Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC

Morten Heine Sørensen, Robert Glück & Neil D. Jones

Authors' address:[**] DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, E-mail: rambo@diku.dk, glueck@diku.dk, neil@diku.dk

Abstract. We study four transformation methodologies which are automatic instances of Burstall and Darlington's fold/unfold framework: *partial evaluation, deforestation, supercompilation,* and *generalized partial computation (GPC).* One can classify these and other fold/unfold based transformers by how much information they maintain during transformation.

We introduce the *positive supercompiler*, a version of deforestation including more information propagation, to study such a classification in detail. Via the study of positive supercompilation we are able to show that partial evaluation and deforestation have simple information propagation, positive supercompilation has more information propagation, and supercompilation and GPC have even more information propagation. The amount of information propagation is significant: positive supercompilation, GPC, and supercompilation can specialize a general pattern matcher to a fixed pattern so as to obtain efficient output similar to that of the Knuth-Morris-Pratt algorithm. In the case of partial evaluation and deforestation, the general matcher must be rewritten to achieve this.

1 Introduction

Our aim is to compare certain, automatic instances of the Burstall-Darlington framework. As is well-known, the basic techniques of the framework are: *unfolding, instantiation, definition, folding,* and *abstraction.* In addition to applying these mechanisms, the transformers we consider maintain information such as the previously encountered terms, the previously encountered tests, *etc.*

Partial evaluation, discussed at length in [Jon93], replaces calls with some arguments known, *e.g.* $f 2 v$, by specialized calls, *e.g.* $f_2 v$, where f_2 is an optimized version of f taking into account that the first argument is known to be 2. In *offline* partial evaluators, application of Burstall-Darlington transformations is guided by automatically generated *program annotations* that tell where to unfold, instantiate, define, and fold.

Deforestation, due to Wadler [Wad88, Fer88], can remove some intermediate data structures altogether, thus reducing the number of "passes" over data.

[**] This work was supported by the Austrian Science Foundation (FWF) under grant number 10780 and the Danish Research Council.

Perhaps less well known, it can also partially evaluate. For instance, applying deforestation to $a\,(A : B : x)\,(C : y)$, where a is the well-known *append* function, yields $A : B : f\;x\;y$ where f is defined as

$$\begin{array}{ll} f\;[]\;ws & \leftarrow C : ws \\ f\;(v : vs)\;ws & \leftarrow v : (f\;vs\;ws) \end{array}$$

Termination-safe extensions of deforestation [Chi93, Sor94a] use automatically precomputed annotations to tell where to abstract (=generalize=extract) so that enough folding takes place to ensure termination.

Supercompilation is a powerful technique due to Turchin [Tur86] (continuing Soviet work from the early 1970's) which can achieve effects of both deforestation and partial evaluation. Supercompilation performs *driving:* unfolding and propagation of information, and *generalization:* a form of abstraction which enables folding. The decision when to generalize is taken online. Recent work by Glück and Klimov has expressed the essence of driving in the context of a more traditional tail-recursive language manipulating Lisp-like lists [Glu93].

Generalized partial computation (GPC) due to Futamura and Nogi [Fut88] and later applied to a lazy functional language [Tak91] has similar effects and power as supercompilation, but has not yet been implemented.

The remainder of the paper is organized as follows. In Section 2 we introduce some terminology that allows us to discuss the quality of the output of transformers. In Section 3 we introduce the positive supercompiler, discuss its correctness, and point out the differences in its rules compared to deforestation. Following up on this, we compare in Section 4 the notion of information propagation in deforestation with that in positive supercompilation. We observe that the deforestation algorithm cannot derive efficient Knuth-Morris-Pratt style pattern matchers, while the positive supercompiler can, and we explain why. In Section 5 we extend the comparison to partial evaluation, GPC, and traditional supercompilation. The last section concludes and reviews directions for future research.

2 A test for program transformers

A way to test a method's power is to see whether it can derive certain well-known efficient programs from equivalent naive and inefficient programs. One of the most popular such tests is to generate, from a general pattern matcher and a fixed pattern, an efficient specialized pattern matcher as output by the Knuth-Morris-Pratt algorithm [Knu77]. We shall call this *the KMP test.*

2.1 General, naively specialized, and KMP specialized matchers

Two different *general* pattern matchers are at work in the literature: a tail-recursive one, and one with nested calls. We shall be concerned with the tail-recursive one. Except where we deny it explicitly, everything carries over to the nested version.[3]

Definition 1 General, tail-recursive matcher.

```
$match\ p\ s$                          $\leftarrow loop\ p\ s\ p\ s$
$loop\ []\ ss\ op\ os$                 $\leftarrow True$
$loop\ (p : pp)\ []\ op\ os$           $\leftarrow False$
$loop\ (p : pp)\ (s : ss)\ op\ os$     $\leftarrow p = s\ \rightarrow loop\ pp\ ss\ op\ os \ \square\ next\ op\ os$
$next\ op\ []$                         $\leftarrow False$
$next\ op\ (s : ss)$                   $\leftarrow loop\ op\ ss\ op\ ss$
```

Consider the *naively specialized* program $f\ u \leftarrow match\ AAB\ u$ which matches the fixed pattern AAB with a string u. Evaluation of $match\ AAB\ u$, given u, proceeds by comparing A to the first component of u, A to the second, B to the third. If at some point the comparison failed, the process is restarted with the tail of u.

