Eliminating Intermediate Lists in pH using Local Transformations

by

Jan-Willem Maessen

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of
the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May, 1994

© Jan-Willem Maessen 1994. All rights reserved.

The author hereby grants to MIT permission to reproduce
and to distribute copies of this thesis document in whole or in part,
and to grant others the right to do so.

Signature of Author_____
Department of Electrical Engineering and Computer Science
May 6, 1994

Certified by_____
Arvind
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by_____
F. R. Morgenthaler
Chairman, Departmental Committee on Graduate Theses

WITHDRAWN
Eng.
FROM
MASSACHUSETTS INSTITUTE
MIT LIBRARIES
JUN 15 1994
LIBRARIES

Eliminating Intermediate Lists in pH using Local Transformations

by

Jan-Willem Maessen

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 1994
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The extensive use of lists in functional programming languages exacts a cost penalty for important operations such as array construction. In addition, because lists are linear data structures, it is difficult to generate and traverse them efficiently in parallel given a purely-functional environment. Three common methods of traversal—left and right fold, and reduction—can be described in terms of `map` and `reduce` operations yielding higher-order functions; these higher-order functions can be represented entirely at compile time. This thesis examines this representation of list traversal, describes its implementation in the desugaring phase of the compiler for pH (an implicitly-parallel, mostly-functional dialect of the language Haskell). This approach to desugaring gives rise to a set of rules for compiling list comprehensions efficiently, and to a pair of functions `unfold` and `synthesize` which can be immediately eliminated if the lists they generate are traversed by folding or reduction.

Thesis Supervisor: Arvind

Title: Professor of Computer Science and Engineering

Eliminating Intermediate Lists in pH using Local Transformations

by

Jan-Willem Maessen

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 1994
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The extensive use of lists in functional programming languages exacts a cost penalty for important operations such as array construction. In addition, because lists are linear data structures, it is difficult to generate and traverse them efficiently in parallel given a purely-functional environment. Three common methods of traversal—left and right fold, and reduction—can be described in terms of `map` and `reduce` operations yielding higher-order functions; these higher-order functions can be represented entirely at compile time. This thesis examines this representation of list traversal, describes its implementation in the desugaring phase of the compiler for pH (an implicitly-parallel, mostly-functional dialect of the language Haskell). This approach to desugaring gives rise to a set of rules for compiling list comprehensions efficiently, and to a pair of functions `unfold` and `synthesize` which can be immediately eliminated if the lists they generate are traversed by folding or reduction.

Thesis Supervisor: Arvind

Title: Professor of Computer Science and Engineering

It is impossible that two things only should be joined together without a third. There must be some bond in between both to bring them together. — Plato, *Timaeus* 31 b-c

Acknowledgments

My graduate studies have been supported in large part by a National Science Foundation Graduate Fellowship.

I am indebted to many people for many things:

To Arvind, for getting the ball rolling, supervising this work, forcing me to repeatedly rewrite a proposal which deperately needed it, and providing feedback and encouragement on the final drafts.

To Lennart Augustsson, for providing the foundation upon which the pH effort is being built, and for reading and commenting on a late draft in well-nigh superhuman time.

To Shail Aditya, for equalling Lennart's time-to-read and fixing several more trouble spots as a result.

To my numerous fellow students in the MIT Computation Structures Group, who have been around at every conceivable hour of the day or night to offer advice, encouragement, and sympathy.

To my family and friends, for dealing with my rather shadowy existence over the past few months.

And finally, to Jennifer, who has played the part of Thesis Widow with sympathy and good humour, despite distressingly mismatched schedules. I love you.

Contents

List of Figures

Chapter 1

Introduction

Lists are a basic computational "glue" in many functional and almost-functional languages. They are easily expressed and understood by programmers, and offer a compact notation for representing collections of data. This allows us to reason about operations on such collections. The functional language *Haskell* [8] is no exception to this rule, and in fact encourages or even requires the use of lists to express certain sorts of computation.

1.1 The Problem

An example of the pervasiveness of lists is the canonical definition of the function `elem` in the standard prelude of Haskell:[1]

```
elem                    =  any . (==)
```

`elem` takes an object and a list, and states whether that object is contained somewhere in the given list. It is defined in terms of `any`; the code can be read as "if any are equal". The `.` operator is functional composition, and `==` is equality. Thus, by applying `==` to its first argument `elem` builds a predicate which is true for objects which are equal to that argument.[2]

To understand the definition of `elem`, one must know how to compute the function `any`. `any` takes a predicate and a list as arguments, and returns the boolean value `True` if the predicate is true of any element in the list, or `False` otherwise. Referring to the canonical definitions in the Haskell prelude once again reveals that:

```
any p                   =  or . map p
```

That is, any of the elements of a list satisfy predicate `p` if applying `p` to the elements of the list (yielding a list of boolean values) and taking the boolean `or` of the elements of the result yields the value `True`.

[1] The standard prelude in Haskell has the role of the standard libraries in most other languages—it is the collection of datatypes and functions that are available by default in Haskell programs.

[2] Leaving aside the question of what it means to be equal, a question which Haskell attempts to address cleanly through the `Eq` type class.

Now there are two as-yet-undefined functions on lists—`map` and `or`.

First there is the function `or`, which takes the boolean or of all the elements of a list. The Haskell prelude defines it as follows:

```
or                      =  foldr (||) False
```

The `foldr` basically says "starting with the rightmost end of the list and the value False, use the binary boolean or operator `||` to combine the value so far with the element of the list we are currently looking at; when the list has been completely traversed, return the value thus accumulated." Thus, the way Haskell expresses the `or` operation implies a specific way of computing the result. Several ways of expressing this computation will be examined later on.

If we like, the `map` operation can be defined in a manner similar to `or`, that is: [3]

```
map f                   =  foldr (\x soFar -> f x : soFar) []
```

"Start with the empty list, and to its front add the value of `f` applied to the elements of the given list, working from right to left." Again, a fairly operational definition. However, Haskell provides list comprehensions—a nice way of expressing functions which iterate over lists and produce a list as a result:

```
map f xs                =  [ f x | x <- xs ]
```

This says "build a list of the values of `f x`, drawing `x` from the elements of `xs`, so that the order of elements in the two lists match." Apart from the last restriction—that the order of elements in the result match the order of elements in `xs`—there is nothing stating precisely *how* the result ought to be constructed. This leaves the compiler free to choose the best possible way of doing so (naturally, we as programmers would like to be able to *trust* it to actually do so; this concern is addressed at length in this thesis).

There is syntax for expressing `map` in a manner which leaves its exact implementation to be chosen by the compiler; now, can the same thing be done with `or`? Essentially, an operator is needed to express the fact that the `or` operation can be defined by breaking the list into pieces and combining those pieces using the `||` operation, and that the identity of `||` is the value `False`. This operator is `reduce`, and is used as follows:

```
or                      =  reduce (||) False
```

For the time being, suppose that the compiler understands `reduce` as a separate operation, and will choose to perform reductions on lists as efficiently as it can.

[3] I do not make use of the standard definition of `map` in Haskell, because it expresses the computation directly by breaking apart its list argument and applying the function explicitly, like so:

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

Nonetheless, the definitions are equivalent.

The implementation of `elem` is still not terribly efficient, even given that `map` and `or` are defined efficiently. `elem` is currently defined using function composition and the list operation `any`, and `any` is in turn defined by the composition of the two list operations `or` and `map`. The definition of `any` can be simplified, thus:

```
any p                 =  (\f g x -> f (g x)) or (map p) -- in-line (.)
any p                 =  (\x -> or (map p x))           -- beta reduce
any p x               =  or (map p x)                   -- consolidate lambdas
```

The steps shown above can be performed by hand, or by a decent code inliner; a similar process may be used on `elem` itself to eliminate the composition operation. Nevertheless, `map` must still be called in order to compute `any`, and `or` must be applied to the resulting list of boolean values.

The intermediate list created in the definition of `any` is unnecessary. List construction can potentially be quite costly—a cons cell must be allocated for each element, that cell must be filled in with the element's value, the value must be fetched back from the cell, and finally the cell must be deallocated. If the definition of `any` is written more operationally—say using `foldr`—the calls to `map` and `or` can be combined, eliminating the intermediate list:

```
any p                 =  foldr (\x soFar -> (p x) || soFar) False
```

The compiler is relied upon to choose efficient ways to iterate over lists expressed using list comprehensions and `reduce`; it must be expected to eliminate unnecessary intermediate lists as well.

1.2 Parallelism

Exploiting the parallelism implicit in a program further complicates the problem. For example, the list append operation `++` can be used with `elem` to discover if `x` is an element of either of the lists `a` or `b`:

```
elem x (a ++ b)
```

That is, `x` is an element of the list `(a++b)` if it is a member of either `a` or `b`. But the following code is also correct:

```
elem x a  ||  elem x b
```

Here, the lists `a` and `b` can be traversed in parallel and the boolean or of the results can be found at the end.

What if the expression `(a ++ b)` has been uncovered by the compiler? If `any` is defined in terms of `foldr` as above, this expression would eventually result:

```
foldr (\x' soFar -> (x==x') || soFar) False (a ++ b)
```

Because computation must start from the right and proceed, accumulating results, to the left, the compiler can use the following relationship:

```
foldr f z (a ++ b)  ≡  foldr f (foldr f z b) a
```

Unfortunately, this means that the outer `foldr` (over list `a`) might not be able to be fully computed until the inner `foldr` has produced a result. Any scheme devised to handle list operations must be capable of recognizing potential parallelism, and of generating code which can exploit it.

1.3 Motivation

We are implementing a compiler for pH [10], a dialect of Haskell with eager semantics and imperative features (I-structures and M-structures) similar to the language Id [11]. In doing so, we would like to incorporate a phase in the compiler that eliminates intermediate lists, so that we can retain Haskell's compositional coding style without paying a heavy price in execution speed. Since we are targeting parallel machines, and would like to exploit the parallelism implicit in the programs we write, we would like to devise a scheme which also maximizes the amount of parallelism available. Such a scheme would have immediate benefits for list and array comprehensions, in addition to making ordinary list operations cheaper.

Id provides syntax for array comprehensions, allowing every element of an array to be computed in parallel (modulo data dependencies). These can easily be desugared into `for` loops [19]. In Haskell, such a comprehension would be expressed by applying a function, `array`, to a series of concatenated list comprehensions. This sequentializes the generation of array elements along the spine of the list, and requires an intermediate list larger than the resulting array itself! If we define `array` as a traversal over a list, our compiler should eliminate the list itself and generate the loops directly, using I-structure operations to write each element into the array at most once as in the Id desugaring and thus exposing the same amount of parallelism.

In Id, list comprehensions use open lists to construct lists starting at the head and working toward the tail. The head of a list can thus be made available before the remainder of the list has been fully constructed. In addition, if sublists are being created in nested loops, each sublist can be generated independently, and a single I-structure operation can be used to join pairs of them together. This is a form of producer/consumer parallelism, and allows parts of a list that have been fully used to be discarded before the remainder of the list even exists. A general traversal mechanism ought to generate lists using open lists if it is beneficial to do so.

Finally, it is impossible to eliminate intermediate lists without providing a systematic way of constructing them. The approach chosen must integrate nicely with the traversal methods available, so that a traversal over a systematically generated list will become a simple loop. Haskell has functions which return lists of indices and special notation for generating lists of numbers; both of these will immediately benefit from being integrated into the desugarer through a more general mechanism.

1.4 The Thesis

By expressing list traversal in terms of `map` and `reduce` over higher-order functions a large body of different list traversals can be captured, including `foldl`, `foldr`, and `for` loops. This representation concisely captures the dependencies between adjacent elements, and thus exposes the parallelism that exists in traversals which use `reduce` while retaining the dependencies inherent in traversals with `foldl` and `foldr`. Moreover, the higher-order functions required to express traversals in this manner are entirely compile-time artifacts; by representing them only at compile time, and by using techniques such as η-abstraction, they can be eliminated from generated code. At the same time, lists can be constructed using the functions `synthesize` (which mirrors `reduce`) and `unfold` (whose operation is similar in spirit to the fold operations). Both of these functions combine nicely with reduction to produce iterative loops or recursive functions—eliminating the need for lists.

A simple set of desugaring rules will express list comprehensions in terms of `map` and `reduce`. Since reduction behaves in well-defined ways when applied to comprehensions, all the special syntax provided for lists in the Haskell and pH compilers is automatically open to optimization and parallelization. The optimization of list traversal can therefore be combined with desugaring. This phase can still be considered "desugaring", since the transformations it performs are local—that is, nested folds are combined together into a single fold. Doing so does not interfere with other aspects of desugaring, such as pattern matching compilation. There is a simple set of heuristics which can be used to improve the resulting code, by choosing the traversal direction to generate an iterative function, by minimizing the number of list elements examined before a result is obtained, and by eliminating constant arguments from loops and recursive function calls.

While the desugarer can produce efficient code for a broad class of functions, it requires a certain degree of intelligence on the part of the programmer to realize the best performance. It is therefore important to characterize exactly what sorts of program are amenable to optimization, and what the effects of that optimization will be. For example, certain useful data structures can be expressed as list traversals in the framework provided by the desugarer. This allows us to optimize the construction of arrays using I-structures, and replace ordinary lists with open lists.

The existing implementation of the desugarer using these transformations produces measurable improvements in benchmark programs where certain function definitions have been expanded in-line; this shows promise for future implementations, and suggests that programming in an applicative style is worthwhile. Nonetheless, though the desugarer as it stands will process the pH language correctly and efficiently, there are a number of improvements neccessary to realize the full power of these techniques.

Chapter 2

Describing List Traversals

This chapter describes three major methods of traversing lists—`foldr`, `foldl`, and `map/reduce`. All three methods of traversal are very closely related, but each captures properties that are useful in different sets of circumstances. The need to exploit parallelism, combined with a slightly more general notion of "traversal", leads to the choice of `map` and `reduce` to represent list traversals in the pH compiler.

2.1 `foldl` and `foldr`

The most lucid definitions of the two folds—`foldl` and `foldr`—are mathematical rather than operational:

$$\begin{aligned} \texttt{foldr} \ \oplus z \ [x_1, x_2, \ldots, x_n] &= x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus z))) \\ \texttt{foldl} \ \oplus z \ [x_1, x_2, \ldots, x_n] &= (((z \oplus x_1) \oplus x_2) \oplus \cdots) \oplus x_n \end{aligned}$$

The parenthesization in the above definitions show that `foldr` traverses a list "right to left", while `foldl` traverses the list "left to right."

Using these definitions, some useful functions can be defined; for example `sum` can be defined as either of the following:

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

Notice that the $\oplus$ operation forms a spine which combines the "result so far" (initially z) with the value of the requisite list element (x_i). `foldl` goes "with the pointers"—data flows from the head of the list to the tail, and the result is immediately available when the end of the list is reached. It thus seems natural that the Haskell code for `foldl` is tail recursive (iterative):

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

On the other hand, in `foldr` data goes "against the pointers"—flowing from the tail of the list towards the head. Generally, the result will not be fully computed until the list is traversed all the way to

its end—the actual computation takes place (conceptually) on the way back. This is, of course, the description of a recursive function:

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

Assuming iteration will produce better code than recursion[1] favors `foldl` for expressing most kinds of list traversals. And, indeed, the pH compiler will generate iterative code in favor of recursive code.

