a\ else\ apply_T\ f\ y$
```

Afterwards *assoc* can be removed by cleaning. So we see that the transformation can allow further simplifications by unfolding functions in the arms of *apply* functions.

We can also have functions which are encapsulated under a polymorphic type as in list of functions. In [3], Chin signals that these functional arguments can sometimes disappear by higher-order deforestation. However, a defunctionalization algorithm possibly followed by a first-order deforestation is more general. In the following example, the the constructor of the polymorphic type itself is higher-order.

```
$m\ xs\ y$      $= maph\ (add5\ xs)\ y$
$maph\ lf\ y$=  case $lf$ of
              $[] \Rightarrow []$
              $f :: fs \Rightarrow (fy)\ ::\ (maph\ fs\ y)$
$add5\ l$       = case $l$ of
              $[] \Rightarrow []$
              $x :: xs \Rightarrow (kx)\ ::\ (add5\ xs)$
$kxz$           $= z + 5 * x$
```

Defunctionalization also removes functions that are arguments of constructor applications. By defunctionalization, the above example is transformed into the following:

$$\alpha\ T = Fun\ \textbf{of}\ \alpha$$

```
$m\ xs\ y$        $= maph\ (add5\ xs)\ y$
$maph\ lf\ y$ $=$ $\textbf{case}\ lf\ \textbf{of}$
                  $[] \Rightarrow []$
                  $x :: fs \Rightarrow (apply\ x\ y) :: (maph\ fs\ y)$
$apply\ f\ y$     $= \textbf{case}\ f\ \textbf{of}$
                  $Fun\ x \Rightarrow k\ x\ y$
$add5\ l$         $= \textbf{case}\ l\ \textbf{of}$
                  $[] \Rightarrow []$
                  $x :: xs \Rightarrow (Fun\ x) :: (add5\ xs)$
$k\ x\ z$         $= z\ +\ 5 * x$
```

Now first-order deforestation applies:

```
$m\ xs\ y =$       $maph'\ xs\ y$
$maph'\ xs\ y =$ $\textbf{case}\ xs\ \textbf{of}$
                   $[] \Rightarrow []$
                   $x :: xs \Rightarrow (apply\ x\ y) :: (maph'\ xs\ y)$
$apply\ f\ y =$    $\textbf{case}\ f\ \textbf{of}$
                   $Fun\ x \Rightarrow k\ x\ y$
```

An unfolding of *apply*, and the new type becomes useless!.

– *Functions encapsulated into tuples:* A function can also be encapsulated into a tuple as in the following example:

```
$fmin\ t =$ $\textbf{case}\ t\ \textbf{of}$
    $Leaf\ a \Rightarrow$         $(Leaf, a)$
    $Tree\ \ (t1, t2) \Rightarrow$
        $\textbf{case}\ (fmin\ t1)\ \textbf{of}$
        $(f1, m1) \Rightarrow$ $\textbf{case}\ (fmin\ t2)\ \textbf{of}$
                         $(f2, m2) \Rightarrow (k\ f1\ f2, min(m1, m2))$

          $k\ f\ g\ x =$   $Tree\ (f\ x, g\ x)$
          $mintree\ t =$ $\textbf{case}\ (fmin\ t)\ \textbf{of}$
                         $(f, m) \Rightarrow (f\ m)$
```

This can be transformed as a special case of functional constructor application. The above programs becomes:

$$T = C0 \mid C\ \textbf{of}\ T\ \times\ T$$

```
fmin t =  case t of
       Leaf a ⇒          (C0, a)
       Tree (t1, t2) ⇒
             case (fmin t1) of
             (f1, m1) ⇒  case (fmin t2) of
                         (f2, m2) ⇒ (C(f1, f2), min(m1, m2))

