J. Functional Programming 6 (6): 811–838, November 1996

A positive supercompiler

M. H. SØRENSEN, R. GLÜCK and N. D. JONES

DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
(e-mail: {rambo,glueck,neil}@diku.dk)

Abstract

We introduce a *positive supercompiler*, a version of Turchin's supercompiler maintaining only positive information during transformation, and using folding without generalization. The positive supercompiler can also be regarded as a variant of Wadler's deforestation maintaining an increased amount of information. We compare our algorithm to deforestation and, in less detail, to partial evaluation, Turchin's supercompiler, Generalized Partial Computation (GPC), and partial deduction by classifying these transformers by the amount of information they maintain during transformation. This factor is significant, as a differentiating example reveals: positive supercompilation, Turchin's supercompiler, GPC and partial deduction can specialize a general pattern matcher with respect to a fixed pattern to obtain an efficient matcher very similar to the Knuth–Morris–Pratt algorithm. Deforestation and traditional partial evaluation achieve this effect only after a non-trivial hand rewriting of the general matcher.

Capsule Review

Supercompilation is a technique of function transformation developed originally by Turchin (1986) for a functional language Refal. The technique uses DRIVING (the forced unfolding guided by functional configurations) to construct a potentially infinite tree of states (configurations) and transitions. This tree is then converted into a self-sufficient graph by reducing some of the configurations to their predecessors, which is similar to folding, or creating new configurations by generalization and redoing the driving. Supercompilation has been shown to do partial evaluation and deforestation, as well as more difficult transformations.

Sørensen, Glück and Jones present and discuss the essential features of supercompilation, using a simplified version of the supercompiler which they call POSITIVE. The reason for this name is that the information which is propagated through the graph during driving includes only POSITIVE elements, i.e. statements that a variable matches a certain pattern, while the NEGATIVE statements that a variable does not match a pattern (which are propagated in Turchin's supercompiler as RESTRICTIONS) are not taken along. Generalization is also not done in the positive supercompiler.

These simplifications allow the authors to present supercompilation in more traditional formalism and in more detail, as well as compare it with similar techniques: partial evaluation, deforestation and generalized partial computation. The authors consider in detail one well-known example of program transformation: the transformation of a naive string matcher by specializing it for a given specific pattern into an efficient algorithm known as KMP (it became a kind of standard test in partial evaluation and similar techniques). They show that while supercompilation (both full and positive) and generalized partial computation solve the problem, 'regular' partial evaluation and deforestation cannot do this without a non-trivial modification in the original unspecialized matcher.

1 Introduction

We are concerned with certain automatic instances of Burstall and Darlington's (1977) framework. The techniques of the framework are: *unfolding, instantiation, definition, folding* and *abstraction.* While applying these mechanisms the transformers we consider maintain information about the terms previously encountered.

Partial evaluation, discussed at length in Jones *et al.* (1993), performs *program specialization.* For instance, a call $f\ 2\ v$ is replaced by $f_2\ v$ 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, the decision when to unfold, instantiate, define and fold is guided by program annotations generated before transformation; in *online* versions such decisions are taken during transformation.

Deforestation, due to Wadler (1990), performs *program composition* by eliminating intermediate data structures, thus reducing the number of passes over data. For instance, a call $f_a\ (f_b\ xs)$, where f_a and f_b are functions that remove all a's and b's in a list, respectively, is replaced by a call $f_{ab}\ xs$ to a function f_{ab} that removes all a's and b's in one pass.* To ensure that deforestation terminates and that the resulting program is no less efficient than the original, Wadler restricts application of deforestation to compositions of functions with *treeless* definitions, a syntactic restriction on the terms defining functions. However, we will see that deforestation has valuable applications well beyond this original, intended scope by applying it to arbitrary definitions – see sections 5.5 and 6.2.

Supercompilation, conceived by Turchin (1979, 1980, 1986) in the early 1970's in Russia for the language Refal, achieves the effects of both deforestation and partial evaluation, as well as some more dramatic optimizations. This is done by *driving,* i.e. unfolding and propagation of information, and *generalization* (Turchin, 1988), a form of abstraction which enables folding. The decision when to generalize is taken online.

Generalized partial computation (GPC), due to Futamura (1988), has similar effects and power as supercompilation, but requires the use of a theorem prover.

The above methodologies have been developed for functional languages. Similar methodologies are also being studied for other language paradigms, e.g. *partial deduction* in logic programming (Lloyd and Shepherdson, 1991; Komorowski, 1992).

In this paper we present a *positive supercompiler* comprising a driving and a folding component. The driving component can be viewed as a reformulation of a certain fragment of driving as defined in Turchin's supercompiler. For the second component we fold only calls that are identical up to renaming, the same strategy as in *deforestation.*†

The *formulation* of our two components is inspired by Wadler's formulation of deforestation (Wadler, 1990). This choice makes it easy to explain the essence of driving and its relation to techniques used in other transformers, e.g. deforestation.

* Earlier work by Turchin *et al.* (1982), Feather (1982), Bird (1984), Wadler (1984) and others also addressed elimination of intermediate data structures.

† Recent work (Sørensen and Glück, 1995) develops generalization for the positive supercompiler; this is beyond the scope of this paper.

q	::=	$d_1 \ldots d_m$	(program)
d	::=	$f\, v_1 \ldots v_n = t$	(definition)
t	::=	v	(variable)
	|	$c\, t_1 \ldots t_n$	(constructor)
	|	$f\, t_1 \ldots t_n$	(function call)
	|	**case** t_0 **of** $p_1 \rightarrow t_1; \ldots; p_m \rightarrow t_m$	(case-expression)
	|	**if** $t_1 = t_2$ **then** t_3 **else** t_4	(conditional)
p	::=	$c\, v_1 \ldots v_n$	(flat pattern)

Fig. 1. Syntax.

The paper is based on Sørensen, Glück and Jones (1994) and Sørensen (1994a). It belongs to a line of work which aims at a better understanding of supercompilation and its relation to other program transformers (Glück and Klimov, 1993; Jones, 1994; Glück and Jørgensen, 1994; Glück and Sørensen, 1994; Nielsen and Sørensen, 1995).

The remainder of the paper is organized as follows. In section 2 the object language is presented. In section 3 we introduce the *KMP test* which allows us to assess the amount of information that a program transformer maintains. In section 4 we present some preliminaries which are used in section 5 to present the positive supercompiler. Using the KMP test, we compare the information propagation in positive supercompilation and deforestation in section 6. In section 7 we extend the comparison to partial evaluation, GPC, Turchin's supercompiler and partial deduction. Finally, in section 8 we give some examples illustrating more capabilities of positive supercompilation.

2 Object language

We are concerned with a first-order functional language; the exact syntax is presented below. The intended operational semantics is normal-order graph reduction to weak head normal form in the sense of Bird and Wadler (1988). This could be formalized by techniques similar to Launchbury (1993) or Ariola *et al.* (1995), but we shall not have any need to do so.

Definition 1

A program q is a sequence of function definitions d where the body of each definition is a term t constructed from variables, constructors, function calls, case-expressions, and conditionals – see figure 1 (where $m > 0, n \geq 0$).

To ensure uniqueness of reduction, we require that each function in a program have at most one definition and that no two patterns p_i and p_j in a case-expression contain the same constructor. As usual we require that patterns in case-expressions

be *linear*, i.e. no variable occurs more than once in a pattern p_i. We also require that all variables in the right side of a definition be present in its left side.*

We have made a number of syntactic choices. First, we use *case-expressions* rather than functions defined by patterns, following Wadler (1990). The advantage of case-expression is that the self-contained syntax is easier to analyse. Moreover, the difference between deforestation and positive supercompilation is easiest to explain for explicit case-expressions.

Second, case-expressions may only have *non-nested patterns*. This restriction is quite common, and semantics-preserving methods exist for translating arbitrary patterns into the restricted form (Augustsson, 1985; Wadler, 1987).

Third, we have adopted an explicit *equality construct*. For a data type with a *finite* set of constructors, equality can be programmed by means of case-expressions, i.e. by listing all possible cases. We avoid this approach because examples become longer. While the difference between positive supercompilation and deforestation manifests itself even without the equality construct, the difference between positive supercompilation and Turchin's supercompilation is less significant when the equality construct is absent.

Finally, as a matter of simplicity, our language is *first-order*. Several formulations of higher-order deforestation exist. For positive supercompilation, certain new problems arise in the extension to the higher-order case (see section 5.2), and these are beyond the scope of this paper.

3 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 see whether the method generates, from a general pattern matcher and a fixed pattern, a specialized pattern matcher of efficiency similar to the one generated by the Knuth–Morris–Pratt algorithm (Knuth, Morris and Pratt, 1977). We call this *the KMP test.*

Subsection 3.1 introduces general and specialized pattern matchers, and subsection 3.2 discusses the time complexity of these matchers.

3.1 General, naively specialized and KMP specialized matchers

Figure 2 shows a *general matcher*. It takes a pattern and a string and returns *True* iff the pattern occurs as a substring in the string.

Now consider the *naively specialized matcher* in figure 3 which matches the fixed pattern *AAB* with a string *u* by calling *match*. Evaluation 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*.

