Implementing Type-Based Deforestation

by Marie-Christine (Kirsten) Chevalier

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor A. Aiken
Research Advisor

(Date)

* * * * * * *

Professor R. Fateman
Second Reader

(Date)

Contents

Chapter 1

Introduction to Deforestation

Deforestation is a program transformation to eliminate trees. It is usually applied to lazy functional languages, where it is important to efficiently compile common programming idioms that, if compiled naïvely, allocate excessive amounts of space at runtime. In this report, we discuss some of the previous approaches to deforestation, followed by the type inference algorithm for deforestation, which is our focus. After outlining the algorithm, we discuss our implementation of it in the Glasgow Haskell Compiler and the experimental results. We found that type-based deforestation decreases the amount of memory allocated by a significant percentage of benchmark programs. We conclude that type-based deforestation can be effective when applied to real Haskell programs.

1.1 What Is Deforestation?

In functional languages, and particularly in lazy functional languages, programmers often structure their programs as collections of many small functions that communicate with each other by means of intermediate data structures. The advantages of this style of programming have been touted by advocates of lazy functional languages: elegance, improved program understandability and maintainability resulting from modularity, and separation of control flow from essential computation [Hug89]. On the latter point, consider writing a function that constructs a list of numbers from 1 to n. In a strict language, this would have to be done using a loop or recursion that checks whether the upper limit n has been reached, and increments the index variable on each iteration, as in this Haskell function:

```
intsTo n = intsFromTo 1 n
 where intsFromTo m n =
   (if (m == n) then
     []
   else
     (m:(intsFromTo (m+1) n)))
```

The essential computation (consing m on to the front of the list) is intermingled with control flow code (the if statement and the recursive call to `intsFromTo`. In contrast, the following equivalent Haskell function is more typical of the lazy language approach:

```
intsTo n = take n ints
```

```
  where ints n = n:(ints (n+1))
```

Here, the control flow, encapsulated in the call to `(take n)`, is separated from the basic computation, performed by the auxiliary function `ints`. This definition of `intsTo` is more concise and easier to understand than the first one, and these differences are even more noticeable on a larger scale. However, the second definition constructs an intermediate data structure: the list `ints`. Of course, the entire infinite list is not constructed, but as each element of the list is demanded by the call to `take`, a new list cell is constructed and its contents are copied into the result list, the list that is returned as the result of the call to `intsTo`. Allocating memory for these list cells takes time and resources, particularly since they need to be garbage collected later. The first definition only creates one list: the list it returns as its result.

For a slightly more complicated example, consider writing a function to return the sum of the squares of the numbers between m and n. A straightforward recursive definition is as follows:

```
sos m n =
  if (m == n) then
     0
  else
     (square m) + (sos (m+1) n)
```

A more typical Haskell definition:

```
sos m n = sum (map square [1..n])
```

(The Haskell syntax `[m..n]` is syntactic sugar for a call to the function `enumFromTo` with arguments `m` and `n`, which constructs the list of integers from `m` to `n`.) This definition is more concise and easier to understand, as it uses existing higher-order functions (`map`, and `foldr`, which is used in the definition of `sum` and which we will examine more closely later). However, it creates two intermediate lists: `[1..n]` and `(map square [1..n])`, which is strange for a function that solves a problem having nothing to do with lists on the surface. Actually, laziness means that neither list is constructed explicitly all at once – each cell is created when it is demanded by `sum` or `map`, then freed – but again, the extra work necessary to allocate and garbage-collect these structures still makes a significant difference in performance. In fact, the first definition doesn't execute in constant space either, because of the recursive call to `sos` in non-tail position, but it is easy to see how to write a tail-recursive equivalent, whereas this doesn't apply to the second definition, since it contains no explicit recursion.

We would like the compiler to automatically transform definitions like the second definition of `sos` into a form closer to the form of the first definition above. An omniscient compiler could optimize either of these definitions even further by replacing the recursive computation with the non-recursive formula for the sum of the squares between `m` and `n`. However, this goes beyond the scope of the kinds of optimizations most compilers attempt to do, so we settle for a definition that, while not optimal, allocates less memory than the original does.

Deforestation, a program transformation to eliminate trees, can achieve this goal of automatically transforming modular code into monolithic code. As originally stated, deforestation can be applied to eliminating arbitrary data structures defined by Haskell-style algebraic datatype declarations, but most work on deforestation has concentrated on eliminating lists, and list deforestation will be the focus of this report.

1.2 Previous Work On Deforestation

Deforestation was originally described by Wadler in [Wad90], in which he coined the term "deforestation" for a program transformation to eliminate trees. As originally presented, deforestation extended on Wadler's "listlessness transformer", [Wad85], a more limited deforestation algorithm that handled functions that manipulate lists rather than arbitrary trees. This work in turn can be traced back to the more general program transformation framework described in [BD77]. Most work on deforestation has been done in the context of pure, lazy functional languages (usually, Haskell), with a few exceptions.

1.2.1 Deforestation in Other Languages

There are two notable techniques, both of which predate most of the work on deforestation *per se*, that are similar to deforestation but apply to strict and impure languages. The first concerns the efficient compilation of operations on arrays, the principal aggregate datatype in APL. [GW78] and [BT84] describe how compositions of operators drawn from a limited subset of APL's array-transforming operators can be transformed into a single operator, avoiding multiple traversals of the same array. This work was more restricted than deforestation: it could only handle compositions of a few built-in functions, and each of these functions had to both consume and produce an array (unlike deforestation, in which functions may consume a list but produce a value of a different type, and vice versa). Nevertheless, it was practical (exhibited by the fact that it was implemented in working APL compilers), and it is likely that most array expressions in APL were simple enough to be optimized by even this restricted algorithm.

A somewhat more general technique is that of series expressions, which Waters discusses mainly in the context of Lisp, though a prototype version was also implemented for Pascal [Wat89]. A major advantage of this approach over all the other deforestation methods we consider here is predictability: the compiler guarantees that all compositions of functions operating on the "series" datatype (analogous to the list type) will be deforested as long as certain constraints are satisfied, and warns the user about series expressions that cannot be optimized. This is accomplished by adding the series as a new datatype, isomorphic to but distinct from lists (except that they are evaluated lazily), and providing a set of useful operations for series (similar to the usual higher-order list functions provided by Haskell). Making series a distinct type allows a fairly simple, dataflow-based algorithm for deforestation to be employed. It seems advantageous for the user to receive feedback from the compiler about which expressions are and are not deforested, but on the other hand, creating a new type that must be used wherever the programmer wishes deforestation to occur has disadvantages. First, deforestation can never be performed unless the programmer writes her program in the specific style required for deforestation. Second, series must be coerced into lists whenever the programmer wishes to use existing list operations on them, and this may entail some inefficiency. Writing programs in the style required for deforestation does not seem to be a very large burden; the restrictions it places on program form are similar to those imposed by other deforestation techniques, though here they are expressed in a somewhat different way. Essentially, series must be consumed and produced in a uniform way, and series arguments to functions may not be used more than once. These are analogous to Wadler's restrictions (see Section 1.2.2), but notably, Waters lifts the restriction that an

expression which is to be deforested must consist of compositions of functions that do not themselves create intermediate data structures. This work has been implemented as a library for Common Lisp [Ste90], and has been used in developing practical applications in Lisp.
It is interesting to contrast these two techniques with the work on deforestation in Haskell. Both techniques are more limited, but have greater practical use to their credit – only recently has deforestation been implemented in a usable and practical way in a Haskell compiler [PJTH01]. They must impose certain restrictions due to their target languages, APL and Lisp, being impure – the APL work is limited to compositions of operators drawn from a set of non-side-effecting functions, and the Lisp work uses a special type to ensure the correct deforestation of side-effecting expressions. In addition, both approaches are limited to linear data structures rather than general trees. However, both approaches seem to be well-suited to handle the most common programming idioms in their respective languages, and are similar to shortcut deforestation for Haskell (see Section 1.2.3) in that the simplicity of their implementations relies on programmers' use of a predefined set of combinators.

1.2.2 Wadler's Algorithm

Wadler's algorithm attempts to handle programs like the modular definition of `sos` given above that consist of compositions of several functions that consume or produce lists (actually, the algorithm applies to general tree-shaped data structures, but to simplify the presentation we will restrict it to lists here). More specifically, these functions must be in *treeless form*, and fortunately, the functions `sum`, `map`, and `enumFromTo` can all be defined in this form.
Treeless form can be defined for a simple language consisting of variable references, data constructor applications, function applications, and `case`-expressions as follows: A term is treeless if:

- Every argument of a function application and every scrutinee of a `case`-expression is a variable.
- All variables are linear – that is, used only once in the function where they are defined.

The first restriction ensures that the definition creates no intermediate data structures (function or constructor applications, which create data structures, may only appear as the result of the function – such an expression being examined by `case` or passed to another function implies creating a data structure that is not part of the result). The second restriction ensures that inlining functions, an essential step of the deforestation algorithm, will not duplicate work.
Wadler's algorithm can transform any composition of treeless functions to a single, treeless function. The algorithm consists of a number of transformation rules, but the most important one is the `case`-elimination rule, as follows:

```
case (c t_1 ... t_k) of
  (c_1 v_1..v_n) -> e_1
  [...]
  (c_n v_1..v_n) -> e_n

   ===>
```

```
e_i[t_1/v_1, ... , t_k/v_n]
  where c = c_i
```

In words, this rule takes a `case`-expression where the scrutinee is an explicit application of a data constructor to some arguments. The `case`-expression has several alternatives, each corresponding to a particular data constructor in the type that `c` constructs. In this situation, we already know which data constructor constructs the scrutinee of the `case`-expression, so we can eliminate the `case` and replace it with the body of the relevant alternative, substituting the constructor arguments for the variables introduced by the `case`-pattern. This rule performs deforestation: it eliminates an application of a constructor (`c`), which previously created an intermediate data structure.

The other transformation rules all attempt to get the program into a form where this rule can be applied. For example, another rule performs inlining, or replacing a function call with the function's body, with the function's arguments substituted for the formal parameters. This illustrates the need for the linearity requirement as part of the definition of treeless form: under call-by-name evaluation, inlining a function that uses one of its arguments more than once will duplicate work if the expression passed in for said argument is used more than once. Inlining is necessary in order to bring function definitions together with their arguments: for example, without inlining, we wouldn't be able to deforest the sum-of-squares program, `(sum (map square (enumFromTo [1..n])))`. There are no explicit `case`-expressions in the program as is, but if the definitions of `sum`, `map`, and `enumFromTo` are inlined, then opportunities to apply the `case`-elimination rule will appear.