However, this strategy is not optimal *e.g.* if the string u begins with three A's. It is inefficient to restart the comparsion with the tail $AA...$ of u since it is already known that the first two tests of AAB against $AA...$ will succeed. The following *KMP-style specialized* matcher corresponding to the DFA constructed by the Knuth-Morris-Pratt algorithm [Knu77] takes this information into account.

Example 1 KMP-style specialized matcher for AAB.

```
$match_{AAB}\ u$          $\leftarrow loop_{AAB}\ u$
$loop_{AAB}\ []$          $\leftarrow False$
$loop_{AAB}\ (s : ss)$    $\leftarrow A = s\ \rightarrow loop_{AB}\ ss \ \square\ loop_{AAB}\ ss$
$loop_{AB}\ []$           $\leftarrow False$
$loop_{AB}\ (s : ss)$     $\leftarrow A = s\ \rightarrow loop_{B}\ ss \ \square\ loop_{AAB}\ ss$
$loop_{B}\ []$            $\leftarrow False$
$loop_{B}\ (s : ss)$      $\leftarrow B = s\ \rightarrow T \ \square\ A = s\ \rightarrow loop_{B}\ ss \ \square\ loop_{AAB}\ ss$
```

2.2 A comment on measuring complexity

One must be careful when discussing complexity of multi-input programs, especially in the context of program specialization when some inputs are fixed. For an example, let π be the general tail-recursive pattern matchers, and let $|p|, |s|$ denote the length of the pattern p and string s, respectively. Let $t_\pi(p, s)$ be the running time of program π on inputs p, s. Finally, let π_p be the result of specializing π to known pattern p by some transformation algorithm.

When π_p is a specialized KMP style pattern matcher, it is customary to say that the general $O(|p| \cdot |s|)$ time program has been transformed to an $O(|s|)$ time

[3] We use the Miranda notation for lists, *e.g.* $[A, A, B]$, as well as the short notation AAB. We shall even continue to do so after having introduced a language with slightly different syntax in Section 3.

program. This is, alas, *always true*, even for trivial transformations such that of Kleene's $s - m - n$ Theorem [Kle52]. The reason is that as soon as $|p|$ is fixed, then $O(|p| \cdot |s|) = O(|s|)$, even though the coefficient in $O(\_)$ is proportional to $|p|$.

To be more precise, define the *speedup function* as in [Jon93] as

$$speedup_p(s) = \frac{t_\pi(p, s)}{t_{\pi_p}(s)}$$

Now for any p there is a constant a and there are infinitely many subject strings s such that $t_\pi(p, s) \geq a \cdot |p| \cdot |s|$. Using a trivial specializer as in the $s - m - n$ Theorem it is easy to see that π_p has essentially the same running time as π, so $speedup_p(s) \approx 1$.

On the other hand, using non-trivial transformers (see later), the program π_p satisfies $t_{\pi_p}(s) \leq b \cdot |s|$ for any subject string s, where b is independent of p. As a consequence

$$speedup_p(s) \geq \frac{a \cdot |p|}{b}$$

This is particularly interesting because the speedup is not only significantly large, but becomes larger for longer patterns.

We shall say that the KMP test is passed by a transformer when it holds that there is a constant b such that for all s, $t_{\pi_p}(s) \leq b \cdot |s|$.

In Section 4 we investigate the KMP test for deforestation and positive supercompilation. In Section 5 we review and explain the known results of the KMP test for partial evaluation, supercompilation, and GPC, and relate all these results.

3 Positive supercompilation

We first present the object language. Next we describe some notions that are convenient for the formulation of the positive supercompiler. We then define the positive supercompiler. Finally we describe its relation to deforestation as defined in [Fer88], and consider the correctness of positive supercompilation.

3.1 Language

The language below extends that of [Fer88] by the presence of conditionals (equality tests between arbitrary terms).

Definition 2 Object language.

$$\begin{array}{lll} t & ::= v & \text{(variable)} \\ & \mid\ c\, t_1 \ldots t_n & \text{(constructor)} \\ & \mid\ f\, t_1 \ldots t_n & \text{(f-function call)} \\ & \mid\ g\, t_0\, t_1 \ldots t_n & \text{(g-function call)} \\ & \mid\ t_1 = t_2\ \rightarrow\ t_3 \,\Box\, t_4 & \text{(conditional)} \end{array}$$

$$\begin{array}{lll} d & ::= f\, v_1 \ldots v_n \leftarrow t & (f\text{-function definition, no patterns}) \\ & \mid\ g\, p_1\, v_1 \ldots v_n \leftarrow t_1 & \\ & \quad\quad \vdots & (g\text{-function definition with patterns}) \\ & \quad g\, p_m\, v_1 \ldots v_n \leftarrow t_m & \end{array}$$

$$\begin{array}{lll} p & ::= c\, v_1 \ldots v_n & \text{(patterns with one constructor)} \end{array}$$

As usual we require that left hand sides of definitions be *linear*, *i.e.* that no variable occurs more than once.[4] We also require that all variables in a definition's right side be present in its left side. To ensure uniqueness of reduction, we require that each function in a program have at most one definition and, in the case of a g-definition, that no two patterns p_i and p_j contain the same constructor. The semantics for reduction of a variable-free term is call-by-name, as realized in Miranda [Tur90] by "lazy evaluation."

Apart from the fact that the language is first-order there are two obvious and quite common restrictions: function definitions may have *at most one pattern matching argument*, and only *non-nested* patterns. (Methods exist for translating arbitrary patterns into the restricted form [Aug85]). In some examples we assume for simplicity that both the deforestation algorithm and the positive supercompiler can handle multiple pattern matching arguments.