* We use the shorthand notation $(x : xs)$ and [] for the constructors *Cons x xs* and *Nil*, respectively. For $x_1 : x_2 : \ldots : x_n : []$ we write $[x_1, x_2, \ldots, x_n]$, or sometimes $x_1 x_2 \ldots x_n$.

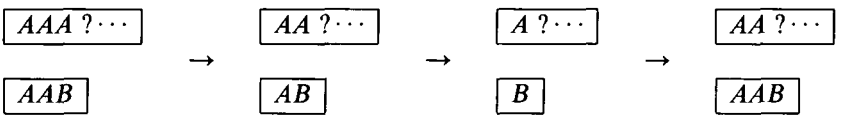
```
match p s          =  loop p s p s
loop pp ss op os   =  case pp of
                         []      →  True
                         (p : pp) →  case ss of
                                        []      →  False
                                        (s : ss) →  if p = s
                                                    then loop pp ss op os
                                                    else next op os
next op os         =  case os of
                         []      →  False
                         (o : os) →  loop op os op os
```

Fig. 2. General matcher.

$$match_{AAB}\ u \quad = \quad match\ AAB\ u$$

Fig. 3. Naively specialized matcher.

This strategy is not optimal. Suppose, for instance, that the string u begins with three A's. Then the steps of the naively specialized matcher can be depicted as follows:



After matching the two A's in the pattern with the first two A's in the string, the B in the pattern fails to match the A in the string. Then the process is restarted with the string's tail, even though it is known that the first two comparisons will succeed. Rather than performing these tests whose outcome is already known, we should *skip* the three first A's in the original string, i.e. proceed directly to compare the B in the pattern with the fourth element of the original string. This is done in the *KMP specialized matcher** in figure 4.

After finding two A's and a third symbol which is not a B in the string, this program checks (in $loop_B$) whether the third symbol of the string is an A. If so, it continues immediately by comparing the next symbol of the string with the B in the pattern (by calling $loop_B$), thereby avoiding repeated comparisons.

* Knuth, Morris and Pratt (1977) give a matching algorithm running in time $O(|p| + |s|)$, where p is the pattern and s the string. Their technique first computes in time $O(|p|)$ a *next-table* showing how much of the string can be skipped in the case of a mismatch. The string is then read from left to right in worst-case time $O(|s|)$ using the next-table. Our KMP specialized matcher implicitly represents the next-table in its call structure. At the end of their paper, Knuth *et al.* consider variations where the string is scanned from right to left, thereby arriving at another linear matching algorithm with *average-case* time complexity $O((|s| \cdot log|p|)/|p|)$. These variations are not relevant to this paper.

```
match_AAB u    =  loop_AAB u
loop_AAB ss    =  case ss of
                        []    →  False
                      (s' : ss') →  if A = s'
                                    then loop_AB ss'
                                    else loop_AAB ss'
loop_AB ss     =  case ss of
                        []    →  False
                      (s' : ss') →  if A = s'
                                    then loop_B ss'
                                    else loop_AAB ss'
loop_B ss      =  case ss of
                        []    →  False
                      (s' : ss') →  if B = s'
                                    then True
                                    else if A = s'
                                         then loop_B ss'
                                         else loop_AAB ss'
```

Fig. 4. KMP specialized matcher.

3.2 *Complexity of the matchers*

Let for any function f and values $d_1,\ldots,d_n$ the number of normal-order graph reduction steps required to print the whole value of $f\ d_1 \ldots d_n$ be denoted by $t_f(d_1,\ldots,d_n)$. Let M be the general matcher, N_p be the naively specialized matcher for pattern p, and K_p be the KMP specialized matcher for pattern p.

It is obvious that $t_M(p,s)$ is $O(|p| \cdot |s|)$. This implies that for every p, both $t_{K_p}(s)$ and $t_{N_p}(s)$ are $O(|s|)$. Spelled out:

$$\forall p\ \exists c > 0\ \forall s: \quad t_{N_p}(s) \leq c \cdot |s| \quad \text{and} \quad t_{K_p}(s) \leq c \cdot |s| \tag{1}$$

This property is a consequence of the mere fact that p is fixed in both N_p and K_p.

In terms of time complexity there is thus no difference between the naively specialized and KMP specialized matchers. However, if we count the actual number of comparisons made by N_p and K_p, the former number is larger. More importantly, the former number, but not the latter, becomes larger with larger p. That is, the following strengthening of (1) holds for K_p but not for N_p:

$$\exists c > 0\ \forall p\ \forall s: \quad t_{K_p}(s) \leq c \cdot |s| \tag{2}$$

This interchange of quantifiers is the crucial difference between N_p and K_p.

Whenever a program transformer takes as input the naively specialized matcher N_p and returns as output a specialized matcher f_p satisfying (2), like K_p, we shall say that the transformer *passes the KMP test.* In section 6 we investigate whether deforestation and positive supercompilation pass the KMP test. In Section 7 we review and explain the known results about the KMP test for partial evaluation, Turchin's supercompilation, GPC, and partial deduction, and relate all these results.

We end this section by stressing that the complexity considerations in this paper

regard normal-order graph reduction. In terms of observable weak head-normal forms, normal-order reduction and normal-order graph reduction are indistinguishable. The only difference is in efficiency: in normal-order reduction a computation may be duplicated and therefore executed several times.

4 A normal-order reduction semantics

For the purpose of explaining deforestation and positive supercompilation as different generalizations of an operational semantics, we now present a plain normal-order reduction semantics for our language. However, we stress that the intended operational semantics of our language is normal-order *graph* reduction, and that the present normal-order reduction semantics will not be used for the complexity considerations in this paper.

Subsection 4.1 describes some notions that are useful for expressing which part of a terms should be reduced, and subsection 4.2 uses these notions to state the normal-order reduction semantics.

4.1 Redex and evaluation context

A term with no free variables is called *closed*. For every closed term t two possibilities exist with respect to normal-order reduction:

(i) $t \equiv c\ t_1 \ldots t_n$, and then interpretation proceeds to the arguments* $t_1 \ldots t_n$;
(ii) $t \not\equiv c\ t_1 \ldots t_n$, and then the leftmost outermost evaluation of term t forces a unique call to be unfolded, or a unique conditional or case-expression to be reduced to one of its branches.

For instance, evaluation of

$$\textbf{case}\ (f\ t)\ \textbf{of}\ \ [] \to t';\ \ (x : xs) \to t''$$

forces the call $f\ t$ to be unfolded to decide which branch of the case-expression to choose. Thus in case (ii) the term will be decomposed into:

(1) a redex r – the next function call to unfold, or conditional or case-expression to reduce to a branch; and
(2) an evaluation context e – the part of t surrounding r.

Above, $r = f\ t$ and $e = \textbf{case}\ \diamond\ \textbf{of}\ \ [] \to t';\ \ (x : xs) \to t''$. We write $t \equiv e\langle r \rangle$ to denote the result of replacing $\diamond$ in e by r, so in case (ii) the term t can be written uniquely in the form $e\langle r \rangle$. We now define these notions more precisely.

Definition 2
Let b, o, r, e range over *values*, *observables*, *redexes* and *evaluation contexts* as defined by the grammar in figure 5.

* Based on the assumption that the user demands that the whole term's value be printed.

b	::=	$c\ b_1 \ldots b_n$	(value)
o	::=	$c\ t_1 \ldots t_n$	(observable)
r	::=	$f\ t_1 \ldots t_n$	(redex)
	|	**case** o **of** $p_1 \to t_1; \ldots; p_m \to t_m$	
	|	**if** $b_1 = b_2$ **then** t **else** t'	
e	::=	$\diamond$	(evaluation context)
	|	**case** e **of** $p_1 \to t_1; \ldots; p_m \to t_m$	
	|	**if** $e' = t_2$ **then** t_3 **else** t_4	
	|	**if** $b = e'$ **then** t_3 **else** t_4	
e'	::=	$e \mid c\ b_1 \ldots b_{i-1}\ e'\ t_{i+1} \ldots t_n$	

Fig. 5. Value, observable, redex and evaluation context.

A *value* b is a term without redexes. An *observable* o is a term with a known outermost constructor. A *redex* r is a term in which the outermost construction (function call, conditional or case-expression) can be reduced without reducing subterms first. An *evaluation context* e is a term with a 'hole' $\diamond$.

The rules for the evaluation context e state where one is allowed to reduce a redex: (i) under a case-expression (so that the tested term can produce an outermost constructor); (ii) in the leftmost tested term in a conditional (so that the tested term can reduce to a value); (iii) in the rightmost tested term in a conditional, provided the leftmost term has been reduced to a value.

Note that the rule for e' ensures that the tested terms in a conditional will be evaluated to values, not just observables. One could imagine a different reduction semantics which would allow choosing the false branch as soon as the two tested terms are known to have different outermost constructors. For instance, if $c_1 \neq c_2$, then the term

$$\textbf{if } c_1\ t_1 = c_2\ t_2 \textbf{ then } s_1 \textbf{ else } s_2$$

could be reduced to s_2 directly, even if reduction of t_1 never terminates. Whether one prefers this semantics is perhaps a matter of taste, but it would complicate the formulation of the reduction semantics.

