LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Simplifying Parallel List Traversal

Computation Structures Group Memo 370
January 18, 1995

Jan-Willem Maessen

Submitted to FPCA '95

This research was supported in part by an NSF Graduate Fellowship. Funding for the Laboratory for Computer Science is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Simplifying Parallel List Traversal

Jan-Willem Maessen*

Abstract

Computations described using Bird's constructive algebra of lists are nicely amenable to parallel implementation. Indeed, by using higher-order functions, ordered list traversals such as `foldl` and `foldr` can be expressed as unordered reductions. Based on this observation, a set of optimizations have been developed for list traversals in the parallel Haskell (pH) compiler[12]. These optimizations are inspired by, and partially subsume, earlier work on the optimization of sequential list traversal.

1 Introduction

Lists are a basic computational "glue" in many functional languages. They are easily expressed and understood by programmers, and offer a compact notation for representing collections of data. The functional language *Haskell*[7] is no exception to this rule, and in fact encourages or even requires the use of lists to express certain sorts of computation.

Functional languages also encourage a compositional coding style. This causes lists to be used extensively as intermediate data structures. For example (all code is in Haskell[7]):

```
any p = or . map p
or    = reduce (||) False
```

Here the function `any` takes a list, maps `p` across it to produce a list of booleans, and then consumes that list by calling the function `or`.

A simple set of transformations are needed which will combine the generation, traversal, and consumption of a list. One simple and compact transformation which does the job is `foldr/build` optimization[5]. However, the transformation centers around the `foldr` function, which forces results to be combined in a particular order. In *parallel* Haskell, this is a serious problem. In particular, it rules out some important optimizations of the following sort:

```
sum (a ++ b) = sum a + sum b
```

In a parallel language, the above rewriting will permit the expressions `sum a` and `sum b` to be computed in parallel.

*Computations Structures Group, MIT Laboratory for Computer Science (email: jmaessen@mit.edu). This research was supported in part by an NSF Graduate Fellowship. Funding for the Laboratory for Computer Science is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

This paper presents two closely related results. First, a set of local transformations are given which will eliminate intermediate lists. These transformations capture ordering constraints on list traversals by using higher order functions, and do not commit to any particular ordering if none is specified. Second, a set of heuristics are given for transforming the optimized computations into loops. These heuristics are designed to detect and exploit ordering constraints captured by the optimized list traversals. In order to do so, a method for generating lists systematically is introduced into the language.

The optimizations have a number of useful features:

- They embody well-known techniques and theorems from the field of constructive programming.
- They can be used to describe computations on arbitrary data types which are homomorphic to lists or bags.
- Many of the standard list processing functions can be described and optimised within this framework. In particular, list comprehensions are open to optimization if they are desugared using the naive scheme shown in Figure 1[17].

2 The Bird-Meertens Formalism—highlights

The goal of the Bird-Meertens formalism is to provide an algebra with which efficient algorithms can be derived from simple specifications[10]. The goals of such an endeavor are quite broad—roughly speaking, they involve proving complex theorems about this new algebra. The goal of this compiler optimization is much more modest—to consistently improve performance using a set of simple constructive transformations.

2.1 Traversal

In functional languages, there are two commonly-used functions which traverse lists—`foldl` and `foldr`. They have the following mathematical definitions:

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

The parenthesization in the above definitions shows that `foldr` traverses a list "right to left", while `foldl` traverses the list "left to right." These functions correspond respectively to recursive or iterative list traversals.

$$
\begin{array}{ll}
\mathcal{TE}[\![\texttt{[}E \texttt{ | } Q\texttt{]}]\!] & = \mathcal{TF}[\![\texttt{[}E \texttt{ | } Q\texttt{]}]\!] \\
 & \\
\mathcal{TF}[\![\texttt{[}E \texttt{ |]}]\!] & = \texttt{[}\mathcal{TE}[\![E]\!]\texttt{]} \\
\mathcal{TF}[\![\texttt{[}E \texttt{ | } B \texttt{ , } Q\texttt{]}]\!] & = \texttt{if } \mathcal{TE}[\![B]\!] \texttt{ then } \mathcal{TF}[\![\texttt{[}E \texttt{ | } Q\texttt{]}]\!] \texttt{ else []} \\
\mathcal{TF}[\![\texttt{[}E \texttt{ | } P \texttt{ <- } L \texttt{ , } Q\texttt{]}]\!] & = \texttt{concat (map f } \mathcal{TE}[\![L]\!]\texttt{)} \\
 & \qquad \texttt{where f } P \texttt{ = } \mathcal{TF}[\![\texttt{[}E \texttt{ | } Q\texttt{]}]\!] \\
 & \qquad\qquad\quad \texttt{f \_ = []}
\end{array}
$$

Figure 1: Desugaring rules for lists using `map` and `concat`

The problem with these functions is their inherent "handedness". The fold operations impose a particular ordering on the flow of data in the computations they represent. Folded computations must wait for data to reach them along the "spine" of the list. This problem is most evident when the input can be broken up into pieces, say `a` and `b`:

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

This ordering, while sometimes necessary, is often superfluous. For example, if we define the `sum` of a list using `foldl` or `foldr`, we will be unable to show that `sum (a ++ b)` is equivalent to `sum a + sum b`.

