On Deforesting Parameters of Accumulating Maps (Extended Abstract)

Kazuhiko Kakehi[1]***, Robert Glück[2]†, and Yoshihiko Futamura[2]

[1] Graduate School of Science and Engineering, Waseda University
[2] Institute for Software Production Technology, Waseda Univeristy
Okubo 3-4-1 59-406, Shinjuku, Tokyo, 169-8555, JAPAN
Tel & Fax: +81-(0)3-5286-3339
{kaz,futamura}@futamura.info.waseda.ac.jp, glueck@acm.org

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 [20] is a well-known program transformation technique which can eliminate intermediate data structures that are passed between functions. *Short-cut 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 higher-order functions. Compared to deforestation, shortcut deforestation has the advantages 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).

One of the weaknesses of such transformations is the inability to deforest programs using accumulating parameters [2] properly. This is not just a deficiency of these techniques, for other program transformers suffer from the same

*** Research fellow of the Japan Society for the Promotion of Science
† PRESTO, Japan Science and Technology Corporation (JST); on leave from DIKU, Dept. of Computer Science, University of Copenhagen

weakness (*e.g.*, *supercompilation* [18], *calculational methods* [10]). 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. 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.

Clearly, this situation is not satisfactory for program composition. While accumulating parameters often let many functions be naturally expressed or 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 it is a generalization of the map function, our transformations will not 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) which shortcut deforestation cannot do. This is due to the restriction to map-style list producers and consumers. In fact, the laws for `dmap` show some interesting combinatorics.

Recently, composition methods using results of *attribute grammars* have been proposed [4, 12, 13, 15] which allow transformations not possible by ordinary deforestation. The problem of accumulator composition studied in this note 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 may help to bridge the gap between the functional and the attribute world.

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

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: by an additional function parameter `g` building the elements of an accumulator and a list parameter `ys` which terminates the list constructed by applying `f` and `g` to the elements of `xs`. Informally, `dmap` is defined by

$$\mathtt{dmap\ f\ g}\ [\mathtt{x}_1, ..., \mathtt{x}_n]\,[\mathtt{y}_1, ..., \mathtt{y}_m] = [\mathtt{f\ x}_1, ..., \mathtt{f\ x}_n,\ \mathtt{g\ x}_n, ..., \mathtt{g\ x}_1,\ \mathtt{y}_1, ..., \mathtt{y}_m]\ .$$

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

Fig. 1. Definition of `dmap` and examples

The application of `g` to $[x_1, ..., x_n]$ in reverse order can be implemented by an accumulating parameter. A definition of `dmap` is given in Figure 1. In addition, we require that the corresponding sublist is dropped from the result 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 also shows several familiar list processing functions defined in terms of `dmap` (*e.g.*, append, reverse, map). Here `dmap` serves as a primitive for the recursion scheme. Functions `rev`, `mir`, and `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 `dmap` presented here enjoys syntactic sugar of `#`, and we assume `dmap` with `#` are simplified into a suitable form before execution in order not to examine function parameters or produce unused lists.

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.

$$\texttt{dmap f g (dmap h k xs ys) zs} = \texttt{dmap (f.h) (f.k) xs (dmap f g ys (dmap (g.k) (g.h) xs zs))} \tag{3}$$

Fig. 2. Composition of dmap over dmap

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. The fusion rule for dmap, which has an accumulating parameter ys, is shown in Figure 2. 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 the fusion rule can be justified using the definition of dmap and list properties:

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

The *rhs* of (3) is more efficient than the *lhs* in terms of avoiding the construction of an intermediate list as well as using only one traversal over the elements of xs. The compositions of the function parameters on the *rhs* may enable further optimizations (of f.h, f.k, g.h, g.k). Below we will demonstrate this by several examples. 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 the compositions cannot be optimized.

A word about the underlying semantics. The fusion rule (3) always holds under eager evaluation, and it also holds under lazy evaluation in case all lists are finite. When infinite lists are involved in lazy evaluation, the *rhs* of the equation may terminate and deliver a result, while the *lhs* does not. For example, under lazy evaluation composition rev (rev x : ⊥) does not produce a result, while copy x:⊥ reduces to head-normal form.