It is easy to verify that any closed term t is either an *observable* o, or it decomposes uniquely into the form $t \equiv e\langle r \rangle$ (*the unique decomposition property*). This property ensures that the clauses of the reduction semantics below are mutually exclusive, and together exhaustive over all closed terms.

4.2 Normal-order reduction semantics

Definition 3

The reduction semantics is given in figure 6. The expression $t\{v_i := t_i\}_{i=1}^n$ denotes the result of simultaneously replacing all occurrences of v_i in t by the corresponding terms t_i. Term identity is indicated by $\equiv$.

(1) $\mathscr{S}[\![c\ t_1 \ldots t_n]\!]$	$= c\ (\mathscr{S}[\![t_1]\!]) \ldots (\mathscr{S}[\![t_n]\!])$
(2) $\mathscr{S}[\![e\langle f\ t_1 \ldots t_n \rangle]\!]$	$= \mathscr{S}[\![e\langle t\{v_i := t_i\}_{i=1}^n \rangle]\!]$ where $f\ v_1 \ldots v_n = t$
(3) $\mathscr{S}[\![e\langle \textbf{case}\ (c\ t_1 \ldots t_n)\ \textbf{of}\ p_1 \to s_1; \ldots; p_m \to s_m \rangle]\!]$	$= \mathscr{S}[\![e\langle s_j\{v_i := t_i\}_{i=1}^n \rangle]\!]$ where $p_j \equiv c\ v_1 \ldots v_n$
(4) $\mathscr{S}[\![e\langle \textbf{if}\ b = b'\ \textbf{then}\ t\ \textbf{else}\ t' \rangle]\!]$	$= \mathscr{S}[\![e\langle t \rangle]\!]$ if $b \equiv b'$
(5) $\mathscr{S}[\![e\langle \textbf{if}\ b = b'\ \textbf{then}\ t\ \textbf{else}\ t' \rangle]\!]$	$= \mathscr{S}[\![e\langle t' \rangle]\!]$ if $b \not\equiv b'$

Fig. 6. Normal-order reduction semantics.

5 Positive supercompilation

We shall express the positive supercompiler by rules for rewriting terms. The rules can be understood as mimicking the actions of the normal-order reduction semantics – but extended to continue the transformation whenever a value that is needed is not available at transformation time. For example, if a case-expression cannot be decided at transformation time then residual 'case' code will be generated to account for every branch at run-time.

After generalizing the notions of redex and evaluation context (section 5.1), we define *driving* (section 5.2), the core of the positive supercompiler, and add *folding* (section 5.3). Then we consider the correctness of the positive supercompiler (section 5.4), and compare positive supercompilation to deforestation (section 5.5).

5.1 Redex and evaluation context

Analogously to normal-order reduction of closed terms, two cases should exist for every general term t with respect to normal-order transformation:

(i) $t \equiv c\ t_1 \ldots t_n$, in which case transformation will proceed to the arguments; or $t \equiv v$, in which case transformation terminates;
(ii) $t \not\equiv c\ t_1 \ldots t_n$ and $t \not\equiv v$, then leftmost-outermost evaluation of t forces a unique call to be unfolded, or a unique conditional or case-expression to be reduced.

To describe transformation as opposed to reduction we thus need to extend the notions of 'observable', etc. Compared to notions for reduction, the only difference is that variables are added to the clauses for observables b and values o; this of course indirectly affects the definition of redexes and contexts. We use the same names 'value, observable, redex, context' for these extended notions.

Definition 4
Let b, o range over *values* and *observables* as re-defined by the grammar

$$\begin{array}{lcll} b & ::= & c\ b_1 \ldots b_n & \mid\ v \\ o & ::= & c\ t_1 \ldots t_n & \mid\ v \end{array}$$

(1) $\mathscr{P}[\![v]\!]$
$= \ulcorner v \urcorner$

(2) $\mathscr{P}[\![c\ t_1 \ldots t_n]\!]$
$= \ulcorner c\ (\mathscr{P}[\![t_1]\!]) \ldots (\mathscr{P}[\![t_n]\!]) \urcorner$

(3) $\mathscr{P}[\![e\langle f\ t_1 \ldots t_n \rangle]\!]$
$= \mathscr{P}[\![e\langle t\{v_i := t_i\}_{i=1}^n \rangle]\!]$
where $f\ v_1 \ldots v_n = t$

(4) $\mathscr{P}[\![e\langle \textbf{case}\ (c\ t_1 \ldots t_n)\ \textbf{of}\ p_1 \to s_1; \ldots ; p_m \to s_m \rangle]\!]$
$= \mathscr{P}[\![e\langle s_j\{v_i := t_i\}_{i=1}^n \rangle]\!]$
where $p_j \equiv c\ v_1 \ldots v_n$

(5) $\mathscr{P}[\![e\langle \textbf{case}\ v\ \textbf{of}\ p_1 \to s_1; \ldots ; p_m \to s_m \rangle]\!]$
$= \ulcorner \textbf{case}\ v\ \textbf{of}\ p_1 \to \mathscr{P}[\![(e\langle s_1 \rangle)\{v := p_1\}]\!]; \ldots ; p_m \to \mathscr{P}[\![(e\langle s_m \rangle)\{v := p_m\}]\!] \urcorner$

(6) $\mathscr{P}[\![e\langle \textbf{if}\ b = b'\ \textbf{then}\ t\ \textbf{else}\ t' \rangle]\!]$
$= \mathscr{P}[\![e\langle t \rangle]\!]$
if b, b' are closed and $b \equiv b'$

(7) $\mathscr{P}[\![e\langle \textbf{if}\ b = b'\ \textbf{then}\ t\ \textbf{else}\ t' \rangle]\!]$
$= \mathscr{P}[\![e\langle t' \rangle]\!]$
if b, b' are closed and $b \not\equiv b'$

(8) $\mathscr{P}[\![e\langle \textbf{if}\ b = b'\ \textbf{then}\ t\ \textbf{else}\ t' \rangle]\!]$
$= \ulcorner \textbf{if}\ b = b'\ \textbf{then}\ \mathscr{P}[\![(e\langle t \rangle) MGU(b, b')]\!]\ \textbf{else}\ \mathscr{P}[\![e\langle t' \rangle]\!] \urcorner$
if b, b' are not both closed

Fig. 7. Positive driving.

Any term t is either an observable o or it decomposes uniquely into the form $t \equiv e\langle r \rangle$. This implies that the clauses of driving, defined in the next subsection, are mutually exclusive and together exhaustive over all terms.

5.2 Positive driving

Definition 5

The transformation rules for driving are given in figure 7. Quine's 'quasi-quotes' $\ulcorner$ and $\urcorner$ are used to bracket code that will appear in the transformed program. Subexpressions, such as $\mathscr{P}[\![e\langle t' \rangle]\!]$ appearing inside $\ulcorner \urcorner$, are to be replaced by the code they generate (without quotes), e.g. in the conditional in clause (8).

The notation $MGU(b, b')$ denotes the most general unifier $\{v_i := t_i\}_{i=1}^n, (n \geq 0)$ of b, b' if it exists, and *fail* otherwise. It is convenient to define $t\,fail \equiv t$. The unifier must be chosen to be *idempotent*, which is always possible – see Sørensen (1994a).

To avoid name capture, in clause (3) the definition of f should always be renamed so that the variables in patterns in case-expressions are fresh.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathscr{P}[\![\ \mathit{append}\ (1 : xs)\ ys\ ]\!]} \\
= & \mathscr{P}[\![\ \textbf{case}\ (1 : xs)\ \textbf{of} & (3) \\
 & \qquad [\,] \rightarrow ys & \\
 & \qquad (x' : xs') \rightarrow x' : \mathit{append}\ xs'\ ys\ ]\!] & \\
= & \mathscr{P}[\![\ 1 : \mathit{append}\ xs\ ys\ ]\!] & (4) \\
= & \mathscr{P}[\![\ 1\ ]\!] : \underline{\mathscr{P}[\![\ \mathit{append}\ xs\ ys\ ]\!]} & (2) \\
= & 1 : \mathscr{P}[\![\ \textbf{case}\ xs\ \textbf{of} & (1,3) \\
 & \qquad [\,] \rightarrow ys & \\
 & \qquad (x' : xs') \rightarrow x' : \mathit{append}\ xs'\ ys\ ]\!] & \\
= & 1 : \textbf{case}\ xs\ \textbf{of} & (5) \\
 & \qquad [\,] \rightarrow \mathscr{P}[\![\ ys\ ]\!] & \\
 & \qquad (x' : xs') \rightarrow \mathscr{P}[\![\ x' : \mathit{append}\ xs'\ ys\ ]\!] & \\
= & 1 : \textbf{case}\ xs\ \textbf{of} & (1,2) \\
 & \qquad [\,] \rightarrow ys & \\
 & \qquad (x' : xs') \rightarrow \mathscr{P}[\![\ x'\ ]\!] : \underline{\mathscr{P}[\![\ \mathit{append}\ xs'\ ys\ ]\!]} & \\
= & \ldots &
\end{array}
$$

Fig. 8. Positive driving $\mathscr{P}[\![\ \mathit{append}\ (1 : xs)\ ys\ ]\!]$.

