On Deforesting Parameters of Accumulating Maps

Kazuhiko Kakehi[1], Robert Glück[2⋆], and Yoshihiko Futamura[3]

[1] Graduate School of Information Science and Technology, The University of Tokyo
& Institute for Software Production Technology, Waseda University
Tokyo 113-8656, Japan, kaz@ipl.t.u-tokyo.ac.jp
[2] PRESTO, JST & Institute for Software Production Technology
Waseda University, School of Science and Engineering
Tokyo 169-8555, Japan, glueck@acm.org
[3] Institute for Software Production Technology
Waseda University, School of Science and Engineering
Tokyo 169-8555, Japan, futamura@futamura.info.waseda.ac.jp

Abstract. Deforestation is a well-known program transformation technique which eliminates intermediate data structures that are passed between functions. One of its weaknesses is the inability to deforest programs using accumulating parameters.
We show how intermediate lists built by a selected class of functional programs, namely 'accumulating maps', can be deforested using a single composition rule. For this we introduce a new function `dmap`, a symmetric extension of the familiar function `map`. While the associated composition rule cannot capture all deforestation problems, it can handle accumulator fusion of functions defined in terms of `dmap` in a surprisingly simple way. The rule for accumulator fusion presented here can also be viewed as a restricted composition scheme for attribute grammars, which in turn may help us to bridge the gap between the attribute and functional world.

1 Introduction

Deforestation [24] is a well-known program transformation technique which can eliminate intermediate data structures that are passed between functions. *Shortcut deforestation* [9], the most successful variant of deforestation, restricts the optimization to lists that are created with `build` and consumed by `foldr`. This exploits the fact that many list consuming functions are easily expressed in terms of these two higher-order functions. Compared to deforestation, shortcut deforestation has the advantage that a single, local transformation rule suffices to optimize a large class of programs which makes it well suited as compiler optimization (*e.g.*, it is used in the Glasgow Haskell compiler).

⋆ On leave from DIKU, Dept. of Computer Science, University of Copenhagen.

A. Pettorossi (Eds.): Logic Based Program Synthesis and Transformation. Proceedings. LNCS 2372, pp. 46-56, 2002.

One of the weaknesses [2] of such transformations is the inability to deforest programs using accumulating parameters. This is not just a deficiency of these techniques, for other program transformers suffer from the same weakness (*e.g.*, *supercompilation* [22][1], *calculational methods* [10]). When composing a consumer of lists with a producer of lists, where the producer is written using an accumulating parameter, the intermediate list cannot be removed. For example, the expression `rev` (`rev xs`) cannot be fused into a single `copy xs` (let alone identity) when `rev` is the fast version of list reversal.

Clearly, this situation is not satisfactory for program composition. While accumulating parameters often let functions be naturally expressed (*e.g.*, `rev`) and generally improve efficiency, they disable deforestation. As it turns out, the problem of accumulator deforestation is harder to solve in an automatic fashion than one might expect.

We show how intermediate lists built by 'accumulating maps' can be deforested using a single composition rule. For this, we introduce a new function `dmap`, a symmetric extension of the familiar function `map`, and study the composition of list producers and consumers defined in terms of `dmap`. Since `dmap` is a generalization of the map function, our transformations cannot solve all deforestation problems shortcut deforestation can (the latter is based on `foldr`). However, we can handle accumulator fusion in a surprisingly simple and easy way (such as the composition of fast reverses), an optimization which shortcut deforestation cannot do. This is due to the restriction to map-style list producers and consumers. In fact, the laws for `dmap` exhibit some interesting combinatorics.

Recently, composition methods using results of *attribute grammars* have been proposed [4, 13, 14, 16] which allow transformations not possible by ordinary deforestation. The problem of accumulator composition studied in this paper is a restricted instance of the more powerful *composition of attribute grammars* (or also *tree transducers* [6]) [5, 7, 8]. While our class is very limited compared to the general method, it can successfully compose programs with an accumulating parameter, and importantly, *simply*. It also highlights some key features of attribute compositions, in the *functional* world. This in turn may help us to bridge the gap between the functional and the attribute world.