4 Using the Fusion Rule

When composing functions defined by `dmap`, we will make use of several simplifications of the fusion rule (3). One is the case when some of the parameters of `dmap` are initialized to fixed values, for example, to an empty list or the null function. In fact, it is not uncommon that the second parameter is initialized to `ys` = [ ] (*e.g.*, `rev`, `map`, `mir` in Figure 1). We can immediately simplify the fusion rule:

$$\begin{aligned}&\texttt{dmap f g (dmap h k xs [ ]) zs}\\&= \texttt{dmap (f.h) (f.k) xs (dmap (g.k) (g.h) xs zs)}\end{aligned}$$

Another simplification occurs when function parameters are initialized to the null function (`#`). For example, take the case when two of the function parameters are the null function. Then we obtain the following four equations which 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*. Thus `h` and `k` are never computed more than once for the elements of `xs`.

$$\texttt{dmap f \# (dmap h \# xs ys) zs} = \texttt{dmap (f.h) \# xs (dmap f \# ys zs)} \qquad (4)$$

$$\texttt{dmap f \# (dmap \# k xs ys) zs} = \texttt{dmap \# (f.k) xs (dmap f \# ys zs)} \qquad (5)$$

$$\texttt{dmap \# g (dmap h \# xs ys) zs} = \texttt{dmap \# g ys (dmap \# (g.h) xs zs)} \qquad (6)$$

$$\texttt{dmap \# g (dmap \# k xs ys) zs} = \texttt{dmap \# g ys (dmap (g.k) \# xs zs)} \qquad (7)$$

A combination of both situations may occur, which drastically simplifies the composition rule. Take the four equations from above, and let `ys` = [ ] . Then we obtain four cases (and after renaming the function parameters).

$$\texttt{dmap f \# (dmap g \# xs [ ]) zs} = \texttt{(dmap (f.g) \# xs zs)} \qquad (8)$$

$$\texttt{dmap f \# (dmap \# g xs [ ]) zs} = \texttt{(dmap \# (f.g) xs zs)} \qquad (9)$$

$$\texttt{dmap \# f (dmap g \# xs [ ]) zs} = \texttt{(dmap \# (f.g) xs zs)} \qquad (10)$$

$$\texttt{dmap \# f (dmap \# g xs [ ]) zs} = \texttt{(dmap (f.g) \# xs zs)} \qquad (11)$$

(The reader may have spotted that (2) is an instance of (8) where `zs` = [ ] when recalling (1) for `map` and `dmap`.)

Finally, two algebraic properties are handy for further simplifications. The identity function `id` works as a unit element for all functions `f`: `id.f` = `f.id` = `f`; the special symbol `#` is a zero element for all functions `f`: `#.f` = `f.#` = `#` . In fact, we used the last property for the simplifications in (4)–(11).

Using the fusion rule and its instances, we can enjoy several interesting optimizations. We show some of them using the examples defined in Figure 1. When accumulation is involved, current program composition techniques like deforestation fail to derive them.

4.1 `app` over `app`

This is a classical example of deforestation which tries to compose two `app` functions. `app` is defined in our framework as `app xs ys` = `dmap id # xs ys`. The rule (4) applies here:

$$\begin{aligned}
\texttt{app (app xs ys) zs} &= \texttt{dmap id \# (dmap id \# xs ys) zs} \\
&= \texttt{dmap (id.id) \# xs (dmap id \# ys zs)} && \text{—(4)} \\
&= \texttt{dmap id \# xs (dmap id \# ys zs))} && \text{—}\textit{simplification}
\end{aligned}$$

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 `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, (11) applies for this composition.

$$\begin{aligned}
\texttt{rev (rev xs)} &= \texttt{dmap \# id (dmap \# id xs [ ]) [ ]} \\
&= \texttt{dmap (id.id) \# xs [ ]} && \text{—(11)} \\
&= \texttt{dmap id \# xs [ ]} && \text{—}\textit{simplification}
\end{aligned}$$

By our simple rule, we have easily derived the equation `rev (rev xs)` = `copy xs`. The obtained program does not produce any intermediate result.