It is instructive to compare the transformation rules for driving with the rules for normal-order reduction. Clauses (1, 5, 8) for $\mathscr{P}$ have no counterpart in $\mathscr{I}$ since $\mathscr{P}$ deals with general terms whereas $\mathscr{I}$ deals with closed terms. In these cases $\mathscr{P}$ generates a residual expression to account for the different run-time possibilities. Note the substitutions $\{v := p_i\}$ and $MGU(b, b')$ in clauses (5) and (8). The assumed outcome of case and equality test is propagated to the terms on the right hand side.

Otherwise, the clauses (2, 3, 4, 6, 7) of $\mathscr{P}$ are identical to the clauses (1–5) of $\mathscr{I}$. An induction proves that $\mathscr{I}$ and $\mathscr{P}$ give the same result on closed terms.

Note the difficulty that would arise in rule (5) if our language was higher-order: instead of simply having a variable v in the test, we might have a stuck application $v\ v'$, in which case no direct instantiation can be made in the branches.

5.3 Folding

The driving algorithm hardly ever terminates. The reason is that rule (3) unfolds *all* function calls regardless of whether the same term has been encountered before or not. Figure 8 shows what happens when we apply $\mathscr{P}$ to the term *append* $(1 : xs)$ *ys*. Transformation of *append xs ys* leads to transformation of the same term, modulo variable renaming, and the transformation process continues *ad infinitum*.

This is not to say that there is a *problem* with the driving algorithm, but rather that it would not make sense to consider driving in isolation. Similarly, one does

$\mathscr{P}[\![\ append\ (1 : xs)\ ys\ ]\!]$

$= \mathscr{P}[\![\ \textbf{case}\ (1 : xs)\ \textbf{of}$ (3)
$\quad [] \rightarrow ys$
$\quad (x' : xs') \rightarrow x' : append\ xs'\ ys\ ]\!]$

$= \mathscr{P}[\![\ 1 : append\ xs\ ys\ ]\!]$ (4)

$= \mathscr{P}[\![\ 1\ ]\!] : \underline{\mathscr{P}[\![\ append\ xs\ ys\ ]\!]}$ (2)

$= 1 : h\ xs\ ys$ (1,definition)

$h\ xs\ ys$

$= \mathscr{P}[\![\ \textbf{case}\ xs\ \textbf{of}$ (3)
$\quad [] \rightarrow ys$
$\quad (x' : xs') \rightarrow x' : append\ xs'\ ys\ ]\!]$

$= \textbf{case}\ xs\ \textbf{of}$ (5)
$\quad [] \rightarrow \mathscr{P}[\![\ ys\ ]\!]$
$\quad (x' : xs') \rightarrow \mathscr{P}[\![\ x' : append\ xs'\ ys\ ]\!]$

$= \textbf{case}\ xs\ \textbf{of}$ (1,2)
$\quad [] \rightarrow ys$
$\quad (x' : xs') \rightarrow \mathscr{P}[\![\ x'\ ]\!] : \underline{\mathscr{P}[\![\ append\ xs'\ ys\ ]\!]}$

$= \textbf{case}\ xs\ \textbf{of}$ (1,fold)
$\quad [] \rightarrow ys$
$\quad (x' : xs') \rightarrow x' : h\ xs'\ ys$

Fig. 9. Positive driving $\mathscr{P}[\![\ append\ (1 : xs)\ ys\ ]\!]$ with folding.

not consider the transformation rules for deforestation presented in Wadler (1990) in isolation; in both cases one adds *folding*.

Specifically, when we encounter the term *append xs ys* while transforming, we *define* a new function $h\ xs\ ys = \mathscr{P}[\![\ append\ xs\ ys\ ]\!]$, and *fold* when the term *append xs ys* is encountered again, as illustrated in figure 9. The result of transforming the term *append* $(1 : xs)\ ys$ then is the term $1 : (h\ xs\ ys)$, where $h\ xs\ ys =$ **case** xs **of** $[] \rightarrow ys$; $(x : xs') \rightarrow x : (h\ xs'\ ys)$.

This folding strategy, called *α-identical folding*, works in general as follows. In every step where clause (3) is applied, the result of transformation is a call to a new function:

$$\mathscr{P}[\![\ e\langle f\ t_1 \ldots t_n\rangle\ ]\!] = \ulcorner f'\ u_1 \ldots u_m \urcorner$$

where $u_1 \ldots u_m$ are the free variables of $e\langle f\ t_1 \ldots t_n\rangle$ in order of their first occurrence. At the same time one adds to the residual program a definition of f' satisfying:

$$f'\ u_1 \ldots u_m = \mathscr{P}[\![\ e\langle t\{v_i := t_i\}_{i=1}^n\rangle\ ]\!]$$

where $f\ v_1 \ldots v_n = t$. If, during the execution of $\mathscr{P}[\![\ e\langle t\{v_i := t_i\}_{i=1}^n\rangle\ ]\!]$, the transformer

encounters a renaming $(e\langle f\ t_1 \ldots t_n\rangle)\sigma$ of $(e\langle f\ t_1 \ldots t_n\rangle)$, it uses the rule

$$\mathscr{P}[\![\ (e\langle f\ t_1 \ldots t_n\rangle)\sigma\ ]\!] = \ulcorner (f'\ u_1 \ldots u_m)\sigma \urcorner$$

All this is left implicit in the subsequent examples.*

The reason for folding only on terms with a function call in the redex is that any infinite sequence of transformation steps must include such a term. The same approach is taken in Wadler (1990).

5.4 Correctness

There are three issues of correctness for $\mathscr{P}$: preservation of operational semantics, non-degradation of efficiency, and termination.

Preservation of operational semantics. The output of $\mathscr{P}$ (if any) should be semantically equivalent to the input. That each step of the transformation rules for $\mathscr{P}$ preserves normal-order graph reduction semantics is easily proved, but extending rigorously the proof to account for folding is more involved. A general technique due to Sands (1995a) can be used to prove this for deforestation as well as positive supercompilation – see Sands (1995b).

The point is that the residual term and program terminates neither more nor less than the original term and program. As an example, consider the term $s \equiv e\langle \textbf{if } b = b' \textbf{ then } t \textbf{ else } t'\rangle$, where b loops at run-time. Is it safe to transform s into $\textbf{if } b = b' \textbf{ then } e\langle t\rangle MGU(b,b') \textbf{ else } e\langle t'\rangle$ which loops at run-time, since b loops? It *is*, because the redex $\textbf{if } b = b' \textbf{ then } t \textbf{ else } t'$ is needed in normal-order graph reduction of s, so s too loops at run-time. In short, the residual program terminates as often as the original program.

Conversely, some precaution is taken to ensure that the residual program terminates no more often that the original program. For instance, it is tempting to use rule (6) when the *MGU* of b and b' is the identity, even when b, b' are not closed. However, this would not be sound; at run-time the term at the right hand side of (6) might be more terminating than the one at the left hand side if b loops.

Non-degradation in efficiency. The output of $\mathscr{P}$ should be at least as efficient as the input. There are several aspects of this problem.

First of all, there is the problem of avoiding *duplication of computation.* Since rewriting to a non-linear right hand side can cause function call duplication, transformation can change a polynomial time program into an exponential time program. In Wadler's deforestation (1990) this is avoided by considering only *linear* terms. Some weaker restrictions are adopted in partial evaluation (Sestoft, 1988; Bondorf, 1990) and other work on deforestation (Chin, 1992; Hamilton, 1993). We consider this an issue in its own right which should be dealt with separately, and is beyond the scope of this paper. The problem does not turn up in the examples we consider.

* Definitions that were not used for folding during the transformation can be unfolded in the residual program; this is done in some examples.

Second, there is the problem of *code duplication.* Unrestrained unfolding may increase the size of a program dramatically. However, the size of a program does not degrade its efficiency. Again, this is an issue in its own right, and is beyond the scope of this paper.

Finally, one may ask conversely: how much can the efficiency of programs be increased by positive supercompilation? – after all, improving efficiency is what we are after. In Sørensen (1994a) it is shown that transformed programs arising from deforestation and positive supercompilation run faster by at most a constant factor (the factor may depend on the program's initial term). A related result for partial evaluation is due to Andersen and Gomard (1992).

Termination. The algorithm $\mathscr{P}$ together with folding should always terminate, but in fact does not, just as deforestation does not always terminate. However, an important aspect of deforestation is that there is a syntactic class of function definitions, *treeless* definitions (Wadler, 1990), such that deforestation of any composition of functions with treeless definitions is guaranteed to terminate. This fact also forms the core of the techniques of Chin (1992) and Hamilton (1993); different techniques are due to Sørensen (1994b) and Seidl (1996)

The problem is more complex for positive supercompilation – the extra power does not come for free. The following term is treeless (though not linear) so deforestation terminates, and yet positive supercompilation does not terminate.

$$\begin{array}{rcl} & & f\ xs\ xs \\ f\ ys\ zs & = & \textbf{case}\ ys\ \textbf{of}\ [\,] \rightarrow zs;\ (x : xs') \rightarrow f\ xs'\ ys \end{array}$$

The problem is that $\mathscr{P}$ encounters the successively larger terms

$$f\ xs\ xs, \quad f\ xs'\ (x' : xs'), \quad f\ xs''\ (x' : x'' : xs''), \quad \ldots$$

Recent work by Sørensen and Glück (1995) gives a general solution to this problem.