3.2 How to find the next call to unfold

We shall express the positive supercompiler by rules for rewriting terms. The rewrite rules can be understood intuitively as mimicking the actions of a call-by-name evaluator — but extended to continue the transformation whenever a value is not sufficiently defined at transformation time to know *exactly which* program rule should be applied. If the applicable rule is not unique, then sufficient code will be generated to account for every run-time possibility.

For every term t two possibilities exist during transformation. (i) In the first case there are two subcases. If $t \equiv c\, t_1 \ldots t_n$, then transformation will proceed to the arguments (based on the assumption that the user will demand that the whole term's value be printed out), and if $t \equiv v$ transformation terminates.

(ii) Otherwise call-by-name evaluation forces a unique call to be unfolded or a branch in a unique conditional to be chosen. For instance, in the term

[4] Instead of adding an equality test, one could allow non-linear patterns. This would, however, encumber the formulation of the positive supercompiler algorithm.

$g\ (f\ t_1 \ldots t_n)\ t'_1 \ldots t'_m$ we are forced to unfold the call to f in order to be able to decide which clause of g's definition to choose. As another example, in the term $g\ (f\ t_1 = []\ \rightarrow\ t_2 \,\Box\, t_3)\ t_4$ we are forced to unfold the call to f to be able to decide between the branches. This, in turn, is forced by the need to decide which clause of g's definition to apply.

In case (ii) the term will be written: $t \equiv e[r]$ where r identifies the next function call to unfold, or the conditional to choose a branch in, and e is the surrounding part of the term. Traditionally, these are the *redex* and the *evaluation context*. The intention is that r is either a call which is ready to be unfolded (no further evaluation of the arguments is necessary), or a conditional in which a branch can be chosen (the terms in the equality test are completely evaluated).

We now define these notions more precisely.

Definition 3 Evaluation context, redex, observable, value.

$e ::= []$	Evaluation contexts
$\mid\ g\ e\ t_1 \ldots t_n$	
$\mid\ e' = t_2\ \rightarrow\ t_3 \,\Box\, t_4$	(First reduce left of =)
$\mid\ b = e'\ \rightarrow\ t_3 \,\Box\, t_4$	(Then reduce right of =)
$e' ::= e \mid c\, b_1 \ldots b_{i-1}\, e'\, t_{i+1} \ldots t_n$	(Left to right under constructor in test)
$r ::= f\ t_1 \ldots t_n$	Redex
$\mid\ g\ o\ t_1 \ldots t_n$	
$\mid\ b_1 = b_2\ \rightarrow\ t \,\Box\, t'$	
$o ::=\ c\, t_1 \ldots t_n \mid v$	Observable
$b ::=\ c\, b_1 \ldots b_n \mid v$	Value

The expression $e[t]$ denotes the result of replacing the occurrence of $[]$ in e by t. A term with no variables is called *ground*.

It is easy to verify that any term t is either an *observable* o, which is a variable or has a known outermost constructor; or it decomposes uniquely into the form $t \equiv e[r]$ (*the unique decomposition property*). This provides the desired way of finding the next function call to unfold or the conditional in which to select a branch.

3.3 The Positive supercompiler

We can now define the positive supercompiler. The positive supercompiler consists of three elements divided into two phases. In the *transformation* phase, *driving* and *folding* is performed. In the *postprocessing* phase, postunfolding is performed. We first describe driving, then folding and postunfolding.

Driving. The driving part of the positive supercompiler is given in Figure 1. It takes a term and a program (the latter not written explicitly as an argument) and returns a new term and a new program. Following is some notation used in the algorithm.

Fig. 1. Positive supercompiler.

Definition 4.

(1) $\mathcal{W}[\![\, v \,]\!] \quad\Rightarrow v$

(2) $\mathcal{W}[\![\, c\, t_1 \ldots t_n \,]\!] \quad\Rightarrow c\,(\mathcal{W}[\![\, t_1 \,]\!]) \ldots (\mathcal{W}[\![\, t_n \,]\!])$

(3) $\mathcal{W}[\![\, e[f\, t_1 \ldots t_n] \,]\!] \quad\Rightarrow f^{\Box}\, u_1 \ldots u_k$
where
$$f^{\Box}\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, e[t^f\{v_i^f := t_i\}_{i=1}^n] \,]\!]$$

(4) $\mathcal{W}[\![\, e[g\,(c\, t_{n+1} \ldots t_{n+m})\, t_1 \ldots t_n] \,]\!] \Rightarrow f^{\Box}\, u_1 \ldots u_k$
where
$$f^{\Box}\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, e[t^{g,c}\{v_i^{g,c} := t_i\}_{i=1}^{n+m}] \,]\!]$$

(5) $\mathcal{W}[\![\, e[g\, v\, t_1 \ldots t_n] \,]\!] \quad\Rightarrow g^{\Box}\, v\, u_1 \ldots u_k$
where
$$g^{\Box}\, p_1\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, (e[t^{g,c_1}\{v_i^{g,c_1} := t_i\}_{i=1}^n])\{v := p_1\} \,]\!]$$
$$\vdots$$
$$g^{\Box}\, p_l\, u_1 \ldots u_k \leftarrow \mathcal{W}[\![\, (e[t^{g,c_l}\{v_i^{g,c_l} := t_i\}_{i=1}^n])\{v := p_l\} \,]\!]$$

