UNIVERSITY
of
GLASGOW

Department of Computing Science

Cheap Deforestation for Non-strict Functional Languages

Andrew John Gill

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computing Science at the University of Glasgow

January 1996

Cheap Deforestation for Non-strict Functional Languages

by

Andrew John Gill

A thesis submitted in partial fulfilment of
the requirements for the degree of Doctor of Philosophy
in Computing Science at the University of Glasgow

January 1996

Abstract

In functional languages intermediate data structures are often used as glue to connect separate parts of a program together. Deforestation is the process of automatically removing intermediate data structures. In this thesis we present and analyse a new approach to deforestation. This new approach is both practical and general.

We analyse in detail the problem of *list* removal rather than the more general problem of arbitrary data structure removal. This more limited scope allows a complete evaluation of the pragmatic aspects of using our deforestation technology.

We have implemented our list deforestation algorithm in the Glasgow Haskell compiler. Our implementation has allowed practical feedback. One important conclusion is that a new analysis is required to infer function arities and the linearity of lambda abstractions. This analysis renders the basic deforestation algorithm far more effective.

We give a detailed assessment of our implementation of deforestation. We measure the effectiveness of our deforestation on a suite of real application programs. We also observe the costs of our deforestation algorithm.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Simon Peyton Jones for his constant support, encouragement and guidance. John Launchbury also provided outstanding support in his role as second supervisor.

The Glasgow Functional Programming group is a very exciting team to work with. The annual workshops were especially enjoyable, and allowed the opportunity to meet the wider FP community. Working with the AQUA project brought great joy, and thanks especially to Will Partain and Jim Mattson, who answered many questions about the Glasgow Haskell compiler.

I would like to thank the members of the examination committee, Mark Jones, John O'Donnell and David Watt for providing many interesting suggestions to improve this thesis. Thanks also to Simon Marlow, Dave King and André Santos, my long-suffering officemates made my time in Glasgow as a Ph.D. student so enjoyable.

I would also like to thank John Butler from Edinburgh University, who encouraged me to consider doing research into the efficient implementation of functional languages, and Russell Green, formerly of Edinburgh University, who recommended that I consider Glasgow as a venue.

I thank my parents, who never fail to love, support or believe in me. Finally, my gratitude and love to my wife, Lindsay, for everything. Thank you.

This work was supported for three years by a grant from the EPSRC.

Andrew J. Gill
Glasgow,
January 1996

Chapter 1

Introduction

Despite widespread interest in functional languages, there are still some objections to their overall performance. This is because functional languages have a higher execution cost than imperative languages, requiring more processing power and memory.

There is a particular programming style that functional languages encourage, in which programs are expressed as a sequence of transformations on data structures. In this paradigm the programmer deliberately introduces extra intermediate data structures. Doing so allows clearer and more modular programs, but directly contributes towards the higher runtime overhead of functional programming. These data structures are often lists, and this style of programming is called *listfulness*. This dissertation is about *automatically* removing many of the intermediate lists produced when programming in the listful style, thereby reducing the cost of listful programs.

In this chapter we introduce the listful programming style and outline the contributions made by this dissertation toward the more efficient compilation of listful programs. § 1.1 discusses functional languages, focusing on non-strict functional languages. § 1.2 presents the listful style of programming, showing why it is elegant, modular, but inherently inefficient. Finally, § 1.3 explains the contributions made by this dissertation.

1.1 Functional languages

Functional languages put emphasis on the *value* of a particular computation, rather than the computation's external effects. Writing a program in a functional language involves writing function definitions, rather than specifying a sequence of actions. For example, a function that computes the sum of the squares of the numbers 1 to n, written in the functional language Haskell (Hudak, Peyton Jones, Wadler et al. 1992) could be expressed as:

```
sumSq n = sum (map square [1..n])
square x = x * x
```

In the imperative language C (Kernighan & Ritchie 1978), a function to perform the same computation might be written as:

```
int sumSq(n)
    int n;
    {
       int i,sum;
       sum = 0;
       for (i=1;i<=n;i++)
           sum += i * i;
       return(sum);
    }
```

In a C program the evaluation is specified as a sequence of actions. In a functional language the same task can be performed as a composition of functions.

Some functional languages strongly *encourage* the functional programming style while still allowing the vices of imperative languages, such as side effects, explicit sequencing, and assignment. Examples of such languages include Lisp (Steele Jr. 1984), ML (Harper, McQueen & Milner 1986) and Id (Nikhil 1988). Other functional languages actively forbid (by means of the language's semantics) any functions from having side effects, and because of this are called *pure* functional languages. Examples of pure functional languages include LML (Augustsson & Johnsson 1989), Miranda (Turner 1985) and Haskell. Most languages that disallow side effects have *non-strict* (or normal order) semantics, and use a lazy evaluation strategy. In this dissertation we are concerned with the optimisation of non-strict functional languages. Because of this we will refer to such languages as simply "functional languages", and be explicit about references to other styles of functional language.

Functional languages such as Haskell have many things to offer the programmer, including:

- **Modularity provided by Lazy Evaluation** – Functional languages use data structures as *glue* to combine modular, reusable components of code. It is not necessary to completely evaluate intermediate data structures; instead the production and consumption of data structures can be done in lock-step. The modular components can act like co-routines (Henderson 1980).

- **Reasoning** – Functional languages are easier to reason about than imperative languages. Techniques like equational reasoning (Bird & Wadler 1988) can be used, due to the absence of side effects. Absence from side effects also makes functional languages particularlv amicable for manipulation and optimisation via automatic transformations (Santos & Peyton Jones 1992, Santos 1995).
- **Parallel Execution** – Another attractive feature of functional languages is that they are especially suitable for evaluation on parallel hardware (Peyton Jones 1989, Roe 1991). As a direct consequence of lack of side effects there is no need to compute dependencies between parallel sections of a program.

There are also other important characteristics that are not actually specific to non-strict purely functional languages, such as dynamic allocation, higher order functions and strong type systems. Although all these characteristics do appear in other programming paradigms, they actively contribute, along with the above advantages, to provide a productive programming environment.

The advantages of lazy functional languages do not come for free. There are two principal costs:

- **Efficiency** – Functional languages are less efficient than languages like C. This is especially true when heavy use is made of expressive programming techniques such as infinite data structures and using lists as a glue. Significant progress has been made toward cutting the cost of using such features automaticallv during compilation (Peyton Jones 1992). However there is still scope for improvement. This dissertation makes a further contribution towards this quest for efficiency.
- **Expressing side effects** – The fundamental feature of *pure* functional languages can also be its drawback. Occasionally it appears that the only clean way of doing something is via an imperative feature. Examples include generating unique labels, performing input/output, interacting with imperative languages, and destructively updating a data structure. Many solutions have been proposed to this apparent deficiency in purely functional languages (Hudak & Sundaresh 1989, Hudak 1987, Wadler 1987*c*). They all provide some technique for emulating side effects, but without compromising purity. One proposal that has gained widespread favour with the lazy functional programming community is using monads (Wadler 1992*a*, Wadler 1992*b*).

The availability of good compilers for lazy functional languages that both generate reasonably efficient code and provide mechanisms for side effects make lazy

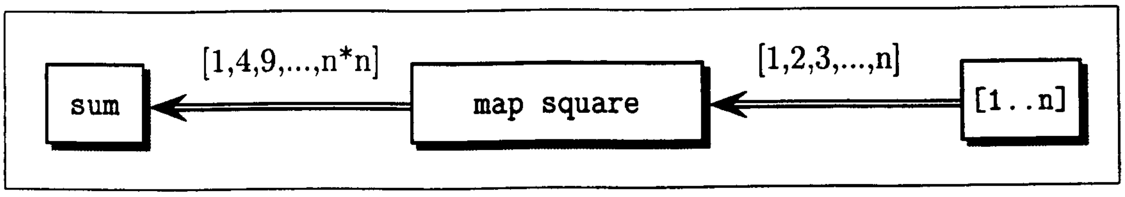

Figure 1.1: The pipeline structure of sum of squares

functional programming a viable option for the applications programmer. We extend the viability of using functional languages further, by allowing a particular class of programs to be compiled more efficiently. Haskell has become the *de-facto* standard non-strict, purely functional language in the lazy functional programming community. We use Haskell as our source language throughout, and put theory into practice inside the Glasgow Haskell compiler.

1.2 The listful style of programming

In functional languages the listful style is strongly encouraged (Hughes 1989). Listful algorithms are expressed in terms of pipelines of list transformers, glued together with intermediate lists. The listful style tends to be pervasive in the fine grain level of lazy functional programming. There are many list manipulating functions provided in the Haskell standard library (called the standard prelude). Furthermore, in Haskell (and many other functional languages) a special syntax, called list comprehensions, provides concise ways of expressing list-based manipulations. The presence of such conveniences provides further testimony of the importance of lists in functional languages.

We now look at two concrete examples of small programs, presenting implementations with and without intermediate lists.

1.2.1 Example 1: Sum of squares

Consider computing the sum of the squares of the numbers 1 to n. We compute two intermediate lists, one of which is the list 1 to n, and the second is the squares of 1 to n. Figure 1.1 gives an illustration of the pipeline structure of this example.

In Haskell we could write:

```
sumSq :: Int -> Int
sumSq n = sum (map square [1..n])
square :: Int -> Int
square x = x * x
```

It is also possible to write a function that computes the sum of the squares of the numbers 1 to n without using any intermediate lists. For example:

```
sumSq :: Int -> Int
sumSq n = sumSq' 1
 where
  sumSq' x = if x > n
             then 0
             else square x + sumSq' (x+1)
```

Although more efficient, the program is arguably less modular and harder to understand, specifically because the program no longer expresses the natural pipeline structure of this computation.

1.2.2 Example 2: Digits of a natural

As another example of listful programming, consider a function that takes a natural number and returns a list of the number's digits.

```
natural :: Int -> [Int]
natural = reverse
        . map ('mod' 10)
        . takeWhile (/= 0)
        . iterate ('div' 10)
```

This example was taken from Bird & Wadler (1988, pp 172). This time there are four components to the pipeline, with three intermediate lists connecting them. Again it is possible to re-write this problem without using intermediate lists:

```
natural :: Int -> [Int]
natural n = natural' n []
  where
    natural' :: Int -> [Int] -> [Int]
    natural' n m =
      if n /= 0
      then natural' (n 'div' 10) (n 'mod' 10:m)
      else m
```

We also observe that again the listless implementation has lost its pipeline structure and its modularity.

1.2.3 Advantages of removing intermediate lists

Programs written using intermediate lists can express solutions concisely and clearly. However such solutions can be inefficient by virtue of their key component, their intermediate lists. Using the Haskell Users Gofer System (HUGS), and taking reduction counts as an approximation of execution cost, a simple quantitative measurement can be taken for our two examples, giving an estimate of the cost of using listfulness as a programming style.

Example	Test	listful	listless
Sum of Squares	`sumSq 10`	123 reductions	73 reductions
Digits of a Natural	`natural 1234`	82 reductions	44 reductions

Though these are small examples, quite clearly a significant part of the cost of their computation was a direct consequence of using intermediate lists. The best of both worlds would be to write the clear listful program, and have the compiler automatically produce the efficient listless equivalent.

Removing an intermediate list produces *two* tangible benefits:

- The cost of constructing and deconstructing an intermediate list is avoided. As well as saving the cost of the memory accesses, not using intermediate structures puts less strain on the memory allocation and garbage collection mechanisms.
- Two components of a program that were previously separated by the opaque interface of the intermediate list are now exposed to each other. Removing the intermediate list may allow other optimising transformations, that previously could not penetrate the barrier of the intermediate list, to optimise the listless program.

This dissertation presents a technique that gives compilers for functional languages the power to automatically detect and remove many intermediate lists, and thus providing the functional language programmer with the opportunity to use listfulness *without* paying the performance cost normally associated with this style of programming.

Removal of intermediate data structures (including intermediate lists) is traditionally called deforestation (Wadler 1990). Our deforestation algorithm is considerably simpler than the Wadler style of deforestation. Because of this we call our deforestation system *cheap deforestation*.

We concentrate on the problem of *list* removal rather than the more general problem of arbitrary data structure removal. Though many of the ideas presented in this dissertation have a natural extension to other data structures, our concern was to complete an evaluation of the pragmatic aspects of list removal.

1.3 Contributions and synopsis

This dissertation explores in detail a small set of optimising transformations that allows programmers to use the listful style of programming in lazy functional languages without paying a substantial performance penalty. We make the following distinct contributions:

- This dissertation as a whole presents and analyses the first non-trivial deforestation system to be included as an active part of a production quality functional language compiler.
- In **Chapter 2** we describe a new, simple and pragmatic technique for removing intermediate lists. Previous list removal schemes were either complicated (and computationally expensive) or lacked generality. Our intermediate list removal is both automatic and practical.
- We make an interesting use of parametricity to prove our core list removal rules correct (§ 2.4).
- In **Chapter 3** we discuss some of the general pragmatics behind implementing our style of deforestation.
- We provide a new translation for list comprehensions in § 3.3, superseding the current state of art, as given in Wadler (1987*a*) and Augustsson (1987). The previous schemes had some specific optimisations concerning appending together list comprehensions. We use our general purpose list removal capabilities to accomplish the benefits of the previous schemes, and add a number of new ones.
- Furthermore, several previously *ad-hoc* optimisations specifically aimed at improving the performance of list based computations can now be unified under a single framework. One important example is the expression:

  ```
  (xs ++ ys) ++ zs
  ```

  This is automatically transformed into a form equivalent to:

  ```
  xs ++ (ys ++ zs)
  ```

- We seek algorithms that are simple enough to implement, and cheap enough to execute; we study the issues raised by implementing our list removal technique in a state-of-the-art compiler (**Chapter 4**).

- We have identified a number of critical additions needed to the list removal system, by way of "enabling technologies". Specifically, we need to provide a stronger compile-time inlining mechanism (§ 4.3), and provide a function arity expansion scheme that allows efficient accumulating parameters (§ 4.4).

- In **Chapter 5** we provide detailed quantitative measurements of gains provided by the list removal technique. We measure its effectiveness on a suite of real application programs, not hand crafted benchmarks designed to demonstrate our ideas in a favourable light.

- As the listful style of programming can be compiled efficiently, so can Haskell monolithic arrays (§ 5.2). In particular, the array creation fragment

  ```
  array (1,n) [ (i,<exp>) | i <- [1..n]]
  ```

  can now be compiled as well as the equivalent program in an imperative language using constructs like `for` loops, modulo the cost of suspending the expression `<exp>`. Actually *building* the array is no longer a significant overhead compared to imperative languages.

- Our deforestation algorithm has been recognised by others as practical, and some work has already been done by others to extend the ideas presented in this dissertation. In **Chapter 6** we put our work in context of other data structure removal techniques, and explain proposed extensions to cheap deforestation.

In **Chapter 7** we draw conclusions, and suggest further work. Finally, each entry in the reference list is annotated with the page number of the original citation inside the body of this dissertation.

Parts of this work have been previously presented in Gill, Launchbury & Peyton Jones (1993) and Gill & Peyton Jones (1994). The proof on page 27, taken from Gill et al. (1993), was performed by John Launchbury.

Chapter 2

The `foldr/build` Transformation

In this chapter we present a new, simple, automatic transformation — the `foldr/build` transformation — which can be used to remove intermediate lists. As an aid to understanding the rationale behind our new rule, we first present a simple and well known transformation that removes explicitly intermediate constructors (§ 2.1). We then present canonical ways of expressing list consumption and production (§ 2.2), and introduce our new transformation (§ 2.3). Finally, we show how types can be used to guarantee correctness of our rule (§ 2.4).

First we define some terminology.

- **To Inline** – To replace an instance of a binder v with its definition. If no instances of the binder v remain the original binding can also be removed. i.e. to inline `v` in `let v = 2 in 1 + v` gives `1 + 2`.

  Performing inlining can introduce name clashes, so in our examples we sometimes perform trivial renamings to avoid name clashes, as well as for clarity.

- **To β-reduce** – To replace an expression of the form

  $$(\lambda\ v \rightarrow e_1)\ e_2$$

  with the expression

  $$\texttt{let}\ v = e_2\ \texttt{in}\ e_1$$

  and then to perform the inlining of v.

- **To Unfold** – This is the same as inlining, but after the inlining is complete we then take advantage of any new opportunities for β-reduction. For example, consider:

  ```
  let
      fn = \ a -> a + 1
  in
      fn 2
  ```

  To unfold `fn`, we would first inline `fn`, giving:

  ```
  (\ a -> a + 1) 2
  ```

  Then perform the newly avaliable β-reduction, giving:

  ```
  2 + 1
  ```

  In Haskell sometimes the λs are not explicit, because of Haskell's syntactical sugar. In particular:

  ```
  foo x y = <exp>
  ```

  is shorthand for

  ```
  foo = \ x y -> <exp>
  ```

  and

  ```
  foo (x,y) = <exp>
  ```

  is shorthand for

  ```
  foo = \ v -> case v of (x,y) -> <exp>
  ```

2.1 On removing intermediate data structures

Sometimes removing intermediate data structures can be easy! In functional languages we use constructors (like tuples, `(:)`, etc) to build data structures, and we use `case` (or pattern matching) to deconstruct these structures. Occasionally a simple combination of construction followed by immediate deconstruction can occur, and we can use a simple and well known rule to totally eliminate this explicitly intermediate data structure.

2.1.1 Removing non-recursive intermediate data structures

Consider this Haskell program:

```
data Pair a b = Pair a b
first ab = case ab of
               Pair a b -> a
funny_const a b = first (Pair a b)
```

`funny_const` uses `Pair` to create an intermediate data structure (of type `Pair`) with two identical elements, then selects the first component of this `Pair`, using `first`. If the function `first` is unfolded inside `funny_const`, we get the definition:

```
funny_const a b = case Pair a b of
                       Pair a' b' -> a'
```

We now have an explicit construction and deconstruction of the intermediate `Pair`. We can use a simple, local automatic transformation to remove this intermediate `Pair`. This transformation, called *case reduction* (Santos 1995), takes expressions of the form

$$\begin{array}{l} \texttt{case}\ \mathrm{Cons}\ e_1\ e_2 \ldots e_n\ \texttt{of} \\ \quad\vdots \\ \quad \mathrm{Cons}\ v_1\ v_2 \ldots v_n\ \texttt{->}\ e \\ \quad\vdots \end{array}$$

and, for *any* "Cons", transforms this into:

$$\begin{array}{l} \texttt{let} \\ \quad v_1 = e_1 \\ \quad v_2 = e_2 \\ \quad\quad\vdots \\ \quad v_n = e_n \\ \texttt{in} \\ \quad e \end{array}$$

totally removing the (explicitly intermediate) Cons. After using this transformation, we get:

```
funny_const a b =
   let
     a' = a
     b' = b
   in
     a'
```

Now we can inline a' and b', giving:

```
funny_const a b = a
```

This version of funny_const does not use any intermediate structures.

2.1.2 Removing recursive intermediate data structures

This thesis concerns itself with the removal of intermediate lists. Since the above transformation is true for *any* arbitrary constructor, not just the Pair constructor, could we not simply extend this transformation to deal with the constructors that are used to create the lists? Unfortunately this is not straightforward, because the list datatype is recursive. We could use this transformation to eliminate *individual* constructors. For example, consider:

```
foo g = case f 1 : b of
           x : y -> g x
           [] -> z
```

Using case reduction gives:

```
foo g = g (f 1)
```

However this transformation alone is not a complete solution to removing recursive intermediate data structures, like lists. The transformation can eliminate individual constructors, but if the list is recursively produced case reduction can no longer be straightforwardly applied. Consider, for example, the expression sum (from n m):

```
sum_from n m = sum (from n m)

sum xs = case xs of
           [] -> 0
           (x:xs') -> x + sum xs'
from x y =
        if x > y
        then []
        else x : from (x+1) y
```

We can unfold sum to expose the case, and unfold from to expose the (:) and [], obtaining:

```
sum_from n m = if n > m
        then case [] of
               [] -> 0
               (x:xs') -> x + sum xs'
        else case n : from (n+1) m of
               [] -> 0
               (x:xs') -> x + sum xs'
```

There are two instances of the case reduction transformation, one with an explicit [] and one with an explicit (:). We can now use the case reduction transformation, to remove these intermediate constructors, and can obtain:

```
sum_from n m =
       if n > m
       then 0
       else n + sum (from (n+1) m)
```

We have removed the first constructor in the intermediate list between sum and from, but a new instance of sum (from ...) has appeared. Repeated applications of this constructor removal technique would remove more and more intermediate (:) cells, *but only a finite number of them.* Furthermore, the program size would increase at each step.

The trick to making this approach to list removal work is to "tie the knot"; that is spot the relationship between the different iterations of this list removal technique, and *fold* the original definition back into the unrolled version. This can give an efficient, recursive definition that has eliminated the intermediate list.

In the above example, we know from the definition of sum_from that:

$$\forall n \,.\, \forall m \,.\, \texttt{sum\_from}\ n\ m\ =\ \texttt{sum}\ (\texttt{from}\ n\ m)$$

So we can "tie the knot" by using this equation, and replace

```
sum (from (n+1) m)
```

with

```
sum_from (n+1) m
```

Performing this replacement gives a definition of sum_from that does not use any intermediate lists.

```
sum_from n m =
        if n > m
        then 0
        else n + sum_from (n+1) m
```

This is the essence of the data structure removal algorithm used in Burstall & Darlington (1977), and is also the approach taken by Wadler (1990), in traditional deforestation. For our present purposes, we simply observe that this style of deforestation is, in general, quite hard both to implement and to prove termination for, even with first order languages. The extension to higher order polymorphic languages (like our target language, Haskell) is even harder. We postpone a more detailed discussion of this style of data structure removal until Chapter 6.

2.2 Consumption and production of lists

We take a totally different approach to data structure removal from traditional deforestation. Rather than trying to use a rule that works over one level of a data structure, and make it work over recursive structures, we ***invent a new rule that works over the whole length of the data structure.***

- Instead of using `case` to express how we take a list apart, consuming one constructor at a time, we use a function, `foldr`, which consumes a list as a complete unit.
- Instead of using individual constructors to construct a list, a constructor at a time, we use a new function, `build`, which constructs the whole list.
- Because of the uniform treatment of lists as complete objects, rather than being considered as composed of many individual constructor cells, we can have a single rule, the `foldr/build` rule, which is analogous to the case reduction rule, ***but instead of removing a single constructor, it removes the entire data structure.***

We represent this analogy in Table 2.1. `foldr` and `build` are introduced into our programs either explicitly by the programmer, or by unfolding pre-defined versions of prelude functions (like `map`, `filter`, `concat`, etc.) that use `foldr` and `build` for list consumption and construction. Using our cancellation rule we can fuse suitable consumers and producers together, resulting in programs that use fewer intermediate lists.

	Non-Recursive	Recursive (lists)
Construction	Constructors	`build`
Deconstruction	`case`	`foldr`
Cancellation Rule	case reduction	`foldr/build` rule

Table 2.1: Analogy between case reduction and the `foldr/build` rule

Opportunities suitable for our form of deforestation are illustrated in Figure 2.1, where a good consumer is `foldr`, and a good producer is `build`. When a list is produced without using `build` or consumed without using `foldr` our deforestation scheme does not remove the list. However, the scheme is not hampered by the presence of other methods of list construction and consumption, rather it simply uses the `foldr/build` rule where applicable, and leaves the other lists intact.

We now introduce our list consumption technique (§ 2.2.1) and our list production technique (§ 2.2.2). In the next section we present the `foldr/build` rule itself (§ 2.3).

2.2.1 `foldr` : A super-case

Many functions that consume lists do so in a "regular" way, that is functions that treat each element in the same way. An example of such a regular consumption is the function `and`:

```
and :: [Bool] -> Bool
and []     = True
and (x:xs) = x && and xs
```

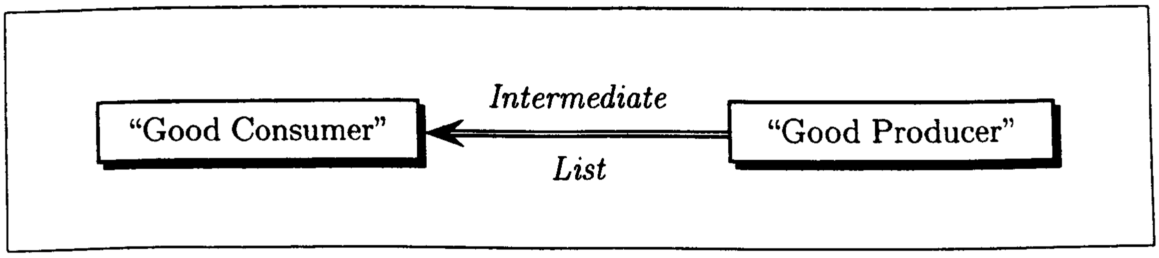

Figure 2.1: An opportunity for list removal

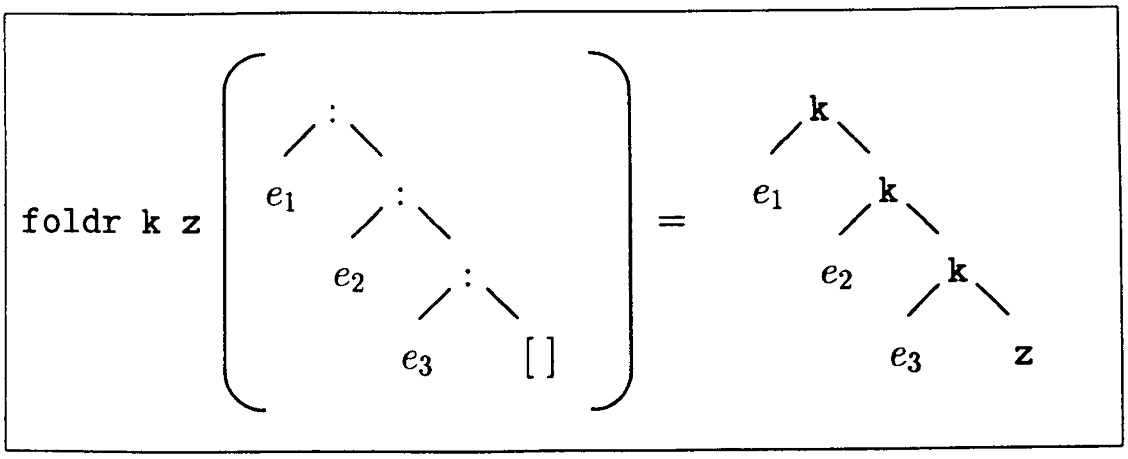

Figure 2.2: How `foldr` transforms a list

and consumes a list, checking to see if all the elements in this list are `True`, and if so returning `True`. Many list-consuming functions can be written recursively, in a similar way. However, in line with the spirit of functional programming in general, instead of using a separate recursive definition for each and every time a list is consumed, functional programmers are encouraged to use a general function, `foldr`, that encapsulates "regular", recursive consumption (Hughes 1989, Bird 1989). The Haskell definition of `foldr`, from the standard prelude, is:

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

`foldr` expresses recursive list consumption, and `foldr`'s arguments express how the consumption is performed. When `foldr k z` is applied to a list, each `(:)` cell is replaced with `k`, and the terminating `[]` is replaced with `n`. This behaviour is illustrated in Figure 2.2. Conceptually we consider `foldr` like a "super-case", because `foldr` deconstructs not just a single cons cell, but the whole list, applying provided functions to each element of the list.

If we return to our and example, and rewrite it using `foldr`, we get:

```
and xs = foldr (&&) True xs
```

This is our canonical form for the function and. The `foldr` is a `super-case` over the list `xs`. If we look at the original (recursive) definition of and, we see that the function and indeed replaces `[]` with `True`, and `(:)` with `(&&)`, just as the `foldr` definition suggests.

Many list consuming functions can be expressed using the *general* pattern of recursion that `foldr` provides. Common examples include:

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

`sum` replaces all `(:)` with `(+)` and `[]` with `0`, returning the sum of the elements of the `Int` list.

```
map f xs = foldr (\ a b -> f a : b) [] xs
```

`map` applies a provided function `f` to all the elements of its list argument.

```
xs ++ ys = foldr (:) ys xs
```

`(++)` consumes its first argument, copying this list onto the front of its second argument.

All these examples are instances of `foldr` being used as a super-`case`. We now consider `foldr`'s dual, our super-constructor `build`.

2.2.2 `build` : A super-constructor

`foldr` consumes a list by dynamically descending the spine of its list argument, and treating each element of this list in a uniform manner. If we could do this descent *statically*, at compile time, we could avoid ever actually building the list consumed by `foldr`. Our "super-constructor", `build`, allows this by providing a mechanism for gaining access to the places in an expression where the `(:)` and `[]` of an output list are used.

