A Transformation System Combining Partial Evaluation with Term Rewriting *Revision: 1.1

Françoise Bellegarde,
Computer Science,
Oregon Graduate Institute of Science & Technology,
PO BOX 91000,
Portland, OREGON, 97006.
Bellegar@cse.ogi.edu

Abstract

This paper presents a new approach to optimizing functional programs based on combining partial evaluation and rewriting. Programs are composed of higher-order primitives. Partial evaluation is used to eliminate higher-order functions. First-order rewriting is used to process the transformation. Laws about the higher-order primitives that are relevant for the optimizations are automatically extracted from a library and transformed into first-order terms using partial evaluation. Such a combination of a partial evaluation system and an intrinsically first-order rewriting tool allows a form of higher-order rewriting at a first-order level. This way, it is possible to automate deforestation of higher-order programs.

Introduction

The so-called Squiggol [10] style for program construction is a high-level programming technique that consists of building a program by composing primitives or other functions while taking into account well-known laws on the primitives. Functions are usually defined according to recursion patterns that are attached to the inductive structure of the data types. These recursive patterns can be captured by higher-order functions. For example a particular kind of recursive pattern attached to the recursive data type list called catamorphism [10] can be captured by the higher-order primitive *foldr*. This higher-order primitive is provided in most functional languages. It is called *fold* in ML and *reduce* in Common Lisp.

Although constructing programs by composing higher-order primitives provides the user with a high degree of abstraction, it comes at the expense of efficiency. Indeed, compositions produce many intermediate data structures when computed in an eager (call by value) evaluation. One way to circumvent this problem is to perform *deforestation* on programs as advocated by Wadler [16]. Deforestation algorithms [16, 9], as well as algorithms based on

---

promotion theorems [12] eliminate these useless intermediate data structures, but the optimizations they perform are limited because they do not take into account any particular laws about the operands of the compositions. As described in [17], laws about higher-order polymorphic primitives can be derived *for free* from their type. When appropriately defined, laws can constitute a powerful calculus to derive efficient programs from higher-level specifications. The full paper describes an automatic process that uses laws on higher-order primitives to perform an extended form of deforestation.

Related work

Wadler has proposed an algorithm for deforestation in [16]. It works intrinsically on first-order programs though it is extended to higher-order programs by treating higher-order programs as macros. His algorithm performs automatic deforestation on *treeless* terms. Chin's work on fusion [5] applies to higher-order program in general, it skips over terms to which the technique does not apply. More recently, promotion theorems have been applied to normalize programs [12]. This technique is applicable to *potentially normalizable terms* which are similar to *treeless* terms. A new automatic way to implement deforestation inside the Haskell's compiler is shown in [9].

General purpose program transformation systems are based on a folding-unfolding strategy *à la* Burstall and Darlington [4]. Deforestation is a particular instance of this strategy. In the Focus system [11], folding and unfolding are seen as term rewritings. It has been pointed out in [7] how a folding-unfolding strategy can be directed by a completion procedure. Following this idea, the transformation system Astre [1, 2] is based on *partial* completion procedures[1]. Astre takes into account of inductive laws provided by the user during the completion process. All these systems are interactive. However a currently implemented *automatic mode* of Astre performs automatically a simple [2] deforestation of a program. In this mode, it has the same functionality and the same limitations as the above deforestation algorithms. Both Focus and Astre are based on first-order term rewriting techniques therefore they are limited to first-order programs.

The paper

We describe a way to mimic an extended form of deforestation [3] of higher-order functional programs using first-order rewriting. This is achieved by using partial evaluation to transform a class of higher-order programs into first-order ones.

Partial evaluation aims at specializing a program with respect to part of its input (static parts). This process produces a specialized (residual) program [8]. This specialized program when applied to the remaining input value parts (dynamic parts) yields the same result as the original program applied to a complete input. In this paper we are using Schism [6] a partial evaluator for pure functional programs. Our goal is to use partial evaluation to eliminate higher-order functions by specializing higher-order primitives with respect to their functional arguments.

---

[1] The completion is *partial* because it computes only part of the superpositions between rewrite rules.

[2] The deforestation is said to be *simple* if its processing does not use particular laws between the functions and primitives.

[3] The deforestation is said to be *extended* if it can be achieved only by using particular laws between the functions and primitives.

The objective of this paper is to show how a large class of higher-order programs can be automatically improved by using powerful laws on higher-order primitives, partial evaluation, and term-rewriting. To our knowledge, no general purpose transformation system supports this kind of transformation.

1   Maxsub example

We illustrate our transformational approach with a program presented by S. Thompson [14] and first introduced by J. Bentley [3]. The problem solved by this program is stated as follows by Thompson:

> Given a finite list of numbers, find the maximum value for the sum of a (contiguous) sublist of the list.

Numbers can be positive as well as negative integers. In his book [15], S. Thompson presents formally the derivation of a functional program that achieves the computation described above. It is displayed in Figure 1. This program is quadratic in the length of the list.

