UNIVERSITY
of
GLASGOW

Department of
Computing Science

Deforestation
for
Higher-Order Functional Programs

Simon David Marlow

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

September 1995

Abstract

Functional programming languages are an ideal medium for program optimisations based on source-to-source transformation techniques. Referential transparency affords opportunities for a wide range of correctness-preserving transformations leading to potent optimisation strategies.

This thesis builds on deforestation, a program transformation technique due to Wadler that removes intermediate data structures from first-order functional programs.

Our contribution is to reformulate deforestation for higher-order functional programming languages, and to show that the resulting algorithm terminates given certain syntactic and typing constraints on the input. These constraints are entirely reasonable, indeed it is possible to translate any typed program into the required syntactic form. We show how this translation can be performed automatically and optimally.

The higher-order deforestation algorithm is *transparent.* That is, it is possible to determine by examination of the source program where the optimisation will be applicable.

We also investigate the relationship of deforestation to cut-elimination, the normalisation property for the logic of sequent calculus. By combining a cut-elimination algorithm and first-order deforestation, we derive an improved higher-order deforestation algorithm.

The higher-order deforestation algorithm has been implemented in the Glasgow Haskell Compiler. We describe how deforestation fits into the framework of Haskell, and design a model for the implementation that allows automatic list removal, with additional deforestation being performed on the basis of programmer supplied annotations. Results from applying the deforestation implementation to several example Haskell programs are given.

Acknowledgements

First of all, I'd like to thank my supervisor Phil Wadler, whose constant supply of ideas and encouragement made this possible. I'd also like to thank my Viva Committee: Dave Sands, Simon Peyton Jones and David Watt who provided many useful comments and suggestions during my Viva.

Thank you to everyone in the Functional Programming Group at Glasgow University, and especially my roommates Andy Gill, André Santos, and David King, for providing a varied and stimulating environment in which to do my research.

This work was supported by a grant from the Engineering and Physical Sciences Research Council.

I'd like to express my thanks to the following companies and establishments who indirectly supported me throughout my research and especially during the writing-up period: Guinness Breweries Ltd., Benson & Hedges Ltd., Curlers, Little Italy, and the Ashoka Restaurant, Ashton Lane.

Most of all, I'd like to thank Tinkerbell who gave me more encouragement and support than I could ever need. *Thoir mi mo ghraidh dhuit.*

Cheers!

Contents

Chapter 1

Optimisation by Transformation

1.1 Introduction

The problem of productive software development is frustrated by three conflicting aims: to develop software that is reliable, to develop software that is efficient, and to develop software quickly. Developing reliable software entails an organised approach to software engineering, making use of high level languages, constructs and abstractions. These techniques are also essential if the software is to be portable and extensible. However, a general rule is that the higher the level of generality or abstraction used, the greater the penalty on the efficiency of the program.

Functional programming languages provide a high degree of flexibility and reliability to program developers. Strong type systems give the programmer increased confidence in a program's correctness. Features such as lazy evaluation, abstract datatypes and overloading facilitate efficient software development. Unfortunately, all this imposes a burden on the efficiency of functional programs.

In order to have fast programs while retaining these desirable language features, the onus of optimisation falls to the compiler. We must build compilers that are capable of taking programs written in a clear, abstract style and produce programs that run as efficiently as possible. This raises the question of scope: it is obviously possible to build a compiler that knows, for example, that $x + 0$ is equivalent to x, but can a compiler conceivably translate

an algorithm to improve its complexity?

The transformational approach to program optimisation takes the view that many program improvements can be expressed as source-to-source mutations of the subject program. This is a powerful technique: as we will show later in this chapter, a transformation system that consists of a few simple correctness-preserving transformation rules is capable of providing a framework in which many powerful optimisations can be expressed, including those which alter the complexity of the original algorithm.

This thesis examines one aspect of global program optimisation through transformation: that of the removal of intermediate data structures.

1.1.1 Transparency

Since program optimisation techniques are usually not generally applicable, it is imperative that the programmer knows exactly when his program will be optimised. We call this property *transparency.*

Many small low-level optimisations are generally not transparent, but this is regarded as acceptable because the effect of the optimisation averages out as the size of the program increases. But as specific optimisations become more powerful, the aspect of transparency becomes more important. Non-transparent optimisations tend to produce unpredictable results; a small change in the original program can produce wild variations in performance, as the optimisation changes in applicability.

Because of its subjective nature, transparency is not subject to a rigorous definition; indeed, many optimisations lie somewhere between the fully transparent and opaque extremes. A rule-of-thumb definition is that if an optimisation guarantees to be applicable to programs that meet a syntactic criterion, then it is transparent. This renders most optimisations based on analysis techniques non-transparent, whereas optimisations based on program transformation are often transparent.

There are exceptions to this rule-of-thumb: for example, binding time analysis as used in partial evaluation is close to transparent, because the results of analysis can be used by the programmer to restructure his program.

Programmers are generally more comfortable with purely transformation-based optimisations, because the effect of the optimisation can be determined solely by examination of the source program. For this reason, we consider transparency to be an important factor in the design of optimisation techniques.

1.2 Unfold/Fold Transformations

The unfold/fold transformation system as devised by Burstall and Darlington [BD77] has been the basis for most of the program transformation techniques developed for optimising functional programs.

The system consists of six basic rules which are applied to a set of equations. The initial set of equations will be the program to be transformed, but more equations may be added as transformation takes place. Each equation has a left-hand side of the form $v(e_1 \ldots e_n)$ and a right-hand side which can be an arbitrary expression (this is similar to the pattern matching function definitions seen in many functional programming languages). The rules that can be applied to the program are as follows:

1. *Definition.* A new equation is introduced, with a left-hand side that overlaps with no other equation already in the set.

2. *Instantiation.* A new equation is introduced which is a substitution instance of an existing equation.

3. *Unfolding.* If $e = e'$ and $f = f'$ are equations and f' contains a substitution instance of e, Se, then it may be replaced by Se'.

4. *Folding.* If $e = e'$ and $f = f'$ are equations and f' contains a substitution instance of e', Se', then it may be replaced by Se.

5. *Abstraction.* A **where** clause may be introduced into any equation $e = e'$ by replacing the right-hand side with

$$e'[x_1/f_1 \ldots x_n/f_n] \textbf{ where } (x_1, \ldots, x_n) = (f_1, \ldots, f_n)$$

6. *Laws.* Any laws about primitives used in expressions may be used (for instance: associativity, commutativity, etc.).

These six laws are sufficiently general to transform any program into almost any other. The system is not entirely complete, since it cannot transform functions such as:

$$\begin{aligned} f([]) &= 0 \\ f(x : xs) &= f(xs) \end{aligned}$$

into

$$f(xs) = 0$$

although an extra rule, called *redefinition* is suggested by Burstall and Darlington to allow such transformations to be made.

It should also be noted that the transformation system is only partially correct. By this we mean that the if the result of a transformation is a program that gives a result, this result will be the same as the original program would have given. Unfortunately it is possible, by transformation, to produce a program that is less defined than the original. This can only happen if arbitrary use of the fold step is allowed; for example, if an equation $f(n) = e$ is in the set, then application of the fold rule can replace this with $f(n) = f(n)$, which is obviously a non-terminating function. Several people have proposed solutions to this. Kott [Kot78] showed that correctness was guaranteed provided there the number of unfold steps is always greater than or equal to the number of fold steps, given certain conditions on the transformation. Also, Scherlis [Sch80] proposes a restriction on the use of the fold step that can guarantee total correctness.

Sands [San95b] provides a general improvement theorem which can be used to guarantee correctness of higher-order unfold/fold based program transformation schemes. His theorem can also be used to verify that a particular transformation strictly improves the program.

The generality of this system means that it is possible to obtain, by transformation, a program which is *less* efficient than the original, as well as a program which is more

efficient. This means that in order to transform a program profitably, the transformation steps must be guided either by the user or by a well defined set of strategies (or *tactics*).

The examples presented by Burstall and Darlington are user guided in the sense that they involve a central *eureka* step which allows the subsequent transformation process to achieve the desired result. Some eureka steps can be classified by the type of optimisation they achieve, and this fact can be used to specialise and hence automate a transformation strategy. This is the basis of many proposed automatic transformation techniques [Chi90]. Feather shows how semi-automation of the unfold/fold system can ease the transformation of large systems [Fea79, Fea82].

Burstall and Darlington recognised that many transformations followed a similar style, so they proposed a *strategy* for applying the rules. This goes as follows: make any necessary definitions, instantiate, and unfold repeatedly while trying to apply laws, abstraction and folding. Using this strategy they show that, for instance, the quadratic *nfib* function can be transformed into an equivalent function with linear complexity, and that some list processing functions can be transformed to avoid building intermediate structures.

1.2.1 An Example

Here we give an example of the unfold/fold transformation system in action. The particular example we have chosen is a program fragment which performs a useful computation: find the length of the initial segment of a sequence which specifies a certain predicate. The following expression performs the required function:

$$\mathit{length}\ (\mathit{takewhile}\ p\ (\mathit{iterate}\ f\ 0))$$

Here we have broken the problem into three manageable sub-problems, and used a simple combinator to perform each part. The *length* function takes a list and returns its length. The *takewhile* function takes a list and returns the initial segment whose values all satisfy the predicate p. The *iterate* function generates a list by repeatedly applying a function to an initial value. The definitions of these combinators are given in Figure 1.1.

$$
\begin{array}{ll}
\mathit{iterate}\ f\ x & = x : \mathit{iterate}\ f\ (f\ x) \\
\\
\mathit{takewhile}\ p\ (x : xs) & = \textbf{if}\ (p\ x)\ \textbf{then}\ (x : \mathit{takewhile}\ p\ xs)\ \textbf{else}\ [] \\
\\
\mathit{length}\ [] & = 0 \\
\mathit{length}\ (x : xs) & = 1 + \mathit{length}\ xs
\end{array}
$$

Figure 1.1: Some Definitions

The above program clearly expresses the intention of the programmer, but unfortunately in doing this a serious inefficiency has been introduced. The expression as written generates two intermediate data structures: the list generated by *iterate* which is consumed by *takewhile*, and the result of *takewhile* which is in turn consumed by *length*. The aim of the transformation will be to derive a program that performs the same computation without manipulating intermediate lists.

Figure 1.2 gives a transformation sequence that achieves the desired result. We begin by defining a function g to be equal to our expression. We will explain each step in turn:

1. Define a new function h, which is identical to g except that the second argument to *iterate* has been abstracted. This is the *eureka* step in the sequence.

2. Fold the original expression with respect to h. Our program is now represented by a single call to the new function h.

3. Unfold the call to iterate on the right-hand side of h.

4. Unfold the call to *takewhile* on the right-hand side of h.

5. Use the following law of **if**:

$$f\ (\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2) \rightarrow \textbf{if}\ e\ \textbf{then}\ f\ e_1\ \textbf{else}\ f\ e_2$$

```
                g f p     = length (takewhile p (iterate f 0))                    (1)
define          h f p x   = length (takewhile p (iterate f x))                    (2)
fold            g f p     = h f p 0                                               (3)
unfold          h f p x   = length (takewhile p (x : iterate f (f x)))            (4)
unfold                    = length ( if (p x) then                                (5)
                                       (x : takewhile p (iterate f (f x)))
                                   else
                                       [])
law if                    = if (p x) then                                         (6)
                              (length (x : takewhile p (iterate f (f x))))
                            else
                              (length [])
unfold × 2                = if (p x) then                                         (7)
                              (1 + length (takewhile p (iterate f (f x))))
                            else
                              0
fold                      = if (p x) then                                         (8)
                              (1 + h f p (f x))
                            else
                              0
```

Figure 1.2: Unfold/Fold Example

which is valid provided the function f is strict (i.e., $f \perp = \perp$). Using this law we can push the call to *length* inside the branches of the **if** (because *length* is strict).

6. Unfold each call to *length*.

7. Now we have an instance of the original right-hand side of h, so we can apply the fold rule.

The program fragment has been transformed into a recursive function that performs the function of the combinator composition used in the original expression. Rather than using general components, the program is now *specialised* to the task in hand, and gains a benefit from this: there are no longer any intermediate structures, and the program will use less

store than before. This will translate to real improvements in the observed run-time of the program.

1.3 Removing Intermediate Data Structures

Algorithms which involve many intermediate data structures are common in programs written in a declarative style. Expressions are built up using components or *combinators*, each of which performs a particular function. The example in the previous section is an illustration of this methodology.