Consider a simple constant list

```
1 : 2 : 3 : []
```

We can lift out the `(:)` and `[]` by abstracting the list over `(:)` and `[]`[1]:

```
(\ c n -> 1 `c` (2 `c` (3 `c` n))) (:) []
```

To allow our transformation to spot when lists are abstracted over their `(:)` and `[]` we use a function `build` to express such lists. `build`'s definition is[2]:

```
build :: ((a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

`build` is a function that, if given a function that builds a list *using the* `(:)` *and* `[]` *provided by* `build`, returns that list. It is a "super-constructor", building a

[1] `<exp1> `c` <exp2>` is Haskell syntactic sugar for `c <exp1> <exp2>`.
[2] Actually `build`'s type is not a "standard" Hindley-Milner type, a point we return to in § 2.4.

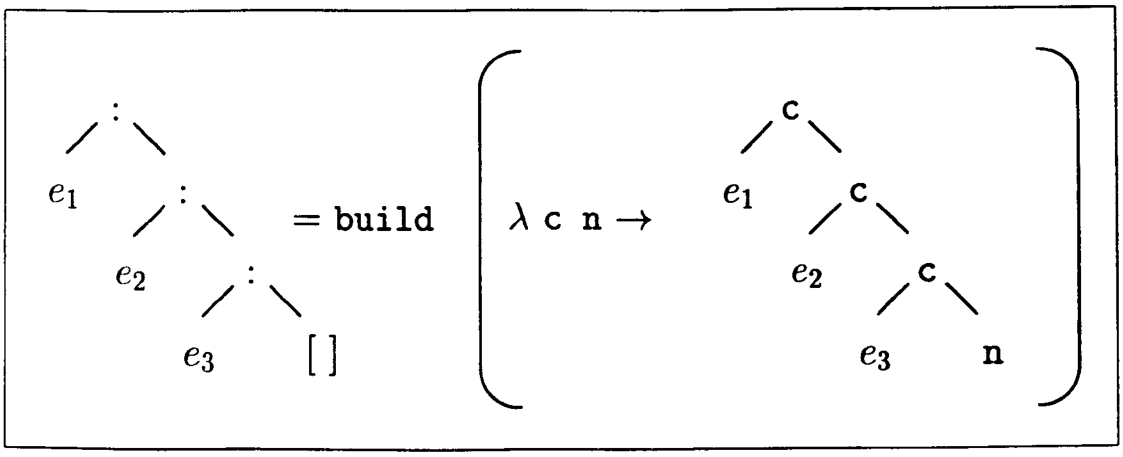

Figure 2.3: How **build** expresses a list

whole list. So how do we use `build` to express lists? Our constant list example would become:

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

Figure 2.3 illustrates how `build` expresses the constant list e_1 : e_2 : e_3 : `[]`.

`build` can also be used to express lists generated recursively. For example:

```
from x y =
        if x > y
        then []
        else x : from (x+1) y
```

We can bind a local version of this function, and abstract over the `(:)` and `[]` of the output list:

```
from x y =
   build (\ c n ->
   let
      from' x y = if x > y
                  then n
                  else x `c` from' (x+1) y
   in from' x y)
```

2.2.3 Using `foldr` and `build` inside list filters

Functions that both produce and consume lists can be expressed simultaneously as both good producers and good consumers by using both `foldr` and `build` in the same definition. For example, `map` can be re-written as

```
-- List consumer
sum = foldr (+) 0 xs

-- List producer
from x y =
   build (\ c n ->
   let
      from' x y = if x > y
                  then n
                  else x `c` from' (x+1) y
   in from' x y)

-- Functions that both produce and consume lists.
map f xs  = build (\ c n -> foldr (\ a b -> f a `c` b) n xs)
concat xs = build (\ c n -> foldr (\ x y -> foldr c y x) n xs)
xs ++ ys  = build (\ c n -> foldr c (foldr c n ys) xs)
```

Figure 2.4: Selective `foldr/build` versions of function definitions

```
map f xs = build (\ c n -> foldr (\ a b -> f a `c` b) n xs)
```

We can use both `foldr` and `build` in this manner in many of our definitions, and where possible use function definitions that use both `foldr` and `build`. A list of the definitions used in this chapter is given in Figure 2.4[3].

2.3 The `foldr/build` rule

If a list is expressed using `build`, and is consumed using `foldr`, our key transformation — the `foldr/build` rule — can be used to eliminate an intermediate list.

$$\boxed{\texttt{foldr}\ k\ z\ (\texttt{build}\ g)\ =\ g\ k\ z}$$

Assuming that g constructs its result using the "cons" and "nil" values passed to it as arguments by `build`, and because `foldr k z` substitutes `k` for `(:)` and `z` for `[]`, the same effect as producing and consuming the list can be achieved by passing `k` and `z` directly to `g`. (We can actually assert that g uses the "cons" and "nil" values passed to it by giving `build` a special type, which we do in § 2.4.) Figure 2.5 illustrates how the rule works on a small constant list.

Our initial blueprint for using the `foldr/build` rule, and performing `foldr/build`

[3]We actually redefine the definitions of both (++) and sum later, in Chapter 3.

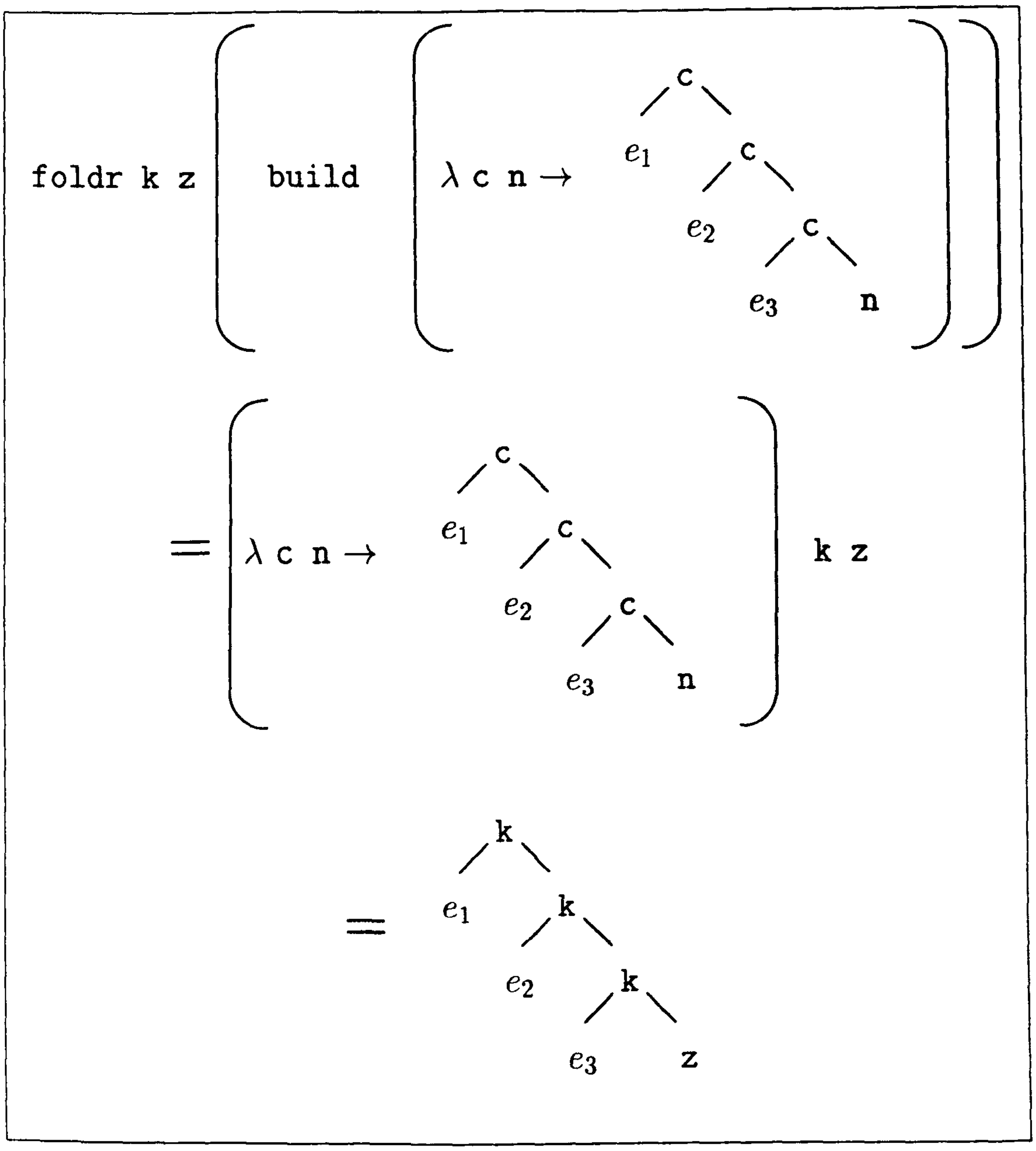

Figure 2.5: How the `foldr/build` rule works

deforestation is, where possible:

- Express list consumers in terms of `foldr`, including prelude functions that consume lists.
- Express list producers in terms of `build`, including prelude functions that produce lists.
- Express functions that both consume and produce lists in terms of both `foldr` and `build`, and including suitable prelude functions.
- Unfold these list processing functions, exposing `foldr` and `build` to the `foldr/build` rule. Even if the user has not explicitly used `foldr` and `build`, they still get some benefit, via the unfolded prelude functions that use `foldr` and `build`.
- Use the `foldr/build` rule, along with other simple transformations.

We now proceed to look at four examples of our deforestation in action. It should first be pointed out, however, that there is a large caveat with the `foldr/build` rule: it *only* works for instances of `build`'s argument that are truly abstracted over the intermediate lists *cons*'s and *nil*'s. This is an important restriction to which we return to in § 2.4.

2.3.1 Example 1: sum of constant list

Consider as our first example:

```
sum (1 : 2 : 3 : [])
```

If we express the producer `(1 : 2 : 3 : [])` and the consumer (`sum`) in our canonical form, we get the expression:

```
foldr (+) 0 (build (\ c n -> 1 ‘c‘ (2 ‘c‘ (3 ‘c‘ n))))
```

Now we can use our `foldr/build` rule, and obtain:

```
(\ c n -> 1 ‘c‘ (2 ‘c‘ (3 ‘c‘ n))) (+) 0
```

β-reduction gives:

```
1 + (2 + (3 + 0))
```

This transformed expression succeeds in summing the constant list, but *without ever building the list at all.*

2.3.2 Example 2: `sum` of `from`

As our second example, reconsider the example in § 2.1.2. However, this time we use function definitions that are expressed in our canonical form, allowing good production and consumption of intermediate lists:

```
sum xs = foldr (+) 0 xs
from x y =
   build (\ c n ->
   let
        from' x y =
            if x > y
            then n
            else x `c` from' (x+1) y
   in
        from' x y)
sum_from n m = sum (from n m)
```

Now we can unfold `sum` and `from` into the right hand side of `sum_from`.

```
sum_from n m = foldr (+) 0 (build (\ c n ->
   let
        from' x y =
                if x > y
                then n
                else x `c` from' (x+1) y
   in
        from' n m))
```

Now we can use our `foldr/build` rule and β-reduction to get:

```
sum_from n m =
   let
        from' x y =
                if x > y
                then 0
                else x + from' (x+1) y
   in
        from' n m
```

This definition of `sum_from` sums the list `1` to `n`, without ever actually building the list. Furthermore, no knot-tying was required.

2.3.3 Example 3: unlines

For our next example, consider the function `unlines`, which has the definition

```
unlines :: [String] -> String
unlines ls = concat (map (\l -> l ++ ['\n']) ls)
```

This function takes a list of strings, and joins them together, inserting a newline character after each one. An intermediate version of the list of strings is created, together with an intermediate version of each string (when the newline character is appended).

To deforest this definition, we first provide our 'good' definitions of the functions, taken from Figure 2.4. Unfolding these definitions into `unlines` gives:

```
unlines ls =
 build
  (\ c n ->
   foldr
    (\ xs b -> foldr c b xs)
    n
    (build
     (\ c1 n1 ->
      foldr
       (\ l t ->
        c1
        (build
         (\ c2 n2 ->
          foldr c2
                (foldr c2
                       n2
                       (build (\ c3 n3 -> c3 '\n' n3))
                ) l ))
        t
       ) n1 ls)))
```

Now we apply the `foldr/build` transformation, to get:

```
unlines ls =
 build
  (\ c n -> (\ c1 n1 ->
             foldr (\ l t ->
                      c1
                       (build
                        (\ c2 n2 ->
                         foldr c2
                               ((\ c3 n3 -> c3 '\n' n3)
                                 c2
                                 n2)
                               l))
                       t)
                    n1 ls)
            (\ xs b -> foldr c b xs)
            n)
```

Performing three β-reductions gives:

```
unlines ls =
  build
   (\ c n ->
    foldr (\ l t -> foldr c t
                     (build
                        (\ c2 n2 ->
                           foldr c2 (c2 '\n' n2) l))
          ) n ls)
```

This in turn exposes a new opportunity to use the foldr/build rule. After using it, and performing some more β-reductions we get:

```
unlines ls =
  build
   (\ c n -> foldr (\ l b -> foldr c (c '\n' b) l) n ls)
```

Now no more applications of the transformation are possible. We may choose to leave the definition in this form, so that any calls of unlines may also be deforested. After exploiting any possible deforestation opportunities we may now unfold build, revealing the (:)'s and []. After simplification we get:

```
unlines ls
    = foldr (\ l b -> foldr (:) ('\n' : b) l) [] ls
```

If we also unfold foldr, using the definition

```
foldr f z xs =
  let h []     = z
        (x:xs) = f x (h xs)
  in h xs
```

we get

```
unlines ls = h ls
  where h []     = []
        h (l:ls) = g l
          where g []     = '\n' : h ls
                g (x:xs) = x : g xs
```

This is as efficient a coding of `unlines` as we may reasonably hope for. Furthermore, we have derived this efficient version automatically, modulo the use of our "deforestation friendly" prelude functions.

2.3.4 Example 4: sum of squares

As our final example we take a listful program from the introduction[4].

```
square x = x * x
sumSq n  = sum (map square (from 1 n))
```

Now we express list consumption in terms of `foldr`, and list production in terms of `build`, by again providing suitable unfoldings for `from`, `map` and `sum`:

```
square x = x * x
sumSq n
  = foldr (+) 0
      (build (\ c n ->
       foldr (\ a b -> square a `c` b)
             n
             (build (\ c1 n1 ->
               let
                 from' x y =
                    if x > y
                    then n1
                    else x `c1` from' (x+1) y)
               in
                 from' 1 n)))
```

Now we can use our `foldr/build` transformation, twice, along with β-reductions, to get:

[4] `[1..n]` in syntactical sugar for `from 1 n`

```
square x = x * x
sumSq n
  = let
        from' x y =
            if x > y
            then 0
            else square x + from' (x+1) y
    in
        from' 1 n
```

Again the foldr/build transformation has provided us with a route to automatically eliminate the intermediate lists in the original, listful definition.

2.4 Using a free theorem to guarantee correctness

There appears to be a serious problem with the approach we have described: the foldr/build rule can be wrong! For example,

```
foldr k z (build (\ c n -> [True])) ≠ (\c n -> [True]) k z
```

Using the definitions of foldr and build we can see that the left-hand side is k True z, while the right hand side is just [True]. These two values do not even necessarily have the same type!

The trouble with the counter-example is that the function passed to build constructs its result list without using c and n. The time we can guarantee that the foldr/build rule does hold is when build's argument truly is a list which has been "uniformly" abstracted over all its *conses* and *nil*, which was an informal requirement of using build. Fortunately, it turns out that we can use the type of build to guarantee this property. Consider again the definition of build

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

Now suppose that g has the type

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

for some fixed type A. We can argue informally that g must construct its result only using the supplied cons and nil, as follows: because g is polymorphic in β, it can only manufacture its result (which has type β) by using its two arguments, k and z. Furthermore, the types of k and z mean that they can only be composed into an expression of the form

$$(k\ e_1\ (k\ e_2 \ldots (k\ e_n\ z) \ldots))$$

which is exactly the form we require.

We can render this argument formally, by using the "free theorem" for g's type (Reynolds 1983, Wadler 1989).

Theorem
(The `foldr/build` rule)

If for some fixed A we have

$$g : \forall\beta.(A \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$$

then

$$\texttt{foldr}\ k\ z\ (\texttt{build}\ g) = g\ k\ z$$

Proof
The "free theorem" associated with g's type is that, for all types B and B', and functions $f : A \rightarrow B \rightarrow B$, $f' : A \rightarrow B' \rightarrow B'$, and (a strict) $h : B \rightarrow B'$ the following implication holds:

$$\begin{array}{l}(\forall a : A.\ \forall b : B.\ h\ (f\ a\ b) = f'\ a\ (h\ b)) \\ \Rightarrow\ (\forall b : B.\ h\ (g_B\ f\ b) = g_{B'}\ f'\ (h\ b))\end{array}$$

where g_B and $g_{B'}$ are the instances of g at B and B' respectively. (From now on we will drop the subscripts from g since languages like Haskell have silent type instantiation).

We now instantiate this result. Let $h = \texttt{foldr}\ k\ z$, $f = (:)$, and $f' = k$. Now the theorem says,

$$\begin{array}{l}(\forall a : A.\ \forall b : B.\ \texttt{foldr}\ k\ z\ (a : b) = k\ a\ (\texttt{foldr}\ k\ z\ b)) \\ \Rightarrow\ (\forall b : B.\ \texttt{foldr}\ k\ z\ (g\ (:)\ b) = g\ k\ (\texttt{foldr}\ k\ z\ b))\end{array}$$

The left hand side is a consequence of the definition of `foldr`, so the right hand side follows. That is,

$$\forall b : B.\ \texttt{foldr}\ k\ z\ (g\ (:)\ b) = g\ k\ (\texttt{foldr}\ k\ z\ b)$$

Now let $b =$ `[]`. By definition, `foldr` $k\ z$ `[]` $= z$, so finally we obtain,

$$\texttt{foldr}\ k\ z\ (g\ (:)\ \texttt{[]}) = g\ k\ z$$

which, given the definition of build, is exactly what we require. □

The impact of this result is significant: as long as build is only applied to functions of the appropriate type, our deforestation transformations may proceed via the foldr/build rule with complete security.

build's full type is therefore:

$$\texttt{build} :: \forall\alpha\, .\, (\forall\beta\, .\, (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha]$$

Unfortunately build's type does not conform to the Hindley-Milner type system (Milner 1978), of which variants are used by many lazy functional languages, including our target language Haskell. A more general type system, such as that of Ponder (Fairbairn 1985) or Quest (Cardelli & Longo 1991) would allow this type for build, but they lack the type-inference property. However, a simple extension to the Hindley-Milner type checker can check that applications of build are well typed. We explain the details of the required extension to the Haskell type system in § 4.2.1, where we use 2nd-rank polymorphism (McCracken 1984).

2.5 Summary

In this chapter we have presented a new approach to removing intermediate lists. This technique is based on a simple rule that allows a transformation system to treat lists as complete objects. The technique requires a highly stylistic list production and consumption. This can be achieved by using predefined (prelude) functions that have been defined in this restrictive style.

Several questions remain open, and must be addressed before we can build a successful implementation of cheap deforestation. Most importantly, we need to understand the issues behind writing good list producers and consumers. This is the subject of the next chapter.

Chapter 3

Expressing List-manipulating Functions in `foldr/build` Form

In this chapter we take `foldr/build` deforestation, as presented in the previous chapter, and turn it into a practical deforestation scheme. We examine the issues behind good list consumption and good list production, making the necessary changes to the basic `foldr/build` deforestation model. This enhanced deforestation system we call *cheap deforestation*.

First we examine how we can provide new definitions of prelude functions, and the issues behind using these "alternative" definitions (§ 3.1). We then examine the problems with writing effective list-manipulating functions in `foldr/build` form (§ 3.2). In § 3.3 we give new translation rules for list comprehension rules that produce code that is amicable to our deforestation scheme. In § 3.4 we introduce a new list producer, `augment`. Finally, in § 3.5 we summarise our augmented set of deforestation rules, and classify when our technique is expected to remove intermediate lists.

3.1 A new prelude

The simplest method of exposing `foldr` and `build` to the `foldr/build` rule, and providing deforestation opportunities, is via pre-defined, inlineable definitions of prelude functions such as `map` and `filter`. We now consider some of the pragmatics of using this method for providing deforestation opportunities.

Any compiler that included an implementation of cheap deforestation would need, at the very least, some method for transferring literal code from the prelude to user modules. This could be of the form proposed for Haskell 1.3. For example, the prelude definition of map might be written:

```
{-# INLINE map #-}
map f xs = build (\ c n -> foldr (\ a b -> f a `c` b) n xs)
```

(In the context of an implementation the distinction between inlining and unfolding is not significant, since inlining would usually be followed by further optimisations that have been enabled by the newly inlined definition.)

There are two important criteria when re-writing functions in terms of `foldr` and `build`, correctness (§ 3.1.1), and efficiency (§ 3.1.2). We also briefly consider the problem of increased code size caused by inlining many list producers and consumers (§ 3.1.3).

3.1.1 Correctness of our inlined functions

It is important that our "deforestation-friendly" function definitions are correct. The new definition must exactly match the semantics of the original, including strictness semantics.

The old and new definitions can be proved equivalent using straightforward equational reasoning and induction. For example, here is a proof that the new version of map is equal to the original definition:

Theorem

$$\forall\, f\ .\ \forall\, xs\ .\ \mathtt{map}\ f\ xs\ =\ \mathtt{map}_{new}\ f\ xs$$

where

```
map f []     = []
map f (x:xs) = f x : map f xs
```

and

$\mathtt{map}_{new}$ `f xs = build (\ c n -> foldr (\ a b -> f a `c` b) n xs)`

Proof
By induction over the lifted list datatype, xs.

Base case 1. [ ]

Both sides reduce to `[]`.

Base case 2. $\perp$

Both sides reduce to $\perp$.

Inductive Case. Assume

$$\mathtt{map}\ f\ xs = \mathtt{map}_{new}\ f\ xs$$

The goal is to show that

$$\mathtt{map}\ f\ (x : xs) = \mathtt{map}_{new}\ f\ (x : xs)$$

Starting from the left hand side:

```
map f (x:xs)
    = f x : map f xs
```

Now we use the inductive hypothesis

```
    = f x : mapnew f xs
    = f x : build (\ c n -> foldr (\ a b -> f a `c` b) n xs)
    = build (\ c n -> f x `c` foldr (\ a b -> f a `c` b) n xs)
    = build (\ c n -> foldr (\ a b -> f a `c` b) n (x:xs))
    = mapnew f (x:xs)
```

□

3.1.2 Efficiency of our inlined functions

Sometimes a `foldr/build` version of a prelude function might not fuse with a good producer or good consumer. When writing prelude functions in terms of `foldr` and `build` we have to take this into consideration. We need to make sure that any list processing function written in terms of `foldr` and/or `build` can be compiled as efficiently as the definitions it is replacing.

Consider this `foldr/build` definition of `map`, as defined inside our new prelude:

```
map f xs = build (\ c n -> foldr (\ a b -> f a `c` b) n xs)
```

If we unfold `foldr` and `build`, we get:

```
map f xs =
   let
      h xs = case xs of
                [] -> []
                (a:as) -> f a : h as
   in
     h xs
```

Compare this with the original, recursive definition of **map**:

```
map f xs = case xs of
             [] -> []
             (a:as) -> f a : map f xs
```

A trade-off has occurred between these two definitions. The first definition has a local recursion, but *allocates* a closure every time **map** is called (cf. § 4.1.2). The second definition has an extra, static argument, but does not perform extra allocations. These extra allocations were found to be significant in practice. Somewhat fortunately, the compiler performs lambda lifting (Peyton Jones 1987, Santos 1995), transforming the unfolded `foldr/build` version of **map** into the version with the top-level recursion.

This example highlights a difficulty with implementing cheap deforestation. Using cheap deforestation can result in the production of code that can be substantially different in structure to that written by human programmers. This code is then optimised by other passes, including the lambda lifter. Other optimisations sometimes perform heuristics that, over a wide range of examples written by humans, are observed to improve programs. It is possible however, that the `foldr/build` transformation may create examples that these heuristics perform poorly on. We examine this problem again in Chapter 5. It turns out that for only one benchmark (out of over 50) the execution time got worse, and by only 1%.

3.1.3 Increase in code size

Using inlining technology to inline many list processing functions has influence on the target code size (cf. Chapter 5). Evidence suggests this is not as great a problem as might be suspected. In our measurements, the binaries produced by our compiler grew by only about 8%, and about 6% of this was recovered by enhanced code optimsation and simplification opportunities. Ultimately a selective inlining scheme would be useful, where functions are only inlined if there are deforestation opportunities to exploit. We do not discuss this further here, but return to this point in Chapter 7.

3.2 Writing list-manipulating functions in foldr/build form

In this section we examine the difficulties of writing list-manipulating functions in terms of foldr and build. We first introduce some issues behind using build (§ 3.2.1), and foldr (§ 3.2.2). We then argue that foldl should also be classed as a good producer, and given a first class status (§ 3.2.3). We then look at other "regular" consumers (§ 3.2.4), before finally considering zip (§ 3.2.5).

3.2.1 Expressing list production using build

Expressing the production of a result list using build is straightforward, provided that all the (:) and [] cells are produced internally by the function in question. For example, consider the function init, which is defined:

```
init []     = error "init{PreludeList}: init []\n"
init [x]    = []
init (x:xs) = x : init xs
```

We can bind a local instance of init, and scope our build produced *cons* and *nil*, giving:

```
init xs = build (\ c n ->
  let
        init' []     = error "init{PreludeList}: init []\n"
        init' [x]    = n
        init' (x:xs) = x `c` init' xs
  in
        init' xs)
```

In general, it is straightforward to re-express list production in this way, provided that *the list producing function produces that whole list.* Difficulties arise when a list producing function constructs only part of its result, and obtains the tail of its result from an external source. For example, consider the (++) function:

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Part of the result list is passed in as an argument (in this case, ys). We present a solution to this problem that allows such list producers to be written using a generalisation of build in § 3.4.

There are also difficulties when a list producer internally consumes its own result. For example, consider:

```
transpose :: [[a]] -> [[a]]
transpose xs =  foldr
                 (\xs xss -> zipWith (:) xs (xss ++ repeat []))
                  [] xs
```

We cannot easily abstract over `(:)` and `[]`, and accept this list producer as a "bad" list producer.

Constant lists, however, are straightforward to represent using **build**. We have already seen an example of this in § 2.2.2. Haskell represents `String`'s as a list of characters, and has a special form of constant list especially for this. For example

```
"Hello, World"
```

is syntactic sugar for:

```
['H','e','l','l','o',',',' ','W','o','r','l','d']
```

We could treat constant strings in the same way as traditional constant lists, though in our implementation we actually perform an optimsation that allows us to keep the strings in a more compact form.

3.2.2 Expressing list consumption using foldr

As we have already suggested, many functions can be written so that a list argument can be consumed using `foldr`. Some Haskell prelude functions are *already* defined in term of `foldr`, and just need inlining. For example:

```
and xs = foldr (&&) True xs
or xs = foldr (||) False xs
```

With some functions, we just needed to add the list construction aspect (using `build`) to the suitable list producers.

```
concat xs = build (\ c n -> foldr (\ x y -> foldr c y x) n xs)
```

Some other prelude functions can easily be re-written in `foldr` form:

```
head xs
  = foldr (\ x _ -> x) (error "head{PreludeList}: head []\n") xs
null xs
  = foldr (\ _ _ -> False) True xs
map f xs
  = build (\ c n -> foldr (\ a b -> f a `c` b) n xs)
filter f xs
  = build (\ c n -> foldr (\ a b -> if f a
                                    then c a b
                                    else b) n xs)
```

Finally, some functions consume their list argument using a "good" consumer anyway, for example:

```
unlines xs = concat (map (++ "\n") xs)
```

There are however, a significant class of functions that cannot be re-written quite so straightforwardly. We now look at some examples.

3.2.3 `foldl` : Another super-case

Sometimes list consumption can be neatly expressed using a recursive function with an accumulating parameter. For example, to find the length of a list, you can carry an accumulating argument (the length so far) down a list, incrementing the accumulator at each cons cell:

```
length xs = len 0 xs
    where
        len a []     = a
        len a (x:xs) = len (a+1) xs
```

This accumulating consumption can be expressed using the function `foldl`:

```
length xs = foldl (\ a x -> a + 1) 0 xs
```

`foldl`, like `foldr`, is a common template for list consumption. It has the definition (from the Haskell prelude):

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

Other functions that can be expressed in terms of `foldl` include `reverse` and `sum`.