A different, potentially more revealing view of the fold operations is used in [6]. By writing $\oplus$ wherever the code to generate our list used the cons operation (`:`), and replacing the tail `[]` of the list by z, the mathematical definition of `foldr` given above is obtained. For example, if `sum xs = foldr (+) 0 xs`, then the following equivalence is a matter of straightforward substitution:

$$\texttt{sum (3:(1:(4:(1:(5:(9:[])))))) } \equiv 3+(1+(4+(1+(5+(9+0)))))$$

Note that the parentheses on the left side are actually superfluous, since the cons operator is right associative. `foldl` operates the same way on a *reversed* view of the list as `foldr` does on the list itself. Wadler, in [16], imagines that the list is actually being constructed in the opposite direction, using a "snoc" (backwards cons) operation, and that `foldl` replaces "snoc" and "lin" constructors. Conceptually, the technique of replacing cons and nil has been known for quite some time [17, 13, 6]. It provides a springboard for the transformations presented in this thesis.

2.1.1 Unifying `foldl` and `foldr`

Given that `foldl` is often a more efficient way of traversing lists, but that `foldr` cancels easily, the obvious unifying step is to define one in terms of the other. Indeed, this can be done in either direction (`foldl` in terms of `foldr` is shown in [6], and the other case is analogous):

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

These definitions are somewhat difficult to grasp—examine what happens when `foldl` is defined in terms of `foldr`. The flow of data through the recursion must be turned around. The computation at each list element requires two pieces of data—first the value of that element, and second the accumulated result flowing in from the computation to its left. Thus, instead of getting a result *from* the left, `foldr` will build a function which can be passed as data *to* the left. Thus, the function being folded, `(\b g a -> g (f a b))`, packages up `b`, the list element, and `g`, the computation for the piece of the list to the right. It returns a function with a single argument, `a`, which performs the computation for this element `(f a b)` and passes the resulting value to the right by calling `g`. When the list is traversed, the resulting accumulated value is returned, so the rightmost (initial) function must be the identity function.

[1] Not true in most lazy languages! In addition, lazy languages permit the use of inifnite lists, for which `foldl` will never terminate. For more on wh to choose `foldr` over `foldl`, refer to the excellent discussion on the topic in [3].

Essentially the whole complicated process amounts to "wiring together" the calls to `f` so that data flows from left to right as in `foldl`, but actually performing the wiring starting from the rightmost end of the list.

2.1.2 `foldr` is universal

Since `foldl` and `foldr` can be defined in terms of each other, choosing one ought to allow *every* fold over a list to be represented. To do so effectively, however, there must be a guarantee that the resulting code is at least as efficient as the original folding code. Examine what happens when the original definition of `foldr` is inlined in the higher-order definition of `foldl`:

```
foldl f z xs   =
  let foldr' []       =  (\a -> a)
      foldr' (x:xs)   =  (\b g a -> g (f a b)) x (foldr' xs)
  in foldr' xs z
```

The first two arguments of foldr, which are invariant during the recursion, have been eliminated to obtain `foldr'`. β-reducing the second disjunct of the definition yields:

```
foldl f z xs   =
  let foldr' []       =  (\a -> a)
      foldr' (b:xs)   =  (\a -> foldr' xs (f a b))
  in foldr' xs z
```

Finally, observe that the values of the two disjuncts above are both lambda expressions with one argument. Moreover, the only applications of `foldr'` provide two arguments, rather than one! Thus, adding an argument to `foldr'` (η-abstracting it) produces the following:

```
foldl f z xs   =
  let foldr' []     a  =  a
      foldr' (b:xs) a  =  foldr' xs (f a b)
  in foldr' xs z
```

Note that this is virtually identical to the original definition of `foldl`—in particular, the inner procedure `foldr'` is tail-recursive, so this definition can execute iteratively. Thus, using `foldr` to represent folds in *either* direction is perfectly acceptable.

This process of converting `foldl` to `foldr` and then optimizing is best demonstrated with an example. `implodeNum` takes a list of digits (base ten) and computes the corresponding integer—that is, the list `[1,2,3,4,5]` would convert to the integer `12345`. `implodeNum` is simple to define using `foldl`:

```
implodeNum ds   = foldl (\num digit -> num*10 + digit) 0 ds
```

Transforming `foldl` to `foldr` and simplifying somewhat results in:

```
implodeNum ds   = foldr (\ b g a -> g (a*10 + b)) (\a -> a) ds 0
```

The call to `foldr` can be expanded to generate an explicit recursive function:

```
implodeNum ds   =
  let foldr' []         = (\a -> a)
      foldr' (b:ds)     = (\a -> foldr' xs (a*10 + b))
  in foldr' ds 0
```

Finally, the η-abstraction step yields an iterative function:

```
implodeNum ds   =
  let foldr' []     a   = a
      foldr' (b:ds) a   = foldr' xs (a*10 + b)
  in foldr' ds 0
```

On the other hand, `foldr` doesn't convert terribly nicely when expressed in terms of `foldl`:

```
foldr f z xs   =
  let foldl' z []      =  z
      foldl' z (x:xs)  =  foldl' ((\ g a b -> g (f a b)) z x) xs
  in foldl' (\b -> b) xs z
```

Neither of the disjuncts in the definition of `foldl'` is a lambda expression; the first disjunct returns a previously computed function, and the second requires a recursive call to compute the function to be applied. Thus, attempting to η-abstract `foldl'` will not eliminate the higher-order functions which the definition of `foldr` introduced. This result is unsurprising, since `foldr` parallels the recursive structure of the list itself, while `foldl` inverts that structure.

2.1.3 Folds and `for` loops

An additional advantage to viewing all folds in terms of `foldr` is evident when defining a `for` loop as a fold. Assume the loop is expressed as in Id[11], as a series of bindings which includes free references to a set of k *circulating variables* $c_1, c_2, \ldots c_k$ and which binds a corresponding set of *nextified variables*, $n_1, n_2, \ldots n_k$.[2] A `for` loop over the list `xs` is thus written:

```
let
  initial bindings for c1, c2, ... ck
in
 for x <- xs do
   bindings, including ones for n1, n2, ... nk
 finally f c1, c2, ... ck
```

For example, to find the arithmetic mean of a list of numbers, the following function can be used:

```
mean xs =
  let sum = 0
      len = 0
  in for x <- xs do
```

[2]Note that in Id, n_i is written as `next` c_i. This renders the rules virtually unreadable; thus the choice of notation here.

```
        next sum = sum + x
        next len = len + 1
    in sum / len
```

Traditionally, `foldl` and tupling are used when explaining `for` in terms of folds over a list:

```
let
  initial bindings for c1, c2, ... ck
in
 ((\(c1, c2, ... ck) -> f c1, c2, ... ck)
     (foldl (\(c1, c2, ... ck) x ->
               let
                 body bindings as above, including n1, n2, ... nk
               in (n1, n2, ... nk))
            (c1, c2, ... ck)
            xs))
```

This produces the following translation for `mean`. Notice how a new tuple is laboriously contructed and taken apart again at each iteration:

```
mean xs =
  let sum = 0
      len = 0
  in ((\(sum, len) -> sum / len)
        foldl  (\(sum, len) x ->
                    let next sum = sum + x
                        next len = len + 1
                    in (sum, prod))
               (sum, len)
               xs)
```

The most obvious way to express `for` in terms of `foldr` is to translate the above definition using the definition of `foldl` in terms of `foldr`. Instead, note that the argument which is η-abstracted in the earlier definition represents the circulating value in the iteration over the list. Thus, extra arguments can simply be added to the functions:

```
let
  initial bindings for c1, c2, ... ck
in
 foldr (\x cont c1 c2 ... ck ->
           let
             body bindings as above, including n1, n2, ... nk
           in cont n1 n2 ... nk)
       f
       xs
       c1 c2 ... ck
```

This definition completely eliminates the tupling and detupling operations which occur throughout the first definition. In addition, using inlining and η-abstracting all arguments which correspond to circulating variables, once again produces a completely iterative expression of the loop. The function being

passed backward to construct the iteration is effectively the continuation of the loop; thus this definition uses continuation-passing style to capture the passing of multiple values between loop iterations. This leads to a more concise compiled definition for `mean`:

```
mean xs =
  let sum = 0
      len = 0
  in foldr (\x cont sum len ->
                let next sum = sum + x
                    next len = len + 1
                in cont sum prod)
           (\sum len -> sum / len)
           xs sum len
```

2.1.4 Bidirectional traversal and `foldr`

Having shown that `foldr` can easily express left folds and loops, an obvious question is whether it can embody bidirectional traversal—that is, one where the information passed to the right must initially be computed; after new results are obtained computation must occur to derive the information that is returned to the left. This can be expressed generally with the following function:

```
leftright right left rightmost leftmost xs =
  let lr fromleft xs =
        case xs of
          []     -> rightmost fromleft
          (x:xs) -> right x (lr (left fromleft x) xs)
  in lr leftmost xs
```

Here `left` and `right` correspond to the functions that are folded as in `foldl` and `foldr`. Similarly, `leftmost` describes the initial value which is passed to the right. `rightmost` is a function which *yields* the initial value to be passed from right to left in terms of the final value passed from left to right. Thus, the value passed right is "transformed" to begin the return trip to the left.

The continuation-passing definition of `leftright` in terms of `foldr` is routine:

```
leftright right left rightmost leftmost xs =
  foldr (\x cont fromleft -> right x (cont (left fromleft x)))
        rightmost
        xs
        leftmost
```

This definition has the same nice properties as the previous definitions which used `foldr`—inlining and η-reduction produce the original definition, and replacing `fromleft` with a series of variables allows multiple values to be passed from left to right without using tuples, just as was done in the `for` loop translation.

2.2 map and reduce

Given the relationships between the fold operations and the various ways in which a list can be traversed, one can simply proclaim that "`foldr` is the universal fold," and design desugaring and optimization phases accordingly. This is the approach taken in [6] to eliminate repeated traversal and construction of intermediate lists, and it is assumed for the case of lists by most papers which address the subject (including [13, 9, 3]). pH is intended as an implicitly parallel programming language, and the linear structure of lists makes them unappealing. Nevertheless, it is always beneficial to eliminate intermediate list construction, since this will reduce the number of places in the program where such sequentiality can be introduced. Choosing the right list representation, for example an *open list* [7], can help to further exploit parallelism (this will be discussed in greater depth in Section 4.6). However, the fact remains that fold operations impose a particular ordering on the flow of data in their computations, and that this ordering can create bottlenecks where computations are left waiting for the data being passed along the "spine" of the fold. The bottleneck is best expressed algebraically by the following identities:

```
foldr f z (a ++ b)     =     foldr f (foldr f z b) a
foldl f z (a ++ b)     =     foldl f (foldl f z a) b
```

These identities state that if the list folded over can be separated into two pieces, `a` and `b`, then in order to fold over the list, simply fold over the two pieces, feeding the result of the first fold into the spine of the second. These results are clear from the original mathematical definitions of the folds:

$$\begin{aligned}
&\texttt{foldr } \oplus\ z\ \texttt{[}a_1, a_2, \ldots, a_n\texttt{] ++ [}b_1, b_2, \ldots, b_n\texttt{]} \\
&= a_1 \oplus (a_2 \oplus (\cdots a_n \oplus (b_1 \oplus (b_2 \oplus (\cdots b_n \oplus z))))) \\
&= \texttt{foldr } \oplus\ \texttt{(foldr } \oplus\ z\ \texttt{[}b_1, b_2, \ldots, b_n\texttt{])\ [}a_1, a_2, \ldots, a_n\texttt{]}
\end{aligned}$$

$$\begin{aligned}
&\texttt{foldl } \oplus\ z\ \texttt{[}a_1, a_2, \ldots, a_n\texttt{] ++ [}b_1, b_2, \ldots, b_n\texttt{]} \\
&= (((((z \oplus a_1) \oplus a_2) \oplus \cdots a_n) \oplus b_1) \oplus b_2) \oplus \cdots b_n \\
&= \texttt{foldl } \oplus\ \texttt{(foldl } \oplus\ z\ \texttt{[}a_1, a_2, \ldots, a_n\texttt{])\ [}b_1, b_2, \ldots, b_n\texttt{]}
\end{aligned}$$

Neither of these definitions captures the fact that (for example) `sum (a++b)` is equivalent to `sum a+sum b`. The solution is to describe the traversal of lists symmetrically—that is, by saying (the backquoted operator `` `op` `` is infix):

```
traversal (a ++ b)     =     traversal a `op` traversal b
```

Bird describes a general version of such an operator, called `reduce`, in [1, 2]. A partial mathematical definition of `reduce` is very similar indeed to the definitions of `foldl` and `foldr`:

$$\begin{aligned}
&\texttt{reduce } \oplus\ z\ \texttt{[}x_1, x_2, \ldots, x_n\texttt{]} = x_1 \oplus x_2 \oplus \cdots x_n \\
&\texttt{reduce } \oplus\ z\ \texttt{[]} = z
\end{aligned}$$

The obvious differences in the definition are the absence of parentheses and the fact that z is only used to define the value of `reduce` on the empty list. $\oplus$ and z must satisfy several properties which will allow the exact operation of `reduce` (for any value of a, b, or c of the appropriate type) to be characterized:

$$a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c$$
$$a \oplus z \equiv z \oplus a \equiv a$$

That is, $\oplus$ is associative (thus the absence of parentheses in the above definition), and z is the left and right identity for $\oplus$. This leads to a more rigorous (though still mathematical) definition of `reduce`:

```
reduce f z (a ++ b)     =     (reduce f z a) `f` (reduce f z b)
reduce f z [e]          =     e
reduce f z []           =     z
```

The first line of the definition restates the original goal. Thus, `reduce` is a way of traversing a list by breaking it into two arbitrary sublists and reducing them in parallel.

The structure of `reduce` suggests a third view of lists which parallels the "cons" and "snoc" views reflected by `foldr` and `foldl`. In this view, every list is either empty, of unit size, or is obtained by appending two arbitrary lists. This view is codified by the list monoid, which will be discussed in Section 2.2.3.

2.2.1 How `reduce` works

Defining what is meant *operationally* by `reduce` is more difficult. Bird notes that either of the following definitions is perfectly valid, since `z` is the identity of `f`:

```
reduce f z     =     foldl f z
reduce f z     =     foldr f z
```

Because these definitions simply re-introduce the sequential fold operations, they are quite unsatisfactory. Instead,the precise definition of `reduce` is left up to the compiler. In this way, the compiler can choose different implementations of `reduce` based on context. Indeed, if a call to `reduce` is inlined in several different places, quite different implementations of the reduction can be chosen in each case.

