Automating Synthesis by Completion

Françoise Bellegarde*
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology
PO Box 91000, Portland, Oregon 97291-1000, USA.
{bellegar}@cse.ogi.edu

April 26, 1994

Abstract

When using a completion procedure, simple static analyses allow automatizing the manipulation of first-order functional programs. This paper shows how to apply these techniques in order to fully automatize program transformation by the two strategies: *deforestation* (eliminating useless intermediate data structures) and *tupling* (eliminating parallel traversals of identical data structures). These techniques enable us to transform a larger class of programs than the current deforestation algorithms. Programs are translated into a constructor-based rewrite system. Then, a completion procedure can be used for synthesis of the transformed rewrite system. Static analyses of the rewrite system are needed to automatize a given transformation strategy. They permit the following: (1) to automatically discover the specifications of the functions that need to be synthesized by completion in order to accommodate the strategy, (2) to control the production of critical pairs during the completion process according to the strategy, (3) to determine how and when to orient the critical pairs into rewrite rules without losing termination of the rewrite system, (4) to guarantee termination of the process, not only by ensuring termination of the completion, but also by ensuring that the transformation does not require infinitely many completions, and finally (5) to guarantee the effectiveness of the transformation.

*The author is supported in part by a contract with Air Force Material Command (F19628-93-C-0069).

1 Introduction

1.1 Deforestation and Functional Programming

It has often been said that functional programs are constructed using only functions as pieces. Data structures such as lists and trees are the glue to hold them together. Although this compositional style of programming is attractive [14], it comes at the expense of efficiency. Compositions produce many intermediate data structures. One way to circumvent this problem is to perform *fusion or deforestation* on programs as advocated by Burge in [5] and by Wadler [18, 19]. Several approaches for eliminating useless intermediate data structures have been proposed. The algorithm proposed by Wadler [19] performs automatic deforestation on a restricted class of terms called *treeless* terms. Later, Chin's remarkable work on fusion [7] applies to a wider class of *e-treeless* terms and to higher-order programs in general. Sheard and Fegaras [16] utilizes laws about combinator compositions (*Promotion theorems*) to *deforest* programs written with combinators. This technique is applicable to a class of *potentially normalizable* terms built with a special set of higher-order combinators. An automatic way to implement deforestation inside the Haskell's compiler has been shown in [12]. Also a *widened deforestation* algorithm is proposed in a recent paper [17].

Deforestation algorithms do not recognize that an expression contains two or more functions that consume the same data structure. Such functions create a "parallel" traversal of a data structure. These functions can be put together in a tuple as a single function that traverses the data structure only once. This is another way of transforming programs according to the tupling lemma [9].

General purpose program transformation systems are based on a unfold-fold method proposed by Burstall and Darlington [6]. Deforestation and tupling are particular instances of this strategy. In the Focus system [13], folding and unfolding are seen as rewritings. It has been pointed out by Dershowitz [11] that an unfold-fold strategy can be controlled by a completion procedure. Following this idea, the transformation system Astre [1, 2] is based on completion procedures. The first versions of Astre were highly interactive, requiring intervention of an expert user. A transformation system must be automated to be effectively used as a tool in a transformational approach to software design.

1.2 This work

Simple static analyses of first-order recursive equations allows us to automate both deforestation and tupling. We only consider *terminating rewrite systems*.

Dershovitz has shown in [11, 10] that a completion procedure is good at synthesizing a constructor-based equational definition of a function h from a *synthesis rule*[1] which specifies how h is defined in terms of other functions. That is what we need for deforestation, where h is defined by a composition of other functions that generates intermediate data structures, and for tupling, where h is defined by a tuple (pair) of functions that traverse the same data structure.

[1] This rule is called an "eureka" in the fold-unfold methodology.

If the synthesis has to be fully automatized, these *synthesis rules* must be discovered automatically by a static analysis of the system of equations at hand.

Moreover, a completion procedure is far too general. Its purpose is to generate *critical pairs*, equational consequences of a set of rules coming from overlapping left-hand sides of existing rules. What N. Dershowitz noticed in [11] is that it is possible that the completion generates the critical pairs which constitute a constructor-based definition of a function h. However this generation: (1) must be controlled by a suitable ordering, (2) can be explosive, and (3) can even loop by generating infinitely many critical pairs. A static analysis of the system of equations allows us to to restrict the overlaps between left-hand sides of rules so that the completion procedure computes exactly the critical pairs that are needed for the synthesis.

For automating the process, not only the production of critical pairs needs to be limited, but the orientation of the critical pairs into rewrite rules has to be automated. Once again, an analysis of a critical pair allows us to know if the synthesis is achieved so that the critical pairs can be oriented in a rule of the constructor-based synthesized definition of h. Moreover, the process must preserve termination of the rewrite system.

The discovery of a *synthesis rule* demands at least one synthesis, accomplished by a completion procedure, which is guaranteed to never fail and to always terminate. We also guarantee that the process does not perform infinitely many syntheses.

Moreover, the process must choose the strategy ensuring the *effectiveness* of the transformation, i.e. that the resulting program is, in some way, more efficient than the given program.

1.3 Outline

The paper is organized as follows: Section 2 introduces basic notations in term rewriting systems that are used for our analyses and it presents naive examples illustrating how a completion procedure performs synthesis for deforestation and tupling. Section 3 presents the analysis necessary for synthesis rules discovery in the case of deforestation and tupling strategies. Section 4 presents the analysis that is necessary to control the completion process. Section 5 discusses termination, effectiveness, limitations and possible extension issues. Section 6 relates our techniques with other work followed by concluding remarks in Section 7.

2 Application of Completion to Unfold-Fold Strategies

2.1 Basic Notations

Let F be a set of function symbols and V be a set of variables, $T(F,V)$ is the set of terms with symbols in F and variables in V. $V(t)$ is the set of all the variables occurring in t. A position or occurrence within a term t is represented as a finite sequence ω of positive integers describing the path from the root of t to the root of the subterm at that position, denoted by $t|_\omega$. The position of the root of a term t is ϵ. $G(t)$ is the set of the positions of function symbols in t. A term s is less than t for the subsumption ordering if and only if t is an instance of s. We say that t *encompasses* s if a **subterm** of t is an instance of s. A term t is said to be linear if no variable occurs more than once in t.

A rewrite rule is an ordered pair of terms, written as $l \to r$, where $V(r) \subseteq V(l)$. A rule $l \to r$ is *left-linear* if l is linear, it is *right-linear* if r is linear. A rewrite system is a set of rewrite rules. The rewriting relation is denoted as $\to$. The rewrite system R *is terminating* if and only if there is no infinite sequence of terms $t_1, t_2, \ldots$, such that $t_1 \to_R t_2 \to_R \ldots$ The *R-normal form of a term t* is a term $t\downarrow_R$ such that $t \to_R^* t\downarrow_R$ and there is no u such that $t\downarrow_R \to_R u$.

A rewrite system is *overlapping* if there exists an *overlap* between left-hand sides of two rules $g \to d$ and $l \to r$, i.e. if there exists a position ω in $G(l)$ such that $l|_\omega$ and g are unifiable with the most general unifier σ. A *critical pair* is the identity $\sigma(l[\omega \leftarrow \sigma(d)]) = \sigma(r)$ where $t[\omega \leftarrow u]$ denotes the replacement in t of the subterm at position ω by u.