(6) $\mathcal{W}[\![\, e[b = b' \;\rightarrow\; t \,\Box\, t'] \,]\!] \quad\Rightarrow \mathcal{W}[\![\, e[t'] \,]\!]$
if b, b' are ground and $b \equiv b'$

(7) $\mathcal{W}[\![\, e[b = b' \;\rightarrow\; t \,\Box\, t'] \,]\!] \quad\Rightarrow \mathcal{W}[\![\, e[t] \,]\!]$
if b, b' are ground and $b \not\equiv b'$

(8) $\mathcal{W}[\![\, e[b = b' \;\rightarrow\; t \,\Box\, t'] \,]\!] \quad\Rightarrow b = b' \;\rightarrow\; \mathcal{W}[\![\, (e[t])MGU(b, b') \,]\!] \,\Box\, \mathcal{W}[\![\, e[t'] \,]\!]$
if if not both b, b' are ground

Notation. Let p be the implicit program argument.

For f-functions, t^f denotes the right hand side of the definition for function f in p, and $v_1^f \ldots v_n^f$ are the formal parameters in f's definition.

For a g-function definition, $t^{g,c}$ is the right side of g corresponding to the left side whose pattern contains the constructor c. Further, $v_1^{g,c} \ldots v_n^{g,c}, v_{n+1}^{g,c} \cdots v_{n+m}^{g,c}$ are the formal parameters of g in the clause whose pattern contains the constructor c. Here $v_1^{g,c} \ldots v_n^{g,c}$ are those not occurring in the pattern (these are the same for all c) while $v_{n+1}^{g,c}, \ldots v_{n+m}^{g,c}$ are the variables in the pattern (here m depends on c). Finally, $p_1^g \ldots p_l^g$ are the patterns of g.

The expression $t\{v_i := t_i\}_{i=1}^n$ denotes the result of replacing all occurrences of v_i in t by the corresponding value t_i. In $(e[t])\{v_i := t_i\}_{i=1}^n$, the substitution is to be applied to all of $e[t]$, and not just to t.

The notation $MGU(b, b')$ denotes the most general unifier $\{v_i := t_i\}_{i=1}^n$, $(0 \leq n)$ of b, b' if it exists, and $fail$ otherwise. It is convenient to define $t\,fail \equiv t$.

The symbol $\Rightarrow$ denotes evaluation in the metalanguage, i.e. transformation.

For instance, in clause (3) the result of transforming $e[f\, t_1 \ldots t_n]$ is a call to a new function $f^{\square}$. This function is then defined with right hand side the result of transforming $t^f\{v_i^f := t_i\}_{i=1}^n$. The symbol "$\leftarrow$" refers to a definition in the object language (the language of definition 2).

The **where** should be read as a code generation command. A term $e[r]$ is transformed into a call to a new function $f^{\square}$; these new functions are collected somehow in a new program.[5] The variables $u_1 \ldots u_k$ in these calls are simply all the variables of $e[r]$. In clause (5) the variable v is not included in $u_1 \ldots u_k$.[6]

Folding and postunfolding. We imagine that the name of the new function and the order of the variables in the list of arguments are uniquely determined by $e[r]$, so if $e[r]$ is encountered later on, the same name and argument list are generated. If this happens the function should not be defined again (a *fold* step is performed). More: we shall assume that a fold step is also performed when a call is encountered which is a renaming of a previously encountered call.

After the transformation phase, all f-functions that are called exactly once in the residual program are unfolded.

3.4 Deforestation versus positive supercompilation

We henceforth call the deforestation algorithm $\mathcal{S}$ and the positive supercompiler $\mathcal{W}$.

The actions of $\mathcal{W}$ can be cast into the fold/unfold framework as follows. In clause (5) the term is transformed into a call to a new *residual* function. This involves a *define* step: a new function is defined; an *instantiation* step: the new function is defined by patterns; an *unfold* step: the body of the new function is unfolded one step; and a *fold step*: the right hand side of the original function is replaced by a call to the newly defined function.

Clauses (3) and (4) are similar except that there is no need for an instantiation step. We might also say that the instantiation step is trivial, regarding a variable as a trivial pattern and f-functions as defined by patterns. The operation of instantiation followed by unfolding of the different branches is called *driving* by Turchin.

Clauses (6),(7) can be understood as unfold steps similar to clauses (3),(4), and clause (8) can be understood as an instantiation step similar to clause (5).

The *essential difference* between $\mathcal{S}$ and $\mathcal{W}$ is found in clause (5): the pattern p_j^g is substituted for *all* occurrences of the variable v in $e[g\, v\, t_1 \ldots t_n]$. In $\mathcal{S}$ the pattern is only substituted for the occurrence of v between g and t_1; if there are more occurrences of v, then v must be included among the parameters $u_1 \ldots u_k$ in both the transformed call and the left hand side of the transformed definition.

[5] We take the liberty of being imprecise on this point.

[6] As a matter of technicality, the patterns p_j^g and terms t^{g,c_j} in clause (5) must actually be chosen as renamings of the corresponding patterns and bodies for g, and in clause (8) the unifier must be chosen *idempotent* which is always possible, see [Sor94b].

This very important difference implies that in $\mathcal{W}$ we are propagating more information: the information that v has been instantiated. It will turn out that this accounts for the fact that $\mathcal{W}$, but not $\mathcal{S}$, is able to derive KMP style pattern matchers.

It is easy to see that the algorithms have identical effects on linear programs.