```
reverse xs = build (\ c n -> foldl (flip c) n xs)
sum xs = foldl (+) 0 xs
```

It would be useful to be able to remove lists that are consumed with `foldl` as well as removing lists consumed by `foldr`. It turns out that we can express any `foldl` consumption in terms of a `foldr` consumption! We do this by exploiting the relationship:

```
foldl f z xs = (foldr (\ b g a -> g (f a b)) id xs) z
```

An extra pair of brackets has been inserted into this equation for clarity. `foldr` is taking an *extra* argument, and `foldr`'s arguments also have an extra argument, the accumulator.

Because of this relationship between `foldl` and `foldr`, we can achieve deforestation for lists consumed in terms of `foldl`. We prove this relationship between `foldl` and `foldr` at the end of this section.

So how does this new definition of `foldl` work? The first argument acts as a rebracketing tool, changing the direction of the fold. Consider the example:

```
foldl f z (e1:e2:e3)
  = (((z `f` e1) `f` e2) `f` e3)
```

We can see that `f` is acting left associative. However, if we express `foldl` in terms of `foldr`, we can get

```
foldl f z (e1:e2:e3)
  = (foldr (\ b g a -> g (f a b)) id (e1:e2:e3)) z
  = (e1 `fn` (e2 `fn` (e3 `fn` z))) z
     where fn = \ b g a -> g (f a b)
```

`fn`, the first argument to `foldr` is now bracketing to the right.

Now if we unfold `fn`, we "reverse" the associativity of this sequence, getting the correct associativity for `foldl`:

```
  = (((z `f` e1) `f` e2) `f` e3)
```

It is imperative that if we are going to generate efficient code, we unfold `fn` to allow a simplification system to explicitly turn round the associative ordering. This was found to be an issue in practice. In order to allow us to better control the inlining of `fn`, we include `foldl` as a new good consumer, and perform the unfolding of `fn` on the fly. When we do this, we tag `fn` with a "must be inlined if possible" flag.

Example of foldl as a good consumer

As a concrete example of using foldl as a good consumer, consider this example:

```
length (filter p xs)
```

We can express length in terms of foldl as well as giving our unfolding for filter, exposing list production.

```
foldl (\ a _ -> a + 1) 0
  (build (\ c n ->
     foldr (\ x r ->
        if p x then x `c` r else r) n xs))
```

Now we can represent foldl in terms of foldr:

```
foldr (\ b g a -> g (a + 1)) id
  (build (\ c n ->
     foldr (\ x r ->
        if p x then c x r else r) n xs)) 0
```

Notice how we have unfolded (\ a _ -> a + 1) into the first argument of foldr. Now we can use our foldr/build rule to remove the intermediate list. After using the foldr/build rule we get:

```
foldr (\ x r -> if p x then (\ a -> r (a + 1)) else r) id xs 0
```

After inlining foldr, we get:

```
let
   h ys = case ys of
          [] -> \ x -> x
          (x:xs) -> if p x then (\ a -> h (a + 1)) else h xs
in
  xs 0
```

Now we can expand the arity of h, by adding an extra argument, giving the efficient:

```
let
   h []     a = a
   h (x:xs) a = if p x
                then h xs (a + 1)
                else h xs a
in
   h xs 0
```

In general this final step might duplicate work. For example if the body of the `let` had been `map (h xs) ys` then we would re-evaluate `(p x)` for each element of the list instead of sharing a single call of `(p x)` as the previous version would do. In this specific case the transformation *is* safe, because we can prove that every time `h` is called, it will be called with at least two arguments, and therefore there are no partial applications of `h`. We introduce a new analysis that allows this transformation in § 4.4.

Proof of relationship between `foldl` and `foldr`

Theorem
$\forall f \,.\, \forall z \,.\, \forall xs$.

```
foldl f z xs = foldr (\ b g a -> g (f a b)) id xs z
```

where `b`, `g` and `a` are new names.

Proof
By induction over the lifted list datatype, *xs*:

Base case 1.

```
foldl f z [] = foldr (\ b g a -> g (f a b)) id [] z
```

```
foldr (\ b g a -> g (f a b)) id [] z
  = id z
  = z
  = foldl f z []
```

Base case 2.

```
foldl f z ⊥ = foldr (\ b g a -> g (f a b)) id ⊥ z
```

```
foldr (\ b g a -> g (f a b)) id ⊥ z
  = ⊥
  = foldl f z ⊥
```

Inductive Case.

Assume

```
foldl f z xs = foldr (\ b g a -> g (f a b)) id xs z
```

The goal is to show that

```
foldl f z (x : xs) = foldr (\ b g a -> g (f a b)) id (x : xs) z
```

```
foldl f z (x:xs)
= foldl f (f z x) xs
```

Now we perform the inductive step.

```
= foldr (\ b g a -> g (f a b)) id xs (f z x)
= (\ b g a -> g (f a b))
        x (foldr (\ b g a -> g (f a b)) id xs) z
= foldr (\ b g a -> g (f a b)) id (x:xs) z
```

Giving the right hand side.

□

3.2.4 Other regular consumptions

There are other classes of list consumption that, like the `foldl` consumption, can be coerced into a `foldr` consumption. Consider the function `take`, defined as:

```
take 0 _      = []
take n []     = []
take n (x:xs) = x : take (n-1) xs
```

We can write this function in terms of `foldr` (parameterisation over cons and nil using `build` is not important for illustrating this example, so we omit this step):

```
take n xs =
    case n of
       0 -> []
       _ -> let
              fn x g 0 = []
              fn x g a = x : g (a - 1)
            in
              foldr fn (const []) xs n
```

Notice that we have a bi-directional flow of data; the cons-cells being produced from the right, and the accumulating argument counting the number of cells taken so far. The same concept that let `foldl` be expressed as `foldr` (the inlining of the "higher-order" arguments) can also be used here. `fn` and `const []` could be unfolded, to produce efficient code.

However, as we have already discussed (§ 3.1.2), when we replace a function with a `foldr/build` version, we must be convinced that we don't lose efficiency, compared to the original function definition. In some cases, though we *can* express consumption, directly or indirectly via `foldr`, the extra runtime overhead makes such consumptions in terms of `foldr` undesirable. For example, consider the prelude function `foldr1`:

```
foldr1              :: (a -> a -> a) -> [a] -> a
foldr1 f [x]        =  x
foldr1 f (x:xs)     =  f x (foldr1 f xs)
foldr1 _ []         =  error "foldr1{PreludeList}: empty list\n"
```

We can write `foldr1` in terms of `foldr`:

```
data Maybe a = Just a | Nothing
foldr1 f xs =
    case foldr (\ a b ->
            case b of
              Nothing -> a
              Just x -> f a x) Nothing xs of
      Nothing -> error "foldr1{PreludeList}: empty list\n"
      Just ans -> ans
```

Here we have lifted the result over the `Maybe` type. If we were to use this style of `foldr1`, we would need to be convinced that the extra data structures (in this example, `Just` and `Nothing`) were eliminated in the final code produced by our compiler.

3.2.5 Handling zip

The `foldr/build` rule cannot be used to *split* deforestation down two branches of a zip. Consider the example:

```
zip (map f xs) (map g ys)
```

We would like to have a definition of `zip` that consumes *both* its arguments in terms of `foldr`, so that *both* the list `map f xs` and the list produced by `map g ys` may be removed. Unfortunately this is, in general, not possible. We can consume the first list using `foldr`:

```
zip xs ys = foldr f (\ x -> []) xs ys
   where
     f x g []     = []
     f x g (y:ys) = (x,y) : g ys
```

However, it is not possible to use `foldr` to consume the second list at the same time as consuming the first argument using `foldr`. Furthermore, we cannot even choose what argument to consume using `foldr`. It looks like we can just use the above technique for `zip`'s second argument, but the definition of `zip` has an asymmetric semantics.

```
zip xs ys
```

does not equal

```
map (\ (x,y) -> (y,x)) (zip ys xs)
```

This is because

```
zip ⊥ [] = ⊥
```

but

```
zip [] ⊥ = []
```

If `zip` did have symmetrical semantics we could lift `zip` into our transformations as a primitive, and choose when optimising which list to deforest down. This, however, would not solve the problem that we cannot deforest down *both* lists.

Lack of deforestation through zip is a significant shortcoming of cheap deforestation. There is one optimisation that can be done to help minimise the damage caused by `zip`. `zip` is commonly used in conjunction with an enumerator, for example `zip [n..] <exp>` or `zip <exp> [n..m]`. The compiler could provide special versions of `zip` that have the same behaviour as these small expressions, *but can deforest down the single remaining branch.*

For example

```
zip [n..] <exp>
```

can be "specialised" to

```
foldr (\ y g n -> (n,y) : g (n+1)) (\ n -> []) <exp> n
```

now if *this* specialised instance of `zip` appears in the middle of a pipeline of intermediate lists, deforestation could occur.

$$
\begin{array}{rcl}
[\, e \mid b \,] & = & \texttt{if}\ b\ \texttt{then}\ [\, e \,]\ \texttt{else}\ [\,] \\
[\, e \mid q_1\, ,\, q_2 \,] & = & \mathit{concat}\ [\, [\, e \mid q_2 \,] \mid q_1 \,] \\
[\, e \mid p \leftarrow l \,] & = & \texttt{let}\ \ \mathit{ok}\ p\ =\ \mathit{True} \\
 & & \qquad\ \mathit{ok}\ \_\ =\ \mathit{False} \\
 & & \texttt{in}\ \mathit{map}\ (\lambda\, p \rightarrow e\,)\ (\mathit{filter}\ \mathit{ok}\ l)
\end{array}
$$

Figure 3.1: Semantics of Haskell list comprehensions

3.3 List comprehensions

List comprehensions are a particularly powerful form of syntactic sugar, and have become quite widespread in functional languages. For example, given two lists of pairs `r1` and `r2`, each of which is intended to represent a relation, the relational join of the second field of `r1` with the first field of `r2` can be expressed like this:

```
[(x,y1,z) | (x,y1) <- r1, (y2,z) <- r2, y1==y2]
```

This can be read as "the list of all triples `(x,y1,z)`, where `(x,y1)` is drawn from `r1`, `(y2,z)` is drawn from `r2`, and `y1` is equal to `y2`".

The semantics of list comprehensions are given by a straightforward translation (Hudak et al. 1992). Figure 3.1 gives this translation, in the form of identities that can be used to remove the syntactic sugar from list comprehensions.

3.3.1 Traditional techniques for desugaring list comprehensions

There are well established techniques for desugaring list comprehensions into a form which guarantees to construct only one *cons* cell for each element of the result (Wadler 1987*a*, Augustsson 1987). In addition, they include extensions that have:

- provision for optimising a chain of appended list comprehensions, upholding the "one *cons* cell for each element in the result" criterion.
- extra rules for optimising the consumption of enumerated lists.

Figure 3.2 gives the traditional rules for desugaring list comprehensions (Wadler 1987*a*, Augustsson 1987). The $\mathcal{TE}$ scheme translates expressions in a rich syntax, including list comprehensions, into a much simpler functional language.

```
                  $\mathcal{TE} :: Expr \to Expr$

        $\mathcal{TE}[\![ [E \mid QS] ]\!]$ = $\mathcal{TE}[\![ [E \mid QS] \mathbin{+\!\!+} [\,] ]\!]$
     $\mathcal{TE}[\![ [E \mid\,] \mathbin{+\!\!+} R ]\!]$ = $\mathcal{TE}[\![ E ]\!]$ : $\mathcal{TE}[\![ R ]\!]$
  $\mathcal{TE}[\![ [E \mid B\,, QS] \mathbin{+\!\!+} R ]\!]$ = if $\mathcal{TE}[\![ B ]\!]$
                                  then $\mathcal{TE}[\![ [E \mid QS] \mathbin{+\!\!+} R ]\!]$
                                  else $R$
$\mathcal{TE}[\![ [E \mid P \leftarrow L, QS] \mathbin{+\!\!+} R ]\!]$ =
           let
            h xs = case xs of
                     [] -> $\mathcal{TE}[\![ R ]\!]$
                     (x:xs') ->
                        $\mathcal{TE}[\![$ case x of
                                     $P$ -> $[E \mid QS]$ ++ h xs'
                                     _ -> h xs' $]\!]$
           in h $L$

                               ⋮  and other $\mathcal{TE}$ rules
```

Figure 3.2: Traditional translation for Haskell list comprehensions

3.3.2 A new desugaring scheme for list comprehensions

List comprehensions can be used to produce intermediate lists, for example, it may be appended to some other list or it may be summed. List comprehensions can also consume intermediate lists produced by their *generators*. For example:

```
[ f x | x <- map g xs, odd x]
```

Clearly we would like to ensure that list comprehensions are translated in a way which allows the intermediate lists between them and their producers or consumers to be eliminated. It turns out not only can we do this, but it actually makes the translation rules simpler than before, because some of the work usually done in the translation of list comprehensions is now done by the later, more general, cheap deforestation transformations.

Figure 3.3 gives the revised translation rules for list comprehensions. Again, the $\mathcal{TE}$ scheme translates expressions in a rich syntax, including list comprehensions, into a much simpler functional language. The rule deals with list comprehensions, by invoking the $\mathcal{TF}$ scheme. Notice that in each case a `build` is used to create

$$
\begin{aligned}
& \mathcal{TE} :: \mathit{Expr} \rightarrow \mathit{Expr} \\
\mathcal{TE}\llbracket\, [E \mid QS\,]\,\rrbracket &= \texttt{build (\textbackslash\ c n -> } \mathcal{TF}\llbracket\, [\,E \mid QS\,]\,\rrbracket\ \texttt{c n)} \\
&\ \vdots\ \ \textit{and other } \mathcal{TE} \textit{ rules}
\end{aligned}
$$

$$
\begin{aligned}
& \mathcal{TF} :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr} \\
\mathcal{TF}\llbracket\, [E \mid\,]\,\rrbracket\ c\, n &= c\,(\mathcal{TE}\llbracket\, E\,\rrbracket)\, n \\
\mathcal{TF}\llbracket\, [E \mid B\ ,\ QS\,]\,\rrbracket\ c\, n &= \ \texttt{if}\ \mathcal{TE}\llbracket\, B\,\rrbracket \\
&\ \ \texttt{then}\ \mathcal{TF}\llbracket\, [E \mid QS\,]\,\rrbracket\ c\, n \\
&\ \ \texttt{else}\ n \\
\mathcal{TF}\llbracket\, [E \mid P \leftarrow L,\ QS\,]\,\rrbracket\ c\, n &= \texttt{foldr}\ f\, n\, \mathcal{TE}\llbracket\, L\,\rrbracket \\
&\ \textit{where}\ f\ P\ b = \mathcal{TF}\llbracket\, [E \mid QS\,]\,\rrbracket\ c\, b \\
&\ \qquad\quad f\ \_\ \ b = b
\end{aligned}
$$

Figure 3.3: New list comprehension compilation rules

the final list, ready to cancel with the `foldr` from any list consumer.

The $\mathcal{TF}$ scheme is used only for list comprehensions, and has the following defining property:

$$\mathcal{TF}\llbracket\, E\,\rrbracket\ c\, n = \texttt{foldr}\ c\, n\, E$$

The $\mathcal{TF}$ scheme has three cases: either the qualifiers after the "|" are empty, or they begin with a guard B, or they begin with a generator $P \leftarrow L$ (in general P can be a pattern, not just a simple variable). Notice, crucially, that in this third case, the list L is consumed by a `foldr`, so any `build` at the top of L will cancel with the `foldr`.

3.3.3 Example of the modified list comprehension desugaring scheme

As an example of the rules translating a list comprehension, consider again the example given above:

```
[ f x | x <- map g xs, odd x]
```

The standard technology would construct an intermediate list for the result of the `map`. Using the rules in Figure 3.3 instead, desugaring the list comprehension will give:

```
build (\ c n ->
  foldr h n (map g xs)
    where
      h x b = if odd x then c (f x) b else b)
```

Unfolding map gives:

```
build (\ c n ->
  foldr h n (
        build (\ c1 n1 -> foldr (c1.g) n1 xs))
    where
      h x b = if odd x then c (f x) b else b)
```

Now we can apply the foldr/build rule, giving:

```
build (\ c n ->
 (\ c1 n1 -> foldr (c1.g) n1 xs) h n
   where
      h x b = if odd x then c (f x) b else b)
```

Now some β-reductions can be done:

```
build (\ c n -> foldr (h.g) n xs
 where
   h x b = if odd x then c (f x) b else b)
```

Lastly, foldr and build can be unfolded, and after further simplification, we get the efficient expression:

```
h' xs
  where
   h' []     = []
   h' (x:xs) =
        if odd x'
        then f x' : h' xs
        else h' xs
      where x' = g x
```

This is an efficient translation of the original expression that has succeeded in eliminating the intermediate list created by the map and consumed by the list comprehension.

3.3.4 Proof of correctness for the new list comprehension desugaring scheme

We need to show that the new rules in Figure 3.3 are consistent with the semantics of list comprehensions, as given in Figure 3.1.

Theorem

$$\forall E . \forall QS . \mathcal{TE}[\![\, [E \mid QS] \,]\!] = \mathcal{TO}[\![\, [E \mid QS] \,]\!]$$

where $\mathcal{TE}$ is from Figure 3.3, and $\mathcal{TO}$ is:

$$\begin{array}{rcl}
\mathcal{TO}[\![\, [E \mid B] \,]\!] & = & \texttt{if } B \texttt{ then } [E] \texttt{ else } [\,] \\
\mathcal{TO}[\![\, [E \mid QS_1 \, , \, QS_2] \,]\!] & = & \texttt{concat } (\mathcal{TO}[\![\, [\, \mathcal{TO}[\![\, [E \mid QS_2] \,]\!] \mid QS_1] \,]\!]) \\
\mathcal{TO}[\![\, [E \mid P \leftarrow L] \,]\!] & = & \texttt{let ok } P \texttt{ = True} \\
 & & \quad\quad \texttt{ok \_ = False} \\
 & & \texttt{in map (\textbackslash } P \texttt{ -> } E\texttt{) (filter ok } L\texttt{)}
\end{array}$$

and ok is a new name.

Proof
By induction over QS:

Base Case. $\mathcal{TE}[\![\, [E \mid Q] \,]\!] = \mathcal{TO}[\![\, [E \mid Q] \,]\!]$

The base case has two cases:

Base Case 1. $\mathcal{TE}[\![\, [E \mid B] \,]\!] = \mathcal{TO}[\![\, [E \mid B] \,]\!]$

By straightforward unfoldings, show both sides are equal to:

```
if B then [E] else []
```

Base Case 2. $\mathcal{TE}[\![\, [E \mid P \leftarrow L] \,]\!] = \mathcal{TO}[\![\, [E \mid P \leftarrow L] \,]\!]$

Starting from the right hand side:

$\mathcal{TO}[\![\, [E \mid P \leftarrow L] \,]\!]$

```
    = let ok P = True
          ok _ = False
      in map (\ P -> E) (filter ok L)
```

Now we can fuse `map` and `filter`:

```
    = let ok P = True
          ok _ = False
          f a b = if ok a then (\ P -> E : b) a else b
      in foldr f [] L
```

Next we unfold ok, and perform simplifications:

```
    = let f a b = case a of
                    P -> E : b
                    _ -> b
      in foldr f [] L
    = let f P b = E : b
          f _ b = b
      in foldr f [] L
```

Now considering the left hand side, we have:

```
$\mathcal{TE}[\![\, [\, E \mid P \leftarrow L\, ]\, ]\!]$
  = $\mathcal{TF}[\![\, [\, E \mid P \leftarrow L\, ]\, ]\!]$ (:) []
  = foldr f [] $L$
      where
        f $P$ b = $\mathcal{TF}[\![\, [\, E \mid\mid ]\, ]\!]$ (:) b
        f _   b = b
  = let f P b = $E$ : b
        f _ b = b
    in foldr f [] $L$
```

Inductive Case.

Assume

$$\mathcal{TE}[\![\, [\, E \mid QS\,]\,]\!] = \mathcal{TO}[\![\, [\, E \mid QS\,]\,]\!]$$

The goal is to show that

$$\mathcal{TE}[\![\, [\, E \mid Q,\ QS\,]\,]\!] = \mathcal{TO}[\![\, [\, E \mid Q\ ,\ QS\,]\,]\!]$$

Again there are two cases:

Inductive Case 1.

$$\mathcal{TE}[\![\, [\, E \mid B\ ,\ QS\,]\,]\!] = \mathcal{TO}[\![\, [\, E \mid B\ ,\ QS\,]\,]\!]$$

Starting from the left hand side:

```
$\mathcal{TE}[\![\, [\, E \mid B\ ,\ QS\, ]\, ]\!]$
  = $\mathcal{TF}[\![\, [\, E \mid B\ ,\ QS\, ]\, ]\!]$ (:) []
  = if $B$ then $\mathcal{TF}[\![\, [\, E \mid QS\, ]\, ]\!]$ (:) [] else []
  = if $B$ then $\mathcal{TE}[\![\, [\, E \mid QS\, ]\, ]\!]$ else []
```

And by the inductive hypothesis:

```
  = if $B$ then $\mathcal{TO}[\![\, [\, E \mid QS\, ]\, ]\!]$ else []
```

We now use the fact that concat [*xs*] equals *xs*:

```
  = if $B$ then concat [$\mathcal{TO}[\![\, [\, E \mid QS\, ]\, ]\!]$] else concat [[]]
  = concat (if $B$ then [$\mathcal{TO}[\![\, [\, E \mid QS\, ]\, ]\!]$] else [])
  = concat ($\mathcal{TO}[\![\, [\, \mathcal{TO}[\![\, [\, E \mid QS\, ]\, ]\!] \mid B\, ]\, ]\!]$)
```

which equals the definition of $\mathcal{TO}$.

Inductive Case 2.

$$\mathcal{TE}[\![\, [\, E \mid P \leftarrow L\ ,\ QS\,]\,]\!] = \mathcal{TO}[\![\, [\, E \mid P \leftarrow L\ ,\ QS\,]\,]\!]$$

Starting from the right hand side:

$\mathcal{TO}[\![\, [\, E \mid P \leftarrow L \, , \, QS \,] \,]\!]$

Using $\mathcal{TO}$ gives:

$= \texttt{concat} \; (\mathcal{TO}[\![\, [\mathcal{TO}[\![\, [\, E \mid QS \,] \,]\!] \mid P \leftarrow L \,] \,]\!])$

By using base case 1.

$= \texttt{concat} \; (\mathcal{TE}[\![\, [\mathcal{TO}[\![\, [\, E \mid QS \,] \,]\!] \mid P \leftarrow L \,] \,]\!])$

We can now use $\mathcal{TE}$ and $\mathcal{TF}$, giving:

```
= concat (build (\ c n ->
    let f P b = $\mathcal{TF}[\![ \, [\, \mathcal{TO}[\![ \, [\, E \mid QS \,] \, ]\!] \mid \,] \, ]\!]$ c b
        f _ b = b
    in foldr f n L)
= concat (build (\ c n ->
    let f P b = c ($\mathcal{TO}[\![ \, [\, E \mid QS \,] \, ]\!]$) b
        f _ b = b
    in foldr f n $L$)
```

Unfolding `concat` and using the `foldr/build` rule gives:

```
= build (\ c n ->
    let f P b = foldr c b ($\mathcal{TO}[\![ \, [\, E \mid QS \,] \, ]\!]$)
        f _ b = b
    in foldr f n $L$)
```

Now we can used the inductive hypothesis.

```
= build (\ c n ->
    let f P b = foldr c b ($\mathcal{TE}[\![ \, [\, E \mid QS \,] \, ]\!]$)
        f _ b = b
    in foldr f n $L$)
```

which, by the definition of $\mathcal{TE}$ and the `foldr/build` rule gives:

```
= build (\ c n ->
    let f P b = $\mathcal{TF}[\![ \, [\, E \mid QS \,] \, ]\!]$ c b
        f _ b = b
     in foldr f n $L$)
```

□

3.4 augment: A new super-constructor

Consider this definition of append:

```
ys ++ xs = foldr (:) ys xs
```

How do we express the list production of (++) in terms of build? We cannot just abstract over the *cons* and *nil*:

```
ys ++ xs = build (\ c n -> foldr c ys xs)        -- WRONG
```

This example is ill-typed, and correctly so. It violates our rule of constructing our lists, that we must use c to construct *all* the *cons* cells, not just the visible *cons* cells. The problem is that ys is of type list, and we want it to be of type β, as scoped by the build combinator (c.f. § 2.4).

The same problem occurs when trying to represent a lone *cons* cell using build.

```
x : xs = build (\ c n -> x `c` xs) -- WRONG
```

There are three possible solutions to this problem.

- We could use an extra foldr. For our examples ((++) and (:)), we could write:

  ```
  xs ++ ys = build (\ c n -> foldr c (foldr c n ys) xs)
  x : xs = build (\ c n -> x `c` foldr c n xs)
  ```

  However, we have introduced an extra list consumption (using foldr c n). Experience has shown that even if we are conscientious about our attempts to remove this extra consumption, *we cannot guarantee that this*. When using cheap deforestation we observed some programs that ran substantially slower, specifically because of this extra foldr. Clearly this is unacceptable, and we therefore must look for alternative schemes.

- We can have extra rules to handle problematic list creators, like (++) and (:), in a special way. We consider this in § 3.4.1.

- We can generalise the build function in such a way that we can handle this type of problematic list creation, which we do in § 3.4.2. This is the solution that we finally adopt.

3.4.1 Using extra rules

In order to handle `(++)` and `(:)`, we could lift them into our "alphabet" of combinators, and derive new rules to handle them. Currently our alphabet consists of one list producer (`build`), one consumer (`foldr`), with one rule fusing them together § 2.3. (For clarity, we ignore `foldl`, re-introducing it in § 3.5.) We could add two new producers `(:)` and `(++)`. Furthermore, because we are no longer expressing the consumption of `++`'s first argument with `foldr`, we also need to add `(++)` as an extra producer.

So we have the three list producers:

```
build (\ c n -> ...)
x : xs
xs ++ ys
```

and the two list consumers:

```
foldr f z (...)
(...) ++ zs
```

Matching each list producer with each consumer gives six rules:

```
(1)    foldr f z (build g)  = g f z
(2)    foldr f z (x:xs)     = f x (foldr f z xs)
(3)    foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs

(4)    build g     ++ zs = g (:) zs
(5)    (x:xs)      ++ zs = x : (xs ++ zs)
(6)    (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Six rules are still a manageable list removal schema. Notice that two of the rules (2) and (5) are simply instances of the original function definitions.

However, in this set of rules, the result we obtain can depend on the order in which we apply the rules. Consider:

```
foldr f z (map g xs ++ ys)
```

We can express map in terms of foldr and build giving:

```
foldr f z (build (\ c n ->
                        foldr (\ a b -> g a `c` b) n xs) ++ ys)
```

Now we have a choice of what rule to apply. If we use rule (3) (and perform β-reductions) we get:

```
foldr f (foldr f z ys) (build (\ c n ->
                            foldr (\ a b -> g a `c` b) n xs))
```

Now we can perform the foldr/build rule, giving:

```
foldr (\ a b -> g a `f` b) (foldr f z ys) xs
```

In this ordering, our extended deforestation has been successful. But reconsidering the example, before using rule (3):

```
foldr f z (build (\ c n ->
                        foldr (\ a b -> g a `c` b) n xs) ++ ys)
```

If we had chosen the other possible rule, rule (4), we would have obtained:

```
foldr f z (foldr (\ a b -> g a (:) b) ys xs)
```

There is now no further rule to apply, and we have missed a deforestation opportunity.

The problem is rule (4). After using it we do not have build or (++) at the outermost level to give a handle into the creation of the result list. We could rewrite rule (4) thus:

```
(4)     build g ++ zs = build (\ c n -> g c (foldr c n zs))
```

but this brings us back to the problem of introducing extra foldr's. It would be nice if we could capture the essence of these extended rules in a set of rules that did not have the ordering constraint, and at the same time generalise our list production capacity. This is what we do in the next section.

3.4.2 Expanding our `foldr/build` rule

In this section we introduce a new function **augment**, which can be given the definition:

```
augment g h = build g ++ h
```

where **augment** has the type

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

augment hides the bad properties of (++), allowing a new `foldr/augment` rule to achieve the nice properties of the above system, but without the ordering problem. (We give **augment** a new definition later that uses neither `build` nor (++).)

augment can be used instead of `build`, (:) and (++), when expressing list production.

- `build` can be expressed by giving **augment** an empty list as its second argument.

  ```
  build g  = augment g []
  ```

- (:) can be expressed by passing on its second argument as the second argument to **augment**.

  ```
  x : xs = augment (\ c n -> c x n) xs
  ```

- (++) can be expressed by passing on its second argument as the second argument to **augment**.

  ```
  xs ++ ys = augment (\ c n -> foldr c n xs) ys
  ```

  Furthermore, this definition of (++) is a good list consumer of `xs`. The troublesome extra traversal ys no longer happens.