Wadler's algorithm has not been widely used in practice, due partly to its restrictiveness. For example, the definition of treeless form given above excludes the definition of `enumFromTo`, because it creates an intermediate "tree" – `(m+1)`! It is possible to modify the definition so that certain data types, such as integers, are not considered "trees" and so can be created by treeless functions, but this only complicates things more. Moreover, there are compositions of functions we wish to deforest where the functions cannot or are not expressed in treeless form (Wadler's algorithm doesn't attempt to convert function definitions that are not in treeless form, but could be rewritten in treeless form, into treeless form). Also, it is difficult to implement the algorithm in a way that guarantees termination. However, Wadler's algorithm has served as inspiration for further research on deforestation.

1.2.3 Shortcut Deforestation

Whereas Wadler's algorithm attempts to eliminate any list structures that are examined by `case`-expressions, it is possible to simplify the approach by observing that many functions that consume lists are defined in terms of the function `foldr`. Figure 1.1 illustrates the behavior of `foldr`.

In this example, the application of foldr to the function `(+)`, the starting value 0, and the list `[3, 4, 5]` replaces all the cons (`(:)`) constructors in the list with the function `(+)`, and all the nil (`[]`) constructors with the value 0. Evaluating the resulting expression gives $3 + 4 + 5 = 12$. In general, `foldr` consumes a list and produces an expression of type β, given a function argument of type $\alpha -> \beta -> \beta$, a nil value of type β, and a list of type $[\alpha]$.

The shortcut rule [GLP93] is inspired by this observation about `foldr`. On first glance, it seems easy to deforest applications of `foldr`. Since the first two arguments of `foldr`, `k` and

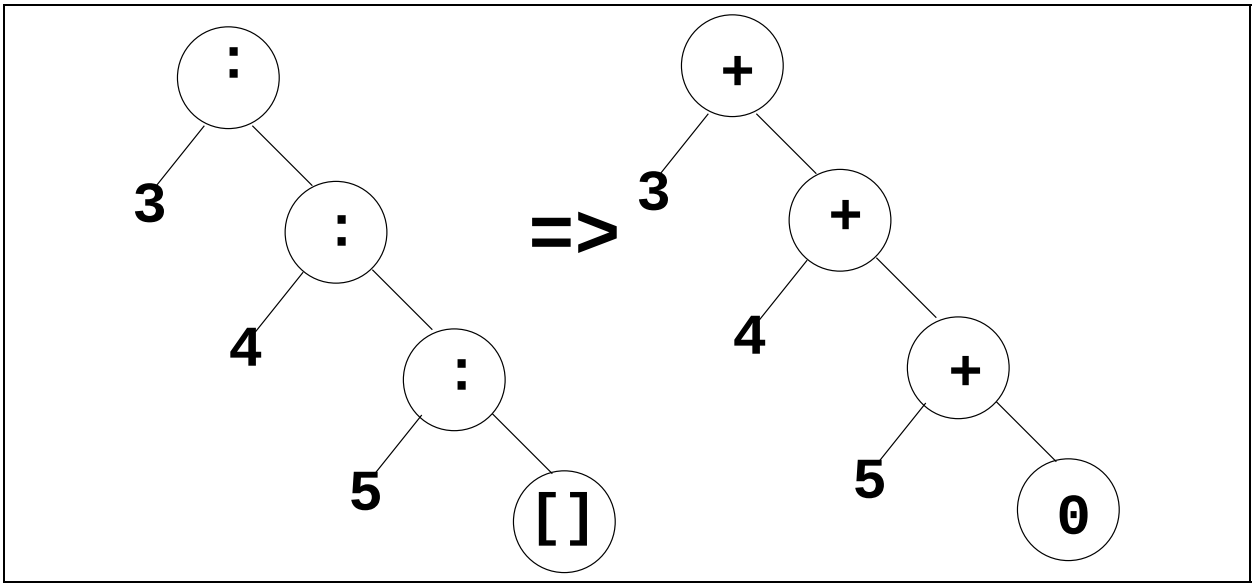

Figure 1.1: `(foldr (+) 0 [3, 4, 5])`

`z`, replace `(:)` and `[]` in the list expression, if we see an application `(foldr k z e)`, we can just replace every occurrence of `(:)` in `e` with `k` and every occurrence of `[]` with `z`. Indeed, this works in simple cases (in the following example, the constant list is written using explicit `(:)`s and `[]`s rather than Haskell's syntactic sugar for lists):

```
foldr (+) 0 (1:(2:(3:[])))
   ==>
(1 + (2 + (3 + 0)))
   ==>
 6
```

which eliminates the intermediate list `[1,2,3]`. However, consider another example (`++` is the function which appends two lists):

```
foldr (+) 0 ([1,2] ++ [3,4])
  ==> (inlining (++) and simplifying)
foldr (+) 0
   (let append xs ys =
       case xs of
         [] -> ys
         (x:xs') -> x:(append xs' ys) in
      append [1,2] [3,4])
  ==>
let append xs ys =
   case xs of
     [] -> ys
     (x:xs') -> x + (append xs' ys)
```

This code is not even type-correct, since it returns a list value in the `[]` case alternative, but an integer in the other alternative. The problem is that `(++)` constructs its result

partially from list constructors in its second argument list, `ys`, and not just from explicit list constructors. Consequently, in order to create a sound rule for deforestation using `foldr`, we must introduce some extra syntactic complications.
Any list-producing function that constructs its result using only explicit list constructors can be written in terms of "abstract constructors" – for example, the recursive definition of `map`:

```
map f xs =
  case xs of
    [] -> []
    (y:ys) -> (f y):(map f ys)
```

can be written with two extra arguments, `c` and `n`, as:

```
map c n f xs =
  case xs of
    [] -> n
    (y:ys) -> (c (f y) (map c n f ys))
```

The type of this version of `map` is `forall a b c . (c -> b -> b) -> b -> (a -> b) -> [a] -> b`. Thus, it can construct a value of any type, depending on the types of the `c` and `n` it is applied to.
To complete the picture, we define a function `build`, which takes a "generalized list producer" like the second definition of `map` given above, and applies it to `(:)` and `[]`, in effect transforming a generalized list producer back into a regular list producer:

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

Now we can finally define the shortcut rule:

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

The list eliminated is the list constructed by `(build g)`: the result of the `foldr` is now constructed directly by `g`, without the intermediate list resulting from the application of `build`.
Though the shortcut rule is simple, its practical effectiveness depends on other transformations, such as inlining the bodies of functions that use `foldr` and `build` in order to bring `foldr`s and `build`s together. Moreover, it depends on programmers' willingness to define their list-producing functions using `build`, which is likely to be low, since `build` serves no purpose except to help out the deforestation algorithm by pointing out which list constructors construct the result list. The shortcut rule has been effective when commonly used library functions are defined in terms of `foldr` and `build`; [PJTH01] describes the implementation of the shortcut rule that is currently used in the Glasgow Haskell Compiler, based on rewrite rules. However, it is unlikely that shortcut deforestation by itself could ever be effective for deforestating compositions of user-defined functions.

1.2.4 Warm Fusion

In [LS95], Launchbury and Sheard present a method for automatically transforming arbitrary recursive definitions into foldr/build form, a technique they call "warm fusion" ("fusion" is

another name for deforestation). Like Wadler's algorithm, warm fusion is applicable to arbitrary data types (since versions of `foldr` and `build` can be expressed for any algebraic datatype), but we concentrate on list fusion here. The algorithm for warm fusion consists of two stages. In the first stage, function definitions are rewritten using `foldr` and `build`. Given a definition of a list-producing function `f`, this rewriting is easy to do:

```
f = \ l -> body  -- where body is some expression containing l
  ==>
f = \ l -> build (\ c n ->
                     foldr c n body[(foldr (:) [] l)/l])
```

First of all, the entire function body is changed into a generalized list producer, which is easy to do: the existing function body is a list, so we just apply `(foldr c n)`, where `c` and `n` are the arguments that replace the list constructors, to the list. We apply `build` to the resulting expression so that the function definition still returns a list-valued expression. Finally, in the original body of the function, we substitute `(foldr (:)  [] l)` for any occurrences of the list argument `l`. (`(foldr (:)  [])` is equivalent to the identity function on lists.) The hope is that when we apply the next stage of warm fusion, the `foldr`s and `build`s we introduced here will meet with other `foldr`s and `build`s and cancel by the shortcut rule.
In addition, certain list-consuming functions are also automatically rewritten in terms of `foldr`. This only works for functions in which the outermost `case` expression deconstructs some list-valued argument. Since many functions that consume lists are expressed in terms of pattern-matching, with one case for the empty list and another case for a list consisting of an element consed onto a list, the method is widely applicable. More precisely, it applies to some definitions which recurse uniformly over their list arguments. Essentially, the outermost `case` is replaced with an application of `foldr` whose first two arguments correspond to the `(:)` and `[]` alternatives of the `case`. For example, the recursive definition of `map` is transformed as follows:

```
map = \x . \f.
  case x of
    [] -> []
    (y:ys) -> (f y):(map ys f)
   ==>
map = \x. \f. foldr ((\ x1 x2 f -> (f x1):x2) f) [] x
```

Now that list-consuming functions are expressed in terms of `foldr` and list-producing functions are expressed in terms of `build` and `foldr`, fusion can proceed. In particular, compositions of functions that are defined in terms of `foldr` can be fused together into single functions, and using a system of rewrite rules, `foldr`s are brought together with `build`s and the shortcut rule is applied.
Warm fusion is quite complicated and expensive to implement, and is not widely used in real compilers.

1.2.5 Vanish Combinators

In [Voi02], Voigtlander presents an approach to deforestation which extends the shortcut rule to deforest certain combinations of list functions, essentially by treating these functions

as additional constructors for the list datatype. Since this method is closely related to the type inference algorithm, we will defer discussion of it until Chapter 2.

