Cheap Deforestation in Practice: An Optimiser for Haskell

Andrew J Gill and Simon L Peyton Jones[a]

[a] Department of Computing Science, University of Glasgow, Scotland, UK, G12 8QQ.
Email: {andy,simonpj}@dcs.glasgow.ac.uk

We present a simple, automatic transformation — the `foldr/build` transformation — which successfully removes many intermediate lists from programs written in non-strict functional programming languages. While the idea is simple and elegant, it turns out that some care is needed in the compiler to set up the right conditions for the `foldr/build` transformation to be applicable. We report on this practical experience, and present results which quantify the benefits that can in practice be achieved.

Keyword Codes: D.1.1; I.1.4
Keywords: Programming Techniques, Applicative (Functional) Programming; Algebraic Manipulation, Applications

1. INTRODUCTION

Lazy functional programs tend to contain many *intermediate* data structures; that is, structures that are used for internal computation purposes, but form no part of the final result. For example, consider an expression to compute the sum of the squares of the numbers 1 to n, written in the lazy functional language Haskell[2].

```
sum [ i * i | i <- [1..n]]
```

Two intermediate lists are constructed during the computation of the final result. The expression `[1..n]` produces an intermediate list that is consumed by the list comprehension; in turn the latter produces a second intermediate list that is consumed by `sum`. This paper is about a transformation that can be used to removes such intermediate lists.

Deforestation is the process of automatically removing intermediate data structures[6]. In [1] we proposed a simple, automatic deforestation technique, which we call `foldr/build` deforestation. Section 2 explains the basis of our deforestation. In Section 3 we address some practical issues raised by our implementation of `foldr/build` deforestation inside the Glasgow Haskell compiler. Section 4 gives some performance results from using our deforestation. Finally Section 5 explains how we intend to expand the scope of our deforestation.

2. BACKGROUND: THE `foldr/build` TRANSFORMATION

In this section we give a brief summary of our deforestation technique (see [1] for full details). We achieve deforestation by expressing the production and consumption of lists in a manner that allows a correctness preserving transformation to remove redundant lists. There are three conceptual stages:

2.1. Express list consumption using `foldr`.

The first step is to *standardise the way in which lists are consumed*, by expressing list-consuming functions in terms of a single function, `foldr`. The function `foldr` replaces all the *cons* in a list with its first argument, and the terminating *nil* with its second argument. The Haskell definition for `foldr` is:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z (x:xs) = f x (foldr f z xs)
foldr f z [ ]    = z
```

Currently we express many of the prelude functions (`map, filter, array,` etc) as instances of `foldr`, and require our compiler to inline these definitions. For example, here is the definition of `sum` written in terms of `foldr`.

```
sum xs = foldr (+) 0 xs
```

We also use a new translation scheme for list comprehensions, allowing them perform list consumption using `foldr`.

Later we hope to be able to derive automatically a `foldr` based definition of a list consumer from a recursive definition. Some progress has been made in this area of our current research.

2.2. Express list construction in terms of a new function, `build`.

The second step is to *standardise the way in which lists are produced*. We abstract the constructors inside the list using the function `build`, defined thus:

```
build g = g (:) [ ]
```

For example, `build` can be used to express the abstraction of the constructors inside the list `[1,2,3]` like this:

```
build (\ c n -> c 1 (c 2 (c 3 n)))
```

In reality, many list-producing expressions can be written in terms of `build`. All constant lists, enumerations, and list comprehensions are expressed in terms of `build` when we remove the syntactical sugar in Haskell. We also express list-producing prelude functions in terms of `build`, and inline them at their call sites.

For example, the standard function `map` can be written like this:

```
map f xs = build (\ c n -> foldr (\ x ys -> c (f x) ys) n xs)
```

2.3. Lists that meet both these criteria (2.1 and 2.2) are removed.

If we consume our lists using `foldr`, and build our lists using `build`, we can exploit the following rule. For any `f`,`z`, and `g`