In this program, [] is the empty list, :: is the constructor (often called *cons*) of the data type list and *consl* is its prefix version. The binary operation @ produces the concatenation of two lists. The operation ○ is the composition of two functions.

This program also assumes that the higher-order operators *fold*, *map*, *foldr* and the function *bimax* are primitive operations. The function *bimax* is the maximum function. The operation + is prefixed. For example:

$$foldr \; + \; 0 \, [2, -3, 4, 3] = \, + \, 2 \, (+ \, (-3) \, (+ \, 4 \, (+ \, 3 \, 0))) = 6.$$

The operator *fold* is similar to *foldr*, except that it only applies to non empty lists. For example:

$$fold \; bimax \; [2, -3, 4, 3] = bimax \; 2 \; (bimax \; (-3) \; (bimax \; 4 \; 3)) = 4.$$

The function *sublists* returns all the contiguous sublists of a given list. The function *frontlists* returns all the sublists that are prefixes of the list. For example:

$$frontlists \, [2, -3, 4, 3] = [[2, -3, 4, 3], [2, -3, 4], [2, -3], [2], []].$$

Faithful to the Squiggol method, *maxsub* takes all the contiguous sublists of the list using *sublists*, then computes all the sum of the elements of every sublists using *map sum*, finally computes their maximum using *fold bimax*.

The final result of the transformation is the functional program shown in Figure 2 which is linear. The theorems used during the transformation process are listed in Figure 3.

The details of the manual transformations using these laws can be found in [14]. Let us now consider, a way to automate this transformation using completion and partial evaluation.

The system Astre [1, 2] based on a partial completion procedure, deals with programs presented by a set of first-order equations. A partial evaluator can automatically convert a class of higher-order programs into first-order ones.

$$
\begin{aligned}
maxsub &= (fold\ bimax) \circ (map\ sum) \circ sublists \\
sum &= foldr + 0 \\
sublists\,[] &= [[]] \\
sublists\,(a :: x) &= (map\ (consl\ a)\ (frontlists\ x))\ @\ (sublists\ x) \\
frontlists\,[] &= [[]] \\
frontlists\,(a :: x) &= (map\ (consl\ a)\ (frontlists\ x))\ @\ [[]]
\end{aligned}
$$

Figure 1: program source

$$
\begin{aligned}
maxsub\,[] &= 0 \\
maxsub\,(a :: x) &= (bimax\ (+\ a\ (maxfront\ x))\ (maxsub\ x)) \\
maxfront\,[] &= 0 \\
maxfront\,(a :: x) &= (bimax\ (+\ a\ (maxfront\ x))\ 0)
\end{aligned}
$$

Figure 2: final program

$$
\begin{aligned}
maplaw : &\quad map\ f \circ map\ g = map\ (f \circ g) \\
map\,@law : &\quad map\ f\ (x@y) = (map\ f\ x)\ @\ (map\ f\ y) \\
fold\,@law : &\quad fold\ f\ (x@y) = f\ (fold\ f\ x)\ (fold\ f\ y) \\
&\qquad\qquad \text{if f is associative and} \\
&\qquad\qquad \text{x and y are non empty lists} \\
fold\,maplaw : &\ fold\ f \circ map\ g = g \circ fold\ f \\
&\qquad\qquad \text{if } f\ (g\ x)\ (g\ y) = g\ (f\ x\ y)
\end{aligned}
$$

Figure 3: laws for the transformations

2 Conversion to first-order programs

In order to understand the partial evaluation process, consider a functional program expressed as a *typed* λ-term M. Some of the arguments (static arguments) of functions are substituted by values according to their respective types. Let this substitution be δ, the partial evaluation operates as if it normalizes $\delta(M)$ by β-reduction and evaluates by unfolding the fixpoint operator when possible. We denote by $M \downarrow_{beta}$ the beta-normal form of the λ-term M.

For example, consider the function *append*, that is, the prefix version of the operator @ used in the *maxsub* example. The definition is:

$$\begin{aligned} append\,[]\,y &=& y \\ append\,(a::x)\,y &=& a::(append\,x\,y) \end{aligned}$$

A partial evaluation of the term $(append\,[1,2,3]\,y)$, where the first argument of *append* is the static argument substituted by $[1,2,3]$, returns the specialized definition:

$$append1\,y = 1::(2::(3::y))$$

The evaluation involves unfolding the recursive definition of *append*. But the partial evaluation of the term $(append\,x\,[4,5,6])$, where the second argument of *append* is the static argument substituted by $[4,5,6]$, returns the specialized definition:

$$\begin{aligned} append2\,[] &=& [4,5,6] \\ append2\,(a::x) &=& a::(append2\,x) \end{aligned}$$

In this case unfolding the recursion is impossible, only β-reduction is involved. Similarly, consider a partial evaluation of the term $(map\,sum\,x)$ given the following definition of *map*:

$$\begin{aligned} map\,f\,[] &=& [] \\ map\,f\,(a::x) &=& (f\,a)::(map\,f\,x) \end{aligned}$$