1.2.6 Destroy/Unfoldr

There are some list functions that cannot be expressed in terms of `foldr` and `build`. Among these, of course, are functions that do not express patterns of regular recursion over lists. But there are other such functions as well. For example, consider the function `zip`, which takes two lists `[x1,..,xn]` and `[y1,..,yn]` and returns the list `[(x1,y1),..,(xn,yn)]`:

```
zip xs ys =
   case xs of
     [] -> []
     (z:zs) -> case ys of
                 [] -> []
                 (z':zs') -> (z,z'):(zip zs zs')
```

We would like to deforest expressions where `zip` is the list consumer, such as `(zip [1..n] (map square [1..n]))`. However, `zip` cannot be defined in terms of `foldr`, since it does not express a pattern of recursion over a single list. On the other hand, it can be expressed in terms of two functions called `destroy` and `unfoldr` [Sve02].
The function `unfoldr` constructs lists in a way analogous to the way in which `foldr` consumes lists. It takes a function `f` and a starting value `b`, and computes each successive element of the result list by applying `f` to the previous list element. `f` actually returns a value of type `Maybe`: either it returns `(Just (a,b'))`, signifying that `a` should be consed on to the result of recursively calling `unfoldr` on `b'`, or `Nothing`, signifying the end of the list. The definition follows:

```
unfoldr f b =
  case f b of
    Nothing -> []
    Just (a,b') -> (a:(unfoldr f b'))
```

For example, the `enumFromTo` function can be defined this way:

```
enumFromTo m n =
  unfoldr (\ x -> (if (x < n) then
                     Just (x,x+1)
                   else
                     Nothing)) m
```

The function `destroy`, like `foldr`, consumes lists in a regular fashion. It takes a function argument `g` and a list argument `xs`, but the function `g` cannot examine `xs` directly. It can only do so via a helper function that is supplied to it:

```
destroy g xs = g listpsi xs
  where listpsi [] = Nothing
        listpsi (x:xs) = Just (x,xs)
```

For example, `sum` can be defined this way:

```
sum xs = destroy sumDU xs
   where sumDU listpsi xs =
                case (listpsi xs) of
                  Nothing -> 0
                  Just (x1,xs1) -> x + (sumDU listpsi xs1)
```

Certain functions, such as `zip`, can be defined in terms of destroy/unfoldr but not in terms of foldr/build. Other functions can be defined in terms of either. There is a fusion rule for destroy/unfoldr analogous to the shortcut rule:

```
destroy g (unfoldr psi e) ==> g psi e
```

Much as the shortcut rule eliminates the intermediate list constructed by `build`, this rule eliminates the intermediate list constructed by `unfoldr`.
Like shortcut deforestation, this approach has the problem that programmers may not write their functions in terms of `destroy` and `unfoldr`, to an even greater extent, since `foldr` is a commonly used function, but `destroy` and `unfoldr` are little-known. Moreover, it is unclear how to combine the foldr/build rule with the destroy/unfoldr rule, since some functions can be defined either way, and it is unclear how to choose which one. This approach may be effective if library functions are defined in terms of `destroy` and `unfoldr`, but we still lack an effective approach for deforesting compositions of user-defined functions. The type inference algorithm for deforestation is effective for deforesting compositions of user-defined functions.

Chapter 2

The Type Inference Algorithm for Deforestation

2.1 Inlining and Deforestation

All of the deforestation algorithms discussed in the previous chapter require some amount of inlining to be effective. However, inlining is a "black art"; when implemented in a practical compiler, many subtle heuristics are involved in determining whether to inline a given function (see for example [PJM99]). If inlining is performed indiscriminately, the size of the resulting program can blow up exponentially, resulting in worse performance. Moreover, inlining certain expressions indiscriminately can result in work duplication, so the compiler must be careful to conservatively determine which expressions may be used more than once. For the purposes of – for example – shortcut deforestation, we would like a method for inlining which inlines exactly those functions which need to be inlined in order for `foldr`s to meet `build`s. The type inference algorithm for deforestation addressed this issue, and was originally motivated by the problem of inlining. Inlining is integrated with the deforestation algorithm itself, and both make use of the same type information.

2.2 List Abstraction

The algorithm that follows was originally presented by Chitil in [Chi99]. It is based on the observation that many list-consuming functions are already defined in terms of `foldr`, so it is less important to derive `foldr` forms for list consumers than `build` forms for list producers. The basic idea is to replace the list constructors in a function definition (assuming the function produces a single list) with variables `c` and `n`, and turn the definition into an abstraction over `c` and `n`, then applying `build` to it so it still produces a list. However, this cannot be done indiscriminately, since the function may not construct its result by using explicit list constructors (if it does not, then we cannot transform it into a generalized list producer without performing other transformations first), and not all list constructors in the function necessarily contribute to the result. Type inference can determine whether it is possible to transform the function into a generalized list producer, and which constructors should be abstracted over if so.

We will illustrate the list abstraction algorithm with an example. Consider the following

definition of `map`, written with explicit type annotations on all variables:

```
map :: forall a b . (a -> b) -> [a] -> [b]
map @a @b (f::(a -> b)) (xs::[a]) =
  case (xs::[a]) of
    ([]::[a]) -> []
    ((y::a):(ys::[a])) -> (f y):(map @a @b f ys)
```

We replace every list type with a new type variable, and every list constructor with a new term variable:

```
map :: forall a b . (a -> b) -> t1 -> t2
map @a @b (f::(a -> b)) (xs::t3) =
  case (xs::t4) of
    ([]::t5) -> n
    ((y::a):(ys::t6)) -> (c (f y) (map @a @b f ys))
```

Performing type inference on this expression reveals that although `t1` is really the same as `[a]` due to the `case` expression, `t2` is unconstrained, and hence this definition can be turned into a generalized list producer. Moreover, `n` and `c` have types `t2` and `b -> t2 -> t2`, respectively. (The details of type inference are explained later.) So we can construct the following definition of `map`:

```
map :: forall a b t . (b -> t -> t) -> t -> (a -> b) -> [a] -> t
map @a @b @t (c::(b -> t -> t)) (n::t) (f::(a -> b)) (xs::[a]) =
  case (xs::[a]) of
    ([]::[a]) -> n
    ((y::a)::(ys::[a])) -> (c (f y) (map @a @b @t c n f ys))
```

We can apply `build` so that it has the same type as before:

```
map :: forall a b . (a -> b) -> [a] -> [b]
map @a @b (f::(a -> b)) (xs::[a]) =
  build (\ @t (c::(b -> t -> t)) (n::t) ->
    case (xs::[a]) of
      ([]::[a]) -> n
      ((y::a)::(ys::[a])) -> (c (f y) (map @a @b @t c n f ys))
```

This is the basic idea of list abstraction. More complicated cases are addressed later in this chapter.

2.3 Type-Based Inlining

In some cases, a function constructs a list by calling another function. This prevents list abstraction, as described above, from occurring. For example, consider the following function, which constructs the list of squares from 1 to `n`:

```
squares n = map square (enumFromTo 1 n)
```

It is obvious that list abstraction fails in this case, because type inference determines that the body of `squares` has type `[Int]`, based on the type of `map` in the global type environment. We need to inline the bodies of `map` and `enumFromTo` in order to perform list abstraction on the body of `mapsquares`. We accomplish this by replacing list types with new type variables in the types of `map` and `enumFromTo` in the type environment. In order to do this properly, we must use versions of `map` and `enumFromTo` that are specialized to type `Int` (the reason for this is explained in the section on the type inference algorithm). So we perform list abstraction on the following definition of `squares`, with the following bindings in the type environment:

```
mapInt :: (Int -> Int) -> t1 -> t2
enumFromTo :: Int -> Int -> t3

squares :: Int -> t4
squares (n::Int) = mapInt square (enumFromTo 1 n)
```

Type inference determines that `t4` is unconstrained (and equivalent to `t2`, the result type of `mapInt`. However, we can't just construct the list-abstracted definition of `squares` as is. The fact that the type of `mapInt` given by the type environment contains a free type variable, `t2`, that was not unified with a list type, tells us that `mapInt` needs to be inlined. (Note that `enumFromTo` does not need to be inlined, for its result type `t3` was unified with a list type because its result is passed to `mapInt`, which takes a list.) A future inlining pass will inline the body of `mapInt` here, giving (after simplification):

```
squares n = (let mapInt xs =
                 case xs of
                   [] -> []
                   (y:ys) -> (square y):(mapInt ys) in
                 mapInt (enumFromTo 1 n))
```

When list abstraction is applied to this definition, it will successfully create a generalized list producer. Moreover, we have avoided inlining the body of `enumFromTo`, since doing so is not necessary to abstract over the lists in the definition of `squares`.
The original type-based deforestation algorithm, as presented in [Chi99], is as follows:

- Find all expressions of the form `(foldr k z p)`.
- Apply list abstraction, as detailed above, to `p` in each such expression, giving an expression on the form `(foldr k z (build ( c n -> ...)))`
- Apply the shortcut rule in each such case.

This algorithm works, but may still increase the size of the program to an undesirable level. We would hardly want the benefits of deforestation to be cancelled out by worsened performance due to increased program size. Instead of inlining the entire body of each function that type inference determines is necessary to inline, we can use the approach described in the next section.

2.4 Worker/Wrapper Splitting

Instead of inlining the entire body of a function, we can split this function into a worker and a wrapper, where the wrapper is much smaller than the worker. Only the wrapper needs to be inlined, so the size of the program will not increase appreciably. This approach is described in [Chi00].
Consider the definition of `squares` from above. Just as before, type inference will determine that `mapInt` needs to be inlined. We construct a worker and wrapper for `mapInt` as follows. First, we do list abstraction on the body of `mapInt`, resulting in the following definition (with the bindings for the newly introduced type and term variables omitted):

```
mapInt :: (Int -> Int) -> [Int] -> t
mapInt f xs =
  case xs of
    [] -> n
    (y:ys) -> c (f y) (mapInt f ys)
```

In order to create the worker/wrapper split, the body of `mapInt` must be non-recursive. We transform it as follows:

```
mapInt :: (Int -> Int) -> [Int] -> t
mapInt f xs = go xs
  where go xs =
          case xs of
            [] -> n
            (y:ys) -> c (f y) (go ys)
```

Finally, we construct the worker, consisting of this function with λ-bindings for `t`, `c`, and `n`, and a wrapper which applies the worker to `[Int]`, `(:)`, and `[]`:

```
mapInt :: (Int -> Int) -> [Int] -> [Int]
mapInt f xs = mapIntW @[Int] ((:) Int) ([] Int) f xs
mapIntW :: forall t . (Int -> t -> t) -> t -> (Int -> Int) -> [Int] -> t
mapIntW @t c n f xs = go xs
   where go xs =
     case xs of
       [] -> n
       (y:ys) -> c (f y) (go ys)
```

When processing the body of `squares`, we only inline `mapInt`, not `mapIntW`:

```
squares (n::Int) = (mapIntW @[Int] ((:) Int) ([] Int)) square (enumFromTo 1 n)
```

It is easy to see the result of performing list abstraction on this definition.
The method for worker/wrapper splitting given above cannot construct worker/wrapper definitions that are polymorphically recursive in their constructor arguments – that is, where arguments other than the original `c` and `n` are passed to the recursive call of the worker. Some definitions have list-abstracted versions that have this property, although relatively

few of them do. Another method for worker/wrapper splitting which can handle this case is discussed later.