5.5 Deforestation versus positive supercompilation

We now describe deforestation and its relation to positive supercompilation. By $\mathscr{D}$ we shall mean the algorithm obtained from $\mathscr{P}$ by replacing rules (5, 8) of driving by (5d, 8d) – see figure 10. The resulting set of rules is identical to the rules for deforestation as defined in Wadler (1990), except that $\mathscr{D}$ also deals with the equality construct and that $\mathscr{D}$ is formulated using evaluation contexts as in Ferguson and Wadler (1988). However, it should be noted that we are using $\mathscr{D}$ beyond the original, intended scope of deforestation, since we apply $\mathscr{D}$ to arbitrary terms rather than just compositions of functions with treeless definitions. From now on, we shall identify deforestation with algorithm $\mathscr{D}$ with α-identical folding in all examples.

The *essential difference* between positive supercompilation and deforestation is thus revealed in clause (5) of the transformer $\mathscr{P}$: the pattern p_j is substituted in the corresponding branches; in deforestation this is not the case. Similarly with rule (8). Obviously, if v occurs neither in e nor in the branches s_i of

$$e\langle \textbf{case}\ v\ \textbf{of}\ \ p_1 \rightarrow s_1; \ldots ; p_m \rightarrow s_m \rangle$$

$$(5d)\quad \mathscr{D}[\![\, e\langle \textbf{case } v \textbf{ of } p_1 \to s_1; \ldots; p_m \to s_m\rangle \,]\!] = \ulcorner \textbf{case } v \textbf{ of } p_1 \to \mathscr{D}[\![\, e\langle s_1\rangle \,]\!]; \ldots; p_m \to \mathscr{D}[\![\, e\langle s_m\rangle \,]\!] \urcorner$$

$$(8d)\quad \mathscr{D}[\![\, e\langle \textbf{if } b = b' \textbf{ then } t \textbf{ else } t'\rangle \,]\!] = \ulcorner \textbf{if } b = b' \textbf{ then } \mathscr{D}[\![\, e\langle t\rangle \,]\!] \textbf{ else } \mathscr{D}[\![\, e\langle t'\rangle \,]\!] \urcorner \quad \text{if } b, b' \text{ are not both closed}$$

Fig. 10. The rules for deforestation.

there is *no* difference between deforestation and positive supercompilation. More generally, an easy induction shows that for a linear term comprising only calls to function definitions with linear right-hand sides, deforestation and positive supercompilation give exactly the same result.

It may seem that the difference in clause (5) and (8) only manifests itself in 'naive' programs, and that this can be corrected by a simple change in the program before or after the actual transformation. In the next section we will see that this assumption is wrong. Situations may arise *during transformation* where positive supercompilation can take advantage of *sharing between different arguments* of certain calls. The seemingly small difference between rules (5, 8) and (5d, 8d) drastically increases the power of $\mathscr{P}$ compared to $\mathscr{D}$: the former passes the KMP test, while the latter does not.

6 KMP Test of the positive supercompiler and deforestation

In this section we show that $\mathscr{P}$ transforms naively specialized matchers into matchers similar to the KMP specialized matchers. Even the matchers derived by $\mathscr{P}$ are not entirely optimal; we explain why and show how $\mathscr{P}$ can be extended to generate the optimal versions.

We also show that $\mathscr{D}$ cannot derive KMP specialized matchers, and we explain how hand rewriting the matcher using an idea due to Consel and Danvy (1989) solves the problem.

6.1 Pattern matching with the positive supercompiler

Applying the positive supercompiler $\mathscr{P}$ to the term *match AAB* ss_0 returns the almost optimal program in figure 11.

This is the desired KMP specialized matcher, disregarding the repeated innermost tests $A = s'$ in $loop_{AB}$ and $loop_B$. But these redundant tests do *not* affect run-time seriously: there is a constant c such that the total number of redundant tests in the entire evaluation of $loop_p\ ss$ for any pattern p and subject string ss is bound by $c \cdot |ss|$ (Sørensen, 1994a). In the terminology of section 3.2, $\mathscr{P}$ passes the KMP test.

Elaboration of the example. It is instructive to try to understand in detail why $\mathscr{P}$ produces the program in figure 11. Application of the positive supercompiler to the term *match AAB* ss_0 begins as shown in figure 12. The occurrences of ss_0 in

```
                  loop_AAB ss_0
loop_AAB ss  =    case ss of
                       []      → False
                  (s' : ss')   → if A = s'
                                 then loop_AB ss'
                                 else loop_AAB ss'
loop_AB ss   =    case ss of
                       []      → False
                  (s' : ss')   → if A = s'
                                 then loop_B ss'
                                 else if A = s'
                                      then loop_AB ss'
                                      else loop_AAB ss'
loop_B ss    =    case ss of
                       []      → False
                  (s' : ss')   → if B = s'
                                 then True
                                 else if A = s'
                                      then loop_B ss'
                                      else if A = s'
                                           then loop_AB ss'
                                           else loop_AAB ss'
```

Fig. 11. Almost KMP specialized matcher.

```
P[[ match AAB ss_0 ]]

=  P[[ loop AAB ss_0 AAB ss_0 ]]                                   (3)

=  P[[ case ss_0 of                                                (3,4)
          []      → False
     (s_1 : ss_1) → if A = s_1
                    then loop AB ss_1 AAB ss_0
                    else next AAB ss_0                  ]]
=  case ss_0 of                                                    (5,2,8)
       []      → False
  (s_1 : ss_1) → if A = s_1
                 then P[[ loop AB ss_1 AAB (A : ss_1) ]]
                 else P[[ next AAB (s_1 : ss_1) ]]
```

Fig. 12. Positive driving $\mathcal{P}[\![$ *match AAB* ss_0 $]\!]$.

the branches of the case-expression are instantiated to $(s_1 : ss_1)$, and in the true branch of the conditional, s_1 is instantiated to A. Note that in the false branch of the conditional, no instantiation of s_1 is made.

Transformation of the call to *loop* proceeds as shown in figure 13, and transformation of the calls to *next* as in figure 14. Figures 12–14 with α-identical folding together give the almost optimal program shown in figure 11.

$$\mathscr{P}[\![\ loop\ AB\ ss_1\ AAB\ (A : ss_1)\ ]\!]$$

$$= \quad \textbf{case}\ ss_1\ \textbf{of} \qquad\qquad (3,4,5,2,8)$$
$$[\,] \rightarrow False$$
$$(s_2 : ss_2) \rightarrow \textbf{if}\ A = s_2$$
$$\textbf{then}\ \ \mathscr{P}[\![\ loop\ B\ ss_2\ AAB\ (A : A : ss_2)\ ]\!]$$
$$\textbf{else}\ \ \mathscr{P}[\![\ next\ AAB\ (A : s_2 : ss_2)\ ]\!]$$

$$\mathscr{P}[\![\ loop\ B\ ss_2\ AAB\ (A : A : ss_2)\ ]\!]$$

$$= \quad \textbf{case}\ ss_2\ \textbf{of} \qquad\qquad (3,4,5,2,8)$$
$$[\,] \rightarrow False$$
$$(s_3 : ss_3) \rightarrow \textbf{if}\ B = s_3$$
$$\textbf{then}\ \ \mathscr{P}[\![\ loop\ [\,]\ ss_3\ AAB\ (A : A : B : ss_3)\ ]\!]$$
$$\textbf{else}\ \ \mathscr{P}[\![\ next\ AAB\ (A : A : s_3 : ss_3)\ ]\!]$$

$$\mathscr{P}[\![\ loop\ [\,]\ ss_3\ AAB\ (A : A : B : ss_3)\ ]\!] \quad = \quad True \qquad (3,4,2)$$

Fig. 13. Positive driving $\mathscr{P}[\![\ match\ AAB\ ss_0\ ]\!]$—continued (1).

Transformation of the call *next AAB* $(A : A : s_3 : ss_3)$ in figure 14 corresponds to the situation where the string starts with two A's and a symbol different from B. This information is represented in the instantiations made to the original variable ss_0, and these instantiations imply that the transformer can reduce away some of the comparisons that would otherwise have taken place during run-time.

The reason for the redundant test $A = s$ in $loop_{AB}$ and $loop_B$ is that the positive supercompiler ignores *negative* information (restrictions) that could be gathered during transformation: when the transformer proceeds to the false branch of a conditional, the information that the equality does not hold is not recorded. The transformer maintains only *positive* information (assertions): in the true-branch of a conditional a substitution is performed representing the information that the equality test is assumed to come out true.

Representing negative information, i.e. the information that an equality does not hold, requires a more sophisticated representation than positive information; we return to this in section 7.2. A program transformer that has the capacity to eliminate all unreachable branches in a program has *perfect* information propagation (Glück and Klimov, 1993). A perfect version of driving that propagates positive and negative information is defined in the same work.