Note that the distinction in clause (5) is not made explicit in [Fer88] since $\mathcal{S}$ is restricted to "treeless" programs, all of whose right sides are linear (although transformation of non-linear terms is briefly considered in [Fer88]).

In subsequent examples we shall assume that $\mathcal{S}$ has been extended to handle the conditional by adopting rules (6-8) leaving out the substitution in rule (8).

3.5 Operational semantics, efficiency, and termination

There are three issues of correctness for $\mathcal{W}$: preservation of operational semantics, nondegradation of efficiency, and termination.

First, the output of $\mathcal{W}$ should be semantically equivalent to the input. A proof of preservation of operational semantics, in a reasonable sense, is given in [Sor94b].

Second, the output of $\mathcal{W}$ should be at least as efficient as the input. Since rewriting to a nonlinear right hand side can cause function call duplication, this will not generally hold unless appropriate precautions are taken. The problem is well-known in partial evaluation [Ses88, Bon90, Jon93] and deforestation [Wad88, Chi93]. Essentially the same principles could be applied to $\mathcal{W}$, see [Sor94b].

Third, $\mathcal{W}$ should always terminate. The algorithm $\mathcal{W}$ does, in fact, not always terminate. Techniques to ensure termination of $\mathcal{W}$ for all programs are studied in [Sor94b].

4 KMP test of positive supercompiler and deforestation

In this section we observe that $\mathcal{S}$ cannot derive the KMP style pattern matcher. We explain why and show that $\mathcal{W}$ can derive a program almost as efficient as the KMP style pattern matcher. The derived program does contain inefficiency. We explain why and suggest how $\mathcal{W}$ can be extended to produce programs exactly as efficient as the KMP pattern matchers.

4.1 Pattern matching with deforestation

The result of applying deforestation $\mathcal{S}$ to the term $match\,AAB\;u$ is as follows, assuming post unfolding:

Example 2 Non-improved specialized matcher.

$$\begin{array}{lll}
 & & loop_{AAB}\ u\ u \\
loop_{AAB}\ []\ os & \leftarrow & False \\
loop_{AAB}\ (s : ss)\ os & \leftarrow & A = s\ \rightarrow\ loop_{AB}\ ss\ os \ \Box\ next\ os \\
loop_{AB}\ []\ os & \leftarrow & False \\
loop_{AB}\ (s : ss)\ os & \leftarrow & A = s\ \rightarrow\ loop_{B}\ ss\ os \ \Box\ next\ os \\
loop_{B}\ []\ os & \leftarrow & False \\
loop_{B}\ (s : ss)\ os & \leftarrow & B = s\ \rightarrow\ True \ \Box\ next\ os \\
next\ [] & \leftarrow & False \\
next\ (s : ss) & \leftarrow & loop_{AAB}\ ss\ ss
\end{array}$$

This program is only improved in the sense that the p argument has been removed.[7] But each time a match fails, the head of the string is skipped, and the match starts all over again.

4.2 Pattern matching with the positive supercompiler

We can draw a graph of terms that $\mathcal{W}$ encounters when applied to $matchAAB\,u$, see Figure 2.

The nodes labelled (A)-(C) in the right column signify that the transformation terminates since the next term has previously been encountered (they signify arcs back to the nodes labelled (1)-(3), respectively, in the left column).

Notice how the instantiation of *all* occurrences of u and ss allows information to be passed to $next$ above (1), (2), (3); this effect was not achieved by $\mathcal{S}$ due to the fact that it instantiates only one occurrence of u. These calls to $next$ can then be unfolded and some of the subsequent comparisons can be calculated. Specifically, above (C) it is known that we have a string $(A : A : s : ss)$, where s was not B. In moving one step to the right in the string we thus already know that the first comparison between the head of the string and A will succeed; this is in fact what is calculated above (C). The program generated is:

Example 3 Almost KMP style matcher.

$$\begin{array}{lll}
 & & loop_{AAB}\ u \\
loop_{AAB}\ [] & \leftarrow & False \\
loop_{AAB}\ (s : ss) & \leftarrow & A = s\ \rightarrow\ loop_{AB}\ ss \ \Box\ next_{AAB}\ ss\ s \\
loop_{AB}\ [] & \leftarrow & False \\
loop_{AB}\ (s : ss) & \leftarrow & A = s\ \rightarrow\ loop_{B}\ ss \ \Box\ next_{AB}\ ss\ s \\
loop_{B}\ [] & \leftarrow & False \\
loop_{B}\ (s : ss) & \leftarrow & B = s\ \rightarrow\ True \ \Box\ A = s\ \rightarrow\ loop_{B}\ ss \ \Box\ next_{AB}\ ss\ s \\
next_{AAB}\ ss\ s & \leftarrow & loop_{AAB}\ ss \\
next_{AB}\ ss\ s & \leftarrow & A = s\ \rightarrow\ loop_{AB}\ ss \ \Box\ next_{AAB}\ ss\ s
\end{array}$$

[7] Incidentally, this shows that deforestation can partially evaluate.