An *orthogonal* system is a left-linear and non-overlapping rewrite system. A 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*. C and D denote respectively the set of constructors and the set of defined symbols. A *constructor term* is a member of $T(C, V)$. R_f is the set of all the rules $l \to r$ of a constructor-based rewrite system R where the root of l is f. A rewrite system is *confluent* if and only if the relation $\to^*$ verifies the diamond property. Confluence ensures the unicity of the normal form while termination ensures its existence. A non-overlapping and terminating rewrite system is confluent. A completion procedure aims at discovering critical pairs in a terminating rewrite system R to check whether the two sides of the pair rewrite to the same term. Otherwise, it adds the critical pairs to R, orienting them in such a way as to preserve the termination property. If the procedure does not fail and terminates, it returns a confluent and terminating system equivalent to R. If for each rule $l \to r \in R$, l, and r are irreducible by R, then R is *interreduced*.

2.2 Completion Procedure and Unfold-Fold Method

In this paragraph, we illustrate by basic examples how a completion procedure interactively controls a fold-unfold method. The unfold-fold method [6] consists of 6 rules, namely *Definition, Instantiation, Unfolding, Folding, Abstraction*, and *Law*, that allow new identities to be introduced that are equational consequences of existing identities. Dershowitz [11] has shown how the combination of *Instantiation* and *Folding* is enabled by critical pair generation. *Unfolding* and *Law* are simplifications by rewriting. *Definition* is the introduction of a *synthesis* rule by the user, *Abstraction* is used for a tupling tactic.

Deforestation Example

Consider a naive example of a single deforestation of one term: $length(x@y)$ where

$$R_{length} : \begin{cases} length([]) & \to & 0 \\ length(x :: xs) & \to & 1 + length(xs) \end{cases} \qquad R_{@} : \begin{cases} []@y & \to & y \\ (x :: xs)@y & \to & x :: (xs@y) \end{cases}$$

The list x is traversed once to append it to y and once more to count the length of the result. A synthesis rule $length(x@y) \to h(x, y)$ is introduced. It overlaps with rules of $R_@$ yielding

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

two critical pairs:

$$length(y) = h([], y)$$
$$length(x :: (xs@y)) = h(x :: xs, y)$$

The last pair simplifies by the second rule in R_{length} into $1 + length(xs@y) = h(x :: xs, y)$. Now h is defined by:

$$\begin{cases} h([], y) & = & length(y) \\ h(x :: xs, y) & = & 1 + length(xs@y) = 1 + h(xs, y) \end{cases}$$

which makes only one traversal of x to compute the result. For this very simple example, no law is necessary. But suppose the synthesis rule is $length(flat(x)) \rightarrow h(x)$ where one rule of R_{flat} is: $flat(x :: xs) \rightarrow x \ @ \ flat(xs)$, the rule $(Law)\ length(x@y) \rightarrow length(x) + length(y)$ is required to simplify the left-hand side of the pair $length(x \ @ \ flat(xs)) = h(x :: xs)$ according to the following derivation:

$$length(x \ @ \ flat(xs)) \rightarrow length(x) \ + \ length(flat(xs)) \rightarrow length(x) \ + \ h(xs)$$
yielding a R_h rule: $h(x :: xs) \rightarrow length(x) \ + \ h(xs)$

This needs for laws in the method is an obstacle to full automatization. We will see how, we avoid the input of laws by using completion to synthesize exactly what is needed to achieve the synthesis of h.

Tupling Tactic Example

Consider another naive example, $Ave(x) \rightarrow sum(x)/length(x)$ where

$$R_{sum} : \begin{cases} sum([]) \rightarrow 0 \\ sum(x :: xs) \rightarrow x + sum(xs) \end{cases} \quad R_{length} : \begin{cases} length([]) & \rightarrow & 0 \\ length(x :: xs) & \rightarrow & 1 + length(xs) \end{cases}$$

The list x is traversed twice "in parallel" to compute the average. In this case, we introduce the rules:

$$Synthesis : \begin{cases} sum(x) \rightarrow fst(h(x)) \quad (1) \\ length(x) \rightarrow snd(h(x)) \quad (2) \\ pair(fst(h(x)), snd(h(x))) \rightarrow h(x) \end{cases} \qquad Comp : \begin{cases} fst(pair(x, y)) & \rightarrow & x \\ snd(pair(x, y)) & \rightarrow & y \end{cases}$$

By rewriting left-hand sides by rules (1) and (2), we get: $Ave(x) \rightarrow fst(h(x))/snd(h(x))$ which can be computed with a single traversal of x by sharing the common computation of $h(x)$. The two first synthesis rules overlap respectively with rules of R_{sum} and R_{length} yielding the pairs:

$$\begin{array}{lcllcl} 0 & = & fst(h([])) & 0 & = & snd(h([])) \\ x + sum(xs) & = & fst(h(x :: xs)) & 1 + length(xs) & = & snd(h(x :: xs)) \end{array}$$

which can be turned into rules from right to left:

$$\begin{cases} fst(h([])) & \rightarrow & 0 \ (1) \\ fst(h(x :: xs)) & \rightarrow & x + sum(xs) \ (2) \end{cases} \qquad \begin{cases} snd(h([])) & \rightarrow & 0 \\ snd(h(x :: xs)) & \rightarrow & 1 + length(xs) \end{cases}$$

Afterwards, these rules overlap with the third synthesis rule yielding R_h rules:

$$\begin{cases} h([]) & \rightarrow & pair(0,0) \\ h(x :: xs) & \rightarrow & pair(x + fst(h(xs)), 1 + snd(h(xs))) \end{cases}$$

These last two rules reduce the left-hand sides of Rules 1 and 2. The *Comp* rules further reduce these left-hand sides so that they become identical to the right-hand sides. Then the rules can be deleted. It is worthwhile to notice that this tactic can be applied to transform a function that computes the n^{th} fibonacci number k in time proportional to k itself into a function that computes the same number in only n steps. This example has been used in [15] showing how a completion procedure produces useless explosion of critical pairs when controlling an unfold-fold transformation. Our way of automating the tupling and deforestation tactics always generates exactly the needed critical pairs. It also discovers automatically the *synthesis rules*.

3 Automatic Synthesis Rules Discovery

The first-order functional program is presented by a constructor-based orthogonal terminating and interreduced rewriting system. The purpose of the analysis is to discover synthesis rules: 1) to eliminate useless data structures by deforestation, we call **fusion rules**, and 2) to eliminate multiple traversal removal by tupling, we call **tupling rules**.

3.1 Fusion Rules

In this paragraph, we show how we analyze the term t in order to build a **fusion rule** *(synthesis rule for deforestation)*. A fusion rule must reduce t into a term t' which contains less intermediate data structures than t. Such a rule has the form $s \rightarrow h(x_1, x_2, \cdots, x_n)$ where h is a fresh symbol. and $\{x_1, x_2, \cdots, x_n\} = V(s)$. Obviously, s must subsume a subterm of t so that the fusion rule rewrites t, but s must be carefully chosen. Consider for example a term $t = f(x)$ @ $g(t)$ where @ is defined by the constructor-based rules $R_@$:

$$[] \ @ \ x \ = \ x \qquad (x \ :: \ xs) \ @ \ y \ = \ x \ :: \ (xs \ @ \ y)$$

The subterm $f(x)$ **produces** an intermediary list which is **consumed** by @ but though the subterm $g(t)$ produces a list argument of @, this list is not traversed by @. According to a vocabulary invented by Chin in [7], the symbol @ is a **consumer**, and $f(x)$ is a **producer** in the term t. Contrary to Chin, we do not consider $g(t)$ as a producer. We say that @ can be a consumer only at **inductive position** 1. In t, there is only one useless data structure produced by $f(x)$ and consumed by @. A rule $f(x)$ @ $y \rightarrow h(x, y)$ reduces t into $t' = h(x, g(t))$ which does not contain the useless data structure produced by $f(x)$. The **inductive positions** of a symbol f (see Figure 1) defined by the constructor-based rules R_f indicate the positions where f can be a consumer.