While both positive and negative information arise from an equality test, only positive information arises from a case-expression **case** v **of** $p_1 \rightarrow t_1; \ldots ; p_m \rightarrow t_m$. If there were a 'catch-all' branch at the end of the case-expression, then in this branch there would be negative information just as in the false branch of a conditional; the difficulties in representing it would be the same as for equality tests.

$\mathscr{P}[\![\ next\ AAB\ (s_1 : ss_1)\ ]\!]$

$= \quad \mathscr{P}[\![\ loop\ AAB\ ss_1\ AAB\ ss_1\ ]\!]$ (3,4)

$\mathscr{P}[\![\ next\ AAB\ (A : s_2 : ss_2)\ ]\!]$

$= \quad \mathscr{P}[\![\ loop\ AAB\ (s_2 : ss_2)\ AAB\ (s_2 : ss_2)\ ]\!]$ (3,4)

$=$ **if** $A = s_2$ (3,4,4,8)
then $\mathscr{P}[\![\ loop\ AB\ ss_2\ AAB\ (A : ss_2)\ ]\!]$
else $\mathscr{P}[\![\ next\ AAB\ (s_2 : ss_2)\ ]\!]$

$\mathscr{P}[\![\ next\ AAB\ (A : A : s_3 : ss_3)\ ]\!]$

$= \quad \mathscr{P}[\![\ loop\ AAB\ (A : s_3 : ss_3)\ AAB\ (A : s_3 : ss_3)\ ]\!]$ (3,4)

$=$ **if** $A = s_3$ (3,4,4,8)
then $\mathscr{P}[\![\ loop\ B\ ss_3\ AAB\ (A : A : ss_3)\ ]\!]$
else $\mathscr{P}[\![\ next\ AAB\ (A : s_3 : ss_3)\ ]\!]$

Fig. 14. Positive driving $\mathscr{P}[\![\ match\ AAB\ ss_0\ ]\!]$—continued (2).

6.2 *Pattern matching with deforestation*

Applying $\mathscr{D}$ to the term *match AAB* ss_0 gives the term and program in figure 15.

This program is only improved in the sense that the p argument has been removed. (Incidentally, this again shows that deforestation can perform program specialization.) But each time a match fails, the head of the string is ignored, and the match starts all over again. Deforestation does not pass the KMP test.

There seems to be no simple change to the general matcher in figure 2, or to the specialized matcher in figure 15, to obtain the result from figure 11 (the next subsection contains a non-trivial change to do it, though). It is only after a number of transformation steps that the extra power of rule (5) in positive supercompilation comes into play.

Elaboration of the example. As for positive supercompilation, it is instructive to understand the details of deforestation. Applying $\mathscr{D}$ to the term *match AAB* ss_0 proceeds as shown in figure 16.

The occurrences of ss_0 in the branches of the case-expression remain uninstantiated; the information that s_1 and ss_1 are the head and tail of ss_0 is lost. The calls to *next* have no instantiations recording assumptions about the string.

Transformation then proceeds as shown in figure 17 which, together with figure 16 and α-identical folding, gives the program in figure 15.

```
                    loop_AAB ss0
loop_AAB ss0   =    case ss0 of
                       []      → False
                     (s1 : ss1) → if A = s1
                                  then loop_AB ss1
                                  else case ss1 of
                                         []      → False
                                       (s : ss)  → loop_AAB ss
loop_AB ss1    =    case ss1 of
                       []      → False
                     (s2 : ss2) → if A = s2
                                  then loop_B ss2
                                  else case ss1 of
                                         []      → False
                                       (s : ss)  → loop_AAB ss
loop_B ss2     =    case ss2 of
                       []      → False
                     (s3 : ss3) → if B = s3
                                  then True
                                  else case ss1 of
                                         []      → False
                                       (s : ss)  → loop_AAB ss
```

Fig. 15. Inefficient specialized matcher.

```
𝒟[[ match AAB ss0 ]]

  =  𝒟[[ loop AAB ss0 AAB ss0 ]]                                   (3)

  =  𝒟[[ case ss0 of                                               (3)
           []      → False
         (s1 : ss1) → if A = s1
                      then loop AB ss1 AAB ss0
                      else next AAB ss0                  ]]
  =  case ss0 of                                                   (5)
       []      → False
     (s1 : ss1) → if A = s1
                  then 𝒟[[ loop AB ss1 AAB ss0 ]]
                  else 𝒟[[ next AAB ss0 ]]
```

Fig. 16. Deforesting $\mathscr{D}[\![$ *match AAB* ss_0 $]\!]$.

6.3 Two alternative ways to pass the KMP test

There are two ways of overcoming insufficient transformational power: the interpretive approach, and binding-time improvements.

In the *interpretive approach* an interpreter is specialized with respect to the source program, instead of transforming the source program directly. It is a surprising fact that, given an appropriate interpreter, this method may drastically increase the

$$\mathscr{D}[\![\ loop\ AB\ ss_1\ AAB\ ss_0\ ]\!]$$

$$= \quad \textbf{case}\ ss_1\ \textbf{of} \qquad (3,4,5,2,8)$$
$$[\,] \rightarrow False$$
$$(s_2 : ss_2) \rightarrow \textbf{if}\ A = s_2$$
$$\textbf{then}\ \ \mathscr{D}[\![\ loop\ B\ ss_2\ AAB\ ss_0\ ]\!]$$
$$\textbf{else}\ \ \mathscr{D}[\![\ next\ AAB\ ss_0\ ]\!]$$

$$\mathscr{D}[\![\ loop\ B\ ss_2\ AAB\ ss_0\ ]\!]$$

$$= \quad \textbf{case}\ ss_2\ \textbf{of} \qquad (3,4,5,2,8)$$
$$[\,] \rightarrow False$$
$$(s_3 : ss_3) \rightarrow \textbf{if}\ B = s_3$$
$$\textbf{then}\ \ \mathscr{D}[\![\ loop\ [\,]\ ss_3\ AAB\ ss_0\ ]\!]$$
$$\textbf{else}\ \ \mathscr{D}[\![\ next\ AAB\ ss_0\ ]\!]$$

$$\mathscr{D}[\![\ loop\ [\,]\ ss_3\ AAB\ ss_0\ ]\!] \quad = \quad True \qquad (3,4,2)$$

$$\mathscr{D}[\![\ next\ AAB\ ss_0\ ]\!]$$

$$= \quad \textbf{case}\ ss_0\ \textbf{of} \qquad (3,5,2)$$
$$[\,] \rightarrow False$$
$$(o : os) \rightarrow \mathscr{D}[\![\ loop\ AAB\ os\ AAB\ os\ ]\!]$$

Fig. 17. Deforesting $\mathscr{D}[\![\ match\ AAB\ ss_0\ ]\!]$ – continued.

power of the overall transformation. The interpreter may use various techniques such as fixing a non-standard evaluation order and manipulating various kinds of information. The overall effect is the same as if the transformer had adopted these techniques.

More specifically, Glück and Jørgensen (1994) show that partial evaluators can produce KMP specialized matchers by specializing an *information propagating* interpreter with respect to the general matcher and a fixed pattern. We conjecture that the same approach works for deforestation (although not within the original scope of deforestation since the interpreter is not treeless).

Binding-time improvements are semantics-preserving transformations on source programs. Instead of extending the transformer with more powerful techniques, or specializing an interpreter with these techniques to the source program as in the interpretive approach, one internalizes the extra techniques in the source program.

The reason that deforestation does not pass the KMP test is that information about tests is lost during the transformation; the same holds for partial evaluators. Consel and Danvy (1989) show that partial evaluators can derive specialized KMP matchers if the general matcher is rewritten so as explicitly to *maintain the information that positive supercompilation gathers* during transformation. The program in

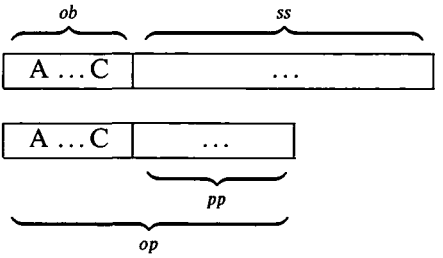
```
  match pp ss
= loop pp ss pp [] []

  loop pp ss op bb ob
= case pp of
      []     → True
    (p : pp) → case bb of
                 []     → case ss of
                            []     → False
                          (s : ss) → if p = s
                                     then loop pp ss op [] (append ob [p])
                                     else next op ss ob
               (b : bb) → if p = b
                          then loop pp ss op bb ob
                          else loop op ss op (tl ob) (tl ob)
  next op ss ob
= case ob of
      []     → loop op os op [] []
    (o : ob) → loop op os op ob ob
```

Fig. 18. Information propagating matcher.

figure 18, inspired by Consel and Danvy (1989) and Jones *et al.* (1993), does this (where *tl* takes the tail of a list).

The formal parameters *pp*, *ss*, *op*, *ob* of *loop* have the following meaning:

- *pp* is the current pattern: the suffix of the original pattern that has not yet been matched with the string.
- *op* is the original pattern, and *pp* is always a suffix of *op* as in the general matcher (figure 2).
- *ss* is the current string: the suffix of the original string that has not yet been matched with the pattern.
- *ob* is the *backup*: the prefix of the original string that has been successfully matched with the pattern. The original string is always the concatenation of *ob* and *ss*. The backup is maintained by appending the head of the current pattern to the backup after every successful match.