  Because of these properties we call **augment** our new super-constructor.

If we express our productions using our new **augment** function, we can use the `foldr/augment` rule to eliminate intermediate lists: The `foldr/augment` rule is:

$$\boxed{\texttt{foldr}\ k\ z\ (\texttt{augment}\ g\ h) = g\ k\ (\texttt{foldr}\ k\ z\ h)}$$

The `foldr/build` rule is simply a special case of our new `foldr/augment` rule. The proof for the `foldr/augment` rule is straightforward.

Theorem
(The `foldr/augment` Rule)

$$\texttt{foldr}\ k\ z\ (\texttt{augment}\ g\ h) = g\ k\ (\texttt{foldr}\ k\ z\ h)$$

Proof

$\texttt{foldr}\ k\ z\ (\texttt{augment}\ g\ h)$
$\quad = \texttt{foldr}\ k\ z\ (\texttt{build}\ g\ \texttt{++}\ h)$
$\quad = \texttt{foldr}\ k\ (\texttt{foldr}\ k\ z\ h)\ (\texttt{build}\ g)$
$\quad = g\ k\ (\texttt{foldr}\ k\ z\ h)$

□

The definition of `augment` given above is not very efficient:

```
augment g h = build g ++ h
```

We can give `augment` a very efficient implementation. If we unfold ++, to give:

```
augment g h = foldr (:) h (build g)
```

We can now use our `foldr/build` rule, giving:

```
augment g h = g (:) h
```

which is an efficient definition that does not rely on either `(++)` or `build`.

This definition of `augment` leads to an interesting observation, namely we have already proved the `foldr/augment` rule correct in Chapter 2. When proving the `foldr/build` rule (page 27) we actually prove the equation

$$\texttt{foldr}\ k\ z\ (g\ (:)\ b) = g\ k\ (\texttt{foldr}\ k\ z\ b)$$

and then instantiate b to `[]`. This equation *is* the `foldr/augment` rule.

3.4.3 Example of deforestation using augment

We have seen how `augment` can be used to capture the essence of `build`, `(:)` and `(++)`. Consider again the definition:

```
foldr f z (map g xs ++ ys)
```

This time we can deforest without worrying about orderings. Unfolding `(++)` and `map` in terms of `foldr` and our new list producer, `augment`, gives:

```
foldr f z (
   augment (\ c n ->
        foldr c n (augment (\ c n ->
                               foldr (\ a b -> f a 'c' b) n xs) []))
    ys)
```

If we perform the outermost instance of the `foldr/augment` rule (with β-reductions), we get:

```
foldr f (foldr f z ys) (augment (\ c n ->
                                   foldr (\ a b -> g a 'c' b) n xs) [])
```

Now perform the remaining instance of the `foldr/augment` rule (again with β-reductions, etc.), gives the efficient, deforested code:

```
foldr (\ a b -> g a 'f' b) (foldr f z ys) xs
```

3.5 Summary of cheap deforestation

In the previous sections we discussed the pragmatics of consuming and producing lists. In this section we summarise cheap deforestation, giving a complete list of transformations, as well as a simple "transparency" guide for when cheap deforestation will succeed in removing an intermediate list.

3.5.1 Transformations used in cheap deforestation

In this chapter we have added a new producer (`augment`) and a new consumer (`foldl`).

- `augment` could totally supersede `build`. However, *most* examples would then be of the form `augment g []`, the definition of `build`. For this pragmatic reason, we allow `build` to remain in our alphabet of list producers.
- We have already argued that it is convenient to have `foldl` as well as `foldr`.
- We do not want to have to re-write `(:)` in terms of `augment` to handle expressions like

  ```
  foldr f z (a:b:map f xs)
  ```

  Instead we add `(:)` as a "good" list producer.

This does not mean that `build` and `augment` are redundant! The transformation system can only handle explicit consumption of `(:)` (c.f. § 2.1.2), not if `(:)` is produced recursively, etc.

- We also add `[]` to our list of "good" producers.

If we cross our four list producers with our two list consumers, we get the transformation rules that are at the heart of cheap deforestation. They are summarised in Figure 3.4.

3.5.2 The transparency of cheap deforestation

We have already seen how cheap deforestation allows programmers to write clearer programs (that use intermediate lists) relying on the compiler transforming many such programs into efficient, listless programs. If the programmer is to rely on the compiler to remove intermediate lists, it is important to give a clear specification of the conditions under which the optimisation performs deforestation.

An intermediate list is removed by cheap deforestation if:

- The list is produced by a *good producer*.
- The list is consumed by a *good consumer*.
- There is exactly one consumer.
- The producer is inlineable into the consumer.

We now define in some detail what good producers and consumers are. There is, however, a caveat that should be mentioned first. Cheap deforestation does not occur in a vacuum, but interacts with other transformations. In particular, in our implementation we decide to run a transformation called full laziness before our deforestation. Unfortunately sometimes full laziness can stop deforestation. We say more about this in § 4.2.3.

```
    List Production  augment (\ c n -> ....) h
                     build (\ c n -> ....)

                     x : xs
                     []

  List Consumption   foldr f z (...)

                     foldl f z (...)


      List Removal   foldr f z (augment g h) = g f (foldr f z h)
                     foldr f z (build g)     = g f z
                     foldr f z (x : xs)      = f x (foldr f z xs)
                     foldr f z []            = z

                     foldl f z (augment g h) =
                                 g (\ b k a -> k (f a b))
                                   (\ z -> foldl f z h)
                                   z
                     foldl f z (build g)     =
                                 g (\ b k a -> k (f a b))
                                   (\ z -> z)
                                   z
                     foldl f z (x : xs)      = foldl f (f z x) xs
                     foldl f z []            = z
```

Figure 3.4: Summary of cheap deforestation rules

3.5.3 Good producers

Good producers are:

- Any list explicitly expressed by the programmer using `build` or `augment`. For example:

```
build (\ c n -> c e1 (c e2 (c e3 n)))

augment (\ c n -> c e1 (c e2 (c e3 n))) xs
```

- Any constant list, for example:

  `[e1,e2,e3,e4]` *or* `[ ]`

- Any enumerated lists, for example:

  `[e1..]` *or* `[e1,e2..]` *or* `[e1..e2]` *or* `[e1,e2..e3]`

- Any list comprehension, for example:

```
[ f x | x <- y, g x ]
```

- Full application of the following prelude functions:

  `++`, `assocs`, `concat`, `cycle`, `elems`, `filter`, `init`, `indices`, `iterate`, `map`, `nub`, `repeat`, `reverse`, `take`, `takeWhile`, `unlines`, `unzip`, `words`, `zip`, `zipWith`

- $e_1 : e_2$ is a good producer. This includes examples like:

$$e_1 : e_2 : e_3 : \ldots$$

3.5.4 Good consumers

A good consumer is a context in which a list is ultimately consumed using `foldr`. The following are all good consumers, where Φ marks the placement of the consumed list:

- The following prelude functions:

  `all f` Φ, `and` Φ, `any f` Φ, `array (x,y)` Φ, `concat` Φ, `filter f` Φ, `foldl f z` Φ, `foldr f z` Φ, `head` Φ, `length` Φ, `map f` Φ, `null` Φ, `or` Φ, `partition` Φ, `product` Φ, `sum` Φ, `takeWhile` Φ, `unlines` Φ, `unzip` Φ.

- The first argument to (++) is a good consumer:

```
Φ++ys
```

  Furthermore, the second argument to (++) is a good consumer, if (++) itself is consumed by a good consumer.

- (:)'s second argument is a good consumer, if the fully applied (:) is itself consumed by a good consumer.

- A *generator* inside a list comprehension:

```
[ ... | ... , p1 <- Φ, ... ]
```

- `reverse` is a good consumer. However there is a caveat with its use (§ 3.5.6).

3.5.5 Common examples

Here are some common examples of Haskell expressions that have their intermediate lists removed by cheap deforestation.

- A chain of list comprehensions:

```
[ f x | x <- e1 ] ++
[ f x | x <- e2 ] ++
[ f x | x <- e3 ]
```

  The traditional scheme for translating list comprehensions (Wadler 1987*a*) has a special case for handling chains of appends. Cheap deforestation automatically includes such examples as part of its regular optimisation pattern. (§ 3.3) Also in our scheme *any* good producer can be substituted for any list comprehension in the chain of appends.

- Array Comprehensions

```
array (1,n) [ (i,i * i) | i <- [1..n]]
```

  After deforestation this builds an array without using any intermediate lists. We return to this example in § 5.2.

- General List Processing

  Functional programmers often express their problems using a pipeline. For example[1]:

  ```
  map f . takeWhile g . iterate h
  ```

  Functional programmers often express their problems using a pipeline like this example.

3.5.6 Caveat with using reverse

There is a caveat with using `reverse`. It is a good consumer of its list argument, but it can replace the list with a sequence of thunks (unevaluated suspensions). Sometimes a tangle of thunks can be created. As an example of this, consider:

```
reverse xs = build (\ c n -> foldl (flip c) n xs)
list_id xs = reverse (reverse xs)
```

First we unfold `reverse` twice in the right hand side of `list_id`:

```
list_id xs = build (\ c n ->
  foldl (flip c) n
        (build (\ c' n' -> foldl (flip c') n' xs)) n)
```

Now we use the `foldl/build` rule.

```
list_id xs = build (\ c n ->
   foldr (\ b g a -> g (\ b' g' a' -> g' (c b' a') b a)) id xs id
        n)
```

Now we can unfold `build` and `foldr`, and apply some other straightforward transformations to give:

```
list_id xs =
  let
     h []     a = a
     h (x:xs) a = h xs (\ a' -> a (x : a'))
  in
     h xs id []
```

The strange looking function works by accumulating a list abstracted over its tail. The intermediate list has simply been replaced with a sequence of suspensions.

[1] `(.)` has the definition `f . g = \ x -> f (g x)`, and can always be unfolded, possibly revealing pairs of good producer/good consumer, allowing deforestation.

We might have hoped that our transformation system could have found the apparent equality:

$$\texttt{reverse (reverse xs)} = \texttt{xs}$$

However this is not true for lazy languages, where `reverse` is spine-strict, i.e.:

$$\texttt{reverse (reverse (x:}\bot\texttt{))} \neq \texttt{x : } \bot$$

So the transformed code *needed* to descend the lists, finding its tail before returning the list.

There is an important advantage, however, of being able to handle both list production and consumption with `reverse`. Consider the function:

```
foo f g = map f . reverse . map g
```

Here, after applying deforestation, we achieve:

```
foo f g xs =
   let
       h []     xs = xs
       h (x:xs) xs = h xs (f (g x) : xs)
   in
       h xs []
```

Now the functions `f` and `g` have been brought together, and further optimisations could act on this exposed application that was previously separated by the medium of an intermediate list.

3.5.7 What our transparency algebra does not handle

This transparency models accurately when deforestation *will* occur, modulo other transformations. What it does not manage to capture is what we call second level deforestation. Consider the Haskell program:

```
map (map (+ 1)) [[1..]]
```

The model does predict the fusion of the outer `map` with the singleton list. However it does not predict that the inner `map` will then fuse with the enumeration `[1..]`

It is hard to see how the transparency model could be expanded to cope with this sort of "second level" deforestation, without including *every* optimisation the compiler does in the model, which is obviously impractical.

Chapter 4

Implementing Cheap Deforestation

In this chapter we explain how to implement cheap deforestation inside a real compiler. To allow us to explore a concrete implementation we add our list removal technique to the Glasgow Haskell compiler (GHC). Adding our optimisation to a *real* compiler allows us to examine the *real* problems and let us perform cheap deforestation on *real* examples. We start by describing GHC, concentrating on the internal intermediate language, which is the grammar we will be handling (§ 4.1). We then discuss adding the cheap deforestation transformations to GHC (§ 4.2).

We then go on to give some more details about some aspects of our implementation that we added because of feedback we received for examining the output code. We need to enhance how we inline through lambdas to achieve successful deforestation, which we explain in § 4.3. Then we explain how to perform *arity expansion* on recursive functions, in § 4.4. Finally, we make some changes to allow the successful handling of `build` inside our transformation system, in § 4.5.

4.1 The Glasgow Haskell compiler

The Glasgow Haskell compiler is an industrial strength compiler. It is a result of the GRASP and AQUA projects at Glasgow University. The compiler has intentionally been written to be a "motherboard", so it is straightforward to "plug in" a new optimisation.

The Glasgow Haskell compiler expresses its various compilation stages as correctness preserving transformations. This paradigm of compilation via transformation is common in the functional language compiler community (Appel 1992,

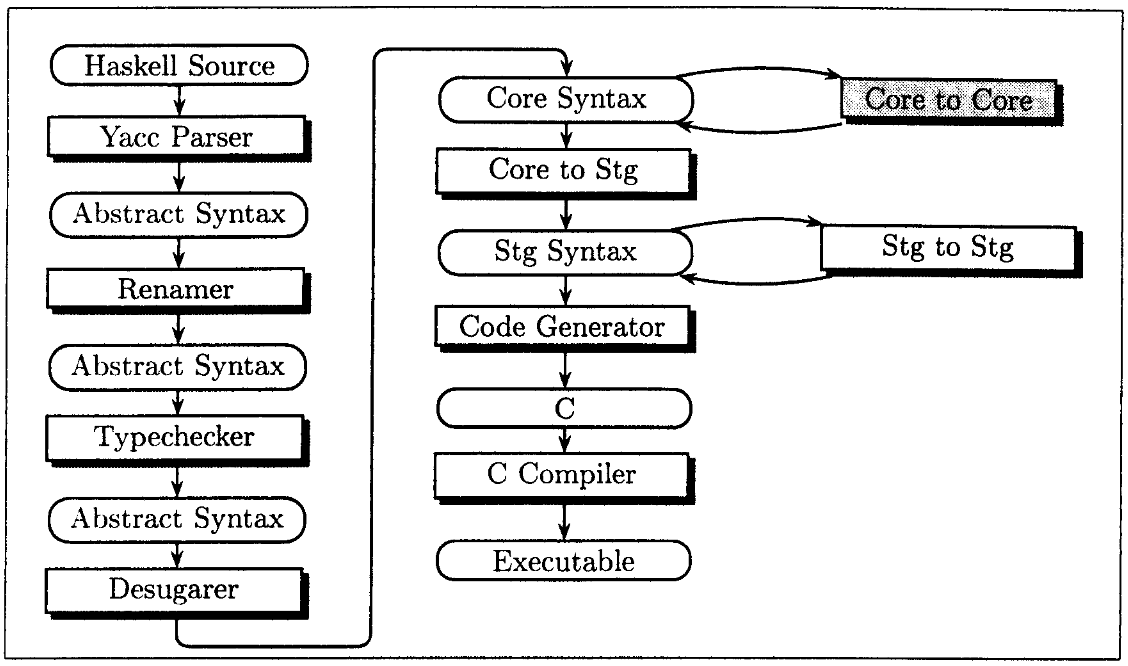

Figure 4.1: Components in the Glasgow Haskell compiler

Fradet & Metayer 1991, Kelsey 1989). These transformations can either translate from one syntax to a simpler syntax, or perform an optimising transformation that transforms a single syntax, making the program more efficient.

4.1.1 Organisation of the Glasgow Haskell compiler

Figure 4.1 illustrates the principal components of the compiler. Apart from the parser and C compiler, all the components are written in Haskell. These components have the following functions:

1. A *Yacc parser* (Johnsson 1983) reads the Haskell program, and passes the abstract syntax tree to the compiler proper.
2. A *renamer* resolves naming issues, such as name scoping and information propagation across module boundaries.
3. A *type inference pass* annotates the program with type information, using an extended variation of the Hindley-Milner type inference algorithm (Wadler & Blott 1989).
4. A *desugaring* pass converts the full Haskell syntax to a straightforward functional language, called the *Core language*.

5. Several different *Core-to-Core* optimising transformations are performed. This component (which is shaded in Figure 4.1) is the natural place to add the `foldr/build` rule.
6. Core is converted into an even simpler language, called the Shared Term Graph (STG) language.
7. Further transformations are done to this simple language.
8. Finally a code generator turns the STG language into C. This target language, C, can then be compiled to an executable via any C compiler.

4.1.2 The Core language in the Glasgow Haskell compiler

The part of the compiler we are concerned with contains a series of optimisation passes expressed as Core to Core transformations. The Core language is an augmented second order lambda calculus, whose syntax appears in Figure 4.2.

Core was designed to be used by optimising transformations. Because of this the Core language was intentionally kept to a minimal set of constructs. This allows optimisers to work in a world free from the syntax clutter of larger languages, such as Haskell itself. Using the second order λ-calculus as a starting point, `let`, `case` for single-level patterns, explicit constructors and primitives were added to handle modern functional languages efficiently (Peyton Jones 1987).

- In Core, `let` is used to represent allocations:

  ```
  let
     h = <exp>
  in
     ...
  ```

  This code means that a suspension for `<exp>` is allocated in the heap.

- `case` is used to trigger evaluations:

  ```
  case f x of
     True -> <exp1>
     False -> <exp2>
  ```

  This mean the `f x` is evaluated, and then `<exp1>` or `<exp2>` is evaluated, depending on the result of `f x`.

Program	$Prog$	$\rightarrow$	$Binding_1$; ... ; $Binding_n$	$n \geq 1$
Bindings	$Binding$	$\rightarrow$	`nonrec` $Bind$	
		$\mid$	`rec` $Binds$	
	$Binds$	$\rightarrow$	$Bind_1$; ... ; $Bind_n$	$n \geq 1$
	$Bind$	$\rightarrow$	v `=` $Expr$	
Expressions	$Expr$	$\rightarrow$	$Expr\ Atom$	Application
		$\mid$	$Expr\ ty$	Type application
		$\mid$	$\lambda\ v_1 \ldots v_n$ `->` $Expr$	Lambda abstraction, $n > 0$
		$\mid$	$\Lambda\ ty$ `->` $Expr$	Type abstraction
		$\mid$	`case` $Expr$ `of` $Alts$	Case expression
		$\mid$	`let` $Binding$ `in` $Expr$	Local definition
		$\mid$	$Con\ Atom_1 \ldots Atom_n$	Constructor, $n \geq 0$
		$\mid$	$prim\ Atom_1 \ldots Atom_n$	Primitive, $n \geq 0$
		$\mid$	$Atom$	
Atoms	$Atom$	$\rightarrow$	v	Variable
		$\mid$	$Literal$	Unboxed Object
Literal values	$Literal$	$\rightarrow$	$integer \mid float \mid \ldots$	
Alternatives	$Alts$	$\rightarrow$	$Calt_1$; ... ; $Calt_n$; $Default$	$n \geq 0$
		$\mid$	$Lalt_1$; ... ; $Lalt_n$; $Default$	$n \geq 0$
Constr. alt	$Calt$	$\rightarrow$	$Con\ v_1 \ldots v_n$ `->` $Expr$	$n \geq 0$
Literal alt	$Lalt$	$\rightarrow$	$Literal$ `->` $Expr$	
Default alt	$Default$	$\rightarrow$	v `->` $Expr$	
		$\mid$	ϵ	

Figure 4.2: Syntax of the Core language in the Glasgow Haskell compiler

- A global objective inside GHC is to maintain type information right through to the code generator. In order to allow arbitrary transformations within the typed Core language, the language is based on the second order λ-calculus. Because of this, it is possible to straightforwardly derive the complete type of any object. The `foldr/build` rule can easily be expanded into the second order λ-calculus.

- The arguments of applications are atomic. This allows optimsation systems to have a smaller set of transformations, because there is a canonical way to express Haskell expressions like `f (g 2)`, specifically:

  ```
          let t = g 2 in f t
  ```

  If arbitrary expressions were allowed on the right hand side of an application, then there would be two ways of expressing the above expression, and there would need to be two cases inside any optimiser to deal with the differences.

4.1.3 Desugaring Haskell to Core

The desugaring pass transforms the syntactic baggage from the large Haskell language, translating it into the more restricted Core language. The implementation inside GHC adheres closely to Peyton Jones (1987). Specifically, the desugaring pass performs the following transformations:

- Pattern matching is converted into single level `case` operations.

- Haskell `let` and `where` declarations are converted into the equivalent, single level `let`.

- Constructors and primitives are saturated, if necessary by adding extra lambdas.

- The arguments of applications, constructors, and primitives are made atomic, by `let`-binding any non-atomic arguments.

- Information from the type-checker is used to add type lambdas and type applications.

- List comprehensions are translated into recursive list producers and consumers. The compiler previously employed a traditional scheme to perform this operation (Wadler 1987*a*, Augustsson 1987). We replace this scheme with the "deforestation-friendly" scheme, as presented in (§ 3.3).

As an example of Core, consider the Haskell function:

```
reverse xs = rev xs []
 where
  rev (x:xs) ys = rev xs (x:ys)
  rev []     ys = ys
```

After desugaring, this function would be expressed as:

```
reverse :: \/ a . [a] -> [a]
reverse = /\ ty ->
           \ t ->
             let
               rev = \ t ys ->
                       case t of
                           x : xs -> let t1 = (:) ty x ys
                                     in rev xs t1
                           [] -> ys
             in
               rev t []
```

Here we use /\ to represent Λ, \ to represent λ, and \/ to represent ∀.

The Core program is more verbose than the original Haskell program! For reasons of clarity we transliterate as many as possible future examples back into a more Haskell-like syntax wherever possible. We do this by omitting the explicit type applications/abstractions, and take liberties with the atomic argument rule of Core. So, the reverse example, in our "concise" Core would be written:

```
reverse :: [a] -> [a]
reverse = \ t ->
          let
             rev = \ t ys ->
                     case t of
                        x : xs -> rev (x : ys) xs
                        [] -> ys
          in
              rev xs []
```

4.2 Adding the foldr/build rule to the Glasgow Haskell compiler

We now can add cheap deforestation into the Glasgow Haskell compiler. As we have seen for our earlier examples, there are three key steps to our deforestation technique:

1. Providing `foldr` and `build` (and `foldl`, and `augment`) versions of prelude functions, and unfolding these definitions.
2. Performing any instances of the rules given in Figure 3.4, as well as performing general purpose simplifications.

   We have already observed that performing simplifications *after* using the `foldr/build` rule can reveal new instances of the `foldr/build` rule (§ 2.3.3). Because of this, we want to add cheap deforestation into the simplification framework provided by the general purpose simplifier, as extra transformations.
3. Cleaning up after deforestation. This also involves general purpose simplifications, as well as possibly unfolding `foldr`, `foldl`, `build` and `augment`.

   Again we want to use the current simplifier to do this.

So, in summary, we want to run a modified version of the simplifier twice, once to performing cheap deforestation reductions, and once to unfold `foldr`, `build`, etc.

In this section we first explain how we extended the Haskell typechecker to handle `build` and `augment` (§ 4.2.1). Then, in § 4.2.2, we explain how the current simplifier works, and how to add the cheap deforestation rules to it. Then we consider the best time to use cheap deforestation in relation to the other components of GHC's optimisation system in § 4.2.3. Finally, in § 4.2.4 we consider how to handle Strings, which have a compact representation in GHC.

4.2.1 Adding `build` and `augment` to Haskell programs

Haskell has a type system based on the Hindley-Milner type system. In the Hindley-Milner system all type variables are quantified at the outer level[1]. So, for example, the Haskell type

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

really means:

$$\texttt{foldr} :: \forall\,\alpha\,.\,\forall\,\beta\,.\,(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$$

`build`, however, does not have a type that obeys this restriction.

$$\texttt{build} :: \forall\,\alpha\,.\,(\,\forall\,\beta\,.\,(\alpha \to \beta \to \beta) \to \beta \to \beta\,) \to [\alpha]$$

[1]In Haskell things are actually much more complicated, because of overloading. However, this is irrelevant to the concepts behind our new type rules.

$$\textit{var} \qquad \Gamma, x : S \vdash x : S \qquad x \notin \{\texttt{build}, \texttt{augment}\}$$

$$\textit{build} \qquad \frac{\Gamma \vdash g : \forall t.\ (T \rightarrow t \rightarrow t) \rightarrow t \rightarrow t}{\Gamma \vdash (\texttt{build}\ g) : [T]} \qquad t \notin FV(T)$$

$$\textit{augment} \qquad \frac{\Gamma \vdash g : \forall t.\ (T \rightarrow t \rightarrow t) \rightarrow t \rightarrow t}{\Gamma \vdash (\texttt{augment}\ g) : [T] \rightarrow [T]} \qquad t \notin FV(T)$$

Figure 4.3: Type rule extensions for `build` and `augment`

Fortunately, there is a simple extension to the Haskell type system that can be made to allow `build` to have a correct type. Figure 4.3 gives two extra rules, one for `build` and one for `augment`. We also need to check that we do not attempt to assign a type to any occurrences of `build` and `augment` that are not fully applied.

4.2.2 Modifying the simplifier

A general purpose simplifier (Santos 1995) performs many small, local optimisations, like β-reductions and `let`-inlining. There are two stages to the simplifier.

- A pre-processor performs occurrence analysis on the program, computing when something can be inlined. For example, consider:

```
let
   v = <exp>
in
   v x
```

  In this example `v` could be inlined. We say more about the occurrence analyser in § 4.3.

- Guided by this occurrence information, a second pass, the simplifier proper, performs many straightforward transformations in a single pass over the program, like β-reductions and `case`-reduction.

These two passes are repeatedly performed on the Core syntax, until either no more simplifications are possible, or a fixed maximum number of iterations has been reached.

One important function of the simplifier is to "clean up" programs, by removing dead code or inlining trivial `let` bindings. This also frees other Core-to-Core passes from concerning themselves with the cleanliness of the code they produce. For example, other optimsation passes can rely on the dead code elimination in the simplifier. Furthermore many optimisations expose new opportunities for local optimisations to be used. Because of these roles, the simplifier, as well as being an optimiser in its own right, is also used as "mortar" separating several different optimisations.

Adding the cheap deforestation rules extra transformations is straightforward within the framework of the simplifier. `foldr`, `foldl`, `build` and `augment` are "tagged", so that the compiler can spot them as *special* identifiers. Inside the simplifier any application to `foldr` (and `foldl`) has its atomic argument tested, to see if it is bound to a `build` (or `augment`). For example:

```
let
    v = build g
in
   foldr f z v
```

In this case `foldr`'s third argument is bound to `build g`, and the occurrence analyser would marked `build g` as suitable for inlining, so the `foldr/build` transformation can be used, to give:

```
g f z
```

We discuss what makes a binding suitable for inlining in § 4.3.

4.2.3 When to use the `foldr/build` rule inside the Glasgow Haskell compiler

We have already decided to run a modified version of the simplifier twice, once to perform cheap deforestation reductions, and once to unfold `foldr`, `build`, etc. We need to answer the question, however, of *when* we should use these two distinct versions of the simplifier. The principal factors in this decision are:

- We should run full-laziness before deforestation. For example, consider:

  ```
  foo f = map f (map g xs)
  ```

  where `g` and `xs` are free variables inside the function `foo`. If we use full laziness, we could transform this to:

```
v = map g xs
foo f = map f v
```

If we instead performed deforestation *before* full laziness, we would get the definition:

```
foo f =
  let
     h [] = n
     h (a:as) = f (g a) : as
  in
     h xs
```

Now the two maps have fused. We cannot pull out an expression that computes the intermediate list only once, because there now is no intermediate list.

So which should have precedence, full laziness or deforestation? There is evidence to suggest that full laziness should be run first. Consider if the free variables xs and g were bound to:

```
xs = [1..n]
g x = if x > 2 then g (x-1) + g (x-2) else 1
```

Running full laziness first results in the program building the list

$$g\ 1\ :\ g\ 2\ :\ g\ 3\ :\ \cdots$$

once, while if deforestation was performed first, the list would be re-created each time.

This behaviour is intuitive. Deforestation merges producers and consumers together. If the consumer or the producer could have its evaluation shared, this might produce larger savings than just removing the intermediate list between them. Informally we argue that typically it is the elements of a list that are expensive to compute, compared to actually building the list.

- We would like to run strictness analysis *after* cheap deforestation. This is because unfolding foldr, which is done after performing cheap deforestation, produces functions that are more 'strictness friendly'. One concrete example is:

```
foo n = sum [1..n]
```

Ultimately, after cheap deforestation, (and other optimisations we discuss later) this reduces down to:

```
foo n =
 let
  h x a =
   if x < n
   then h (x+1) (a + x)
   else a
 in
   h 1 0
```

The strictness analyser can exploit the strictness of both of h's arguments, giving very efficient code. Doing cheap deforestation after strictness analysis would deny us this important optimisation opportunity.