```
foldr f z (build g) = g f z
```

provided the left-hand side of the rule is well-typed. We call this rule the `foldr/build` rule. The motivation behind the development of this rule is to allow compiler optimisation passes to eliminate intermediate list structures, with each use of the `foldr/build` rule eliminating a single intermediate list.

The `foldr/build` rule is only valid if `g` really does use the arguments it is given to construct its result. In can be shown that this is the case if `g` has the type

$$\forall\beta.\ (A \to \beta \to \beta) \to \beta \to \beta$$

for some fixed type A[1]. So we give `build` the type

$$\forall\alpha.\ (\forall\beta.\ (\alpha \to \beta \to \beta) \to \beta \to \beta)\ \to [\alpha]$$

which forces any argument to `build` to be well behaved. Using this (non-Hindley-Milner) type requires some modifications to the type checker to handle rank-2 polymorphism[3].

3. OUR IMPLEMENTATION

Our current implementation works like this:

- We provide definitions for prelude functions and de-sugaring that use and expose instances of `foldr` and `build`. We do not inline the definitions of `foldr` and `build` at this point, because the `foldr/build` rule needs to see `foldr` and `build`.
- We now run a general purpose program simplifier, to which we have added our `foldr/build` rule.
- We *now* inline the definitions of `foldr` and `build`, giving us efficient list processing programs.

When we first implemented `foldr/build` deforestation, we found that it did not perform as we expected. This section explains what had gone wrong, and how we fixed it.

3.1. Pushing `foldr` through `let` and `case`

The `foldr/build` rule was not always occurring, simply because `foldr` and `build` were not in the right positions for the rule to work. It turned out that auxiliary transformations were necessary to bring together `foldr` and `build`, allowing the `foldr/build` rule to work.

Consider this example:

```
foldr f z (let v = fib 20
           in build (\ c n -> ...))
```

The `let` is getting in the way of the `foldr/build` rule. To allow deforestation, we have to perform an extra transformation. Because `foldr` is strict in its third argument, this transformation is valid:

```
foldr f z (let <defs>       let <defs>
           in <expr>)   ⇒   in foldr f z <expr>
```

In the above example, the `foldr` and `build` are brought together successfully by this transformation. An equivalent transformation is valid for *any* strict argument of *any* function, and using it saves work, so it is a useful addition to our compiler, irrespective of the presence for `foldr/build` deforestation. A similar transformation is needed for `case`.

3.2. Linearity

Consider the expression:

```
map f (map g xs)
```

This transforms into the expression:

```
let    {- (map g xs) -}
  lst = build (\ c n -> ...)
in  {- map f -}
  build (\ c n -> foldr (\ x ys -> c (f x) ys) n lst)
```

We can see that `lst` is an intermediate list, but the `foldr/build` rule can not take effect, because `foldr` and `build` are separated. We need to inline the right hand side of `lst`. Unfortunately this sort of inlining is, in general, not safe. Consider inlining the right hand side of `v` in this expression:

```
let v = fib 20
    f = \ w -> w + v
in (f 1) + (f 2)
```

In this example `v` only occurs once, but inlining would introduce re-computation problems. The transformed program would compute `fib 20` twice!