What is needed is an operation which describes the traversal of lists symmetrically. Bird calls this operation reduction [1, 2]. It has the following definition:

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

The obvious difference between this definition and the definitions of `foldl` or `foldr` is the absence of parentheses, and the fact that z is used explicitly only to define the value of reduction on the empty list. For `reduce` to be safe, $\oplus$ and z must satisfy the following identities (for any value of a, b, or c of the appropriate type):

$$
\begin{array}{rl}
a \oplus (b \oplus c) \equiv (a \oplus b) \oplus c & \oplus \text{ associative} \\
a \oplus z \equiv z \oplus a \equiv a & z \text{ identity of } \oplus
\end{array}
$$

The above conditions mean that $\oplus$ satisfies the requirements for the binary operation of a monoid with an identity. The conditions also lead to a more rigorous definition of `reduce`:

$$
\begin{array}{lcl}
\texttt{reduce } (\oplus) \texttt{ z (a ++ b)} & = & \texttt{reduce } (\oplus) \texttt{ z a } \oplus \\
 & & \texttt{reduce } (\oplus) \texttt{ z b} \\
\texttt{reduce } (\oplus) \texttt{ z [e]} & = & \texttt{e} \\
\texttt{reduce } (\oplus) \texttt{ z []} & = & \texttt{z}
\end{array}
$$

This definition uses the constructive view of lists: Every list is either empty (`[]`), contains a single element `e` (`[e]`), or can be built by appending two lists, `a` and `b` (`(a ++ b)`).

The advantage of `reduce` is that it avoids overspecification. In particular, if we are attempting to develop a parallel program, we would like to write algorithms in such a way that there are no spurious dependencies forcing our algorithm to be sequential. We write:

```
sum = reduce (+) 0
```

and quickly discover from the definition of `reduce` that:

$$
\begin{array}{rl}
\texttt{sum (a ++ b)} & = \texttt{reduce (+) 0 (a ++ b)} \\
 & = \texttt{reduce (+) 0 a + reduce (+) 0 b} \\
 & = \texttt{sum a + sum b}
\end{array}
$$

For this reason, constructive approaches have been viewed by some as the natural successor to data-parallel programming[15].

2.2 Homomorphisms and Monoids

Unfortunately, using only `reduce` it is impossible to express the following idea:

```
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
length :: [a] -> Int
```

There needs to be some way of transforming individual elements of the list into objects of the correct type, which can then be combined using `reduce`. The operation which accomplishes this is `map`:

```
map f (a ++ b) = map f a ++ map f b
map f [e]      = [f e]
map f []       = []
```

Thus `length` can be expressed in terms of `map` and `reduce` as follows (here `const` is the constant combinator which returns the first of its two arguments):

```
length (a ++ b) = sum . map (const 1)
```

This simply separates the definition of `length` into two pieces—a piece which replaces each list element by 1 (a map) and a sum (reduction) of the resulting list.

Bird [2] notes that `map` and `reduce` can be used in this way to express the homomorphism from the monoid of lists over α, $(\texttt{[}\alpha\texttt{]},\texttt{++},\texttt{[]},\texttt{(:[])})$, to an arbitrary monoid $(\beta, \oplus, id_\oplus, i)$:

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

In effect, every append operation `(++)` is replaced by $\oplus$, every empty list `[]` by $id_\oplus$, and every unit operation `(:[])` by i.

This unconditional replacement gives rise to the *promotion rules* for lists described by Bird [2]. These rules push calls to `map` and `reduce` inwards through `concat`:

$$
\begin{array}{ll}
\texttt{map g (concat } L\texttt{)} & = \texttt{concat (map (map g) } L\texttt{)} \\
\texttt{reduce } \oplus \texttt{ z (concat } L\texttt{)} & = \texttt{reduce } \oplus \texttt{ z} \\
 & \quad \texttt{(map (reduce } \oplus \texttt{ z) } L\texttt{)}
\end{array}
$$

2.3 Higher-order Homomorphism

Having defined homomorphism in terms of `map` and `reduce`, the typical approach (when using constructive techniques to derive an algorithm) is to immediately translate reductions into folds using either of the following (perfectly legal) definitions:

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

This defeats the purpose of introducing `reduce` into a parallel language, however. Instead, it would be desirable if all three methods of list traversal could be fit into a single framework based on reduction.

The trick is to use higher-order functions to separate the behavior of `foldl` and `foldr` into pieces. There are three basic steps:

- Package up the computation which will be performed on a given list element when data arrives along the spine of the traversal. This will transform each element of the list into a function.
- Use a higher-order function to "wire together" the functions along the spine.
- Call the resulting function with the initial value of the traversal.

The first two steps correspond to mapping and reduction respectively. The third step actually performs the computation which has conceptually been set up by the first two steps. The resulting definitions of `foldl` and `foldr` look like this (the operator `(.)` is function composition, and `flip` reverses the order of the first two arguments of the given function):

```
foldr f i xs = reduce (.) id
                      (map f xs)
                      i
foldl g j xs = reduce (flip (.)) id
                      (map (flip g) xs)
                      j
```

It should come as no surprise that `id` (the identity function) is the identity of both monoids. It is, of course, the identity of function composition, but more fundamentally we expect it to represent the idea that "if there is no list element, no computation should be performed."

The structure of the above definitions for `foldl` and `foldr` suggests that a similar formulation for reduction might exist. And indeed it does:

```
reduce ⊕ z xs = reduce (\l r t -> (l z) ⊕ (r z))
                       id
                       (map const xs)
                       z
```

This definition 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. Care must be taken, however, since this means that the higher-order function now being reduced is only correct when its third argument, `t`, is `z`.