Based on these two criteria, we have modified two specific runs of the simplifier, one to perform, as well as its normal duties, cheap deforestation after full laziness, and another after the first, but before strictness analysis, to inline `foldr` and `build`, etc.

4.2.4 Handling strings

In Core, literal strings are represented by a *Literal* that contains the string. We notate this with:

```
"hello, world"L
```

During translation from Core to the STG language literal strings are transformed into vectors of bytes. They are wrapped in a function that unpacks the vector at runtime into a list of `Chars`. We notate a vector of bytes with

```
"hello, world"#
```

We can handle strings as good producers by wrapping the unpacking of the string up in a function written with `build`. We transform:

```
"hello, world"L
```

into

```
build (\ c n -> unpackWith c n "hello, world"#)
```

allowing the deforestation of literal strings.

4.3 The enhanced occurrence analyser

In this section we give some extensions to GHC's occurrence analyser. In § 4.3.1 we explain the current occurrence analyser. In § 4.3.2 we explain why the occurrence analyser needs to be enhanced. In § 4.3.3 we introduce our enhancement.

4.3.1 The original occurrence analyser

The occurrence analyser counts statically the number of occurrences of each bound variable in such a way that the simplifier can decide on suitability for inlining. As an example, consider:

```
let
   f = \ v -> <expr>
in
   f 3
```

The occurrence analyser annotates the binding of `f`, stating that it is only used once, and suitable for inlining. The simplifier will then transform this example to:

```
(\ v -> <expr>) 3
```

Further iterations of the simplifier allow further simplification, in this case a β-reduction.

However, a simple "head count" is not sufficient for determining suitability for inlining. Consider:

```
let
   f = g 2
   h = \ x -> ... f ...
in
   h 1 + h 2
```

Now, although `f` only occurs once, it could be *used* many times. If `f` was naïvely inlined into `h`'s right hand side the result of the computation `g 2` would occur twice!

```
let
   h = \ x -> ... g 2 ...
in
   h 1 + h 2
```

The bottom line is that it is *unsafe* to inline a redex through arbitrary λ's. (From now on, we use the terms "safe" and "unsafe" in the context of the possible duplication of work caused by inlining through λ's.)

The occurrence analyser annotated all binders with information about how that definition is used, taking into account recursive bindings. This annotation can be one of three things.

- **DeadCode**. This requests that this binding be discarded, because it is never used.
- **Inlineable**. This is where a binding is only used once, and in such a way that it could be safely inlined.
- **Many Occurrences**. Everything else falls into this category.

Using this information the simplifier can remove dead code, and optionally inline the right hand side of a binding. A binding would typically be inlined if it was very small (and not a redex), or occurred only once in its scope in a *safe* location, as determined by the occurrence analyser.

4.3.2 Why `map` of `map` fails

Unfortunately, there are cases where we want to inlining through λ's to maximise deforestation opportunities. Consider the expression:

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

When we examine the transformations taking place inside our compiler when compiling this expression, we find that extra transformations, that allow selective inlining through λ's, are need. The `map` of `map` example transforms into:

```
let
   v = map g xs
in
   map f v
```

We first inline our good producer and consumer version of `map`, which has the definition:

```
map = \ f xs -> build (\ c n -> foldr (\ a b -> f a `c` b) n xs)
```

Inlining this into `map f (map g xs)` gives

```
let
   v = (\ f xs -> build (\ c n ->
                        foldr (\ a b -> f a `c` b) n xs)) g xs
in
   (\ f xs -> build (\ c n ->
                    foldr (\ a b -> f a `c` b) n xs)) f v
```

We now turn the saturated λs into `lets`, using the *safe* transformation:

$$(\lambda\ v \to e_1)\ e_2 \quad = \quad \texttt{let}\ v = e_2\ \texttt{in}\ e_1$$

This gives the expression:

```
let
  v = build (\ c n -> foldr (\ a b -> g a `c` b) n xs)
in
  build (\ c n -> foldr (\ a b -> f a `c` b) n v)
```

To allow deforestation of `map f (map g xs)`, we need to inline the right hand side of `v`, through the λ `c n`. Unfortunately, this inlining is, as we discussed in § 4.3.1, in general unsafe. The problem is that inlining a redex through a lambda can lose laziness. In this specific case, however, it is perfectly safe to inline the right hand side of `xs`: This can be illustrated by considering what would happen if we inlined the lower `build`:

```
let
  v = build (\ c n -> foldr (\ a b -> g a `c` b) n xs)
in
  (\ c n -> foldr (\ a b -> f a `c` b) n v) (:) []
```

Now we can perform β-reductions, eliminating the troublesome λs.

```
let
  v = build (\ c n -> foldr (\ a b -> g a `c` b) n xs)
in
  foldr (\ a b -> f a : b) [] v
```

Now it is easy to determine the `v`'s can be inlined, because there are no λs to cross. After this inlining, an instance of the `foldr/build` rule will occur, and the intermediate list between the two `maps` will be deforested.

4.3.3 The enhanced occurrence analyser

We now introduce a simple extension to GHC's occurrence analyser, that exploits properties of `build`, but without needing to inline `build` to get the benefits.

Specifically, we observe that for `build`:

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

In other words, if `build`'s argument has two λs around it, they *will* be saturated and removed when `build` is unfolded. So it is possible to inline through these two specific λs without losing laziness.

Our extensions to the occurrence analysis exploit these properties. We give every expression a "type" that represents the dynamic properties it has. Types are of the form τ, where τ is defined as:

$$\begin{aligned} \tau &= \odot \mid \eta \rightarrow \tau \\ \eta &= \perp \mid 1 \mid 2 \mid \ldots \end{aligned}$$

$\odot$ means unknown. $3 \rightarrow \odot$ means that the expression with this "type" will take (at least) one argument, it will enter that argument only once, and there will be (at least) three arguments to this argument. $\perp \rightarrow \odot$ means that you will take an argument, but you can assert nothing about it. When analysing applications, if the left hand side has "type"

$$\eta \rightarrow \odot$$

then it is safe to inline through the first η λs. `build` has the "type"

$$2 \rightarrow \odot$$

and `augment` has the "type"

$$2 \rightarrow \perp \rightarrow \odot$$

The implementation of the enhanced occurrence analyser simply carries a counter for the number of λs that can be inlined through. Our algorithm does not handle the automatic derivation of "types", rather we explicitly wire in the type of `build` and `augment`.

4.4 Arity analysis

Sometimes after using cheap deforestation programs have further scope for optimisation. Consider this code fragment:

```
sum [1..n]
```

Cheap deforestation can straightforwardly remove the intermediate list, but the resulting code has an implicit accumulating parameter:

```
let
   h :: Int -> Int -> Int
   h = \ x ->
           if x < n
           then let v = h (x+1)
                in \ y -> v (x+y)
           else \ y -> y
in
  h 1 0
```

Now we would like to change the arity of `h`, lifting the `\ y ->` through to beside the `\ x ->`, making this accumulating parameter explicit. However, this is not straightforward. Consider the sub-expression:

```
let v = h (x+1)
in \ y -> v (x+y)
```

If we could inline `v` through `\ y ->`, we could apply the translation

```
if x < n                                \ y -> if x < n
then \ y -> h (x+1) (x+y) ==>                  then h (x+1) (x+y)
else \ y -> y                                  else y
```

However `h (x+1)` is a redex, and we cannot inline redexes though λs. But we are trying to increase the arity of `h`, and it would no longer be a redex if we succeeded! The fact that `h` currently has an arity of 1 is stopping us increasing its arity to 2.

Our algorithm for getting round this problem works like this:

For each `let` binding

- First we look at the body of the `let`, to see the "optimal" number of arguments a binding has.
- If the binding's right hand side does not have at least this number of λs, we then try to prove that extra λs can be added.

- If the proof is successful we add more λs to the binding's right hand side, and thus increase the arity of the binding.

We are using a simple fixpointing technique. In our example we presume that h has the arity we want it to have, in this case 2, because h's initial application with two arguments. Then we analyse h's right hand side, carrying this assumption. Most of the time (as in this case) the assumption is demonstrated to be correct. Sometimes, however, the assumption is wrong, and we have to re-analyse the right hand side of the let-binding with a weaker assumption.

Consider our example:

```
let
   h :: Int -> Int -> Int
   h = \ x ->
           if x < n
           then let v = h (x+1)
                in \ y -> v (x+y)
           else \ y -> y
in
  h 1 0
```

We want to *prove* that h can have an arity of two without losing laziness. So we carry this information into the right hand side of h. Now h is found in the form:

```
let v = h (x+1)
in \ y -> v (x+y)
```

It appear that h is only applied to *one* argument here. But if we presume that h has an arity of two, it is safe to inline v through the λ, and get:

```
\ y -> h (x+1) (x+y)
```

and we have our desired recursive call of h, but now with two arguments. We do not actually perform this inlining during arity analysis, but maintain sufficient information to infer what variables can be inlined through what λs, given the pre-conditions like presuming h has an arity of two. Now we can apply further, traditional optimisations, giving:

```
let
   h :: Int -> Int -> Int
   h = \ x y ->
           if x < n
           then h (x+1) (x+y)
           else y
in
  h 1 0
```

4.5 Handing `build` within Core transformations

In this section we explain a small number of changes we had to make to our transformation system to allow it to interact amicably with `build` and `augment`. Everything below applies equally for `build` and `augment`, but we just talk about `build` for clarity.

Firstly, we need to make `build` "float" in a special way (§ 4.5.1). Secondly, we need to be careful about inlining definitions that are of the form "`build g`" (§ 4.5.2).

4.5.1 Let floating `build`

Consider the Haskell expression

```
let
    v = build (\ c n -> <exp>)
in
  ...
```

There are two alternative ways to represent this in Core.

```
let
   v = let g = \ c n -> <exp>
       in build g
in
    ...
```

or

```
let
   g = \ c n -> <exp>
in
   let
      v = build g
   in
      ...
```

The first alternative can save a heap allocation (for g's closure) when v is not entered. The second always allocates g's closure, but because there are two consecutive allocations, the compiler can optimise both allocations, and only perform one heap check. Santos (1995) found that first alternative was "better", except when v was a Weak Head Normal Form (WHNF), when the second alternative gave better code, partly because it enhances optimisation opportunities.

As we explained in § 4.2.2, the `foldr/build` rule works by examining the right hand side of the binding of the third argument of `foldr`. We would rather the second alternative representation above, because it makes spotting `build` much easier. We modified the simplifier to treat `build g` as a special case, allowing us to use the second of the two alternatives.

4.5.2 Duplicating build

There is a caveat with inlining expressions of the form

```
build g
```

Though this is a "small" expression, performing inlinings that duplicate this is undesirable, *even if the duplication is safe*. For example, consider

```
let
   g = \ c n -> < big expression >
   v = build g
in
   case f x of
     True -> v
     False -> v
```

Because v's right hand side is small, and v only occurs once down each branch, v could be inlined. But if v is inlined, we get:

```
let
   g = \ c n -> < big expression >
in
   case f x of
     True -> build g
     False -> build g
```

Now when we inline `build` we get:

```
let
   g = \ c n -> < big expression >
in
   case f x of
     True -> g (:) []
     False -> g (:) []
```

There is now no (straightforward) way of replacing `c` and `n` with `(:)` and `[]`, because the inlining mechanism is reluctant to duplicate the large right hand side of g. This is inferior to the alternative if we did not inline v:

```
let
   g = \ c n -> < big expression >
   v = g (:) []
in
   case f x of
     True -> v
     False -> v
```

Now we can safely inline g and perform β-reductions on the result.

We fix this problem by encouraging the simplifier *not* to inline `build g` down different branches of a `case`.

Chapter 5

Measuring Cheap Deforestation

In this chapter we quantify performance gains from using our implementation of cheap deforestation. What are the important characteristics of a good optimisation? We use the explanation from Aho, Sethi & Ullman (1986), which we abridge here:

- Firstly, **an optimisation must preserve the semantics of the program.** This issue has already been addressed in § 2.4 and § 3.4.2.
- Secondly, **an optimisation must, on average, reduce the resource requirements of an optimised program.** Resources include run-time, heap usage and heap residency. This chapter addresses this with quantitative measurements.
- Finally, **an optimisation must be worth the effort!** This applies in two ways. It must be worth the compiler implementers' time to add a particular optimisation, and the optimisation must be sufficiently efficient to not unduly affect the average compilation time.

This chapter is concerned with the measurable aspects of the second and third points. We examine the usefulness of cheap deforestation in three parts. We measure a couple of small, hand-picked examples that cheap deforestation works well on § 5.1. We then see how cheap deforestation allows very efficient array creation § 5.2. Finally, we measure the effect of cheap deforestation over a wide range of benchmarks.

The benchmarks in this chapter were performed on various[1] SparcStations, as resources allowed. A typical SparcStation was a Sparc 5 with 64 Megabytes of

[1] Individual benchmarks were measured on the *same* machine, to allow fair comparisons.

memory. For the crude, order of improvement measurements (§ 5.1) we simply used the Unix `time` command. For our more detailed benchmarking (in § 5.4) we used the actual number of native code instructions executed, to filter out noise from other processes, network traffic, etc.

5.1 Deforestation on hand-picked benchmarks

In this section we look at how cheap deforestation improved a couple of example benchmarks.

5.1.1 Queens

Queens was the one and only benchmark used in Gill et al. (1993). The benchmark was:

```
main = (print.sum.concat.queens) 10
  where
    queens :: Int -> [[Int]]
    queens 0 = [[]]
    queens m = [ p ++ [n] | p <- queens (m-1),
                            n <- [1..10],
                            safe p n]

    safe :: [Int] -> Int -> Bool
    safe p n = and [ (j /= n) && (i + j /= m + n)
                              && (i - j /= m - n)
                            | (i,j) <- zip [1..] p]
      where m = length p + 1
```

This program compiled without cheap deforestation turned on this benchmark took 9.4 seconds to run, and allocated a total of 61.3 megabytes of heap. Compiled with cheap deforestation turned on the benchmark took 5.4 seconds, and allocated a total of 15.6 megabytes of heap. It is interesting to compare these with the results reported in Gill et al. (1993).

> "The original program run without our optimisation, and averaged over several runs, took 24.4 seconds and consumed 179 megabytes of heap. The transformed program under the same conditions ran about three times faster (8.8 seconds) and allocated only 20% as much heap (36 megabytes)."

Though direct time comparisons are not valid (because the benchmarking machines are different) the differences in heap figures are interesting. Other optimisations inside the compiler are now much more aggressive. In particular, full laziness will be contributing towards this improvement.

5.1.2 A ray-tracer

The current release of the Glasgow compiler (0.26) contains a slightly earlier incarnation of cheap deforestation, which is turned on by default. One Glasgow Haskell compiler user is performing a systematic study of a ray-tracer written in Haskell, in particular, measuring the benefits of deforestation (Kort 1996). His goal is to "create an elegant, but practical program with many higher order functions and list comprehensions". Kort argues that deforestation should have a great effect on such programs.

We obtained Kort's ray-tracer, and verified his claims. The ray-tracer was about 1000 lines of commented code. Kort identified that a critical function (the function that was at the innermost "loop") was part of matrix multiplication, and called `vecDot`:

```
vecDot :: [Double] -> [Double] -> Double
vecDot v1 v2 = sum (zipWith (*) v1 v2)
```

Putting this function though the Glasgow Haskell compiler with cheap deforestation turned on gives:

```
vecDot :: [Double] -> [Double] -> Double
vecDot v1 v2 =
  let
     h as bs k =
       case as of
         [] -> k
         (a:as') ->
            case bs of
              [] -> k
              (b:bs') -> h as' bs' (a*b+k)
   in
      h v1 v2 0
```

This efficient, deforested rendition of the key function results in substantial performance improvement. We measured the ray-tracer over one example; Kort (1996) states that the improvement is relatively constant over different examples. Without cheap deforestation the ray-tracer took 57.6 seconds to run, and allocated a total of 494 megabytes of heap. With cheap deforestation the ray tracer took 36.0 seconds, and allocated a total of 212 megabytes of heap.

Both these examples are encouraging, and highlight the potential of deforestation if it can optimise the critical components of a program.

5.2 Compiling array comprehensions

In this section we explore how to optimise Haskell array comprehensions. We take our deforestation technique, and show how, combined with a system for expressing "update" in a functional language, deforestation can contribute towards efficient Haskell arrays. The consequence of deforestation on the overall usefulness of functional languages for certain specialised applications (like numerical computations) is important and far reaching.

5.2.1 Deforesting a simple array comprehension

Haskell arrays are monolithic. This means that the encouraged style of programming is to declare the contents of an array in one go, rather than the more traditional style of incrementally filling in the contents of an array. We are concerned with the efficient construction of these monolithic arrays.

Arrays in Haskell are constructed using the prelude function `array`. This function takes as its arguments the size of the array and a list of index-value pairs. A useful technique for building these index-value pairs is to use a list comprehension. For example, this Haskell function constructs an array, where for each index location the array holds the square of its index:

```
fun :: Int -> Array Int Int
fun n = array (1,n) [ (i,i * i) | i <- [1..n]]
```

(Technically, in Haskell 1.2 we use the infix paring constructor `:=` rather than the 2-tuple.) It was claimed in Anderson & Hudak (1990) that a deforestation scheme would be an integral part of an efficient implementation of Haskell arrays. In this example, deforestation has two tasks:

- Deforestation has the job of removing the intermediate list produced by the `[1..n]` and consumed by the list comprehension. This is a straightforward application of cheap deforestation.
- Deforestation has the job of removing the intermediate list produced by the list comprehension, and consumed by `array`. Expressing a list comprehension in terms of `build` is again a straightforward application of cheap deforestation. Therefore using deforestation on this example can be achieved by writing `array` as a good consumer.

The Glasgow Haskell implementation of the **array** function performs the following steps:

- First, a new mutable array is allocated, with every location in this array pointing to a suspension of the value $\perp$.
- The index-value pair list is consumed, with every pair generating a modification to the mutable array.
- Finally, the array is frozen (made immutable) before returning it.

The technical aspects of handling mutable arrays in a pure functional language are explained in detail in Launchbury & Peyton Jones (1996). Of course, this is only one possible implementation of `array`.

Writing `array` as a good producer is reasonably straightforward. At the heart of `array`, we use `foldl` to consume the index-value pairs,

```
foldl fill_one_in s ivs
```

where `s` is the state token, and `ivs` is the index-value pair list. `fill_one_in` has the specification:

```
fill_one_in s (i,v) = writeArray arr (index ixs i) v s
```

One interesting aspect to this implementation is the reuse of the expanded occurrence analysis in § 4.3. The stateful computation (i.e. the building and updating of the mutable array) is "fired up" using a built in function called **runST**. Internally, uses of **runST** produce expressions of the form:

```
runST (\ s -> ... )
```

A problem is that the consumption of our index-value pairs is inside this internal λ, but the production of the index-value pairs is outside the λ.

```
let xs = < creation of index-value pairs >
in
  runST (\ s -> ... foldl f z xs ... )
```

We reuse exactly the technology from § 4.3, giving **runST** the type

$$1 \rightarrow \odot$$

5.2.2 Compiling a simple array comprehension

When we apply all our optimisations to the original example,

```
fun :: Int -> Array Int Int
fun n = array (1,n) [ (i,i * i) | i <- [1..n]]
```

the compiler produces the following code for the inner loop that updates the values in the array.

```
loop :: Int# -> State# s -> _State s
loop = \ x#  world# ->
 case (leInt# x# top#) of
  True ->
   case (timesInt# x# x#) of
    y# ->
     case (plusInt# x# 1#) of
      z# ->
       case (leInt# 1# x#) of
        True ->
         case (leInt# x# top#) of
          True ->
           case (minusInt# x# 1#) of
            i# ->
             let v = I# y#
             in case (writeArray# arr# i# v world#) of
                 world'# -> loop z# world'#
          False -> _rangeComplaint_Ix_Int x# 1# top#
        False -> _rangeComplaint_Ix_Int x# 1# top#
  False -> S# world#
```

Int#'s are *unboxed* Int's (Peyton Jones & Launchbury 1991), that is integers that are already evaluated and can be given a direct and efficient representation. Operations like timesInt# act on these efficient representation of integers, and can be directly implemented using C arithmetic operations.

Although there is scope for improvement (for example, leInt# x# top# is performed *twice*), this code is much more efficient than the original listful Haskell definition might have suggested.

Deforestation allowed further (conventional) optimisations to work, because the medium of the intermediate list was removed. For example, after deforestation the place where the index–values pairs were built could be matched with the place where array deconstructs such pairs. Because of deforestation an instance of the case reduction transformation is exposed:

```
let v = ( i , i * i )           let a = i
in case v of              ==>       b = i * i
    ( a , b ) -> ...            in  ...
```

Furthermore, exactly the same technology can be used to optimise other Haskell array operators, like \\ and `accum`.

If we compile the above Core program to C, then we get the following output:

```
loop:
   HEAP_CHK(1);
   if (x <= top) {
        y = x * x;
        z = x + 1;
        if (1 <= x) {
            if (x <= top) {
                i = x + 1;
                ALLOC_CON(Int);
                *Hp=y;
                arr[i] = Hp;
                x = z;
                goto loop;
           } else {
                :
                :
```

It is within the ability of current imperative optimising compilers to compile this efficiently (Aho et al. 1986). Indeed, GHC could be straightforwardly augmented to produce `while` loops. Such a compiler, for our array example, could output C looking something like:

```
if (1 <= x) {
    while (x <= top) {
        HEAP_CHK(1);
        ALLOC_CON(Int);
        *Hp=(x * x);
        arr[x + 1] = Hp;
        x++;
   }
} else ...
```

Further optimisations could be performed to lift the heap check out of the loop, allocating the space for all the `Int`'s in one go.

5.3 Measuring cheap deforestation

Our implementation of cheap deforestation is actually a suite of optimisations:

- We perform optimising transformations, like the `foldr/build` rule.
- We aggressively inline many prelude functions.
- We perform other analysis and transformations, like arity analysis.

To allow us to identify the causes of any performance gains, we have constructed several distinct versions of GHC.

- As our control we have taken a slightly modified GHC version 0.26, with cheap deforestation totally disabled. This control version also has a few minor improvements and bug fixes which will be incorporated into version 0.27. In our control we also revert to the original prelude, not giving definitions in terms of `foldr` and `build`.

  We call this version of GHC "$\mathcal{C}$".

- In order to factor out inlining, we have a compiler with inlining turned on, that uses a version of prelude that use `foldr` and `build`, but does not perform any cheap deforestation or extra transformations.

  We call this version of GHC "$\mathcal{C}_i$" (C-inline).

- We then have a version of the compiler that has our cheap deforestation transformations, but without our *extra* enhancing transformations, namely arity analysis and enhanced occurrence analysis.

  We call this version of GHC "$\mathcal{C}_{ic}$" (C-inline-and-cheap deforestation).

- We have a version of the compiler that has all of cheap deforestation turned on.

  We call this version of GHC "$\mathcal{C}_{ice}$" (C-inline-and-cheap deforestation-and-extras).

- Finally, we also have a version that has *only* the extra enhancing transformations.

  We call this version of GHC "$\mathcal{C}_e$" (C-extras).

The relationship between the different versions of GHC is illustrated in Figure 5.1. Note this shows the sum of transformations used, and does not imply the order which enhancements are actually applied to GHC.

$$\begin{tikzcd}
 & \mathcal{C}_e & & \\
\mathcal{C} \arrow[ur, "\text{Extra}", "\text{Transformations}"'] \arrow[dr, "\text{Agressive}", "\text{Inlining}"'] & & & \mathcal{C}_{ice} \\
 & \mathcal{C}_i \arrow[r, "\text{and Cheap}", "\text{Deforestation}"'] & \mathcal{C}_{ic} \arrow[ur, "\text{and Extra}", "\text{Transformations}"'] &
\end{tikzcd}$$

Figure 5.1: Different versions of the Glasgow Haskell compiler

5.3.1 What we want to measure

Having decided that we need five different versions of the compiler, we also need to decide what to measure. There are many interesting aspects of these different versions of the compiler to be examined.

- **Execution Speed.** The whole point of cheap deforestation is to improve performance! We measure instruction count rather than wall clock or Unix user time to get a more robust result. To do this we use the Sun *Spix-Tools* (Sun Microsystems 1993).
- **Heap Size, Heap Residency and Stack Usage.** Because deforestation removes intermediate data structures, we expect it to reduce heap consumption. We want to observe the demands that our transformed programs make on the storage management system.
- We are interested, where applicable, how many times the different cheap deforestation transformations were used.
- We want to measure how long our different compilers take to compile our benchmarks.
- We are also interested in the size of the object files that the compiler produces.

5.3.2 The nofib benchmark suite

To allow a more realistic idea of what effect cheap deforestation has on real programs, we use programs from the nofib suite (Partain 1992), as distributed with GHC, version 0.26. The nofib suite is divided into 3 subsets:

- The real subset – Programs that are written to get a job done.
- The spectral subset – other, smaller programs. The benchmark suite used by Pieter Hartel (Hartel & Langendoen 1993, Hartel 1994) is included as part of this suite.
- The toy subset – trivial benchmarks, like fib and queens. Following the advice given by Partain (1992), we ignore this set.

For pragmatic reasons, we omit some benchmarks. Firstly, we omitted any benchmark that ran in less that 100K bytes of heap. Secondly, a few of the benchmarks included in the 0.26 release did not compile in "reasonable" time and/or space. These included the largest test, anna and also symalg, which refused to compile (with or without cheap deforestation) in a 40 megabyte heap. We believe missing out these benchmarks does not effect our results or conclusions. Tables 5.1 and 5.2 gives short descriptions of all the programs we used from the *real* and *spectral* components of the nofib suite.

5.3.3 Our control: GHC without cheap deforestation

Tables 5.3 and 5.4 gives details about all our benchmarks compiled with our control compile, $\mathcal{C}$. Table 5.3 gives some static details about each benchmark, as well as the total time (in seconds) of how long it took to compile all the individual modules of each benchmark, and the total size (in bytes) of the produced object files. Table 5.4 gives runtime details for each benchmark. We include:

- Total allocations (with number of garbage collections in brackets).
- The maximum heap (with number of samples in brackets). We performed a sample every 100,000 bytes, by forcing a major garbage collection.
- Total number of machine cycles required to execute the benchmark.
- Maximum stack size(s). We have two stacks, one holds pointers to closures (stack A), and the other holds basic values (stack B) (Peyton Jones 1992).

In order to obtain accurate figures, each benchmark was run three times, once to measure heap and stack residency, and once to measure machine cycles (using *SpixTools*), and once to measure heap usage.