The paper is organized as follows. First we introduce `dmap` and show examples of functions defined using `dmap` (Sect. 2). Then we state the main rule for accumulator fusion and discuss important instances of the rule (Sect. 3), and show several deforestation examples (Sect. 4). After comparing our idea to related works, especially to the theory of attribute grammars (Sect. 5), we conclude with mentioning open problems and limitations (Sect. 6).

Preliminaries In this paper, we borrow the notation of Haskell to describe programs and transformation rules. The syntax of the language is not defined explicitly in this paper; the transformation rules apply to a function skeleton `dmap`. The semantics can be either call-by-need or call-by-value. An important restric-

[1] The relation between supercompilation and deforestation is discussed in [21].

Definition of `dmap`

```
dmap f g []       ys = ys
dmap # # (x : xs) ys = ys
dmap # g (x : xs) ys =          dmap # g xs (g x : ys)
dmap f # (x : xs) ys = (f x) : ( dmap f # xs ys )
dmap f g (x : xs) ys = (f x) : ( dmap f g xs (g x : ys) )
```

Examples

```
copy xs = dmap id # xs []        fba xs = dmap # f xs []
                                          where f a = b, f b = b, f c = c
 rev xs = dmap # id xs []
                                  fcb xs = dmap # g xs []
app xs ys = dmap id # xs ys               where g a = a, g b = c, g c = c
 map f xs = dmap f # xs []
 mir xs = dmap id id xs []
```

Fig. 1. Definition of `dmap` and examples

tion is that the length of input lists is finite. This will be discussed in Sect. 4.5. We assume that there are no side-effects in the functions used in this paper.

2 Accumulating Map

In order to perform accumulator fusion of map-style functions, we introduce a higher-order function skeleton `dmap` ('double map') which extends function `map` in two respects: (1) by an additional function parameter `g` building the elements of an accumulator; (2) by a list parameter `ys` that terminates the list constructed by applying `f` and `g` to the elements of `xs`. Informally, `dmap` is defined by

$$\texttt{dmap f g}\ [\texttt{x}_1, ..., \texttt{x}_n]\ [\texttt{y}_1, ..., \texttt{y}_m] = [\underbrace{\texttt{f x}_1, ..., \texttt{f x}_n}_{\text{apply f}}, \underbrace{\texttt{g x}_n, ..., \texttt{g x}_1}_{\text{apply g}}, \underbrace{\texttt{y}_1, ..., \texttt{y}_m}_{\text{'zero'}}] \ .$$

A definition of `dmap` is shown in Fig. 1. The application of `g` to $[\texttt{x}_1, ..., \texttt{x}_n]$ in reverse order is implemented by an accumulating parameter. In addition, we define that the corresponding sublist is dropped from the output if function `f` or `g` is not present, for notational convenience expressed by a special symbol `#` ('null function'). For example, when `g` is not present, we have the identity:

$$\texttt{dmap f \# xs}\ [\,] = \texttt{map f xs} \ . \tag{1}$$

Figure 1 shows several familiar list processing functions defined in terms of `dmap` (*e.g.*, reverse, append, map). Here `dmap` serves as a primitive for the recursion scheme. Functions `rev`, `mir`, `fba` and `fcb` are implementations using

accumulators. The asymptotic complexity of a function is not changed by defining it in a direct way using `dmap`.

It should be noted that we use the special symbol `#` as syntactic sugar and that we assume `dmap` with `#` is implemented in a suitable form in order not to reexamine function parameters or produce unused lists. Though all four combinations can be defined separately, use of this notation simplifies the composition rule. This will be demonstrated in Sect. 4.

Remark From an attribute grammar perspective, we can say that `dmap`, while performing a recursive descent over the input list `xs`, computes an *inherited attribute* with `g` (passed downwards) and a *synthesized attribute* with `f` (passed upwards). At the end of the recursion, the value of the inherited attribute is passed upwards as the synthesized attribute. We will discuss this connection in more detail in Sect. 5.