The parameter *bb* is [] until a mismatch occurs, and then *ob* is set to its own tail, and *bb* is set to *ob*. We then start matching the original pattern against *bb* until the latter becomes empty. When this happens we proceed by matching the remaining

part of the pattern against the current string *ss*, maintaining the backup *ob*. If a mismatch occurred while matching *bb* against the pattern, we set *ob* to it own tail, set *bb* to *ob*, and restart the process with the tail of *ob*, thereby skipping an element of the original string.

Applying deforestation (or positive supercompilation) to the term *match AAB ss* with this new program yields the almost KMP specialized matcher from figure 11. Note that the new program in figure 18 is not more efficient than the general matcher in figure 2 (actually, on the contrary); it simply manipulates more information.

7 Other optimization techniques

In this section we relate, more briefly, information propagation as used in positive supercompilation and deforestation to that in partial evaluation, Turchin's supercompiler, GPC and partial deduction.

7.1 Partial evaluation of functional programs

Partial evaluation as in Jones *et al.* (1993) propagates only the values of static variables, namely constants. Partial evaluation can specialize programs but is strictly weaker than positive supercompilation, Turchin's supercompiler, GPC, and partial deduction since it propagates no information obtained from predicates or pattern matching. This explains the result found in Consel and Danvy (1989), that partial evaluation does not pass the KMP test.

7.2 Turchin's supercompiler

Recall that $\mathscr{P}$ is an extension of the *rewrite* interpreter $\mathscr{I}$: when $\mathscr{P}$ and $\mathscr{I}$ unfold a function call they replace the call by the body of the called function and substitute the actual arguments into the term being transformed or interpreted. Alternatively, one can think of an *environment based* interpreter which creates bindings of the formal parameters to the actual arguments and a corresponding environment based version of $\mathscr{P}$, as in Glück and Klimov's formulation (1993) of Turchin's supercompiler.*

The driving mechanisms in the positive supercompiler and in Turchin's supercompiler are essentially 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 definitely fail, and this is done in Turchin's supercompiler. A technique using substitutions does not seem possible.

There is no difference between using environments or substitution for positive

* Glück and Klimov (1993) consider perfect driving for a tail-recursive functional language. The connection between driving and deforestation is not shown since the language has no intermediate data structures, at least not in the form of nested calls.

information with respect to transformation power.* If one applies Turchin's supercompiler to the general matcher, but using only positive information, one gets the same program that $\mathscr{P}$ produces; applying Turchin's supercompiler unrestricted yields the desired optimal program (Glück and Klimov, 1993). That a supercompiler can pass the KMP test on a similar general matcher was first shown in Glück and Turchin (1990).

We should note that supercompilation, as defined by Turchin, is a normal-order transformation that is applied to applicative-order programs and that transformed programs are again interpreted call-by-value. As a result, supercompilation may make programs terminate more often. On the other hand, the positive supercompiler transforms programs with normal-order graph reduction semantics into programs with the same semantics and termination properties (assuming that the transformer itself stops). The same holds for deforestation.

7.3 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. Whenever a test is encountered, a theorem prover checks 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 propagates predicates rather than just value information, and it assumes the (unlimited) power of a theorem prover. It was shown in Futamura and Nogi (1988) that this information suffices to pass the KMP test on a general matcher.

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

Takano (1991) concretized GPC for the language of section 2. His formulation includes an environment-based version of our rule (5):

$$\begin{aligned} &G[\![\ \textbf{case}\ v\ \textbf{of}\ \ p_1 \to t_1; \ldots ; p_n \to t_n\ ]\!]E = \\ &\quad \ulcorner \textbf{case}\ v\ \textbf{of}\ \ p_1 \to G[\![\ t_1\ ]\!]E_1; \ldots ; p_n \to G[\![\ t_n\ ]\!]E_n \urcorner \\ &\quad \textbf{where}\ E_i = E \cup \{v \leftrightarrow p_i\} \end{aligned}$$

where the E's are sets of equalities which are used in the manner described in more general terms in Futamura and Nogi (1988); concretely, they represent positive information arising from pattern matching.

* In self-application of an offline partial evaluator one does a binding-time analysis of the partial evaluator itself; such an analysis usually gives better results for the environment-based version because it has a better separation of static/dynamic data. However, supercompilation and deforestation do not employ binding-time analysis and do not divide objects into being completely known or completely unknown, so in this setting the difference seems immaterial.

$$
\begin{array}{lcl}
 & & \mathit{append}\ [A,B]\ (\mathit{append}\ xs\ (C:ys)) \\
\mathit{append}\ xs\ ys & = & \textbf{case}\ xs\ \textbf{of}\ [] \rightarrow ys;\ (x:xs) \rightarrow x : \mathit{append}\ xs\ ys \\
 & & \\
 & & A:B:a\ xs\ ys \\
a\ xs\ ys & = & \textbf{case}\ xs\ \textbf{of}\ [] \rightarrow C:ys;\ (x:xs) \rightarrow x : a\ xs\ ys
\end{array}
$$

Fig. 19. Specialization by positive supercompilation.

7.4 Partial deduction

Glück and Sørensen (1994) show that unfolding in partial deduction and driving in positive supercompilation are essentially the same. The essential aspect of partial deduction, in this case, is the way goals are unified and the resulting substitutions are applied to the goals in the next step of transformation, much like in clauses (5) and (8) of positive driving. This explains why, unlike the situation in the functional case, partial deducers for Prolog can derive KMP matchers from a general Prolog matcher similar to our general matcher (Smith, 1991).

8 More applications of positive supercompilation

We consider more applications of positive supercompilation: specialization (section 8.1), composition, (section 8.2) and other effects (section 8.3).

8.1 Specialization

Positive supercompilation can perform program specialization. For instance, applying positive supercompilation to *append* $[A,B]$ (*append* xs $(C:ys)$) yields the new term $A:B:a\ xs\ ys$ where a has no computations involving A, B, or C – see figure 19.

The same effect is achieved by partial evaluation (with partially static structures), deforestation, GPC, Turchin's supercompiler and, for a logic programming version of the program, partial deduction.

8.2 Composition

Positive supercompilation can perform program composition. For instance, applying positive supercompilation to *append xs* (*append ys zs*) yields the new term *da xs ys zs* where *da* does not construct an intermediate data structure – see figure 20.

The same effect is achieved by deforestation and Turchin's supercompilation. In contrast, it is well-known that partial evaluation does not, in general, eliminate intermediate data structures (Consel and Danvy, 1991). The reason is that in order to eliminate intermediate data structures, the transformer must use normal-order transformation order, whereas partial evaluation typically uses applicative-order (Nielsen and Sørensen, 1995). Partial deduction of a logic programming version also

```
                    append xs (append ys zs)
append xs ys    =   case xs of [] → ys; (x : xs) → x : append xs ys

                    da xs ys zs
da xs ys zs     =   case xs of [] → append ys zs; (x : xs) → x : da xs ys zs
```

Fig. 20. Composition by positive supercompilation.

fails to eliminate the intermediate data structure. The reason is that partial deduction does something that corresponds, in the functional setting, to transforming the two calls to *append* independently, thereby preventing the intermediate data structure constructed by one call to be consumed by the other. Other transformers for logic programs *can* do this optimization (Proietti and Pettorossi, 1991).

8.3 Other effects

Positive supercompilation can invert functions just like interpreters for logic programs, and can be used for theorem proving and problem solving (Turchin, Nirenberg, and Turchin, 1982; Turchin, 1986). In general, to obtain these effects, the additional power of positive supercompilation over deforestation is required. Most of these examples have been considered for Turchin's supercompiler, but the positive supercompiler can achieve similar effects (as demonstrated by the KMP example above). Finally, positive supercompilation can, as partial evaluation, translate programs by specializing an interpreter.

9 Conclusion and future work

We have introduced positive supercompilation and compared this to deforestation and, more briefly, to partial evaluation, supercompilation, GPC and partial deduction. We have shown 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 have 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 simple and general string matcher and a fixed pattern. Deforestation and partial evaluation for functional languages cannot achieve this.

Thus, while deforestation and positive supercompilation have identical effects on linear terms (comprising only calls to functions with linear right hand sides) there is in general a difference between the two on non-linear terms. In some cases positive supercompilation strictly improves deforestation, in other cases deforestation terminates while positive supercompilation loops. It would be interesting to state general results about the relationship between deforestation and positive supercompilation on non-linear terms.

It would also be interesting to find a translation internalizing positive information

propagation. Such a translation could conceivably, as a special case, transform the general matcher in figure 2 to that in figure 18; it would introduce some kind of instantiation management explicitly into terms.

A direct comparison of methods is often blurred because of different language paradigms and different perspectives. We believe that future work should aim at bringing different methodologies closer.