Program	Description	Origin
`bspt`	BSP-tree modeller	Iain Checkland (York)
`compress`	Text compression	Paul Sanders (BT)
`compress2`	Text compression	Paul Sanders (BT)
`ebnf2ps`	Syntax diagram generator	Peter Thiemann
`fluid`	Fluid-dynamics program	Xiaoming Zhang (Swansea)
`fulsom`	Solid Modeling	Duncan Sinclair (Glasgow)
`gamteb`	Monte Carlo photon transport	Pat Fasel (Los Alamos)
`gg`	Graphs from GRIP statistics	Iain Checkland (York)
`hidden`	Hidden line removal	Mark Ramaer/Stef Joosten
`HMMS`	Speech analysis	David Goblirsch (MITRE Corporation)
`hpg`	Haskell program generator	Nick North (NPL)
`infer`	Hindley-Milner type inference	Phil Wadler (Glasgow)
`lift`	Fully-lazy lambda lifter	David Lester (Manchester) & Simon Peyton Jones (Glasgow)
`maillist`	Mailing-list generator	Paul Hudak (Yale)
`parser`	Partial Haskell parser	Julian Seward (Manchester)
`pic`	Particle in cell	Pat Fasel (Los Alamos)
`prolog`	"mini-Prolog" interpreter	Mark Jones (Nottingham)
`reptile`	Escher tiling program	Sandra Foubister (York)
`rsa`	RSA encryption	John Launchbury (Glasgow)
`veritas`	Theorem-prover	Gareth Howells (Kent)

Table 5.1: The real suite in `nofib`

Program	Description	Origin
`boyer`	Gabriel suite 'boyer' benchmark	Denis Howe (Imperial)
`boyer2`	Gabriel suite 'boyer' benchmark	Denis Howe (Imperial)
`calendar`	Unix `cal` command	Mark Jones (Nottingham)
`cichelli`	Perfect hashing function	Iain Checkland (York)
`clausify`	Propositions to clausal form	Colin Runciman (York)
`cse`	Common subexpression elimination	Mark Jones (Nottingham)
`fft2`	Fourier transformation	Rex Page (Amoco)
`knights`	Knight's tour	Jon Hill (QMW)
`mandel`	Mandelbrot sets	Jon Hill (QMW)
`mandel2`	Mandelbrot sets	David Hanley
`minimax`	tic-tac-toe (0s and Xs)	Iain Checkland (York)
`multiplier`	Binary-multiplier simulator	John O'Donnell (Glasgow)
`primetest`	Primality testing	David Lester (Manchester)
`rewrite`	Rewriting system	Mike Spivey (Oxford)
`simple`	Standard Id benchmark	Andy Shaw (MIT)
`sorting`	Sorting algorithms	Will Partain (Glasgow)
`treejoin`	Tree joining	Kevin Hammond (Glasgow)

Program	Description (all Hartel Benchmarks)
`comp_lab_zift`	Image processing application
`event`	Event driven simulation of a set-reset flipflop
`fft`	Two fast fourier transforms
`genfft`	Generation of synthetic FFT programs
`ida`	Solution of a particular configuration of the n-puzzle
`listcompr`	Compilation of list comprehensions
`listcopy`	Compilation of list comprehensions (with extra list copying function for output)
`nucleic2`	Pseudo knot
`parstof`	Lexing and parsing based on Wadler's parsing method
`sched`	Calculation of an optimum schedule of parallel jobs with a branch and bound algorithm
`solid`	Point membership classification algorithm from a solid modeling library for computational geometry
`transform`	Transformation of a number of programs represented as synchronous process networks into master/slave style parallel programs
`typecheck`	Polymorphic type checking of a set of function definitions
`wang`	Wang's algorithm for solving system of linear equations
`wave4main`	Calculation of the water heights in a square area of 8×8 grid points of the North Sea over a long time period

Table 5.2: The spectral suite in `nofib`

	Source		Object	
	Number of		Compile	Binary
Benchmark	Modules	Lines	Time	Size
HMMS	16	4,221	670.6	790,528
boyer	1	1,016	79.3	335,872
boyer2	5	723	110.1	376,832
bspt	17	2,151	878.5	663,552
calendar	1	129	58.4	278,528
cichelli	5	249	95.3	344,064
clausify	1	177	33.5	254,600
comp_lab_zift	1	884	112.5	279,912
compress	5	821	146.2	262,144
compress2	3	337	97.1	311,296
cse	2	459	57.6	344,064
ebnf2ps	16	2,675	968.6	638,976
event	1	451	39.1	246,224
fft	1	824	52.9	475,136
fft2	3	215	59.7	483,328
fluid	18	2,391	812.1	688,128
fulsom	12	1,397	641.0	671,744
gamteb	13	709	362.8	542,040
genfft	1	502	63.5	254,904
gg	9	810	562.1	696,320
hidden	15	509	458.6	598,016
hpg	8	2,059	381.1	630,784
ida	1	490	54.7	254,480
infer	13	585	361.0	360,448
knights	5	879	111.3	335,872
lift	5	2,132	239.9	311,296
listcompr	1	522	53.1	262,144
listcopy	1	527	53.7	262,144
maillist	1	177	37.9	327,680
mandel	3	497	54.7	475,136
mandel2	1	222	31.5	507,904
minimax	6	257	118.2	327,680
multiplier	1	490	85.5	279,952
nucleic2	6	3,389	621.8	819,200
parser	1	4,595	568.9	557,056
parstof	1	1,275	346.9	450,560
pic	9	526	284.1	507,904
primetest	4	276	76.8	344,064
prolog	7	637	219.0	344,064
reptile	13	1,553	532.4	450,560
rewrite	1	631	144.1	368,640
rsa	2	97	58.2	327,680
sched	1	555	47.7	246,632
simple	1	1,134	478.8	688,128
solid	1	2,488	158.6	574,632
sorting	2	160	49.3	254,576
transform	1	1,1[illegible]2	354.2	385,024
treejoin	1	725	33.0	319,488
typecheck	1	658	89.1	278,528
veritas	32	11,147	2161.7	983,040
wang	1	357	47.8	466,944
wave4main	1	1,196	68.5	466,944

Table 5.3: Control run of compiler – $\mathcal{C}$

	Total		Max Heap		Max Stack		Instruction
Benchmark	Allocations		Residency		A	B	Count
HMMS	356,655,820	(396)	1,882,772	(3,570)	8,052	14,010	2,779,259,509
boyer	21,635,660	(12)	101,912	(216)	93	337	103,534,897
boyer2	2,198,776	(1)	284,760	(22)	206	499	20,554,198
bspt	4,618,764	(2)	416,072	(46)	58	456	21,235,082
calendar	240,040	(0)	4,172	(2)	38	86	1,193,173
cichelli	30,730,952	(11)	1,395,024	(307)	362	1,830	330,482,780
clausify	19,979,216	(13)	48,384	(200)	28	64	118,776,175
comp_lab_zift	103,988,876	(15)	1,238,300	(1,043)	233	2,587	478,507,684
compress	144,170,528	(116)	167,060	(1,442)	1,547	1,294	792,701,502
compress2	70,692,404	(21)	10,381,044	(707)	723	100	195,101,045
cse	350,768	(0)	57,252	(3)	88	419	1,692,397
ebnf2ps	3,103,484	(1)	301,288	(31)	2,342	3,476	14,699,163
event	42,947,016	(3)	4,118,096	(445)	44,020	484,298	237,937,906
fft	11,249,504	(0)	1,769,608	(121)	112	349	49,903,850
fft2	47,647,004	(34)	944,516	(483)	138	420	185,192,456
fluid	3,691,372	(1)	71,708	(37)	2,099	3,191	19,204,087
fulsom	212,641,236	(66)	3,838,444	(2,148)	155	361	915,349,230
gamteb	85,565,904	(63)	540,424	(857)	3,003	15,010	494,159,401
genfft	17,452,128	(1)	5,956	(174)	16	96	75,352,238
gg	6,860,076	(3)	358,680	(69)	842	1,203	39,159,765
hidden	428,854,340	(250)	327,852	(4,291)	2,086	3,485	2,009,208,670
hpg	59,039,772	(46)	611,268	(591)	38	12,629	267,134,620
ida	51,701,896	(4)	437,100	(517)	211	7,910	228,109,941
infer	10,233,948	(8)	2,087,436	(102)	312	1,523	134,233,568
knights	790,980	(0)	23,016	(7)	49	347	15,911,527
lift	391,612	(0)	25,932	(3)	194	878	1,179,545
listcompr	71,280,392	(8)	7,506,988	(714)	19	91	344,368,543
listcopy	78,792,524	(10)	7,512,564	(790)	19	96	375,042,961
maillist	4,150,968	(1)	15,940	(41)	2,059	3,111	18,191,820
mandel	219,721,832	(169)	12,828	(2,199)	40	144	572,572,857
mandel2	10,037,152	(4)	600	(100)	27	49	38,658,307
minimax	1,966,848	(0)	3,244	(19)	44	148	9,401,134
multiplier	84,762,436	(89)	1,819,988	(929)	276	703	381,407,199
nucleic2	46,600,536	(22)	60,540	(466)	391	641	226,860,802
parser	12,202,072	(8)	947,256	(122)	738	483	80,783,219
parstof	47,348,996	(4)	563,012	(473)	768	397	312,770,038
pic	5,203,152	(2)	327,876	(52)	498	1,151	31,944,711
primetest	120,710,408	(89)	200,424	(1,209)	1,234	4,856	5,430,259,340
prolog	642,184	(0)	19,840	(6)	999	1,515	3,336,869
reptile	4,746,512	(1)	619,732	(47)	46	96	23,124,635
rewrite	22,935,328	(10)	19,528	(229)	92	168	115,103,552
rsa	32,137,272	(15)	12,188	(321)	1,545	2,340	1,101,964,625
sched	19,663,212	(1)	2,552	(196)	34	107	64,235,884
simple	118,048,208	(36)	9,030,920	(1,187)	1,976	5,782	1,086,483,698
solid	66,794,720	(6)	522,284	(669)	925	2,250	300,315,438
sorting	413,316	(0)	71,648	(4)	200	481	2,550,308
transform	194,699,284	(16)	146,632	(1,952)	686	2,758	1,053,307,565
treejoin	67,038,476	(6)	8,108,940	(670)	69,142	115,239	349,537,058
typecheck	117,666,440	(10)	9,932	(1,180)	54	130	738,414,743
veritas	362,188	(0)	13,884	(3)	107	352	1,580,923
wang	27,232,932	(2)	5,074,216	(272)	25	6,543	120,072,103
wave4main	128,944,808	(17)	1,952,356	(1,290)	16,062	40,005	1,121,639,955

Table 5.4: Control run of compiler – $\mathcal{C}$ (continued)

5.4 Results from using cheap deforestation

In this section we give performance results of comparisons between our different versions of GHC. We expect the following results.

- Some performance improvements will be due to aggressive inlining. We can observe this by comparing C with C_i, which we do in § 5.4.1.
- The extra transformations are beneficial, irrespective of the presence of cheap deforestation. We can observe this by comparing C with C_e, which we do in § 5.4.2.
- We expect to see our deforestation without the extra enabling transformations having some (limited) effect. We can observe this by comparing C_i with C_{ic} in § 5.4.3.
- Finally, and most importantly, we expect the combination of cheap deforestation and extra enabling transformations to improve our benchmark programs. We can observe this by comparing C_i with C_{ice}, which we do in § 5.4.4.

We also expect to see evidence that our enabling technology increases the number of instances of our deforestation rules.

We give our results in the form:

Benchmark	Option A
wave4main	0.78
calendar	0.91
⋮	⋮
n other programs	1.00
Minimum	0.78
Maximum	1.02
Geometric Mean	0.98

The figure "0.78" means that "Option A" uses 78% of the specified resources, when compared with the base compiler. This means that, for example, if we were to compare compiling with C_i (this table's base compiler) against compiling with C_{ice} (this table's "enhanced" compiler), a program that takes 100,000 instructions to execute when compiled with C_i compiler would be taking 78,000 instructions to execute when compiled with C_{ice}.

Programs that showed a difference of less than 0.5% are omitted, and included under the "n other programs" line. The results from these omitted programs, however, are included in the calculation of the mean. We use the geometric mean because we are averaging normalised numbers (Fleming & Wallace 1986).

5.4.1 $\mathcal{C}$ vs $\mathcal{C}_i$: Gains from inlining

The first thing we want to observe is how performing aggressive inlinings effect code side and compilation time. Table 5.5 gives both these factors. As we anticipated, both of these aspects increased. The code size increased by, on average, 8%, and the compile time increased by, on average, 15%. These figures are the down-payment that we have to pay to achieve cheap deforestation.

We also want to observe how much performance gain we get because of performing aggressive inlinings. Table 5.6 gives this. The average improvement was 2%.

Benchmark	Compile Time
nucleic2	1.02
mandel2	1.04
sched	1.05
treejoin	1.05
fulsom	1.06
typecheck	1.06
boyer2	1.07
bspt	1.08
event	1.08
compress	1.09
compress2	1.09
veritas	1.09
ebnf2ps	1.10
listcompr	1.10
listcopy	1.10
mandel	1.10
primetest	1.10
cichelli	1.11
infer	1.11
sorting	1.11
boyer	1.12
clausify	1.12
comp_lab_zift	1.13
gg	1.13
lift	1.13
minimax	1.13
parser	1.13
parstof	1.13
hidden	1.14
gamteb	1.15
prolog	1.15
wave4main	1.15
genfft	1.16
hpg	1.16
ida	1.16
HMMS	1.18
knights	1.18
maillist	1.18
reptile	1.18
fluid	1.19
rsa	1.19
cse	1.20
rewrite	1.21
wang	1.21
fft	1.26
transform	1.27
solid	1.29
pic	1.30
calendar	1.31
multiplier	1.37
fft2	1.45
simple	1.53
Minimum	1.02
Maximum	1.53
Geometric Mean	1.15

Benchmark	Binary Size
comp_lab_zift	1.01
fulsom	1.01
listcompr	1.01
listcopy	1.01
transform	1.01
typecheck	1.01
boyer	1.02
bspt	1.02
genfft	1.02
parser	1.02
ebnf2ps	1.03
gg	1.03
primetest	1.03
veritas	1.03
event	1.04
maillist	1.04
clausify	1.05
sorting	1.05
compress2	1.06
compress	1.07
hpg	1.08
infer	1.08
mandel	1.08
reptile	1.08
HMMS	1.09
ida	1.09
lift	1.10
minimax	1.10
cichelli	1.11
cse	1.12
knights	1.12
prolog	1.12
hidden	1.13
gamteb	1.14
rewrite	1.14
wave4main	1.14
fluid	1.15
wang	1.15
rsa	1.17
calendar	1.20
simple	1.22
fft	1.26
pic	1.33
multiplier	1.35
fft2	1.49
7 other programs	1.00
Minimum	1.00
Maximum	1.49
Geometric Mean	1.08

Table 5.5: The effect inlining has on compile time and code size

Benchmark	Instruction Count
wave4main	0.78
calendar	0.91
fft	0.91
fft2	0.91
simple	0.93
fluid	0.95
hidden	0.95
multiplier	0.95
HMMS	0.96
maillist	0.96
boyer	0.97
minimax	0.97
ebnf2ps	0.98
pic	0.98
rewrite	0.98
compress2	0.99
hpg	0.99
lift	0.99
mandel	0.99
comp_lab_zift	1.01
reptile	1.01
sched	1.01
transform	1.01
veritas	1.01
genfft	1.02
listcompr	1.02
listcopy	1.02
25 other programs	1.00
Minimum	0.78
Maximum	1.02
Geometric Mean	0.98

Table 5.6: The effect inlining has on execution count

5.4.2 $\mathcal{C}$ vs $\mathcal{C}_e$: Gains from enabling technologies

Table 5.7 summarises the instruction execution count improvement. Only 6 programs were noticeably improved of which 5 were only slight improvements.

Table 5.8 summarises the nominal change caused by our enabling technologies (by comparing $\mathcal{C}$ with $\mathcal{C}_e$). Both the compilation time and binary size stayed much the same, and both averaged no change. One reason an optimisation can actually *reduce* the compilation time is because if an optimisation manages to reduce code size later aspects of the compilation have less work to do!

Benchmark	Instruction Count
maillist	0.81
parser	0.98
HMMS	0.99
cse	0.99
ebnf2ps	0.99
pic	0.99
46 other programs	1.00
Minimum	0.81
Maximum	1.00
Geometric Mean	0.99

Table 5.7: The effect our enabling technology has on execution count

Benchmark	Compile Time
mandel2	0.97
cse	0.98
minimax	0.98
treejoin	0.98
typecheck	0.98
calendar	0.99
comp_lab_zift	0.99
ebnf2ps	0.99
hidden	0.99
ida	0.99
listcopy	0.99
multiplier	0.99
nucleic2	0.99
parstof	0.99
primetest	0.99
boyer	1.01
boyer2	1.01
bspt	1.01
cichelli	1.01
compress2	1.01
event	1.01
fft	1.01
fluid	1.01
gamteb	1.01
infer	1.01
mandel	1.01
reptile	1.01
solid	1.01
transform	1.01
wang	1.01
wave4main	1.01
clausify	1.02
fft2	1.02
maillist	1.02
rsa	1.02
simple	1.03
veritas	1.11
15 other programs	1.00
Minimum	0.97
Maximum	1.11
Geometric Mean	1.00

Benchmark	Binary Size
compress	0.99
ebnf2ps	0.99
event	0.99
fulsom	0.99
hidden	0.99
infer	0.99
46 other programs	1.00
Minimum	0.99
Maximum	1.00
Geometric Mean	1.00

Table 5.8: The effect our enabling technology has on compile time and code size

5.4.3 C_i vs C_{ic} : Gains from raw deforestation opportunities

In this section we examine cheap deforestation without our enabling technology. We call this "raw" deforestation. We compare C_i with C_{ic}, so as to factor out the less interesting effect of inlining. Several aspects of the compilation and running of our benchmarks have been measured.

- Table 5.9 gives the compile time and code size of our benchmarks. Using raw deforestation cost, on average, an extra 5% in compilation time, but the binary size was, again on average, slightly reduced.
- Table 5.10 gives the improvement in execution time. Several programs are showing a slight improvement, and the average is only 1%.

  Some examples are getting worse. We have already discussed possible reasons for this (§ 3.1.2).
- Table 5.11 gives the improvements in total heap allocations. Somewhat surprisingly there was an average of no improvement, with one benchmark using 15% more heap.

  The results can be explained by the lack of arity analysis, one of our enabling technologies, in this test. Arity analysis improves the passing of accumulating arguments, and often "cleans up" after cheap deforestation, giving efficient code.
- Tables 5.12 and 5.13 give the heap residency and maximum stack sizes.

  There does not appear to be a relationship between the heap residency improvement and the heap usage improvement.
- Finally, Table 5.14 gives a list of how often each of the principal transformations in cheap deforestation are performed. The labels are as follows
  - fr/b – `foldr/build` rule
  - fr/a – `foldr/augment` rule
  - fr/[] – `foldr` of empty list
  - fr/(x:xs) – `foldr` of a sequence of `(:)`
  - fl/b – `foldl/build` rule
  - fl/a – `foldl/augment` rule
  - fl/[] – `foldl` of empty list
  - fl/(x:xs) – `foldl` of a sequence of `(:)`

Many of the benchmarks did trigger one or more of the deforestation rules. The table indicates, however, many instances of the fr/(x:xs) and fr/[] were occurring, but very few of the true deforestation rules (fr/b,fr/a,fl/a,fl/b) were actually triggered. We argue that this is because this test was performed without the enabling technologies, which are designed specifically to enhance deforestation opportunities.

Benchmark	Compile Time
mandel2	0.97
solid	0.98
boyer2	1.01
hidden	1.01
infer	1.01
primetest	1.01
reptile	1.01
rsa	1.01
sorting	1.01
fft2	1.02
hpg	1.02
minimax	1.02
parstof	1.02
transform	1.02
typecheck	1.02
clausify	1.03
ebnf2ps	1.03
gg	1.03
ida	1.03
listcompr	1.03
listcopy	1.03
maillist	1.03
mandel	1.03
pic	1.03
prolog	1.03
rewrite	1.03
simple	1.03
fulsom	1.04
sched	1.04
treejoin	1.04
wang	1.04
HMMS	1.05
cichelli	1.05
compress2	1.05
fft	1.05
knights	1.05
nucleic2	1.05
comp_lab_zift	1.06
event	1.06
fluid	1.06
boyer	1.07
gamteb	1.07
genfft	1.08
lift	1.08
multiplier	1.08
bspt	1.09
calendar	1.09
cse	1.09
parser	1.10
wave4main	1.11
veritas	1.17
compress	1.21
Minimum	0.97
Maximum	1.21
Geometric Mean	1.05

Benchmark	Binary Size
solid	0.61
parstof	0.77
transform	0.87
simple	0.91
reptile	0.92
hpg	0.93
listcompr	0.93
listcopy	0.93
parser	0.93
HMMS	0.95
genfft	0.95
hidden	0.95
infer	0.95
minimax	0.95
pic	0.95
cichelli	0.96
compress2	0.96
ebnf2ps	0.96
fft	0.96
prolog	0.96
rsa	0.96
treejoin	0.96
cse	0.97
gg	0.97
ida	0.97
maillist	0.97
comp_lab_zift	0.98
knights	0.98
mandel	0.98
typecheck	0.98
veritas	0.98
boyer	0.99
boyer2	0.99
clausify	0.99
fft2	0.99
fulsom	0.99
multiplier	0.99
rewrite	0.99
sorting	0.99
wang	0.99
bspt	1.01
wave4main	1.01
lift	1.07
gamteb	1.16
compress	1.17
7 other programs	1.00
Minimum	0.61
Maximum	1.17
Geometric Mean	0.96

Table 5.9: The effect raw deforestation has on compile time and code size

Benchmark	Instruction Count
multiplier	0.94
infer	0.95
lift	0.95
bspt	0.97
listcompr	0.97
reptile	0.97
listcopy	0.98
cse	0.99
ebnf2ps	0.99
fluid	0.99
hidden	0.99
knights	0.99
maillist	0.99
parser	0.99
pic	0.99
prolog	0.99
rewrite	0.99
transform	0.99
typecheck	0.99
wave4main	0.99
minimax	1.03
fft2	1.05
30 other programs	1.00
Minimum	0.94
Maximum	1.05
Geometric Mean	0.99

Table 5.10: The effect raw deforestation has on execution count

Benchmark	Heap Allocations	
multiplier	0.93	(86)
lift	0.94	(0)
wave4main	0.94	(12)
bspt	0.97	(2)
pic	0.97	(2)
maillist	0.98	(1)
reptile	0.98	(1)
rewrite	0.98	(10)
ebnf2ps	0.99	(1)
hidden	0.99	(228)
infer	0.99	(8)
prolog	0.99	(0)
typecheck	0.99	(10)
fulsom	1.01	(66)
hpg	1.01	(47)
parser	1.01	(8)
HMMS	1.02	(316)
fluid	1.02	(1)
knights	1.04	(0)
minimax	1.11	(1)
simple	1.12	(37)
fft2	1.15	(41)
30 other programs	1.00	
Minimum	0.93	
Maximum	1.15	
Geometric Mean	1.00	

Table 5.11: The effect raw deforestation has on heap allocation

Benchmark	Heap Residency
solid	0.56 (667)
transform	0.75 (1964)
parstof	0.78 (472)
fluid	0.89 (35)
gamteb	0.89 (858)
minimax	0.90 (22)
maillist	0.91 (40)
mandel	0.92 (2192)
prolog	0.92 (6)
genfft	0.96 (174)
bspt	0.98 (45)
pic	0.98 (50)
fulsom	0.99 (2161)
gg	0.99 (69)
fft	1.01 (120)
infer	1.01 (100)
typecheck	1.02 (1157)
rewrite	1.03 (218)
simple	1.05 (1218)
knights	1.09 (8)
fft2	1.37 (518)
26 other programs	1.00
Minimum	0.56
Maximum	1.37
Geometric Mean	0.97

Table 5.12: The effect raw deforestation has on heap residency

Benchmark	Max A Stack
reptile	0.96
minimax	0.98
cse	0.99
lift	0.99
knights	1.02
HMMS	1.06
fft2	4.95
simple	6.07
44 other programs	1.00
Minimum	0.96
Maximum	6.07
Geometric Mean	1.07

Benchmark	Max B Stack
solid	0.94
minimax	0.97
rewrite	0.98
HMMS	0.99
multiplier	0.99
bspt	1.07
veritas	1.33
fft2	6.63
simple	9.65
43 other programs	1.00
Minimum	0.94
Maximum	9.65
Geometric Mean	1.09

Table 5.13: The effect raw deforestation has on maximum stack sizes

Benchmark	fr/b	fr/a	fr/[]	fr/(x:xs)	fl/b	fl/a	fl/[]	fl/(x:xs)
HMMS	1		22	22	3			
boyer	1							
boyer2								
bspt	2	1	34	35	1			
calendar			1	1	1			
cichelli	2							
clausify								
comp_lab_zift			10	12				
compress			4	4				
compress2			1	2				
cse			6	7				
ebnf2ps	4		11	17	1			
event								
fft			3	3				
fft2			2	2	2			
fluid	2		2	6	14			
fulsom			4	4				
gamteb			10	10				
genfft			3	3				
gg			13	13				
hidden	10				1			
hpg	1		5	7	1			
ida			4	5				
infer	4	1	5	5				
knights			5	5				
lift	3		20	20	1			
listcompr			6	7				
listcopy			6	7				
maillist								
mandel				2				
mandel2								
minimax	3		3	5	1			
multiplier	1		2	3	1			
nucleic2								
parser	2	4	5	5				
parstof			43	44				2
pic			14	14				
primetest								
prolog			4	5				
reptile			8	9				
rewrite	3		3	3				
rsa	2		1	1				
sched								
simple	1		74	74	13			
solid								
sorting								
transform			19	20				
treejoin								
typecheck	2		5	5				
veritas	12		27	30	3			4
wang			2	3				
wave4main			2	2				

Table 5.14: How often raw deforestation transformations are used

5.4.4 C_i vs C_{ice} : Gains from complete cheap deforestation

We have already argued that we need enabling technologies to allow the successful deforestation of many key examples, including `map` of `map`. We have also seen that very few instances of our key rules occur in practice without any enabling technology. We now "turn on" our enabling technology, allowing it to enhance cheap deforestation. Again, to factor out the less interesting effects of inlinings, we compare C_i with C_{ice}.

We measure exactly the same aspects as in the previous test.

- Table 5.15 gives the compile time and code size of our benchmarks. Again we see a small increase in compile time, but we see a significant drop in average binary size, of 6%. This almost offsets the average binary size increase caused by inlining (8%).

- Table 5.16 gives the improvement in execution time. Around half of our benchmarks showed an improvement. Around a dozen benchmarks improved by more that 5%. The average was 3%. Only one example got noticeably worse, and only by 1%.

- Table 5.17 gives the improvements in total heap allocations. As expected, we reduce the heap allocations, by up to almost half in one case.

- Tables 5.18 and 5.19 give the heap residency and maximum stack sizes.

  It is interesting that for one of our best benchmarks (`simple`) we improve the execute count by 10%, improve the heap usage by 24%, but also increase the maximum stack size by a factor of 50. This is undoubtedly due to turning heap allocations into stack allocations.

  Again there does not appear to be a relationship between the heap residency improvement and the heap usage improvement.

- Finally, Table 5.20 gives a list of how often each of the principal transformations in cheap deforestation are performed.

  Comparing this with Table 5.14, we observe that our enabling technology has substantially enhanced the occurrences of our deforestation transformations. This gives justification to our argument for using our enabling technology.

Sometimes we saw improvements in execution time caused by cheap deforestation, but did not see any deforestation reductions taking place in Table 5.14.

For example, consider `maillist`, which shows a 21% improvement, but did not perform any deforestation reductions. We attribute this to the fact that for each version of the compiler a new version of the Haskell prelude was also compiled, and the number of deforestation rules invoked during the compilation of the prelude itself are not included in Tables 5.14 and 5.20, because this is constant across all the benchmarks. Furthermore, the prelude included several opportunities for deforestation, so we suspect that the discrepancy is caused by `maillist` calling a deforested function in the prelude.