Note that worker/wrapper splitting makes it unnecessary to replace type variables in the type environment, as we discussed in the previous section. We can just inline all wrappers unconditionally, since they are small and unlikely to lead to code blowup. It is possible to combine the worker/wrapper splitting and type-based inlining approaches, and in fact our implementation does this. To simplify the presentation for now, we will assume that after worker/wrapper splitting, wrappers are inlined everywhere (even across module boundaries).

2.5 The Type Inference Algorithm

Here we discuss the type inference algorithm that is the core of type-based deforestation, as originally presented in [Chi99]. Our examples are presented in Haskell syntax, but keep in mind that the underlying language is actually the second-order *lambda*-calculus with explicit type annotations, extended with `let` (for introducing mutually recursive groups of function definitions) and `case` (for pattern-matching on algebraic datatypes). Type inference for the second-order λ-calculus is undecidable, but here we have explicit type annotations, which make it possible to perform type inference as needed for deforestation. This is reasonable, since in a compiler we would be performing the type-based deforestation algorithm on code that had been augmented with explicit type annotations in the process of desugaring from Haskell to a λ-calculus-based intermediate language. The type inference algorithm is based on Algorithm $\mathcal{M}$, for the Hindley-Milner type system [LY98], and in fact is so similar to Algorithm $\mathcal{M}$ that we will not repeat it here (see [Chi99] for details). The key step is that before type inference, list types and constructors will have been replaced with new type and term variables.

Given a type environment Γ, an expression e, and a type τ, $\mathcal{M}$ returns a principal typing for (Γ, e, τ) – that is, a type substitution mapping some of the type variables in e and τ to other types. Essentially, a typing σ for (Γ, e, τ) is a principal typing if all other typings for (Γ, e, τ) are substitution instances of σ. That is, σ is the "most general" possible typing, in that it avoids putting any unnecessary constraints on type variables occurring in e and τ. In an expression where list types and constructors have been replaced with new type variables, σ will map some of these variables to `[Int]`, signifying that they really must be list types (i.e., because the corresponding values are deconstructed by a `case` statement or passed to a function expecting a list). Some type variables may only be mapped to other type variables – these are the type variables that can be abstracted over. Furthermore, constructor variables whose types (after applying the substitution σ) are of the form `a -> t -> t` (for variables that replaced `(:)`) or `t` (for variables that replaced `[]`, where `t` is a variable introduced by list abstraction, are variables that can be abstracted over. If the result of applying σ to the type of the expression τ (in which list types have been replaced by type variables) is a type whose result is a type variable that was introduced by list abstraction, then list abstraction is successful and a worker/wrapper definition for e can be constructed.

2.6 Type-Based Deforestation

Now all the pieces to present the complete type-based deforestation algorithm (as presented in [Chi00]) are in place. The algorithm takes a collection of function definitions as input (for now, consider these to be Haskell definitions; we will be more concrete when we discuss the implementation), and returns a new collection of function definitions, with hope one where deforestation has occurred in some instances.

1. For each top-level definition `(v,ty,e)` where `ty` is either `[a]`, for some `a`, or a function type whose result is `[a]`, for some `a`:
   (a) Replace every list type with a fresh type variable, and every occurrence of a list constructor `(:)` or `[]` with a fresh term variable (this is detailed below). This gives a new expression `e'`.
   (b) Perform the type inference algorithm on (Γ,e',ty). (A global type environment Γ, containing types for all top-level definitions and any library functions used, is assumed to have been constructed at the beginning.) This gives a substitution σ.
   (c) Apply σ to `ty`. If the result type `ty'` contains type variables, we know that it's worthwhile to worker/wrapper split this definition.
   (d) Reinsert any type variables `t` such that σ(`t`) = `[a]` for any `a` with the original types for which they were replaced. Reinsert any constructor variables `c` or `n` whose types after applying σ result in a list type with the original list constructors for which they were replaced. This gives a new expression `e''` (This is described in more detail below.)
   (e) e'' may now contain different constructor variables whose types after applying σ are the same. Set a "canonical" constructor variable for each type and replace each constructor variable with the canonical constructor variable of its type (detailed below). This gives a new expression `e'''`
   (f) Does `ty'` contain type variables in its result? If so:
      i. Construct a new type `t'' = (forall t1 ... tn . ty')`, where `t1..tn` are all the free type variables in `ty'`.
      ii. Construct the worker definition by wrapping `e'''` with type and term variable bindings for all the free type variables in `ty'` and all the constructor variables remaining in `e'''`, giving a new expression `e''''`.
      iii. Generate a fresh name for the worker (here we call it vW). Return a pair of a wrapper and worker definition, where the wrapper definition is:
      `(v, ty, (vW @ t1 ... @ tk c1 n1 ... cn nn))`
      where `t1..tk` are the types for which all the new type variables were replaced and `c1 n1 ...cn nn` are the constructors for which all the new constructor variables were replaced. And the worker definition is:
      `(vW, t'', e'''')`.
      i. If not, return the original definition `(v,ty,e)`.
2. Now we have a collection of definitions that include workers and wrappers, as well as some of the original definitions. (Assume there is a way to tell whether a function name represents a worker, a wrapper, or neither.) Inline wrappers everywhere.

3. Perform β-reduction where necessary.

4. The previous two steps may be repeated some number of times, as inlining wrappers provides new opportunities for worker/wrapper splitting.

5. Find all expressions of the form `(foldr k z e)`. Perform list abstraction on `e`, just as presented above, except that we only try to abstract a single list, and no worker/wrapper splitting is done (list abstraction simply returns what would be the worker definition otherwise). If it succeeds, we have an expression of the form `( c n -> ...)`. Return the expression `( c n -> ...)  k z`. (We are eliding type arguments here.)

6. Perform β-reduction where necessary.

7. Repeat the previous steps some number of times, since applying the shortcut rule may expose other opportunities to apply the shortcut rule.

.

2.6.1 Abstracting Over Multiple Lists

Note that worker/wrapper splitting can produce a worker such that multiple lists are abstracted. For example, when worker/wrapper splitting the definition of `unzip`, which has the type `(forall a b .  [(a,b)] -> ([a],[b]))`, a worker can be derived whose type is `(forall a b t1 t2 .  (a -> t1 -> t1) -> t1 -> (b -> t2 -> t2) -> t2 -> [(a,b)] -> (t1,t2))`. We only attempt to abstract over multiple lists when doing worker/wrapper splitting. When we invoke the list abstraction algorithm during the deforestation stage, we only construct an abstraction when the result type has a single type variable as its result. So although our worker definition of `unzip` is such that either list can be removed, at any particular instance where `unzip` is called, only one list will be removed.

2.6.2 Polymorphically Recursive Worker Definitions

A slight modification of the worker/wrapper splitting algorithm given above is necessary to construct polymorphically recursive workers. In essence, we construct the most general possible worker type (the type obtained by replacing all list types in the type of the definition with type variables), and construct the wrapper body assuming the worker has this type. We replace any recursive calls to the function with this wrapper body and perform type inference, iterating to a fixed point of the result type. If the resulting type contains type variables, we construct the worker definition. (For more details, see [Chi00].) This is the method used in our implementation. It can occasionally achieve some staggering results, such as changing the quadratic time definition of `reverse` given above to a linear time definition. Chitil warns that since this can depend on the code being in a particular form, this method may be unpredictable by the programmer, since a small change to the code can suddenly turn a linear algorithm into a quadratic one if the code is changed to be in a form that our algorithm cannot deforest. Nevertheless, our implementation does this, since it cannot possibly make the performance of a particular function worse.

2.7 Related Work

The technique of Voigtlander [Voi02] seems quite amenable to being combined with type-based deforestation. Rather than attempting to deforest all list-producing functions, this approach focuses on three common ones, this list append function `(++)`, `reverse`, and `map`. For each of these functions, a so-called `vanish` combinator is defined, analogously to the `build` function – for example, `vanish(++)` for the append function. This combinator is applied to an expression which has had all occurrences of `(++)` abstracted out. Rules analogous to the shortcut rule are applied, and applications of these functions are fused together. Type inference could derive these special forms automatically, by treating these list functions as additional list constructors; it would be straightforward to extend the algorithm to do this. It is unclear whether this would be desirable. In the cases of the `(++)`, `reverse`, and `map` functions, type-based deforestation can derive worker/wrapper forms without any additional modifications. The only possible gain would be if `vanish` combinators were defined for functions which cannot be deforested by the shortcut rule, such as `zip`. Further study would be necessary to determine the feasibility of combining these two methods for deforestation.

Chapter 3

Implementing the Type Inference Algorithm

3.1 GHC and External Core

Our implementation of type-based deforestation depends on a feature of the Glasgow Haskell Compiler (GHC) which allows code in Core, the intermediate language used by GHC, to be written to files and read back in by GHC or other tools. Core, a typed intermediate language based on the second-order typed λ-calculus, is used in the middle end of GHC, which encapsulates most of the transformations and optimizations performed by GHC [PJS98]. External Core, the language on which our implementation operates, is a version of Core that has been modified to make it more suitable for exchange with external programs. The abstract syntax of External Core is given in Figure 3.1 as a Haskell datatype declaration, based on [Tol01].

A module consists of a name, a list of definitions for user-defined types (Haskell `data` and `newtype` declarations), and a list of binding groups, each of which consists of either a single non-recursive definition or a possibly mutually recursive group of definitions. We omit the syntax for `Tdef` as it is irrelevant.

The types `Var`, `Dcon`, `Tvar`, `Tcon`, and `Mname` are all synonyms for `String`. Uses of variables, data constructors, and type constructors (`Var`, `Dcon`, and `Tcon`) are always qualified (paired with a module name); λ-bound and *let*-bound variables have the module name `""`. The `Lit` type includes integer, rational, character, and string literals. Type application is distinguished from term application syntactically (which it must be, since a type is distinct from a term), but type abstractions and term abstractions are both represented by the `Lam` constructor. All variables bound by `Lam`, `Let`, and `Case` expressions are annotated with their types (or kinds, in the case of type variables; we omit the syntax for `Kind` here). `Let` binds a mutually recursive group of definitions. A `Case` statement binds the scrutinee expression to a new name which may be used in the alternatives; case alternatives can pattern-match on either data constructor applications, literals, or "anything" (`Adefault`). A `Note` form attaches some compiler-inserted annotation to an expression; an `External` form represents a foreign function call, whose type must be provided. A type is either a type variable, a type constructor, a type application, or a forall type.