3 Fusion of Accumulating Maps

The composition of two functions defined by `map` is given by the familiar rule

$$\texttt{map g (map f xs)} = \texttt{map (g.f) xs} \tag{2}$$

where dot (.) denotes function composition. The expression on the *rhs* avoids the construction of an intermediate list and the composition of the functions `f` and `g` may enable further optimizations.

The composition of two functions defined by `dmap`, which has an accumulating parameter `ys`, is shown in Fig. 2. It can be justified as follows. Let $\texttt{xs} = [\texttt{x}_1, ..., \texttt{x}_n]$, $\texttt{ys} = [\texttt{y}_1, ..., \texttt{y}_m]$, and $\texttt{zs} = [\texttt{z}_1, ..., \texttt{z}_l]$ $(n, m, l \geq 0)$. Then using the definition of `dmap` and list properties we have:

$$\begin{aligned}
&\texttt{dmap f g (dmap h k } [\texttt{x}_1, ..., \texttt{x}_n]\ [\texttt{y}_1, ..., \texttt{y}_m])\ [\texttt{z}_1, ..., \texttt{z}_l] \\
&= \texttt{dmap f g } [(\texttt{h x}_1), ..., (\texttt{h x}_n),\ (\texttt{k x}_n), ..., (\texttt{k x}_1),\ \texttt{y}_1, ..., \texttt{y}_m]\ [\texttt{z}_1, ..., \texttt{z}_l] \\
&= [\,(\texttt{f.h x}_1), ..., (\texttt{f.h x}_n),\ (\texttt{f.k x}_n),\ ..., (\texttt{f.k x}_1), \\
&\quad\ (\texttt{f}\quad \texttt{y}_1), ..., (\texttt{f}\quad \texttt{y}_m),\ (\texttt{g}\quad \texttt{y}_m), ..., (\texttt{g}\quad \texttt{y}_1), \\
&\quad\ (\texttt{g.k x}_1), ..., (\texttt{g.k x}_n),\ (\texttt{g.h x}_n),\ ..., (\texttt{g.h x}_1),\ \texttt{z}_1, ..., \texttt{z}_l\,] \\
&= \texttt{dmap (f.h) (f.k) } [\texttt{x}_1, ..., \texttt{x}_n] \\
&\quad\ (\texttt{dmap f g } [\texttt{y}_1, ..., \texttt{y}_m] \\
&\quad\quad (\texttt{dmap (g.k) (g.h) } [\texttt{x}_1, ..., \texttt{x}_n]\ [\texttt{z}_1, ..., \texttt{z}_l]))
\end{aligned}$$

The *rhs* of (3) in Fig. 2 has the advantage that it is more efficient than the *lhs* in that it avoids the construction of an intermediate list. The compositions of the function parameters (`f.h`, `f.k`, `g.h`, `g.k`) may enable further optimizations. Note that the *rhs* applies `h` and `k` twice to the input `xs` which may degrade performance in case these functions are expensive to calculate and their compositions cannot be optimized.

```
dmap f g (dmap h k xs ys) zs
  = dmap  (f.h)  (f.k) xs
          (dmap f g ys
                (dmap (g.k) (g.h) xs zs))
```
(3)

Fig. 2. Composition of `dmap` over `dmap`

4 Using the Fusion Rule

When composing functions defined by `dmap`, we will make use of several simplifications of the fusion rule (3) in Fig. 2. An important case is when some of the parameters of `dmap` are initialized to fixed values, for example, to the empty list ([]) or to the null function (`#`). We will now discuss the most interesting cases.

1) It is not uncommon that the last parameter of `dmap` is initialized to the empty list (*e.g.*, see the definitions of `copy`, `rev`, `map`, `mir` in Fig. 1), and we can immediately simplify the fusion rule:

```
dmap f g (dmap h k xs []) zs
= dmap (f.h) (f.k) xs (dmap (g.k) (g.h) xs zs)
```
(4)