Definition 1 (inductive positions) *A symbol f consumes a data structure at* **inductive positions** *i if there exists a rule $l \rightarrow r$ in R_f where $l|_i$ is a constructor term and not a variable.*

By following the inductive positions of the defined symbols in a term t, we can extract recursively a **fusion term s** (see Figure 3) from a term t which contains the symbols that produces useless data structures in t. These symbols are located at the **spine positions** of t (see Figure 2).

Definition 2 (spine positions) *A position ω in a term t is a* **spine position** *if*

- *either ω is ϵ, or*
- *$\omega = u.i$ where u is a spine position, the root f of $t|_u \in D$ and i is an inductive position of f.*

The inductive positions of a fresh symbol h can be known by analyzing its fusion rule (see Figure 4). Indeed, *if a variable x occurs in the fusion term at a spine positions, the position of x in the list $(x_1, x_2, \cdots, x_n)$ in the right-hand side indicates an inductive position of h.* The reason is that overlaps with the constructor-based rules of the producers can only substitute constructor terms at these positions.

To perform deforestation of a constructor-based system R, it is enough to search for fusion terms in the right-hand sides of definition rules, then to build a fusion rule r (see Figure 3), and finally to synthesize the constructor based system R_h of the fresh symbol h.

Definition 3 (fusion rule) *Let $f(tc_1, tc_2, \cdots, tc_n) \rightarrow r$ be a rule of a constructor-based terminating and interreduced rewrite system R, a* **fusion term** *s can be extracted from r, if there exists a position u in r such that:*

1. *$r|_u$ is an instance of s (encompassment),*
2. *the top symbol of s is a defined symbol, and every other functional symbol in s is a defined symbol or a constructor, and every functional symbol that occurs in s is located at a spine position of s (deforestation),*
3. *s is linear (linearity),*
4. *f does not occurs in s (skip over recursive calls)*
5. *there exists a spine position of s of length greater that 1 (nontriviality)*
6. *s is not contained in any deforestation term extracted from r (maximality)*

When such a fusion term s exists, the rule $s \rightarrow h(x_1, x_2, \cdots, x_n)$, where h is a fresh symbol and $x_1, x_2, \cdots, x_n$ are the variables in s, is a **fusion rule**.

The **linearity** ensures preservation of the termination of the union of the rewrite system R with the fusion rule as proved in [4]. As we will see later, the **skip over recursive calls** is required for ensuring the termination of the deforestation process. For example, no fusion term can be found in the recursive rule

$$S : subs(x :: xs) = subs(xs) \ @ \ map\_cons(x, subs(xs))$$

$$k(0) \rightarrow 0 \qquad k(s(0)) \rightarrow s(0) \qquad k(s(s(x)) \rightarrow k(x) + k(s(x))$$

$k : 1, 1.1$, s and 0 are constructors,

$$g_1(0) = [] \qquad g_1(s(x)) = g_1(x) :: s(x)$$

$$g_1 : 1$$

$$f([], x, z) = [] \qquad f(x, [], z) = [] \qquad f(x :: xs, y :: ys, z) = (x + z) :: f(xs, ys, z)$$

$f : 1, 2, []$ and $::$ are constructors,

$$g_2(z, []) = [] \qquad g_2(z, x :: xs) = (z + x) :: g_2(z, xs)$$

$$g_2 : 2$$

Figure 1: Function definitions and inductive positions analysis

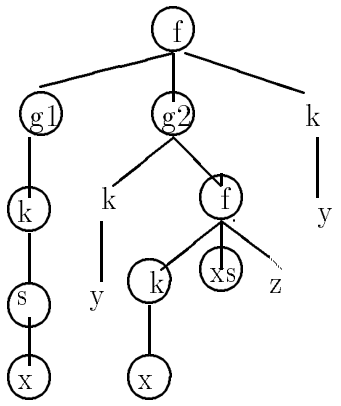


Figure 2: Spine positions:$\{\epsilon, 1, 1.1, 1.1.1, 2, 2.2, 2.2.1, 2.2.1.1, 2.2.2\}$

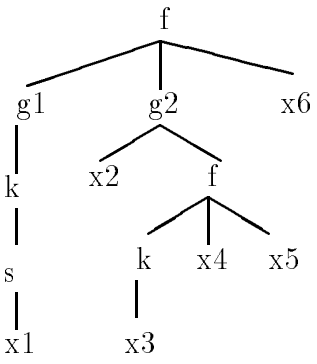


Figure 3: Fusion rule:$f(g_1(k(s(x_1))), g_2(x_2, f(k(x_3), x_4, x_5)), x_6) \rightarrow h(x_1, x_2, x_3, x_4, x_5, x_6)$.

where *map_cons* has inductive position 2. *Our method attempts a fusion for all the intermediate data structures produced by defined symbols but those produced by the recursive calls.* Moreover, it performs a degenerative kind of fusion when the producer is a constructor . For example, $zip(x :: xs, y)$ where zip has inductive positions 1 and 2, is a fusion term.

3.2 Tupling Rules

Two functional symbols that traverse the same data structure are used to construct **tupling rules**. For example, $sum(x)$ and $length(x)$ traverse the same list in $sum(x)/length(x)$, $fib(x)$ and $fib(s(x))$ traverse the same natural number in $fib(x) + fib(s(x))$. Also $downto(xs)$ and $tails(xs)$ traverse the same list in $(x :: downto(xs)) :: tails(xs)$. Two **tupable subterms** $f(t_1, t_2, \cdots, t_n)$ and $g(s_1, s_2, \cdots, s_m)$ of a term r share the same set of variables located at the inductive positions of f and g. Figure 5, and Figure 7 show less naive situations.

When it finds a tupling pair as subterms of a right-hand side r, the process introduces automatically two different rewrite rules (**tupling rules**):

$$f(t_1, t_2, \cdots, t_n) \rightarrow fst(h(x_1, x_2, \cdots, x_n)) \qquad g(s_1, s_2, \cdots, s_m) \rightarrow snd(h(x_1, x_2, \cdots, x_n))$$

where $\{x_1, x_2, \cdots, x_n\} = V(s_1) = V(s_2)$ and h is a fresh symbol. The tupling rules rewrite r, creating two common subterms $h(t_1, t_2, \cdots, t_n)$ which can be shared at the level of the translation. For example, after this replacement, $fib(x) + fib(s(x))$ becomes $fst(h(x)) + snd(h(x))$.