It returns the following specialization of *map* with respect to the static argument f substituted by the value *sum*:

$$\begin{aligned} map1\,[] &= [] \\ map1\,(a::x) &=& (sum\,a)::(map1\,x) \end{aligned}$$

Note that the partial evaluation need not unfold the recursive definition of *map* because the recursive call applies to a variable function f. It only specializes the definition of *map f* with respect to a sustitution of f by *sum*. It is important not to forget that *map1* is a specialization of *map sum* in order to be able to know that the higher-order laws on *map* apply to *map1* during the transformation of the converted program. This information consists of a triple:

$$(map1, \{f \leftarrow sum\}, map\,f)$$

In the following, RHOP will denote the set of the symbols of the recursive higher-order primitives like *map*, *fold*, *foldr*..., and κ will be the substitution unfolding the definitions of the RHOP symbols. We assume that the definitions of the RHOP functions are not mutually recursive. The $\beta-$normalization of a λ-term term M is noted by $M \downarrow_{\beta}$.

Definition 1 *A specialization triple is a triple (F, δ, T) where F is the symbol of a first-order specialization of a RHOP function, δ is a specialization substitution, and T is a specialized term such that a definition of F is obtained by folding every occurrence of $\delta(T)$ in the term $s = \delta(\kappa(T)) \downarrow_\beta$.*

We note by $s[t \backslash F]$ the replacement of each occurrence of the subterm t in the term s by the symbol F (folding of F). Then, the definition of F according to a specialization triple (F, δ, T) is:

$$F = \delta(\kappa(T)) \downarrow_{beta} [\delta(T) \backslash F]$$

Triples (F_i, δ_i, T_i) and (F_j, δ_j, T_j) such that δ_i, δ_j are α-convertible and T_i and T_j are also α-convertible are said to be α-convertible. In this case, F_i and F_j must be the same so that double definitions are not introduced.

Let $\rightarrow^{first}$ be the relation between λ-term and set of triples defined as follows:

Definition 2 $M_{ST} \rightarrow^{first} M'_{ST'}$ *if and only if there exists an occurrence in M of a subterm $F \, U_1 \, U_2 \, \cdots \, U_n$ where F is a symbol of a RHOP function, and U_i are terms instantiating all the functional arguments of F, then $M' = M[F \, U_1 \, U_2 \, \cdots \, U_n \backslash F_1]$ where F_1 is the symbol of the specialization of F with respect to the arguments $U_1, U_2, \cdots, U_n$. There can be two cases:*

- *either there exists in ST a specialization triple (F_1, δ, N), then $ST' = ST$,*

- *or, the adequate specialization triple does not exist in ST: In this case a new specialization triple is created for a new specialized function symbol F_1. The set of specialization triple ST' is then:*

$$ST \cup \{(F_1, \delta = \{(f_1 \leftarrow U_1), (f_2 \leftarrow U_2), \cdots, (f_n \leftarrow U_n)\},$$
$$N = \lambda \, x_1 \, x_2 \cdots x_m \, (F \, f_1 \, f_2 \cdots f_n))\}$$

where $x_1, x_2, \cdots, x_m$ are the free variables in the λ-term terms U_i.

We will omit the subscript by the set of specialization triples ST and we will consider the updating of ST as a side effect. Then $M \downarrow_{first}$ denotes a normalization of M by the relation $\rightarrow_{first}$.

The conversion to first-order processed by the partial evaluator for the class of programs we are considering can be viewed as: $M \downarrow_{first}$. The class of higher-order programs that are tackled by our approach can be characterized as follows. These programs consist of first-order terms and *constant or variable-only* higher-order primitives. *Variable-only* higher-order primitives are functions whose higher-order arguments are solely made up of variables in each recursive call to this function. This is called the *variable-only* criterion by Chin [5]. In the context of partial evaluation this criterion ensures that specializing functions with respect to higher-order values always terminates. For example *map*

$$map \, f \, (a :: x) \; = \; (f \, a) :: (map \, f \, x)$$

is *variable only*, so are *fold* and *foldr*. A function defined by :[4]

$$mapTwo \, f \, g \, [] \; = \; [] \; ; \; mapTwo \, f \, g \, (a :: x) \; = \; (f \, a) \; :: \; mapTwo \, g \, f \, x$$

---

[4] This example comes from an anonymous referee

is also variable only (the specialization of $mapTwo$ $f1$ $g1$ into $mapTwo1$ calls a specialization of $mapTwo$ $g1$ $f1$ into $mapTwo2$ which itself calls $mapTwo1$). But a function H defined by:

$$H\ f\ (a :: x)\ =\ (f\ a) :: (H\ (f\ \circ\ f)\ x)$$

and

$$G\ f\ (a :: x)\ =\ (f\ a) :: (G\ (sqr)\ x)$$

are not. However G is *constant-only* because its functional argument in the recursive call is a constant. The variable-only criteria is different from the restrictions given to higher-order functions that can be defined using higher-order macros as advocated in [16]: the functional arguments of these higher-order primitives must be fixed or unchanged across recursion. This criteria excludes $mapTwo$ and H but includes G.