2) Another important simplification occurs when function parameters are initialized to the null function (`#`). For example, take the case when two of the four function parameters are the null function. Then we obtain the following four fusion rules. They tell us that an entire class of two-pass compositions can be simplified into a single pass. Note that none of the list parameters appears twice on the *rhs*, and that `h` and `k` are computed only once for the elements of `xs`.

```
dmap f # (dmap h # xs ys) zs = dmap (f.h) # xs (dmap f # ys zs)      (5)
dmap f # (dmap # k xs ys) zs = dmap # (f.k) xs (dmap f # ys zs)      (6)
dmap # g (dmap h # xs ys) zs = dmap # g ys (dmap # (g.h) xs zs)      (7)
dmap # g (dmap # k xs ys) zs = dmap # g ys (dmap (g.k) # xs zs)      (8)
```

3) Both cases may occur at the same time, which drastically simplifies the fusion rule. Take the four equations (5– 8) and let `ys` = [ ]. Then we obtain four simple fusion rules (after renaming the function parameters):

```
dmap f # (dmap g # xs []) zs = (dmap (f.g) # xs zs)      (9)
dmap f # (dmap # g xs []) zs = (dmap # (f.g) xs zs)      (10)
dmap # f (dmap g # xs []) zs = (dmap # (f.g) xs zs)      (11)
dmap # f (dmap # g xs []) zs = (dmap (f.g) # xs zs)      (12)
```

The reader may have spotted that the well-known composition of map (2) is an instance of (9) where `zs` = [ ], and recalling equality (1) between `map` and `dmap`.

4) Two algebraic properties are handy for further simplifications: (i) The identity function `id` works as a *unit element* for all functions `f`: `id.f` = `f.id` = `f`; (ii) the special symbol `#` is a *zero element* for all functions `f`: `#.f` = `f.#` = `#` . In fact, we used the last property to obtain rules (5–12).

Using the fusion rule and the simplifications above, we can enjoy several interesting optimizations. We show some of them using the programs defined in Fig. 1. When accumulation is involved, deforestation techniques fail to optimize the programs.

4.1 Example: `app` over `app`

This is a classical example of deforestation that composes two `app` functions. In our framework `app` is defined as `app xs ys` = `dmap id # xs ys`. Rule (5) applies directly:

```
app (app xs ys) zs = dmap id # (dmap id # xs ys) zs
                   = dmap (id.id) # xs (dmap id # ys zs)   — (5)
                   = dmap id # xs (dmap id # ys zs))       — simplification
```

This result can be translated, for readability, to `app xs (app ys zs)`. This example shows that our idea achieves the same transformation as deforestation if functions are defined with `dmap`.

4.2 Example: `rev` over `rev`

This example cannot be fused by ordinary deforestation, but is straightforward with our fusion rule. `rev xs` is defined as `dmap # id xs []`. Since `rev` has an operation only on its accumulator, (12) applies for this composition.

```
rev (rev xs) = dmap # id (dmap # id xs []) []
             = dmap (id.id) # xs []          — (12)
             = dmap id # xs []               — simplification
```

We can easily derive the equation `rev (rev xs)` = `copy xs`. As in deforestation, the optimized program does not produce an intermediate list.

4.3 Example: `mir` over `mir`

This example fused by the powerful attribute grammar method, is also in our scope. `mir xs` is defined as `dmap id id xs []`. The original rule (3) is required.

```
mir (mir xs) = dmap id id (dmap id id xs []) []
             = dmap (id.id) (id.id) xs (dmap id id []
                                (dmap (id.id) (id.id) xs []))  — (3)
             = dmap id id xs (dmap id id xs []))               — simplification
```

This can be folded into `mir' xs (mir xs)`, where `mir' xs ys` = `dmap id id xs ys`. The intermediate result is eliminated by this composition and runs efficiently.

4.4 More Examples

As previous examples show, the transformation is straightforward. Table 1 summarizes the results of other transformations. They show that our method easily derives `rev (mir xs) = mir xs`. As for composition of `fcb` and `fba`, while the original definitions are iterative and use an accumulator, the resulting definition is recursive. This is a tradeoff of the transformation: while the original program traverses the list twice in an iterative manner, the transformed program traverses the list only once, but does so in a recursive manner.