The optimizations will therefore assume that all list consumption has the following canonical form:

`(reduce` a `id (map` u `l))` t

Here a is a function of at least three arguments, u a function of at least two, and t represents either the initial value of a fold or the identity of a reduction. For example, for `foldr f z` a =`(.)`, u =`f`, and t =`z`. This allows the optimizer to represent list traversal uniformly and compactly.

2.4 Ordering Rules and Bags

The `map` operation is unaffected by the ordering of its input. However, a reduction does carry some weak ordering information. This is because the operation passed to `reduce` may not be commutative. The most obvious example so far is `concat`. If we write `concat [[1,2],[3],[4,5]]`, we cannot tell which two lists are appended together first; however, we do know that our answer will be `[1,2,3,4,5]` and not, say, `[3,1,2,4,5]` or `[4,5,3,1,2]`. The order independence of `map` and the order dependence of `reduce` are used to derive the following rules:

`map f (reverse` L`)` $=$ `reverse (map f` L`)`
`reduce ⊕ z (reverse` L`)` $=$ `reduce (flip ⊕) z` L

Often, an operation being reduced *is* commutative—or we as programmers may be using lists as bags, meaning that we specifically don't care about the ordering of elements. For these instances, there is an order-independent function called `someOrder`. This function returns a permutation of the list provided to it. In general, it could be defined as the identity function. When `someOrder` is encountered by the optimizer, assumptions about ordering constraints on a canonical traversal are dropped; the optimizer then tries to choose the best possible traversal direction for each sublist of the list being traversed. For example, assume the list `[1..5]` can be computed most efficiently forwards. The optimizations will reorder the list `([1..5] ++ reverse [1..5])` if it is passed to `someOrder`. Thus, a computation which would ordinary yield the list `[1,2,3,4,5,5,4,3,2,1]` after optimization will instead yield the list `[1,2,3,4,5,1,2,3,4,5]`.

2.5 Generating Lists

To entirely eliminate lists, and not simply optimize their traversal, the generation of 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: `[`k_1`,`k_2`..`k_n`]`. In addition, because lists are a full-fledged data structure in the language a partially-constructed list (an "accumulative result variable"[14]) can be traversed by the computations which is constructing the list. Doing so will preclude eliminating the list, since it must be built once so that the repeated traversals will behave as they were written.

Instead, functions can be defined which generate lists systematically. The list anamorphism [11] generates lists one element at a time, using a predicate to terminate generation, so that finite lists can be created systematically. This list anamorphism is visible to the programmer as the `unfold` function, which is defined as follows:

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

The chief virtue of this function is that its structure is regular enough to permit the use of heuristics which will generate different code depending on how the resulting list is consumed.

$$
\begin{aligned}
&\mathcal{LE}[\![\texttt{[}E_1\texttt{,}\ E_2\texttt{,}\ \ldots E_n\texttt{]}]\!] &&= \mathcal{LF}[\![\texttt{[}E_1\texttt{,}\ E_2\texttt{,}\ \ldots E_n\texttt{]}]\!]\ a_{list}\ u_{list}\ t_{list}\\
&\mathcal{LE}[\![E_1\texttt{++}E_2]\!] &&= \mathcal{LF}[\![E_1\texttt{++}E_2]\!]\ a_{list}\ u_{list}\ t_{list}\\
& && \qquad\qquad \textit{etc.}\text{ for other list expressions}\\
&\mathcal{LE}[\![\texttt{reduce}\ f\ t\ L]\!] &&= \mathcal{LE}[\![\texttt{let z = }t\\
& && \qquad \texttt{in } \mathcal{LF}[\![L]\!]\ (\lambda l\, r\, t.\ f\ (l\ \texttt{z})(r\ \texttt{z}))\ \texttt{const t}]\!]\\
&\mathcal{LE}[\![\texttt{foldr}\ f\ t\ L]\!] &&= \mathcal{LE}[\![\mathcal{LF}[\![L]\!]\ \texttt{(.)}\ f\ t]\!]\\
&\mathcal{LE}[\![\texttt{foldl}\ f\ t\ L]\!] &&= \mathcal{LE}[\![\mathcal{LF}[\![L]\!]\ \texttt{(flip (.))}\ \texttt{(flip}\ f\texttt{)}\ t]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{LF}[\![\texttt{[}E_1\texttt{,}\ E_2\texttt{,}\ \ldots E_n\texttt{]}]\!] && a\ u\ t = \texttt{((}u\ E_1\texttt{)}\ \texttt{`}a\texttt{`}\ \texttt{(}u\ E_2\texttt{)}\ \texttt{`}a\texttt{`}\ \ldots\ \texttt{(}u\ E_n\texttt{))}\ t\\
&\mathcal{LF}[\![E_1\texttt{++}E_2]\!] && a\ u\ t = \texttt{(}\mathcal{LF}[\![E_1]\!]\ a\ u\ \texttt{`}a\texttt{`}\ \mathcal{LF}[\![E_2]\!]\ a\ u\texttt{)}\ t\\
&\mathcal{LF}[\![\texttt{[]}]\!] && a\ u\ t = t\\
&\mathcal{LF}[\![A\texttt{:}L]\!] && a\ u\ t = \texttt{(}u\ A\ \texttt{`}a\texttt{`}\ \mathcal{LF}[\![L]\!]\ a\ u\texttt{)}\ t\\
&\mathcal{LF}[\![\texttt{map}\ f\ L]\!] && a\ u\ t = \mathcal{LF}[\![L]\!]\ a\ \texttt{(}u\ \texttt{.}\ f\texttt{)}\ t\\
&\mathcal{LF}[\![\texttt{concat}\ L]\!] && a\ u\ t = \mathcal{LF}[\![L]\!]\ a\ (\lambda l.\ \mathcal{LF}[\![l]\!]\ a\ u)\ t\\
&\mathcal{LF}[\![\texttt{reverse}\ L]\!] && a\ u\ t = \mathcal{LF}[\![L]\!]\ (\texttt{swap}\ a)\ u\ t\\
&\mathcal{LF}[\![\texttt{someOrder}\ L]\!] && a\ u\ t = \mathcal{LF}[\![L]\!]\ (\texttt{bothways}\ a)\ u\ t
\end{aligned}
$$