```
let f = \ w -> w + fib 20
in (f 1) + (f 2)
```

The problem is that, in general, inlining a redex through a lambda loses laziness. However, in the case we are interested in it is perfectly safe to inline `lst` in the `map f (map g xs)` example above, because:

- `build` only uses (enters) its argument once.
- When `build` uses its argument, it always applies two items to it.

Exploiting these properties when considering the suitability of local definitions for inlining is vital for deforesting many examples.

4. RESULTS

Measurements of the effectiveness of `foldr/build` deforestation are given in Table 1. The first set of benchmarks are tests we expect to get good results for. All these benchmarks make good use of the combinator style of programming and intermediate lists. We can see that we do indeed get a significant performance increase.

The second set of benchmarks are taken from the real and spectral set of programs in the nofib benchmarking suite[4]. These programs vary in size from a few dozen lines up to 10,000 lines. Although, as would be expected, the performance increase is not as great as the first group, it is still substantial, considering the relative simplicity of our optimisation. In more that half the benchmarks `foldr/build` deforestation decrements the total heap usage, as well as making noticeable runtime improvements.

Table 1
Program speedups using `foldr/build` deforestation

	Original		With `foldr/build` rule	
Benchmark	Time (s)	Heap Usage (M)	Time (s)	Heap Usage (M)
array creation	96.8	772.4	25.5	325.5
array creation (2D)	24.8	132.4	18.6	113.2
word matching	5.5	20.3	3.9	8.6
pancake sort	28.6	180.3	15.7	102.0
queens	16.3	91.6	9.3	35.0
word search puzzle	5.4	28.5	3.5	15.2
anna	13.5	34.2	12.8	30.0
boyer	5.0	30.1	4.8	29.0
cichelli	13.2	41.5	12.6	39.9
clausify	6.3	23.9	5.5	22.4
compress	31.8	165.5	31.8	165.5
fft2	12.0	57.6	10.5	40.6
fulsom	92.1	597.6	92.1	597.6
hpg	20.5	93.2	19.9	92.5
infer	8.1	20.3	7.2	10.4
life	56.9	308.4	50.8	299.6
maillist	4.5	7.5	4.5	7.4
mandel	39.1	264.0	39.1	264.0
multiplier	24.4	112.1	24.1	105.0
rsa	19.0	54.0	19.0	54.0
gamteb	27.2	177.2	27.2	177.2
hidden	91.7	476.3	87.1	448.3
primetest	96.7	205.7	96.7	205.7
rewrite	5.3	29.4	4.5	24.5
treejoin	22.8	107.0	22.8	107.0

5. CURRENT WORK

One exciting development is a way of overcoming a current restriction of `foldr/build` deforestation. At present intermediate lists are not eliminated if they are produced by one function and consumed by another function. For example, consider the program segment:

```
h = \ args -> build ( \ c n -> <expr> )
j = ... foldr f z (h args) ...
```

If we inline `h` then we would have an instance of the `foldr/build` rule, and deforestation would occur. But it may be inadvisable to inline `h` — for example, `h` might also be used elsewhere, and `<expr>` might be too large to duplicate. We need some way of *encapsulating* the list production properties of `h`, and showing this encapsulation to the `foldr`. We do this by ‘borrowing’ a technique that was previously used to exploit strictness information[5]. We split an amicable function into a worker and wrapper. The worker contains the body of the original function, while the wrapper contains the *encapsulation* of the list production. In the above case, our 'amicable' function, h, would split into:

```
h = h_wrapper
h_worker  = \ args -> \ c n -> <expr>
h_wrapper = \ args -> build (\ c n -> h_worker args c n)
```

Now we can safely inline the (small) h_wrapper at all calls of h. After this inlining, the function j would become:

```
j = ... foldr f z (build (\ c n -> h_worker args c n)) ...
```

and we have an instance of the foldr/build rule (We also have a way of using workers and wrappers to express good *consumption* across the function boundary).

This type of 'long range' foldr/build deforestation can even take place over a module boundary. It should substantially increase the applicability of the foldr/build rule.

We also intend to push foldr/build deforestation in two further directions.

- We are developing a simple method of automatically deriving foldr and build for *some* recursive functions. This should lift several of the current restrictions, including the need for the programmer to use either prelude list processors or write programs using foldr and build explicitly.

- We will experiment with foldr/build deforestation for datatypes other than lists. The deforestation has a natural extension to other datatypes.

REFERENCES

1. A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Functional Programming and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 1993. ACM SIGPLAN/SIGARCH, ACM.
2. Paul Hudak, Simon L. Peyton Jones, and Philip Wadler (Editors). Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.
3. N. J. McCracken. The typechecking of programs with implicit type structure. *Semantics of data types*, pages 301–315, June 1984.
4. W. D. Partain. The nofib benchmarking suite. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*, Workshops in Computing, Ayr, Scotland, 1992. Springer Verlag, Workshops in Computing.
5. S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In RJM Hughes, editor, *Functional Programming and Computer Architecture, Boston*, Cambridge, September 1991. LNCS 523, Springer Verlag.
6. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.