Table 1. Transformation results

source program	result of composition	note
`rev (mir xs)`	`dmap id id xs []`	equal to `mir xs`
`map f (app xs ys)`	`dmap f # xs (dmap f # ys [])`	
`map f (rev xs)`	`dmap # f xs []`	equal to `rev (map f xs)`
`dmap f g (rev xs) ys`	`dmap # f xs (dmap g # xs ys)`	
`fcb (fba xs)`	`dmap h # xs []`, where `h a = c`, `h b = c`, `h c = c`	

4.5 Termination

All transformation rules preserve the semantics under a call-by-value semantics, and under a lazy semantics if we assume the length of the input lists is finite. When infinite lists are involved under a lazy semantics, the *rhs* of an equation may terminate and deliver a result, while the *lhs* does not.

Table 2 illustrates this for the rules (5–8) where, for simplicity, we assume the function parameters (`f`, `g`, `h`, `k`) are `id`. There are two cases in which the outputs differ (marked with '!'). While the *lhs* whose outer `dmap` has the second function parameter accumulates the result traversing an input lists of infinite length, the *rhs* produces a partial result.

As an example consider the case `rev (rev (x : ⊥))`. When `rev` is defined by `dmap` where the first function parameter is absent (Fig. 1), fusion rule (12) applies. While the *lhs* of (12) does not return a result, namely ⊥, the *rhs*, a function equivalent to `copy (x : ⊥)`, reduces to head-normal form `x : ⊥`.

5 Related Work

This section gives a comparison to related works on composing functions.

Table 2. Nested `dmap`s (*lhs*) and their fused version (*rhs*) under lazy evaluation

	input		output		
	`xs`	`ys`	*lhs*	*rhs*	Remark
(5)	`x1` : ⊥	[`y1`, `y2`]	`x1` : ⊥	`x1` : ⊥	
	[`x1`, `x2`]	`y1` : ⊥	`x1` : `x2` : `y1` : ⊥	`x1` : `x2` : `y1` : ⊥	
(6)	`x1` : ⊥	[`y1`, `y2`]	⊥	⊥	
	[`x1`, `x2`]	`y1` : ⊥	`x2` : `x1` : `y1` : ⊥	`x2` : `x1` : `y1` : ⊥	
(7)	`x1` : ⊥	[`y1`, `y2`]	⊥	`y2` : `y1` : ⊥	!
	[`x1`, `x2`]	`y1` : ⊥	⊥	⊥	
(8)	`x1` : ⊥	[`y1`, `y2`]	⊥	`y2` : `y1` : `x1` : ⊥	!
	[`x1`, `x2`]	`y1` : ⊥	⊥	⊥	

5.1 Attribute Grammar Composition

We extended `map` with an accumulating parameter `ys` and a function `g` computing its elements. Function `g` is similar to function `f`, which is also applied to the elements of `xs`, except that `g` computes information flowing downwards to the end of the list, while `f` computes information flowing upwards as the list is traversed. In terms of attribute grammars, we speak of computing *inherited* and *synthesized attributes*, respectively. At first, attribute grammars are very loosely connected to functional programs, but it was already pointed out [11] that they can be seen as special functional programs. In fact, restricted functional programs can be translated into attribute grammars and vice versa.

Moreover, since there are also composition techniques for attribute grammars, we can use these techniques to eliminate intermediate results of functional programs. In [5, 7, 8] powerful methods for composing attribute grammars are presented. Indeed, function `dmap` represents a class of attributed tree transducers [6] restricted to one inherited and one synthesized attribute, and lists as data structure. The computation of both attributes is 'regular' in the sense that functions `f` and `g` are applied to each element of the input list, and the computation of both attributes is strictly separated. Thus, `dmap` falls into the class of *single-use* attributed tree transducers. The operations on inherited or synthesized attribute can be present or not (expressed by the null function).

The fusion rule presented here captures some of these ideas, but for a selected case. Two attributes of the same kind are fused into a synthesized attribute, and the rest turn into an inherited attribute. The four fusion rules in (9–12) display these properties. The main advantages of our fusion rule are the simplicity of transformation and the elegant elimination of copy rules. Attribute grammars composition is a powerful technique, and it can handle wider range of problems than ordinary deforestation or our techniques do. In order to enjoy this benefit, however, we have first to translate functions into attribute grammars, then compose, and finally translate back again into the functional world. As another drawback, attribute grammars sometimes suffer from copy rules [4] which just

pass the information intact to other attributes. With our composition rule, the handling and elimination of copy rules are straightforward.