Theorem 1 *Assuming that the non RHOP functions in M are first-order, that the RHOP functions are* constant *or* variable-only, *then M has a unique (modulo α-conversion) normal form $M \downarrow_{first}$, and all the function calls in the normal form are calls to first-order functions.*

The theorem ensures the soundness of the *conversion to first-order* process.

    Proof: The only higher-order recursive functions are the RHOP functions. They are called in M by first-order functions. Let F be a RHOP function and $F\ U_1\ U_2, \cdots U_n$ be an occurrence of a call of F in M. The terms U_i instantiate all the functional arguments of F because the functions in M are first-order. This call to a higher-order function must disappear. For that, it must correspond to a potential application of the relation $\rightarrow_{first}$. Let $x_1, x_2, \cdots, x_m$ be the free variables in the U_i. Such a call must correspond to a specialization triple

$$(F_1, \delta = \{(f_1 \leftarrow U_1), (f_2 \leftarrow U_2), \cdots, (f_n \leftarrow U_n)\},$$
$$N = \lambda\ x_1\ x_2 \cdots x_m\ (F\ f_1\ f_2 \cdots f_n))$$

where $F_1 = \delta(\kappa(N)) \downarrow_{beta} [\delta(N) \backslash F_1]$ according to the definition of $\rightarrow_{first}$ We need to prove that we can define such a function F_1. The symbol F is a RHOP function. By hypothesis, a recursive call of F in the λ-term term defining F invokes variables only or constants. The functional variables are substituted by the U_i, therefore, if the variables in the recursive call appears in the same order and are not duplicated in the recursive calls, they can be *folded* into calls to F_1. If they are swapped (with eventual repetitions) like in the example of $mapTwo$ above, a sequence of specialization will eventually generate mutually recursive specializations according to the number of inversions of the permutation. If there is a contant argument, another specialization which also generates mutually recursive specializations is necessary. The *beta*-reduction itself must terminate. Therefore, the relation $\rightarrow_{first}$ applies, and the term M' such that $M \rightarrow M'$ contains one occurrence of a higher order call less than M. Therefore normalization eliminates every call to RHOP functions from M, replacing them by calls to their specializations. The result of eliminating these calls is a first-order term when the specialized functions definition (definition of F_1 for the call above) does not contain any call to a higher-order function. It is obviously the case when the definition of the higher-order function

F, which is specialized, contains no calls to another RHOP function. If it does contain such a call, it instantiates all its functional arguments and this call can be the initiation of a specialization in the same way than above. The process is terminating because the RHOP functions are not mutually recursive. This proves that the specialization process is terminating and that the result is a first-order program. The normal-form $M \downarrow_{first}$ exists. Finally, the calls to RHOP functions are distinct and fully applied in M, thus the order we treat the calls does not change the result. This gives unicity of the result modulo α-conversion. $\square$

For the *maxsub* example, the first-order conversion uses the following set of triples ST:

$$
\begin{aligned}
\{(map1, &\{f \leftarrow sum\} & ,&\lambda\, x.(map\ f\ x)) \\
(map2\ , &\{f \leftarrow (consl\ x)\}, &&\lambda\, l\ x.(map\ f\ l)) \\
(fold1\ , &\{f \leftarrow bimax\ \} & ,&\lambda\, x.(fold\ f\ x))\}
\end{aligned}
$$

A partial evaluator performs only the adequate specializations of the RHOP functions, there is no unfolding of recursive calls: Given M, its result is the normal form $M \downarrow_{first}$. We are currently using the partial evaluator Schism [6] for first-order conversion. Schism uses a typed dialect of Scheme as its object language. A translator from ML to Schism is available.

Modulo some renaming, the result of partial evaluation of the *translated* program for *maxsub* is shown in Figure 4. The constructor :: becomes **cons**, [] becomes $'()$ in Scheme.

However, this program cannot be the input of a transformation system based on rewriting such as Astre. Even if we forget about the differences of syntaxes, an equation like:

$$
\begin{aligned}
(map1\ l) = &\ \mathbf{if}\,(\mathbf{null?}\ l)\ \mathbf{then}\ [] \\
&\ \mathbf{else}\,(sum\,(\mathbf{car}\ l)) :: (map1\,(\mathbf{cdr}\ l))
\end{aligned}
$$

gives a non terminating rewriting rule. A translator from Schism to ML is currently implemented. This translator will reintroduce the patterns that the translator from ML to Schism has eliminated. This way, the *first-order converted* program is translated into the first-order set of equations shown in Figure 5.