2.2.2 What `reduce` cannot do

Above it was shown that `foldr` can be used as a universal fold operation. Since `foldr` doesn't seem to be the right way to express parallel list traversals, however, it would be nice to express fold-like operations using just the `reduce` operation. For example, the `length` operation ought to be expressible using a `reduce`-like scheme:

```
length (a ++ b)     =     (length a) + (length b)
```

The problem is obvious when the types of the functions are examined:

```
reduce :: (a -> a -> a) -> a -> [a] -> a
foldl  :: (b -> a -> b) -> b -> [a] -> b
foldr  :: (a -> b -> b) -> b -> [a] -> b
length :: [a] -> Int
```

`reduce` requires that the list it traverses have elements with the same type as the type of the value it returns. What is needed is some way of transforming individual elements of the list into objects of the correct type, which can then be combined using `reduce`. The most natural way of accomplishing this is with list comprehensions; this is how `map` was defined in the introduction.

However, the duty of the desugarer is to *eliminate* list comprehensions, since they simply represent a clean syntax for a class of list traversals which yield lists as results. Instead, a simple operation can be chosen which will transform individual elements of a list regardless of context. `map` itself is an ideal candidate! (In Chapter 4 a translation between list comprehensions and `map/reduce` will be defined.) `map` transforms every element of the list separately. It is easy to define `length` in terms of `map` and `reduce`:

```
length          =       reduce (+) 0 . map (\_->1)
```

The length of each element of the list is one; the number of list elements is simply summed. A definition of `length` using `foldl`would have been:

```
length          =       foldl (\l _ -> l + 1) 0
```

The `map/reduce` definition simply separates this definition into two pieces—the addition (a reduction) and the part of the function which says "ignore the element's value and return 1" (a map).

2.2.3 Expressing monoids using lists

Bird [2] notes that `map` and `reduce` used in this way express the homomorphism between an arbitrary monoid with an identity over some type α, and a list of element type α. In effect, this means that any type which is expressible as a monoid can be written in terms of lists and list comprehensions, which are then mapped over and reduced. The compiler exploits this structure to optimize the construction of such objects. A monoid $(\beta, \oplus, id_{\oplus}, i)$ over some type α is a mathematical structure consisting of a type (β), an associative binary operation on that type $(\oplus)$, and i, a *unit* function from α to the monoid type. The monoids of interest are those whose operation has an identity, $id_{\oplus}$ (the associativity of $\oplus$ means that this will be both a right and a left identity). Note that the definition of $\oplus$and $id_{\oplus}$ correspond nicely to the requirements for the arguments $\oplus$and z of the reduce function from the beginning of the section. Types which can be expressed nicely by monoids include the integers (with addition and zero, or multiplication and one), bags, sets, sorted lists, open lists, and arrays. Some of these data structures will be examined in Section 5.4. For our purposes, however, the most important monoid is the list monoid over an arbitrary type α, given by (`[`α`]`,`++`,`[]`,(`$\lambda$`x.(x:[])`).

A homomorphism between two monoids over the same α must insure that the resepctive operations behave in equivalent ways (here for the homomorphism h from the monoid $(\beta, \oplus, id_{\oplus}, i)$ to $(\gamma, \otimes, id_{\otimes}, j)$, taken from [2]):

$$
\begin{aligned}
h\, id_{\oplus} &= id_{\otimes} \\
h\,(i\ a) &= j\ a \\
h\,(x \oplus y) &= (h\ x) \otimes (h\ y)
\end{aligned}
$$

Thus, the homomorphism must translate one identity into the other, and that using the homomorphism on the result of a binary operation $\oplus$ is equivalent to using it on both arguments and then applying the binary operator of the second monoid $\otimes$. The interesting result is that `map` and `reduce` together form the homomorphism from the monoid of lists over α to an arbitrary monoid $(\beta, \oplus, id_{\oplus}, i)$:

$$h\ l = \texttt{reduce}\ \oplus\ id_{\oplus}(\texttt{map}\ i\ l)$$

This structure will prove very powerful for describing a whole class of list traversals.

2.2.4 Expressing folds using `map` and `reduce`

In unifying `foldl` and `foldr`, the computation on an individual element of a list was "packaged up" by a higher-order function; the list traversal then "wired together" these higher-order functions in the correct direction. Conceptually, exactly the same thing can be done to express the folds in terms of `map` and `reduce`. Here, the "packaging up" stage is expressed as a `map` and the "wiring together" stage as a `reduce`:

```
foldl f z xs      =     reduce (\ l r -> r . l) (\x -> x)
                              (map (\e t -> f t e) xs) z

foldr f z xs      =     reduce (\ l r -> l . r) (\x -> x)
                              (map (\e t -> f e t) xs) z
```

In both cases, the function being mapped packages up a list element and produces a function which is waiting for the value arriving along the spine of the traversal. Thus, the reduction takes and produces functions waiting for values. In order to do so, it simply composes the functions in the appropriate order—when we're folding left, we apply the right-hand function to the result of the left-hand function, and vice versa for `foldr`. In either case, the identity function is the identity of the reduction, since it is the left and right identity of composition. The identity function says "for an empty sublist, simply return the same value as was passed in." The final argument, `z`, is the input to the function which is built—that is, the initial value of the fold. For example, here is another formulation of the `implodeNum` function (originally defined using `foldl`):

```
implodeNum ds     =     reduce (\ l r -> r . l) (\x -> x)
                              (map (\e t -> 10*t + e) ds) 0
```

In this framework, `for` loops make use of the same continuation-passing transform as before, turning them into right folds, and the above translation for `foldr` yields and equivalent homomorphism

(the continuation-passing trick is necessary if tupling or multiple value return are to be avoided). This will require more arguments (including the continuation) in place of `t` in the function being mapped, but `reduce` still uses function composition to build up the loop. Similarly, the continuation-like transformation presented earlier and the above definition of `foldr` can be used to define `leftright` as a homomorphism.

This new formulation once again begs the question of efficiency. A good inlining of these functions ought to exist. Using the fact that `reduce` can be defined as `foldr` the latest definition of `foldl` can be rewritten:

```
foldl f z xs        =     foldr  (\ l r -> r . l) (\x -> x)
                                 (map (\e t -> f t e) xs) z
```

The following equivalence can be used to simplify the above expression:

$$(\texttt{foldr f z}) \ . \ (\texttt{map g}) \equiv \texttt{foldr (f . g) z}$$

Yielding the following definition:

```
foldl f z xs        =     foldr ((\ l r -> r . l).(\e t -> f t e))
                                (\x -> x) xs z
```

Inlining the two occurrences of composition, performing a number of β-reductions, and consolidating the resulting arguments results in:

```
foldl f z xs        =     foldr (\e r t -> r (f t e)) (\x -> x) xs z
```

This is identical to the original definition of `foldl` in terms of `foldr`! Thus, for example, the above rewritings recover the definition of `implodeNum` (modulo renaming) in terms of `foldr` from the above definition of `implodeNum` in terms of `reduce`:

```
implodeNum ds       =     foldr (\e r t -> r (10*t + e)) (\x->x) ds 0
```

Inlining and reducing the definition of `foldr` in terms of `reduce` (in an attempt to recover the definition of `foldr` once again) doesn't quite produce an equivalent definition:

```
foldr f z xs        =     foldr (\e r t -> f e (r t)) (\x -> x) xs z
```

Nonetheless, like all the previous definitions which used `foldr` and higher-order functions, this one will inline and η-reduce nicely:

```
foldr f z xs        =      foldr' xs z
                      where foldr' []     r = r
                            foldr' (x:xs) r = f x (foldr' xs r)
```

This definition is almost equivalent to the original definition of `foldr`. The difference is that the argument `r` to `foldr'` is always `z`; exploiting this fact eliminates `r` and replaces the only remaining occurrence of `r`

(in the first disjunct) with `z`. The difference is minor, and possibly irrelevant in a compiler which performs lambda-lifting—as a free variable of `foldr'`, `z` would be lifted away and turned into an argument.

For the sake of symmetry, `reduce` itself can be defined so that it has the same identity as `foldl` and `foldr`:

```
reduce f z xs       =       reduce (\l r z -> (l z) `f` (r z)) (\x -> x)
                                   (map (\e z -> e) xs) z
```

This definition essentially "scatters" the value of z across the reduction, where it is generally simply dropped. Only when an empty list is encountered is this extra argument actually put to use.

This approach is clean—a list traversal is represented as a monoid with the identity function as its identity, and the homomorphism of the monoid is then applied to a single argument to start the actual computation. The desugarer uses the above definitions to describe all list traversals, yielding a canonical list traversal of the following form:

`(reduce` a `(\x->x) (map` u `l))` t

Here a is the higher-order monoid operation, and consequently is a function of at least three arguments; u is the higher-order unit operation, with at least two arguments; t represents either the initial value of a fold or the identity of an operation being reduced. For example, here are the values of a, u, and t for `foldr f z`:

a = `(\l r -> r . l)`
u = `(\e t -> f t e)`
t = `z`

This choice of canonical form raises a number of issues, however. The most critical is parallelism. The approach outlined above seems to advocate turning occurrences of `reduce` into loops—or worse, recursive function calls—despite the fact that the reductions were originally introduced to *eliminate* such sequential behavior unless it is needed.

Several steps must be taken to make the approach practical. First, the desugaring rules must exploit the parallelism of a reduction over a list with well-understood structure. Remaining reductions should generate iterative loops if doing so is efficient (in the absence of special provisions for reduction in the target language). This requires some degree of cleverness in constructing the desugarer.

Chapter 3

Describing the generation of lists

To actually *eliminate* intermediate lists, and not simply optimize their traversal, the generation of the lists must be described as systematically as the consumption. List comprehensions are a partial solution, since they describe the generation of new lists in terms of traversals over existing lists. Nevertheless, there are many cases where lists are generated without reference to other lists—for example, Haskell contains special syntax for arithmetic series expressed as lists written as $[k_1, k_2 .. k_n]$. In such cases, there needs to be a systematic way to generate a list from the initial data.

3.1 Overview of list generation techniques

The chief problem with list generation is that lists are a full-fledged data structure in the language. This means that in the process of constructing a list, the partially-constructed result (the "accumulative result variable" of [13]) can be traversed at any time. This precludes eliminating the list, since it must either be built once so that the repeated traversals will behave as they were written, or be re-constructed every time it is re-traversed. In the first case, nothing is gained, and in the second case there is a risk that reconstruction and re-traversal could proceed *ad infinitum.*

Much of the literature on the elimination of intermediate data structures has focused on ways of preventing this problem from occurring. In order to do so without adding language constructs, Wadler's deforestation algorithm [17] simply legislates against it, requiring the program to be in so-called *treeless form.* Chin [5] describes a conservative method of figuring out when arbitrary expressions can be subject to deforestation. In both cases, higher-order functions must either be inlined or left unoptimized. Proposed solutions in this vein all require intensive, non-local analysis of the program, and are thus ill-suited for the desugarer.

Another alternative is to define special functions which allow a list to be generated without providing access to the accumulated result. Various variations on this approach have been tried. The short cut to deforestation [6] focuses on an operator called `build`, which takes a function as an argument. That function (the built function) is called with the constructors cons and nil as arguments. Thus, to generate

the list of integers from one to ten (written in Haskell as `[1..10]`) one can write:

```
build (\cons nil -> let loop 0     l = l
                        loop (n+1) l = loop n (cons (n+1) l)
                    in loop 10 nil)
```

An application of `foldr` to `build` simply calls the built function, passing it the function being folded with and the initial value for the fold in place of cons and nil. The parameters to the built function effectively become the constructors of an abstract, list-like datatype, and they can be used exactly as cons and nil are used in everyday code. The difference, of course, is that the nice syntax for lists is lost (or must be handled by an inner fold), and the function's result, because it does not explicitly create a data structure of some particular type, cannot be traversed. Also, the actual `build` operator itself is not Hindley-Milner typable, though making it into a "black box" circumvents this difficulty.

A similar approach is to actually use an abstract datatype to construct a list, and call a coercion function to convert the fully-constructed abstract object into an ordinary list again. This introduces a systematic set of names into the translation. Scoping problems make it hard to use for local optimizations, since it becomes difficult to characterize what it means for a function to return an object of type "untraversable list" unless it is immediately coerced to an ordinary list.

All of these approaches were explored in an effort to find the best way to express parallel list generation so that it can be combined with mapping and reduction. The complexity of "blazing" a program ruled out approaches based on deforestation. A `build`-like operation was implemented, and worked to some extent; however, the structure of the desugarer is such that the arguments of the built function must be bound after the function body itself has already been optimized. This eliminated a number of useful transformations which would otherwise have been performed, most notably the immediate elimination of copy operations (traversals which do nothing, a frequent and ordinarily easily spotted problem). This led to the introduction of abstract datatypes for untraversable lists, which were ruled out because of the scoping problems mentioned above.

Instead, functions can be defined which generate lists systematically. For example, Bird and Wadler [3] describe a function called `iterate` which takes an initial element and a generator and produces an infinite list one element at a time. Each succeeding element is produced by applying the generator to the previous element. Thus, to generate the list of integers from one to infinity one writes `iterate (1+) 1`. A more general operation, the list anamorphism [9], includes a predicate to terminate generation, so that finite lists can be created systematically, and separates the element value which is generated from the data which is used to generate succeeding elements. This approach is explored in more detail in the remainder of the chapter.

3.2 unfold and synthesize

The programmer uses two functions, **unfold** and **synthesize**, to produce lists. The former has the following declaration (where a vertical bar | denotes a conditional disjunct):

```
unfold p f v = h v
         where h v | p v       = []
                   | otherwise = a:h b
                       where (a,b) = f v
```

unfold is not the function of the same name described by Bird and Wadler[3], but rather the closely related list anamorphism [9]. Its structure is very similar to the structure of both **foldl** and **foldr**. A value **v** is propagated from left to right along the spine of the list as it is created. When predicate **p** is true of the value, the end of the list has been reached; otherwise, the function **f** is used to generate a new list element and a fresh value for propagation. The list is created recursively rather than iteratively—nevertheless, the elements at the head of the list are conceptually computed before the elements which succeed them. The following transformation can then be used to combine **map**, **reduce** and **unfold** (note that the **map** and **reduce** are assumed to be in the canonical form used above):

```
(reduce   a  (\x->x) (map   u   l)) t
≡
h i z
  where h v r | p v       = r
              | otherwise = f (g a) (h b) r
                    where (a,b) = j v
```

Note that generating this code directly will yield a fully iterative function if the reduction was generated by a call to **foldl**; however, the parameter **r** will be constant when the reduction is created by a call to **foldr** or **reduce**. In both cases, however, a better function can be generated using heuristics similar to those used to generate list traversals (see 4.3.2).

The list of integers from one to ten is simply given by `unfold (>10) (\e->(e,(1+e))) 1`. Consuming this list using **foldl** yields:

```
foldl f z (unfold (>10) (\e->(e,(1+e))) 1)
≡
h 1 z
  where h v r | v>10      = r
              | otherwise = h (v+1) (f v r)
```

This is completely iterative. If **foldr** is used instead, the following equivalence results:

```
foldr f z (unfold (>10) (\e->(e,(1+e))) 1)
≡
h 1 z
  where h v r | v>10      = r
              | otherwise = f v (h (v+1) r)
```

Finally, if map and reduce are used on the list, the following code is generated:

```
reduce f z (map g (unfold (>10) (\e->(e,(1+e))) 1))
≡
h 1 z
  where h v r | v>10      = r
              | otherwise = f (g v) (h (v+1) r)
```

Thus, the latter two cases produce (by default) code which is recursive, in which the r argument to h is constant. Note that all of the above results are very similar to the code generated for simple list traversals. Indeed, list traversal can be expressed as an identity operation:

```
traverse xs     = unfold (\l -> case l of
                                  []->True
                                  _ ->False)
                         (\(x:xs) -> (x,xs))
                         xs
```

The similarity to list traversals suggests that unfold expresses the *sequential* generation of lists.