This kind of modularity in program design is seen not just at the lowest level demonstrated in the example, but throughout the structure of many programs. Each part of a program is built using smaller components, which aids readability, extensibility and software reuse but is detrimental to performance. By rewriting a program in order to remove the inefficiencies imposed by its modularity, the programmer is likely to degrade the clarity and maintainability of his program. As modern software engineering techniques dictate that programs are developed so as to be easily maintainable, the burden falls to the compiler writer to lessen the impact on efficiency.

Methods to automatically remove intermediate data structures (especially lists) have been proposed by several researchers. Wadler [Wad81] proposed a small set of list combinators (equivalent to *map*, *foldl* etc.) and showed that any composition of these combinators could be effectively reduced to a single function application which wouldn't build any intermediate structure. Augustsson [Aug87] describes a method to eliminate the intermediate sequence of values in list comprehensions [Tur82] of the following common form:

$$[\, e \mid p \leftarrow [n...m] \,]$$

It is also well known that by abstracting over the **nil** component of a list we can achieve constant time list append [Hug84], effectively removing intermediate lists from chains of appends. Wadler [Wad87] uses this technique to suggest a systematic transformation that can remove calls to *append* in functional programs (for example, he shows how to transform

$$
\begin{array}{lll}
tt & : & x \\
 & | & f\ x_1 \ldots x_n \\
 & | & C\ tt_1 \ldots tt_n \\
 & | & \textbf{case}\ tt\ \textbf{of}\ \{p_1 \rightarrow tt_1\,; \ldots ;\ p_n \rightarrow tt_n\}
\end{array}
$$

Figure 1.3: Treeless Form

a quadratic-time *reverse* function into a linear version). Burge [Bur77] goes one step further by abstracting over both **cons** and **nil**, allowing simple list processing functions to be fused together (a similar technique has been used to perform intermediate list removal by Gill et.al., see Section 1.4.2).

1.4 Deforestation

Wadler's deforestation algorithm grew from his work on Listlessness [Wad84, Wad85]. It is a transformation system based on the unfold/fold strategy of Burstall and Darlington, designed to automatically remove intermediate data structures ("trees") from functional programs. It can be viewed as a strategy for applying the unfold/fold rules that can be performed automatically, repackaged as a stand-alone transformation algorithm.

Deforestation applies to first order functional programs composed of functions written in a certain form, called *treeless form*, shown in Figure 1.3. As suggested by the name, a function in treeless form is guaranteed never to generate any intermediate data structures (i.e. trees). An additional restriction on functions in treeless form is that function arguments must be *linear* (no argument can be referred to more than once in the body of the function).

The deforestation theorem can be expressed thus: the deforestation algorithm can remove all intermediate data structures from an expression involving only calls to treeless functions with definitions in treeless form. The algorithm therefore takes as input a program consisting of treeless functions and a non-treeless term, and yields a program consisting of treeless functions and a treeless term.

$$
\begin{aligned}
\mathcal{T}\ x &= x \\
\mathcal{T}\ (C\ t_1 \ldots t_n) &= C\ (\mathcal{T}\ t_1) \ldots (\mathcal{T}\ t_n) \\
\mathcal{T}\ (f\ t_1 \ldots t_n) &= \mathcal{T}\ (t[t_1/x_1, \ldots, t_n/x_n]) \\
&\quad \text{where } f \text{ is defined by } f\ x_1 \ldots x_n = t \\
\mathcal{T}\ (\textbf{case}\ x\ \textbf{of}\ \{p_1 \rightarrow t_1\,; \ldots; p_n \rightarrow t_n\}) & \\
&= \textbf{case}\ x\ \textbf{of}\ \{p_1 \rightarrow \mathcal{T}\ t_1\,; \ldots; p_n \rightarrow \mathcal{T}\ t_n\} \\
\mathcal{T}\ (\textbf{case}\ (f\ t_1 \ldots t_n)\ \textbf{of}\ \{p_1 \rightarrow t'_1\,; \ldots; p_n \rightarrow t'_n\}) & \\
&= \mathcal{T}\ (\textbf{case}\ t[t_1/x_1, \ldots, t_n/x_n]\ \textbf{of}\ \{p_1 \rightarrow t'_1\,; \ldots; p_n \rightarrow t'_n\}) \\
&\quad \text{where } f \text{ is defined by } f\ x_1 \ldots x_n = t \\
\mathcal{T}\ (\textbf{case}\ (C\ t_1 \ldots t_n)\ \textbf{of}\ \{\ldots;\ C\ x_1 \ldots x_n \rightarrow t\ ; \ldots\}) & \\
&= \mathcal{T}\ (t[t_1/x_1, \ldots, t_n/x_n]) \\
\mathcal{T}\ (\textbf{case}\ (\textbf{case}\ t_0\ \textbf{of}\ \{p'_1 \rightarrow t'_1;\ \ldots;\ p'_n \rightarrow t'_n\})\ \textbf{of}\ \{p_1 \rightarrow t_1\,; \ldots; p_n \rightarrow t_n\}) & \\
&= \mathcal{T}\ (\textbf{case}\ t_0\ \textbf{of} \\
&\qquad p'_1 \rightarrow \textbf{case}\ t'_1\ \textbf{of}\ \{p_1 \rightarrow t_1\,; \ldots; p_n \rightarrow t_n\} \\
&\qquad p'_n \rightarrow \textbf{case}\ t'_n\ \textbf{of}\ \{p_1 \rightarrow t_1\,; \ldots; p_n \rightarrow t_n\})
\end{aligned}
$$

Figure 1.4: Deforestation Algorithm

Informally, the algorithm works as follows. The transformation function (shown in Figure 1.4) is applied to the initial composition of functions, which continuously unfolds function calls while applying reduction and commuting conversion transformations to the expression. If at any time a sub-expression occurs which is a *renaming* of an expression previously transformed, then the fold step is triggered. The fold step generates a new recursive function (which is guaranteed to be in treeless form). The algorithm continues until no more sub-expressions can be transformed, when there will exist an expression in treeless form and a new set of treeless functions. By the definition of treeless form, this means that the result must include no intermediate data structures.

Termination of the algorithm is based solely on the fact that the fold step will always occur eventually. Treeless form is formulated in such a way that this will always happen, and Wadler provides an outline of a proof of termination in his paper. A more complete proof of termination is presented by Ferguson and Wadler [FW89].

Unfortunately treeless form is a very restrictive language, both in its form and expressiveness. It is first-order only, whereas most modern functional programming languages are higher order. The two other restrictions, that arguments to a function application can only be variables, and that function arguments must be linear, mean that it isn't possible to write any arbitrary function in treeless form.

Wadler goes some way to removing these restrictions in his paper, using two methods:

- *Blazing* relaxes the restriction on variable-only arguments by allowing expressions of numeric (or other non-recursive) type in argument positions. These expressions can only generate a simple result, and as such don't really count as intermediate data structures. In the blazed deforestation algorithm (an extension of the basic algorithm) expressions that are blazed minus are not transformed further, but expressions blazed plus are processed as normal. Blazing extends the range of functions that can be used as input to the deforestation algorithm, but still fails to encompass all possible functions.

- A well-known technique called *higher order macros* enables certain functions with arguments of function type to be represented as macros by abstracting the offending arguments. This enables higher order functions such as *map* to be represented in first order treeless form.

The remaining restriction, that all function arguments must be linear, is present to ensure that the deforestation algorithm cannot produce a program that is *less* efficient than the original. This can happen during the unfold phase, where a function application is replaced by an instance of the function definition. If an argument were allowed to occur multiple times in the body of the function, then expression duplication could occur as a result of unfolding. This can cause an arbitrarily large computation to be duplicated at run-time, thus the restriction.

The prospect of lifting these restrictions further, and obtaining an algorithm that performs deforestation on a wider variety of programs, is the subject of much of the work that has been inspired by Wadler's deforestation, which is discussed in the next section.

An attractive property of the deforestation algorithm is that it is completely transparent. By a syntactic restriction on the input, deforestation ensures that intermediate data structures can always be removed. As deforestation is generalised, this property is difficult to maintain, a concern which has so far not been addressed acceptably.

Example

Without going into the details of the transformation, deforestation produces the following result given the example program presented in Section 1.2.1 (with suitable redefinitions of the functions in blazed treeless form):

$$
\begin{array}{l}
h\ f\ p\ 0 \\
\text{where}\quad h\ f\ p\ x = \textbf{case}\ p\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{True} \rightarrow 1 + h\ f\ p\ (f\ x) \\
\qquad\qquad\qquad\qquad \mathit{False} \rightarrow 0
\end{array}
$$

which is equivalent to the result derived by the unfold/fold style transformation.

Although the result of optimisation is a program that is obviously more efficient than the original, the overall complexity of the algorithm remains the same. This is true for deforestation in general, which can only achieve constant factor speedups. However, as we will show later in this thesis, the constant factor can be large enough for the optimisation to produce significant increases in efficiency.

1.4.1 Extensions to deforestation

Chin recognised in his thesis that Wadler's deforestation could be generalised to cover a wider range of functions by extending the definition of treeless form and modifying the algorithm to accommodate these changes [Chi90].

Firstly, the definition of treeless form was changed to allow functions that had one or more non-linear arguments. Chin extended the concept of blazing to function parameters,

blazing a parameter minus if it was non-linear. He also relaxes the restriction on variable-only function arguments by blazing a parameter minus if a call to the function exists with a non-variable expression in the same position.

The textual linearity criterion of treeless form is replaced in Chin's extended treeless form by a definition based on Sharing Analysis [HG85]. Non-linear arguments are subject to substitution if and only if the argument can only be evaluated once at execution time. This particular extension means that the algorithm gains some power at the expense of transparency.

Finally, Chin extends deforestation to cover *all* first order functions by introducing the **let** construct to his language, to indicate the presence of an intermediate data structure that the deforestation algorithm should not attempt to remove. Thus all functions can be converted into treeless form by the introduction of **let**. Chin doesn't attempt to identify how **let**s should be added to functions. An intermediate structure that is normally removable by the deforestation algorithm can become residual without proper control on the use of the **let** construct.

Chin also extends his deforestation algorithm to higher-order functions by a process of *higher-order removal* combined with some extensions to the algorithm (which essentially ignore residual higher-order features in the input). We consider this method, while valid, to be unacceptable for several reasons: it increases the size of the program, adds complexity to the deforestation algorithm, and results in a loss of transparency.

Chin further extends his ideas giving a generalised annotation scheme for first-order deforestation that uses a producer-consumer model to determine when a certain fusion is *safe* (i.e. will terminate). A proof of termination is given.

Chin doesn't comment on the scope of his technique as a whole, nor on the effect of his generalised deforestation on real example programs.

Hamilton [Ham93] also extends first order deforestation by relaxing the definition of treeless form. Like Chin, he recognises that textual linearity is too strong a criteria to avoid duplication of work, and introduces a sharing analysis to more accurately detect non-linearity.

Hamilton also relaxes treeless form by showing that not all non-variable function argu-

ments represent intermediate data structures, and indeed some of these can be deforested successfully. The impact of this is that function arguments can be non-variable provided that the called function doesn't decompose the structure in the equivalent argument position. An example is the second argument to *append*, and this renders the definition of *concat* in treeless form. Hamilton's usage analysis also performs the function of detecting these non-intermediate structures.

Hamilton also adds the **let** construct to his language in order to be able to express all functions in treeless form. His **let** construct is identical to Chin's, in that it indicates the presence of an intermediate structure which will not be removed by the deforestation algorithm.

Hamilton attempts only the theoretical study of his extended deforestation, without commenting on the practicalities. Some recent work by Hamilton [Ham95] gives a new deforestation algorithm for a higher-order language, and gives a termination proof. Hamilton's work influenced the author in formulating the material presented in Chapter 2 (see Section 2.1).

Sørenson, Glück and Jones [SGJ94], extended deforestation in order to be able to derive optimal Knuth-Morris-Pratt [KMP77] specialised pattern matchers from a general matching algorithm. Their extension involves adding a rule to the basic deforestation algorithm to enable this partial-evaluation style optimisation to take place.

Sands provides some insight into proving the semantic correctness of unfold/fold style transformations, and uses his method to prove that a higher-order variant of deforestation is correct [San95b, San95a].