3 First-Order Transformation

The transformation can now begin at a first-order level using the system Astre. Let us run Astre giving the equations in Figure 5. The reader can refer to [2] to get a precise idea of the possibilities of the current version of the system. First, Astre turns the set of equations into a rewriting system, directing the equations from left to right automatically and eventually, (if it is required by the user) it verifies its ground convergence. Let us consider now the transformation process. The main function *maxsub* is defined in a Squiggol way by composition of functions and not inductively. The term

$$
t = fold1\,(map1\,(sublists\ l))
$$

is a candidate for a deforestation [16]. Deforestation algorithm of P. Wadler will not perform the deforestation of t because it requires the function definitions of *fold1*, *map1*, and *sublists* that occurs in t to be *treeless*. *fold1* and *sublists* are not *treeless* because their definitions contain applications of functions to terms that are not variables such as the application of *bimax* in

(**program** () () (

(**define**	(*maxsub l*)
	(*fold1* (*map1* (*sublists l*))))
(**define**	(*sublists l*)
	(**if** (**null?** *l*) **then** (*cons* '() '())
	else (@ (*map2* (*frontlists* (**cdr** *l*)) (**car** *l*)) (*sublists* (**cdr** *l*)))))
(**define**	(*frontlists l*)
	(**if** (**null?** *l*) **then** (*cons* '() '())
	else (@ (*map2* (*frontlists* (**cdr** *l*)) (**car** *l*)) (**cons** '() '()))))
(**define**	(*map2 l x*)
	(**if** (**null?** *l*) **then** '()
	else (**cons** (**cons** *x* (**car** *l*)) (*map2* (**cdr** *l*) *x*))))
(**define**	(*map1 l*)
	(**if** (**null?** *l*) **then** '()
	else (**cons** (*sum* (**car** *l*)) (*map1* (**cdr** *l*)))))
(**define**	(*fold1 l*)
	(**if** (**null?** (**cdr** *l*)) **then** (**car** *l*)
	else (*bimax* (**car** *l*) (*fold1* (**cdr** *l*)))))
(**define**	(*sum l*)
	(**if** (**null?** *l*) **then** '0
	else (+(**car** *l*) (*sum* (**cdr** *l*)))))

(**definePrimitive** +)
(**definePrimitive** *bimax*)
(**definePrimitive** @)
))

Figure 4: First-Order converted program source

$$
\begin{aligned}
maxsub\ l &= fold1\ (map1\ (sublists\ l))\\
sublists\ [] &= [[]]\\
sublists\ (a :: x) &= (map2\ (frontlists\ x)\ a)\ @\ (sublists\ x)\\
frontlists\ [] &= [[]]\\
frontlists\ (a :: x) &= (map2\ (frontlists\ x)\ a)\ @\ [[]]\\
(map2\ []\ x) &= []\\
(map2\ (l :: lx)\ x) &= (x :: l) :: (map2\ lx\ x)\\
map1\ [] &= []\\
map1\ (a :: x) &= (sum\ a) :: (map1\ x)\\
fold1\ [a] &= a\\
fold1\ (a :: (b :: c)) &= (bimax\ a\ (fold1\ (b :: x)))\\
sum\ [] &= 0\\
sum\ (a :: x) &= (+\ a\ (sum\ x))
\end{aligned}
$$

Figure 5: Equations source

$(bimax\ a\ (fold1\ (b :: x)))$ and the application of @ in $(map2\ (frontlists\ x)\ a)\ @\ (sublists\ x)$. The normalization algorithm of [12] rejects also the deforestation of t which is *not potentially normalizable* for this algorithm [5].

We are currently implementing an automatic mode of Astre, we call *Automatic Astre* which discovers automatically the terms that are candidate for deforestation such as t and introduces automatically an *eureca rule* such as:

$$fold1\ (map1\ (sublists\ l)) \rightarrow maxsub'\,l \tag{1}$$

when it discovers a candidate for deforestation term. After that, the *partial* completion procedure processes the transformation. It automatically overlaps this reversed rule with the rules for *sublists* generating the following normalized equations:

$$
\begin{aligned}
maxsub'\ [] &= 0\\
maxsub'\ (a :: x) &= fold1\ (map1\ (map2\ (frontlists\ x)\ a)\ @\ (sublists\ x))
\end{aligned}
\tag{2}
$$

The transformation with pencil and paper at the higher-order level gives a similar result:

$$
\begin{aligned}
maxsub\ [] &= 0\\
maxsub\ (a :: x) &= fold\ bimax\ (map\ sum\\
&\qquad (map\ (consl\ a)\ (frontlists\ x)\ @\ (sublists\ x)))
\end{aligned}
\tag{3}
$$

---

[5]The term t is not *potentially normalizable* because the call to *frontlists* where

$$frontlists\ (a :: x)\ =\ (map2\ (frontlists\ x)\ a)\ @\ [[]]$$

applies an inner *catamorphism:map2* to the recursive call of *frontlist*.

Now it becomes interesting for the system to look for some input of theorems so that it rewrites (*folds*) the right-hand side of the above equation by the *eureca rule* in order to it introduces a recursive occurrence of *maxsub'*. In the current version of Astre, the user must provide the laws. Let us consider how they can be automatically extracted for a library of general laws on higher-order primitives and transposed at the first-order level using a *law extractor* which is based on the set of instantiation triples given by a partial evaluator. Such a combination of partial evaluation tools and intrinsically first-order rewriting tools can result in a powerful transformation system.