A related technique [15], based on the composition of attribute grammars but without transforming functions into the attribute world, can also compose functions for which ordinary deforestation fails. It realizes the composition of producers with accumulation and consumers without accumulation. It differs from our idea in that the producer can have more than one accumulating parameter, but this comes at the expense of no accumulating parameter in the consumer.

5.2 Using Abstraction

Another way is to use abstraction of expressions, namely *lambda abstraction* or *continuation*. Lambda abstraction enables a subexpression including an accumulating parameter to be taken out of an expression and it often helps program optimization, like deriving accumulative reverse from naive reverse which uses append, or tupling to avoid multiple traversals over inputs (*e.g.*, [19, 25]; the latter also points out the relation of continuation or accumulation to attribute grammars). Lambda abstraction and composition strategy is presented in [18]. However, it does not show how the composition of two nested expressions with lambda abstraction is treated. It seems they will not always be easy to compose. To examine this, let us examine shortcut deforestation.

In shortcut deforestation the function `dmap` can be defined with `foldr` with the help of continuation. For example,

$$\texttt{dmap f g xs ys} = (\ \texttt{foldr}\ (\backslash\ \texttt{a e b} \rightarrow (\texttt{f a}) : \texttt{e}\ ((\texttt{g a}) : \texttt{b})\ )\ \texttt{id xs}\ )\ \texttt{ys}$$

holds. Through properly abstracting constructors with `build`, function fusion is possible by the `foldr/build` rule (or `foldr/augment` if need be).

This fusion, which replaces occurrences of (abstracted) constructors in the input expression with the parameters of `foldr`, however, does not produce the simple result as `dmap` derives when continuations are involved. The benefit of `dmap` fusion is its simplicity.

6 Conclusion and Future Work

We presented the composition of a selected, restricted class of functional programs using `dmap`s, which enables the deforestation of a certain kind of accumulating parameters. This idea is not only related to the theory of attribute grammars, as explained above, but also to `foldr/build`. Both `foldr/build` and `dmap` share the same advantages compared to unfold/fold-transformation techniques [1]. There is no risk of performing an infinite transformation and there is no need to keep track of previous function calls to perform fold and generalization steps, since a single, local rule is sufficient.

The composition of two `dmap` functions both wins and loses in comparison with `foldr/build`. We win because functions can be deforested by `dmap`-fusion

which cannot be optimized by `foldr/build` (*e.g.*, `mir` over `mir`); we lose because not all functions definable as `foldr/build` can be expressed as `dmap`. One example is `filter`. This does not fit into the format of `dmap`.

It is an open question how to transform programs in a `dmap` representation automatically. The same problem arise in shortcut deforestation and there are some suggestions [17, 3] how to derive the desired representation by `foldr` and `build`. The latter automatically realizes function fusion through type inference without explicitly introducing `build`. This idea does not apply to our framework, however: while `foldr` and `build` only matters *where* proper constructors appear, `dmap` defines *how* constructors are produced.

It is well-known [26] that an optimization can enable certain optimizations, while disabling others. It is a task for future work to determine the order of deforestation-like optimizations as to achieve the best overall optimization effect. Since `dmap` can be expressed using other, simpler list processing functions, it will be an interesting task to redefine functions in the manner of 'concatenate vanishes' [23, 20]. Finally, we note that the laws for `dmap` can be proven using the standard properties of append, reverse, and map.

We have not yet taken the detailed benchmarks of the composition. We expect better performance by composition in most examples, though there are two possible cases of degradation. The function `dmap` works under both of eager and lazy evaluation. As the first case of degradation, it is possible that two `dmap`s only with an accumulative function parameter (*e.g.*, `rev`) run faster than their composed result in eager evaluation. This is because *dmap* benefits from running as tail recursion. The other case of degradation occurs, as was explained in Sect. 3, when expensive function parameters are copied by `dmap` composition.