Fig. 2. $\mathcal{W}$ applied to $\mathit{match}\ AAB\ u$.

$$\begin{tikzcd}
\mathit{match}\ AAB\ u \arrow[d] & \\
(1)\mathit{loop}\ AAB\ u\ AAB\ u \arrow[r] \arrow[d] & \mathit{False} \\
(\mathbf{2})\begin{array}{l} A = s\ \rightarrow\ \mathit{loop}\ AB\ ss\ AAB\ (s : ss) \\ \qquad \square\ \mathit{next}\ AAB\ (s : ss) \end{array} \arrow[r] \arrow[d] & \mathit{next}\ AAB\ (s : ss)\ (\mathbf{A}) \\
\mathit{loop}\ AB\ ss\ AAB\ A : ss \arrow[r] \arrow[d] & \mathit{False} \\
(\mathbf{3})\begin{array}{l} A = s\ \rightarrow\ \mathit{loop}\ B\ ss\ AAB\ (A : s : ss) \\ \qquad \square\ \mathit{next}\ AAB\ (A : s : ss) \end{array} \arrow[r] \arrow[dd] & \mathit{next}\ AAB\ (A : s : ss) \arrow[d] \\
 & \mathit{loop}\ AAB\ (s : ss)\ AAB\ (s : ss)\ (\mathbf{B}) \\
\mathit{loop}\ B\ ss\ AAB\ A : A : ss \arrow[r] \arrow[d] & \mathit{False} \\
\begin{array}{l} B = s\ \rightarrow\ \mathit{loop}\ []\ ss\ AAB\ (A : A : s : ss) \\ \qquad \square\ \mathit{next}\ AAB\ (A : A : s : ss) \end{array} \arrow[r] \arrow[dd] & \mathit{next}\ AAB\ (A : A : s : ss) \arrow[d] \\
 & \mathit{loop}\ AAB\ (A : s : ss)\ AAB\ (A : s : ss) \arrow[d] \\
\mathit{loop}\ []\ ss\ AAB\ A : A : B : ss \arrow[dd] & \begin{array}{l} A = A\ \rightarrow\ \mathit{loop}\ AB\ (s : ss)\ AAB\ (A : s : ss) \\ \qquad \square\ \mathit{next}\ AAB\ (A : s : ss) \end{array} \arrow[d] \\
 & \mathit{loop}\ AB\ (s : ss)\ AAB\ (A : s : ss)\ (\mathbf{C}) \\
\mathit{True} &
\end{tikzcd}$$

Disregarding the test $A = s$ in next_{AB} which is known to be false when next_{AB} is called from false-branches of the the same test in loop_{AB} and loop_{B}, this is the desired KMP pattern matcher. The redundant tests do not affect the asymptotic behaviour of the generated specialized matchers: in the terminology of section 2.2, the first author has proved that $\mathcal{W}$ passes the KMP test [Sor94b]. This is a major improvement of $\mathcal{W}$ over $\mathcal{S}$.

The reason why the unnecessary tests are not cut off in the graph above is that we are only propagating *positive* information. In clause (8) of $\mathcal{W}$ we propagate information that a test was true to the true branch, that certain variables have certain values. We propagate the information by applying the unifier to the term in the true branch.

However, in the false branch we do not propagate the *negative* information that the test failed. Such information restricts the values which variables can take. Above nodes (A),(B),(C) in the graph we know that s can not be A, A and B, respectively, but this information is ignored by $\mathcal{W}$, resulting in the redundant tests in the program. Both positive and negative information can arise from an equality test, but we only propagate the positive information.

The reason why we do not propagate negative information in $\mathcal{W}$ is that it cannot be modeled using substitution (what instantiation should one make to express the fact that v is not equal to w?) We could incorporate negation into $\mathcal{W}$ using other techniques (see the next section) and thereby obtain exactly the KMP pattern matchers by $\mathcal{W}$.

There is also another kind of information. A call $g\,v\,t_1 \ldots t_n$ to a function g defined by patterns can be viewed as a test on v. When we instantiate in clause (5) of $\mathcal{W}$ one might say that we test what v is and propagate the resulting information to each of the branches. Here we also represent our positive information by application of a substitution. There is no notion of negative information arising from such a test. Negative information occurs only in the case of (implicit or explicit) "else" or "otherwise" constructs.

5 Partial evaluation, supercompilation, and GPC

In this section we relate information propagation as used in positive supercompilation and deforestation to that in partial evaluation, supercompilation, and GPC.

5.1 Partial Evaluation of Functional Programs

Partial evaluation as in [Jon93] propagates only simple information, *viz.* the values of static variables. So partial evaluation can specialize programs but is weaker than positive supercompilation, supercompilation or GPC, since it propagates no information obtained from predicates or pattern matching. This explains the result found in [Con89], that partial evaluation does not pass the KMP test on the tail-recursive matcher.

Binding-time improvements. The traditional way to improve the result of partial evaluation is to modify the source programs. These modifications, called *binding-time improvements*, are semantics-preserving transformations of the source program which enable a partial evaluator to propagate more information and to achieve deeper specialization.

Consel and Danvy showed that partial evaluators can be used to derive specialized KMP matchers by an "insightful rewriting" of the tail-recursive matcher [Con89] to improve its binding time separation. In short, when the rewritten matcher encounters a mismatch after k successful comparisons, it starts all over comparing the pattern with the $k-1$ long tail of the pattern, and only after $k-1$ successful comparisons will it return to the string. In other words, information propagation has been added to the matcher. The same rewriting suffices for deforestation to produce KMP style matchers.

Interpretive approach. It was shown by Glück and Jørgensen that partial evaluators can pass the KMP test by specializing an information propagating interpreter with respect to the tail-recursive matcher and a fixed pattern [Glu94].

5.2 Supercompilation

As mentioned, the mechanism ensuring the propagation of information in supercompilation is *driving*. Here we shall be concerned with driving as described in [Glu93] for a language with lists as data structures.