The diligent reader will note that unfold makes use of tupling, a practice which was shunned in the transformations for loops. There are several reasons for this. First, the number of values being returned (two) is fixed for all time. Second, tuples are not passed between recursive calls, so inlining the generation function will eliminate the tuple entirely. This happened in the example above. The most important reason for retaining tuples, however, is that they make unfold type safe. unfold is well typed as defined above:

```
unfold :: (a -> Bool) -> (a -> (b,a)) -> a -> [b]
```

To eliminate tupling, the second argument would need to be converted to continuation-passing style. This would yield the following type:

```
unfold :: (a -> Bool) -> (a -> (b->a -> [b]) -> [b]) -> a -> [b]
```

This would allow definitions which ignore their continuations such as the following:

```
unfold (\_ -> False) (\_ cont -> [1..10]) 1
```

local quantification is required in the type system in order to define a safe version of unfold similar to the above. The type of continuation-passing unfold is directly analogous to the type of build [6].

While unfold reflects the operation of foldl and foldr, the function synthesize mirrors the reduce operation:

```
synthesize p e f v = h v
     where h v | p v       = if e v then [] else [v]
               | otherwise = h l ++ h r
                    where (l,r) = f v
```

Here the task of generating a list is split up into two subtasks—generating the left and right chunk of the list. Function `f` generates two values which are used to generate each piece. Predicate `p` indicates whether `v` can be further split; if it can (`p v` is false), `h` is called recursively on each value returned by `f`. The results are then concatenated together. If `p v` is true, then `e` indicates whether, given `v`, the resulting sublist should be empty or should have unit length.

Like `unfold`, `synthesize` can be combined with the canonical `map` and `reduce` functions produced by the desugarer:

```
(reduce   a  (\x->x) (map   u   l)) t
≡
h i z
  where h v t | p v       = if n v then t else u v t
              | otherwise = a (h l) (h r) t
                    where (l,r) = j v
```

This inlining produces good code. Nonetheless, if `reduce` is called on the value of `synthesize`, `r` will be constant; its argument can be eliminated by a simple heuristic. In any case, `synthesize` is intended expressly to capture the notion of generating lists in parallel.

Either `unfold` or `synthesize` can be used to define versions of Haskell's arithmetic series (these definitions work for nondecreasing series):

```
[k1,k2..kn] = unfold (>kn) (\k->(k,k+(k2 − k1)) k1
            = map Fst (synthesize (\(l,u)->l==u) (\_->True)
                                  (\(l,u)->let mid = (l+u) `quot` 2
                                               lmid = mid - mid `mod` (k2 − k1)
                                               umid = mid + (k2 − k1)
                                           in ((l,lmid),(umid,u))) (k1,kn)
```

The definition using `unfold` actually generates its result arithmetically from left to right, as is customarily done. The second definition, using `synthesize`, breaks the series in half and generates each half recursively.

By marking the spine of the list being unfolded, any `unfold` operation can be expressed as a call to `synthesize`:

```
unfold p f v
≡
synthesize (\(e,v) -> e || p v) (\(e,v) -> p v)
           (\(False,v) -> case f v of (e,v') -> ((True,e),(False,v')))
           (False,v)
```

This construction passes tuples as arguments to recursive calls, however, and is therefore to be avoided. More work is necessary to coherently unite `unfold` and `synthesize`; see Section 7.1.6.

Chapter 4

The map/reduce desugarer

In order to take advantage of the optimizations described in the previous chapters, the program must be translated into a form which uses optimizable functions optimized. Library functions must simply be written using folds, reductions, comprehensions, syntheses, and unfoldings. This library code should then be in-lined wherever the library functions are called, permitting them to be desugared. Ideally, user code should also be inlinable; the possible consequences of automating this process are discussed in Section 7.1.1.

As a first step to optimizing list expressions, the desugarer must translate list comprehensions into maps and reductions. This is similar to other well-understood schemes for compiling list comprehensions, and is outlined in the next section.

The most important aspect of the desugarer is its ability to perform `map/reduce` elimination on the fly. It exploits the fact that all of the transformations necessary are implicit (that is, they will be employed regardless of which desugaring scheme is chosen) or local (requiring only three additional arguments beyond the syntactic information from the outermost expression being desugared). The desugarer also incorporates heuristics so that the various folds, maps, and reductions can be implemented efficiently; the most notable example so far is η-abstraction, which generates `for` loops from recusive functions with higher-order return values.

4.1 Desugaring rules using `map` and `concat`

The problem of compiling list comprehensions is well understood; the canonical approach is the one given by Wadler[15]. A simple and correct desugaring for list comprehensions (very similar to Wadler's may be found in Figure 4.1. The most significant difference between this scheme and Wadler's (apart from the inclusion of more disjuncts) is that Wadler's function `flatMap` has been expanded into `concat` applied to `map` (that is, the "flattening" and the "mapping" are separated). The rules can be used to compile the main loop of `queens`, so that

$\mathcal{TE}[\![[Es\ ]]\!]$ $= \mathcal{TF}[\![[Es\ ]]\!]$
$\mathcal{TE}[\![E_1$`++`$E_2]\!]$ $= \mathcal{TF}[\![E_1$`++`$E_2]\!]$
etc. for other list expressions E for which $\mathcal{TF}[\![E]\!]$ is defined

$\mathcal{TE}$ applies recursively to inner expressions (applications, conditionals, *etc.*) not otherwise desugared

$\mathcal{TF}[\![[E_1, E_2, \ldots E_n]]\!]$ $= [\mathcal{TE}[\![E_1]\!]$`++`$[\mathcal{TE}[\![E_2]\!]$`++`$\cdots[\mathcal{TE}[\![E_n]\!]]]$
$\mathcal{TF}[\![E_1$`++`$E_2]\!]$ $= \mathcal{TF}[\![E_1]\!]$`++`$\mathcal{TF}[\![E_2]\!]$
$\mathcal{TF}[\![[E$ `|` $]]\!]$ $= [\mathcal{TE}[\![E]\!]]$
$\mathcal{TF}[\![[E$ `|` B `,` $Q\ ]]\!]$ $=$ `if` $\mathcal{TE}[\![B]\!]$ `then` $\mathcal{TF}[\![[E$ `|` $Q\ ]]\!]$ `else []`
$\mathcal{TF}[\![[E$ `|` P `<-` L `,` $Q\ ]]\!]$ $=$ `concat (map (\e -> case e of`
P `->` $\mathcal{TF}[\![[E$ `|` $Q\ ]]\!]$
`_ -> []`)
$\mathcal{TF}[\![L]\!]$)

$\mathcal{TF}[\![E]\!]$ $= \mathcal{TE}[\![E]\!]$ for all other forms

Figure 4.1: Desugaring rules for lists using `map` and `concat`

```
queens m = [ q ++ [n] | q <- queens (m-1), n <- [1..k], safe q n]
```

becomes:

```
queens m = concat (map (\q -> concat (map (\n -> if safe q n then
                                                   q ++ ([n]++[])
                                                 else [])
                                          [1..k]))
                       (queens (m-1)))
```

Most schemes for list comprehension compilation eliminate the append operations in the desugaring rules by parameterizing them with respect to the tail of the list being desugared. Indeed, if so many append operations were performed at run time they would irreparably damage program performance. Instead of eliminating calls to append first, however, the program transformations described so far can be applied directly to the desugaring rules. Given the conflicting goals of parallelism and sequential efficiency, this is in the true spirit of Wadler's presentation [15].

The function `concat` can be defined in terms of a reduction:

```
concat xs              = reduce (++) [] xs
```

In the desugaring rules, replacing the occurrence of `concat` by the above definition results in a scheme which constructs lists using only `reduce`, `map`, `[]`, and the list unit operation $[E]$ (expressed as a Haskell function by writing `(:[])`). These are exactly the operations with which the list monoid and its homomorphisms were defined (Section 2.2.3).

Defining list comprehensions in terms of the list monoid allows the comprehensions to be computed in any other monoid. Since traversals express homomorphisms with the list monoid, substituting the relevant monoid operations in a traversal will result in code which eliminates the construction and traversal of the list entirely.

4.2 Incorporating reductions

To realize this goal, a traversal must be carried down to the point where the list being traversed is constructed. Traversals are described canonically, as in Section 2.2.4. After converting every traversal into canonical form, the desugarer substitutes the appropriate functions for the operations in the list monoid. To see how this substitution is derived, compare the canonical form of a traversal over a list `l` to the identity operation on the same list using the above functions:

```
(reduce   a  (\x->x) (map   u   l)) t   canonical traversal
(reduce (++)    []   (map (:[]) l))     list identity
```

So in the code to generate the list, every append operation `(++)` will be replaced by a, every empty list `[]` by the identity function, and every unit operation `(:[])` by u. The result will be a function which, when applied to t, returns the same result as if the list had been separately generated and traversed.

This unconditional replacement is justified by the *promotion rules* for lists described by Bird [2]. These rules push calls to `map` and `reduce` inwards through `[]`, calls to `concat`, and calls to the list unit `(:[])`:

```
map g []                = []
reduce f z []           = z

map g (E:[])            = (g E:[])
reduce f z (E:[])       = E

map g (concat L)        = concat (map (map g) L)
reduce f z (concat L) = reduce f z (map (reduce f z) L)
```

To see how this works, try performing a canonical reduction on the result of `queens m`:

```
(reduce a id (map u (queens m))) t
```

If the definition of queens is inlined, and the promotion rules are applied, the resulting code is:

```
(reduce a id (map (\q -> reduce a id (map (\n ->
                                    if safe q n then
                                      a q (a (u n) id)
                                    else [])
                                  [1..k]))
                  (queens (m-1)))) t
```

The fundamental problem with the canonical form is that it uses higher-order functions; no call to a or u can yield a first-order result until t is provided as an argument. Thus, the above desugaring of `queens` will create closures in place of list elements, gaining nothing.

4.2.1 Performing calls at compile time

Luckily, the structure of the function a is known. It is in one of the following forms:

```
(\l r t -> l (r t))          from foldr
(\l r t -> r (l t))          from foldl
(\l r t -> f (l t) (r t))    from reduce
```

In addition, every application of a in the desugared code can be given two arguments—rather than unconditionally replacing `(++)` by a, simply replace only those occurrences which have arity two (including those which are introduced during desugaring). Since *every* call to the unit function u is introduced by desugaring, all calls to u will have exactly one argument.

Given that the desugarer is generating two-argument applications to a, and that a is known at compile time, the next step is to eliminate a entirely from the desugared code, performing all calls to a at compile time. Every application of a can be viewed as an *administrative redex*, introduced by the transformation and open to immediate reduction at compile time [12]. This is very similar to what Wadler does with calls to `(++)` [15]; indeed, a represents the operator which replaces `(++)`.

The simplest way to perform these calls is to represent a as an ordinary higher-order function in the desugarer. Thus, the desugarer will use *actual* higher-order functions at compile time, rather than *representations* of higher-order functions to be called at run time. Note also that the first two arguments to a are also assumed to be higher-order functions with a single argument apiece. These functions will therefore also be called at compile time.

Unsurprisingly, values of the functions a and u form a class of monoids which parallel the structure of the monoids for traversal—that is, the elements of all the monoid types are compile-time functions which, when called, yield a piece of code to be executed at run time. The binary operation of a particular monoid is the reduction function a imposed by the traversal, and the unit operation is the function u which generates code to process a single list element. The identity of a is the compile-time identity function (since a's arguments are called at compile time). The final argument to all of these higher-order functions is a piece of run-time code named t which has the same role as the final argument in the canonical form—that is, for folds it represents the cumulative value of the computation, and for reductions it represents the identity of the operation used.

The final version of the desugaring rules can be seen in Figure 4.2. The underlined constructs appear in the code generated by the desugarer, as do the values returned by recursive calls to $\mathcal{TE}$ and $\mathcal{TF}$. The remaining functions are evaluated entirely at compile time. $\mathcal{TE}$ schemes have been added for `foldr`, `foldl`, and `reduce` so that their list arguments are desugared in the appropriate monoid. Note also that $\mathcal{TF}$ schemes have been added—not only for cons and nil (desugaring rules for these constructors would have been vacuous in the previous scheme), but also for the functions `reverse` and `map`. The definition of `reverse` simply reverses the order in which sublists are glued together by a; since this function restructures its argument, it is impossible to define using reduction or synthesis without its

$$
\begin{array}{ll}
\mathcal{TE}[\![[Es\ ]]\!] & = \mathcal{TF}[\![[Es\ ]]\!]\ a_{list}\ u_{list}\ t_{list} \\
\mathcal{TE}[\![E_1 \texttt{++} E_2]\!] & = \mathcal{TF}[\![E_1 \texttt{++} E_2]\!]\ a_{list}\ u_{list}\ t_{list} \\
\multicolumn{2}{c}{\textit{etc.}\text{ for other list expressions } E \text{ for which } \mathcal{TF}[\![E]\!] \text{ is defined} \\
\mathcal{TE}[\![\texttt{reduce}\ f\ z\ L]\!] & = \mathcal{TF}[\![L]\!]\ (\lambda l\, r\, z.\, \mathcal{TE}[\![f]\!]\ (l\ z)(r\ z))\ (\lambda e\, z.\, e)\ \mathcal{TE}[\![z]\!] \\
\mathcal{TE}[\![\texttt{foldr}\ f\ z\ L]\!] & = \mathcal{TF}[\![L]\!]\ (\lambda l\, r\, e.\, l\ (r\ e))\ (\lambda e\, t.\, \mathcal{TE}[\![f]\!]\ e\ t)\ \mathcal{TE}[\![z]\!] \\
\mathcal{TE}[\![\texttt{foldl}\ f\ z\ L]\!] & = \mathcal{TF}[\![L]\!]\ (\lambda l\, r\, e.\, r\ (l\ e))\ (\lambda e\, t.\, \mathcal{TE}[\![f]\!]\ t\ e)\ \mathcal{TE}[\![z]\!]
\end{array}
$$