1.4.2 `foldr/build` Deforestation

Gill, Launchbury and Peyton Jones [GLJ93, Gil95] take an entirely different approach to deforestation. The motivation of their approach is to achieve deforestation through a minimum of effort, sacrificing some generality along the way.

The technique is based around two combinators. The first is the all-purpose list reducing function *foldr*, which encapsulates a standard way in which lists are consumed (incidentally,

foldr has nothing to do with the *fold* rule of Burstall and Darlington!). It can be defined as follows:

$$\begin{aligned} foldr\ f\ c\ [] &= c \\ foldr\ f\ c\ (x : xs) &= f\ x\ (foldr\ f\ c\ xs) \end{aligned}$$

Many list consuming functions can be written using *foldr*, including *sum*, *append*, and *map*. The second combinator is in some ways the partner of *foldr*, called *build*, with the following definition:

$$build\ g = g\ (:)\ []$$

Many list producing functions can be abstracted over **cons** and **nil** by redefining them as applications of *build*. For example, *map* can be defined as follows:

$$map\ f\ xs = build(\lambda c.\ \lambda n.\ foldr(\lambda a.\ \lambda b.\ c\ (f\ a)\ b)\ n\ xs$$

The advantage of this is that lists built using *build* and consumed using *foldr* can be removed using the *foldr/build* rule:

$$foldr\ f\ c\ (build\ g) = g\ f\ c$$

The *foldr/build* rule by itself isn't valid for all values of g. It is only valid for those lists which are truly abstracted over their list constructors, that is only values of g that satisfy the following polymorphic type:

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

This restriction is enough to show that the *foldr/build* rule is valid, using the "free theorem" concept [Rey83, Wad89].

The *foldr/build* rule embodies the essence of removing intermediate lists. Whenever an instance of the left-hand side of the *foldr/build* rule occurs in a program, it indicates that an intermediate list can be eliminated, by replacing it with the right-hand side.

The main argument in favour of using this method for deforestation is its simplicity. There is no need for the unfold/transform/fold strategy of classical deforestation, and the associated complex termination proofs. The disadvantage is the loss of generality. All functions to be deforested must be defined in terms of *foldr* and *build* (although some work has been done on performing this transformation automatically [LS95, Gil95]). Difficulties arise with functions such as *zip* which take multiple list arguments, all of which can be deforested successfully with classical deforestation. Other difficulties are presented by functions such as *foldr*1 (a version of *foldr* which takes a list of at least length one), which treat certain cells in the list differently from others.

A more detailed comparison of *foldr/build* deforestation with classical deforestation is left until Section 7.2.2.

1.5 Contribution of Thesis

This work should be viewed as an exploration into the topic of deforestation. In this section we list the main contributions of the thesis.

- The major contribution of the thesis is to show how deforestation can be performed for arbitrary higher order functional programs. We give a deforestation algorithm for programs written in a language based on the lambda calculus, which satisfies the transparency property.
- We identify the conditions which must be satisfied by the input to the deforestation algorithm, and show how these conditions ensure termination of the algorithm.
- We relate deforestation to the cut-elimination principle of logic. By merging a cut-elimination algorithm and simple first-order deforestation, we obtain a new higher-order deforestation algorithm that has advantages in generality over the previous algorithm.

- We show how arbitrary functions can be converted automatically into the form required by deforestation. The conversion process is optimal in the sense that converted functions allow the greatest amount of intermediate data structure removal.

- We describe the problem of ensuring that transformed programs are no less efficient than the original. Several rules are given concerning the form of the input to deforestation and other transformations performed both before and after deforestation that must be adhered to to ensure safety.

- The transparency property of our deforestation algorithm is described in detail. We show how to identify which intermediate structures will be removed for any given input program.

- We design a model for implementing the deforestation optimisation on a real functional language, in this case Haskell. The model involves a compromise between user intervention and automatic optimisation that allows certain classes of intermediate lists to be removed automatically, with more deforestation being possible if the programmer explicitly directs the deforestation algorithm through the use of annotations.

- We implement the deforestation implementation in the Glasgow Haskell Compiler. The details of the implementation are given, many of which are independent of the compiler and language used.

- We propose (and implement) certain extensions to the basic deforestation scheme that were discovered through experimentation to be essential when applying the deforestation implementation to larger programs.

- We apply the prototype implementation to several example programs. Results are given, and we also show how to maximise the benefit of deforestation by annotating the program.

- Finally, we identify several avenues for future research.

1.6 Structure of Thesis

Chapter 2

In Chapter 2 we describe a complete deforestation algorithm for a lambda-calculus based language. The transformation and knot-tying (new function generation) algorithms are described separately, and we provide a proof of termination for the algorithm as a whole. The algorithm presented in this chapter has some shortcomings, which are dealt with in the rest of the thesis.

Chapter 3

In Chapter 3 we take a principle from logic, namely cut elimination, and show how it relates to deforestation. We propose a new language formulation derived from both sequential calculus and natural deduction, give a cut elimination algorithm for this language, and prove termination for it. We then present a deforestation algorithm for a simple first order recursive language, formulated in the same style as the cut elimination algorithm. Finally, these two languages and algorithms are combined to yield a higher order deforestation algorithm. We provide a proposition of termination, arguing that given certain restrictions on the input (which are less restrictive than those of the algorithm in Chapter 2), the algorithm will terminate.

Chapter 4

Chapter 4 looks at some issues related to deforestation that are necessary for deforestation to be put into practice. The first, automatic conversion to treeless form, enables arbitrary recursive functions to be automatically translated into the treeless form language required for deforestation. This translation involves the introduction of some residual data structures (intermediate data that will not be removed by the deforestation). Because our transformation is transparent, the residual data structures introduced by this process are easily identified, should this be required.

The second issue treated in this chapter is that of linearity. The linearity criterion is required to ensure that deforestation does not generate a program that is less efficient

than the original: this can happen if expressions are duplicated during transformation, for example. This subject is treated as a separate issue, since it has no impact on the operation or termination of the deforestation algorithm itself.

Finally, we examine the transparency properties of our deforestation algorithm. We show how the programmer can identify which parts of a program will be optimised, and in particular, exactly which data structures will be removed by deforestation.

Chapter 5

In Chapter 5 we give a detailed description of our prototype deforestation implementation in the Glasgow Haskell Compiler, showing how each part of the system can be implemented practically and efficiently. We also provide some optimisations and improvements to the basic scheme that were developed as a result of experimenting with early versions of the prototype.

Chapter 6

In Chapter 6 we give the results obtained from applying deforestation to a number of example programs, each displaying a different facet of the scope of deforestation as an optimisation strategy. We also provide a discussion of the practicalities of deforestation in a real-world compiler, and give some suggestions for improvements to the optimisation scheme.

Chapter 7

In Chapter 7 we give our conclusions, and describe some topics for future research.

Chapter 2

Higher-Order Deforestation

In this chapter we will present a new deforestation algorithm that removes intermediate data structures from arbitrary higher-order functional programs. The approach taken is to start from scratch: instead of extending the first-order deforestation algorithm to accommodate higher-order constructs, we take the view that the lambda calculus is the core of a higher-order programming language, and not simply an extension of first-order recursion equations.

Our goal can be stated thus: to develop an algorithm for removing intermediate data structures from arbitrary higher-order, lazy, purely functional programs. The decision to apply the algorithm to lazy functional languages (i.e. languages where the evaluation strategy is normal order, and the values of subexpressions are updated once evaluated) is motivated mainly by preference, whereas the requirement for pure functionality is essential: transformation techniques become significantly more complicated in the presence of non-referentially transparent language features.

The above goal leads to a number of subgoals:

- The algorithm should be generally applicable. That is, it should apply to all programs, including those that contain data structures which cannot be removed by the deforestation process.
- The algorithm should be transparent. It should be obvious by examining the subject

program which data structures will be removed by the deforestation process and which will be left in place.

- The algorithm should lead naturally to an implementation with acceptable efficiency (suitable for an optimising compiler).

The algorithm presented in this chapter meets all three of these goals, but nevertheless has a number of shortcomings which will be addressed in the rest of this thesis.

2.1 History

The work presented in this chapter is inspired to some extent by Hamilton, whose recent work on deforestation [Ham95] provoked the author to re-evaluate some old research in a new light.

The deforestation algorithm given in this chapter had been discovered some time earlier, but had been dismissed as having no great advantages over the algorithm we had previously been using [MW92]. Hamilton discovered a similar algorithm independently, and it was his work that inspired us to go back to our algorithm and attempt the termination proof. The resulting proof was somewhat simpler than for the algorithm we had previously been working with.

The main difference between our algorithm and Hamilton's is that Hamilton uses blazing to indicate residual data structures and non-linear function arguments, whereas we use the **let** construct (see Section 2.2.1). Modulo this difference, our definition of treeless form is identical to that of Hamilton.

2.2 Syntax and Semantics

We now introduce the syntax and semantics of the language that our deforestation algorithm will apply to (Figure 2.1). The language we have chosen is the lambda calculus with some straightforward extensions: explicit recursion at the top level using **letrec**,

t, u, v	::=	x	variable
	|	$\lambda x.\, u$	lambda abstraction
	|	$t\ ts$	application
	|	$C\ ts$	constructor application
	|	**case** u **of** *alts*	case expression
	|	**let** $x = t$ **in** u	let expression
xs	::=	$x_1 \ldots x_n$	sequence of variables
ts	::=	$t_1 \ldots t_n$	sequence of terms
alts	::=	$\{C_1\ xs_1 \rightarrow v_1; \ldots; C_n\ xs_n \rightarrow v_n\}$	alternatives
defs	::=	$\{f_1 = t_1;\ \ldots;\ f_n = t_n\}$	declarations
prog	::=	**letrec** *defs* **in** u	program

Figure 2.1: The Language

non-recursive **let** expressions (these have a special meaning to our algorithm, see later), and constructs for building and examining algebraic data structures. This language also represents the smallest subset of functional languages such as Haskell [HPW+92], that the rest of the language can be translated into.

Algebraic data structures are identical to the user definable data structures of Haskell. An algebraic datatype is a sum-of-products; in general, an algebraic datatype consists of a number of *constructors*, each with a number of arguments. Enumerated types are a special case of algebraic datatypes—a set of constructors with no arguments. Similarly, product types consist of a single constructor with an argument for each element of the product. For example, the list type can be defined thus:

$$\begin{aligned} \mathit{List}\ \alpha &= \mathit{Nil} \\ &\mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha) \end{aligned}$$

where *Nil* and *Cons* are the constructors. *Nil* has no arguments, and *Cons* has two: the list element (of type α), and the rest of the list (of type $\mathit{List}\ \alpha$).

A structure is built by applying a constructor to the correct number of arguments. Partially applied constructors can be simulated with functions:

$$cons = \lambda x.\, \lambda xs.\, Cons\ x\ xs$$

Data structures are deconstructed and examined with a **case** expression. A **case** expression consists of a *selector* (an arbitrary expression) and a set of *alternatives*, one for each constructor in the datatype. Each alternative consists of a *simple pattern* (a constructor fully applied to variables) and an expression. The semantics can be expressed informally as follows: the selector is evaluated to weak head normal form, and the top level constructor is matched with the correct alternative. Each of the variables in the pattern is bound to the corresponding field of the data structure, and the expression on the right of the alternative is evaluated in the presence of these new bindings.

We also require that programs are well typed, and that any recursive types in the program are defined using algebraic data types, as in the list example above. These type restrictions correspond to the type system of Haskell and similar functional languages.

There are some constructs missing from this simple language that one would expect to find in a real programming language. Firstly, primitive objects such as integers and operations over them. These can be simulated using algebraic data types, so we do not include in the basic language. Secondly, nested **letrec**: the deforestation algorithm can be extended quite straightforwardly to handle programs with nested **letrec**s, as we have done for our prototype implementation. Nested **letrec** expressions are treated in Section 5.6.2.

The denotational semantics of the language are specified by the following reduction rules:

$$\begin{array}{ll}
(\lambda x.\, u)\ (t : ts) & \Rightarrow u[t/x]\ ts \\
\textbf{case}\ C\ t_1 \ldots t_n\ \textbf{of}\ \{\ldots; C\ x_1 \ldots x_n \rightarrow u; \ldots\} & \Rightarrow u[t_1/x_1, \ldots, t_n/x_n] \\
\textbf{let}\ x = t\ \textbf{in}\ u & \Rightarrow u[t/x] \\
\textbf{letrec}\ \{f_1 = t_1; \ldots; f_n = t_n\}\ \textbf{in}\ u & \Rightarrow u[t'_1/f_1, \ldots, t'_n/f_n] \\
 & \quad \text{where} \quad t'_1 = t_1[t'_1/f_1] \\
 & \qquad\qquad \vdots \\
 & \qquad\qquad t'_1 = t_1[t'_1/f_1]
\end{array}$$

The operational semantics of the language is call-by-need (or lazy evaluation). In the rest of this chapter we will use the terms *normal form* and *weak head normal form.* A term in normal form is one where none of the above reduction rules can be applied. A term in weak head normal form is either a lambda expression or a constructor application.