             apply f m =case f of
                        C0 ⇒ (Leaf m)
                        C(f1, f2) ⇒ (k f1 f2 m)
             k f g m =  Tree(apply f m, apply g m)
             mintree t = case (fmin t) of
                        (f, m) ⇒ (apply f m)
```

A transformation algorithm based on the above ideas will encode all higher-order arguments of constant functions. Moreover it can also encode functions encapsulated as arrows into data type or into polymorphic data-types and tuples. However it multiplies the data-types and it is better to transform the applications to higher-order arguments by *function specialization* when possible.

## 2.4 Constructor based term rewriting system

The goal is to transform a functional program into a constructor-based term rewriting system.

A rewrite system is *constructor-based*[2] if all proper subterms of its left-hand sides have only *free constructor* symbols and variables. The roots of left-hand sides are *defined symbols.*

First, we transform the program into a possibly higher-order language restricted in the following way:

- Patterns are limited to case expressions.
- Patterns are simple.
- Function applications are simple. That is, case expressions do not occur as operators in function applications.
- There are no lambda or let expressions.

[2] A constructor-based system of equalities is similar to set of definition equalities with pattern-matching arguments in functional programming.

- The term evaluated by a case expression must be fully-applied. Thus, `case add 1 of f => f 2` is not allowed.

The grammar of such a language is the following:

|  |  |  |  |
|---|---|---|---|
| Datatypes: | *ddecl* | $::=$ `datatype` $\alpha_1 \ldots \alpha_n$ `T` $=$ *cdecs* |  |
|  | *cdecs* | $::=$ *cdec* $\mid$ *cdec* $\mid\mid$ *cdecs* |  |
|  | *cdec* | $::=$ `C` $type_1 \times \ldots \times type_n$ | $(n \geq 0)$ |
| Function declarations: | *fdecls* | $::=$ *fdecl* $\mid$ *fdecl* `and` *fdecls* |  |
|  | *fdecl* | $::=$ `f v1...vn` $=$ *term* |  |
| Terms: | *term* | $::=$ *rator* $term_1 \ldots term_n$ | $(n \geq 0)$ |
|  |  | $\mid$ `case` *term* `of` $pat_1$ => $term_1$ $\mid\mid \ldots$ $\mid\mid$ $pat_n$ => $term_n$ |  |
|  | *rator* | $::=$ `f` $\mid$ `v` $\mid$ `C` |  |
|  | *pat* | $::=$ `C v1 ...vn` | $(n \geq 0)$ |
|  |  | $\mid$ `(v1,v2)` |  |
| Types | *type* | $::= \alpha \mid type_1 \rightarrow type_2 \mid$ `T` $type_1 \ldots type_n$ |  |
|  |  | $\mid type_1 \times type_2$ |  |

For that the program is transformed

1. by removing *commodity* patterns, *trivial ifs*, and *trivial cases*, then
2. by translating the conditionals into *cases*
3. by *limiting the patterns* to case expressions, then
4. by removing functional cases using *case application*, then
5. by *conversion to simple patterns*, then
6. by *lambda-lifting* which removes the syntactic $\lambda$-abstractions and local let expressions, then
7. by *case application* so that each *case* is fully-applied, then
8. by removing the overlapping patterns in case expressions so that no patterns are reduced to a variable.

Second, the program has to be first-order, so *defunctionalization* transformations must be performed. At this point the grammar of the language is slightly modified: the case $v$ disappears in the *rator* rule.

Third, it remains:

1. to limit case test expression to variables then the grammar rule for term is changed into:

   $$\begin{array}{lll} term & ::= rator\ term_1 \ldots term_n & (n \geq 0) \\ & \quad \mid \texttt{case v of}\ pat_1 \Rightarrow term_1 \mid\mid \ldots & \\ & \quad \mid\mid pat_n \Rightarrow term_n & \end{array}$$

   , and

2. finally patterns can be limited to functions. Then the *case* disappear from the *term* rule, and the patterns appears in the *fdecl* rule.

The final grammar is the following:

$$\begin{array}{llll}
\text{Datatypes:} & ddecl & ::= \texttt{datatype}\ \alpha_1 \ldots \alpha_n\ \texttt{T}\ =\ cdecs & \\
 & cdecs & ::= cdec\ \mid\ cdec \mid\mid cdecs & \\
 & cdec & ::= \texttt{C}\ type_1 \times \ldots \times type_n & (n \geq 0) \\
 & & & \\
\text{Function} & & & \\
\text{declarations:} & fdecls & ::= fdecl \mid fdecl\ \texttt{and}\ fdecls & \\
 & fdecl & ::= \texttt{f}\ \texttt{pat}_1 \ldots \texttt{pat}_\texttt{n}\ = term & \\
 & & & \\
\text{Terms:} & term & ::= rator\ term_1 \ldots term_n & (n \geq 0) \\
 & & \quad \mid\mid pat_n \Rightarrow term_n & \\
 & rator & ::= \texttt{f} \mid \texttt{C} & \\
 & pat & ::= \texttt{C v1} \ldots \texttt{vn} & (n \geq 0) \\
 & & \quad \mid \texttt{(v1,v2)} & \\
 & & & \\
\text{Types} & type & ::= \alpha \mid type_1 \rightarrow type_2 \mid \texttt{T}\ type_1 \ldots type_n & \\
 & & \quad \mid type_1\ \times type_2 &
\end{array}$$

## 3 Conclusions

We have presented a tool kit of basic transformations for a pipeline of transformation from a higher-order polymorphic strongly typed language towards a first-order nonpolymorphic functional language. This can be useful if the target language is a nonpolymorphic strongly typed language and or first-order like ADA or PASCAL or MODULA. It can be also useful to perform transformations which can be easily done on a first-order language. It was the case of Wadler's deforestation algorithm and later of Chin's deforestation algorithm [**?**, 2, 3]. It is also the case of algebraic transformations that are based on term rewriting techniques.

## References

[1] T. Arts and H. Zantema. Termination of constructor system using semantic unification. Unpublished work, 1995.

[2] L. Auguston. A compiler for lazy ml. In ACM, editor, *Proceedings of the 1984 Lisp and Functional Programming conference*, pages 218–227, 1985.

[3] L. Auguston. Compiling pattern matching. In Springer Verlag, editor, *Proceedings of the conference on Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 368–381, 1985.

[4] J. M. Bell and J. Hook. Defunctionalization of Typed Programs. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1994.

[5] F. Bellegarde. Automatic Synthesis by Completion. In *Journées Francophones sur les Langages Applicatifs*, INRIA, collection didactiques, 1995.

[6] Françoise Bellegarde and James Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

[7] W. Chin. Safe Fusion of Functional Expressions. *Proceedings of the ACM Symposium on Lisp and Functional Programming, San Francisco, Ca.*, pages 11–20, June 1992.

[8] W. Chin and J. Darlington. Higher-Order Removal: A modular approach. To be published, 1994.

[9] C. Consel and O. Danvy. Tutorial note on partial evaluation. In *Twentieth Annual ACM Symposium on Principle of Programming Languages*, pages 493–501, 1993.

[10] T. Johnson. Lambda Lifting: Transforming Programs to Recursive Equations. In Springer Verlag, editor, *Proceedings of the conference*

*on Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 190–203, 1985.

[11] P. Wadler. Deforestation: Transforming Programs to eliminate trees. *Theoritical Computer Science*, 73:231–284, 1990.