$\mathcal{TE}$ applies recursively to inner expressions (applications, conditionals, *etc.*) not otherwise desugared

$$
\begin{array}{ll}
\mathcal{TF}[\![[E_1, E_2, \ldots E_n]]\!] & a\ u\ t = ((u\ \mathcal{TE}[\![E_1]\!])\ \text{`}a\text{`}\ (u\ \mathcal{TE}[\![E_2]\!])\ \text{`}a\text{`}\ \ldots\ (u\ \mathcal{TE}[\![E_n]\!]))\ t \\
\mathcal{TF}[\![E_1 \texttt{++} E_2]\!] & a\ u\ t = (\mathcal{TF}[\![E_1]\!]\ a\ u\ \text{`}a\text{`}\ \mathcal{TF}[\![E_2]\!]\ a\ u)\ t \\
\mathcal{TF}[\![[]]\!] & a\ u\ t = t \\
\mathcal{TF}[\![A{:}L]\!] & a\ u\ t = (u\ A\ \text{`}a\text{`}\ \mathcal{TF}[\![L]\!]\ a\ u)\ t \\
\mathcal{TF}[\![\texttt{map}\ f\ L]\!] & a\ u\ t = \mathcal{TF}[\![L]\!]\ a\ (\lambda e\, t'.\, u\ (\mathcal{TE}[\![f]\!]\ e)\ t')\ t \\
\mathcal{TF}[\![\texttt{reverse}\ L]\!] & a\ u\ t = \mathcal{TF}[\![L]\!]\ (\lambda l\, r\, e.\, (r\ \text{`}a\text{`}\ l)\ e)\ u\ t \\
\mathcal{TF}[\![[E\ |\ ]]\!] & a\ u\ t = u\ \mathcal{TE}[\![E]\!]\ t \\
\mathcal{TF}[\![[E\ |\ B\ ,\ Q\ ]]\!] & a\ u\ t = \underline{\texttt{if}}\ \mathcal{TE}[\![B]\!]\ \underline{\texttt{then}}\ \mathcal{TF}[\![[E\ |\ Q\ ]]\!]\ a\ u\ t\ \underline{\texttt{else}}\ t \\
\mathcal{TF}[\![[E\ |\ P \texttt{<-}\ L\ ,\ Q\ ]]\!] & a\ u\ t = \texttt{let}\ f\ e\ t' = \underline{\texttt{case}}\ e\ \underline{\texttt{of}} \\
 & \qquad\qquad P\ \underline{\texttt{->}}\ \mathcal{TF}[\![[E\ |\ Q\ ]]\!]\ a\ u\ t' \\
 & \qquad\qquad \_\ \underline{\texttt{->}}\ t' \\
 & \quad \texttt{in}\ \mathcal{TF}[\![L]\!]\ a\ f\ t \\
\mathcal{TF}[\![E]\!] & a\ u\ t = \underline{\texttt{reduce}}\ a\ (\lambda x.\, x)\ (\underline{\texttt{map}}\ u\ \mathcal{TE}[\![E]\!])\ t \quad \text{for all other forms}
\end{array}
$$

Figure 4.2: Desugaring rules for lists with implicit `map`/`reduce` elimination

argument list being constructed. The rule for `map` desugars the list being mapped over using a new u which is the composition of the current value of u and the function which is being mapped.

Also new in this desugaring scheme are the constant values a_{list}, u_{list} and t_{list}. These are the default values of a, u and t, and actually build a list in memory. While the operations of the list monoid could be used here, it is much more efficient to define them as if they were generated by a `foldr` as follows:

$$
\begin{array}{l}
a_{list} = (\lambda l\, r\, t.\, l\ (r\ t)) \\
u_{list} = (\lambda e\, t.\, e : t) \\
t_{list} = []
\end{array}
$$

This translation linearizes the process of creating a list, but eliminates the enormous number of append operations required if the list monoid were used. Because the list is being created as a linear data structure in memory, it must be traversed linearly in any case—so the loss of parallelism may not be significant. Open lists (discussed in Section 4.6) provide an alternative definition for these three quantities more similar to the original list monoid.

4.3 Loops for innermost expressions

The transformations described so far push the consumption of a list towards the point of its creation. Once there, however, the following disjunct is encountered:

$$\mathcal{TF}[\![E]\!] \quad a\ u\ t = \underline{\texttt{reduce}}\ a\ (\lambda x.\, x)\ (\underline{\texttt{map}}\ u\ \mathcal{TE}[\![E]\!])\ t$$

There is no provision for traversing the list E efficiently; indeed, E could well be a call to `unfold` or `synthesize`, and could therefore be entirely eliminated. Rules and heuristics are necessary to perform much of the inlining explored in the previous two chapters.

4.3.1 Basic rules

The most basic step is to generate a recursive function call in place of the `reduce` and the `map` (all code shown is generated by the desugarer and executed at run time; only the calls to a, u and $\mathcal{TE}$ are performed at compile time):

```
TF[[E]]  a u t = let h lst tl = case lst of
                                  []   -> tl
                                  e:es -> a ((u e) (λt'. h es t')) tl
                 in h TE[[E]] t
```

As noted in Section 2.2.4, this code generates a tail-recursive loop when a is generated by a `foldl`, and otherwise results in a single recursive function whose second argument is constant.

Basic rules for `unfold` and `synthesize` can also be added. They are quite similar to the previous rule:

```
TF[[unfold P F V]]          a u t = let h v tl = case TE[[P]] v of
                                                   True  -> tl
                                                   False -> (a (u e) (λt'. h v' t')) tl
                                                     where (e,v') = TE[[F]] v
                                    in h TE[[V]] t

TF[[synthesize P E F V]]  a u t = let h v tl = case TE[[P]] v of
                                                 True  ->
                                                   case TE[[E]] v of
                                                     True  -> tl
                                                     False -> u v tl
                                                 False ->
                                                   (a (λt'. h v' t') (λt''. h v'' t'')) tl
                                                   where (v',v'') = TE[[F]] v
                                  in h TE[[V]] t
```

With rules in place to generate loops, it is a simple matter to desugar `queens`. To obtain the desugaring in Figure 4.3, the expression `[1..k]` has been replaced by the equivalent unfolding, `unfold (>k) (\v->(v,v+1)) 1`. Note how the tail of the list being constructed is passed in `tl` in favor of using calls to `(++)`.

```
queens m =
    let h1 lst1 = case lst1 of
                  [] -> []
                  e1 : es1 ->
                      let h2 v2 = case v2 > k of
                                      True -> h1 es1
                                      False ->
                                          case safe e1 v2 of
                                              False -> h2 (v2 + 1)
                                              True ->
                                                  (let h3 lst3 =
                                                        case lst3 of
                                                          [] -> v2 : []
                                                          e3 : es3 -> e3 : h3 es3
                                                   in  h3 e1) : h2 (v2 + 1)
                      in  h2 1
    in  h1 (queens (m - 1) )
```

Figure 4.3: A desugared version of **queens** with the introduction of some simple loops

4.3.2 Specializing the generated code

Because a can only take on three different values, it is simple to specialize code generation based on a's value. The values of a can be distinguished either by calling a with appropriate arguments and seeing what is returned, by representing a by a simple enumerated type which can be converted into a function as necessary, or by tagging a with the direction of traversal. The pH desugarer tags the value of a, since this method permits a itself to be specialized on its inputs (for example, by discarding identities) while still retaining an easily-checked indication of its basic structure.

Since it is easy to tell if a results from a call to `reduce`, several improvements can be made in the generated code simply given that reductions are associative. The obvious change is to traverse the list iteratively rather than recursively. Unfortunately, this is not always as efficient as it sounds. In particular, observe the following reduction:

```
reduce (&&) True [False, True, True, True, True]
```

If the reduction is turned into a tail-recursive loop, the following code results (ignoring the fact that the list is constant, and could therefore be eliminated entirely by the desugarer):

```
let h lst tl = case lst of
                   []     -> tl
                   (e:es) -> h es (tl && es)
in h [False, True, True, True, True] True
```

This definition will traverse the entire input list. A programmer writing the same function would undoubtedly write a version similar to the non-tail-recursive one:

```
let h lst = case lst of
              []     -> True
              (e:es) -> e && (h es)
in h [False, True, True, True, True]
```

If (&&) is short-circuiting (as it would be in a lazy implementation) then only the first element of the list will be examined! Even if it isn't (the eager semantics of pH force all function arguments to be evaluated), the result will become available before the recursive calls are complete, and a clever compiler will be able to eliminate the tail call because it neither performs side effects nor influences the function's value (issues such as correctness and termination properties are addressed in the next section). The heuristic used to circumvent cases like these is quite naive—it simply looks for an opening **case** construct in the code generated for a, and generates a recursive definition if it finds one. A more sophisticated heuristic would insure that only one of the disjuncts requires the value of the recursive call before rejecting iteration in favor of recursion.

In addition, the compilation of **for** loops must be made more efficient. This is a simple matter of η-abstracting the ordinary recursive code to consolidate the additional functions which are introduced. In fact, the desugarer incorporates a general η-abstraction mechanism so that arbitrary higher-order funcitons will generate loops when used with **reduce** or **foldr**.

The rules for generating code for concrete list traversals and **unfold** can be found in Figure 4.4; the corresponding rules for **synthesize** are in Figure 4.5. When applied to **queens** (using **unfold** as above), it produces the code in Figure 4.6.

4.4 Concrete implementation

The desugaring rules and the code to generate specialized list traversals have been incorporated into the front end of the **hbcc** compiler. This compiler is being written primarily by Lennart Augustsson, and is intended to compile both Haskell and pH; as of this writing the front end for Haskell is complete and correct. Writing the desugarer involved completely replacing the list comprehension rules, adding rules for the various functions treated specially by the desugaring rules, and installing extensive "plumbing" to propagate necessary information through the remaining portions of the desugarer. A separate module implements the representations for canonical traversals, and is actually responsible for generating the compiled code for reductions. The current implementation of the desugarer operates independently of other compiler phases, meaning that modifications to the other phases of the compiler will not require corresponding changes in the code for the desugarer.

When a is generated by a `foldl`:

For already-constructed lists $\mathcal{TF}[\![E]\!]\ a\ u\ t$:

```
let h lst tl = case lst of
                 []    -> tl
                 e:es -> (λt'. h es t') ((u e) tl)
in h TE[[E]] t
```

For $\mathcal{TF}[\![\texttt{unfold}\ P\ F\ V]\!]\ a\ u\ t$:

```
let h v tl = case TE[[P]] v of
               True  -> tl
               False -> (λt'. h v' t') ((u e) tl)
                        where (e,v') = TE[[F]] v
in h TE[[V]] t
```

When a is generated by a `foldr`, or by a `reduce` for which ((u e) 'a' (λt'. h es t')) tl is either a function or a `case` statement, return the following after η-abstracting `h`:

For already-constructed lists $\mathcal{TF}[\![E]\!]\ a\ u\ t$:

```
let h lst = case lst of
              []    -> t
              e:es -> ((u e) 'a' (λt'. h es)) t
in h TE[[E]]
```

For $\mathcal{TF}[\![\texttt{unfold}\ P\ F\ V]\!]\ a\ u\ t$:

```
let h v = case TE[[P]] v of
            True  -> t
            False -> ((u e) 'a' (λt'. h v')) t
                     where (e,v') = TE[[F]] v
in h TE[[V]]
```

When a is generated by a call to `reduce` whose second disjunct value (see above) is neither a function nor a case:

For already-constructed lists $\mathcal{TF}[\![E]\!]\ a\ u\ t$:

```
let h lst ac = case lst of
                 []    -> ac
                 e:es -> h es ((u e) 'a' ac)
in h TE[[E]]
```

For $\mathcal{TF}[\![\texttt{unfold}\ P\ F\ V]\!]\ a\ u\ t$:

```
let h v tl = case TE[[P]] v of
               True  -> tl
               False -> h v' ((u e) 'a' tl)
                        where (e,v') = TE[[F]] v
in h TE[[V]] t
```

Figure 4.4: Efficient compilation of reductions for lists and `unfold`

For $\mathcal{TF}[\![$`synthesize` $P\ E\ F\ V]\!]\ a\ u\ t$:

When a is generated by any call to `reduce`:

```
let h v = case TE[[P]] v of
              True  -> case TE[[E]] v of
                         True  -> t
                         False -> u v t
              False -> ((λt'. h v') 'a' (λt''. h v'')) t
                       where (v',v'') = TE[[F]] v
in h TE[[V]] t
```

When a is generated by `foldl` or `foldr`:

```
let h v tl = case TE[[P]] v of
               True  -> case TE[[E]] v of
                          True  -> tl
                          False -> u v tl
               False -> ((λt'. h v' t') 'a' (λt''. h v'' t'')) tl
                        where (v',v'') = TE[[F]] v
in h TE[[V]] t
```

Figure 4.5: Efficient compilation of reductions for `synthesize`

```
queens m =
    let h1 lst1 tl1 = case lst1 of
                  [] -> tl1
                  e1 : es1 ->
                    let h2 v2 tl2 =
                      case v2 > k of
                        True -> tl2
                        False ->
                          case safe e1 v2 of
                              False -> h2 (v2 + 1) tl2
                              True ->
                                (let h3 lst3 tl3 = case lst3 of
                                                     [] -> tl3
                                                     e3 : es3 -> e3 : h3 es3 tl3
                                   in  h3 e1 [v2]) : h2 (v2 + 1) tl2
                    in  h2 1 (h1 es1 tl1)
    in  h1 (queens (m - 1) ) []
```

Figure 4.6: A desugared version of `queens` with more agressive loop generation

4.5 Correctness

There are some correctness issues raised by the desugaring rules as they stand. First, the rules above tend to inline certain quantities aggressively, particularly t. In a lazy environment this results at worst in code duplication; in an eager environment, it may cause side-effecting expressions to be skipped over or executed multiple times. Problems of duplication and elimination can be circumvented by binding new names to the relevant values and inlining these new names in the generated code.

A higher-level problem (alluded to in the previous section) is the issue of termination properties. The first question which must be answered is, 'Should program optimizations preserve the precise termination properties of the source program?' The pH compiler is being constructed with the assumption that the termination properties of the generated code must be *at least as good* as those of the source program, in the sense that the generated code must terminate in every situation in which the source program semantically terminates (modulo resource limitations, which are machine-imposed). Thus, it is all right for the desugarer to eliminate non-side-effecting expressions which do not contribute to computing the value of a function, since such expressions can only cause the program to potentially loop forever.

The second question about termination properties which must be answered is, 'What are the termination properties of the language features which are given special treatment by the desugarer?' The answer to this question depends heavily on which representation is chosen for the final loop. For example, depending on whether `reduce` is defined recursively or iteratively, the following piece of code may or may not return a result (it will not terminate if eager semantics are used):

```
reduce (&&) True (False:True:True:True:···
```

As a programmer, one cannot reason systematically about all such cases as one is writing code. The safest stand in such ambiguous cases is simply to state that the program does not produce a result—that is, any infinite or circular list which is used in a list comprehension or as an argument to `reduce` is not guaranteed to produce a result, and so semantically it must be considered not to; if it actually happens to do so, it is simply through the cleverness or beneficence of the compiler.

4.6 Language Features

The specialized comprehensions of Id originally motivated this research, since a well-understood but highly specialized set of rules existed to handle them[19]. The earliest goal was to match or improve upon the code produced in Id, but to do so in a more general and flexible framework.