Benchmark	Compile Time
fft2	0.97
simple	0.98
hidden	1.01
infer	1.01
minimax	1.01
pic	1.01
primetest	1.01
reptile	1.01
clausify	1.02
event	1.02
hpg	1.02
maillist	1.02
mandel2	1.02
parstof	1.02
prolog	1.02
solid	1.02
sorting	1.02
veritas	1.02
boyer2	1.03
ebnf2ps	1.03
mandel	1.03
rewrite	1.03
sched	1.03
wang	1.03
compress2	1.04
fluid	1.04
fulsom	1.04
listcompr	1.04
listcopy	1.04
nucleic2	1.04
gg	1.05
transform	1.05
typecheck	1.05
HMMS	1.06
fft	1.06
ida	1.06
knights	1.06
treejoin	1.06
boyer	1.07
cichelli	1.07
gamteb	1.07
lift	1.07
multiplier	1.07
calendar	1.08
comp_lab_zift	1.08
parser	1.08
wave4main	1.08
bspt	1.11
cse	1.12
genfft	1.17
compress	1.20
1 other program	1.00
Minimum	0.97
Maximum	1.20
Geometric Mean	1.04

Benchmark	Binary Size
solid	0.61
pic	0.76
parstof	0.77
fft2	0.85
simple	0.85
transform	0.86
HMMS	0.89
reptile	0.89
fft	0.90
parser	0.90
hpg	0.91
listcompr	0.91
listcopy	0.91
minimax	0.91
genfft	0.92
hidden	0.92
rsa	0.92
wang	0.92
fluid	0.93
infer	0.93
clausify	0.94
ebnf2ps	0.94
event	0.94
prolog	0.94
cichelli	0.95
compress2	0.95
cse	0.95
gg	0.95
treejoin	0.95
wave4main	0.95
ida	0.96
knights	0.97
maillist	0.97
multiplier	0.97
rewrite	0.97
veritas	0.97
boyer2	0.98
calendar	0.98
comp_lab_zift	0.98
fulsom	0.98
mandel	0.98
sorting	0.98
typecheck	0.98
boyer	0.99
lift	1.07
gamteb	1.13
compress	1.15
5 other programs	1.00
Minimum	0.61
Maximum	1.15
Geometric Mean	0.94

[illegible]

Benchmark	Instruction Count
maillist	0.79
fft2	0.81
wave4main	0.82
minimax	0.89
simple	0.90
cse	0.93
fluid	0.94
multiplier	0.94
infer	0.95
lift	0.95
pic	0.95
HMMS	0.96
ebnf2ps	0.96
fft	0.96
hidden	0.96
reptile	0.96
calendar	0.97
listcompr	0.97
listcopy	0.97
bspt	0.98
gg	0.98
parser	0.98
rewrite	0.98
transform	0.98
hpg	0.99
knights	0.99
nucleic2	0.99
prolog	0.99
typecheck	0.99
event	1.01
22 other programs	1.00
Minimum	0.79
Maximum	1.01
Geometric Mean	0.97

Table 5.16: The effect cheap deforestation has on execution count

Benchmark	Heap Allocations	
maillist	0.52	(1)
wave4main	0.55	(6)
simple	0.76	(28)
fft2	0.83	(21)
minimax	0.83	(0)
HMMS	0.88	(298)
fluid	0.91	(1)
cse	0.92	(0)
parser	0.92	(8)
pic	0.92	(2)
ebnf2ps	0.93	(1)
multiplier	0.93	(86)
hidden	0.94	(214)
lift	0.94	(0)
calendar	0.95	(0)
fft	0.96	(0)
gg	0.97	(3)
reptile	0.97	(1)
rewrite	0.97	(10)
bspt	0.98	(2)
infer	0.98	(8)
nucleic2	0.98	(22)
hpg	0.99	(43)
prolog	0.99	(0)
typecheck	0.99	(10)
event	1.02	(3)
knights	1.04	(0)
25 other programs	1.00	
Minimum	0.52	
Maximum	1.04	
Geometric Mean	0.95	

Table 5.17: The effect cheap deforestation has on heap allocation

Benchmark	Heap Residency	
solid	0.56	(667)
transform	0.73	(1959)
parstof	0.78	(472)
minimax	0.82	(16)
ebnf2ps	0.85	(26)
fluid	0.89	(31)
fulsom	0.89	(2156)
gamteb	0.89	(854)
prolog	0.89	(6)
mandel	0.92	(2192)
genfft	0.95	(174)
fft2	0.97	(376)
nucleic2	0.97	(457)
bspt	0.98	(45)
gg	0.99	(67)
maillist	0.99	(21)
parser	0.99	(112)
pic	0.99	(47)
infer	1.01	(100)
rsa	1.02	(321)
typecheck	1.02	(1157)
simple	1.05	(826)
knights	1.09	(8)
24 other programs	1.00	
Minimum	0.56	
Maximum	1.09	
Geometric Mean	0.96	

Table 5.18: The effect cheap deforestation has on heap residency

Benchmark	Max A Stack
reptile	0.96
cse	0.99
lift	0.99
ebnf2ps	1.02
knights	1.02
minimax	1.05
HMMS	1.12
pic	3.27
hidden	23.63
simple	49.53
42 other programs	1.00
Minimum	0.96
Maximum	49.53
Geometric Mean	1.18

Benchmark	Max B Stack
HMMS	0.87
rewrite	0.88
minimax	0.91
clausify	0.92
ebnf2ps	0.94
fft2	0.94
solid	0.94
multiplier	0.98
fulsom	0.99
infer	0.99
bspt	1.07
veritas	1.33
pic	1.98
fft	6.38
simple	9.12
hidden	10.08
36 other programs	1.00
Minimum	0.87
Maximum	10.08
Geometric Mean	1.14

Table 5.19: The effect cheap deforestation has on maximum stack sizes

Benchmark	fr/b	fr/a	fr/[]	fr/(x:xs)	fl/b	fl/a	fl/[]	fl/(x:xs)
HMMS	64	44	22	22	4			
boyer	1							
boyer2	3	1						
bspt	10	4	34	35	1			
calendar	1		1	1	1			
cichelli	4							
clausify	4							
comp_lab_zift			10	12				
compress	1		4	4				
compress2	2	1	1	2				
cse	3	4	6	7				
ebnf2ps	16	3	11	17	1			
event	3							
fft	3		3	3	1			
fft2	9		2	2	2			
fluid	29		2	7	14			
fulsom	15	10	4	4				
gamteb	1		10	10	4			
genfft	1	3	3	3				
gg	11	10	15	15				
hidden	17				3			
hpg	6		5	7	1			
ida	1		4	5				
infer	8	1	5	5				
knights	3		5	5				
lift	5		20	20	1			
listcompr	1		6	7				
listcopy	1		6	7				
maillist								
mandel				2				
mandel2								
minimax	8	1	3	5	1			
multiplier	5		2	3	1			
nucleic2		1						
parser	65	59	5	5				
parstof	1		43	44				2
pic	39	8	14	14	3	2		
primetest								
prolog	5		4	5				
reptile	18		8	9				
rewrite	12	1	3	3				
rsa	5		1	1				
sched								
simple	105	16	77	77	16	13		
solid	1	1	1	1				
sorting	3							
transform	4	1	19	20				
treejoin	2	1						
typecheck	2		5	5				
veritas	35	1	27	30	3			4
wang	6		2	3				
wave4main	3	1	2	2	1			

Table 5.20: How often cheap deforestation transformations are used

5.5 Summary

We have seen several examples of cheap deforestation in action. Sometimes the effect of cheap deforestation can be substantial. Results over a large set of benchmarks give a more modest but still tangible benefit. The improvements, however, were spread over many examples, rather than just a few benchmarks showing very good improvements. Approximately half the examples in our benchmark suite showed *some* improvement because of our deforestation. There is quite a high initial down-payment of performing inlinings, but we recover most of the code expansion with enhanced simplification opportunities.

Is cheap deforestation worth it?

- For a Haskell user, yes. It provides the possibility of writing efficient listful code, allows Haskell compilers to handle array creation more efficiently, and sometimes catches instances of "accidental" deforestation, that is deforestation friendly code fragments that the user writes without even considering deforestation.

- For a Haskell compiler implementer, cheap deforestation might not be the most important optimisation to implement, but clearly there are measurable benefits.

  Furthermore, the results are more pessimistic than might be the case for programs written with the knowledge that a deforestation is being used to enhance compilation.

With no other deforestation technology being sufficiently mature to be able run non-trivial benchmarks, cheap deforestation currently has a prominent place as the pragmatic list removal technique for lazy functional languages.

Chapter 6

The State of the Art in Data Structure Removal

There is a long tradition of removing intermediate data structures, both automatically and by user-assisted derivation. This chapter gives an overview of several techniques used for removing data structures, and explains their relationship to cheap deforestation.

Early work by Burstall and Darlington gave a system for performing program transformations (Burstall & Darlington 1977). In their system they use (among other things) the concepts of *unfolding*, to expose optimisation opportunities, and *folding* back to give efficient programs. Their system is user driver. We discuss work that attempts to automate their system in § 6.1.

A new line of attack to data structure removal has recently emerged. Catamorphisms (which we define later) can be considered as *natural* data structure consumers, and using them provides a transformation system with a *handle* into how a data structure is used. This approach to data structure consumption has been used by two deforestation schemes.

- Gill et al. (1993) proposed the use of the list catamorphism and a list building combinator, allowing a straightforward list removal rule, the `foldr/build` rule. This system of deforestation provides the basic list removal capability that we use in this thesis.
- Independently Sheard & Fegaras (1993) also proposed the use of catamorphisms to structure programs. Along with a catamorphism fusion rule this also leads to data structure removal opportunities. We discuss this system in § 6.2.1.

Some work has been done on combining these two new approaches to data structure removal. We explain these in § 6.2.2 and and § 6.2.3.

Another approach to deforestation is to spot common fixed patterns of known combinators. and have a set of reduction rules that removes intermediate data structures. The most common example of this is the `map` of `map` transformation.

$$\forall f \,.\, \forall g \,.\, \texttt{map}\ f\ (\texttt{map}\ g\ \texttt{xs}) = \texttt{map}\ (f . g)\ \texttt{xs}$$

We discuss the pragmatics of using such a style of deforestation, and some attempts to use it, in § 6.3.

Parameterising the creation of data structures, a key concept behind our deforestation technique, has been considered by others as an aid to data structure removal. We discuss previous systems in § 6.4. There has also been work on optimising the representation used for lists, rather than removing them wholesale. We discuss this work in § 6.5.

There are two other list-removal schemes that do not fit into any of the above categories, but deserve a mention. Waters (1991) introduces a datatype called *series* into Pascal and Lisp, which are similar to lazy lists, except he guarantees that this data structure will be removed at compile time. This is done by imposing a number of stringent conditions that must be met for any program using the series datatype to compile.

The other scheme is Listlessness. This is an automatic transformation that removed intermediate lists from a restrictive language (Wadler 1983). Wadler's listless transformer took programs written in the restrictive language, and transformed them into a graph-based language. Programs expressed in this graph-based language were bounded in their space usage, so all required memory could be preallocated. For example, the function to sort a list into two lists containing the odd and even elements of the original list can be done in constant space, by reusing the space statically allocated for the original list.

6.1 Supercompilation and deforestation

Building on the concepts presented in Burstall & Darlington (1977), Turchin developed his supercompiler (Turchin 1986). Turchin *drives* a program, performing symbolic evaluation and constructing a graph of configuration states, and then using this graph to rewrite his program more efficiently. He equates his driving with unfolding, and his directed rewriting with folding. Like the underlying fold/unfold model, the configuration graphs produced by supercompilation can be potentially infinite, and Turchin has an elaborate system for ensuring termination by placing restrictions on function calls (Turchin 1988).

Wadler's deforestation (Wadler 1990) also uses the fold/unfold model. He attacks the termination problem by limiting the form of the input program to compositions of functions in so-called *treeless form*. This is a particularly restrictive form of function definition. These restrictions did however allow two theorems to be proved: first that the result was also in treeless form (guaranteeing that *no* intermediate structures were built), and secondly that the algorithm does always terminate (Ferguson & Wadler 1988).

Many of the restrictions of Wadler's deforestation have been lifted. Specifically:

- Chin (1990) extends deforestation for first-order programs. He uses a *producer* and *consumer* model, where a good producer and good consumer over a single data structure will be guaranteed to *fuse*. Chin also did some limited work on the deforestation of higher-order programs.
- Hamilton (1993) extends the treeless form permitted for successful deforestation.
- Marlow (1996) extends deforestation to higher order programs.

Both Hamilton and Marlow include a proof of termination for their extend systems. Though the specifics of *how* these restrictions have been lifted are not (for our purposes) interesting, the question of the relationship between cheap deforestation and traditional deforestation is both interesting and not obvious. Even if we consider only the list datatype, the relationship is still not clear (Marlow & Gill 1995). We can, however, discuss the principal differences.

It is straightforward to construct examples where traditional deforestation removes lists and cheap deforestation does not. The classical example involves `zip`:

```
zip (map f xs) (map g ys)
```

Traditional deforestation removes both the list produced by `map f xs` *and* the list produced by `map g ys`. Cheap deforestation has sufficient power for removing the first list (though there are caveats, c.f. § 3.2.5), but has no way to remove both.

Another example of the difference, which highlights the way traditional deforestation successfully handles "irregular" consumptions, is the removing of the list inside the function `fourth`:

```
second (x:_:xs) = x : second xs
second [x]      = [x]
second []       = []
fourth          = second . second
```

Traditional deforestation removes the list between the two seconds, giving an efficient cascade of cases:

```
fourth x1 =
   case x1 of
     x:x2 ->
        case x2 of
          _:x3 ->
             case x3 of
               _:x4 ->
                 case x4 of
                   _:x5 -> x : fourth x5
                   [] -> [x]
               [] -> [x]
          [] -> [x]
     [] -> []
```

It would be *possible* to write second in terms of foldr and build, but this would involve a bidirectional flow of information, carrying a token (in this case, a Bool):

```
second xs = build (\ c n ->
             let
              fn x g True  = x `c` g False
              fn x g False = g True
             in
               foldr fn (\ z -> foldr c n z) xs True)
```

After cheap deforestation, fourth would have two Bool counters.

```
fourth xs =
  let
    h []     a b = []
    h (x:xs) a b =
        if a
        then if b
             then x : h xs False False
             else h xs True False
        else h xs True b
  in
    h xs True True
```

This is clearly inferior to the cascade of cases, and highlights the issue that just because it is *possible* to remove lists using cheap deforestation does not mean

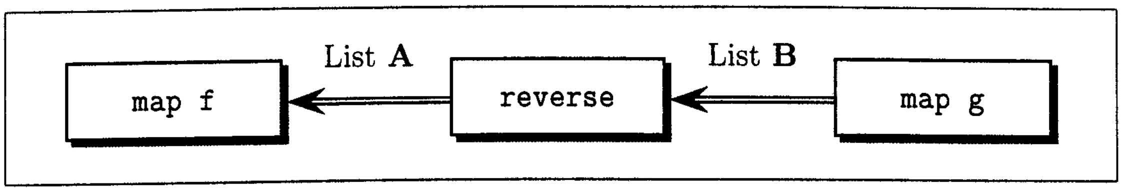

Figure 6.1: `map f . reverse . map g`

that we want to remove it. It is *not* clear that performing such coercions of "irregular" consumers into `foldr` form actually wins in the general case.

As a counter-case, consider the expression `map f . reverse . map g`, as illustrated in Figure 6.1. With cheap deforestation, list **A** *and* list **B** get removed, though list **B** gets replaced with a sequence of suspensions inside an accumulating argument. In some cases the strictness analyser can be used to detect that the accumulating argument is strict, and the space used by list **B** can be totally removed. The final code, after cheap deforestation (including arity analysis) is:

```
let
   h []     z = z
   h (x:xs) z = h xs (f (g x) : z)
in
   h xs []
```

The constructors that formed list **B** have been transformed into suspensions of the form `(f (g x) : z)`. With traditional deforestation, the story is different. Traditional deforestation can successfully merge list **B**, without even needing the arity analysis technology. However, list **A** is "residual" after deforestation (Marlow 1996). Table 6.1 summarises the differences between the way the two deforestation systems act on this example.

So can we generalise our understanding of this relationship between the two deforestation systems, and classify *when* one works and one fails? A formal

Deforestation Style	Remove list **A**?	Remove List **B**?
Traditional Deforestation	No	Yes
Cheap Deforestation	Yes	Yes, but • needs arity analysis • creates suspensions

Table 6.1: How both deforestation schemes act on `map f.reverse.map g`

comparison might be possible (c.f. § 7.3.5), but we can make some simple observations.

- Traditional deforestation can cope with "regular" consumptions as well as cheap deforestation, and can also cope with many "irregular" consumptions.
- There are cases where `build` can give a handle onto the list constructors that construct an intermediate list, but traditional deforestation is unable to locate them. These cases include when the constructors are part of an accumulating argument.

6.2 Catamorphisms for data structure consumption

Catamorphisms are used to consume (or "down-form") data structures. For an "algebra" of type $\mathbf{F}\ A \rightarrow A$, where $\mathbf{F}$ is a functor from $\mathcal{CPO}$ to $\mathcal{CPO}$, there is a catamorphism of type

$$(\mathbf{F}\ A \rightarrow A) \rightarrow \mu\mathbf{F} \rightarrow A$$

Intuitively this reads "give me a way of rewriting a single level of $\mathbf{F}$, and I'll re-write the whole data structure for you". `foldr` has this property, though the type is slightly different because of the use of currying. We have already seen that the list catamorphism is a useful tool for structuring list consumption. It can be argued that catamorphisms are the natural way to consume other data structures (Meijer & Jeuring 1995).

6.2.1 Fold promotion

There is another approach to removing intermediate lists that rather than using rules like the `foldr/build` rule uses a fold promote rule. To see how this works, consider the example:

```
and (map fn xs)
```

We can represent both the `maps` in terms of `foldr`.

```
foldr (\ a1 b1 -> a1 && b1) True (foldr ((:).fn) xs)
```

But now we are stuck because of the lack of a straightforward `foldr` of `foldr` rule.

$$\texttt{foldr}\ k_1\ z_1\ (\texttt{foldr}\ k_2\ z_2\ e) = \ ???$$

It is certainly possible to invent a more specialised transformation, such as this one:

$$\texttt{foldr}\ k_1\ z_1\ (\texttt{foldr}\ ((:).k_2)\ \texttt{[]}\ e) = \texttt{foldr}\ (k_1 . k_2)\ z_1\ e$$

This will successfully transform our example to:

```
foldr ((&&).fn) True xs
```

It would be nice, however, if we had a more general transformation. The reason that there is no straightforward `foldr/foldr` rule is because the outer `foldr` has no "handle" on the way in which the inner `foldr` is producing its output list. It is *exactly* for this reason the we use `build` (c.f. § 2.2.2).

Sheard & Fegaras (1993) encountered exactly the same problem, but developed a different solution. They use a fold promotion system (Malcolm 1989), that is, among other things, a way of generating `foldr` of `foldr` rules. For lists, their promotion theorem is:

$$\begin{aligned} g\ (\texttt{foldr}\ f\ z\ xs) &= \texttt{foldr}\ f'\ z'\ xs \\ &\quad \texttt{where} \\ f'\ r_1\ r_2 &= g\ (f\ x_1\ x_2), \qquad \rho[x_1/r_1, g(x_2)/r_2] \\ z' &= g\ z \end{aligned}$$

The key step is *spotting* instances of $g(x_2)$ and replacing them with r_2. (This is analogous to the fold step in Burstall & Darlington (1977)). Considering our example `and (map f xs)` again. First we define g, f and z:

$$\begin{aligned} g &= \texttt{foldr (\textbackslash\ a1 b1 -> a1 \&\& b1) True} \\ f &= \texttt{\textbackslash\ a2 b2 -> fn a1 : a2} \\ z &= \texttt{[]} \end{aligned}$$

Now we can derive z':

$$\begin{aligned} z' &= \texttt{foldr (\textbackslash\ a1 b1 -> a1 \&\& b1) True}\ z \\ &= \texttt{foldr (\textbackslash\ a1 b1 -> a1 \&\& b1) True []} \\ &= \texttt{True} \end{aligned}$$

And finally we can derive f':

$$
\begin{aligned}
f'\ r_1\ r_2 &= \texttt{foldr (\textbackslash\ a1 b1 -> a1 \&\& b1) True}\ (f\ x_1\ x_2) \\
&\qquad \texttt{where}\ \rho = [x_1/r_1, \texttt{foldr (\textbackslash\ a1 b1 -> a1 \&\& b1) True}\ x_2/r_2] \\
&= \texttt{foldr (\textbackslash\ a1 b1 -> a1 \&\& b1) True}\ (fn\ x_1 : x_2) \\
&= \texttt{fn x\_1 \&\& foldr (\textbackslash\ a1 b1 -> a1 \&\& b1) True}\ x_2 \\
&= \texttt{fn}\ r_1\ \texttt{\&\&}\ r_2
\end{aligned}
$$

Replacing these results into the `foldr` gives:

```
foldr (\ r1 r2 -> fn r1 && r2) True xs
```

This style of deforestation is actually quite close conceptually to traditional deforestation. Consumers are pushed in towards producers. The intuition behind how this technique works is the transformation system is "taught" how `foldr` treats its argument, allowing functions that consume a `foldr` to look inside `foldr`'s arguments in an intelligent way. Because the recursion in captive inside the `foldr`, the fold step is easier than in deforestation.

6.2.2 Warm fusion

There has been some recent work combining the advantages of both cheap deforestation and the fold promotion technique given above. Launchbury & Sheard (1995) generalise the fold promotion to explicit recursion, and then use this to explicitly derive `foldr` and `build`.

This work is not list specific, and they use the `foldr/build` rule (and its generalisation) as their fusion rule. Because this work was done independently from the more recent work on cheap deforestation, no consideration is given to deriving `augment`, and its generalisation. In particular, their algorithm can take:

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

and automatically derive:

```
append xs ys = build (\ c n -> foldr c (foldr c n ys) xs)
```

As we have already discussed in § 3.4, this definition is not practical for a serious implementation. It should be possible to add the automatic derivation of `augment` to their algorithm, overcoming this problem.

6.2.3 `foldr/build` deforestation in calculational form

`foldr/build` deforestation has been re-expressed and extended into a calculational form, in Takano & Meijer (1995). The paper uses the "bananas and lens" notation to explain their ideas (Meijer, Fokkinga & Paterson 1991). Though quite technical, the paper does contain some interesting extensions to the concepts introduced by `foldr/build` deforestation.

First the calculational form allows the generalisation to arbitrary datatypes of the `foldr/build` rule to be concisely expressed. Secondly they observe that `foldr/build` rule has a dual. The `foldr/build` rule works with catamorphisms while the dual rule works over anamorphisms. Anamorphisms are unfolds, so the new rule, expressed in the same framework as cheap deforestation, could be called the `unbuild/unfold` rule. The `unbuild/unfold` rule for lists is:

$$\texttt{unbuild}\ f\ (\texttt{unfold}\ g\ h)\ =\ f\ g\ h$$

where the one possible definition of `unbuild` and `unfold` is:

```
data Maybe a = Nothing | Just a

unfold :: (b -> Maybe (a,b)) -> b -> [a]
unfold fn b =
      case fn b of
        Nothing -> []
        Just (a,b) -> a : unfold fn b

unbuild :: ((b -> Maybe (a,b)) -> b -> c) -> [a] -> c
unbuild fn lst = fn outL lst
    where
        outL []     = Nothing
        outL (x:xs) = Just (x,xs)
```

Like `build`, `unbuild` has a non Hindley-Milner type.

Takano & Meijer (1995) then go on to use hylomorphisms, a generalisation of both catamorphisms and anamorphisms, as their canonical form. They give reduction rules for hylomorphisms (including two fusion laws that correspond to the `foldr/build` rule and the `unbuild/unfold` rule). They also give a non-trivial reduction strategy to achieve the maximum deforestation opportunity. Paterson (1995) has a single rule that is a generalisation of the two fusion laws on hylomorphisms. This simplifies somewhat the requirement on reduction ordering.

Some work has been done to express the "bananas and lens" notation directly in Haskell augmented with constructor classes (Jones 1995). This work could allow a straightforward user driven implementation of these ideas inside an equational reasoning system. For any automated implementation to get past the prototype stage, though, technology equivalent to the material introduced in Chapters 3 and 4 would have to be incorporated.

6.3 Schema based deforestation systems

Schema based transformations look for known fixed patterns of functions in programs, and using rules about these known patterns, optimise the programs. In some cases these transformations can be used to remove intermediate data structure. For example, an algebraic transformation which eliminates an intermediate list is the `map` of `map` rule.

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

It is straightforward to show that a large set of algebraic rules would be impractical as for a general list removal scheme. For n different list processing functions, n^2 rules would be required. Furthermore, each algebraic transformation may potentially produce a totally new list processing function.

One possible way of combating this rule explosion is trying to factor out the essence of list manipulations (like list production, list consumption, mapping over a list, etc) and express list processors in terms of a small set of combinators. Then a small set of rules can transform on these combinators, eliminating intermediate lists. Cheap deforestation is one such system. We now discuss two other systems that use a schema based approach.

6.3.1 Wadler's schema deforestation

Wadler (1981) attempts to capture list removal opportunities by using a small set of combinators. We transliterate the ideas presented into the more convenient Haskell notation. He uses as his key combinators the three functions `map`, `foldl`, and `generate`. `map` and `foldl` we have already met:

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

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

```
             map f (map g xs)  = map (f.g) xs
         foldl f a (map g xs)  = foldl h a xs
                                   where
                                    h a' x = f a' (g x)
    map f (generate p g1 g2 x) = generate p h g2 x
                                    where
                                      h b' = f (g1 b')
foldl f a (generate p g1 g2 x) = h a x
                          where
                             h a' b' =
                               if p b'
                               then a'
                               else h (f a' (g1 b')) (g2 b')
```

Figure 6.2: Wadler's algebraic deforestation transformations

`generate` is a list producer that uses a predicate and two functions to create the elements of the list from a single value.

```
generate :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
generate p f g n =
        if p n
        then []
        else f n : generate p f g (g n)
```

Wadler's small set of algebraic transformations are given in Figure 6.2. He shows that programs that express list processing functions in terms of this small set of functions can have many of their intermediate lists removed. As an example, consider:

```
foo n = sum (map square (from 1 n))
```

We can now express `sum` and `from` in terms of our key list combinators.

```
foo n = foldl (+) 0 (map square (generate (n <) id (+1) 1))
```

Now we can use the `foldl-of-map` rule:

```
foo n = foldl h1 0 (generate (n <) id (+1) 1)
   where
      h1 a x = a + square x
```

Finally, using the `foldl-of-generate` rule gives:

```
foo n = h2 0 1
  where
      h2 a b =
       if b > n
       then a
       else h2 (a + square (id b)) (b + 1)
```

This version of the program has no intermediate lists.

`foldr/build` deforestation subsumes this deforestation algorithm. We can express the two list consumers `foldl` and `map` in terms of `foldr`, and can express the two list producers `generate` and `map` in terms of `build`. We have seen `foldl` and `map` in terms of `foldr` and `build` before. `generate` in terms of `build` has the definition:

```
generate p f g x
  = build (\ c n ->
                let
                   h x = if p x
                         then n
                         else f x 'c' h (g x)
                in
                   h x)
```

The four laws given in Figure 6.2 are directly replaceable with the single `foldr/build` transformation.

The principal shortcoming of this method of list removal is that `generate` is not a very general way of producing lists. There is no way of generating a constant list, for example. `build` is a more general way of producing lists.

6.3.2 Maessen's schema deforestation

Maessen (1994) presents a desugaring system for the functional language pH that uses schemas over list combinators to achieve list removal. To express list consumers Maessen uses the combinators `reduce` and `map`. As many consumers as possible are expressed in the form:

(`reduce` *a* `id` (`map` *u* `xs`)) *t*

where `xs` is the list being consumed. `reduce` is a fold that can be expressed as *either* a `foldr` or `foldl`.

```
reduce f z = foldr f z
```

or

```
reduce f z = foldl f z
```

Because of this, there is a requirement that a is associative. Maessen uses this choice to enhance the handling of parallel lists.

To express `foldr f z` in this canonical form can be done by defining a, u and t to suitable values:

```
a = \ l r -> l . r
u = \ e t -> f t e
t = z
```

Maessen uses the function `unfold` to express list production. `unfold` has the definition:

```
unfold p f v = h v
  where
   h v | p v       = []
       | otherwise = a : h b
           where
              (a,b) = f v
```

and there is a `reduce/unfold` rule, analogous to the `foldr/build` rule.

```
(reduce a (\x -> x) u (unfold p f v)) t
```

$$\equiv$$

```
h i x
  where
   h v r | p v       = r
         | otherwise = f (g a) (h b) r
             where
                (a,b) = j v
```

Maessen also used a large set of other list identities when desugaring.

Maessen's emphasis is somewhat different from traditional list removal techniques, in that he specifically wanted to handle the parallel traversals of lists efficiently. It is as yet unclear how effective his optimisations are, because Maessen (1994) contained only two small benchmarks. Furthermore, it appears that Maessen's will need the arity analysis technology that we introduced in § 4.4 to properly handle the inlining of `foldr`, `reduce`, etc.

Directly comparing Maessens system with cheap deforestation, it appear that cheap deforestation is more straightforward, but it is unclear if Maessen's system is more powerful (c.f. § 7.3.5). `build` is also a more general producer than `unfold`, being able to represent a larger class of producers.

6.4 Parameterisation over data structure construction

Parameterising the creation of data structures has been considered by others. We discuss the parameterisation over `[]` in § 6.4.1 and the parameterisation over both `(:)` and `[]` in § 6.4.2

6.4.1 Parameterisation of `[]`

Parameterising over the *nil* of a list has been proposed as a possible optimisation over traditional lists (Hughes 1984). Hughes, who expresses this in terms of a user applied optimisation, provides two functions.

```
abs :: [a] -> AbsList a
rep :: AbsList a -> [a]
```

`AbsList` is an optimised representation of lists, especially optimised for efficient appends. `abs` takes a conventional list, and returns an "optimised" version, while `rep` takes an "optimised" list, and returns a conventional one. These functions have the property:

$$\forall a.\ \texttt{abs (rep } a\texttt{)} = a$$

In Haskell, these functions can be given the definitions:

```
type AbsList a = [a] -> [a]
abs lst = \ rest -> lst ++ rest
rep abslist = abslist []
```

This technique works by postponing the actual construction of the list until given a list continuation (a tail). The important consequence of using this alternative representation of lists is that the append operation now can be performed in constant-time, because appending is simply providing a list continuation. `(++)` for `AbsList` has the definition:

```
appendR :: AbsList a -> AbsList a -> AbsList a
appendR f g = f . g
```

As an example of AbsList in action, consider a datatype Tree.

```
data Tree = Leaf Int | Node Tree Tree
```

A function to construct a list of the contents of all the leaves might have the definition:

```
printLeaf :: Tree -> [Int]
printLeaf (Leaf i)     = [i]
printLeaf (Node t1 t2) = printLeaf t1 ++ printLeaf t2
```

This program has a runtime complexity of $O(n^2)$, where n is number of leaves. Now, if we re-write the function to use AbsList to create the result, we have:

```
printLeaf :: Tree -> AbsList Int
printLeaf (Leaf i)     = abs [i]
printLeaf (Node t1 t2) = printLeaf t1 `appendR` printLeaf t2
```

This new version has a runtime complexity of $O(n)$, because of the constant-time append. In Haskell, this efficient style of list is used inside the Text class, specifically to give the improved complexity when printing large data structures.