Figure 2: `map/reduce` optimization rules

3 `map/reduce` optimization

Having seen that traversals can be converted to canonical form, and that maps and reductions can eliminate the intermediate lists in `map` and `concat`, it simply remains to combine everything to obtain a simple set of optimizations. The optimization rules, given in Figure 2, are referred to as the `map/reduce` rules, since the last three arguments to $\mathcal{LF}$ represent a canonical reduction being performed using `map` and `reduce` on the expression in the double brackets. The `bothways` operation represents a marker function which is applied to the value of a to express the fact that a is free of ordering constraints. This fact is exploited in rules which generate code to actually perform the reduction (described in the next section).

This structure also permits the elimination of compile-time redexes which would have been present if optimization simply used the promotion rules directly. In particular, all calls to a are performed at compile time—a safe bet, since a has a regular structure. There is one danger in this: it is possible to duplicate t. Note, however, that this can only happen when a is created by a reduction—and t is given a concrete name in this case.

In addition, the rules are carefully structured so that they deal gracefully with nesting—that is, inner subexpressions are not optimized until reductions have been eliminated in the enclosing expression. This permits `map/reduce` optimization to be performed in one pass; however, functional arguments (all labeled f in their respective desugaring rules) may be re-optimized repeatedly if they are applied repeatedly. Right now a fairly simple heuristic is used to control this behavior: if the function is bound to a name and then used in a traversal it will be optimized once, separately, but will not benefit from further `map/reduce` optimization if it happens to return a nested list. If the function is specified explicitly using a lambda, on the other hand, it will be duplicated (and inlined) *ad nauseam*.

Note that the rules for optimization only explicitly specify re-traversal in three cases—for the three list consumers `foldl`, `foldr` and `reduce`. In the remaining cases, list syntax is actually going to be generated. This syntax would then be re-optimized unless a second set of names is introduced for already-optimized list operations. However, it is possible to avoid re-traversing these operations entirely by carefully traversing inner code as the result is produced. This leads to the following definition for a_{list}, u_{list} and t_{list}:

$$
\begin{aligned}
a_{list} &= \texttt{(.)}\\
u_{list} &= (\lambda e t.\ \mathcal{LE}[\![e]\!]\texttt{:}t)\\
t_{list} &= \texttt{[]}
\end{aligned}
$$

No formal termination proof for this optimization structure has yet been undertaken; informally, $\mathcal{LE}$ always either encounters no list and proceeds deeper, encounters a list which must actually be constructed and proceeds deeper on each element of that list, or encounters a list traversal and eliminates it, re-traversing with one less level of list nesting (since Haskell is strongly typed and types are finite, we will not cut our own throat in this instance).

4 Code for Generation

The extensive use of higher-order functions in canonicalized list traversals immediately begs the question of efficiency. Fortunately, the reductions performed by the above rules alleviate most of these concerns. Reasonable code can thus be generated with the following rules:

$\mathcal{LF}[\![\texttt{unfold}\ P\ F\ V]\!]\ a\ u\ t =$

```
    let h v acc | LE[[P]] = acc
                | otherwise = (a (u e) (h v')) acc
                    where (e,v') = LE[[F]] v
    in  h LE[[V]] t
```

Otherwise

$\mathcal{LF}[\![E]\!]\ a\ u\ t =$

```
    let h []     acc = acc
        h (e:es) acc = (a (u e) (h es)) acc
    in  h LE[[E]] t
```

These definitions were obtained by recognizing that list traversal must, as always, take place sequentially along the spine of the provided list, and that `unfold` similarly must wait for data to be propagated from one iteration to the next. The following sequential traversal is then used:

`(\lst -> foldr (a.u) id lst t)`

```
ℒℱ⟦unfold P F V⟧      a u t = let h v acc | ℒℰ⟦P⟧ = acc
                                          | otherwise = ℒ𝒟⟦h v'⟧⟦acc⟧ a (u e)
                                              where (e,v') = ℒℰ⟦F⟧ v
                              in h ℒℰ⟦V⟧ t
                                                          omitting ℒℰ when u ≠ u_list
ℒℱ⟦E⟧              a u_list t = ℒℰ⟦E⟧ ++ t
Otherwise:
ℒℱ⟦E⟧                 a u t = let h []     acc = acc
                                  h (e:es) acc = ℒ𝒟⟦h es⟧⟦acc⟧ a (u e)
                              in h ℒℰ⟦E⟧ t
```

When a is from a reduction or is `bothways`:

$\mathcal{LD}[\![R]\!][\![A]\!]\ \ a\ \ v = R\ \ (v\,A)$	a =`bothways (.)`, v strict
$\mathcal{LD}[\![R]\!][\![A]\!]\ \ a\ \ v = R\ \ (a\,(\lambda x.\,A)\,v\,A)$	a strict in second argument
$\mathcal{LD}[\![R]\!][\![A]\!]\ \ a\ \ v = R\ \ (a\,v\,(\lambda x.\,A)\,A)$	a =`bothways` a', a strict in first argument

If none of the above apply:

$\mathcal{LD}[\![R]\!][\![A]\!]\ \ a\ \ v = a\ \ v\ \ (R)\ \ A$

Figure 3: Rules for efficiently traversing an existing list or `unfold`

By expanding the definition of `foldr` above, the following code is obtained for the lambda body:

```
let h []     = id
    h (e:es) = (a (u e) (h es))
in  h lst t
```

The important observation which leads us to the code at the beginning of the section is that both disjuncts in the definition of `h` will return a function. In fact, because a has a fixed definition, it is open to η-abstraction—that is, no computation beyond pattern matching will be necessary at run time to obtain the function which is returned by `h`. This η-abstraction step yields the expansion given at the beginning of the section. The translation for `unfold` follows by observing the structural similarity of traversal to generation, or analogously by observing that a catamorphism (`foldr`) can be combined with an anamorphism (`unfold`)[11].

4.1 Exploiting the regular structure of a

The fact that the structure of a is known permits a number of important heuristics to be derived which will produce better code than the naive translation given above. Note that the structure of the traversal function `h` is mostly fixed in the expansion at the beginning of the previous section; the only parts which vary are the value of t (which is outside `h` itself) and the value of the second clause of `h`'s definition. There are two basic forms which this second clause can take (where f and g are arbitrary functions):

f `(h xs acc)`	*proper recursive*
`h xs (`g `acc)`	*iterative*

If there is a choice in the matter, the decision must be based on the strictness of expression g. If g is strict, then the iterative form should be chosen; this will permit the generation of a loop which executes in unit space in the final code for `h`. If g is not strict, however, the proper recursive form should be used. In this way, `h` as a whole can return a value before the recursive call has terminated, and in a lazy language can avoid the recursive call entirely. The rules given in Figure 3 will generate the iterative form whenever possible. These are (modulo some tweaking to eliminate useless arguments) the rules which are actually used in the pH compiler.

Note that the optimization rule for `concat` given earlier (Figure 2) specifies that the arguments to u are used to perform a recursive optimization. Given this fact, how can the strictness properties of such a function be determined? Note that the second argument to u will in this case be passed straight to the recursive optimization as the t parameter of that optimization. The t parameter to the optimization is never itself optimized (unless the code is later re-optimized from the ground up by $\mathcal{LE}$). As a result, the recursive optimization can be performed with some dummy second parameter `d`. We can then evaluate the strictness of the resulting (concrete!) code with respect to `d`. Any further call to v in $\mathcal{LD}$ can then simply bind its argument to `d` and incorporate the already-generated code.

Note also that lists which already exist on the heap are not explicitly re-created on the heap, but are instead passed to the standard append operation `(++)`. When t =`[]`, this rewriting can be optimized to just $\mathcal{LE}[\![E]\!]$, meaning that E is not copied. In particular, this optimization means that the expression `a ++ b` optimizes to `a ++ b` rather than `a ++ (b ++ [])`.

5 Example

Now that the description of the optimization rules is complete, an example is certainly in order. The following code (to sum the elements of a list of lists) generates parallel code when optimized.

```
llsum = sum . concat
```

Inlining `sum` and composition, the following definition is obtained:

```
llsum xss = reduce (+) 0 (someOrder (concat xss))
```

Useful optimization begins when the reduction is encountered (since 0 is a constant it is inlined on the fly for brevity):

```
llsum xss = ℒℰ⟦ℒℱ⟦someOrder (concat xss)⟧
                   (λl r t. (l 0) + (r 0)) const 0⟧
```

Now optimization continues through the call to `someOrder`:

```
llsum xss = ℒℰ⟦ℒℱ⟦concat xss⟧
                   (bothways (λl r t. (l 0) + (r 0)))
                   const 0⟧
```

And on to the `concat`:

```
llsum xss = ℒℰ⟦ℒℱ⟦xss⟧
                   (λe. ℒℱ⟦e⟧
                          (bothways ...)
                          const 0)
                   (bothways (λl r t. (l 0) + (r 0)))
                   const 0⟧
```

Now in order to generate the loop over `xss`, the innermost optimization needs to be performed eagerly so that its strictness can be determined. To do so, use the dummy value `d` for t; the symbol `e` is provided by the translation:

```
ℒℱ⟦e⟧ (bothways (λl r t. (l 0) + (r 0))) const 0 d
```

Now, since addition is strict in both arguments, this inner traversal can be performed iteratively as follows:

```
let h []     acc = acc
    h (e:es) acc = h es (acc + e)
in  h e d
```

Since `h` is strict in `acc`, the original function was strict in its last argument. The following code for llsum therefore results:

```
llsum xss =
  ℒℰ⟦h' xss 0
       where h' []     acc' = acc'
             h' (x:xs) acc' = h' xs (acc' + h x 0)
               where h []     acc = acc
                     h (e:es) acc = h es (acc + e)⟧
```

No further optimization is possible on the body, so the call to $\mathcal{LE}$ will simply disappear.