Acknowledgments

Thanks to Sergei Abramov, Andrei Klimov, Andrei Nemytykh, Sergei Romanenko, and last but not least, Valentin F. Turchin for many interesting discussions and hospitality during visits in Russia and New York. We had interesting discussions with Jesper Jørgensen, Kristian Nielsen, and David Sands. Wei-Ngan Chin, Philip Wadler and Stephan Diehl provided useful comments on an earlier version of the paper, and the anonymous referees gave constructive feedback for improvements. Finally, we are indebted to the members of the Topps group at DIKU for providing an excellent working environment.

The first author is indebted to M. Hagyia for support during a visit to Japan. The second author was supported by an Erwin-Schrödinger-Fellowship of the Austrian Science Foundation (FWF) under grant J0780 & J0964. The work was partially supported by the DART project funded by the Danish Natural Sciences Research Council.

References

Andersen, L. O. and Gomard, C. K. (1992) Speedup analysis in partial evaluation. *ACM Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report YALEU/DCS/RR-909, pp. 1–7.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M. and Wadler, P. (1995) A call-by-need lambda calculus. *22nd ACM Symposium on Principles of Programming Languages 1995*, pp. 233–246. ACM Press.

Augustsson, L. (1985) Compiling lazy pattern-matching. *Conference on Functional Programming and Computer Architecture*, Jouannaud, J.-P. (ed.). *Lecture Notes in Computer Science 201*, pp. 368–381. Springer-Verlag.

Bird, R. (1984) Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, **21**: 239–250.

Bird, R. S. and Wadler, P. L. (1988) *Introduction to Functional Programming*. Prentice-Hall.

Bondorf, A. (1990) *Self-applicable partial evaluation*. PhD thesis, DIKU-Rapport 90/17, Department of Computer Science, University of Copenhagen.

Burstall, R. M. and Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM*, **24**(1): 44–67.

Chin, W.-N. (1992) Safe fusion of functional expressions. *ACM Conference on Lisp and Functional Programming*, pp. 11–20. ACM Press.

Consel, C. and Danvy, O. (1989) Partial evaluation of pattern matching in strings. *Information Processing Letters*, **30**(2):79–86.

Consel, C. and Danvy, O. (1991) For a better support of static data flow. *Conference on Functional Programming and Computer Architecture*, Hughes, J. (ed.). *Lecture Notes in Computer Science 523*, pp. 495–519. Springer-Verlag.

Feather, M. S. (1982) A system for assisting program transformation. *ACM Trans. Programming Languages and Systems*, **4**(1):1–20.

Ferguson, A. B. and Wadler, P. L. (1988) When will deforestation stop?. *Glasgow Workshop on Functional Programming*, pp. 39–56.

Futamura, Y. and Nogi, K. (1988) Generalized partial computation. *Partial Evaluation and Mixed Computation*, Bjørner, D., Ershov, A. P. and Jones, N. D. (eds.), pp. 133–151. North-Holland.

Futamura, Y. (1988) Program evaluation and generalized partial computation. *International Conference on Fifth Generation Computer Systems*, pp. 1–8, Tokyo, Japan.

Glück, R. and Klimov, A. V. (1993) Occam's razor in metacomputation: the notion of a perfect process tree. *Static Analysis*, Cousot, P., Falaschi, M., Filè, G. and Rauzy, A. (eds.), *Lecture Notes in Computer Science 724*, pp. 112–123. Springer-Verlag.

Glück, R. and Jørgensen, J. (1994) Generating transformers for deforestation and supercompilation. *Static Analysis*, Le Charlier, B. (ed.), *Lecture Notes in Computer Science 864*, pp. 432–448. Springer-Verlag.

Glück, R. and Sørensen, M. H. (1994) Partial deduction and driving are equivalent. *Programming Language Implementation and Logic Programming*, Hermenegildo, M. and Penjam, J. (eds.), *Lecture Notes in Computer Science 844*, pp. 165–181. Springer-Verlag.

Glück, R. and Turchin, V. F. (1990) Application of metasystem transition to function inversion and transformation. *Proc. ISSAC'90*, pp. 286–287. ACM Press.

Hamilton, G. W. (1993) *Compile-time optimisation of storage usage in lazy functional programs.* PhD thesis, University of Stirling.

Jones, N. D. (1988) Automatic program specialization: a re-examination from basic principles. *Partial Evaluation and Mixed Computation*, Bjørner, D., Ershov A. P. and Jones N. D. (eds.), pp. 225–282. North-Holland.

Jones, N. D., Gomard, C. K. and Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation.* Prentice-Hall.

Jones, N. D. (1994) The essence of program transformation by partial evaluation and driving. *Logic, Language and Computation*, Jones, N. D., Hagiya, M. and Sato, M. (eds.), *Lecture Notes in Computer Science 792*, pp. 206–224. Springer-Verlag.

Komorowski, J. (1992) An introduction to partial deduction. *Meta-Programming in Logic*, Pettorossi, A. (ed.), *Lecture Notes in Computer Science 649*, pp. 49–69. Springer-Verlag.

Knuth, D. E., Morris, J. H. and Pratt, V. R. (1977) Fast pattern matching in strings. *SIAM J. Computing*, **6**(2):323–350.

Launchbury, J. (1993) A natural semantics for lazy evaluation. *20th ACM Symposium on Principles of Programming Languages*, pp. 144–154. ACM Press.

Lloyd, J. W. and Shepherdson, J. C. (1991) Partial evaluation in logic programming. *J. Logic Programming*, **11**(3–4): 217–242.

Nielsen, K. and Sørensen, M. H. (1995) Call-by-name CPS-translation as a binding-time improvement. *Static Analysis*, Mycroft, A. (ed.), *Lecture Notes in Computer Science 983*, pp. 296–313. Springer-Verlag.

Proietti, M. and Pettorossi, A. (1991) Unfolding - Definition - Folding, in this order for avoiding unnecessary variables in logic programs. *Programming Language Implementation and Logic Programming*, Lecture Notes in Computer Science 528, pp. 347–358. Springer-Verlag.

Sands, D. (1995a) Total correctness by local improvement in program transformation. *22nd ACM Symposium on Principles of Programming Languages*, pp. 221–232. ACM Press.

Sands, D. (1995b) Proving the correctness of recursion-based automatic program transformations. *TAPSOFT'95: Theory and Practice of Software Development*, Mosses, P. D., Nielsen,

M. and Schwartzbach, M. I. (eds.), *Lecture Notes in Computer Science 915*, pp. 681–695. Springer-Verlag.

Seidl, H. (1996) Integer constraints to stop deforestation. *Programming Languages and Systems 1996*, to appear in *Lecture Notes in Computer Science*. Springer-Verlag.

Sestoft, P. (1988) Automatic call unfolding in a partial evaluator. *Partial Evaluation and Mixed Computation*, Bjørner, D., Ershov, A. P. and Jones, N. D. (eds.), pp. 485–506. North-Holland.

Smith, D. A. (1991) Partial evaluation of pattern matching in constraint logic programming languages. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 62–71. ACM Press.

Sørensen, M. H. (1994a) *Turchin's supercompiler revisited. An operational theory of positive information propagation.* Master's Thesis, DIKU-rapport 94/9, Department of Computer Science, University of Copenhagen.

Sørensen, M. H. (1994b) A grammar-based data-flow analysis to stop deforestation. *Trees in Algebra and Programming*, Tison, S. (ed.), *Lecture Notes in Computer Science 787*, pp. 335–351. Springer-Verlag.

Sørensen, M. H., Glück, R. and Jones, N. D. (1994) Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. *Programming Languages and Systems*, Sannella, D. (ed.), pp. 485–500. Springer-Verlag.

Sørensen, M. H. and Glück, R. (1995) An algorithm of generalization in positive supercompilation. *Logic Programming: Proceedings of the 1995 International Symposium*, Lloyd, J. (ed.), pp. 465–479. MIT Press.

Takano, A. (1991) Generalized partial computation for a lazy functional language. *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 1–11. ACM Press.

Turchin, V. F. (1979) A supercompiler system based on the language Refal. *SIGPLAN Notices*, **14**(2): 46–54.

Turchin, V. F. (1980) Semantic definitions in Refal and automatic production of compilers. *Semantics-Directed Compiler Generation*, Jones, N. D. (ed.), *Lecture Notes in Computer Science 94*, pp. 441–474. Springer-Verlag.

Turchin, V. F., Nirenberg, R. M. and Turchin, D. V. (1982) Experiments with a supercompiler. *ACM Symposium on Lisp and Functional Programming 1982*, pp. 47–55. ACM Press.

Turchin, V. F. (1986) The concept of a supercompiler. *ACM Trans. Programming Languages and Systems*, **8**(3): 292–325.

Turchin, V. F. (1988) The algorithm of generalization in the supercompiler. *Partial Evaluation and Mixed Computation*, Bjørner, D., Ershov, A. P. and Jones, N. D. (eds.), pp. 341–353. North-Holland.

Wadler, P. L. (1984) Listlessness is better than laziness. *ACM Symposium on Lisp and Functional Programming 1984*, pp. 282–305. ACM.

Wadler, P. L. (1987) Efficient compilation of pattern-matching. *The Implementation of Functional Programming Languages*, Peyton Jones, S. L. (ed.). Prentice-Hall.

Wadler, P. L. (1990) Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, **73**: 231–248 (preliminary version in ESOP'88, *Lecture Notes in Computer Science 300*. Springer-Verlag).