Let us, for a moment, think of $\mathcal{W}$ as the generalization of a *rewrite* interpreter: when it unfolds a function call it replaces the call by the body of the called function and substitutes the actual arguments into the term being interpreted. Alternatively, one can think of an *environment based* interpreter which creates bindings of the formal parameters to the actual arguments. Correspondingly, one could imagine an environment based version of $\mathcal{W}$. This is basically what the supercompiler in [Glu93] is.

Thus, the driving mechanisms in the positive supercompiler and in the supercompiler of [Glu93] are identical with respect to the propagation of positive information (assertions) about unspecified entities, except that the former uses substitution and the latter environments. The technique using environments has the advantage that negative information (restrictions) can be represented as bindings which do not hold, and this is done in [Glu93]. A technique using substitutions does not seem possible.

There does not seem to be any significant difference between using environments or substitution for positive information.[8] If one applies the supercompiler in [Glu93] using only positive information propagation to the tail-recursive matcher, one gets the same program that $\mathcal{W}$ produces; applying the full driving mechanism of [Glu93] with both positive and negative information propagation yields the desired optimal program as shown in [Glu93].

In [Glu90] Glück and Turchin showed that Turchin's supercompiler could pass the KMP test with the nested general matcher.

5.3 GPC

GPC extends partial evaluation as follows. Whenever a conditional (or something equivalent) testing whether predicate P holds is encountered during the transformation, P is propagated to the true branch and the predicate $\neg P$ is propagated to the false branch. Also, whenever a test is encountered, a theorem prover sitting on top of the transformer tests whether more than one branch is possible. If only one is possible, only that branch is taken. GPC is a powerful transformation method because it assumes the (unlimited) power of a theorem

[8] In self-application of partial evaluation one does binding-time analysis of the partial evaluator; such an analysis gives better results for the environment-based version because it gives better separation of static and dynamic data.

prover. It was shown in [Fut88] that this information suffices to pass the KMP test on the tail-recursive matcher.

Supercompilation and GPC are related, but differ in the propagation of information. While the latter propagates arbitrary predicates requiring a theorem prover, supercompilation propagates structural predicates (assertions and restrictions about atoms and constructors).

Takano concretized GPC for a particular functional language, *viz.* the same as the one studied in the original deforestation paper [Wad88].

There is one rule which is of particular interest for our purposes.

$$\begin{array}{l} G[\![\, \textbf{case}\, v\, \textbf{of}\, p_1 : t_1 \mid \ldots \mid p_n : t_n \,]\!]E \Rightarrow \\ \qquad \textbf{case}\, v\, \textbf{of}\, p_1 : G[\![\, t_1 \,]\!]E_1 \mid \ldots \mid p_n : G[\![\, t_n \,]\!]E_n \\ \textbf{where} \\ E_i = E \cup \{v \leftrightarrow p_i\} \end{array}$$

The E's are sets of equalities (sets of predicates) which are used in the manner described in more general terms in [Fut88]; concretely, they represent positive information arising from pattern matching. The algorithm actually uses a mixture of substitution based and environment representation of information. There is no need for negative information because the language of [Tak91] has no else-construct (just like g-functions in our language have no otherwise clause).

A related substitution based version is:

$$\begin{array}{l} G[\![\, \textbf{case}\, v\, \textbf{of}\, p_1 : t_1 \mid \ldots \mid p_n : t_n \,]\!]E \Rightarrow \\ \qquad \textbf{case}\, v\, \textbf{of}\, p_1 : G[\![\, t_1\{v := p_1\} \,]\!]E \mid \ldots \mid p_n : G[\![\, t_n\{v := p_n\} \,]\!]E \end{array}$$

The corresponding rule in deforestation is:

$$\mathcal{S}[\![\, \textbf{case}\, v\, \textbf{of}\, p_1 : t_1 \mid \ldots \mid p_n : t_n \,]\!] \Rightarrow \textbf{case}\, v\, \textbf{of}\, p_1 : \mathcal{S}[\![\, t_1 \,]\!] \mid \ldots \mid p_n : \mathcal{S}[\![\, t_n \,]\!]$$

Modulo syntax, the the step from the latter rule to to the former rule is exactly the same as the step from $\mathcal{S}$ to $\mathcal{W}$. It is exactly this step which allows the derivation of KMP style pattern matchers, as mentioned briefly in the context of the language with case constructs in [Con93].

6 Conclusion and future work

We compared the transformation methodologies deforestation, partial evaluation, supercompilation, GPC, and positive supercompilation, the latter being new. We showed which notions of information propagation they share, what their differences are, and that the amount of information propagated is significant for the transformations achieved by each methodology.

We demonstrated how the positive supercompiler, using only positive information propagation, can derive an algorithm comparable in efficiency to the matcher generated by the Knuth-Morris-Pratt algorithm starting from a general string matcher and a fixed pattern. Deforestation and partial evaluation cannot achieve this.

Our results are strong evidence that one should not restrict the application of techniques developed in one field to a particular methodology. On the contrary, their integration is on the agenda. However, a direct comparison is often blurred because of different notations and perspectives. Future work may bring the different methodologies even closer, as outlined below.