4.6.1 Open lists

To this end, *open lists* must be introduced into the language. For the sake of this exposition, the following definitions will be used for open lists:

```
emptyOpen :: () -> Open a                -- return empty open list
(>:)      :: Open a -> a -> Open a       -- add element to tail
(++:)     :: Open a -> Open a -> Open a  -- append tail to head
closeOpen :: Open a -> Open a -> [a]     -- convert given head and tail
```

Open lists are represented internally by data structures which are identical to lists, except that their tails are initially empty and can later be filled in using I-structure operations. However, the above definitions keep these particular features under wraps. The only observable results are that all of the open list operations can be performed in unit time, and any given open list may only be modified once (that is, supplied as an argument to `>:`, as the second argument to `closeOpen`, or as the first argument to `++:`). Note that open lists form a class of monoids whose types are of the form `openlist` α with unit operation `>:` and binary operation `++:`. Unfortunately, open lists do not have a well-defined identity. The problem is that while a *call* to `emptyOpen` will act as an identity if it appears as either argument to `(++:)`, there is no single, distinguishable *object* which can be so used.

A solution using higher-order functions can be imagined; all operations could be made to take a dummy argument, which would make `emptyOpen` itself the identity of the monoid. Instead, the desugarer exploits the fact that internally it is simply using a higher-order representation of the above functions. The actual process of appending two open lists will require the tail of the list to be created and named, then fed separately to each piece of the list being built. Thus, as with `foldl` and `foldr`, open lists are represented by a monoid at compile time, but not at run time. This approach is chosen because using run-time higher-order functions requires manipulating the *representations* of these higher-order functions, and beta-reducing these representations. While the desugarer is capable of performing the necessary manipulations, doing so introduces substantial overhead. Since detecting and circumventing the problem is trivial, the compiler does so.

Thus, introducing open lists gives the following default values for list traversals:

$$
\begin{aligned}
a_{list} &= (\lambda l\, r\, t.\ \texttt{let rst= emptyOpen()} \\
&\qquad\qquad \texttt{\_ =}(l\ t)\ \texttt{++:}\ \ \texttt{rst} \\
&\qquad\quad \texttt{in}\ (r\ \texttt{rst})) \\
u_{list} &= (\lambda e\, t.\ e\texttt{>:}t) \\
t_{list} &= \textit{name given to outermost}\ \texttt{emptyOpen()}
\end{aligned}
$$

One problem remains—an open list is not a list! A type conversion function, `closeOpen`, exists, but it needs to be introduced *implicitly* by the desugarer in order to maintain type safety. This simply means that the expressions which are desugared by the $\mathcal{TF}$ scheme must have a "wrapper" placed around the corresponding disjuncts in $\mathcal{TE}$. Thus, the desugaring rules for such constructs will now have the following form (where the part inside the double braces changes depending on the rule):

$$
\begin{aligned}
\mathcal{TE}[\![\texttt{[}Es\ \texttt{]}]\!] \quad &= \texttt{let hd = emptyOpen()} \\
&\quad\ \texttt{in closeOpen hd (hd ++:}\ \mathcal{TF}[\![\texttt{[}Es\ \texttt{]}]\!]\ a_{list}\ u_{list}\ t_{list}\texttt{)}
\end{aligned}
$$

Finally, the actual compile-time representation for a_{list} performs a simple optimization based on the following identity:

$$l \texttt{ ++: } (t \texttt{ :> } e) \quad = l \texttt{ :> } e$$
when t is bound to `emptyOpen()`

This insures, for example, that the list `t ++ [1,2]` is desugared as:

```
t:>1:>2
```

and not as:

```
t++:(let t=emptyOpen();_=t>:1 in t)++:(let t=emptyOpen();_=t>:2 in t)
```

A version of queens compiled using open lists can be found in Figure 4.7. Note how the innermost loop constructs its open list sequentially using `(>:)`, while the outer loop simply joins together the inner lists using `(++:)`. Thus, every execution of the inner loop is free to proceed in parallel.

4.6.2 Arrays

One of the chief complaints about Haskell from the Id community is that Haskell's array system is bulky—in particular, a number of array operations require a list of index/value bindings as an argument. In pH, this mechanism supplants Id's specialized array comprehensions and accumulators. Thus, pH trades syntactic compactness and elegance for a potentially large loss in performance. Id array comprehensions make use of I-structures to construct an array in parallel with its use [19]. By defining a list traversal which uses these same techniques, the full performance of Id array comprehensions can be realized.

The basic array constructor in Haskell is the function `array`. For example, this expression creates a grade-school multiplication table for pairs of integers between 0 and 10:

```
table   = array b [ i := a*b | i@(a,b) <- range b ]
            where b = ((0,0),(10,10))
```

The first argument to `array` is the lower and upper bounds of the array; the second argument is a list of bindings between indices and values. Any valid index which is not bound is considered to have no value; indexing into an array is only meaningful if the array has been defined at that index.

In pH, the implementation of arrays is kept under wraps, just as the implementation of lists in terms of open lists is hidden by the compiler. The definition of arrays can be thought of in terms of a primitive I-structure (write-once) array type called `IArray`, with the following basic operations:[1]

```
emptyIArray :: Ix a => (a,a) -> IArray a b        -- empty IArray given bounds
writeIArray :: Ix a => IArray a b -> a -> b -> () -- write b at index a, then ()
readIArray  :: Ix a => IArray a b -> a -> b       -- read from index a
closeIArray :: Ix a => IArray a b -> Array a b    -- convert IArray into array
```

[1]The problem of representing arrays has several other aspects, such as abstracting and linearizing indices, which are ignored in this presentation for simplicity.

```
queens m =
 let hd1 = emptyOpen()
 in
  closeOpen hd1
   (let h1 lst1 tl1 =
         case lst1 of
           [] -> tl1
           e1:es1 ->
              h1 es1
                 (let rst2 = emptyOpen()
                      _  = tl1 ++: rst2
                      h2 v2 tl2 =
                        case v2 > k of
                          True -> tl2
                          False ->
                            let v2' = v2 + 1
                            in
                              h2 v2'
                                 (let rst2 = emptyOpen()
                                      _  = tl2 ++: rst2
                                  in
                                   case safe e1 v2 of
                                     False -> rst2
                                     True ->
                                       rst2 >: (let hd3 = emptyOpen()
                                                in closeOpen hd3
                                                    ((let h3 lst3 t3l =
                                                            case lst3 of
                                                              [] -> tl3
                                                              e3 : es3 ->
                                                                h3 es3 (tl3 :> e3)
                                                      in h3 e1 hd3)
                                                     >: v2)))
                  in  h2 1 rst2)
    in  h (queens (m - 1)) hd)
```

Figure 4.7: A desugared version of queens using open lists

The definition of `array` is as follows:

```
array bnds bindings = let iarr = emptyIArray bnds
                          sideEffect =
                            reduce (\sideEffect1 sideEffect2 -> ()) ()
                                   (map (\(indx:=val) ->
                                                writeIArray iarr indx val)
                                        (someOrder bindings))
                      in iarr
```

Note that the function passed to `reduce` does nothing! Both of its arguments must be evaluated for side effects, but no furthur action is necessary. This articulates the notion that any assignment to the array is completely independent every other assignment (except if indices overlap, in which case the program is erroneous anyway). In addition, by using an I-structure model and executing generation and use in parallel the array can be returned before the `reduce` expression had been completely evaluated for its side effects.

There is one strange function in the above code—`someOrder`. This function returns a permutation of the list provided to it. In general, it could be defined as the identity function. When it is encountered by the desugarer, it causes the desugarer to assume that the internal function a is commutative; the desugarer will then try to choose the best possible traversal directions for the sublists of the list it is desugaring. It is used here because the function being passed to `reduce` really *is* commutative; however, it could also be used to indicate that the program doesn't care about the order in which the result is computed, since it is being used as, say, a bag. One instance where the desugarer would reorder the argument to `someOrder` is the list `([1..5] ++ reverse [1..5])`. This would ordinary yield the list `[1,2,3,4,5,5,4,3,2,1]`. However, if the list `[1..5]` can be computed more efficiently forwards than backwards, then `someOrder ([1..5] ++ reverse [1..5])` will instead yield the list `[1,2,3,4,5,1,2,3,4,5]`.

Now that the necessary machinery is in place, the function `table` can be desugared (inlining `range b`) to obtain:

```
[(a,b) | a <- unfold (>10) (\n->(n,1+n)) 0, b <- unfold (>10) (\n->(n,1+n)) 0]
```

The desugared code for `table` can be found in Figure 4.8. Note that there is an extraneous loop constant—`tl` or `tl'`—being carried around by the loops, but that this value is simply discarded by the next iteration. Loop optimization will eliminate this unused value. Nonetheless, each iteration of the outer and inner loops is free to proceed in parallel with every other iteration, since there are no data dependencies between them.

```
table = let bnds = ((0,0),(10,10))
            iarr = emptyIArray bnds
            sideEffect =
              let h a tl =
                case a > 10 of
                  True  -> tl
                  False ->
                    h (1+a)
                      let sideEffect1 =
                            let h' b tl' =
                              case b > 10 of
                                True  -> tl'
                                False ->
                                  h (1+b)
                                    let sideEffect1' = writeIArray iarr (a,b) (a*b)
                                    in ()
                            in h' 0 ()
                      in ()
              in h 0 ()
        in iarr
```

Figure 4.8: A desugared version of `table`

Chapter 5

Evaluating map/reduce elimination

This chapter evaluates the existing desugarer in an attempt to answer three questions. First, does the desugarer actually implement the transformations described in the preceeding chapters? This can be verified by seeing if the desugarer produces identical code for instances of the transformations it embodies; such an approach also gives insight into the desugarer's shortcomings. Next, does the desugarer improve the performance of programs? Existing results suggest that it does, and that further development of this approach is justified. Finally, what sort of programming style will result in efficient programs in this framework? A few examples will be developed to show how lists can be used to express abstract datatypes and embody a general notion of "iteration" versus "iteration in parallel."

5.1 Verifying the operation of the desugarer

Every program optimization attempts to convert a source program to an equivalent (but hopefully more efficient) target program. The programmer can consider any two pieces of code which produce identical output from the optimizer to be equivalent with respect to that program transformation. A detailed characterization of this equivalence is beyond the scope of this thesis; this chapter will instead explore two uses of program equivalences. First, the desugarer can be used to show the equivalances which it supposedly embodies. This provides a simple (though not exhaustive) check on the correctness of the desugarer. Second, by passing pieces of code which are known to have the same operational behavior to the desugarer, the power of the desugarer's transformations can be assesed. It should thereby be possible to develop a notion of what the desugarer can and cannot do, and to explain the important differences between pieces of code which are operationally identical but produce different output from the desugarer.

As a simple example of the operation of the desugarer, examine the code produced from the following:

```
basic_reduce_ids a b e o z =
  let l1 = reduce o z []          -- should equal z
      l2 = reduce o z [e]         -- should equal e (or o e z or o z e)
```

```
    l3 = reduce o z (a ++ b)    -- these two lines should be equal
    l4 = (reduce o z a) `o` (reduce o z b)
in (l1,l2,l3,l4)
```

The intent of this routine is to verify that the desugarer's transformations of `reduce` fulfill the mathematical definition given for `reduce`—thus, the left hand sides of the mathematical definition appear. The right-hand side of the complex third part of the definition appears as well; because the desugarer will produce loops, the idea is to see if both pieces of code result in the same output. When desugared, we discover that `reduce` does indeed behave as expected:

```
basic_reduce_ids a b e o z =
        (z,
         (o e z),
         (o (let h = \ l tl -> case l of
                               [] -> tl
                               (:) e es -> h es (o tl e)
             in  h a z)
            (let h = \ l tl -> case l of
                               [] -> tl
                               (:) e es -> h es (o tl e)
             in  h b z)),
         (o (let h = \ l tl -> case l of
                               [] -> tl
                               (:) e es -> h es (o tl e)
             in  h a z)
            (let h = \ l tl -> case l of
                               [] -> tl
                               (:) e es -> h es (o tl e)
             in  h b z)))
```

It also suggests a possible optimization, since the second expression can most efficiently be compiled to `e` rather than `(o e z)` (remembering that `z` is the identity of `o`).

The equivalence of the final two expressions is serendipitous in two respects. First, the output of the desugarer is absolutely identical in both cases; in general, expressions will be considered equivalent if they are simply identical up to renaming, so that equivalence in this case just happens to be stronger (and thus easier to check visually). Second, because both expressions consume the lists `a` and `b` using `reduce`, the desugarer's output will be identical for both expressions regardless of context; this is not true of many expressions which generate one list from another. For example, consider the following code, which shows that the desugarer uses a definition of `++` equivalent to the commonly accepted one:

```
basic_append_identity a b =
  let l1 = a ++ b                    -- basic
      l6 = foldr (:) b a             -- definitional
  in (l1,l6)
```

This desugars to:

```
basic_append_identities e a b =
       ((let h = \ l -> case l of
                          [] -> b
                          (:) e es -> (:) e (h es)
         in  h a),
         (let h = \ l -> case l of
                          [] -> b
                          (:) e es -> (:) e (h es)
         in  h a))
```

This is straightforward enough. However, observe what happens when the latter list is consumed:

```
consumed_append o z a b =
        reduce o z (foldr (:) b a)
```

This function yields code quite different from the code for `l3` in the previous example:

```
consumed_append o z a b =
        (let h1 = \ l1 tl1 -> case l1 of
                            [] -> tl1
                            (:) e1 es1 -> h1 es1 (o tl1 e1)
         in  h1 (let h2 = \ l2 -> case l2 of
                                [] -> b
                                (:) e2 es2 -> (:) e2 (h2 es2)
                 in  h a)
                z)
```

This points up two important aspects of the desugarer. First, while the transformations it performs are local in the sense that they are based on small pieces of program text, and only require a fixed amount of information to proceed, that information (in the desugarer, the variables a, u, and t) has important effects on the desugaring process. The values of a, u and t are locally derived, too, since they are chosen when desugaring the immediately surrounding program text. Nonetheless, the behavior of the desugarer is context-dependent for expressions which generate new lists from existing ones.

The second (and related) argument is that the desugarer can only transform traversals of lists which are generated canonically—using simple list operations (such as `(:)`, `[]`, and `++`) directly, or using the list generating expressions `unfold` and `synthesize`. In particular, the traversal `foldl (:) b a` actually *consumes* the list `a`, and generates a completely new list whose contents are the same as `(a++b)`. The call to `(:)` does not appear directly in the source program, and consequently cannot be eliminated. Becaues `foldl` is a list consumer, the desugarer cannot use information about the list which is traversed to systematically generate the list which is produced. This suggests that using `++` to append lists is actually superior to explicitly writing code to traverse the lists, since `++` will be understood by the desugarer and represented efficiently regardless of context.