The completion synthesizes a constructor-based definition of *fst* composed with h: R_{fst} and a constructor-based definition of *snd* composed with h: R_{snd}. For the above example, the result of this synthesis is:

$$\begin{cases} fst(h(0)) \rightarrow 0 \\ fst(h(s(x)) \rightarrow snd(h(x)) \end{cases} \qquad \begin{cases} snd(h(0)) \rightarrow s(0) \\ snd(h(s(x))) \rightarrow fst(h(x)) + snd(h(s(x))) \end{cases}$$

This synthesis is possible only if there is, indeed, overlaps between the elements of the tupling pair and the constructor-based rewrite systems R_f and R_g of their top symbols f and g. This can happen only if the symbols at inductive positions of f and g are constructors or variables and if there is always a variable at a maximal inductive position (see Figure 6).

The process introduces systematically the **pair rule**:

$$pair(fst(h(x)), snd(h(x))) \rightarrow h(x)$$

and the rules $fst(pair(x, y)) \rightarrow x$, $snd(pair(x, y)) \rightarrow y$ which allows the completion to achieve the synthesis of h. For the above example, the result is:

$$\begin{cases} h(0) = pair(0, s(0)) \\ h(s(x)) \rightarrow pair(snd(h(x)), fst(h(x)) + snd(h(s(x)))). \end{cases}$$

Then the completion cancels the now useless rules R_{fst} and R_{snd}.

Definition 4 (Tupling pair) *Let $l \rightarrow r$ be a rule of a constructor-based rewrite system R, the right-hand side r contains a* **tupling pair**

$$(r|_{p_1} = f(t_1, t_2, \cdots, t_n), r|_{p_2} = g(s_1, s_2, \cdots, s_m))$$

where f and g are defined symbols, and p_1, p_2 are positions in r if:

1. *$r|_{p_1}$ and $r|_{p_2}$ share the same variables X at inductive positions of f and g (tupability),*
2. *$V(r|_{p_1}) = V(r|_{p_2})$ (separatability),*
3. *$r|_{p_1}$ and $r|_{p_2}$ have only constructors or variables at inductive positions of f and g and there is always a variable to share at a maximal inductive position (synthesizability),*
4. *$r|_{p_1}$ and $r|_{p_2}$ are different (noncommonality),*
5. *$r|_{p_1}$ and $r|_{p_2}$ are linear (linearity).*

If two tupable subterms s_1 and s_2 are identical, they can be shared (using a local let declaration) by the translation of the resulting rewrite system into an ML program. For example the pair $(k(x), k(x))$ in the right-hand side of the rule $f(x :: xs) \rightarrow (x + k(xs)) :: (k(xs) :: [])$ is not considered for tupling. It can be translated into :

$$\textbf{fun} f(x :: xs) = \textbf{let val } v = k(xs) \textbf{ in } (x + v) :: (v :: []) \textbf{ end}.$$

Linearity and separability ensures that the introduction of the tupling rules preserves the termination of the rewrite system (see [4]). Note that *fst*, *snd*, *pair* are reserved symbols (i.e. $\notin D \cup C$).

Automatic discovery of synthesis rules is based on the above analyses.

4 Analyses for automatization of the syntheses

The synthesis rules we have seen so far are of the form (1) $s \rightarrow h(x_1, x_2, \cdots, x_n)$ for a fusion rule, or of the form (2) $s \rightarrow fst(h(x_1, x_2, \cdots, x_n))$ and $s \rightarrow snd(h(x_1, x_2, \cdots, x_n))$ for the tupling rules. Such rules rewrite right-hand sides of the system R to get a system $\mathbf{R_{fold}}$. So doing, modulo the synthesis of the constructor-based rules R_h, we perform the desired fusion or the desired tupling. Moreover we have proved in [4] that R_{fold} is terminating when R is terminating.

4.1 Critical pair analysis

Overlaps between a synthesis rule and a rule of R_{fold}: $l \rightarrow r$ happen, either when the top symbol f of l occurs as a producer in a fusion term left-hand side of the fusion rule, or when f is the top symbol of an element of a tupling pair in the left-hand side of a tupling rule. Such an overlap produces a critical pair that substitutes constructor terms into the variables that are located

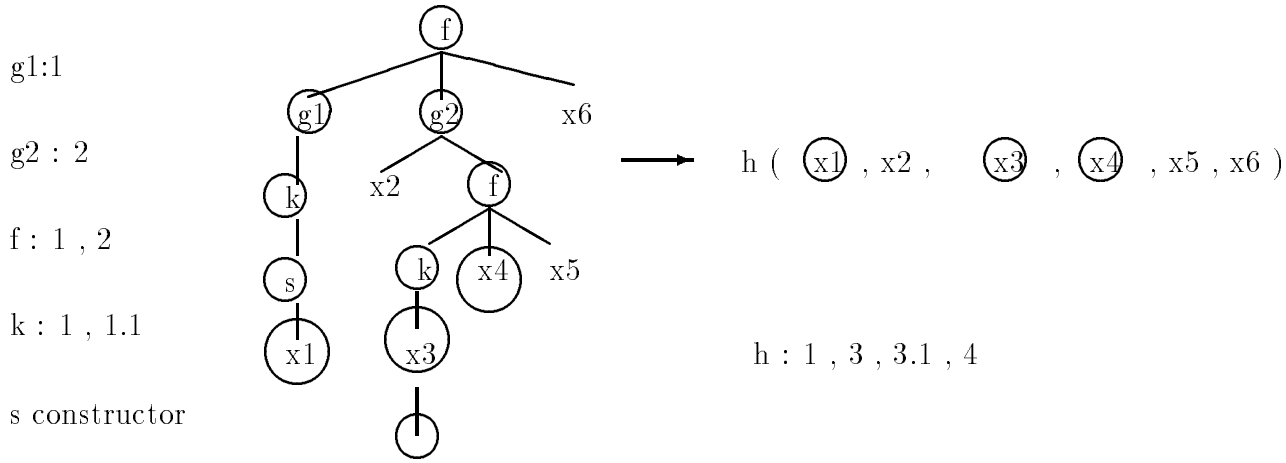
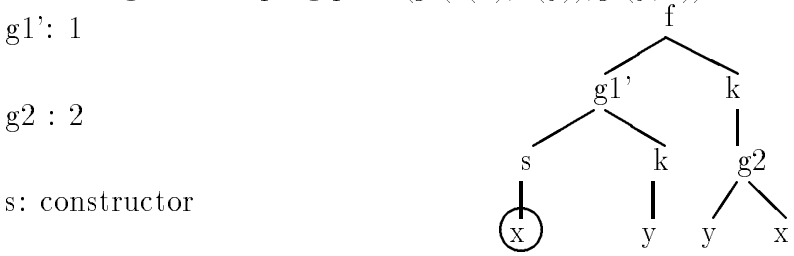