The important line in this definition is the second disjunct of `h'`. In a system without a notion of reduction, the following code will result for that line:

```
h' (x:xs) acc' = h' xs (h x acc')
```

In the `map`/`reduce` system, the call `h x 0` can proceed before `acc'` is available. Thus, by executing the outer loop `h'` eagerly, the sublists can be computed independently; only the computation of the final loop result need wait for the completion of the inner loops.

5.1 A note on efficiency

The optimization rules given here certainly do not produce the best *possible* code. In particular, consider the following example:

```
sort xs = reduce merge [] (map (:[]) xs)
```

This appears to implement mergesort—individual, sorted sublists are merged together two by two to yield the final sorted result. Note, however, that the above produces the following code after optimization:

```
sort xs = h xs []
  where h []     acc = acc
        h (e:es) acc = h es (merge acc [e])
```

Inlining `sort` at the points where it is used is likely to improve matters somewhat (by merging sorted sublists in one go), but the fact remains that the underlying algorithm is simple insertion sort. This turns out to be a fundamental limitation of the traversal rules. In general, to optimize a reduction something must be known about the behavior of the function being reduced. It is thus extremely difficult to come up with rules analogous to those in Figure 3 which will produce good code both for straight-through and divide-and-conquer algorithms without recourse to some additional annotations. It should be possible, however, to add new functions representing, say, divide-and-conquer reduction.

More fundamentally, the optimizations cannot take a bad algorithm and turn it into a good algorithm. For example, asymptotic time behavior ought to be much the same as it is when reduction is implemented using a fold. This also means that optimization will not make an inherently parallel algorithm sequential.

6 Correctness and Referential Transparency

One important problem has been glossed over in the discussion so far—the correctness of the `reduce` operator itself. Early on it was noted that the first two arguments to `reduce` must be an associative operator and its identity. The compiler, however, has no particular notion of an associative operator. This means that it is perfectly permissible to write the following two expressions (which produce different results even though their arguments are identical in a semantic sense):

```
reduce (-) 0 [1..10]
reduce (-) 0 ([1..5]++[6..10])
```

Because of `map`/`reduce` optimization, reduction is very sensitive to the *way* in which its arguments are computed. This sensitivity disappears when `reduce` is used properly, but can potentially cause hard-to-spot problems in incorrect code.

The `someOrder` operation suffers the same referential opacity as as `reduce`. One way to reason about the ordering of the argument to `someOrder` is to assume it is a function of some (unknown) underlying state. Such reasoning is not particularly useful, however, since that state can only be manifested (through calls to `someOrder`) and not manipulated.

The pH compiler takes the position that `reduce` and `someOrder` are not really functions at all—an eminently reasonable assertion, since the optimization rules treat them quite differently from fully-fledged functions. Instead, the programmer treats them as special syntax which happens to resemble function call. From the standpoint of the rest of the compiler, however, `reduce` and `someOrder` can be treated just like ordinary functions.

7 Applications

The `map`/`reduce` mechanism is used to implement a number of additional features in the pH language. This section briefly examines three such features: `for` loops, array comprehensions, and open lists. All three were implemented as distinct mechanisms in the Id language compiler[13, 20]. One of the chief motivating factors behind the introduction of the `map/reduce` mechanism was to handle these three features in a single uniform manner without sacrificing the efficiency of the original Id language translations. In doing so, much has been learned about the behavior of reduction in the presence of higher-order functions and state.

7.1 `for` Loops

The pH compiler contains several syntactic extensions to Haskell. One of these is the `for` loop. It is expressed as in Id[13], 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*, $n_1, n_2, \ldots n_k$.[1] 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
        next sum = sum + x
        next len = len + 1
     in sum / len
```

A continuation-passing expansion for loops can be defined using `foldr`:

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

This leads to a more concise desugared 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 (next sum) (next len))
           (\sum len -> sum / len)
           xs sum len
```

By using inlining and η-abstracting all arguments which correspond to circulating variables, completely iterative code is generated. Since the functions being folded require at least the k arguments representing the circulating variables before performing any computation, such an η-abstraction step is justified. This results in the following optimized code for `mean`:

```
mean xs =
  let sum = 0
      len = 0
  in  h xs sum len
        where h []     sum len = sum / len
              h (x:es) sum len =
                let next sum = sum + x
                    next len = len + 1
                in h es (next sum) (next len))
```

In general, `map/reduce` optimization combined with constant argument removal and η-reduction are powerful enough to handle a large class of traversals involving higher-order functions—which in turn permit an even larger class of list traversals to be captured efficiently in the language. This three-optimization chain is important enough in the pH compiler that the optimizations have been combined with `map/reduce` optimization into a single pass.

7.2 Array comprehensions

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 pairs as an argument. 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))
```

In pH, this mechanism supplants Id's specialized array comprehensions and accumulators. Id array comprehensions make use of I-structures (which must be explicitly written to at most once) to construct an array in parallel with its use [20]. By defining a list traversal which uses these same techniques, the full performance of Id array comprehensions can be realized. Thus, pH need not trade syntactic compactness and elegance for performance.

The approach used is based on monadic state transformers [8]. However, because an I-structure model is being used, the results of a write operation will be visible throughout the program. This requires that the notion of a state transformer be modified somewhat, but permits write operations to occur simultaneously in several parts of the program without loss of safety. The type `IAT a b` is used to encapsulate the necessary state transformations—a value of this type will partially fill an array indexed by type `a` with elements of type `b`. The following functions are used for the definition of array comprehensions[2]:

```
makeIAT  :: (a,a) -> IAT a b -> Array a b
storeIAT :: a -> b -> IAT a b
         ::
parIAT   :: IAT a b -> IAT a b -> IAT a b
noIAT    :: IAT a b
```

These give rise to the following definition for the function `array` (very similar to the one in [5]):

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

[2] Ignoring the orthogonal problem of linearizing array indices, which is treated explicitly in Haskell.