This idea of optimising append using parameterisation over *nil* was taken one step further in Wadler (1987*b*), where Wadler describes a local transformation that removes many appends from a program by using this improved representation of lists. For any function of the type

f :: a_1 -> ... a_n -> [b]

Wadler defines an auxiliary function

f' :: a_1 -> ... a_n -> AbsList b

where

(f x_1 ... x_n) ++ y = f' x_1 ... x_n y

This can be done by taking the original definition:

f a_1 ... a_n = <exp>

and giving:

f a_1 ... a_n = f' a_1 ... a_n []
f' a_1 ... a_n y = <exp> ++ y

All instances of f are replaced with f's definition, like the worker/wrapper scheme in Peyton Jones & Launchbury (1991). Wadler now uses several rules to exploit properties of append to give an efficient program.

We describe a system in § 7.4 that gets all the benefits of Wadler's automation of using `AbsList`, but because it abstracts over both *cons* and *nil*, it gets additional benefits.

6.4.2 Parameterisation of (:) and []

After we had developed the `foldr/build` transformation we discovered that the key concept of parameterisation over both `(:)` and `[]` had been applied to list removal before. It was considered by the pioneer functional programmer, Burge as long ago as 1977 (Burge 1977).

First Burge suggests expressing list consumption, where possible, using a "universal" list consumer. He proposed a function, called `L` (we will call it `reduce`), which is a variant of `foldl`.

```
reduce :: a -> (b -> a -> a) -> [b] -> a
reduce a g []     = a
reduce a g (x:xs) = reduce (g x a) g xs
```

He then gives examples of functions like `map` and `filter`, in terms of this function. For example

```
map f xs = reduce [] ((:).f) xs
```

However, in what will seem foreign to modern functional programmers, his version of `map` (and `filter`, `append`, etc) actually reverses the list they consume. So for Burge's version of `map`:

```
map (+1) [1..4] = [5,4,3,2]
```

Secondly, Burge also suggests that functions that produced lists should be parameterised with respect to their `(:)` and `[]`. Burge advocates having a second version of `map`:

```
map' n c f xs = reduce n (c.f) xs
```

where `c` and `n` are used to construct the result list. He finally gives a set of loop jamming rules, that *jam* loops together. The jamming rules have the form

$$\texttt{reduce f z } (foo\ x_1\ \ldots\ x_n) = foo'\ x_1\ \ldots\ x_n\ \texttt{f z}$$

where *foo* is a list producer, and *foo′* is its abstracted partner. So the rule for `map` is:

```
        reduce f z (map g xs) = map' g xs f z
```

Furthermore, this `map'` can be unfolded, revealing a new instance of `reduce`.

As far as we know, there has been no implementation of this work. Also, some practical considerations have been overlooked. For example, `map` is both a list producer and consumer. Because of this it wants to be both in `reduce` form, so that one of the `reduce` of something rules will be possible, and also left as a `map`, so that the `reduce` of `map` rule can operate. Our introduction of `build` neatly sidesteps these problems, allowing the successful detection of parameterised function that can also be simultaneously expressed in terms of a general list consumer.

6.5 Optimised representations of lists

Sometimes lists can not be removed outright. There are, however, a number of techniques that involve optimising the representation for lists to make them cheaper to produce and consume (Hall 1994*a*, Hall 1994*b*, Shao, Reppy & Appel 1994). For example, consider a list of type `[(a,b)]`. One optimised data structure for such a list is:

```
        data TupleList a b
                = Cons a b (TupleList a b)
                | Nil
```

When using such an optimisation the producer and consumer need to be modified to use the new datatype, either automatically or by hand. Furthermore, in lazy languages there are strictness constraints on which optimisations are applicable. The effect of using optimised representations for lists can be significant. Hall (1994*a*) observed one list intensive program improve in execution time by around 45%.

When using such optimisations at the same time as cheap deforestation some care needs to be taken, but both can be used inside the same framework. The two main points to be considered are:

- If a list can be deforested outright this is the preferable choice.
- If a list can not be removed, an optimised representation is preferable to the unoptimised representation.

These considerations can be implemented in one of two ways. Firstly the deforestation system can act before the list representation optimisation. Alternatively, the deforestation system could be generalised to cope with the new datatypes that are being used to represent lists (c.f. § 7.3.2,§ 7.3.4).

Chapter 7

Conclusions and Further Work

In this thesis we have demonstrated that a cheap form of deforestation is feasible inside the framework of a *real* functional language compiler. We have made two principal contributions.

- We have presented a new *practical* approach to list-removal.
- This new approach has been demonstrated as practical by developing it into a working (list) deforestation scheme inside a real functional language compiler.

In this chapter we reflect on these two contributions, and present several avenues for further work.

7.1 A new list removal algorithm

Cheap deforestation is a new deforestation algorithm, though as we have seen in Chapter 6, many aspects of cheap deforestation have appeared in some form before. There are three key concepts at the heart of cheap deforestation, namely: using `foldr`, using `build`, and using the `foldr/build` rule.

- The use of `foldr` as a list consumer is in line with current thinking on data structure removal. Other state-of-the-art data structure removal and data structure transformation systems that use the same concept, using a catamorphism as a generic consumer. The practical considerations in Chapter 3 enhance our understanding of the pragmatics behind actually using `foldr` to express consumption.

- The use of `build` (and `augment`) appear to be unique. Though from a theoretical point of view, finding constructors is simply a matter of observation, the practical ramifications of being able to gain straightforward access to the constructs are tangible. Indeed, `build` seems to be able to reach the constructors that other deforestation systems fail to reach.

  Using `build` also marks a departure for the traditional data structure removal ordering. Traditional deforestation works by pushing the consumer inside the producer, whereas `build` brings out the data structure to the consumer.

- The simplicity of the `foldr/build` rule has a certain aesthetically pleasing quality. It is always nice when a simple rule is also a powerful and useful rule.

7.2 An implementable list removal algorithm

It was possible to implement cheap deforestation reasonably efficiently. The implementation gave rise to some interesting developments.

- The `foldr/augment` rule was driven by the desire not to do extra `foldr` consumptions. It is an interesting generalisation of the `foldr/build` rule.
- The enabling technologies presented in Chapter 4 are general, and add to the growing army of possible transformations that can be performed on functional languages.
- When arrays in Haskell were being designed, specification techniques like array comprehensions presumed that a *simple* deforestation system would emerge that could remove the intermediate lists that array comprehensions produce. Cheap deforestation was found to fit exactly this criteria.

One very difficult aspect of deforestation that will become an increasing hindrance to more and more powerful transformations is the interaction between transformations. In our case, cheap deforestation competes with full laziness. The code fragment

```
\ f z -> foldr f z [1,2,3,4]
```

has intermediate list successfully removed, but the almost identical

```
\ f z -> foldr f z [1..4]
```

is not deforested. This is because `[1..4]` would be lifted out, though the λ's. If we did not use full laziness, then both would fuse. There are good reasons, however, why we need to use full laziness before cheap deforestation.

With every new optimisation inserted into the compiler, all the optimisation implementers had to carefully consider how their optimisation interacted with other optimisations. It is simply not practical to insert an optimisation pass without knowledge of how the current optimisations are interacting. Even with careful documentation, handling the interaction between the optimisations is a black art.

An automatic approach to these interactions might, at least in principle, be possible. Optimisations could "vote", declaring how much benefit they guarantee to give to some specific code, and a dynamic algorithm could be used to handle iterations of this system. There would be, however, major technical problems to overcome before this system could be practical, such as what form a "vote" would actually take.

7.3 Further work

One especially exciting aspect of the work presented in this thesis is the number of new avenues that have been exposed as suitable for exploration.

7.3.1 Automatically deriving `foldr` and `build`

Under the cheap deforestation system presented in this thesis, deforestation can only occur in two ways:

- A program fragment is written that uses inlined prelude functions (and list comprehensions), exposing `foldr`, `build`, etc.
- A programmer explicitly uses `foldr`, `build`, etc., and also requests explicitly that producers and consumers are inlined.

Though this scheme does allow opportunities for optimisation, it would be much more pleasant for the programmer to be neither tied to using prelude functions or explicitly using our deforestation combinators. As we have already seen there has been some work on deriving `foldr` and `build` automatically (Launchbury & Sheard 1995).

7.3.2 Other data structures

The `foldr/build` transformation, and its generalisation the `foldr/augment` rule, both have a natural extension to many other data structures. There is a lot of work to do considering the pragmatic aspects of data structures other than lists.

`foldr` and `build` are both straightforward to rewrite for other data structures, because their definition follows naturally from the datatype definition. `augment` is slightly tricker, because of the asymmetric treatment of constructors. The general rule for `augment` is that `augment` has an extra argument for each nullary constructor. So `augment` for `Bool` would be:

```
augmentBool :: (a -> a -> a) -> Bool -> Bool -> Bool
augmentBool g t f = g t f
```

On the other hand, for types that do not have nullary constructors (like n-tuples, where $n > 0$), `augment` would be the same as `build`.

7.3.3 Selective inlining of prelude functions

In might be possible to build an unfolding mechanism into the simplifier, so that prelude functions are only inlined if this will expose an instance of a deforestation rule. Producers and consumers could have types that represent if they are good or bad producers and/or consumers. The simplifier would then use these types to decide when to inline list processors.

Though the code size increase that was caused by cheap deforestation was quite small, one reason for the extra compilation time was the time spent reading the definitions of many prelude functions and unfolding them. A selective inlining scheme should cut the cost of using cheap deforestation.

In § 7.4 we present a sketch of a technique that allows deforestation opportunities to cross function boundaries without needing wholesale inlinings.

7.3.4 Dynamic deforestation

In § 6.5 we discussed new, optimised representations for lists that replace the traditional "cons" and "nil" approach. There is interesting way to combine cheap deforestation with this approach. Consider the data definition:

```
begin List a = Augment ((a -> b -> b) -> b -> b) (List a)
             | Nil
```

The type variable b here, just like with augment, is scoped over Augment's first argument only. foldr now has the definition:

```
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z (Augment g h) = g f (foldr f z h)
foldr f z Nil           = Nil
```

When this foldr pattern-matches on an Augment it uses at runtime the foldr/augment rule, providing a *dynamic deforestation.*

There is however a caveat with this representation of lists. Consider the list

```
fib 1 : fib 2 : ... : fib 30 : []
```

using our new representation we would use

```
Augment (\ c n ->
  fib 1 'c' fib 2 'c' ... 'c' fib 30 'c' n)
      Nil
```

Unfortunately now the elements inside the list are recalculated *each time the list is demanded*, not just once as would be the operational semantics for the original list under lazy evaluation. To use this optimised list data structure requires a more elaborate update mechanism, that updates Augment based lists into more traditional representations, if required. In practice this would be a non-trivial change to the runtime system.

This new representation for lists is significant, however, because a completely new form of virtual machine might be possible. Currently inside our virtual machines for symbolic languages we consider a data structure efficient if each constructor in the source language is represented by a vector and a tag on the target architecture. A new style of virtual machine could store all its data structures in an Augment like form, and perform dynamic deforestation as a reduction step. A more elaborate update mechanism could be built into the virtual machine, including a mechanism for dynamically eliminating redundant updates, like dashing (Peyton Jones & Salkild 1988).

7.3.5 The bigger picture

Cheap deforestation is yet another data structure removal technique, albeit a successful and practical one. As we have explored in Chapter 6, there are many other ways of removing intermediate data structures. All the different deforestation systems have their strengths and weaknesses, and relationships between them are still somewhat informal. A formal approach to classifying the relationships of the different systems might lead to some interesting ways of *combining* deforestation systems.

7.4 Crossing the function boundary

In this section we sketch a method for improving the scope of cheap deforestation by allowing the effects of deforestation to cross the function boundary. In the same way as Peyton Jones & Launchbury (1991) transmitted strictness information over the function boundary using workers and wrappers, we can also use workers and wrappers to transmit deforestation opportunities. When such a deforestation opportunity is detected, we take a list producer function definition:

$$\texttt{f}\ x_1 \ldots x_n\ \texttt{= build (\textbackslash\ c n -> <exp>)}$$

and split it into two mutually recursive functions, a worker (f#) and a wrapper (f):

$$\begin{aligned} \texttt{f\#}\ x_1 \ldots x_n\ \texttt{c n} &\ \texttt{= <exp>} \\ \texttt{f}\ x_1 \ldots x_n &\ \texttt{= build (\textbackslash\ c n ->}\ f\texttt{\#}\ x_1 \ldots x_n\ \texttt{c n)} \end{aligned}$$

We now inline the small wrapper `f` at all its call sites.

Consider the Haskell program:

```
foo xs = map (+ 1) xs
bar ys = filter p (foo ys)
```

We now inline `map` and `filter`, as we would in traditional cheap deforestation, giving:

```
foo xs = build (\ c n -> foldr (\ a b -> (a + 1) `c` b) n xs)
bar ys = build (\ c n ->
             foldr (\ a b ->
                if p a
                then a `c` b
                else b) n (foo ys))
```

The list producer `foo` can be split into a worker and wrapper:

```
foo# xs c n = foldr (\ a b -> (a+1) `c` b) n xs
foo xs      = build (\ c n -> foo# xs c n)
```

`foo`, now a small wrapper, can be unfolded into `bar`, giving:

```
foo# xs c n = foldr (\ a b -> (a+1) `c` b) n xs
bar p ys
 = build (\ c n ->
      foldr (\ a b ->
        if p a
        then a `c` b
        else b) n (build (\ c n -> foo# ys c n)))
```

We have an instance of the foldr/build rule. Taking the optimisation opportunity gives:

```
foo# xs c n = foldr (\ a b -> c (a+1) b) n xs
bar p ys
 = build (\ c n ->
      foo# ys (\ a b ->
        if p a
        then a `c` b
        else b) n)
```

We have achieved deforestation without a wholesale inlining of the list producer into the consumer. This technique is especially useful for when foo is a large function, and therefore too expensive, in terms of code size increase, to inline. Furthermore, it is also possible to transmit good list consumption using workers and wrappers.

We can also use the worker/wrapper technology to remove a list that exists between *recursive* calls of a single function. Consider:

```
data Tree a = Tree a [Tree a]
preorder (Tree a ts) = a : concat (map preorder ts)
```

preorder can be automatically transformed, using a slightly modified version of cheap deforestation, into:

```
preorder xs =
  build (\ c n ->
   case xs of
     Tree a ts ->
        a `c` foldr (\ x y -> foldr c y (preorder x))
                    n ts)
```

Even though preorder is recursive, we can still split it into a worker and wrapper:

```
preorder# xs c n =
  case xs of
     Tree a ts ->
        a `c` foldr (\ x y -> foldr c y (preorder x))
                    n ts
preorder xs = build (\ c n -> preorder# xs c n)
```

Now we can once again unfold the wrapper (preorder) into all its call sites, because it is small.

```
preorder# xs c n =
  case xs of
    Tree a ts ->
        a `c` foldr (\ x y ->
                 foldr c y
                      (build (\ c n -> preorder# x c n)))
                    n ts
preorder xs = build (\ c n -> preorder# xs c n)
```

Finally, we can take advantage of the new instance of the `foldr/build` rule, giving:

```
preorder# xs c n =
  case xs of
     Tree a ts ->
        a `c` foldr (\ x y -> preorder# x c y)
                    n ts
preorder xs = build (\ c n -> preorder# xs c n)
```

This version of preorder does not have any intermediate lists between the recursive calls. It also has an improved runtime complexity.

We have a prototype implementation of this worker/wrapper enhancement for cheap deforestation. One factor that is delaying the reporting of results from our prototype is the fact that there is a substantial performance difference between:

```
let foo x y z c n = c x (c y (c z n))
in ... foo a b c (:) [] ...
```

and

```
let foo x y z = x : y : z : []
in ... foo a b c ...
```

Allocating a *cons* cell is cheaper than calling a function that simply ends up allocating a *cons* cell! This problem could be solved by having specialised versions of the workers. Further measurements are needed regarding any code increase this might cause.

References

Aho, A., Sethi, R. & Ullman, J. (1986), *Compilers - Principles, Techniques and Tools*, Addison-Wesley. (pp. 81, 87)

Anderson, S. & Hudak, P. (1990), Compilation of Haskell array comprehensions for scientific computing, *in* Programming Language Design and Implementation, ACM, pp. 137–149. (p 84)

Appel, A. (1992), *Compiling with Continuations*, Cambridge University Press. (p 61)

Augustsson, L. (1987), Compiling lazy functional languages, Part II, PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden. (pp. 7, 42, 65)

Augustsson, L. & Johnsson, T. (1989), The Chalmers lazy ML compiler, *Computer Journal* **32**(2), 127–141. (p 2)

Bird, R. S. (1989), Algebraic identities for program calculation, *Computer Journal* **32**(2), 122–126. (p 16)

Bird, R. S. & Wadler, P. (1988), *Introduction to Functional Programming*, International Series in Computer Science, Prentice-Hall. (pp. 3, 5)

Burge, W. H. (1977), Examples of program optimization. RC 6351, IBM Thomas J Watson Research Centre. (p 134)

Burstall, R. M. & Darlington, J. (1977), A transformational system for developing recursive programs, *Journal of the ACM* **24**(1), 44–67. (pp. 14, 119, 120, 125)

Cardelli, L. & Longo, G. (1991), A semantic basis for Quest, *Journal of Functional Programming* **1**(4), 417–458. (p 28)

Chin, W. N. (1990), Automatic methods for program transformation, PhD thesis, University of London. (p 121)

Fairbairn, J. (1985), Design and implementation of a simple typed language based on the lambda calculus, PhD thesis, University of Cambridge Computer Laboratory. (p 28)

Ferguson, A. B. & Wadler, P. (1988), When will deforestation stop, *in* K. Davis & R. J. M. Hughes, eds, Glasgow Workshop on Functional Programming, Internal Report. (p 121)

Fleming, P. J. & Wallace, J. J. (1986), How not to lie with statistics - the correct way to summarise benchmark results, *CACM* **29**(3), 218–221. (p 96)

Fradet, M. & Metayer, D. L. (1991), Compilation of functional languages by program transformation, *TOPLAS* **13**(1). (p 61)

Gill, A. J., Launchbury, J. & Peyton Jones, S. L. (1993), A short cut to deforestation, *in* Arvind, ed., Functional Programming and Computer Architecture, Copenhagen, Denmark, ACM, pp. 223–232. (pp. 8, 82, 119)

Gill, A. J. & Peyton Jones, S. L. (1994), Cheap deforestation in practice: An optimiser for Haskell, *in* IFIP, Hamburg, Germany, Vol. 1, pp. 581–586. (p 8)

Hall, C. V. (1994*a*), An optimistic view on life: transforming lists representations. Unpublished report, Department of Computing Science, Glasgow University. (p 135)

Hall, C. V. (1994*b*), Using hindley-milner type inference to optimize list representation., *in* Lisp and Functional Programming, New York. (p 135)

Hamilton, G. W. (1993), Compile-time optimisation of store usage in lazy functional programs, PhD thesis, University of Stirling. (p 121)

Harper, R., McQueen, D. & Milner, R. (1986), Standard ML, Technical Report ECS-LFCS-86-2, University of Edinburgh. (p 2)

Hartel, P. H. (1994), Benchmarking implementations of lazy functional languages II: two years later, Technical report, Department of Computer Systems, University of Amsterdam. (p 90)

Hartel, P. H. & Langendoen, K. G. (1993), Benchmarking implementations of lazy functional languages, *in* Arvind, ed., Functional Programming and Computer Architecture, Copenhagen, Denmark, ACM, pp. 341–349. (p 90)

Henderson, P. (1980), *Functional Programming: Application and Implementation*, Prentice-Hall. (p 2)

Hudak, P. (1987), Arrays, non-determinism side-effects and parallelism: A functional perspective, *in* P. Hudak, ed., Graph Reduction Workshop, Santa Fé, Vol. 279 of *LNCS*, Springer-Verlag, pp. 312–327. (p 3)

Hudak, P., Peyton Jones, S. L., Wadler, P. et al. (1992), Report on the functional programming language Haskell, Version 1.2, *SIGPLAN Notices* **27**(5). (pp. 2, 42)

Hudak, P. & Sundaresh, R. S. (1989), On the expressiveness of purely-functional I/O systems, Technical report, Department of Computer Science,Yale University. YALEU/DCS/RR-665. (p 3)

Hughes, R. J. M. (1984), A novel representation of lists and its application to the function reverse, Technical report, Programming Methodology Group, Department of Computer Science, Chalmers University of Technology, Sweden. PMG-38. (p 132)

Hughes, R. J. M. (1989), Why functional programming matters, *Computer Journal* **32**(2), 98–107. (pp. 4, 16)

Johnsson, T. (1983), The G-machine: An abstract machine for graph reduction, *in* Declarative Programming Workshop, University College London, pp. 1–19. (p 62)

Jones, M. P. (1995), Functional programming with overloading and higher-order polymorphism, *in* J. Jeuring & E. Meijer, eds, Proceedings of the First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, Vol. 925 of *LNCS*, Springer-Verlag. (p 127)

Kelsey, R. A. (1989), Compilation by program transformation, PhD thesis, Department of Computer Science, Yale University. YALEU/DCS/RR-702. (p 61)

Kernighan, B. W. & Ritchie, D. M. (1978), *The C Programming Language*, Prentice-Hall. (p 2)

Kort, J. (1996), Deforestation of a raytracer, Master's thesis, Department of Computer Science, University of Amsterdam, The Netherlands. (to appear). (p 83)

Launchbury, J. & Peyton Jones, S. L. (1996), State in Haskell, *Lisp and Symbolic Computation* . (to appear). (p 85)

Launchbury, J. & Sheard, T. (1995), Warm fusion: deriving build-catas from recursive definitions, *in* S. L. Peyton Jones, ed., Functional Programming and Computer Architecture, San Diego, California, ACM, pp. 314–323. (pp. 126, 139)

Maessen, J. (1994), Eliminating intermediate lists in pH using local transformations, Master's thesis, Department of Electrical Engineering and Computer Science. (pp. 130, 131)

Malcolm, G. (1989), Homomorphisms and promotability, *in* Mathematics of Program Construction, Springer-Verlag, pp. 335–347. (p 125)

Marlow, S. (1996), Deforestation for higher-order functional languages, PhD thesis, Department of Computing Science, Glasgow University. (To appear). (pp. 121, 123)

Marlow, S. & Gill, A. (1995), Personal communication. Curlers, Byres Road, Glasgow. (p 121)

McCracken, N. J. (1984), The typechecking of programs with implicit type structure, *Semantics of data types* pp. 301–315. (p 28)

Meijer, E., Fokkinga, M. M. & Paterson, R. A. (1991), Functional programming with bananas, lenses, envelopes and barbed wire, *in* R. J. M. Hughes, ed., Functional Programming and Computer Architecture, Boston, Cambridge, Mass., Vol. 523 of *LNCS*, Springer-Verlag, pp. 124–144. (p 127)

Meijer, E. & Jeuring, J. (1995), Merging monads and folds for functional programming, *in* J. Jeuring & E. Meijer, eds, Proceedings of the First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, Vol. 925 of *LNCS*, Springer-Verlag. (p 124)

Milner, R. (1978), A theory of type polymorphism in programming, *Journal of Computer and System Sciences* **17**(3), 348–375. (p 28)

Nikhil, R. (1988), Id Nouveau (version 88.0) reference manual, Technical report, MIT Laboratory for Computer Science, Cambridge, Mass. (p 2)

Partain, W. (1992), The `nofib` benchmarking suite, *in* J. Launchbury & P. M. Sansom, eds, Glasgow Workshop on Functional Programming, Ayr, Scotland, Workshops in Computing, Springer-Verlag. (p 90)

Paterson, R. A. (1995), Personal communication. (p 127)

Peyton Jones, S. L. (1987), *The Implementation of Functional Programming Languages*, International Series in Computer Science, Prentice-Hall. (pp. 32, 63, 65, 150)

Peyton Jones, S. L. (1989), Parallel implementations of functional programming languages, *Computer Journal* **32**(2), 175–186. (p 3)

Peyton Jones, S. L. (1992), Implementing lazy functional languages on stock hardware: the spineless tagless G-machine, *Journal of Functional Programming* 2(2), 127–202. (pp. 3, 90)

Peyton Jones, S. L. & Launchbury, J. (1991), Unboxed values as first class citizens in a non-strict functional language, *in* R. J. M. Hughes, ed., Functional Programming and Computer Architecture, Boston, Cambridge, Mass., Vol. 523 of *LNCS*, Springer-Verlag. (pp. 86, 133, 142)

Peyton Jones, S. L. & Salkild, J. (1988), The spineless tagless g-machine, *in* K. Davis & R. J. M. Hughes, eds, Glasgow Workshop on Functional Programming, Internal Report, pp. 146–160. (p 141)

Reynolds, J. C. (1983), Types, abstraction and parametric polymorphism, *in* R. E. A. Mason, ed., Information Processing 83, North-Holland, Amsterdam, pp. 513–523. (p 27)

Roe, P. (1991), Parallel programming with functional languages, PhD thesis, Department of Computing Science, Glasgow University. (p 3)

Santos, A. (1995), Compilation by transformation in non-strict functional languages, PhD thesis, Department of Computing Science, Glasgow University. (pp. 3, 11, 32, 68, 78)

Santos, A. & Peyton Jones, S. L. (1992), On program transformation in the Glasgow Haskell Compiler, *in* J. Launchbury & P. M. Sansom, eds, Glasgow Workshop on Functional Programming, Ayr, Scotland, Workshops in Computing, Springer-Verlag. (p 3)

Shao, Z., Reppy, J. H. & Appel, A. W. (1994), Unrolling lists, *in* Lisp and Functional Programming, New York. (p 135)

Sheard, T. & Fegaras, L. (1993), A fold for all seasons, *in* Arvind, ed., Functional Programming and Computer Architecture, Copenhagen, Denmark, ACM, pp. 233–242. (pp. 119, 125)

Steele Jr., G. L. (1984), *Common Lisp - the Language*, Digital Press. (p 2)

Sun Microsystems (1993), Introduction to SpixTools. (p 89)

Takano, A. & Meijer, E. (1995), Shortcut deforestation in calculational form, *in* S. L. Peyton Jones, ed., Functional Programming and Computer Architecture, San Diego, California, ACM. (p 127)

Turchin, V. (1986), The concept of a supercompiler, *TOPLAS* 8(3), 292–326. (p 120)

Turchin, V. (1988), The algorithm of generalization in the supercompiler, *in* Bjørner, Ershov & Jones, eds, Partial Evaluation and Mixed Computation, North-Holland. (p 121)

Turner, D. A. (1985), Miranda: A non-strict functional language with polymorphic types, *in* P. Wadler, ed., Functional Programming and Computer Architecture, Nancy, France, Vol. 201 of *LNCS*, Springer-Verlag, pp. 1–16. (p 2)

Wadler, P. (1981), Applicative style programming, program transformation and list operators, *in* Symposium on Principles of Programming Languages, pp. 25–32. (p 128)

Wadler, P. (1983), Listlessness is better than laziness, PhD thesis, Department of Computer Science, Carnegie Mellon University. (p 120)

Wadler, P. (1987*a*), Compiling list comprehensions, in *The Implementation of Functional Programming Languages* (Peyton Jones 1987). (pp. 7, 42, 58, 65)

Wadler, P. (1987*b*), The concatenate vanishes. Unpublished report, Department of Computing Science, Glasgow University. (p 133)

Wadler, P. (1987*c*), A new array operation, *in* P. Hudak, ed., Graph Reduction Workshop, Santa Fé, Vol. 279 of *LNCS*, Springer-Verlag, pp. 328–335. (p 3)

Wadler, P. (1989), Theorems for free!, *in* D. B. MacQueen, ed., Functional Programming and Computer Architecture, London, Addison-Wesley. (p 27)

Wadler, P. (1990), Deforestation: transforming programs to eliminate trees, *Theoretical Computer Science* **73**, 231–248. (pp. 6, 14, 121)

Wadler, P. (1992*a*), Comprehending monads, *in* Lisp and Functional Programming. Special issue of selected papers from 6'th Conference on Lisp and Functional Programming, 1992. (p 3)

Wadler, P. (1992*b*), The essence of functional programming, *in* Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, Vol. 19. (p 3)

Wadler, P. & Blott, S. (1989), How to make ad-hoc polymorphism less ad-hoc, *in* Symposium on Principles of Programming Languages, Austin, Texas, USA, Vol. 19. (p 62)

Waters, R. (1991), Automatic transformation of series expressions into loops, *TOPLAS* **13**(1), 52–98. (p 120)