Until now we have grouped the transformers according to the amount of information propagation. Another classification criterion is the *handling of nested calls.* Deforestation, positive supercompilation, and Turchin's supercompiler all simulate call-by-name evaluation, whereas partial evaluators simulate call-by-value. It would seem that the strength of transformers depend on the transformers "evaluation strategy." For instance, it is well-known that plain partial evaluation does not eliminate intermediate data strucures, whereas all the above call-by-name transformers do. On the other hand all, the above call-by-name transformers, including deforestation, can perform partial evaluation. This idea also seems worthy of an investigation. Some steps have been taken in [Sor94b], but further clarification is needed. Related research includes the idea of deforestation by CPS-translation and call-by-value partial evaluation.

The second author has on several occasions noted the correspondence between supercompilation and *interpretation* of logic programs. The correspondence has been stated quite precisely in terms of SLD-trees and so-called process trees in [Sor94b]. Possible payoffs from such an idea include the application of a variety of techniques from one community in the other; the idea certainly seems worthy of study.

There are also connections between supercompilation and *transformation*, in particular partial evaluation, of logic programs. Unlike the situation in the functional case, partial evaluators for Prolog can derive KMP matchers from general Prolog matchers similar to our tail-recursive matcher [Smi91]. This is because partial evaluators for Prolog propagate information (by unification) in a way similar to that in supercompilation. Possible gains from a detailed correspondence may, again, be significant.

References

[Aug85] L. Augustsson. Compiling Lazy Pattern-Matching. In *Conference on Functional Programming and Computer Architecture.* (Ed.) J.-P. Jouannaud pp368-381. LNCS 201, Springer-Verlag 1985.

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation.* Ph.D. thesis, DIKU-Rapport 90/17, Department of Computer Science, University of Copenhagen, 1990.

[Bur77] R. M. Burstall & J. Darlington. A Transformation System for Developing Recursive Programs. In *Journal of the ACM.* Vol.24, No.1, pp.44-67, 1977.

[Chi93] W.-N. Chin. Safe Fusion of Functional Expressions II: Further Improvements. In *Journal of Functional Programming.* 1994, *to appear.*

[Con89] C. Consel & O. Danvy. Partial Evaluation of Pattern Matching in Strings. In *Information Processing Letters.* Vol.30, No.2, pp.79-86, 1989.

[Con93] C. Consel & O. Danvy. Tutorial Notes on Partial Evaluation. In *20th ACM Symposium on Principles of Programming Languages.* Charleston, South Carolina, pp.493-501, ACM Press 1993.

[Fer88] A. B. Ferguson & P. Wadler. When will Deforestation Stop? *1988 Glasgow Workshop on Functional Programming.* pp.39-56, 1988.

[Fut88] Y. Futamura & K. Nogi. Generalized Partial Computation. In *Partial Evaluation and Mixed Computation.* Eds. A. P. Ershov, D. Bjørner & N. D. Jones, pp.133-151, North-Holland 1988.

[Glu90] R. Glück & V. F. Turchin. Application of Metasystem Transition to Function Inversion and Transformation. In *Proceedings of the ISSAC '90.* pp.286-287, ACM Press 1990.

[Glu93] R. Glück & And. Klimov.. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In *Static Analysis, Proceedings. LNCS 724.* pp.112-123, Springer-Verlag 1993.

[Glu94] R. Glück & J. Jørgensen. Generating Optimizing Specializers. In *IEEE International Conference on Computer Languages.* IEEE Computer Science Press, 1994, to appear.

[Jon93] N. D. Jones, C. Gomard & P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International. 1993

[Kle52] S. Kleene, Introduction to Metamathematics. Van Nostrand, 1952.

[Knu77] D. E. Knuth, J. H. Morris, V. R. Pratt. Fast Pattern Matching in Strings. In *SIAM Journal on Computing.* Vol.6, No.2, pp.323-350, 1977.

[Ses88] P. Sestoft. Automatic Call Unfolding in a Partial Evaluator. In *Partial Evaluation and Mixed Computation.* Eds. A.P Ershov, D. Bjørner, N.D. Jones, pp.485-506, North-Holland 1988.

[Smi91] D. A. Smith. Partial Evaluation of Pattern Matching in Constraint Logic Programming Languages. In *ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation.* Ed. N. D. Jones & P. Hudak, pp.62-71, ACM Press 1991.

[Sor93] M. H. Sørensen. A New Means of Ensuring Termination of Deforestation with an Application to Logic Programming. In *Workshop of the Global Compilation Workshop in conjunction with the International Logic Programming Symposium.* Vancouver, Canada, October, 1993.

[Sor94a] M. H. Sørensen. A Grammar-based Data-flow Analysis to Stop Deforestation. In *Colloqium on Algebra in Trees and Programming.* Edinburgh, Scotland, April 1994, to appear.

[Sor94b] M. H. Sørensen. *Turchin's Supercompiler Revisited. An Operational Theory of Positive Information Propagation.* Master's Thesis, Department of Computer Science, University of Copenhagen, 1994.

[Tak91] A. Takano. Generalized Partial Computation for a Lazy Functional Language. In *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* Ed. N. D. Jones & P. Hudak, pp.1-11, ACM Press 1991.

[Tur86] V. F. Turchin. The Concept of a Supercompiler. In *ACM TOPLAS.* Vol.8, No.3, pp.292-325, 1986.

[Tur90] D. Turner. An Overview of Miranda. In *Research Topics in Functional Programming.* Ed. D. Turner, Addison-Wesley, 1990.

[Wad88] P. L. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *European Symposium on Programming. Proceedings. LNCS 300.* pp.344-358, Springer-Verlag 1988.

This article was processed using the LaTeX macro package with LLNCS style