2.2.1 Treeless Form

Treeless form is a subset of the full language presented in the previous section. That is, any term which is in treeless form is also a term in the full language, but not vice-versa.

The essential property of a term in treeless form is that evaluating it will not use any intermediate storage except in certain well defined cases. We define intermediate storage as those structures generated (and possibly also consumed) by a computation, that do not appear directly in the result.

In order to define treeless form we need to distinguish two types of variable: those bound by a definition in the top-level **letrec**, and all others. These two types of variables are distinguished because they are treated separately by the algorithm to follow. When this distinction is relevant, we will refer to the **letrec** bound variables by the identifiers f g h and the others by x y z.

There are two forms of application in the treeless form syntax: application of a **letrec**-bound variable, and application of a normal variable. The argument list in both cases must consist of variables only, but it can be empty.

Consider the definition of treeless form in Figure 2.2, but ignore for now the **let** construct. Terms in this form are in normal form with respect to the reduction rules for the language. Hence, evaluating a term in this form would succeed immediately returning the original term, using no intermediate storage. Now add the **let** construct, and we have removed the normal-form property. The essential concept captured here is that *the only intermediate structures built by a term in treeless form are those indicated explicitly by the* **let** *construct.*

In contrast to the treeless form of Wadler [Wad90b], our treeless form allows intermediate data structures, but we are making it clear exactly where these intermediate structures exist.

tt, tu, tv	::=	$z\ xs$	application
	|	$f\ xs$	recursive function application
	|	$\lambda x.\ tu$	lambda
	|	**case** x **of** $talts$	case expression
	|	$C\ tts$	constructor application
	|	**let** $x = tt$ **in** tu	let expression
xs	::=	$x_1 \ldots x_n$	sequence of variables ($n >= 0$)
tts	::=	$tt_1 \ldots tt_n$	sequence of terms ($n >= 0$)
$talts$	::=	$\{C_1\ xs_1 \rightarrow tv_1; \ldots; C_n\ xs_n \rightarrow tv_n\}$	alternatives ($n > 0$)
$tdefs$	::=	$\{f_1 = tt_1; \ldots; f_n = tt_n\}$	declarations ($n > 0$)
$tprog$	::=	**letrec** $tdefs$ **in** u	program

Figure 2.2: Syntax of Treeless Terms

Also, our treeless form is sufficiently general that any term in the full language can be rewritten in treeless form, usually by the judicious addition of **let** to ensure that the argument to any application is a variable. This raises another important point: there is normally more that one way that a term may be converted into treeless form, but is one way better in any sense than another? And if so, is there an optimal translation? Well, the purpose of the **let** construct is to allow intermediate data structures that cannot be removed by deforestation to be present in the program. Thus, the deforestation algorithm presented later will not remove data structures protected by the **let** construct (it will remove all others, with one exception that is discussed later). It is entirely possible to prevent all useful deforestation from taking place by over-use of the **let** construct. However, as we will show in Section 4.1, there is an optimal translation that can be performed automatically.

A program in treeless form is precisely the input that our transformation system will expect. It consists of a set of treeless recursive definitions and an arbitrary (non-treeless) expression. The only restriction on the expression is that it must be obtainable by repeated substitutions of treeless terms for the free variables of an initially treeless term (i.e. the term must have an *order*, see Section 2.6). It is straightforward to check that a given term satisfies this restriction.

This criteria covers the majority of terms including all terms composed of just applications and variables. However, it disallows terms such as

$$\lambda x.f\ (x\ x)$$

We cannot express this term as a treeless substitution without making use of name-capture, so this term is ruled out as input to the deforestation algorithm. However, if we rewrite the term using a **let** expression, as

$$\lambda x.\,\mathbf{let}\ y = x\ x\ \mathbf{in}\ f\ y$$

then it is admissible (although we have lost the possibility of eliminating the intermediate structure between f and its argument by introducing the residual **let**). It is always possible to transform expressions that cannot be represented as treeless substitutions into valid ones by addition of suitable **let** expressions. A possible algorithm would be to identify non-variable function arguments and case selectors in which bound variables appear, and rebind these expressions using a **let** as above.

2.3 The Transformation Algorithm

We will describe the higher-order deforestation algorithm in two stages: this section describes the transformation process, while the next section describes the termination conditions and the generation of new recursive definitions.

The transformation is given in Figure 2.3. It is a recursive operation $\mathcal{T}$ from terms to treeless terms. The recursive definitions bound by the top-level **letrec** are represented by a mapping D, from variables to expressions. Strictly speaking, we should write $\mathcal{T}_D$, since $\mathcal{T}$ also depends on D, but the subscript has been omitted for readability. The only rules which use D are $\mathcal{T}1$ and $\mathcal{T}7$, which replace recursive function variables with their definitions.

The $\mathcal{T}$ operation examines the head of the applicative expression on its input. If the head

$$\mathcal{T}(f\ ts) = \mathcal{T}(u\ ts) \text{ where } (f = u) \in D \quad (\mathcal{T}1)$$

$$\mathcal{T}(x\ t_1 \ldots t_n) = \textbf{let } z_1 = \mathcal{T}\, t_1 \textbf{ in } \ldots \textbf{let } z_n = \mathcal{T}\, t_n \textbf{ in } x\ z_1 \ldots z_n \quad (\mathcal{T}2)$$

$$\mathcal{T}(\lambda x.\, u) = \lambda x.\, \mathcal{T}\, u \quad (\mathcal{T}3)$$

$$\mathcal{T}((\lambda x.\, u)\ t_1 \ldots t_n) = \mathcal{T}(u[t_1/x]\ t_2 \ldots t_n) \quad (\mathcal{T}4)$$

$$\mathcal{T}(C\ t_1 \ldots t_n) = C\ (\mathcal{T}\, t_1) \ldots (\mathcal{T}\, t_n) \quad (\mathcal{T}5)$$

$$\mathcal{T}((\textbf{let } x = u \textbf{ in } v)\ ts) = \textbf{let } x = \mathcal{T}\, u \textbf{ in } \mathcal{T}(v\ ts) \quad (\mathcal{T}6)$$

$$\mathcal{T}((\textbf{case } (f\ us) \textbf{ of } alts)\ ts) \quad (\mathcal{T}7)$$
$$= \mathcal{T}((\textbf{case } (v\ us) \textbf{ of } alts)\ ts) \text{ where } (f = v) \in D$$

$$\mathcal{T}((\textbf{case } (x\ u_1 \ldots u_n) \textbf{ of } \{C_i\ xs_i \to v_i\})\ ts) \quad (\mathcal{T}8)$$
$$= \textbf{let } z_1 = \mathcal{T}\, u_1 \textbf{ in}$$
$$\vdots$$
$$\textbf{let } z_n = \mathcal{T}\, u_n \textbf{ in}$$
$$\textbf{let } z_0 = x\ z_1 \ldots z_n \textbf{ in}$$
$$\textbf{case } z_0 \textbf{ of } \{C_i\ xs_i \to \mathcal{T}(v_i\ ts)\}$$

$$\mathcal{T}((\textbf{case } ((\lambda x.\, v)\ u_1 \ldots u_n) \textbf{ of } alts)\ ts) \quad (\mathcal{T}9)$$
$$= \mathcal{T}((\textbf{case } (v[u_1/x]\ u_2 \ldots u_n) \textbf{ of } alts)\ ts)$$

$$\mathcal{T}((\textbf{case } (C\ us) \textbf{ of } \{\ldots;\ C\ xs \to v;\ \ldots.\})\ ts) \quad (\mathcal{T}10)$$
$$= \mathcal{T}(v[us/xs]\ ts)$$

$$\mathcal{T}((\textbf{case } (\textbf{case } v \textbf{ of } \{C_i\ xs_i \to v_i\})\ us \textbf{ of } alts)\ ts) \quad (\mathcal{T}11)$$
$$= \mathcal{T}((\textbf{case } v \textbf{ of } \{C_i\ xs_i \to \textbf{case } v_i\ us \textbf{ of } alts\})\ ts)$$

$$\mathcal{T}((\textbf{case } ((\textbf{let } x = u \textbf{ in } v)\ us) \textbf{ of } alts)\ ts) \quad (\mathcal{T}12)$$
$$= \textbf{let } x = \mathcal{T}\, u \textbf{ in } \mathcal{T}((\textbf{case } (v\ us) \textbf{ of } alts)\ ts)$$

Figure 2.3: Transformation Scheme

of this term is found to be a **case** expression, then further pattern matching is performed on the head of the applicative expression in the selector of the **case**. This makes the $\mathcal{T}$ operation comparatively complex, as it must delve deeply into the input term to find the next reduction to perform. However, compared to an approach which has simpler transformation steps [MW92], this scheme has no nested calls to $\mathcal{T}$. This makes for not only a more efficient implementation, but also an algorithm that has a simpler termination proof.

All the rules in the algorithm consider applicative terms; note that the argument list in each case can be empty (the exceptions are rules $\mathcal{T}4$ and $\mathcal{T}9$ which perform β-reduction). This enables the algorithm to be presented in a concise manner, as we do not have to consider applicative and non-applicative terms separately.

There is an implicit argument list flattening operation in the algorithm: it is assumed that $((t\ us)\ vs)$ is automatically replaced by $(t\ (us +\!\!+ vs))$.

There are a number of places in the transformation where name-capture can occur (a binding can 'capture' a free variable of the same name in an expression if the binding moves outside that expression). We have avoided this problem by assuming that all variable names are unique, again to retain clarity in the rules. This is an implementation problem that is by no means trivial, and will be discussed further in Chapter 5.

Two rules in the algorithm introduce **lets** into the output, namely rules T2 and T8. This is contrary to our intuition, which says that all intermediate data structures except for those bound by **let** in the input will be removed by deforestation. However, the cases where **let** is introduced are always places where we cannot hope to remove the intermediate structure in question, because the variable at the head of the application is bound elsewhere by a lambda or **case** expression, and hence its value is not known. Nevertheless, we consider this introduction of **lets** to be a deficiency in the algorithm. This subject is discussed further in Section 2.7.1, and we propose a solution in the next chapter.

2.4 Knot Tying

The transformation algorithm terminates when a call to $\mathcal{T}$ is made with an argument that is a renaming of one that has occurred before. When this happens, a new recursive definition is made and the looping transformation is replaced by a call to the new function. In the terminology introduced by Burstall and Darlington [BD77], we are defining new functions and *folding* with respect to these new functions. We call this process knot tying.

This explanation will be made more concrete by considering one possible implementation of the procedure. In order to simplify matters, note that it isn't necessary to remember every single call to $\mathcal{T}$ in order to detect renamings. A subset of the rules for $\mathcal{T}$, namely all rules except 1 and 7, terminate all by themselves. It is only application of the unfolding rules which can lead to non-terminating transformations, so calls which invoke these rules need to be memorised and compared against future calls to detect loops.

We can separate the transformation and knot tying stages of deforestation by having the transformation generate an infinite, annotated output, and the knot tying process examining the output to discover loops and making new recursive definitions. This formulation requires the algorithm to be implemented using a lazy functional language, because of the infinite intermediate structure between the two stages.

We need a way to annotate the output from transformation with the history of calls to $\mathcal{T}$. This is done by introducing a new term form, **label**, which has two fields: the first cannot contain any further **label** terms, and the second can contain **labels** but must be treeless. The terms in both fields are semantically identical, and furthermore, the free variables of the second field are a subset of the free variables of the first field (the **label** concept was first proposed in [MW92]).

The idea is to annotate the output by depositing a **label** with the argument to $\mathcal{T}$ in the first field, and the result of transformation in the second. The knot tying process will then descend the output term, collecting the first field of each **label** term and continue by descending into the second field, looking for **labels** that are renamings of those encountered so far.