Figure 4: Inductive Positions of h

Figure 5: Tupling pair: $(g_1(s(x), k(y)), g_2(y, x))$



circled x is not in a spine position

Figure 6: No tupling pair

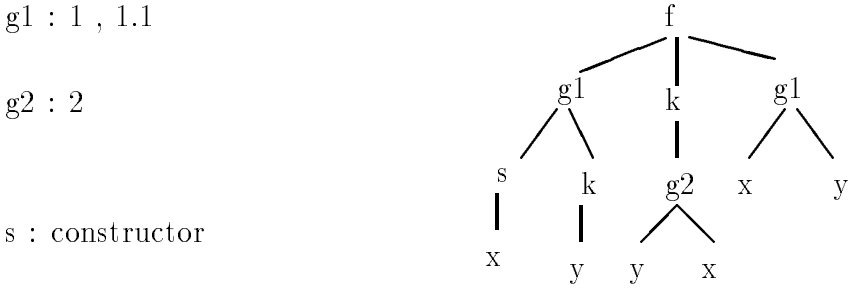


Figure 7: Tupling triple: $(g_1(s(x), k(y)), g_2(y, x), g_1(x, y))$

under the symbol h at inductive positions of h. Therefore, it is a step towards the synthesis of a constructor-based rule of R_h. This critical pair has one of the following forms:

$$(s', h(tc_1, tc_2, \cdots, tc_n)) \qquad (s', fst(h(tc_1, tc_2, \cdots, tc_n))) \qquad (s', snd(h(tc_1, tc_2, \cdots, tc_n)))$$

where $tc_{i_{i=1,n}}$ is a constructor term. Let us now analyze these critical pairs. For example, the critical pair $(f([], k(z), g(y)), h([], y, z))$, where h has inductive positions 1 and 2, is said to be **unfilled** because there is a variable y at position 2. This position can be filled by an overlap between the critical pair and a rule of R_g such as $g([]) \rightarrow []$. In this case, we say that the critical pair is **noncovered**. On another hand, we do not want to process an overlap with a rule of R_k such as $k([]) \rightarrow []$ which fills the noninductive third position. A **constrained** critical pair verifies the following: *Only variables are located at noninductive positions of h. Every critical pair generated by the completion which does not verify this property is rejected by the process.* In a **constrained** critical pair, consider the subterm tc_i where i is an inductive position of h. If tc_i is a variable, we say that the inductive position i in **unfilled**. Otherwise, we say that the position i is **filled**. A constrained critical pair with some unfilled positions is said to be **unfilled**. If there exists an overlap with a rule of R_{fold} that could fill this position, as in the above example, we say that critical pair is **noncovered**, otherwise, we say that it is **covered**. Examples in Figure 8 illustrate each of the three possible kinds of critical pairs. *The process forces completion*[3] *not to use noncovered critical pairs as rewrite rules* because they could destroy the termination property, or not even be a rewrite rule at all when $V(s') \subset V(h(tc_1, tc_2, \cdots, tc_n))$. Rejection and orientation of the critical pairs is automatic.

By definition, there exist no overlap to be done with a **covered** critical pair. We must be able to analyze if we can derive a rule of R_h from such a pair.

4.2 Synthesizability analysis of covered critical pairs

From now on, we consider that the covered critical pairs are in normal form. We want to analyze these critical pairs in order to decide if the synthesis is successful.

Consider the case of the tupling strategy: Let $(f(t_1, t_2, \cdots, t_n), g(s_1, s_2, \cdots, s_m))$ be the tupling pair. Our **criterion for success** is simple: *f and g* **must not occur** *in* **normalized covered** *critical pairs.* For example, let the tupling pair be $(g(x, []), flat(x))$, the covered critical pair: $P : (pair(g(xs, [x]), x \ @ \ snd(h(xs))), h(x :: xs))$ does not satisfy the criterion. In P, g and h share the variable xs at inductive positions. The **criterion for success** guarantees that the definition of R_h does not show a parallel traversal for f and/or g. Therefore **it guarantees that the transformation is effective.**

Consider now the case of a fusion: Consider the fusion term s. A consumer can only consume constructor symbols that are produced by a producer. This is so because it is defined by constructor-based rules. When a producer does not produce a constructor, e.g. when it has defined symbols or primitive symbols at its top, those symbols cannot be consumed. They stay

[3] Ordered (or unfailing) type of completion also keep nonorientable critical pairs as pairs. We can qualify our synthesis process as being controlled by an unfailed (according to our analysis, it keeps noncovered critical pairs as pairs) partial (according to our analysis, it rejects some critical pairs) completion procedure.

$$\begin{cases} f(x :: xs) \rightarrow (x + x) :: f(xs) \\ f([]) = [] \end{cases} \qquad \begin{cases} g(x :: xs) \rightarrow (x * x) :: g(xs) \\ g([]) = [] \end{cases}$$

fusion rule: $zip(f(x), g(y)) \rightarrow h(x, y)$

overlapping with g to be done

$(Zip ((x + x) :: f (x) , g (y)) , h (x :: xs , y))$

unfilled position

Unfilled Uncovered Critical Pair

---

fusion rule: $zip(f(x), y :: ys) \rightarrow h(x, y)$

filled positions

$((x + x , y) :: Zip (f (xs) , ys) , h (x :: xs , y :: ys))$

Filled Covered Critical Pair

---

fusion rule: $zip(f(x), y) \rightarrow h(x, y)$

no more overlapping to be done

unfilled

$([] , h ([] , y))$

Unfilled Covered Critical Pair

Figure 8: Diverse kinds of Critical Pairs

in s, obstructing the fusion. Therefore the **criterion for successful fusion** is the following: *the fusion term s must not be* **embedded** *(modulo a renaming of the variables) into a covered normalized critical pair.* Intuitively, a term t is embedded into a term s if we can get t from s by removing some nodes to s. For example, the fusion term $length(flat(x))$ is embedded into the covered critical pair:

$$(length(x \ @ \ flat(xs)), h(x :: xs)).$$

This embedding exists because the consumer $length$ is unable consume the symbol @ produced by $flat$ (see Figure 10)[4]. The **criterion for successful fusion** guarantees that fusion has been done.

Successful covered critical pairs are oriented into rules of R_h. For the tupling strategy, the discovery of an unsuccessful covered critical pair results in automatically undoing the completion and automatically rejecting the tupling pair from the search. For the fusion strategy, completion is used to turn an unsuccessful covered critical pair into a successful one.

4.3 Secondary fusions

Let us illustrate the process using the above example (see Figure 10). The process introduces the **secondary fusion rule** $length(x \ @ \ y) \rightarrow h_1(x, y, length(y))$. This indicates that the argument y which subsumes $flat(xs)$ must be consumed by $length$. Let us call $length(y)$ a **subsumption term** in the unsuccessful covered critical pair. The **secondary fusion rule** forces the consumer $length$ to consume @ produced by $flat$. Using this rule, the unsuccessful covered critical pair is reduced into the now successful (according to our criterion for success) covered critical pair:

$$(h_1(x, flat(xs), h(xs)), h(x :: xs))$$

In this example, the useless and costly occurrence of $flat(xs)$ will disappear later in the process. We will discuss this issue in the next section. By synthesizing R_{h_1}, one of the rules obtained by completion is:

$$h_1(x :: xs, y, length(y)) \rightarrow 1 + h_1(x, y, length(y))$$

It is not a constructor-based rule yet because of the presence of $length(y)$ as an argument. However, this argument can be **subsumed** by a variable. The result of this generalization is the constructor-based rule:

$$h_1(x :: xs, y, z) \rightarrow 1 + h_1(x, y, z)$$

Now we can analyze this rule, looking for useless arguments. This analysis shows that the second argument is useless. The process automatically introduces a rewrite rule:

$$h_1(x, y, z) \rightarrow h'_1(x, z)$$

By interreduction, the synthesis of R_h is completed as shown in Figure 10. In a few cases, there will be no useless arguments to remove (see next section).