4 Interaction with the theorems

Let us come back to the higher-order level of the transformation. Suppose a higher-order law $L \to R$ *rewrites* the λ-term M to N. There exists a substitution σ and an occurrence in M of a subterm S $\beta\eta$-equal to $\sigma(L)$. The rewritted term N is the result of the replacement of the subterm S in M by a term $\beta\eta$-equal to $\sigma(R)$. We will denote $M[S \wr T]$ the replacement in M of a subterm S by T. Let l be $\sigma(L) \downarrow_{first}$ and r be $\sigma(R) \downarrow_{first}$. We want to show that $M \downarrow_{first}$ reduces to $N \downarrow_{first}$ by the rule $l \to r$.

As in [13], we introduce the η-expanded form of λ-term. For example the η-expanded form of $(map\ f \circ map\ g)$ of type order two is $((map\ f \circ map\ g)\ x)$ of type order one (or elementary). Then the Church-Rosser theorem for the $\beta\eta$-calculus can be expressed in the following form: For every two λ-term M and N, we have $M =_{\beta\eta} N$ if and only if their η-expanded form are β-equals. It is not a restriction to require that L and R are in β-normal form and that they are η-expanded so that their common type is elementary. For example the *maplaw*:

$$map\ f \ \circ \ map\ g \ = \ map\ (f \ \circ \ g)$$

where the type of both hand-sides is functional, can be η-expanded into:

$$((map\ f \ \circ \ map\ g)\ x\ ) = \ map\ (f \ \circ \ g)\ x$$

or into:

$$(map\ f\ (map\ g\ x)) = \ map\ (f \ \circ \ g)\ x$$

when the definition of $\circ$ is unfolded.

Theorem 2 *Assume that the non RHOP functions in M are first-order and that the RHOP functions are constant or variable-only, let $L \to R$ be a law that rewrites M with the substitution σ into the λ-term N, let δ be the restriction of σ to the subset of the functional variables in the domain of σ, let $m = M \downarrow_{first}$, and $n = N \downarrow_{first}$, then m reduces to n by application of the rule $l \to r$ where r is $\delta(R) \downarrow_{first}$ and l is $\delta(L) \downarrow first$.*

> **Proof:** We require that L is in η-expanded form so that its type is elementary.
> Or a subterm S $\beta\eta$-equal to $\sigma(L)$ occurs in M, therefore, if L and M are in β-normal
> form, the subterm S is equal to $\sigma(L)$ whose type is elementary. The subterm $\sigma(R)$
> in β-normal form occurs in the β-reduction of N. Consider now $m = M \downarrow_{first}$ and
> the corresponding set ST of triples $(F_i, \delta_i, T_i), i = 1, n$. By definition of $\to_{first}$,
> $m = M[\delta_i(T_i) \backslash F_i\ \forall i, i = 1, n]$, where the $\forall i = 1, n$ means that a *folding* is done
> for all triples in ST. In the same way, $l = \delta(l) \downarrow_{first} = \delta(L)[\delta_i(T_i) \backslash F_i\ \forall i, i = 1, n]$.

Let ρ be the remaining part of the substitution δ such that $\sigma = \rho \circ \delta$ where $\circ$ is the composition of substitutions. The term $\rho(l)$ is a subterm of m. On another hand, $N = M[S \wr \sigma(R)]$, $r = \delta(R)[\delta_i(T_i)\backslash F_i \ \forall i, i = 1, n]$, and $n = N \downarrow_{first} = M[S \wr \sigma(R)][\delta_i(T_i)\backslash F_i \ \forall i, i = 1, n]$. Therefore m reduces to n by the rule $l \to r$. $\square$

For example, the $map@law$ rewrites the left hand-side of the equation 3 with the substitution σ:

$$\{f \leftarrow sum, x \leftarrow (map\ (consl\ a)\ (frontlists\ x)), y \leftarrow (sublists\ x)\}$$

The result is:

$$
\begin{aligned}
maxsub'(a :: x) \quad = \quad & fold\ bimax\ ((map\ sum\ (map\ (consl\ a)\ (frontlists\ x))) \\
& @ \ (map\ sum\ (sublists\ x)))
\end{aligned}
\tag{4}
$$

At the first-order level the rule:

$$
\begin{aligned}
& map1\ ((map2\ (frontlists\ x)\ a)\ @\ (sublists\ x)) \\
& \to (map1\ (map2\ (frontlists\ x)\ a))\ @\ (map1\ (sublists\ x))
\end{aligned}
$$

reduces the first-order converted equation 2. The result is:

$$
\begin{aligned}
maxsub'(a :: x) \quad = \quad & fold1\ ((map1\ (map2\ (frontlists\ x)\ a)) \\
& @\ (map1\ (sublists\ x)))
\end{aligned}
\tag{5}
$$

Theorem 2 shows the link between the transformation done at the higher-order level and the transformation at the first-order level.