The transformation algorithm can now be modified to annotate its output. The new rules are:

$$\mathcal{T}\, l@(f\ ts) \ = \textbf{label}\ l\ (\mathcal{T}\,(u\ ts)) \text{ where } (f = u) \in D \qquad (\mathcal{T}1)$$

$$\mathcal{T}\, l@((\textbf{case}\ (f\ us)\ \textbf{of}\ alts)\ ts) \qquad (\mathcal{T}7)$$
$$= \textbf{label}\ l\ (\mathcal{T}\,((\textbf{case}\ (v\ us)\ \textbf{of}\ alts)\ ts)) \text{ where } (f = v) \in D$$

where the syntax $l@t$ is used to provide a single identifier l that refers to the term t, to avoid having to duplicate t on each right-hand side.

Each of the unfolding and reduction rules now annotates the output with a record of the call. To complete the knot tying process, we need to define what we mean by a loop, and how loops are used to generate new recursive definitions.

Definition 1 An expression t is a *renaming* of expression u if and only if there exists a substitution σ from variables to variables (excluding **letrec** bound variables) such that $\sigma u = t$.

Note that the renaming substitution σ may possibly map several different variables to the same result. Now, recall that the first field of a **label** term is identical in meaning to its second field, and contains no other **labels** . We can now define what we mean by a loop:

Definition 2 A *loop* is defined as a term of the form **label** t u where u has at least one subterm of the form **label** t' u', where t' is a renaming of t.

To generate a new set of mutually recursive functions, we first identify loops in the output from the transformer. Each loop is replaced with a new recursive function call as follows: let $x_1 \ldots x_n$ be the free variables (excluding those in D) of the loop **label** t u. The new expression takes the form $f\ x_1\ \ldots\ x_n$, and the following function is added to the list of top-level definitions:

$$f = \lambda x_1. \ldots \lambda x_n.\ u'$$

where u' is the expression u with each looping subterm (**label** t' v where t' is a renaming of t) replaced by $f\ y_1\ \ldots\ y_n$, where the y_i are the free variables of t'. The y_i must be

in the same order as the x_i: that is, if σ is the substitution renaming t to t', then each $y_i = \sigma(x_i)$.

Note that u' may contain further knots to be tied, leading to nested recursion in the output.

The above definition suffices for our proof of termination in Section 2.6, but note that an implementation can perform several optimisations to this simple scheme. For example, similar loops may occur in several branches of the transformed expression and it makes sense to combine these into a single new recursive definition rather than create several that are identical modulo renaming. Additionally, it can sometimes be beneficial to cache *all* calls to $\mathcal{T}$ instead of just those that unfold definitions, because this enables the algorithm to terminate earlier and generate smaller output. An example of this is provided in the next section. The topic of optimising the knot tying process is discussed further in Chapter 5.

2.5 Example

In this section we will examine a particular example of the application of the deforestation algorithm in detail, to provide an insight into how it can be used to remove intermediate data structures from higher-order programs.

Our goal will be to deforest the expression:

$$\lambda xs.\ concat\ (map\ (map\ f)\ xs)$$

using the definitions in Figure 2.4. The intermediate data structure we wish to remove is the entire list of lists produced by $map\ (map\ f)\ xs$ and consumed by *concat*.

Looking at the definitions in Figure 2.4, we can see that two of them are not in treeless form: *foldr*, because the application of f has a non-variable argument, and *concat*, because *append* occurs to the right of an application.

We can render *foldr* in treeless form by the addition of a **let**:

$$
\begin{aligned}
\mathit{append} &= \lambda xs.\,\lambda ys.\,\mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad \mathit{Nil} \rightarrow ys \\
&\qquad\qquad \mathit{Cons}\ x\ xs \rightarrow \mathit{Cons}\ x\ (\mathit{append}\ xs\ ys) \\
\mathit{foldr} &= \lambda f.\,\lambda c.\,\lambda xs.\,\mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad \mathit{Nil} \rightarrow c \\
&\qquad\qquad \mathit{Cons}\ x\ xs \rightarrow f\ x\ (\mathit{foldr}\ f\ c\ xs) \\
\mathit{map} &= \lambda f.\,\lambda xs.\,\mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad \mathit{Nil} \rightarrow \mathit{Nil} \\
&\qquad\qquad \mathit{Cons}\ x\ xs \rightarrow \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ xs) \\
\mathit{concat} &= \mathit{foldr}\ \mathit{append}\ \mathit{Nil}
\end{aligned}
$$

Figure 2.4: Example Definitions

$$
\begin{aligned}
\mathit{foldr} = \lambda f.\,\lambda c.\,\lambda xs.\,&\mathbf{case}\ xs\ \mathbf{of} \\
&\mathit{Nil} \rightarrow c \\
&\mathit{Cons}\ x\ xs \rightarrow \mathbf{let}\ z = \mathit{foldr}\ f\ c\ xs\ \mathbf{in}\ f\ x\ z
\end{aligned}
$$

Because we had to add a **let** to make the function treeless, there must exist a residual data structure (an intermediate structure that is not removed by deforestation) in this definition of *foldr*. In this case, it is part of the result from the function; this topic is discussed further in Section 4.3.

We could use the same technique to write a treeless version of *concat*, but this is not ideal: it would involve rebinding *append* with a **let**, which we'd rather not do since *append* would be residual, and calls to it could never be reduced.

Another technique for taking non-treeless expressions and making them treeless is staring us in the face: apply the deforestation algorithm! To obtain a treeless version of *concat*, we can deforest *foldr append Nil* using a set of definitions D that contains *append* and the treeless version of *foldr*. The transformation in shown in Figure 2.5.

At this point, we can knot-tie and make some new definitions. The first call to $\mathcal{T}$ has

$$\mathcal{T}\,(\mathit{foldr}\ \mathit{append}\ \mathit{Nil})$$

$$= \mathcal{T}\,((\lambda f.\,\lambda c.\,\lambda xs.\,\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to c;\ \mathit{Cons}\ x\ xs \to \mathbf{let}\ z = \mathit{foldr}\ f\ c\ xs\ \mathbf{in}\ f\ x\ z)\ \mathit{append}\ \mathit{Nil})$$

$$= \mathcal{T}\,(\lambda xs.\,\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to \mathit{Nil};\ \mathit{Cons}\ x\ xs \to \mathbf{let}\ z = \mathit{foldr}\ \mathit{append}\ \mathit{Nil}\ xs\ \mathbf{in}\ \mathit{append}\ x\ z)$$

$$= \lambda xs.\,\mathcal{T}\,(\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to \mathit{Nil};\ \mathit{Cons}\ x\ xs \to \mathbf{let}\ z = \mathit{foldr}\ \mathit{append}\ \mathit{Nil}\ xs\ \mathbf{in}\ \mathit{append}\ x\ z)$$

$$= \lambda xs.\,\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to \mathcal{T}\,\mathit{Nil};\ \mathit{Cons}\ x\ xs \to \mathcal{T}\,(\mathbf{let}\ z = \mathit{foldr}\ \mathit{append}\ \mathit{Nil}\ xs\ \mathbf{in}\ \mathit{append}\ x\ z)$$

$$= \lambda xs.\,\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to \mathit{Nil};\ \mathit{Cons}\ x\ xs \to \mathcal{T}\,(\mathbf{let}\ z = \mathit{foldr}\ \mathit{append}\ \mathit{Nil}\ xs\ \mathbf{in}\ \mathit{append}\ x\ z)$$

$$= \lambda xs.\,\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to \mathit{Nil};\ \mathit{Cons}\ x\ xs \to \mathbf{let}\ z = \mathcal{T}\,((\lambda f.\,\lambda c.\,\lambda xs.\,\ldots)\ \mathit{append}\ \mathit{Nil}\ xs)\ \mathbf{in}\ \mathcal{T}\,(\mathit{append}\ x\ z)$$

$$= \lambda xs.\,\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to \mathit{Nil};\ \mathit{Cons}\ x\ xs \to \mathbf{let}\ z = \mathcal{T}\,(\mathbf{case}\ xs\ \mathbf{of}\ \mathit{Nil} \to \mathit{Nil};\ \mathit{Cons}\ x\ xs \to \mathbf{let}\ z = \mathit{foldr}\ \mathit{append}\ \mathit{Nil}\ xs\ \mathbf{in}\ \mathit{append}\ x\ z)\ \mathbf{in}\ \mathcal{T}\,(\mathit{append}\ x\ z)$$

Figure 2.5: Converting *concat* to treeless form

occurred before, and we can generate a new function from this. The call $\mathcal{T}$ (*append* x z) will simply yield an identical definition of *append* modulo renaming, since it is applied to only variables and no intermediate data structure removal can take place. Thus, we won't bother to expand this any further, and just leave the call to *append* in place.

After knot tying, we have the following definition of *concat*:

$$
\begin{array}{lll}
h & = \lambda xs.\,\textbf{case}\ xs\ \textbf{of} & \\
 & \qquad Nil & \rightarrow Nil \\
 & \qquad Cons\ x\ xs & \rightarrow \textbf{let}\ z = h\ xs\ \textbf{in}\ append\ x\ z \\
concat & = \lambda xs.\, h\ xs &
\end{array}
$$

This is an almost optimal definition of *concat*, save for the residual **let** that prevents the output from *concat* being deforestable. The **let** is there as a result of the definition of *foldr*, which required a **let** to be represented in treeless form. However, there is an alternative definition of *concat* that has no **let**, so this can be considered a failure, albeit one which is easily rectified. In fact, since *append* doesn't call h, we can deforest the expression *append* x (h xs) and substitute the result for the subexpression (**let** $z = h\ xs$ **in** *append* x z) in the definition of h, to remove this intermediate structure. This yields the optimal definition of *concat*. The details are omitted here, since the above definition suffices for our example.

As an aside, note that if we don't cache all calls to $\mathcal{T}$ then the call which we used to tie the knot above wouldn't be recognised, but a later call would be. This is an example of the tradeoff between the amount of caching performed and the size of the resulting code mentioned in the last section.

Also, note that we could have used this technique for finding the treeless version of *foldr*, and it would have yielded the same result that we achieved above by the manual insertion of a **let**. In order to do this, we would have to treat *foldr* as a free variable when transforming the body of *foldr* itself.

The technique cannot be used in general, however, because between the stages of applying the deforestation algorithm to the non-treeless function and using the new definition for further deforestation, we need to modify the contents of D, which may render the definition

non-treeless again. A separate treeless-form conversion process is required (more about this later).

Now, let's go ahead and deforest our goal expression, using the new definition of *concat* and its subsidiary function h. The steps in the transformation are given in Figures 2.6 and 2.7. At this point, we have two calls to $\mathcal{T}$ that have occurred before. Knot tying produces the following program:

$$
\begin{array}{l}
\textbf{letrec} \\
\quad g1 \;\; = \lambda f.\, \lambda xs.\, \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \qquad\quad \rightarrow Nil \\
\qquad\qquad\qquad\qquad Cons\ x\ xs \rightarrow \textbf{let}\ z = g1\ f\ xs\ \textbf{in}\ g2\ f\ x\ z \\
\quad g2 \;\; = \lambda f.\, \lambda xs.\, \lambda z.\, \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad Nil \qquad\quad \rightarrow z \\
\qquad\qquad\qquad\qquad\qquad Cons\ x\ xs \rightarrow Cons\ (f\ x)\ (g2\ f\ xs\ z) \\
\textbf{in}\ \lambda xs.\, g1\ f\ xs
\end{array}
$$

The result is a treeless program, and hence it builds no intermediate structure. We can see by examining the code that the only list built is the result of the computation itself.

In this example we have demonstrated the power of the higher-order deforestation algorithm by showing how non-treeless functions can be made treeless by application of the deforestation algorithm, and how deforestation can be performed on expressions that use generic higher-order functions such as *map* and *foldr*.