4.3 `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.

$$\begin{aligned}
\texttt{mir (mir xs)} &= \texttt{dmap id id (dmap id id xs [ ]) [ ]} \\
&= \texttt{dmap (id.id) (id.id) xs (dmap id id [ ]} \\
&\qquad\qquad \texttt{(dmap (id.id) (id.id) xs [ ]))} && \text{—(3)} \\
&= \texttt{dmap id id xs (dmap id id xs [ ]))} && \text{—}\textit{simplification}
\end{aligned}$$

This can be folded in to `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 Other examples

As previous examples demonstrate, the transformation process is simple and easy. Table 1 shows the results of other transformations. As the table shows, our method easily derives `rev (mir xs)` = `mir xs`. As for composition of `fcb` and `fba`, while the original definitions are accumulative, the resulting definition is executed in a recursive manner. This is a tradeoff of transformation: the original program traverses the list twice with creating an intermediate results, beneficially in an iterative manner; the derived result traverses the list only once without any intermediate data, but 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`	

5 Relation to Attribute Grammars

We extended the functionality of `map` by introducing an accumulating parameter `ys` and a function `g` computing its elements, just in the same way as function `f` is applied to the elements of the input list `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 can speak of *inherited* and *synthesized attributes*, respectively. Such a relation between functional programming languages and attribute grammars has been pointed out, [11] for the notable example.

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 the use of 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 instances in (8)–(11) 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 [14], 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.

6 Conclusion and Future Work

We presented the composition of a selected, restricted class of functional programs using `dmaps`, which enables the deforestation of a certain kind of accumulating parameters. This idea is not only related to 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` (*e.g.*, `filter`). It is an open problem how to transform programs in a `dmap` representation automatically. In this respect, shortcut deforestation has some works which aim to derive the desired form of `foldr` and `build` [16, 3].

It is well-known [21] that an optimization can enable certain optimizations, while disabling others. It is an open task for future work how to order deforestation-like optimizations in order to achieve the best overall 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' [19, 17]. Finally, we note that the laws for `dmap` can be proven using the standard properties of append, reverse, and map.

Using `dmap`, compositions of functions with recursion and accumulating parameters become quite simple and easy to optimize. As is demonstrated in shortcut deforestation, `map` can also be defined in the framework of `foldr` and `build` [9]. We are now formalizing the idea of `dmap` to fit in this framework, by introducing an accumulating variant of `foldr`. This may help to bridge the gap between the functional and the attribute world, and enable further optimizations.

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. ICFP*, 249–260, 1999.
4. L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: A deforestation case-study. In G. Nadathur, ed., *Proc. PPDP*, LNCS vol. 1702, 360–377, Paris, France, 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.* Springer-Verlag, Berlin, 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. FPCA*, 223–232, Copenhagen, Denmark, 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 *Proc. FPCA*, 154–173, Portland, Oregon, USA, 1987.
12. A. Kühnemann. *Berechnungsstärken von Teilklassen primitiv-rekursiver Programmschemata.* Ph.D thesis, Technical University of Dresden, 1997. In German.
13. A. Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and R. Pamanujan, eds., *Proc. FSTTCS*, LNCS vol. 1530, 146–157, Chennai, India, 1998.
14. A. Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In A. Middeldorp, T. Sato, eds., *Proc. FLOPS*, LNCS vol. 1722, 114–130, Tsukuba, Japan, 1999.
15. A. Kühnemann, R. Glück, and K. Kakehi. Relating accumulative and non-accumulative functional programs. In A. Middeldorp, ed., *Proc. RTA*, LNCS vol. 2051, 154–168, Springer-Verlag 2001.
16. J. Launchbury, and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proc. FPLCA*, 314–323, 1995.
17. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM TOPLAS*, 18(2): 175–234, 1996.
18. V. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3): 292–325, 1986.
19. P. Wadler. *The concatenate vanishes.* Internal report, Dept. of Comp. Sci., Univ. of Glasgow, 1987.
20. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoret. Comput. Sci.*, 73(2): 231–248, 1990.
21. D. Whitfield, and M. Soffa. An approach for exploring code-improving transformations. *ACM TOPLAS*, 19(6): 1053–1084, 1997.