Our implementation reads in External Core files, transforms them, and writes the results to new External Core files that will then be compiled to executable form by GHC. GHC

```
data Module = (Module Mname [Tdef] [Vdefg])

data Exp
   = Var (Qual Var)
   | Dcon (Qual Dcon)
   | Lit Lit
   | App Exp Exp
   | Appt Exp Ty
   | Lam Bind Exp
   | Let Vdefg Exp
   | Case Exp Vbind [Alt] -- the list must be non-empty
   | Coerce Ty Exp
   | Note String Exp
   | External String Ty

data Ty
  = Tvar Tvar
  | Tcon Tcon
  | Tapp Ty Ty
  | Tforall Tbind Ty

data Vdefg
  = Nonrec Vdef
  | Rec [Vdef]

type Qual a = (Mname, a)

newtype Vdef = Vdef (Qual Var,Ty,Exp)

data Bind
  = Vb Vbind
  | Tb Tbind

type Vbind = (Var,Ty)
type Tbind = (Tvar,Kind)

data Alt
  = Acon (Qual Dcon) [Tbind] [Vbind] Exp
  | Alit Lit Exp
  | Adefault Exp
```

Figure 3.1: The syntax of External Core

generates External Core after performing all Core simplifications and optimizations; we turn on most of GHC's optimizations, since this gives us code in a more useful form (for example, some inlining is already done and type classes are mostly specialized away). However, we are sure to turn off GHC's deforestation when compiling code to be read in by our implementation, since we would like to find out whether our implementation performs deforestation in all the cases that GHC does. We chose to take this approach rather than implementing type-based deforestation as a transformation pass in GHC in order to avoid making modifications to GHC. Sadly, in the end, we have probably made more modifications to GHC than we would have made otherwise, since we found and fixed many bugs related to the External Core facility.

3.2 Implementation Details

We list each major phase of the implementation here, and give more details on some of them in sections to follow.

1. Get the command-line arguments, consisting of a list of filenames containing External Core code.

2. Read in the files and parse their contents, resulting in a list of Core ASTs (which we will refer to from now on as "the program"). In addition, read in Core versions of certain library modules which contain code that might be amenable to deforestation (including the `List` and `Enum` modules, among others).

3. Perform dead code elimination, in order to remove any unused library code (this is only necessary to improve the performance of the implementation).

4. Read in the type environment for all the modules in GHC's `base` package, since the type inference phase will need to know the types of every variable mentioned in the program. The type environment is stored as an external file; alternatively, we could parse in all library files every time we run deforestation, but this would be very slow.

5. Build a call graph for the program, which will be used later by the heuristics that determine which functions to worker/wrapper split.

6. Specialize any polymorphically typed functions that produce lists. (see Section 3.2.1)

7. Perform list abstraction, which creates worker and wrapper definitions for as many functions as possible (this phase includes inlining any wrappers that are determined necessary to inline). (see Section 3.2.2)

8. Apply the shortcut rule wherever possible. (see Section 3.2.3)

9. Perform dead code elimination, to get rid of any definitions that might have been inlined everywhere they were used.

10. Specialize any worker applications to the types and terms to which they are applied, if possible. (see Section 3.2.4)

11. Perform the static argument transformation. (see Section 3.2.4)

12. At this point, we may have created worker/wrapper definitions for some library functions. To avoid duplicating code already provided by libraries that hasn't been changed by the transformation, we rename any functions present in the library modules that were not originally defined in these modules, and write these modules to new modules that have different names.

13. Write out each transformed module to a file.

3.2.1 Specializing Polymorphic Functions

For reasons discussed in Section 3.2.2, having to do with the combination of worker/wrapper splitting and type-based inlining that we use, it is desirable to make specialized copies of all type applications of polymorphic functions that produce lists. For example, if the program contains the following expression:

```
map @ Int @ Int square [1,2,3]
[...]
map :: forall a b . (a -> b) -> [a] -> [b]
map a b (f::(a -> b)) (xs::[a]) = \ @ a @ b f xs -> ...
```

we will create a copy of `map` that is specialized to the type arguments `Int`, `Int` and replace this application of `map` with a call to the specialized copy:

```
map_spec1 square [1,2,3]
[...]
map_spec1 :: (Int -> Int) -> [Int] -> [Int]
map_spec1 (f::(Int -> Int)) (xs::[Int]) = \ f xs -> ...
```

In addition, if a function is applied more than once to the same combination of type arguments, we will call the same specialized copy in each case, in order to avoid creating multiple identical specialized definitions.

3.2.2 List Abstraction

List abstraction consists of the following steps:

1. Perform worker/wrapper splitting on the program.

2. Some definitions may require certain other functions to have worker/wrapper definitions in order to be worker/wrapper split. If the previous pass of worker/wrapper splitting, discovered any of these, we inline the wrappers of those that have already been worker/wrapper split.

3. Perform β-reduction where necessary.

We repeat these steps some fixed number of times, since worker/wrapper splitting some definitions may enable more definitions to be worker/wrapper split.

Worker/Wrapper Splitting Combined With Type-Based Inlining

In [Chi00], the deforestation algorithm presented only performs worker/wrapper splitting during the list abstraction stage, and in the following stage inlines all wrappers everywhere. However, Chitil does suggest that this might be combined with the type-based inlining originally described in [Chi99], so that wrappers of functions are only inlined within definitions where the type of the definition depends on the type of the function. We implemented this combined approach, because we found that when wrappers are inlined everywhere, the performance of the resulting program can actually become worse (incongruously enough, since the worker/wrapper scheme was originally devised in order to avoid the inefficiency introduced by aggressive inlining). This can happen when the result of a function that has a worker/wrapper pair is not actually consumed by `foldr`. The resulting program will contain expressions like `(mapW @ Int @ Int @ [Int] @ [Int] ((:) @ Int) ([] @ Int) f xs)`, in which `mapW` has additional (useless) arguments that the original definition of `map` does not have. In practice, this can make the performance of the resulting program quite poor, possibly overshadowing any gains from deforestation. In Section 3.2.4, we discuss the ways in which we originally attempted to address the problem of inefficient workers, but selectively inlining wrappers supersedes these approaches. (However, they remain present in the implementation, for redundancy's sake.)

Now we can explain why the specialization pass mentioned in Section 3.2.1 is necessary. A subtle detail of the list replacement phase of worker/wrapper splitting is that we cannot replace a list type `[a]`, where `a` is a free type variable, with a new type variable. To explain why, we give an example:

```
foo =
  let (l1::[Int]) = map square [1,2]
      (l2::[Char]) = map toUpper ['a','b']
   in ....

map :: forall a . b . (a -> b) -> [a] -> [b]
```

When we do list replacement on the type of `map` in the type environment, we assign the type `forall a b. (a -> b) -> t1 -> t2` to `map`, where `t1` and `t2` are free type variables. We also replace list types in `foo`, giving us:

```
foo =
  let (l1::t3) = map square [1,2]
      (l2::t4) = map toUpper ['a','b']
   in ....

map :: forall a . b . (a -> b) -> t1 -> t2
```

In effect, `map` is being used polymorphically, but replacing `[a]` with a single type variable has made `map` no longer polymorphic – `map` now returns something of type `t2`, a type which has no relationship to the type of the function argument, `(a -> b)`. Type inference will unify `t3` and `t4` with `t2`, implying that `l1` and `l2` have the same type, and if list abstraction succeeds, that `l1` and `l2` can be built using the same constructor variables. This is clearly unsound. Thus, list replacement will not replace list types containing free type variables, which means

that worker/wrapper splitting will never succeed on functions that build their results by calling polymorphic list-producing functions. We solve this problem by specializing these polymorphic functions, as described in Section 3.2.1.
To summarize the worker/wrapper splitting algorithm we use:

1. For each definition in the program, try to perform worker/wrapper splitting as described in Section 2.6. In addition, during the list replacement phase, we determine the set of free names used in the definition body e, and for each name n in this set, we look up its type t in the global environment; if t exists, we replace all list types in it with type variables, and extend the global environment with (n,t).

2. If abstraction fails, we return the original definition d. If it succeeds, we must determine whether or not the resulting, abstracted type t depends on type variables that are free in some type in the global environment. We look up the type of each free variable v in e and check whether it contains any free variables that were unified with variables contained in t. If so, we add v to a global environment[1] mapping d onto a list of wrappers that should be inlined in d. If the resulting list of wrappers is non-empty, we return the original definition; otherwise, we return the worker/wrapper pair.

3. For each definition d in the program, look up the list of wrappers l that should be inlined in d. Of course, not all the variable names in this list may actually have wrapper definitions, so we check whether all the names are actually wrapper names (this can be done using a global data structure). If so, we inline them. If not, we return d unchanged. The reason why we only inline the wrappers for d if *all* of them are wrappers is that it may not be *possible* to construct worker/wrapper pairs for all the definitions named in l; if it's not, we run the risk of inefficiency resulting from workers not involved in deforestation, as explained above.

4. Perform β-reduction where necessary.

Consumer Worker/Wrapper Splitting

At this point, the astute reader might be wondering how the shortcut rule can ever be performed on an expression like `(sum (someListProducer))`. Even if `(someListProducer)` is worker/wrapper split and the wrapper is inlined, the expression takes the form `(sum (someListProducerW @ [a] ((:) a) ([] a)))`. `sum` is defined in terms of `foldr`, but that's not apparent here – the shortcut rule only applies in cases where `foldr` is literally applied to an expression that be list-abstracted. We could address this issue by inlining the definitions of all functions that contain `foldr`, and our implementation originally did this, but as one might expect, this sometimes made the performance of the resulting program worse, since not every instance of `foldr` consumes an expression that can be list-abstracted. Instead, we adopt a variation on the worker/wrapper scheme for list consumers, which was outlined in [Chi00]. Worker/wrapper splitting for list consumers is simpler than the analogous method for list producers, as it requires no type-based analysis. We illustrate it with an example:

[1] that is, "global" as in "contained in the state of the monad in which the entire deforestation algorithm runs

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

   ==>

sum :: [Int] -> Int
sum (xs::[Int]) =
   sumW @ [Int]
      (\ @ t (v_cons::(Int -> t -> t)) (v_nil::t) ->
          foldr @ Int @ t v_cons v_nil xs)
sumW :: (forall t . (Int -> t -> t) -> t -> t) -> Int
sumW (xs::(forall t . (Int -> t -> t) -> t -> t)) = xs @ Int (+) 0
```

For any definition that applies `(foldr @ a @ t k z)`, for some `k` and `z`, to one of its list arguments `(l::[a])`, where `l` is not used in any other way, we replace the `foldr` application with `(l @ t k z)`, and change the type of `l` in its λ-binding to `forall t . (a -> t -> t) -> t -> t`, in order to make the worker body. Now, instead of taking a list argument, the worker takes a generalized list producer. Note that this is an example of higher-order polymorphism, which is not supported in the Haskell standard, but which is allowed in GHC Core. The wrapper body applies the worker to `[a]` and passes in a generalized list producer which simply applies `(foldr @ a @ t c n)` to the wrapper's list argument, where `t`, `c`, and `n` are the arguments passed to the generalized list producer. Effectively, the `foldr` is brought outside the function body, where it can be inlined as part of the wrapper. Applying the shortcut rule can then proceed as follows:

```
(sum someListProducer)
  ==> (after worker/wrapper splitting of {\tt sum} and inlining)
(sumW @ [Int] (\ @  t (v_cons::(Int -> t -> t)) (v_nil::t) ->
                  foldr @ Int @ t v_cons v_nil someListProducer))
  ==> (after worker/wrapper splitting of {\tt someListProducer} and inlining)
(sumW @ [Int] (\ @  t (v_cons::(Int -> t -> t)) (v_nil::t) ->
                  foldr @ Int @ t v_cons v_nil (someListProducerW @ [Int] ((:)
  [Int]) ([] Int)))))
  ==> (after applying the shortcut rule)
(sumW @ [Int] (\ @ t (v_cons::(Int -> t -> t)) (v_nil::t) ->
                  (someListProducerW @ Int t  v_cons v_nil)))
```

`sumW` just applies its argument to `(+)` and `0`, so in this modified expression, `someListProducer` produces its sum directly rather than constructing an intermediate list and summing it up. For functions that both consume and produce a list, such as `map`, a worker/wrapper pair can be constructed that combines consumer and producer worker/wrapper splitting: just perform consumer worker/wrapper splitting first, and then perform producer worker/wrapper splitting on the resulting worker.

3.2.3 Applying the Shortcut

After all worker/wrapper splitting has concluded, we perform shortcut deforestation as described in Section 2.6. The only other detail is that in some cases where list abstraction

fails on an expression, it may designate a list of wrappers to be inlined in that expression, so when abstraction fails, we inline those wrappers as in Section 3.2.2 and return the resulting expression, in the hopes that some future iteration of deforestation may be able to perform the shortcut on it.

3.2.4 Eliminating Inefficient Workers

A major problem with worker/wrapper splitting is that workers whose results are not consumed by `foldr` at some particular instance where wrapper inlining was performed are less efficient than their original definitions, as they take extra arguments (and must pass these arguments to any recursive calls they make), and they call polymorphic functions to construct their results rather than data constructors whose types are statically apparent. This is less of an issue in our current implementation, since wrappers are inlined only on demand. It is still somewhat of an issue, since every wrapper is defined in terms of its worker, so even when wrappers are not inlined, calling the wrapper means applying the worker to some constructor arguments. It is possible to inline each worker in the body of its wrapper, but a more general solution is to use specialization.

Specialization

After deforestation, we search for applications of workers to "constant" type and constructor arguments – that is, any arguments that don't contain free type or term variables. We replace these applications with calls to specialized copies of the workers, such as in the following example:

```
mapW @ Int @ Int @ [Int] ((:) Int) ([] Int) square [1,2]

mapW :: forall a b t . (b -> t -> t) -> t -> (a -> b) -> [a] -> t
mapW @ a @ b @ t (c::(b -> t -> t)) (n::t) (f::a -> b) (xs::[a]) =
  case xs of
    [] -> n
    (y:ys) -> c (f y) (mapW @ a @ b @ t c n ys)

   ===>
mapW_spec1 square [1,2]
mapW_spec1 :: (Int -> Int) -> [Int] -> Int
mapW_spec1 (f::Int -> Int) (xs::[Int]) =
  case xs of
    [] -> []
    (y:ys) -> (f y):(mapW_spec1 f xs)
```

`mapW_spec1` looks a lot like the original definition of `map`, except that it has been specialized to type `Int` for its first two type arguments.
Specialization can potentially lead to code blowup if workers are applied in many different ways (we attempt to use the same specialized copies to replace applications of the same worker to the same arguments, but this sometimes fails, for example when arguments are only equal modulo α-renaming), but in practice this does not seem to be a problem.

The Static Argument Transformation

Most worker definitions derived by our algorithm only call themselves recursively on the same constructor arguments are passed (if at all) – for example, this is true of the worker definition of `map` given above. It is unnecessary to pass these extra arguments through each recursive call. Instead, as [Chi00] suggests, we can perform the static argument transformation:

```
fW @ t c n xs =
  ... (fW @ t c n xs) ...

  ==>

fW @ t c n xs =
   let fW' xs =
      .... (fW' xs) ....
   in fW' xs
```

As formulated in [Chi99], the worker/wrapper splitting does this already, but the polymorphically recursive worker/wrapper splitting algorithm detailed in [Chi00], which we implement, does not, so we must perform the static argument transformation on all worker definitions as a postprocessing step.

Chapter 4

Experimental Results

4.1 The Nofib Suite

We tested our implementation on programs from the `nofib` suite, a benchmark suite for Haskell [Par93]. The suite is divided into three categories: "imaginary", containing toy programs such as programs to solve the n-queens problem and compute Fibonacci numbers (contrary to the name of the suite); "spectral", containing medium-sized programs such as a constraint solver and various scientific simulations; and "real", consisting of real applications contributed by users, such as a strictness analyzer and an implementation of the "grep" search-and-replace program. We included nine "imaginary" programs, 55 "spectral" programs, and 25 "real" programs in our tests. The programs range from 12 to 10,000 lines in size, and each consists of anywhere from one to 32 modules.

4.2 Experimental Methods

For each benchmark, we produced three executable programs, compiled in each of the following ways:

1. No deforestation (compiled with `ghc -frules-off` to disable GHC's deforestation)
2. GHC's deforestation turned on (compiled with normal GHC)
3. Compile the program to External Core, run type-based deforestation on it, compile the reuslt with `ghc -frules-off`

We compiled all the programs with GHC's profiling turned on, so that we could measure the amount of heap memory allocated by each program. When we speak of programs getting "better" or "worse" in the coming sections, we are always referring to allocation, rather than time. We have not yet collected data on the effect type-based deforestation has on running time, as we found that GHC's time profiling was not as reliable as its allocation profiling.
We always ran GHC with the `-O1` option to enable most optimizations, since when optimization is on, GHC does many transformations such as resolving type class overloading that put the program into a more convenient form for deforestation.
We made one change to GHC, having to do with list comprehensions. List comprehensions are a feature of Haskell that can either be desugared using `foldr` and `build` [GLP93], or

in a more efficient manner. Normally, GHC only uses the `foldr/build` desugaring when deforestation is enabled. We changed GHC to use this desugaring unconditionally, since it is important that we run our deforestation on non-deforested code, in order to make sure we catch all the opportunities for deforestation that GHC exploits.
In addition, we changed a few library functions, such as `map` and `filter`, to be defined in terms of `foldr`, since in theory, if deforestation is effective, then these functions should be written this way.

4.3 Results

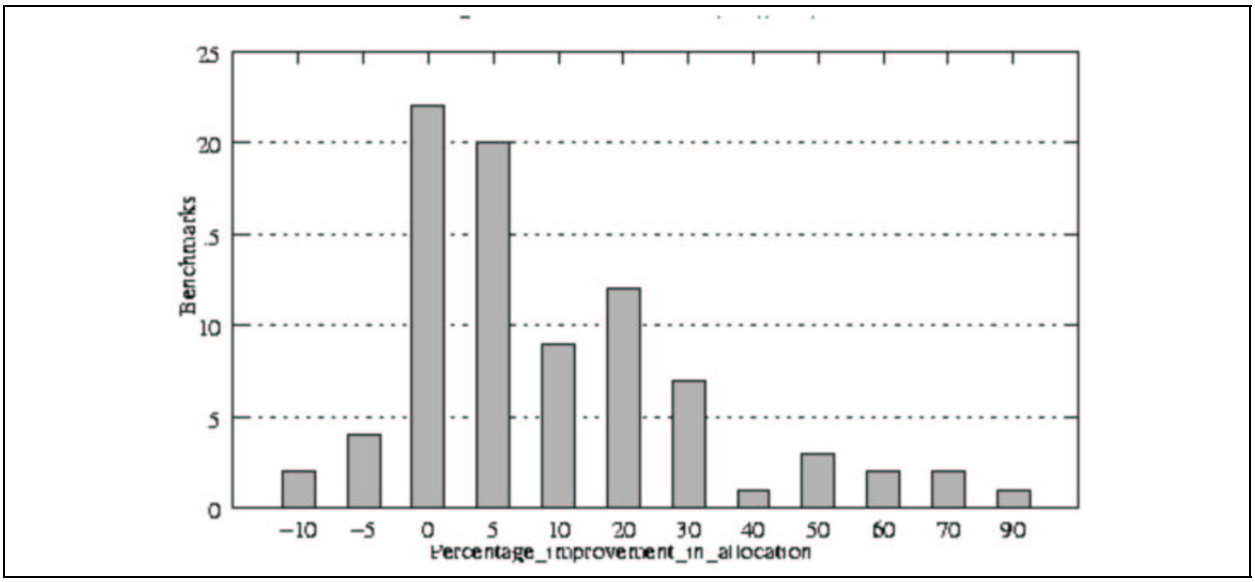

Figure 4.1: Type-based deforestation compared to no deforestation

Figure 4.1 shows the results of compiling the `nofib` suite using type-based deforestation, compared to no deforestation. For each n, bar n shows the number of benchmarks where the amount of memory allocated decreased by anywhere between n and n - 5 percent.
As we would expect, the mode of the improvements is zero. As Peyton Jones, Tolmach and Hoare commented in [PJTH01], "Compiler optimisations are like therapeutic drugs. Some, like antibiotics, are effective on many programs... Others are targeted at specific 'diseases', on which they tend to be devastatingly effective, but have no effect at all on most other programs." However, there is a significant class of programs on which deforestation is devastatingly effective, as we see from the graph.
The benchmark program which improves the most is `parstof`, in the `spectral` category of the `nofib` suite. Deforestation decreases the amount of memory allocated by this program, a parser for a simple language, by 93%. This result was also achieved by GHC's built-in deforestation, as documented in [PJTH01] (though the current version of GHC doesn't achieve this result) – the outer loop of the program parses a sample input 40 times, and deforestation transforms it so that the text is only parsed once. A real program would not be written this way, so this result is somewhat of an artifact of the style in which some benchmarks are written. The next-best improvers are `integrate` (which, as one might

guess, performs numerical integration), in the `imaginary` category, and `cryptarithm1`, in the `spectral` category, a program that solves a particular instance of the "crypto-arithmetic" puzzle – in this case, finding a solution to the equation
`thirty + 5 * twelve = ninety`
where each letter represents a digit between 0 and 9. Both of these programs perform "bulk" operations on lists, in the words of [PJTH01]; for example, `cryptarithm1`'s main body consists of applying `filter` to a list of permutations of the digits from 0 to 9. The body of the function that computes permutations is defined in terms of several list comprehensions, which are desugared using `foldr/build`. Deforestation is well-suited for programs such as this one, that consist a simple "pipeline" where a list is passed to and returned by each "pipeline stage" (in this case, the stages are `enumFromTo`, which constructs the list `[0..9]`; `permutations`, a user-defined function which computes the permutations of this list; and `filter`, which selects those permutations which are solutions of the equation).
The benchmark program whose allocation is worsened the most by deforestation is `integer`, in the `spectral` category, whose allocation is increased by 15%. The program is meant to test the performance of Haskell's built-in integer operation, and its main function looks like this:

```
intbench op astart astep alim bstart bstep blim =
  seqlist ([ a `op` b
   | a <- [ fromInteger astart,
                    fromInteger astart + fromInteger astep ..
                    fromInteger alim ]
   , b <- [ fromInteger bstart,
                    fromInteger astart + fromInteger bstep ..
                    fromInteger blim ]])
```

The `seqlist` function takes a list of elements and forces each one to be evaluated, using Haskell's `seq` primitive, so the `intbench` function constructs a list of integer expressions and forces each one to be evaluated. Thus, there is no real opportunity for deforestation here, since the entire list will always be demanded. However, since the list comprehension that is the argument to `seqlist` is desugared into an expression involving `foldr`, shortcut deforestation will be performed, but the net effect is only to create inefficient workers. The second-worse program, whose allocation is worsened by 7% (compared to no deforestation – it improves by 12% compared to GHC's deforestation, curiously), is `sphere`, a raytracing program, also in the `spectral` category. This program has some auxiliary list functions that are defined in terms of list comprehensions that are not responsible for a significant amount of allocation, and converting these to worker/wrapper form results in inefficient workers that increase the overall amount of allocation, as in `integer`.

4.4 Comparison with GHC

Figure 4.2 is analogous to Figure 4.1, except the baseline is now compiled with GHC's deforestation turned on.
It is clear that many of the positive results obtained above are solely due to type-based deforestation reproducing the same results that GHC achieves; however, there is still a

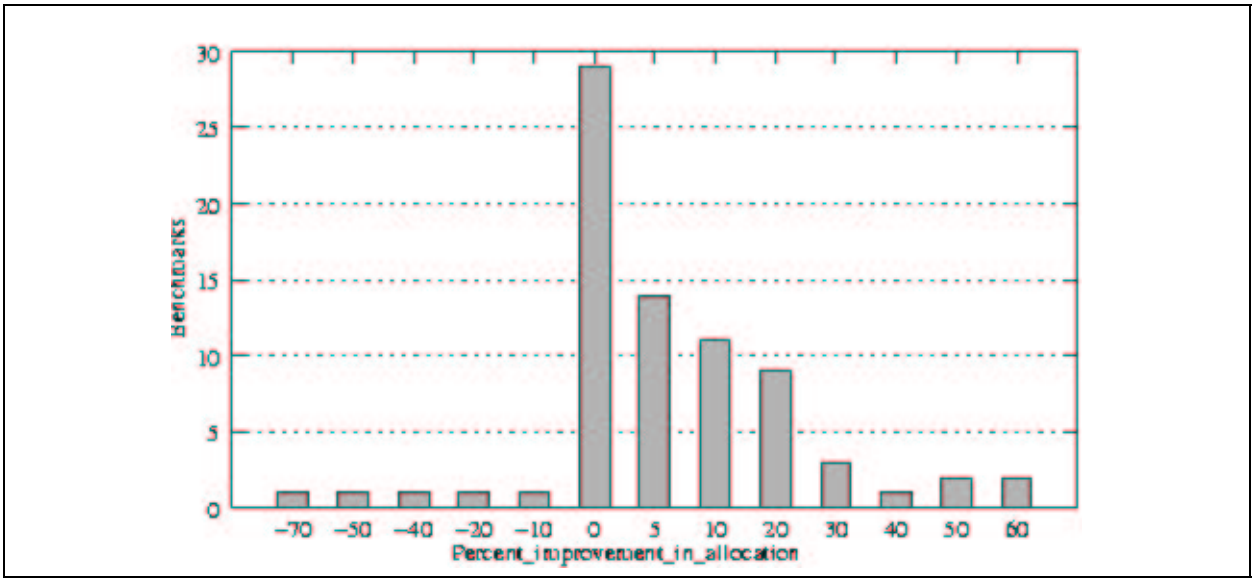

Figure 4.2: Type-based deforestation compared to GHC's shortcut deforestation

significant class of programs that type-based deforestation improves appreciably. Note that several programs now perform worse with type-based deforestation than without – this is not due to type-based deforestation changing the program in a way that makes it perform worse, but because type-based deforestation fails to reproduce the improvements that GHC makes in these cases.

The best improvers compared with GHC are the same as the best improvers listed in Section 4.3. The program whose allocation is worsened the most is `integer`, as in Section 4.3 The second-worst is `infer`, an implementation of Hindley-Milner type inference, in the `real` category, whose allocation is worsened by -12% (ironically enough). This result is due to a few frequently called functions being consumer worker/wrapper split (see Section 3.2.2); the resulting code contains extra, unnecessary calls to `foldr` which are not eliminated by specialization.

4.5 Deforestation Can Make Performance Worse

In the process of testing and refining our implementation, we encountered many programs that were apparently made worse by type-based deforestation. In most of these cases, we were able to adapt the implementation so that these programs were no longer made worse, and whatever benefits it achieved on other programs were preserved. However, we would expect to encounter the phenomena enumerated below if we tested the implementation on a wider range of programs.

Inefficient Workers

A worker/wrapper definition of a given function is inherently less efficient than the original function, since the new function takes extra arguments, constructs its results using unknown functions rather than literal list constructors (possibly defeating other optimizations), and must pass the constructor arguments to any recursive calls it makes. If the static argument

transformation is performed, as it is in our implementation, the last problem is averted, except when we create polymorphically recursive worker definitions. But in general, whenever we create a worker/wrapper pair and inline the wrapper, it is possible that the result of the call to the worker will not be consumed by a `foldr`, and so we have effectively de-optimized the function under consideration, leading to an increase in allocation that may cancel out any decrease achieved by deforestation at other call sites. In an early implementation of our system that inlined wrappers everywhere rather than doing so in a type-directed way, this was a major problem. It is much less of a problem now that we inlined wrappers only where necessary, but it is still somewhat of an issue.

Work Duplication

Consider the following expression:

```
let (x::[Int]) = map someExpensiveFunction [1..n] in
   ... x ... x ...
```

List abstraction creates a worker/wrapper definition for x:

```
let (x::[Int]) = xW @ [Int] ((:) Int) ([] Int)
    (xW :: forall t . (Int -> t -> t) -> t -> t) =
       \ c n -> mapW @ t c n someExpensiveFunction [1..n] in
    ... (xW @ [Int] ((:) Int) ([] Int)) ... (xW @ [Int] ((:) Int) ([] Int)) ...
```

In the original program, `x` is only computed once, due to the call-by-need semantics of Haskell: `x` is computed the first time it is used, and then the closure that represents it is overwritten with its value, which is used the second time. But in the transformed program, `x` has become a function rather than a constant value, so `x` is computed more than once, calling `someExpensiveFunction` twice as often as before.

We encountered cases like this in the `nofib` suite, and attempted to address the problem in two ways. Initially, we avoided worker/wrapper splitting definitions of type `[a]` (rather than a function type) if the definition's right-hand side was determined to do "interesting" work (our heuristic for "interestingness" was that it called functions defined outside the enclosing module). Though this prevented many of the programs that previously were made worse by deforestation from getting worse, it also prevented some of the programs that previously improved from improving. We found that a better solution was to compute an approximation the number of times that a `let`-bound definition of a constant list, like `x` above, was used in its body, and avoid worker-wrapper splitting it if it was used more than once. This method is not foolproof, because in general computing the number of times `x` will really be used is undecidable – however, taking a first-order approximation (counting the number of times `x` literally appears in the body) seems to be adequate. Usually, though, it seems to prevent programs from getting much worse while not negating the benefits achieved by deforesting other programs. It is an inherent limitation of shortcut deforestation that only lists which are used once can be eliminated, so it is natural to avoid trying to deforest lists that are used multiple times.

Mutually Recursive Functions

Consider the following code (simplified from actual code generated by GHC for the `mandel2` benchmark in the `nofib` suite):

```
a = f b
b = g a
```

`a` and `b` compute an infinite list – each one calls the other to generate some portion of the list, and then applies some function to it. We can see that even if `f` is defined in terms of `foldr`, we cannot apply the shortcut rule in the body of `a`. We cannot eliminate an infinite list. However, worker/wrapper splitting may derive worker definitions for these functions:

```
aW = \ @ t c n -> f (bW @ t c n)
bW = \ @ t c n -> g (aW @ t c n)
```

This makes performance much worse, since two extra term arguments are now passed to each recursive call of `a` or `b`. In addition, these worker instances cannot be specialized, since attempting to create specialized versions of `bW` or `aW` would lead to an infinite regress. Our implementation constructs a call graph for the program and avoids worker/wrapper splitting any functions that are involved in a cycle of length 2. Obviously, this does not handle the case where more than 2 functions construct a list in a mutually recursive way, but this avoids worsening performance in the common case.

Interaction With Other Optimizations

When working with compiler optimizations, it is a given that one optimization may prevent another from being applied, resulting in worse performance than if only one optimization had been applied rather than several. It is possible that type-based deforestation may prevent other optimizations that GHC would have applied, leading to overall worse performance, or that optimizations GHC performed before generating External Core might transform a program into a form unsuitable for deforestation. It would be desirable to carefully study the effect the interaction between deforestation and other optimizations, but we leave this as future work. In the meantime, we note that deforestation might result in better performance if it were integrated as a transformation pass in GHC rather than running as an external pass: in our system, GHC applies its transformations a number of times, then we perform deforestation, then GHC does its transformations again; if deforestation were interleaved with other transformations, better results might be achieved.

4.6 Measuring Lists Allocated

Besides measuring the total amount of memory a program allocated with and without allocation, it is also desirable to measure the amount of memory it allocates that is part of a list data structure. If deforestation achieves a 0% improvement on a given program P, we would like to know whether that is because deforestation is missing some opportunities present in P, or because P just doesn't allocate many lists.

We modified GHC's profiling system to output the total amount of memory allocated in lists as well as the total allocation. Figure 4.3 plots, for each of the `nofib` programs, the

percent improvement achieved by type-based deforestation versus the percentage of its total allocation that is part of some list data structure versus (compared to no deforestation).

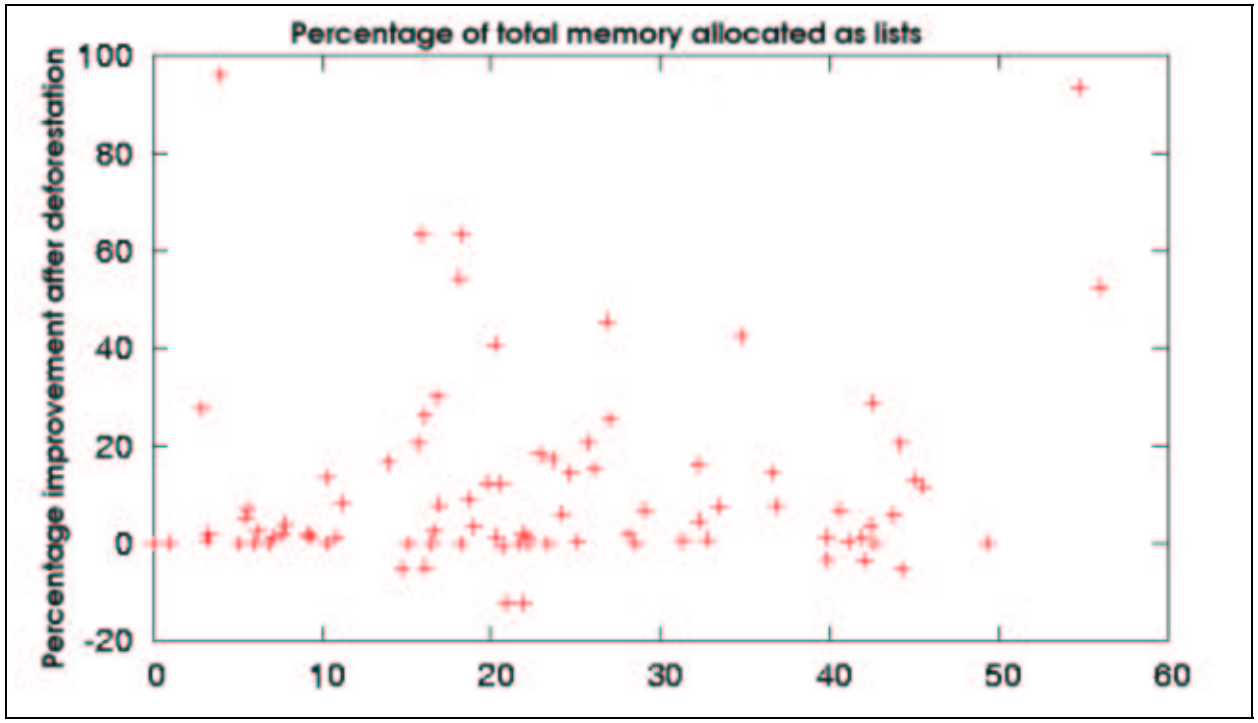

Figure 4.3: Improvement Achieved by Deforestation Versus Percentage of Memory Allocated For Lists

Clearly, there is a large group of programs where 10% of total allocation or less consists of lists, and deforestation achieves no effect. We cannot hope to achieve better results on these programs. However, there are also many programs where more than 40% of allocation consists of lists, but deforestation achieves no effect. This does not necessarily imply that our implementation misses opportunities for deforestation in these cases. Some lists, after all, may be inherently necessary for the program's computation. Explaining the reasons for these results would require either closer examination of the programs for which deforestation's improvement is not proportional to the amount of lists allocated, a static or dynamic analysis to determine which lists are "truly intermediate", or both.

Chapter 5

Conclusions and Future Work

5.1 Type-Based Deforestation Is Effective

We have shown that type-based deforestation achieves significant improvements in the amount of memory allocated by real programs; moreover, that it generally achieves equal or greater improvements than those achieved by GHC's built-in deforestation. There is further work to be done in examining the programs that GHC's deforestation improves more than type-based deforestation does, and in studying the relationship between total amount of lists allocated and the potential improvement achieved by deforestation, but our work has made it clear that the type-inference-based algorithm for deforestation is useful in practice.

5.2 Future Work

5.2.1 Integration with GHC

It would be desirable to incorporate our implementation as a transformation pass in GHC. This would simplify the implementation greatly, reducing the amount of time required to run experiments (the implementation repeats work such as parsing and building a type environment that would have already been done if it were a pass in GHC). Also, it might increase the potential improvements achieved by deforestation, because deforestation would be performed multiple times in concert with other optimizations that might create more deforestation opportunities. The only way to find out whether this is the case would be to integrate type-based deforestation with GHC.

5.2.2 Arbitrary Data Structures

It is not difficult to imagine how the type inference algorithm might be extended to remove arbitrary data structures, and not just lists. Shortcut deforestation can potentially eliminate values of any datatype for which a `fold` operation can be defined (it is trivial to generalize the `build` operation to other types): for example, warm fusion is defined in a way that makes it applicable to arbitrary types [LS95]. It would not be hard to extend our implementation of type-based deforestation to remove arbitrary data structures, but problems might occur in practice, since programmers may be less likely to define functions that produce types other than lists in terms of a `fold` operation. A potential application for this generalized

form of deforestation is the style of generic programming in Haskell described in [LPJ03]. This paper describes a library that provides generic traversal functions for arbitrary user-defined datatypes, for example, combinators to allow a function defined on a certain algebraic datatype that does something interesting only to values with one particular constructor to be applied without writing out the cases for each constructor (what is known as the "visitor pattern" in object-oriented programming). These traversals depend crucially on a "generic fold" operator, so it is possible that there might be many opportunities for deforestation in these generic programs, which consume data structures using functions that are ultimately defined in terms of `fold`.

5.2.3 Programmer Annotations

The basic concern of type-based deforestation is to infer which functions construct lists that are "virtual" – that only exist because of the particular way in which the programmer chose to express the algorithm, and are not essential to the computation being performed. But programmers presumably have some knowledge about which if their functions have this property. Why not allow them to express this knowledge by means of "virtuality" annotations on functions? These annotations could be used in the compiler to suggest places to try to apply deforestation, avoiding the work of trying to derive worker/wrapper forms for functions that produce lists that are not, in fact, virtual. Also, they could be used to provide useful feedback to the programmer. As the situation stands, a small change in the program can make a big difference in its performance – a change that is trivial from the programmer's point of view may prevent deforestation from occurring. The programmer cannot predict when this will happen, without detailed knowledge of the compiler. With virtuality annotations, however, the compiler could report back to the user when uses of "virtual" functions cannot be deforested. This would result in a more efficient optimization process and better predictability for the programmer.

Bibliography

[BT84] Timothy Budd and Joseph M. Treat, "Extensions to Grid Selector Composition and Compilation in APL", Information Processing Letters, Vol 19(3): 117-123, (Oct 1984).

[BD77] Rod M. Burstall and John Darlington, "A transformation system for developing recursive programs", Journal of the ACM, 24(1):44–67, January 1977.

[Chi99] Olaf Chitil, "Type Inference Builds a Short Cut to Deforestation", Proceedings of the International Conference on Functional Programming, 1999.

[Chi00] Olaf Chitil, "Type-Inference Based Deforestation of Functional Programs", Ph.D thesis, Aachen University of Technology, 2000.

[GLP93] Andrew Gill, John Launchbury, and Simon Peyton Jones, "A Short Cut to Deforestation", Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93), pp. 223-232, 1993.

[GW78] Leo J. Guibas and Douglas K. Wyatt, "Compilation and Delayed Evaluation in APL", Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '78), pp. 1-8, 1978.

[Hug89] John Hughes, "Why Functional Programming Matters", Computer Journal, pp. 98-107, 1989.

[LPJ03] Ralf Lämmel and Simon Peyton Jones, "Scrap your boilerplate: a practical design pattern for generic programming", Proceedings of the ACM Workshop on Types in Language Design and Implementation (TLDI '03), pp. 26-37, 2003.

[LS95] John Launchbury and Tim Sheard, "Warm Fusion: Deriving Build-Catas from Recursive Definitions", Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '95), pp. 314-322, 1995.

[LY98] Oukseh Lee and Kangkeun Yi, "Proofs about a folklore let-polymorphic type inference algorithm", ACM Transactions on Programming Languages and Systems, 20(4):707-723, July 1998.

[Par93] Will Partain, "The nofib Benchmark Suite of Haskell Programs", Functional Programming, Glasgow 1992, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 1993.

[PJM99] Simon Peyton Jones and Simon Marlow, "Secrets of the Glasgow Haskell Compiler inliner", Proceedings of the Workshop on Implementing Declarative Languages, 1999.

[PJS98] Simon Peyton Jones and André Santos, "A transformation-based optimiser for Haskell", Science of Computer Programming 32(1-3), pp3-47, September 1998.

[PJTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare, "Playing by the rules: rewriting as a practical optimisation technique in GHC", Proceedings of the ACM/SIGPLAN Haskell Workshop, 2001.

[Ste90] Guy Steele, *Common Lisp: The Language*, 2nd edition, Digital Press 1990.

[Sve02] Josef Svenningsson, "Shortcut Fusion for Accumulating Parameters & Zip-like Functions", Proceedings of the International Conference on Functional Programming, 2002.

[Tol01] Andrew Tolmach and the GHC Team, "An External Representation for the GHC Core Language" (draft), available at `http://www.haskell.org/ghc/docs/papers/core.ps.gz`

[Voi02] Janis Voigtlander, "Concatenate, Reverse and Map Vanish For Free", Proceedings of the International Conference on Functional Programming, 2002.

[Wad85] Philip Wadler, "Listlessness is better than laziness II: Composing listless functions", Proceedings of the Workshop on Programs as Data Objects, Copenhagen, October 1985. LNCS 217, Springer-Verlag, 1985.

[Wad90] Philip Wadler, "Deforestation: transforming programs to eliminate trees", *Theoretical Computer Science*, v. 73, pp. 231-248, 1990.

[Wat89] Richard Waters, "Optimization of Series Expressions: Part II: Overview of the Theory and Implementation", MIT AI Lab Memo No. 1083, December 1989