Using `dmap`, compositions of functions with recursion and accumulating parameters become quite simple and easy to optimize. As is demonstrated in Sect. 5.2, indeed `dmap` can also be defined in the framework of `foldr` and `build`, but they are not fused in a simpler form under shortcut deforestation. We have formalized an idea of function fusion by introducing an accumulating variant of `foldr` and giving its fusion rule [12]. We hope this will enlarge the applicability of function composition in the presence of accumulating parameters.

Acknowledgments The authors would like to thank Alberto Pettorossi and the anonymous reviewers for providing many helpful and detailed comments.

References

1. R. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1): 44–67, 1977.
2. W.-N. Chin. Safe fusion of functional expressions II: Further improvements. *J. Funct. Prog.*, 4(4): 515–555, 1994.
3. O. Chitil. Type inference builds a short cut to deforestation. *Proc. of the International Conference on Functional Programming*, 249–260, ACM Press 1999.
4. L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: A deforestation case-study. In G. Nadathur (ed.), *Principles and Practice of Declarative Programming*, LNCS 1702, 360–377, Springer-Verlag 1999.

5. Z. Fülöp. On attributed tree transducers. *Acta Cybern.*, 5: 261–279, 1981.
6. Z. Fülöp and H. Vogler. *Syntax-directed semantics: Formal models based on tree transducers*. EATCS Monographs Theoret. Comp. Sci., Springer-Verlag 1998.
7. H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Sci. Comput. Prog.*, 3: 223–278, 1983.
8. R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Inf.*, 25(4): 355–423, 1988.
9. A. Gill, J. Launchbury, and S. Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, 223–232, ACM Press 1993.
10. Z. Hu, H. Iwasaki, and M. Takeichi. Calculating accumulations. *New Gener. Comput.*, 17(2): 153–173, 1999.
11. T. Johnsson. Attribute grammars as a functional programming paradigm. In G. Kahn (ed.), *Functional Programming Languages and Computer Architecture*, LNCS 274, 154–173, Springer-Verlag 1987.
12. K. Kakehi, R. Glück, and Y. Futamura. An Extension of Shortcut Deforestation for Accumulative List Folding. *IEICE Trans. on Inf. & Syst.*, to appear.
13. A. Kühnemann. *Berechnungsstärken von Teilklassen primitiv-rekursiver Programmschemata.* Ph.D thesis, Technical University of Dresden, 1997. In German.
14. A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and R. Pamanujan (eds.), *Foundations of Software Technology and Theoretical Computer Science*, LNCS 1530, 146–157, Springer-Verlag 1998.
15. A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In A. Middeldorp and T. Sato (eds.), *Functional and Logic Programming*, LNCS 1722, 114–130, Springer-Verlag 1999.
16. A. Kühnemann, R. Glück, and K. Kakehi. Relating accumulative and non-accumulative functional programs. In A. Middeldorp (ed.), *Rewriting Techniques and Applications*, LNCS 2051, 154–168, Springer-Verlag 2001.
17. J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. International Conference on Functional Programming Languages and Computer Architecture*, 314–323, ACM Press 1995.
18. A. Pettorossi. Program development using lambda abstraction. In K. Nori (ed.), *Foundations of Software Technology and Theoretical Computer Science*, LNCS 287, 420–434, Springer-Verlag 1987.
19. A. Pettorossi and A. Skowron. The lambda abstraction strategy for program derivation. *Fund. Inform.*, 12(4): 541–561, 1989.
20. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM TOPLAS*, 18(2): 175–234, 1996.
21. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *J. Funct. Prog.*, 6(6): 811–838, 1996.
22. V. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3): 292–325, 1986.
23. P. Wadler. *The concatenate vanishes.* Internal report, Dept. of Comp. Sci., Univ. of Glasgow, 1987.
24. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoret. Comput. Sci.*, 73(2): 231–248, 1990.
25. M. Wand. Continuation-based program transformation strategies. *J. ACM*, 27(1): 164–180, 1980.
26. D. Whitfield and M. Soffa. An approach for exploring code-improving transformations. *ACM TOPLAS*, 19(6): 1053–1084, 1997.