2.6 Deforestation Theorem

In order to prove that our deforestation algorithm terminates, we shall show that the size of terms occurring in recursive invocations of the $\mathcal{T}$ operation is a bounded quantity. If the size of these terms is bounded, then because we only have a finite number of recursive function variables, constructors and case alternatives available, the $\mathcal{T}$ operation will eventually discover a renaming of a previous call and terminate.

$$
\begin{aligned}
&\mathcal{T}\,(\lambda xs.\, concat\ (map\ (map\ f)\ xs))\\
&= \lambda xs.\, \mathcal{T}\,(concat\ (map\ (map\ f)\ xs))\\
&= \lambda xs.\, \mathcal{T}\,(h\ (map\ (map\ f)\ xs))\\
&= \lambda xs.\, \mathcal{T}\,(\textbf{case}\ map\ (map\ f)\ xs\ \textbf{of}\\
&\qquad\qquad Nil \rightarrow Nil\\
&\qquad\qquad Cons\ x\ xs \rightarrow \textbf{let}\ z = h\ xs\ \textbf{in}\ append\ x\ z)\\
&= \lambda xs.\, \mathcal{T}\,(\textbf{case}\ (\textbf{case}\ xs\ \textbf{of}\\
&\qquad\qquad\qquad Nil \rightarrow Nil\\
&\qquad\qquad\qquad Cons\ x\ xs \rightarrow Cons\ (map\ f\ x)\ (map\ (map\ f)\ xs))\ \textbf{of}\\
&\qquad\qquad Nil \rightarrow Nil\\
&\qquad\qquad Cons\ x\ xs \rightarrow \textbf{let}\ z = h\ xs\ \textbf{in}\ append\ x\ z)\\
&= \lambda xs.\, \mathcal{T}\,(\textbf{case}\ xs\ \textbf{of}\\
&\qquad\qquad Nil \rightarrow \textbf{case}\ Nil\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Nil \rightarrow Nil\\
&\qquad\qquad\qquad\qquad Cons\ x\ xs \rightarrow \textbf{let}\ z = h\ xs\ \textbf{in}\ append\ x\ z\\
&\qquad\qquad Cons\ x\ xs \rightarrow \textbf{case}\ Cons\ (map\ f\ x)\ (map\ (map\ f)\ xs)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Nil \rightarrow Nil\\
&\qquad\qquad\qquad\qquad Cons\ x\ xs \rightarrow \textbf{let}\ z = h\ xs\ \textbf{in}\ append\ x\ z)\\
&= \lambda xs.\, \textbf{case}\ xs\ \textbf{of}\\
&\qquad Nil \rightarrow Nil\\
&\qquad Cons\ x\ xs \rightarrow \mathcal{T}\,(\textbf{case}\ Cons\ (map\ f\ x)\ (map\ (map\ f)\ xs)\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Nil \rightarrow Nil\\
&\qquad\qquad\qquad\qquad Cons\ x\ xs \rightarrow \textbf{let}\ z = h\ xs\ \textbf{in}\ append\ x\ z)\\
&= \lambda xs.\, \textbf{case}\ xs\ \textbf{of}\\
&\qquad Nil \rightarrow Nil\\
&\qquad Cons\ x\ xs \rightarrow \mathcal{T}\,(\textbf{let}\ z = h\ (map\ (map\ f)\ xs)\ \textbf{in}\ append\ (map\ f\ x)\ z)\\
&= \lambda xs.\, \textbf{case}\ xs\ \textbf{of}\\
&\qquad Nil \rightarrow Nil\\
&\qquad Cons\ x\ xs \rightarrow \textbf{let}\ z = \mathcal{T}\,(h\ (map\ (map\ f)\ xs))\ \textbf{in}\ \mathcal{T}\,(append\ (map\ f\ x)\ z)
\end{aligned}
$$

Figure 2.6: Deforestation of $concat \circ map\ (map\ f)$

$$
\begin{array}{l}
= \lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nil} & \rightarrow \mathit{Nil} \\
\mathit{Cons}\ x\ xs & \rightarrow \textbf{let}\ z = \mathcal{T}\,(h\ (\mathit{map}\ (\mathit{map}\ f)\ xs)) \\
 & \quad \textbf{in}\ \mathcal{T}\,(\textbf{case}\ (\mathit{map}\ f\ x)\ \textbf{of} \\
 & \qquad \begin{array}{ll}
 \mathit{Nil} & \rightarrow z \\
 \mathit{Cons}\ x\ xs & \rightarrow \mathit{Cons}\ x\ (\mathit{append}\ xs\ z))
 \end{array}
\end{array} \\
= \lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nil} & \rightarrow \mathit{Nil} \\
\mathit{Cons}\ x\ xs & \rightarrow \textbf{let}\ z = \mathcal{T}\,(h\ (\mathit{map}\ (\mathit{map}\ f)\ xs)) \\
 & \quad \textbf{in}\ \mathcal{T}\,(\textbf{case}\ (\textbf{case}\ xs\ \textbf{of} \\
 & \qquad \begin{array}{ll}
 \mathit{Nil} & \rightarrow \mathit{Nil} \\
 \mathit{Cons}\ x\ xs & \rightarrow \mathit{Cons}\ (f\ x)\ (\mathit{map}\ f\ x))\ \textbf{of}
 \end{array} \\
 & \qquad \begin{array}{ll}
 \mathit{Nil} & \rightarrow z \\
 \mathit{Cons}\ x\ xs & \rightarrow \mathit{Cons}\ x\ (\mathit{append}\ xs\ z))
 \end{array}
\end{array} \\
= \lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nil} & \rightarrow \mathit{Nil} \\
\mathit{Cons}\ x\ xs & \rightarrow \textbf{let}\ z = \mathcal{T}\,(h\ (\mathit{map}\ (\mathit{map}\ f)\ xs)) \\
 & \quad \textbf{in}\ \mathcal{T}\,(\textbf{case}\ xs\ \textbf{of} \\
 & \qquad \begin{array}{ll}
 \mathit{Nil} & \rightarrow z \\
 \mathit{Cons}\ x\ xs & \rightarrow \mathit{Cons}\ (f\ x)\ (\mathit{append}\ (\mathit{map}\ f\ x)\ z))
 \end{array}
\end{array} \\
= \lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nil} & \rightarrow \mathit{Nil} \\
\mathit{Cons}\ x\ xs & \rightarrow \textbf{let}\ z = \mathcal{T}\,(h\ (\mathit{map}\ (\mathit{map}\ f)\ xs)) \\
 & \quad \textbf{in case}\ xs\ \textbf{of} \\
 & \qquad \begin{array}{ll}
 \mathit{Nil} & \rightarrow z \\
 \mathit{Cons}\ x\ xs & \rightarrow \mathit{Cons}\ (f\ x)\ (\mathcal{T}\,(\mathit{append}\ (\mathit{map}\ f\ x)\ z))
 \end{array}
\end{array}
\end{array}
$$

Figure 2.7: Deforestation of $\mathit{concat} \circ \mathit{map}\ (\mathit{map}\ f)$, continued

The proof will be similar in structure to Wadler's proof of termination for first-order deforestation [Wad90b]. It is separated into two parts: firstly, we will show that there is a bound on the nesting of treeless terms in any given input, and secondly that the actual size of these terms is bounded.

Definition 3 An expression is said to be of *order* o (denoted by a superscript o) if it can be formed from an o-fold substitution of treeless terms. A term of order zero is a (non-**letrec**-bound) variable. A term of order $o+1$ is formed from a term u of order o as follows:

$$u[tt_1/x_1, ..., tt_n/x_n]$$

where the x_i are the free variables of u, and the tt_i are arbitrary treeless terms.

Lemma 1 A term of order $o+m$ can be formed from a term u of order o and terms $t_1 \ldots t_n$ each of order m as follows:

$$u[t_1/x_1, ..., t_n/x_n]$$

where the x_i are free variables of u.

Proof By induction on m.

Case 1: given by definition of order.

Case $m+1$: Taking n to be 1,

$$u^o[t^{m+1}/x]$$

we can decompose t according to the definition of order:

$$u^o[(t^m[tt_1/y_1, ..., tt_p/y_p])/x$$

where $y_1 \ldots y_p$ are free variables of t. If we assume that $y_1 \ldots y_p$ do not occur in u (this can be achieved by simply choosing $y_1 \ldots y_p$ to be unique), then the expression can be rewritten:

$$u^o[t^m/x][tt_1/y_1, ..., tt_n/y_p]$$

By the induction hypothesis, this can be rewritten

$$u'^{o+m}[tt_1/y_1, ..., tt_n/y_n]$$

which, by the definition of order, is a term of order $o + m + 1$ as required. This proof extends in a straightforward way to values of $n > 1$. □

Because the syntax of treeless terms includes variables, we have the following relation (using t^o to refer to the set of all terms with order o):

$$t^0 \subset t^1 \subset t^2 \subset ...$$

Therefore it is not possible to assign a unique order to any given term; a term has only a minimum order (the minimum number of substitutions of treeless terms required to obtain the given term). By placing a bound on the minimum order of a term, we have a bound on the nesting of treeless terms within the term.

One small syntactic convenience: an order superscript on a sequence (ts^o, for example) means that all terms in the sequence have the specified order.

Using this definition of order, we can formulate some useful lemmas:

Lemma 2 An applicative expression $f\ t_1 \ldots t_n$ of order $o + 1$ has subterms $t_1 \ldots t_n$ each of order o. An applicative expression $t_1 \ldots t_n$ (where t_1 is not a **letrec**-bound variable) of order $o + 1$ has subterms $t_1 \ldots t_n$ each of order o.

Proof Because treeless form permits only variables in an applicative term, the term $(t_1 \ldots t_n)^{o+1}$ must be formed by one of the substitutions:

$$(x_1 \ldots x_n)[t_1/x_1, \ldots, t_n/x_n]$$
$$(f\ x_1 \ldots x_n)[t_1/x_1, \ldots, t_n/x_n]$$

By Lemma 1, the terms $t_1, \ldots, t_n$ must have order o for the term as a whole to have order $o+1$. □

An equivalent version of the above lemma applies to **case** selectors: we can extract the selector from a **case** term of order $o+1$ as a term of order o.

Lemma 3 Given a term $\lambda x.\, u$ of order $o+1$ and a term t with order o, then the term $u[t/x]$ also has order $o+1$.

Proof We can decompose the term u using Lemma 1 as follows:

$$u^{o+1} = tt[v_1^o/x_1, \ldots, v_n^o/x_n]$$

where the x_i are the free variables of tt. Since the lambda expression $\lambda x.\, u$ must be derived from $\lambda x.\, tt$ by the substitution above, we can deduce that x can only be free in tt. Therefore, the term $u[t/x]$ can be rewritten:

$$u[t/x] = tt[v_1^o/x_1, \ldots, v_n^o/x_n, t^o/x]$$

which is a term of order $o+1$. □

Lemma 4 Given a term of the form (**case** t **of** $\{C_i x_{i1} \ldots x_{ij} \rightarrow v_i\}$) of order o and a term u of order $o-1$ then each expression $v_i[u/x_{in}]$ has order o.

The proof of this lemma is similar to the proof of lemma 3.

Definition 4 A term of order (m, o), where $0 \leq m \leq o$, is of the form

$$\begin{array}{ll} \textbf{case } v^m\ us^{(m-1)} \textbf{ of } \{C_i\ xs_i \rightarrow v_i^{(m+1,o)}\}, & \text{if } 0 \leq m < o \\ t^o, & \text{if } m = o \end{array}$$

In the case where m is zero, the argument list us is empty.

In other words, a term of order (m, o) is a sequence of $o - m$ **case** expressions, ending in a term of order o. The orders in the selector of the cases increase, starting with a selector $v^m\ us^{(m-1)}$ of order m.

Lemma 5 A term of order (m, o) also has order $o + 1$.

Proof By induction on m, with the base case when $m = o$.

When $m = o$, we have a term of order o which also has order $o + 1$.

Given that a term of order $(m + 1, o)$ has order $o + 1$, we will show that a term of order (m, o) also has order $o + 1$:

$$\textbf{case } v^m\ us^{(m-1)} \textbf{ of } \{C_i\ xs_i - > v_i^{(m+1,o)}\}$$

By the induction hypothesis, we have that the v_i have order $o + 1$. In order for the term as a whole to have order $o + 1$ we need to show that the selector $v^m\ us^{(m-1)}$ has order o. Since m is at most $o - 1$, this term has order o by Lemma 2. □

Definition 5 A term of order $\overline{o}$ is a term of order (m, o) for any m such that $0 \leq m \leq o$.

Note that terms of order o are also terms of order $\overline{o}$, and therefore any term with order less than o is also a term of order $\overline{o}$. Also, by Lemma 5, all terms of order $\overline{o}$ have order $o + 1$.

We are now in a position to state the main lemma in the proof of termination:

Lemma 6 Given a mapping of variables to treeless terms D, for any invocation of the operation $\mathcal{T}$ of the form $\mathcal{T}\,(u^{\overline{o}}\ ts^{o-1})$, all recursive invocations will be of the form $\mathcal{T}\,(u'^{\overline{o}}\ ts'^{o-1})$.