[4]For this example, suppose the rewrite system R contains the rewrite rule (law) $length(x \ @ \ y) \rightarrow length(x) + length(y)$, the normalized covered critical pair is then $(length(x) + h(xs), h(x :: xs))$ which is successful.

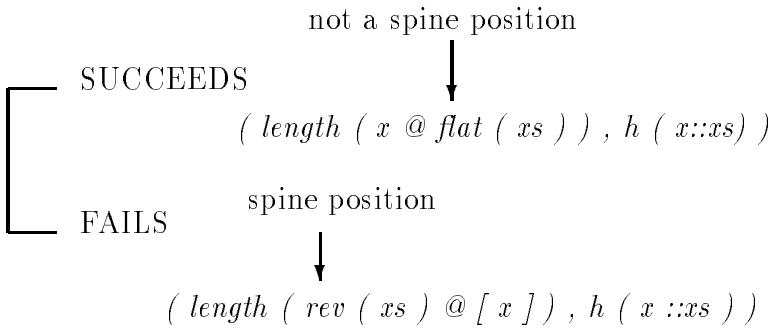


Figure 9: Subsumption Criterion

Unfortunately, subsumption could fail to turn a covered secondary critical pair into a constructor-based rule. Consider, for example, the fusion term $length(rev(x))$ and the unsuccessful covered critical pair: $(length(rev(xs)) \ @ \ [x]), h(x :: xs))$. A secondary fusion rule is:

$$length(x \ @ \ y) \rightarrow h_2(x, y, length(x))$$

where x is located at spine position. The overlaps with rules of $R_@$ substitutes x so that the subsumption term $length(x)$ gets modified. Therefore its subsumption would not give a constructor-based rule. For example, one of the covered critical pairs would be:

$$(1 + h_2(x, y, length(xs)), h_2(x :: xs, y, 1 + length(xs)))$$

Subsumption criterion: *The unsuccessful covered critical pair can be transformed by a secondary fusion if the producer does not occur at spine position* (see figure 9). This criterion guarantees that the subsumption term in the secondary fusion rule is preserved through critical pairs, hence ensuring that its subsumption results in a constructor-based rule.

A criterion for successful argument removal also needs to be developed. For example, the second argument is removable in the definition of h_1 above, but the second and third arguments are not removable in the definition of k below:

$$k([], y, z) = z \qquad k(x :: xs, y, z) = (x :: (xs \ @ \ y)) :: k(xs, y, z)$$

because y and z occurs outside the recursive calls.

Argument removal criterion: *A variable located at a noninductive position in the left-hand side of rules of R_f indicates a useless position if, either it does not occur in the right-hand sides, or it only occurs at the same position in recursive calls.*

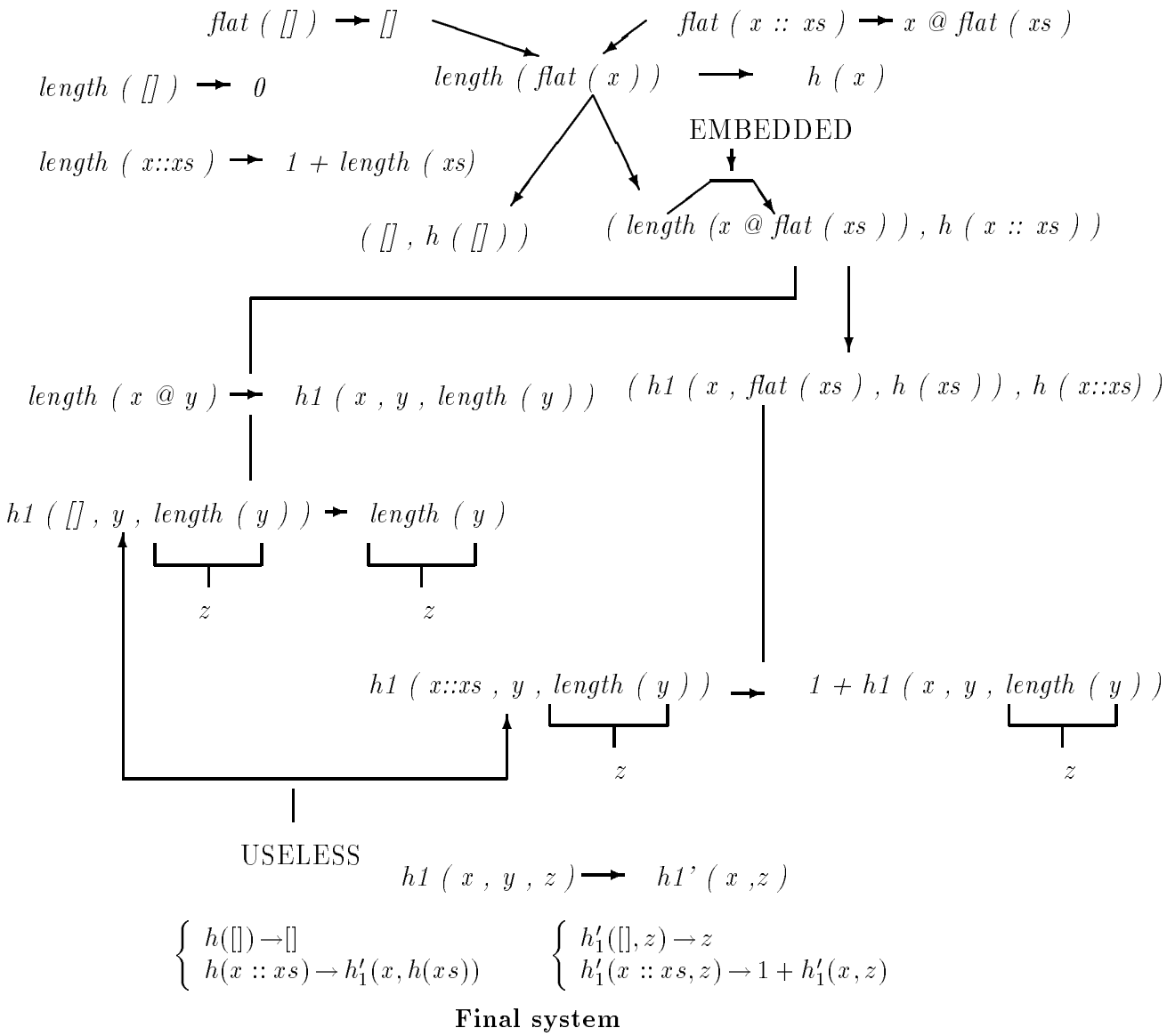


Figure 10: Synthesis and secondary synthesis

5 Diverse Issues

5.1 Termination

There are different kind of termination issues. We proved in [4] that the process preserves the termination of the rewrite system. The termination of the completion which performs each synthesis is guaranteed because there can be only a finite number of overlaps that covers the inductive positions of a fresh symbol. Moreover, we must also guarantee that the whole process does not generate infinitely many fusion or tupling rules. Consider, for example, the rules