As a consequence of Theorem 2, we can justify the following way to find a reduction of a first-order term t by a law at the first-order level:

Find a law $L \to R$, a set of specialization triples ST, a first-order substitution ρ, and a position p in t such that $t|p = \rho(L \downarrow_{first})$, then the first-order rule $L \downarrow_{first} = R \downarrow_{first}$ rewrites t at the position p with the substitution ρ.

The above can be processed by a procedure, we call a *Law-Extractor*, which returns to Astre a first-order theorem from a library of higher-order laws about the higher-order primitives.

For the *maxsub* example, the set ST contains the triple:

$$\{(map1, \{f \leftarrow sum\}, \lambda\ x.(map\ f\ x))\}$$

Considering equation 2 and the $map@law$ of the library in Figure 3, the *Law-Extractor* returns the first-order rule: $(map1\ x\ @\ y) = (map1\ x)\ @\ (map1\ y)$. which simplifies equation 2 into equation 5.

Now, considering the *maplaw* in the library and the set ST containing the triples:

$$
\begin{aligned}
& (map1, \{f \leftarrow sum\} \qquad , \lambda\ x.(map\ f\ x)) \\
& (map2, \{f \leftarrow (consl\ x)\}, \lambda\ l\ x.(map\ f\ l))
\end{aligned}
$$

the *Law-Extractor* returns the first-order rule:

$$map1\ (map2\ l\ x) \to (map\ (sum\ \circ\ (consl\ x))\ l) \downarrow_{first}$$

The first-order conversion of the left hand-side processes a new specialization which adds a triple:

$$(map3, \{f \leftarrow \lambda \ l.(+ \ x \ (sum \ l))\}, \lambda \ l \ x.(map \ f \ l))$$

where the term substituted for f is the η-expanded form of the β-reduction of the functional argument of map: $(sum \ \circ \ (consl \ x))$. This rule rewrites equation 5 into:

$$maxsub'\,(a :: x) = fold1 \ (map3 \ (frontlists\,x) \ a) \ @ \ (map1 \ (sublists\,x)) \qquad (6)$$

The interested reader can follow the remaining of the transformation in the appendix.

Notice that there is always a problem of confluence when a base of theorems is used for program transformation. The choice of a law to rewrite a term is ambiguous when two overlapping laws can be chosen. It is not a problem if the set of laws is confluent (if a notion of confluence is extended to higher-order logic). The set of laws presented in Figure 3 is not free of critical pairs. There is a unification modulo the associativity of $\circ$ between the left hand-sides *foldmaplaw* and the *maplaw*, the unified term being: $fold \ f \ \circ \ map \ f \ \circ \ map \ g$. Fortunately, by replacing the *foldmaplaw* by the *generalized foldmaplaw*

$$fold\,maplaw : fold \ f \ \circ \ map \ (g \ \circ \ h) = g \ \circ \ ((fold \ f) \ \circ \ (map \ h))$$
$$\text{if } f \ (g \ x) \ (g \ y) = g \ (f \ x \ y)$$

the library of higher-order laws becomes "confluent" so that we can use it for the transformation of the *maxsub* example.

5   Conclusion

Higher-order transformations based on well known properties of higher-order primitives are not easily automated. The paper presents a way to mimic such transformations at the first-order level by rewriting techniques. In this way, we obtain a tool that automatically implement the deforestation of higher-order Squiggol programs. Currently, the state of our technology is the following:

- a translator ML into Schism and a translator Schism to ML which restores the patterns (currently implemented).

- the partial evaluator Schism [6] which uses its own (typed) dialect of Scheme as its object language, and

- the interactive transformation system Astre [2] based on rewriting and completion procedures written in CAML. The mode "automatic Astre" for simple deforestations is currently implemented.

We have simulated the interaction of these tools to process the sketch of the transformation of the *maxsub* example. The main piece of software to add is the *Law-Extractor* defined in Section 4. We expect that all these tools will work harmoniously in the near future to fully automate deforestation of higher-order Squiggol programs. These transformations are applicable to functional programs that describe first-order functions using *constant or variable-only* higher-order primitives. For this class of programs, the paper shows that the transformation at the higher-order level can be mimicked by rewriting in first-order logic.

References

[1] F. Bellegarde. Program Transformation and Rewriting. In *Proceedings of the fourth conference on Rewriting Techniques and Applications*, Springer Verlag, Lecture Notes in Computer Science 488, pages 226-239, Como, Italy, 1991.

[2] F. Bellegarde. Astre, a Transformation System using Completion. Technical Report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1991.

[3] J. Bentley. *Programming Pearls*, Addison Wesley, 1986.

[4] R. M. Burstall and J. Darlington. A Transformation System For Developing Recursive Programs. *Journal of the Association for Computing Machinery*, 24, pages 44-67, 1977.

[5] W. N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, University of London, Imperial College of Science, Technology and Medecine, London, UK, 1990.

[6] C. Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. Research report, Pacific Software Research Center, Oregon Graduate Institute of Science and Technology, Beaverton, Oregon, USA, 1992.