Proof By examining each rule of $\mathcal{T}$ in turn, assuming the input is in the form above, we will show that all recursive calls are also of the correct form.

For rules $\mathcal{T}7$–$\mathcal{T}12$ which deal with **case** expressions, we give the case where the input term is of the form $(t^{(m,o)}\ ts^{o-1})$, where $m < o$. The case where $m = o$ (i.e. the term t has order o) is a subset of the more general case given.

Rule $\mathcal{T}1$

$$\mathcal{T}\,(f^1\ ts^{o-1}) = \mathcal{T}\,(u^1\ ts^{o-1}) \text{ where } (f = u) \in D$$

Here, we replace the recursive function variable with its definition, both terms are order 1 so this is safe.

Rule $\mathcal{T}2$

$$\mathcal{T}\,(x\ t_1^{o-1} \ldots t_n^{o-1}) = \textbf{let } z_1 = \mathcal{T}\,t_1^{o-1} \textbf{ in } \ldots \textbf{let } z_n = \mathcal{T}\,t_n^{o-1} \textbf{ in } x\ z_1 \ldots z_n$$

Rule $\mathcal{T}3$

$$\mathcal{T}\,(\lambda x.\, u^o) = \lambda x.\, \mathcal{T}\,u^o$$

Rule $\mathcal{T}4$

$$\mathcal{T}\,((\lambda x.\, u^o)\ t_1^{o-1} \ldots t_n^{o-1}) = \mathcal{T}\,(u[t_1/x]^o\ t_2^{o-1} \ldots t_n^{o-1})$$

The term $u[t_1/x]$ has order o by lemma 3.

Rule $\mathcal{T}5$

$$\mathcal{T}\,(C\ t_1^o \ldots t_n^o) = C\ (\mathcal{T}\,t_1^o) \ldots (\mathcal{T}\,t_n^p)$$

Rule $\mathcal{T}6$

$$\mathcal{T}\,((\textbf{let } x = u^o \textbf{ in } v^o)\ ts^{o-1}) = \textbf{let } x = \mathcal{T}\,u^o \textbf{ in } \mathcal{T}\,(v^o\ ts^{o-1})$$

Rule $\mathcal{T}7$

$$\begin{aligned}&\mathcal{T}\,((\textbf{case } (f^1\ us^{m-1}) \textbf{ of } alts^{(m+1,o)})\ ts^{o-1})\\ &\qquad = \mathcal{T}\,((\textbf{case } (v^1\ us^{m-1}) \textbf{ of } alts^{(m+1,o)})\ ts^{o-1}) \text{ where } (f = v) \in D\end{aligned}$$

Rule $\mathcal{T}8$

$$\begin{aligned}&\mathcal{T}\,((\textbf{case } (x^0\ us^{m-1}) \textbf{ of } \{C_i\ xs_i \to v_i^{(m+1,o)}\})\ ts^{o-1})\\ &\qquad = \textbf{let } z_1 = \mathcal{T}\,u_1^{m-1} \textbf{ in}\\ &\qquad\quad \vdots\\ &\qquad\quad \textbf{let } z_n = \mathcal{T}\,u_1^{m-1} \textbf{ in}\\ &\qquad\quad \textbf{let } z_0 = x\ z_1 \ldots z_n \textbf{ in}\\ &\qquad\quad \textbf{case } z_0 \textbf{ of } \{C_i\ xs_i \to \mathcal{T}\,(v_i^{(m+1,o)}\ ts^{o-1})\}\end{aligned}$$

Rule $\mathcal{T}9$

$$\begin{aligned}&\mathcal{T}\,((\textbf{case } ((\lambda x.\, v)^m\ u_1^{m-1} \ldots u_n^{m-1}) \textbf{ of } alts^{(m+1,o)})\ ts^{o-1})\\ &\qquad = \mathcal{T}\,((\textbf{case } (v[u_1/x]^m\ u_2^{m-1} \ldots u_n^{m-1}) \textbf{ of } alts^{(m+1,o)})\ ts^{o-1})\end{aligned}$$

Again, Lemma 3 gives us that $v[u_1/x]$ has order m.

Rule $\mathcal{T}10$

$$\mathcal{T}\,((\textbf{case } (C\ us^m) \textbf{ of } \{\ldots;\ C\ xs \to v^{(m+1,o)};\ \ldots.\})\ ts^{o-1}) = \mathcal{T}\,(v[us/xs]^{(m+1,o)}\ ts^{o-1})$$

The term $v[us/xs]$ has order $(m+1, o)$ by Lemma 4.

Rule $\mathcal{T}11$

$$\mathcal{T}\,((\textbf{case}\ (\textbf{case}\ v^{m-1}\ \textbf{of}\ \{C_i\ xs_i \rightarrow v_i^m\})\ us^{m-1}\ \textbf{of}\ alts^{(m+1,o)})\ ts^{o-1})$$
$$= \mathcal{T}\,((\textbf{case}\ v^{m-1}\ \textbf{of}\ \{C_i\ xs_i \rightarrow \textbf{case}\ v_i^m\ us^{m-1}\ \textbf{of}\ alts^{(m+1,o)}\})\ ts^{o-1})$$

The recursive call on the right here is of the form $\mathcal{T}\,(u^{(m-1,o)}\ ts^{o-1})$, which is in the required form. When $m = o$ (i.e. the input term has order o) a special case of the above rule applies: for the input term to have order o, us must be empty.

This rule lengthens the string of **case** terms in the input. It cannot be applied indefinitely, because the length of the **case** sequence is limited by o.

Rule $\mathcal{T}12$

$$\mathcal{T}\,((\textbf{case}\ ((\textbf{let}\ x = u^m\ \textbf{in}\ v^m)\ us^{m-1})\ \textbf{of}\ alts^{(m+1,o)})\ ts^{o-1})$$
$$= \textbf{let}\ x = \mathcal{T}\,u^m\ \textbf{in}\ \mathcal{T}\,((\textbf{case}\ (v^m\ us^{m-1})\ \textbf{of}\ alts^{(m+1,o)})\ ts^{o-1})$$

□

Having shown that there is a bound on the nesting of treeless terms in the input to any invocation of $\mathcal{T}$, we can use this lemma to show that the size of these terms is bounded.

We first define a more stringent measure of the size of a term:

Definition 6 The *depth* of an expression is zero for a variable, and one plus the maximum depth of its subexpressions otherwise.

Note that application is not a binary operator, so the applicative expression $t_1 \ldots t_n$ has depth one plus the maximum depth of the t_i. Also, applicative expressions are implicitly flattened so t_1 cannot itself be an applicative term.

Lemma 7 Given a term t with order o, where the treeless terms used to construct t have a maximum depth d, the depth of t is no more than $o \times d$.

Proof By induction on o. For the zero case, t is a variable and therefore has depth zero as required. For $o + 1$, t is of the form

$$t'^{(o+1)} = t^o[tt_1/x_1, ..., tt_n/x_n]$$

By the inductive hypothesis, we have that the maximum depth to any variable in t^o is $o \times d$. The substitution will replace this variable with a term of depth at most d, so the depth of the new expression is $(o \times d) + d$, or $(o + 1) \times d$. □

Lemma 8 For a given call to $\mathcal{T}$ of the form $\mathcal{T}\, t^o$, there exists a bound on the depth of the argument to all recursive calls of $\mathcal{T}$ generated by this initial call.

Proof First, we define a constant depth d to be the maximum depth of any treeless term occurring in the input to the transformation. This includes all recursive definitions in D, and the treeless terms used to construct the initial term.

We know from lemma 6 that all recursive calls to $\mathcal{T}$ are of the form $\mathcal{T}\,(u^{\overline{o}}\ ts^{o-1})$, so we now need to show that the terms in calls of this form have a bounded depth. The maximum depth of any term in ts is $(o - 1) \times d$, since they all have order $o - 1$. The maximum depth of u is $o + (o \times d)$ (since a term of order $\overline{o}$ is a nested sequence of at most o **case** expressions, ending in a term of order o).

Now, assuming that the expression passed to an invocation of $\mathcal{T}$ satisfies the above depth properties, we can show that all recursive calls to $\mathcal{T}$ also satisfy these properties. The proof is by rule-by-rule inspection of $\mathcal{T}$:

- Rules T2, T3, T5, T6, T8, and T12 simply invoke $\mathcal{T}$ on subterms of the input, so depth properties are preserved.
- Rules T1 and T7 replace a call to a recursive function by its definition. The function call has order 1, and its definition has maximum depth d.
- Rules T4, T9, and T10 apply substitutions to make new terms. By lemma 7, using substitution to make a new term in this way yields a new term of depth $o \times d$.

- Rule T11 flattens out a case expression, possibly increasing the depth of the argument to the recursive call of $\mathcal{T}$. However, it generates a term of order $(m-1, o)$ from a term of order (m, o). The depth of the original term is therefore $(o-m)+(o \times d)$, and the new term has depth $(o-(m-1))+(o \times d)$, which is still within our bound above because m has a lower bound of 1.

So, given a term of the form $(u^{\overline{o}}\ ts^{o-1})$, the operation $\mathcal{T}$ can never be invoked on a term of depth greater than

$$1+o+(o \times d)$$

Since the initial call to $\mathcal{T}$ is on a term of order o (which is also in the form $(u^{\overline{o}}\ ts^{o-1})$), then there exists a bound on the depth of terms which can occur in recursive calls to $\mathcal{T}$. □

We have shown that there is a bound on the depth of terms that occur in the input to $\mathcal{T}$. We still do not have a proof of termination, however: we have not established a bound on the width of applicative terms (i.e. what is the limit on n in $t_1 \ldots t_n$?). One example that illustrates why this is a problem is:

$$f \text{ where } f = \lambda x.f\ x\ x$$

If we apply $\mathcal{T}$ to f, it will encounter ever increasing terms on the input (even though the terms have bounded depth!). For this reason, we have placed a restriction on the recursive types that can occur in the input terms and functions. Recursive types must be defined using algebraic data types, and therefore any recursive type must pass through a constructor argument (the above function f does not satisfy this restriction).

Lemma 9 There exists a bound on the width of applicative terms that can occur during transformation.

Proof. We first make the assumption that our transformation algorithm preserves the well-typedness of the original program (this can be verified by simple examination of the

algorithm). Given this, infinitely-wide applicative expressions cannot occur because they would be ill-typed. Considering the term at the head of the application, there are two possibilities:

- The term cannot have a recursive type, since recursive types are defined using algebraic data types and the value of any expression with a recursive type is a constructor application; the applicative term would therefore be ill-typed.
- The term cannot have an infinitely large type of the form $T_1 \rightarrow T_2 \rightarrow \ldots$, since infinitely large types cannot occur in a well-typed program. Even if we consider polymorphism, in a well-typed program each type variable in a polymorphic type can only be instantiated to a finite type.

□

Theorem 1 *The Deforestation Theorem.* When the transformation $\mathcal{T}$ and the knot-tying algorithm are applied to a term t with an order o and a set of treeless definitions D, the process terminates and yields a treeless term together with a new set of treeless functions.

Proof Follows from lemma 8 and 9. To summarise, the structure of the proof is as follows:

- Establish a measure of treeless term nesting, *order*.
- Extend order to cover sequences of **case**-terms that occur during transformation.
- Show that given an input term of a certain order, all recursive calls to the transformation function will be on expressions of the same order.
- Relate order to the actual depth of terms.
- Show that the more stringent depth property is preserved; that is, there is a bound on the depth of any term occurring as an argument to the transformation function.
- Show how the restriction on the well-typedness of the input term guarantees that infinitely-wide applicative terms cannot occur during transformation. This completes the proof.

□

One implication of this termination proof is that the size of the resulting code is related to the number of permutations of the label expressions, and is therefore not linear in the size of the input expressions. The size of the result of deforestation, although bounded, can be far larger than the size of its inputs.

2.7 Summary

In this chapter we have described a new algorithm for performing deforestation of higher-order functional programs. The key insight in formulating a deforestation algorithm for a higher order language was designing treeless form in such a way that the algorithm is guaranteed to terminate. In our proof of termination, we identified the invariant measure on the size of terms that occur during transformation, and used this to show that the size of terms is bounded.

The approach described in this chapter here has a number of shortcomings and areas which remain to be resolved. These issues will be summarised here and addressed in the rest of the thesis.