$$f(x :: xs) = x :: p(f(x)) \qquad p(x :: xs) = x :: p(xs)$$

When looking for a fusion term, our process *skips over the recursive calls* (see Section 3) so that it does not introduce the fusion rule $p(f(x)) \rightarrow h_0(x)$. But suppose it does: A first synthesis would return $h_0(x :: xs) \rightarrow x :: p(h_0(x))$. This would yield to a second fusion rule $p(h_0(x)) \rightarrow h_1(x)$ resulting into $h_1(x :: xs) \rightarrow x :: p(h_1(x))$. This would generate infinitely many syntheses. Unfortunately, the termination is not ensured only by skipping over the recursive calls because of mutually recursive calls. Let us define an equivalence between defined symbols $f \sim g$ if and only if f calls g and g calls f. Let us reinforce our **criterion for successful tupling** into: *Symbols equivalent to the top symbols f and/or g of the tupling pair must not occur in the covered critical pair.* Let us also reinforce our **criterion for successful fusion** into: *The fusion term must not be* **call-embedded** *(modulo renaming of the variables) in the covered critical pair.* The call-embedding is the embedding relation induced by $\sim$. For example, $f(x, g(0, y))$ is embedded in $k(f(x, h(k(g(0, y)))))$; it is also call-embedded in $k(f'(x, h(k(g'(0, y)))))$ if $f' \sim f$ and $g' \sim g$. The covered critical pair is declared successful or unsuccessful according to this stronger criterion. Moreover, a secondary fusion rule can be built, according to the process described in Section 4, by replacing the equality of symbols by $\sim$. These criteria guarantee the termination of the process. A sketch of a proof for fusions without secondary rules is done in [4].

5.2 Effectiveness

The question we raise here, is about the *effectiveness* of our process. Does a fusion or a tupling always remove intermediate data structures or parallel traversals without loss of efficiency? The answer to the question is positive for the tupling: The tupling pair (s_1, s_2) occurring in r is replaced by $fst(h(x_1, x_2, \cdots, x_n)), snd(h(x_1, x_2, \cdots, x_n))$ where $\{x_1, x_2, \cdots, x_n\} = V(s_1) = V(s_2)$. Later, a local let can be introduced to share the common sub-expression $h(x_1, x_2, \cdots, x_n)$ as we suggested in Section 3 so the tupling is effective at this level of the transformation. Moreover, the synthesizability condition in the definition of the tupling pair (see Section 3 together with the criterion for success for covered critical pairs (see Section4) guarantee the effectiveness of the removal.

Unfortunately, the answer to the above question is negative for the fusion when the consumer is nonlinear for an inductive position. Consider, for example, the function:

$$tails([]) = [] :: [] \qquad tails(x :: xs) = (x :: xs) :: tails(xs)$$

and the fusion rule: $tails(x \ @ \ y) \rightarrow h(x, y)$. The completion returns R_h:

$$\begin{cases} h([], y) \rightarrow tails(y) \\ h(x :: xs, y) \rightarrow (x :: (xs \ @ \ y)) :: h(xs, y). \end{cases}$$

h appends each *tail* of x to y which results in an obvious loss of efficiency. We could have chosen to skip over such fusions. However, we noticed that the inefficiency disappears by tupling. For the above example, processing the tupling pair $((xs \ @ \ y), h(xs, y))$ results in:

$$\begin{cases} h([], y) \rightarrow tails(y) \\ h(x :: xs, y) \rightarrow (x :: fst(h'(xs, y)) :: snd(h'(xs, y)). \end{cases}$$

where:

$$\begin{cases} h'([], y) \rightarrow pair(y, tails(y)) \\ h'(x :: xs, y) \rightarrow pair(x :: fst(h'(xs, y)), (x :: fst(h'(xs, y))) :: snd(h'(xs, y))) \end{cases}$$

The translation can now share the common sub-expression $h'(x, y)$. When a consumer is nonlinear for an inductive position, the ineffectiveness of the fusion is corrected by tupling.

This phenomenon is even more obvious with a secondary fusion rule. In the example in Figure 10, the subterms $(flat(xs), h(xs))$ are tupable in: $(h_1(x, flat(xs), h(xs)), h(x :: xs))$. Tupling would apply if the second argument of h_1 could not be removed. As we saw earlier in Section 4, we choose systematically to add possibly too many arguments, creating beforehand and systematically an inefficiency. This inefficiency is recuperated afterwards, either by elimination the useless argument as shown in the fusion $length(flat(x))$, or by tupling when the argument is not removable.

We choose to systematically perform all the fusions before tuplings for the following reasons:

1. Tupling corrects the ineffectiveness of some fusions as we shown above;
2. It is only after fusions, that a subterm $f(t_1, t_2, \cdots, t_n)$ where $f \in D$ meet the synthesizability condition for tupling elements (see Section 3). For example, let f and f' have inductive position 1, the term $f(g(x), k(y))$, where g is a defined symbol (or a constructor) cannot be put in a tupling pair with $f'(x, y)$ because g occurs at a maximal inductive position of f. But first it can be transformed by fusion into $h(x, k(y))$, and second it can become an element of a tupling pair with $f'(x, y)$;
3. Recall that we do not fuse recursive calls. Tupling can remove intermediate data structures provided by recursive calls (like in the fib example).

5.3 Possible Extensions

In this section, we consider the limits of the automatic process for deforestation and parallel traversal removal we described in the paper. We indicate possible ways to overcome these limitations.

Consider the fusion: The process we described performs each fusion that does not involve a recursive call as producer or consumer (see Section 3) and that satisfies the *strong subsumption criterion* (see Subsection 5.1). Another presentation of the function can overcome syntactic limitations. For example, we can *fuse* some recursive calls by using the well known tactic of introducing accumulative variables so that the recursive call occurs at the top of the right-hand side. This tactic is called *nontail recursion removal*. Using this tactic,

$$rev([]) \rightarrow [] \qquad rev(x :: xs) \rightarrow rev(xs) \ @ \ (x :: [])$$

is transformed into

$$rev(x) \rightarrow g(x, [])$$
$$g([], u) \rightarrow u \qquad g(x :: xs, u) \rightarrow g(xs, x :: u)$$

The intermediate data structure created by the recursive call of rev disappears in this new presentation. The tactic requires to know that @ is associative and has [] as right identity. This transformation allows us to perform fusions that would be otherwise rejected by the subsumption criterion: As we shown as example in Section 4), the fusion of $length(rev(x))$ is rejected because rev occurs in an inductive position of @ in $(length(rev(xs)) \ @ \ [x]), h(x :: xs))$. Let us suppose that, at this point in the process, we modify the presentation of rev by using the accumulator introduction tactic, then, the fusion of $length(rev(x)$ becomes possible because the producer rev is at the top, and, as such, is ready to be consumed at any time. Note that when this transformation does not work, laws can be used within our process without difficulty; mixing laws within a rewrite system is always possible providing that it preserves the termination of the rewrite system.

Consider the tupling: The process we described does not remove all parallel traversals because of the separability requirement we impose on the tupling pair. This, we can easily overcome. For example, suppose f and g have inductive position 1, the subterms $f(xs, k(y, z))$ and $g(xs, y)$ of a term r, which have not the same set of variables, will not be considered as a tupling pair though they both traverse the same data structure. If we were processing this pair, we will not only get the tupling rule $f(xs, k(y, z)) \rightarrow h(x, y, z)$ but also $G : g(xs, y) \rightarrow h(x, y, z)$ which cannot be a rewrite rule. We could consider G as a pair whose left element can be overlapped but which cannot be used for rewriting. In this case, the subterm $g(xs, y)$ in r would have to be replaced by $h(xs, y, z)$ by direct replacement and not by rewriting.