Some of the basic identities which the desugarer embodies are shown in Figure 5.1. Note especially the promotion rules, which are the rules used by the desugarer to move traversal towards construction. Also note that some definitions are given as equalities (using $=$) rather than equivalences (using $\equiv$); in

Copy/list identity			
`[x | x<-l]`	=	`foldr (:) [] l`	Operational
Append			
`a ++ b`	=	`foldr (:) b a`	Operational
`[] ++ b`	≡	`b`	Left identity
`a ++ []`	≡	`[x | x<-a]`	Right identity
`[e] ++ b`	≡	`e:b`	Unit
`(e:a) ++ b`	≡	`e:(a++b)`	Definition
Concat			
`[x | xs<-l, x<-xs]`	=	`reduce (++) [] l`	Operational
`[x | xs<-[], x<-xs]`	≡	`[]`	Empty
`[x | xs<-[l], x<-xs]`	≡	`[x | x <- l]`	Unit elim.
`[x | xs<-(l:ls), x<-xs]`	≡	`l ++ [x | xs<-ls, x<-xs]`	Definition
Map definition			
`map f []`	≡	`[]`	Empty
`map f [e]`	≡	`[f e]`	Unit
`map f (a++b)`	≡	`(map f a) ++ (map f b)`	Join
`reduce` definition			
`reduce o z []`	≡	`[]`	Empty
`reduce o z [e]`	≡	`(o z e)`	Unit
`reduce o z (a++b)`	≡	`reduce o z a 'o' reduce o z b`	Join
`foldr` definition			
`foldr f z []`	≡	`[]`	Empty
`foldr f z (a:l)`	≡	`f a (foldr f z l)`	Step
`foldl` definition			
`foldl f z []`	≡	`[]`	Empty
`foldl f z (l++[a])`	≡	`f (foldl f z l) a`	Step
`reverse` identities			
`reverse l`	=	`foldl (:) [] l`	Operational
`reverse (reverse l)`	≡	`l`	Self-inverse
`reduce o z (reverse l)`	≡	`reduce (\a b->o b a) z l`	Swap in red.
`foldl f z (reverse l)`	≡	`foldr (\e v->f v e) z l`	Swap in fold
Promotion rules			
`[f x|x<-[y|ys<-l, y<-ys]]`	≡	`[y|xs<-l, y<-[f x|x<-xs]]`	`map` thru.
`reduce o z [y|ys<-l, y<-ys]`	≡	`reduce o z [reduce o z ys|ys<-l\]`	`reduce` thru.

Figure 5.1: Definitional equivalences proved by the desugarer

such equations, the right-hand side only holds as written, because the list is not systematically constructed. Thus, the previous example appears as an equality. The remaining entries will be true in all contexts.

5.2 Limitations of the desugarer

The most important limitation of the desugarer was discussed in the previous section—it can only optimize traversals of lists which are generated and consumed systematically using the functions handled by the desugarer. With facilities to inline code, this limitation becomes less severe. However, there are classes of list traversal which, though they traverse their argument only once, cannot be described using these transformations. One important (and common) example is `zip`:

```
zip (a:as) (b:bs)       = (a,b):zip as bs
zip _      _            = []
```

This function causes problems with similar schemes for eliminating lists [6]; here the problem is that `zip` is traversing *two* lists. Three different systematic definitions can be given for `zip`; the first two consume an argument systematically, the last one generates the result systematically:

```
zipfold (l,ac) e        = case l of
                            []    ->([],ac)
                            (x:xs)->(xs,(x,e):ac)
zip1 a b                = (reverse . snd) (foldl zipfold (a,ac) b)
zip2 a b                = (reverse . snd) (foldr (flip.zipfold) (b,ac) a)
zip3 a b                = unfold (\(a,b)->case a of
                                            []->True
                                            _ ->case b of
                                                  []->True
                                                  _ ->False)
                                 (\(x:a,y:b)->((x,y),(a,b))
                                 (a,b)
```

A more revealing function is the equally common `takeWhile`. This function *is* representable and optimizable in systems which abstract over the list constructors (for example, using `build`):

```
takeWhile p (x:xs) | p x        = x:takeWhile p xs
                   | otherwise  = []
takeWhile p []                  = []
```

The problem with this function, and many others, is that expressing it as a list traversal requires more information than just the list itself—some information must be carried between iterations. In this example, some record must be kept of whether the predicate has yet returned False.

Several possible solutions to this problem can be imagined. The most obvious is to create functions which can be used to describe such situations. Unfortunately, this solution compromises much of the current desugarer, which relies on a very small core of functions to perform its transformations. In

choosing to use a parallel set of functions (`reduce` and `map`) to represent traversal, the desugarer has been structured in such a way that it will be difficult to incorporate traversals which (because of their data dependencies) must be described as sequential operations. Many of these traversals can be described by the list paramorphism found in Meijer *et al* [9].

Another significant limitation of the desugarer is its inability to handle nested lists cleanly. This is evident in the following statement of `concat.(map (map f))` (the right-hand side of the promotion rule for `map` across `concat`):

```
[y | ys <- [[f x | x <- xs] | xs <- l], y <- ys]
```

This generates the following code:

```
let h1 = \ l1 -> case l1 of
                 [] -> []
                 (:) e1 es1 ->
                     let es' = h1 es1
                     in  let h2 = \l2 -> case l2 of
                                         [] -> es'
                                         (:) e2 es2 -> (:) e2 (h2 es2)
                         in  h2 (let h3 = \l3 -> case l3 of
                                                 [] -> []
                                                 (:) e3 es3 -> (:) (f e3) (h3 es3)
                                 in  h3 e1)
in  h l
```

Here, the outermost traversals are properly combined to form the tail-recursive function `h1`; however, the inner map (`h3`) is separated from the inner loop of `concat` (`h2`). This is because the inner traversals are not nested until the outer traversals have already been combined by the desugarer; by this time the code for both loops has already been generated.

5.3 Performance

Because the existing desugarer lacks a general code inliner, program code which is compiled directly derives very little benefit from the transformations performed by the desugarer. In addition, the compiler for which the desugarer is written is not complete enough to actually run code reliably and consistently. Thus, an extensive performance analysis could not be done. The early results presented in this section have been obtained by simulating the action of an inliner by hand. The resulting code has been compiled using the desugarer, and the output has been massaged somewhat to produce an equivalent Haskell program (type declarations have been inserted, dictionaries passed as arguments to prelude functions have been removed, and overlapping identifier names have been uniquified).

Two versions of each benchmark program have been evaluated—the original source program, and the inlined and desugared version. Both were compiled using `hbc`, the Chalmers Haskell B compiler, producing binary files for a Sun SPARCstation. The code was run on a workstation with a SPARC-2

	queens	shortest-path
hbc wall clock:		
undesugared	45.4	38.4
desugared	25.9	33.8
% improvement	43	11
hbc heap usage (MB):		
undesugared	190	98.6
desugared	66	78.1
% improvement	65	27

Figure 5.2: Performance data for benchmarks

```
main = (print.sum.concat.queens) 10


queens :: Int -> [[Int]]
queens k = queens_1 k
  where
    queens_1 :: Int -> [[Int]]
    queens_1 0 = [[]]
    queens_1 m = [ q ++ [n] | q <- queens_1 (m-1),
                              n <- [1..k],
                              safe q n]

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

Figure 5.3: 10-queens benchmark program

processor, and all of the times obtained varied by approximately .1 of a second at most. The performance data is summarized in Figure 5.2.

5.3.1 Queens

The first benchmark used to evaluate the desugarer was the implementation of the n-queens problem used as a benchmark in [6], and reproduced in Figure 5.3 (note that a simple version of the `queens_1` routine was used as an example in Chapter 4). This program is expressed entirely with lists, and as such stands to benefit immensely from the desugarer's optimizations. Statistics are given for the ten-queens problem, primarily because it runs in a reasonable amount of time when compared with larger and smaller problems. The program simply sums the queens' positions in each solution, and returns this sum; this forces every queen position in every solution to be completely evaluated, but generates a minimal amount of output.

In addition to inlining the prelude functions and replacing enumeration with `unfold`, the source

program itself has been aggressively inlined. The function `safe` is only called once, and so it has been replaced by an equivalent comprehension in the body of `queens_1`; further, since `queens` simply calls `queens_1`, the only call to `queens` was replaced by a call to `queens_1`, and the body of `queens_1` was inlined in `main`. Since this outermost call to `queens_1` is a list traversal which is being summed over, inlining it resulted in a radically different loop structure in `main`.

The most effective inlining, however, was to use the following definition for the function `and`:

```
and             = reduce (\a b -> if a then b else False) True
```

This definition simply inlines the definition of the boolean *and* function `&&` in the reduction. However, this causes the loop-generation heuristics to recognize that the reduction can be terminated early if it is represented as recursion, since the reduction now involves a `case` expression rather than a function application.

The results are quite dramatic—the execution speed improved by about 43% (where 50% corresponds to a twofold speedup), and memory usage improved by 65% (nearly a threefold drop). The speed improves even more (to about 47%) when both versions of the program are compiled with optimization on; this suggests that the desugarer exposes more opportunities for later optimization. All the figures obtained are comparable to the rough estimates for the same benchmark in [6]. Nonetheless, as noted there, this version of `queens` uses lists extensively, and so the performance improvement is likely to be much larger than in most applications.

5.3.2 Shortest Path

The shortest path program is based on a program by Paul Sanders which was obtained over the Internet. The version used for benchmarking repeatedly finds the shortest paths through a small acyclic graph, and returns the sum of the lengths of the paths (in order to force repeated execution). A rather poorly-written "split" routine was rewritten (an algorithm for merge sort will be described in the next section) before benchmarking; the naive shortest-path algorithm used was not changed, however. Each of three functions were only used once, and consequently were inlined and eliminated; because none of the remaining functions is a simple list traversal, more aggressive inlining was not performed.

The performance improvements for this program were more modest. The run time improved by 11%, heap allocation by about 27%. This is because while lists are used extensively in the program, they are treated as data structure, and consequently cannot be easily eliminated. Most of the intermediate lists which do get eliminated are created in one or another of the list comprehensions found in the program code.

5.4 Style

A program optimization is only effective if it speeds up the code which is actually written by programmers. This section explores a coding style which is intended to be abstract while still providing ample opportunity for optimization by the desugarer (assuming that there is a code inliner).

Earlier in this chapter, a number of examples were presented which showed that many expressions cannot be optimized by the desugarer, particularly those which construct lists without using `unfold` or `synthesize`. Often, however, a list needs to be constructed and stored into a data structure anyhow. In cases like this, it doesn't matter much from the compiler's point of view how the list is constructed. The best general rules of thumb for manipulating lists are these:

- Use a list comprehension or a prelude function whenever possible.
- If the list being traversed already exists in memory, it can be traversed by `unfold` or `synthesize` when building an intermediate result.
- If the list being constructed must exist in memory, it can be built using a fold or a reduction.

The last two cases make defining certain common list functions (such as `zip` above) somewhat difficult, since the programmer must guess how the function is being used before writing it. This suggests that a more general solution to such problems is needed.

One way of circumventing such difficulties is to familiarize programmers with some of the theory of lists [1, 2]. This, however, presumes a well-read and mathematically sophisticated programmer. The best alternative in this framework will be libraries of routines which can provide the appropriate structure to programs. For example, a routine which must repeatedly split a list and then recombine it can be written as a reduction along with an abstraction of `synthesize`:

```
splitting :: ([a]->([a],[a])) -> [a] -> [a]
splitting splitter = concat . synthesize smallish empty splitter
  where smallish (_:_:_) = False
        smallish _       = True
        empty []         = True
        empty _          = False
```

This function allows two different versions of sorting to be expressed efficiently. The simplest is quicksort

```
quickSort       = splitting QSplit
  where QSplit (x:xs) = (a,x:b)
          where (a,b) = partition (<x) xs
```

This function sorts stably; an unstable version can be made by partitioning `(someOrder xs)` instead. A more complicated sort is merge sort, which can be defined (unstably) as follows:

```
mergeSort       = reduce merge [] . splitting logSplit
  where merge as@(a:as') bs@(b:bs') | a < b     = a:merge as' bs
```

```
                                | otherwise = b:merge as bs'
logSplit l = reduce joiner . someOrder . map (\e->(e,[]))
joiner (a,b) (a',b') = (someOrder (a++b'),someOrder (b++a'))
```

This uses the splitter `logSplit` to break the list up so that `reduce` will call `merge` a number of times proportional to the length of the list, but with a call depth that is logarithmic in the length.

5.5 Using lifting to create monoids with identities

The true power of the `map/reduce` transformations, however, lies in their ability to express data structures *other* than lists—in fact, any data structure which is representable as a monoid. In order to represent an *arbitrary* monoid, however, a translation must be given which will create an identity for monoids which lack one. Mathematically, this is simply the process of *lifting* the monoid, which adds the meaningless element $\perp$ to it. In Haskell, this is expressed using the type `Maybe`, whose elements are `Nothing` (representing $\perp$) and `Just` *obj*, which represents the object *obj* from the original domain of the monoid. Thus, the following definitions can be used to lift a monoid:

```
lift                            = Just
lifted op (Just a) (Just b)     = Just (op a b)
lifted op Nothing  b            = b
lifted op a        Nothing      = a
unlift (Just a)                 = a
```

The `lift` operation lifts an object, placing it in the lifted domain. The `lifted` version of `op` works identically to `op` itself for elements of the original domain which have been lifted, but given `Nothing` will do nothing with the remaining value, making `Nothing` the identity of `op`. Finally, `unlift` eliminates lifting; because it does not include a clause for `Nothing`, it captures the fact that a lifted computation which returns `Nothing` has no well-defined value.

A simple example of a monoid without an identity is the monoid of lists with at least one element, which is the same as the ordinary list monoid except that the empty list `[]` cannot be used. The function `reduce1` expresses isomorphisms between this monoid and all others, just as `reduce` did for monoids with identities:

```
reduce1 op xs           = unlift (reduce (lifted op) Nothing (map lift xs))
```

There are also equivalents of `foldl` and `foldr` for lists with at least one element:

```
foldl1 op xs            = unlift (foldl (lifted op) Nothing (map lift xs))
foldr1 op xs            = unlift (foldr (lifted op) Nothing (map lift xs))
```

Thus, anything that can be expressed with *non-empty* lists fits nicely into the framework of the optimizations on *arbitrary* lists.

5.6 Representing collections using monoids

The third class of monoids which can be efficiently represented by lists are the Abelian monoids (which are commutative and have an identity). These have already appeared briefly in the discussion of arrays, and are expressed using the `someOrder` operation. This permits reductions to be performed in the most efficient possible order, without regard to the order in which they occur in the program.

It is worth noting that many interesting data structures can be represented *abstractly* as abelian monoids, even if expressing them this way *concretely* is difficult. One example of this is a bag, which (for the purposes of this discussion) is a collection of objects in no particular order. It is most simply represented as a list and can thus be written as follows:

```
bag             = someOrder
contentsBag     = someOrder
```

That is, a bag is simply a list the order of whose contents is immaterial. More complex sorts of bag can be imagined; for example, Id uses imperativeness to implement a bag whose contents are truly arbitrarily ordered. Nonetheless, the above definitions allow producer/consumer relationships like the following to be written:

```
let products = bag [ producer x | x<-rawMaterials ]
in  reduce consumer empty (contentsBag products)
```

5.7 Representing complex iteration systematically

Another use of the `unfold/synthesize` framework is to systematically represent complex iteration. One example of this was the function `splitting`, which expressed complex tree-structured traversals over a list. This section will examine a problem from parallel computing—blocking and strip mining of arrays—and explain how it can be done implicitly in pH using type classes and systematic list generation.

The problem is to divide an $n \times n$ matrix into k^2 $m \times m$ blocks, where $n = km$, so that the work of processing the array can be divided evenly among k^2 processors. Haskell and pH have arrays which can be indexed by any type which is a member of the type class `Ix`. Thus, the idea is to define a special instance of the `Ix` type which generates array indices in the appropriate manner. For the purposes of the example, assume that the arrays are square, have sides whose length n is divisible by k, and have bottommost index `(0,0)`.

The definition of the datatype `blocked` can be found in Figure 5.4. The interesting function from the figure is `range`. The array is split apart first horizontally and then vertically according to the list `blocks`. The call to `synthesize` to construct `blocks` causes the indices to be splist in half, then each half to be split in half again, and so on. As a result, any computation which uses `range` to obtain the indices of an array (this ought to be every computation on that array) will first divide the computation up into `k`

```
data blocked = B (Int, Int)

k::Int
k = 4   -- If we have, say, 16 processors

instance Ix blocked where
    range (S(_,_),S(n-1,_)) = [S(x+xb,y+yb) | (xb,_)<-blocks, (yb,_)<-blocks,
                                              x <- unfold (>=m) stepper 0,
                                              y <- unfold (>=m) stepper 0]
      where m = quot n k
            blocks = synthesize msized (\_->False) split (0,n)
            msized x = x==m
            split (base,side) = ((base,side2),(base+side+1,side-side2))
              where side2 = quotient side 2
            stepper n = (n,n+1)
    index (S(_,_),S(n-1,_)) S(x,y) = xp*m + yp + bsize * (xb*k + yb)
      where m = quot n k
            bsize = m * m
            (xb, xp) = quotRem x k
            (yb, yp) = quotRem y k
    inRange (S(_,_),S(n-1,_)) S(x,y) = 0<=x && x<n && 0<=y && y<n
```

Figure 5.4: Code to define an implicitly-bolcked array indexing scheme

strips in the x direction, and then divide each strip into k blocks in the y direction. Each of these blocks is then iterated over from lowest to highest index in the y and x directions. The linearization specified by the function index is imposed by Haskell, and groups blocks together; the rows and columns of the array as a whole are thus not stored contiguously in the linearized array. The conceptual notion is that each block will map to storage local to a different processor in a multiprocessor implementation of the algorithm. Using the desugarer, that notion can be expressed in one place, and simply used abstractly anywhere it is needed. This avoids a great deal of effort which would otherwise be necessary to strip mine loops throughout the program, and ensure that these loops remain properly coded whan the problem dimensions or machine size change.

Chapter 6

Related work

The transformations described in this thesis are (as noted previously) based heavily on Gill, Launchbury, and Peyton-Jones's "Short Cut to Deforestation" [6]. That work focuses heavily on the interaction of the `foldr` and `build` operations, just as this work centers around the use of `map`, `reduce`, `unfold` and `synthesize`. Both share the view that lists play a pivotal role in functional programs, and that it is therefore worth spending time and effort on optimizations specifically geared towards them.

The work of Waters [18] suggests that lists need not be used so pervasively; rather, a special abstract notion of a *sequence* can be used exclusively to express iteration. This divorces the data structure (lists) from the abstract concept of iteration, and prevents the programmer from unintentionally mixing them in ways which affect performance. Nonetheless, the constraints of Haskell prevent the integration of a separate sequence type into the compiler, though such an extension can probably be added to the language without tremendous difficulty.

Most of the papers on deforestation focus on general transformations which will work for arbitrary algebraic data types. Wadler's original transformation [17] required that the program be in a special *treeless form*, which meant that it could not be used on arbitrary source code. The precursor to deforestation, the listless machine [14], is even more constrained; its programs must execute in bounded space. The work of Chin [5] allows the compiler to discover when code can be safely deforested, and permits higher-order functions to be used with deforestation by perfoming inlining to obtain first-order code wherever possible.

A number of works on constructive programming proved indispensible in fleshing out the theory behind `map` and `reduce`, and in providing inspiration for the unification of reduction and folding. Chief among these are Bird's lecture notes on constructive programming, which heavily emphasize the use of the list monoid to express computation (though the `reduce` operation is invariably discarded in favor of `foldl` or `foldr` when code is actually run) [1, 2]. The text by Bird and Wadler [3] provides a more pragmatic view of the constructive style, directed as much towards explaining the invention of new abstractions as to exploring the efficient uses of existing ones. An amusingly-titled paper by Meijer, Fokkinga, and Paterson [9] explores general relationships between algebraic types and the recursive functions used to

traverse them; the examples for lists are excellent, and the paper presents structures for traversal which are not covered elsewhere, in particular anamorphisms (such as `unfold`) and paramorphisms (the list paramorphism can capture patterns of traversal similar to `map`, but where the resulting list will also depend on information carried between iterations).

Both the works on constructive programming and the works on mathematical tranformation of functional programs have roots in an early paper by Burstall and Darlington [4]. This paper described general approaches which can be used to optimize functional programs by describing their operations algebraically or inductively, and essentially proving them equivalent to simpler or faster programs. The style of their examples is remarkably similar to that found in the later papers on constructive programming. This early paper was intended to describe how such proof techniques might be automated by *unfolding* uses of a function (similar to inlining) and then *folding* them into new functions.

Finally, the actual implementation of the desugarer—particularly the translation for list comprehensions—is based on previous work on list comprehension compilation. As noted in Chapter 4, the canonical presentation of list comprehension compilation can be found in Simon Peyton-Jones's book on functional language implementation [15]. The original motivation for the work presented here was to transparently encapsulate the behavior of the list and array comprehensions from Id, which were designed and implemented over the course of several years by a large group of people (including Aditya, Nikhil, and Arvind) and are described in documentation put together by Zhou [19]. Finally, the paper by Heytens and Nikhil [7] provided a springboard in its discussion of open lists and comprehensions in a parallel environment.

Chapter 7

Conclusions

By using the functions `map` and `reduce`, a large class of list traversals can be described. The particular choice of these two functions permits parallel list traversals to be cleanly and efficiently described. Higher order functions can be used to express data dependencies between elements as they are traversed. Because `map` and `reduce` together express homomorphisms between lists and other monoids with identities, computations on a large class of abstract data structures can be written using list notation for iteration. This permits parallelization of the corresponding data structures without additional compile-time machinery, and suggests that an abstract coding style coupled with a suitable compiler will permit efficient code to be generated with comparatively little effort.

List traversals cannot be entirely eliminated unless the lists they traverse are described systematically. This can be done using a pair of list generating functions, `unfold` and `synthesize`. Here again, an abstract coding style allows such lists to be used to represent *how* to perform iteration; for example, `synthesize` can be used to block an array into smaller chunks which are then iterated over in a more conventional manner using `unfold`. Used this way, these functions capture the notion of "iteration" in a clean and abstract manner.

The desugarer built based on these principles shows a good deal of promise, though it has not yet realized the practical potential of the approach. List comprehensions and loops, crucial expressive tools in pH, can be expressed as canonical list traversals when they are rewritten by the desugarer. The higher order functions necessary to exploit the expressive power of the transformations can be represented entirely at compile time. Thus, the run-time code will generate additional closures, which would impose a heavy performance penalty. In addition, code for list traversals can be generated using heuristics which are based on the structure of the list being traversed and of the traversal itself. Writing list traversals using `map` and `reduce` therefore allows compilation to be adapted to the structure of the list which being traversed.

7.1 Future Work

The pH desugarer and the transformations upon which it is based are complete and work as planned; nonetheless, there is still a great deal of work to be done to make the new optimizations useful and effective. Only when `map/reduce` optimization is working as planned on ordinary program code can its true effectiveness be judged.

7.1.1 Inlining

The most important step in making `map/reduce` optimization broadly applicable is to allow uesr code to be inlined and optimized. Functions from the prelude can also be most cleanly optimized using the general framework of inlining, rather than building their definitions into the desugarer. This eliminates the numerous engineering problems associated with using built-in functions in a language where identifiers can easily be renamed and redefined.

The basic principle of the code inliner is simple. The programmer annotates the program, indicating that certain objects ought to be inlined; the definitions of these objects are then copied by the compiler wherever they occur. Special provision must be made for recursive (or mutually recursive) objects; in general, it is possible to inline a function too many times, resulting in code explosion. Some care is therefore necessary, since the goal is primarily to inline functions so that their code can be optimized away, not to *increase* the net amount of code in the program.

One tricky engineering problem must be overcome to allow a general code inliner to be used with the desugarer. The difficulty is that the inliner would like to represent its code in the clean intermediate language outputted by the desugarer. However, the desugarer itself needs to perform code inlining *before* it optimizes the code it is producing. This means first that the program must be carefully ordered so that desugared code is available to be inlined when it is needed. In addition, all the "plumbing" necessary to implement the transformations during desugaring will need to be extended to the code inliner (otherwise the desired optimizations will not occur).

These difficulties suggest that the transformations may be better performed as a later, separate stage of compilation. This will be discussed in greater depth below (Section 7.1.4).

7.1.2 Loop Introduction

The eventual targets of the pH compiler are the back ends of existing Id compilers. These compilers have special syntax and semantics for iterative loops quite different from the corresponding semantics for tail-recursive functions. The pH compiler represents loops as specially-structured tail-recursive function calls marked by a "loop" pragma. Though the desugarer attempts to use this structure in the code it generates, no effort has yet been made to tag occurrences of loops, or to guarantee that they are "the right shape". These annotations need to be added.

A more general "loop introduction" phase could prove beneficial; such a phase would recognize *any* tail-recursive function expressible as a loop, and re-structure it accordingly. Because existing Id compilers have a large amount of specialized machinery for compiling loops efficiently, this could have a significant impact on the observed efficiency of the code generated by the desugarer (especially when η-abstraction turns recursion into iteration).

7.1.3 Improving heuristics

While the compiler currently generates good code for list traversals, there will always be room for improvement in the heuristics which do so. At the moment the heuristics are very simple—the most complex ones check to see if traversal functions are higher-order or contain a `case` expression. More complex heuristics may be necessary to prevent the needless introduction of loop constants, or to make more sophisticated choices between tail recursion and forward list traversal. Some generated code can also be specialized based on the execution model or language being used—for example, in a lazy execution of a program, not all calls to functions need to be evaluated, thus permitting some traversals to be "cut" when a result is obtained.

Any changes to the heuristics will need to be evaluated to determine their efficacy. Thus, this work is being deferred until the remainder of the compiler stabilizes. Once the compiler works reliably, and the inliner is in place, the effects of different heuristics on the speed of compiled code can be determined.

7.1.4 Separating desugaring from `map/reduce` optimization

One solution to the problem of integrating inlining, desugaring, and `map/reduce` optimization is not to integrate them at all! Given that the latter two are already integrated in the existing compiler, there are several options. The first is simply to discard the desugarer and modify the original Haskell desugarer to use the naive desugaring rules. Alternatively, the compiler can use the desugarer without any inlining, and generate code which includes a special canonical traversal function. This function could then be used in later stages of the compiler to re-create the internal data structures required for optimization.

In either case, a separate optimization phase can make use of most of the same data structures and loop generation routines as the existing desugarer, and as such would not require a large amount of reengineering. If inlining is incorporated into the compiler, there need only be one or two dependencies between the new transformation phase and the prelude routines; these dependencies could even be made internal, so that renaming problems would not occur.

Turning `map/reduce` transformations into a separate compiler phase would also allow experimentation with more sophisticated optimizations. For example, nested lists can be handled by repeatedly applying the transformations, expanding traversals into loops only on the final pass (or as a separate pass at the end).

7.1.5 Combining multiple traversals

One particularly tricky variation on `map/reduce` optimization is to permit it to combine multiple traversals of the same list. This was described in the original shortcut paper [6], but was not implemented in the Glasgow Haskell compiler at the time. It requires a good deal of record-keeping on the part of the compiler, but can completely eliminate a far larger set of list traversals than would otherwise be possible without heavy inlining. There is a tradeoff between code duplication (a constant risk when using inlining) and loss of iteration (for example, when a list is traversed using both `foldl` and `foldr`, combining thse traversals will yield a recursive function).

7.1.6 Bottom-up `unfold/synthesize` optimization

Instead of implementing list optimization as a top-down process, in which traversals are combined and moved towards the lists which they traverse, the compiler can instead move lists upwards towards the code which traverses them. As noted in Section 3.2, this would require a uniform notion of what it means to generate a list. The chief attraction of upwards optimization is that it can potentially solve the `zip` problem (see Section 5.2) by providing rules to combine systematically generated lists. Some information would still need to be carried downwards, however—for example, in order to combine multiple traversals those traversals need to be "brought together" somehow.

Unfortunately, `unfold` and `synthesize` are not easily unified. Even if `unfold` is expressed as a synthesis, it is still unclear how to generate efficient code for both sorts of list traversal. Moreover, while `synthesize` breaks the construction of a list up into the construction of its sublists, it does not need to do so evenly—a simple example is generating a list from the right or from the left:

```
leftnum  = synthesize (\(l,u)->l==u) (\_->False) (\(l,u)->((l,l),(l+1,u))) (1,10)
rightnum = synthesize (\(l,u)->l==u) (\_->False) (\(l,u)->((l,u-1),(u,u))) (1,10)
```

These widely disparate uses of `synthesize` hint at the difficulty of combining list generation techniques without discarding parallelism. A clean solution will likely yield insight into the structure of parallel iteration, and perhaps suggest new ways of formulating data structures which can be iterated over.

Bibliography

[1] Richard Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[2] Richard Bird. Lectures on constructive functional programming. In *Constructive Methods in Computing Science*, pages 151–216. Springer-Verlag, 1989.

[3] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[4] R M Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24, 1977.

[5] Wei-Ngan Chin. Safe fusion of functional expressions. In *Proc. of the ACM Symposium on LISP and Functional Programming*, pages 11–20, 1992.

[6] Andrew Gill, John Launchbury, and Simon L Peyton Jones. A short cut to deforestation. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, 1993.

[7] Hichael L Heytens and Rishiyur S Nikhil. List comprehensions in AGNA, a parallel persistent object system. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, 1991.

[8] Paul Hudak, Simon L Peyton Jones, and Philip Wadler, eds., et. al. Report on the functional programming language Haskell, version 1.2. *SIGPLAN Notices*, 27, 1992.

[9] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes, and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, 1991.

[10] Rishiyur S Nikhil, Arvind, and James Hicks. ph language proposal (preliminary). Working document describing pH extensions to Haskell.

[11] R.S. Nikhil. Id language reference manual, version 90.1. Technical Report 284-2, MIT Computation Structures Group Memo, July 1990.

[12] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proc. of the ACM Symposium on LISP and Functional Programming*, pages 288–298, 1992.

[13] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, 1993.

[14] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proc. of the ACM Symposium on LISP and Functional Programming*, pages 45–52, 1984.

[15] Philip Wadler. Chapter 7: List comprehensions. In Simon L. Peyton-Jones, editor, *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[16] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. of the Workshop on Implementation of Funtional Languages, pub. as as Chalmers PMG -R17*, 1987.

[17] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1991.

[18] Richard C Waters. Automatic transformation of series expressions into loops. *ACM Transactions on Programming Languages and Systems*, 13, 1991.

[19] Yuli Zhou, ed. List and array comprehension desugaring. Chapter in Id-in-id compiler documentation.