[7] N. Dershowitz. Completion and its Applications. *Resolution of Equations in Algebraic Structures*, Academic Press, New York, 1988.

[8] A. Ershov. Mixed computation: potential applications and problem for study. *Theorical Computer Science*, Vol. 18, pages 41-67, 1982.

[9] A. Gill, J. Launchbury and S.L. Peyton Jones. A short cut to Deforestation. In *Sixth Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, Denmark, pp 223-232, June 1993.

[10] E. Meijer, M. Fokkinga and R. Patterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Conference on Functional Programming and Computer Architecture*, Lecture Notes in Computer Science 523, pages 124-144, 1991.

[11] U. S. Reddy. Transformational derivation of programs using the Focus system. In *Symposium Practical Software Development Environments*, pages 163-172, ACM, December 1988.

[12] T. Scheard and L. Fegaras. A fold for All Seasons. *Sixth Conference on Functional Programming Lnaguages and Computer Architecture*, Copenhagen, Denmark, pp 233-242, June 1993.

[13] W. Snyder and J. Gallier. Higher order unification revisited: complete sets of transformations. *Journal of Symbolic Computation*, Vol.8, pages 101-140, 1989. Special issue on Unification. Part two.

[14] S. Thompson. Functional Programming: Executable Specifications and Program Transformation. In *Fifth International Workshop on Software Specification and Design.*, IEEE Press, 1989.

[15] S. Thompson. *Type Theory and Functional Programming*, Addison Wesley, 1991.

[16] P. Wadler, Deforestation: Transforming programs to eliminate trees. In *2nd European Symposium on Programming ESOP'88*, Nancy France, 1988. Lecture Notes in Computer Science 300, Springer Verlag.

[17] P. Wadler. Theorem for free! In *Proc. 1989 ACM Conference on Lisp and Functional Programming*, pages 347-359, 1989.

6   Appendix

Consider following the sketch of the automatic transformation of *maxsub* starting with the equation 6 we found at the end of Section 3:

$$maxsub'\,(a::x) = fold1\ (map3\ (frontlists\,x)\ a)\ @\ (map1\ (sublists\,x))$$

At this point, the set ST contains the triples:

$$
\begin{aligned}
\{(&map1, \{f \leftarrow sum\} & ,\lambda\ x.(map\ f\ x)) \\
(&map2\ , \{f \leftarrow (consl\ x)\} & ,\lambda\ l\ x.(map\ f\ l)) \\
(&fold1\ \ , \{f \leftarrow bimax\ \} & ,\lambda\ x.(fold\ f\ x)) \\
(&map3\ , \{f \leftarrow \lambda\ l\ x.(+\ x\ (sum\ l))\}, \lambda\ l\ x.(map\ f\ l))\}
\end{aligned}
$$

Considering the *fold@law*, the *Law-Extractor* returns the first-order rule:

$$fold1\ (x\ @\ y)\ =\ bimax\ (fold1\ x)\ (fold1\ y)$$

because *bimax* is associative. This rule rewrites equation 6 into:

$$
\begin{aligned}
maxsub'\,(a::x)\ \ =\ \ & bimax\ (fold1\ (map3\ (frontlists\,x)\ a)) \\
& (fold1\ (map1\ (sublists\,x)))
\end{aligned}
$$

At this point, Astre rewrites by the rule 1 which folds *maxsub'*:

$$maxsub'\,(a::x) = bimax\ (fold1\ (map3\ (frontlists\,x)\ a))\ (maxsub'\ x) \qquad (7)$$

Considering the *generalized foldmaplaw* introduced in Section 4, the *Law-extractor* returns the first-order rule:

$$fold1\ (map3\ l\ x)\ =\ +\ x\ (fold1\ (map1\ l))$$

because $bimax(+\ u\ x)\ (+\ u\ y) = +\ u\ (bimax\ x\ y)$. This rule simplifies the equation 7 into:

$$maxsub'\,(a::x) = bimax\ (+\ a\ (fold1\ (map1\ (frontlists\,x))))\ (maxsub'\ x) \qquad (8)$$

Now, the composition of functions $(fold1\ (map1\ (frontlists\,x)))$ is a candidate for a deforestation. As for *maxsub*, an *eureca rule* for a new functional symbol *maxfront* which captures this composition is introduced automatically.

$$fold1\ (map1\ (frontlists\,x))\ \rightarrow\ maxfront\ x$$

This reversed rule simplifies equation 8 into the final result for *maxsub*:

$$maxsub'\,(a :: x) = bimax\,(+\,a\,(maxfront\,x))\,(maxsub'\,x) \qquad (9)$$

The partial completion procedure superposes this reversed rule with the rules for *frontlists*. From the above transformation of *maxsub'*, it knows the full set of first-order laws to generate the following equations for *maxfront*:

$$\begin{aligned}
maxfront\,[] &= 0 \\
maxfront\,(a :: x) &= bimax\,(+\,a\,(maxfront\,x))\,0
\end{aligned}$$