6 Related Work

There have been a number of different proposals for program transformations that are derived from the unfold/fold method. Let us consider only the works which automatize deforestation of first-order functional programs.

The ancestor is Wadler's deforestation algorithm in [19]. Fusion terms must be composed of treeless function symbols to be accepted by the algorithm. Using our terminology, a defined symbol f is treeless if the right-hand side of rules in R_f are linear and built solely with constructors and calls of defined functions with variables as arguments. For example, @ and zip are treeless, but rev, $flat$ and $tails$ are not. It is easy to see the reasons behind these restrictions:

(1) if a right-hand side in R_f is not linear (like for R_{tails}), we have seen that the fusion risks to be inefficient. We have not this limitation because we use tupling afterwards, (2) if a recursive call in a right-hand side has a term as argument, it will not be able to consume a producer at this position (in this case, our process yields an embedding of the fusion term which creates a secondary fusion), (3) if a call of a function g in a right-hand side has a term as argument which contains a recursive call (like @ for $flat$ or rev), g will create an obstruction as producer. Again, in our process, we get an embedding of the fusion term which creates a secondary fusion. Wadler's deforestation algorithm considers composed terms like x @ $d(x)$ that we reject because $d(x)$ occurring at a noninductive position of @ does not create an intermediate data structure. We consider this case as another tactic for program manipulation which, from our point of view, is not a deforestation but a composition removal. This complementary tactic is also automatizable using completion and other kind of analyses.

Chin in [7] has extended Wadler's work by refining the analysis. He annotates an argument position i of a defined symbol as *safe* if, in R_f, the variables of the argument tc_i occurs only once and, moreover, if in the recursive call $f(t_1, t_2, \ldots, t_n)$, t_i is a variable or a constant. A function f can consume only at safe positions. The producer is safe if its recursive calls do not occur in a safe position of a consumer. This makes $tails$ unsafe consumer but safe producer. The functions $flat$ and rev are unsafe producers but they are safe consumers. Therefore Chin's extended deforestation algorithm accepts fusions such as $length(tails(x))$, $flat(x@y)$, and $rev(x@y)$ as safe. It rejects fusions $tails(x@y)$, $length(flat(x))$, and $length(rev(x))$ as unsafe. Wadler's rejects all of them. These annotations permit the *extended deforestation algorithm* to track all the effective fusions that do not require laws or secondary fusions.

A recent article [17] compares diverse transformation techniques: deforestation, supercompilation, partial evaluation, and generalized partial computation with respect to the amount of information propagation they enable. It proposes a *widened deforestation algorithm* which enables more information propagation than Chin's extended deforestation algorithm.

Our superiority with respect to these algorithms for deforestation is: (1) that we combine both deforestation and tupling, and (2) that we perform the secondary fusions. This extends the class of the deforestation that are automatizable.

7 Conclusion

The analyses we propose in this paper allow us to use completion in an effective and automatic way for combining the two transformation tactics of deforestation and tupling[5]. First-order functional programs presented by a set of pattern-matching definition rules are translated into a constructor-based orthogonal rewrite system. Rewrite techniques apply if the system is terminating. We were surprised to discover how combining deforestation and tupling can be fruitful. The superiority of using a rewrite system and completion for synthesis is that we can manipulate equations which are not constructor-based. In other words, we are not limited by a pattern-matching presentation of the rules we use. This permits us to implement the tupling by using the non-constructor-based rules we call *tupling rules* in the paper. This

[5]Presently, the fully automatic process we present in this paper is implemented in ASTRE

also permits us to generate what we call the *secondary fusion rules* in the paper. The purely automatic system we describe in the paper performs most of the fusions which otherwise could not be processed without adding laws. Moreover, the user can introduce laws as he desires. These laws can facilitate fusions without perturbating the process. Without input of laws, thanks to secondary fusions and to tupling, the automatic process performs a more complete deforestation of the program than Chin's extended deforestation algorithm. Another advantage of our method is that it is possible to add incrementally other strategies for transformation. It is also easy to experiment with a new strategy and to try its interaction with others by mixing the user-directed version of the completion with the automatized one(s). However, we are technically restricted to first-order terminating rewrite systems. Presently, we use a higher-order removal transformation performed by partial evaluation as a preprocessor of our system. Moreover, we can combine the partial evaluator Schism [8] with our system [3]. Partial evaluation, on the fly, transforms a given set of higher order laws into the first-order laws that facilitate a transformation.

Acknowledgements I wish to thank Olivier Danvy for its encouragement and comments and Laura McKinney for her help in proof reading the paper.

References

[1] F. Bellegarde. Program Transformation and Rewriting. *Proc. 4th Int. Conf. on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 226-239, Springer Verlag, 1991.

[2] F. Bellegarde. Astre, a Transformation System using Completion. Technical Report, Oregon Graduate Institute, 1991.

[3] F. Bellegarde. A transformation System Combining Partial Evaluation with Term Rewriting, *Proc. HOA'93: An international Workshop on Higher Order Algebra, Logic and Term Rewriting*, to appear in a volume of *Lecture Notes in Computer Science*. Amsterdam, Sept. 93.

[4] F. Bellegarde. Automatic Transformation by Rewriting Techniques. Technical Report N^o 94 – 009, Oregon Graduate Institute, 1994.

[5] W.H. Burge *Recursive Programming Techniques*, Addisson-Wesley, 1975.

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

[7] W. N. Chin. Safe fusion of Functional Expressions II: Further Improvements. *J. of Functional Programming*, 11:1-40, 1994.

[8] C. Consel and O. Danvy. Tutorial Note on Partial Evaluation. *Conf. Record of the Twentieth Annual ACM Symposium on Principle of Programming Languages*, pages 493-501. ACM, 1993.

[9] M. Fokkinga. Tupling and Mutamorphisms. *The Squiggolist*, 1(4), 1989.

[10] N. Dershowitz. *Computing with rewrite systems,* Information and Control:65:122-157, 1985.

[11] N. Dershowitz. Completion and its Applications. *Resolution of Equations in Algebraic Structures,*, 2:31-86, Academic Press, 1988.

[12] A. Gill, J. Launchbury and S.L. Peyton Jones. A short cut to Deforestation. *Proc. of the 6th Conf. on Functional Programming Languages and Computer Architecture*, Copenhagen, pages 223-232, June 1993.

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

[14] J. Hughes. Why Functional Programming Matters. Computer Journal, 32(2):98-107, 1989.

[15] U. S. Reddy. Rewriting Techniques for Program Synthesis. *Proc. of the 3rd Conf. on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 388-403, Springer Verlag, 1989.

[16] T. Sheard and L. Fegaras. A fold for All Seasons. *6th Conf. on Functional Programming Languages and Computer Architecture*, pages 233-242, 1993.

[17] M. H. Sørensen, R. Glück and N.D. Jones. Towards Unifying Deforestation, Supercompilation, Partial Evaluation and Generalized Partial Computation. *Proc. of Eureopean Symposium on Programming*, *Lecture Notes in Computer Science*. Springer Verlag, April 1994.

[18] P. Wadler. Listlessness is better than laziness II: Composing listless functions. *Workshop on Program as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 282-305. Springer Verlag, Copenhagen, 1985.

[19] P. Wadler, Deforestation: Transforming programs to eliminate trees. *Proc. of European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344-358. Springer Verlag, 1988.