```
array bounds ivs = makeIAT bounds stores
  where
    stores = reduce parIAT noIAT
                  (map (uncurry storeIAT) ivs)
```

The call to `makeIAT` creates an array with the requested bounds and performs the necessary stores into it. These stores are specified by the `storeIAT` operation (`uncurry` permits it to be used on the index/value pairs `ivs`). The `parIAT` operation states that any pair of transformations may occur safely in parallel, and the reduction thus states that all the stores may happen simultaneously. The identity transformation is `noIAT`, or "do nothing".

Readers familiar with the implementation of state transformers will recognize that they require the use of higher-order functions; these higher-order functions make state available to component computations. The same techniques of constant argument elimination and η-abstraction which are used for `for` loops are used to eliminate extra functions and generate efficient code for arrays.

7.3 Open Lists

List comprehensions in Id make use of open lists to parallelize list contruction. Open lists are represented internally by data structures which are isomorphic to lists, except that their tails are initially empty and can later be filled in using I-structure operations. This permits lists to be constructed eagerly starting from the head and working towards the tail. It also permits sublists to be computed independently and appended in unit time. This eliminates the chief bottleneck in parallel list creation—that sublists must wait for the result to their right to be computed (or, if one is thinking lazily, computation of each sublist must wait for the completion of the sublist to its left). More information on the open list technique has appeared elsewhere[6].

Parallel state transformers can once again be used to implement open lists in pH. Here the state transformation is encapsulated in the type `OL a`, which represents a computation operating on open lists with element type `a`. The following operations are defined:

```
makeOL   :: OL a -> [a]
unitOL   :: a -> OL a
appendOL :: OL a -> OL a -> OL a
emptyOL  :: OL a
         ::
consOL   :: a -> OL a -> OL a
snocOL   :: OL a -> a -> OL a
```

In principle, an open list computation can thus be written as follows:

```
lst = makeOL (reduce appendOL emptyOL
                   (map unitOL (list computation)))
```

There are a number of complications to this apparently simple scheme. First, the use of open lists is *implicit* in the Id compiler for efficiency reasons. A partial solution to this problem is to re-cast `map/reduce` optimization in terms of the open list operations, as follows:

$$\begin{aligned} a_{list} &= \texttt{appendOL} \\ u_{list} &= (\lambda e.\, \texttt{unitOL}\ \mathcal{LE}[\![e]\!]) \\ t_{list} &= \texttt{emptyOL} \end{aligned}$$

This means that uses of the $\mathcal{LF}$ optimization rule in the rules for $\mathcal{LE}$ must now introduce a call to `makeOL`. For efficiency reasons a_{list} also needs to be written to incorporate the following peephole optimizations (where the right hand forms allocate half as much storage as the forms on the left):

```
appendOL (unitOL x) ol = consOL x ol
appendOL ol (unitOL x) = snocOL ol x
```

The final, and most difficult, problem with open lists is the frequency of copying. In the traversal rules, the following rewriting was given for a list which already exists on the heap:

$$\mathcal{LF}[\![E]\!] \quad a \ u_{list} \ t = \mathcal{LE}[\![E]\!] \ \texttt{++} \ t$$

This permitted a copy operation to be omitted when $t =$ `[]`. Unfortunately, preexisting lists and open lists have entirely different types! Consequently, more trickery is necessary to efficiently compile open list computations. At the time of writing there is no clear, consistent solution to this problem beyond limiting (in an *ad hoc* manner) the situations in which open lists are introduced by the compiler. The solution doubtless requires a deeper understanding of the state transformations underlying the open list mechanism.

7.4 Lifting

There are many associative operations which might be usefully used for list optimization, but which do not posess an identity. For these functions there is an operation called `reduce1`. Using the type called `Maybe`, these operations are explicitly lifted and an ordinary reduction is performed as follows:

```
data Maybe a = Nothing | Just a

reduce1 :: (a -> a -> a) -> [a] -> a
reduce1 f = unLift . reduce (lift f) Nothing
  where unLift (Just x) = x
        lift f Nothing y = y
        lift f x Nothing = x
        lift f (Just x) (Just y) = f x y
```

A similar technique can be used to define `foldl1` and `foldr1`. All of these definitions avoid explicitly destructuring the list being traversed, but introduce additional data structures. However, it is often easy to tell at compile time that the list being traversed is non-empty. Standard optimization techniques such as peeling the initial loop iteration and eliminating circulating data structures whose form is known can often produce excellent code for traversals of non-empty lists. However, performance is still poor when the list being traversed is, for example, a list of possibly-empty lists flattened using `concat`.

8 Limitations

The optimizations described so far eliminate lists which are consumed as soon as they are produced. While this encompasses a useful class of list operations and permits data structures (including, as we have seen, arrays and open lists) to be expressed in terms of list comprehensions, there are a number of common patterns of usage which cannot be optimized at the moment.

8.1 Producing a list from a list

During the presentation of the optimization rules, only two functions—map and concat—have been used which given a list will produce a list as a result. These suffice to capture a large range of list processing functions (those which can be expressed entirely using list comprehensions). There are important functions which *cannot* be expressed in this manner, however. One significant example is the function init, which returns all of its argument except for the last element. It can be written in two different ways:

```
init xs = unfold (emptytail xs)
                 (\(x:xs) -> (x,xs)) xs
  where emptytail [_]   = True
        emptytail (_:_) = False
init xs =
  fst . reduce1 (\(bl1,l1) (bl2,l2)->
                       (bl1++l1:bl2, l2)) xs
```

One workaround is to abstract such operations with respect to the unit, append, and empty list elements. This would require the introduction of an operator analogous to build[5], and consequent modifications to the type system. In addition, uses of a build-like operator cannot benefit from the loop-generation optimizations of Figure 3.

8.2 Multiple traversals

Multiple traversals over the same argument list can in principle be combined into a single traversal. Other approaches to list optimization[5] can perform this optimization. There are several difficulties. Two are evident from the following snippet of code:

```
let total = reduce (+) 0 xs
in  foldl (f total) i xs
```

First, while the list xs is consumed twice, there is a potential data dependency between the two traversals. If such a dependency exists, it will be more space-efficient to construct xs in the heap and traverse it twice than to merge the traversals so that the result of the first iteration depends non-strictly on the result of the final iteration. Second, the list xs is traversed in two different ways—using reduce and foldl. If the two traversals can be merged safely, it is not clear that they can be merged *efficiently*. This problem becomes acute when one of the traversals being combined makes use of higher-order functions (as with for loops, open lists, or array comprehensions). By committing to a sequential list traversal, previous approaches have avoided this problem entirely.

8.3 The zip problem

Equally difficult is producing a result by traversing two lists simultaneously. This is expressed using the function zip, which pairs corresponding elements of its two list arguments to yield a single result. Previous solutions to this problem [5] define either a "left-handed" or "right-handed" view of zip—that is, just one of the two arguments is systematically consumed to produce the result.

The zip problem may well be solvable in the map/reduce framework by inverting the direction of optimization—in other words, proceed from the generation or use of a list outwards, combining map and build, until the resulting list is consumed by reduction, folding, or zip. When both arguments to zip have thus been optimized, combine them (combining calls to unfold should not be difficult). Such an approach would be very difficult to integrate with any solution to the previous (multiple traversal) problem, however. In addition, it is unclear how concat should be handled in this framework. Nonetheless, this technique merits further attention.

9 Related Work

The work on which this paper is based first began as an effort to integrate list optimization with list comprehension desugaring. This effort is described in the author's M. Eng. thesis[9]. Its eventual separation into a separate compiler pass was due to engineering problems (the code inliner worked only on desugared code). Nonetheless, it has provided an opportunity to clean up the transformations substantially and to better understand their behavior. This has resulted in a number of improvements, the most obvious being the the positioning of inner optimizations.

The transformations described in this thesis are (as noted previously) based heavily on Gill, Launchbury, and Peyton-Jones's "Short Cut to Deforestation" [5]. That work focuses heavily on the interaction of the foldr and build operations, just as this work centers around the use of map, reduce, and unfold. 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. This work in turn has its roots in work on deforestation[18, 16, 4].

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 Meertens' seminal paper[10] and Bird's lecture notes on constructive programming, which heavily emphasize the use of the list monoid to express computation[1, 2]. The text by Bird and Wadler[3] provides a more pragmatic view of the constructive style. An amusingly-titled paper by Meijer, Fokkinga, and Paterson[11] 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). As noted earlier, [15] compares the expressive power of the Bird-Meertens formalism favorably to other models of parallel computation, suggesting that it is a fitting generalization of data parallel techniques.

In spirit, using higher-order functions to "plumb together" related computations has become an important part of every functional programmer's bag of tools. Such techniques permit details of state and ordering to be reasoned about; this observations is at the heart of monadic programming techniques[19]. One subgoal of the current work is to form a functional view of imperative concepts from dataflow languages, such as I- and M- structures[13], by adapting state transformers[8] to new patterns of usage.

The original motivation for the work presented here was to transparently encapsulate the behavior of the list and array comprehensions from Id[13], 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[20]. The paper by Heytens and Nikhil[6] provided a springboard in its discussion of open lists and comprehensions in a parallel environment. The programming techniques described in this thesis are an integral part of pH, and are described in the language report[12].

Acknowledgements

This work, and the entire pH effort in general, would never have been possible without the generous cooperation of Lennart Augustsson, whose `hbcc` compiler supports the syntactic extensions of pH. Additional thanks must be given to Arvind, who originally prompted this work at the beginning of the pH effort, and who has kept things heading in the right direction. Finally, this work has benefited greatly from comments and conversations with many people both at MIT and elsewhere, whose ideas have suggested numerous improvements and clarifications in the work leading to this paper.

References

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

[5] 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.

[6] 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.

[7] 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.

[8] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 24–35. ACM SIGPLAN Notices, 1994.

[9] Jan-Willem Maessen. Eliminating intermediate lists in ph using local transformations. Master's thesis, MIT, May 1994.

[10] Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In *Mathematics and Computer Science: Proceedings of the CWI Symposium, Nov. 1983*, pages 289–334. Springer-Verlag, 1986.

[11] 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.

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

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

[14] 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.

[15] D B Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2), 1991.

[16] 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.

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

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

[19] Philip Wadler. The essence of functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.

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