2.7.1 Transparency and Treeless Form

The deforestation algorithm presented in this chapter is not entirely transparent. It removes *most* of the intermediate data structures that are not denoted by **let** terms in the input, but some cannot be removed and are therefore placed in **let** bindings by the transformation (see rules $\mathcal{T}2$ and $\mathcal{T}8$).

The extra **let**s are required to keep the output in treeless form, but it is not immediately clear why we should be adding more residual data structures to the output. In fact, this is a consequence of our formulation of treeless form, which doesn't correspond directly to the normal form of the language. It differs in the terms allowed to the right of an application (normal form would allow arbitrary terms to the right of an application whose head is a variable), and the terms allowed in the selector of a **case** (normal form would allow

a variable applied to a number of terms). If treeless form were generalised in this way, then we would have no need for the additional **lets** introduced during the transformation. Additionally, our treeless form would correspond to the normal form of the language, meaning that evaluation of a term in treeless form would still use no intermediate storage, which is the key property of treeless form. Unfortunately, making this generalisation would also render our proof of termination invalid, since Lemma 2 would not be true.

In the next chapter, we will derive a new deforestation algorithm with a generalised treeless form by using properties of the logics of natural deduction and sequent calculus.

2.7.2 Linearity

So far in this chapter we have not mentioned the efficiency of a program generated by deforestation. Ideally, we would like to guarantee that the program is no less efficient than the original, and when intermediate structure is removed, that the program will be more efficient (given a suitable operational semantics). Unfortunately, in the world of lazy evaluation this property is hard to guarantee. This topic will be discussed fully in Section 4.2.

2.7.3 Generalising the algorithm for real programming languages

Some other issues remain to be solved before the deforestation algorithm described in this chapter can be applied to programs written in a practical programming language, such as Haskell. The first problem is that of how to convert user-written functions into treeless form. There is an optimal translation from general terms to treeless terms (optimal in the sense that the translated function keeps the largest subset of eliminable structure in the original as possible), which is described in Section 4.1.

The remainder of the issues of applying deforestation to real-world programming languages are dealt with in Chapter 5, where we describe our own prototype implementation in the Glasgow Haskell Compiler.

Chapter 3

Cut Elimination

3.1 Introduction

In this chapter we borrow a fundamental principle from logic and use it to derive a higher-order deforestation algorithm. The particular principle we are interested in, cut elimination, is concerned with the simplification of logical proofs. The Curry-Howard isomorphism links proofs in the logic of natural deduction to terms in the simply typed lambda calculus, where simplification of proofs by cut elimination corresponds to normalisation of lambda calculus terms—exactly the goal of deforestation, for normalising a term is equivalent to eliminating its intermediate data structures.

Cut elimination was invented by Gentzen in the 1930s [Gen35] when he wrote down the logics of natural deduction and sequent calculus. He showed that any proposition in sequent calculus can be restructured in such a way that the resulting proof contains no uses of the cut rule. Furthermore, this can be done automatically and on any proof containing cuts. Gentzen, however, had no similar theorem for natural deduction. The strong normalisation property of natural deduction proofs was discovered by Prawitz [Pra65] some 30 years later.

The Curry-Howard isomorphism firmly bonds the mathematics of logic to the science of programming languages. Curry [CF58] noticed the relationship between natural deduction and combinatory logic, and Howard [How80] showed that there is an equivalent one-to-one correspondence between natural deduction and the simply typed lambda calculus.

Propositions in the logic can be viewed as types, and proofs as terms in the lambda calculus. In addition, simplifying a proof corresponds to reduction of terms. In the same way that lambda calculus terms are strongly normalising, natural deduction proofs have a simplest form.

There exists another isomorphism analogous to Curry-Howard, which relates the logic of sequent calculus to its equivalent programming language. The programming language of sequent calculus appears similar to the lambda calculus, with some important differences. The most important difference is that sequent calculus contains a cut rule. Another difference is the syntax for application, which we will discuss in Section 3.4.

The term form corresponding to the cut rule in the logic is the **cut** construct.

$$Cut\ \frac{\Gamma \vdash t : A \quad \Gamma, x : A \vdash u : B}{\Gamma \vdash \mathbf{cut}\ x = t\ \mathbf{in}\ u : B}$$

Without cut, it is only possible to write proofs (and hence terms) in normal form. Contrast this to natural deduction, where it is possible to write arbitrary terms using only the introduction and elimination rules for the various logical constructs. The theorem of cut elimination states that any sequent calculus proof using cut has an equivalent proof with no cuts, i.e. any program in the sequent calculus language written using **cut** can be transformed into one with no **cuts**, which is in normal form.

We distinguish **cut** and **let** for an important reason: by convention, we use **let** to represent *residual* data structures, those that will not be removed during transformation, and use **cut** to represent *eliminable* data structures, those that will.

Cut elimination is similar to normalisation in natural deduction, a process which we view as the essence of computation. Reducing a term in the lambda calculus is a step-by-step process of removing the intermediate structures yielding a term in normal form. Not only is this the essence of computation, it is also the essence of *deforestation.*

Cut elimination as a program transformation scheme possesses an important property, one that is missing in the normalisation of terms in the lambda calculus. This property is transparency—in a sequent calculus term, each intermediate structure is marked by a **cut**, all of which are removed by cut elimination. To base a deforestation algorithm on cut

elimination would mean that the algorithm is not only based on a sound logical principle, but is also truly transparent.

There are some important problems to be surmounted before this is possible, though. Firstly, we must apply cut elimination to a language that is similar enough to the lambda calculus such that we can apply it to modern programming languages based on that system. Secondly, and this is certainly the largest mountain to climb, deforestation is only useful when applied to recursive programs, while cut elimination applies only to non-recursive terms.

The approach we take to the first problem is to define a hybrid language, which draws on features from both the language of natural deduction and the language of sequent calculus. The language describes the subset of terms in the lambda calculus that are in normal form. By adding the **cut** term to this language we regain the full power of the simply typed lambda calculus, with the benefit that the **cut** terms can be removed by our cut elimination algorithm.

For the second problem, we take simple first-order recursion for which we already have a deforestation algorithm that is known to terminate. We reformulate first-order deforestation in a form that is similar to the formulation of cut elimination. Then, taking the non-recursive cut elimination algorithm and first-order deforestation we amalgamate the two, yielding a fully transparent higher-order deforestation algorithm. It should be said that the termination properties of the resulting algorithm are not fully known, but we do give a conjecture as to the conditions under which the algorithm terminates and some convincing arguments as to why these should be sufficient.

The remainder of this chapter is organised as follows. The next two Sections, 3.2 and 3.3, give a brief overview of natural deduction and sequent calculus. Section 3.4 presents our cut elimination algorithm for the hybrid sequent calculus/natural deduction style language, and gives a proof of termination. Section 3.5 discusses two ways in which explicit recursion may be added to the lambda calculus, and explains why we decided to use first-order recursion equations to represent the recursion in our language. Section 3.6 describes the first-order deforestation algorithm. Section 3.7 describes the merged algorithm, and goes on to give a version in which the loop creation is explicit. Then our arguments for termination are given, with some examples.

$$\text{Id}\ \frac{}{\Gamma,\, x : A \vdash x : A}$$

$$\to\text{I}\ \frac{\Gamma,\, x : A \vdash u : B}{\Gamma \vdash \lambda x.\, u : A \to B} \qquad \to\text{E}\ \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u}$$

$$\text{C-I}\ \frac{\Gamma \vdash ts : As_j}{\Gamma \vdash C_j\ ts : \{C_1\ As_1; \ldots; C_n\ As_n\}}\ \text{some } j \in [1..n]$$

$$\text{C-E}\ \frac{\Gamma \vdash t : \{C_1\ As_1; \ldots; C_n\ As_n\} \quad \Gamma,\, xs_1 : As_1 \vdash v_1 : B \quad \ldots \quad \Gamma,\, xs_n : As_n \vdash v_n : B}{\Gamma \vdash \textbf{case}\ t\ \textbf{of}\ \{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\} : B}$$

$$\text{Multi}\ \frac{\Gamma \vdash t_1 : A_1 \quad \ldots \quad \Gamma \vdash t_n : A_n}{\Gamma \vdash t_1 \ldots t_n : A_1 \ldots A_n}$$

Figure 3.1: Natural Deduction

3.2 Natural Deduction

Figure 3.1 shows the logic of natural deduction with its associated programming language, the simply typed lambda calculus. Pure natural deduction can be obtained by simply removing all the terms in the typing rules; i.e. everything to the left of (and including) the ':'.

The syntax of types is given by:

$$\begin{array}{lcl} A, B & ::= & X \mid A \to B \mid \{C_1\ As_1; \ldots; C_n\ As_n\} \\ As & ::= & A_1 \ldots A_n \end{array}$$

where A, B range over types, X, Y, Z range over type variables, C ranges over constructors, and As ranges over sequences of types. Terms are given by:

$$
\begin{array}{lcl}
t, u, v & ::= & x \mid \lambda x.\, t \mid t\ u \mid C\ ts \mid \textbf{case}\ t\ \textbf{of}\ alts \\
alts & ::= & \{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\} \\
ts & ::= & t_1 \ldots t_n \\
xs & ::= & x_1 \ldots x_n
\end{array}
$$

where t, u, v range over terms, x, y, z range over variables, *alts* ranges over lists of case alternatives, *ts* ranges over sequences of terms, and *xs* ranges over sequences of variables.

We use a single sum-of-products type modelled after the system of constructed datatypes in languages such as Haskell. The type $\{C_1\ As_1; \ldots; C_n\ As_n\}$ is the datatype containing constructors C_i each with a type argument list As_i, for all i from 1 to n. An object of this type is built with the expression $C_j\ ts$ where $1 \leq j \leq n$, and ts is a sequence of terms with the same length as As_j and where each term in the sequence ts has the corresponding type in As_j. Objects of this type are deconstructed using the **case** term form. In the following chapter we will use the notation $\{C_i\ As_i\}$ as an abbreviation for $\{C_1\ As_1; \ldots; C_n\ As_n\}$, and $\{C_i\ xs_i \to v_i\}$ as an abbreviation for $\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\}$.

Apart from the Multi rule, which is merely a syntactic convenience, all the rules in the system correspond to the introduction or elimination of one logical construct. Our presentation of natural deduction does not include explicit weakening and contraction rules (also called the *structural* rules), rather we incorporate weakening into the Id rule and allow assumptions to be freely duplicated.

Simplification of natural deduction proofs is equivalent to reduction in the lambda calculus. The two basic reduction rules are:

$$
\begin{array}{ll}
(\lambda x.\, u)\ t & \to u[t/x] \\
\textbf{case}\ C_j\ ts\ \textbf{of}\ \{C_i\ xs_i \to v_i\} & \to v_j[ts/xs_j]
\end{array}
$$

The expressions on the left of the reduction rules correspond to proofs in which an introduction rule appears next to an elimination rule. Such proofs can always be simplified by applying the above reduction rules. However, the introduction rule does not always appear next to the elimination rule, requiring that we also apply *commuting conversions* to the

proof to fully simplify it. The additional rules involved are:

$$\begin{array}{ll} (\textbf{case}\ t\ \textbf{of}\ \{C_i\ xs_i \to v_i\})\ t' & \to \textbf{case}\ t\ \textbf{of}\ \{C_i\ xs_i \to v_i\ t'\} \\ \textbf{case}\ (\textbf{case}\ t\ \textbf{of}\ \{C_i\ xs_i \to v_i\})\ \textbf{of}\ alts & \to \textbf{case}\ t\ \textbf{of}\ \{C_i\ xs_i \to \textbf{case}\ v_i\ \textbf{of}\ alts\} \end{array}$$

Note we are assuming that variables are named such that capture does not occur during any of these rewrite rules.

The four rewrite rules given above are sufficient to reduce to normal form any expression in the language. Hence, the equivalent transformations on proof trees reduce any proof to normal form.

3.3 Sequent Calculus

In Figure 3.2 we give the logic of sequent calculus as a programming language. The grammar of types is identical to that given above, and the grammar of terms is given by:

$$\begin{array}{lcl} t, u, v & ::= & z \mid \lambda x.\, t \mid \textbf{apply}\ z\ \textbf{to}\ t\ \textbf{as}\ y\ \textbf{in}\ v \\ & & \mid C\ ts \mid \textbf{case}\ z\ \textbf{of}\ alts \\ & & \mid \textbf{cut}\ z = t\ \textbf{in}\ u \\ alts & ::= & \{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\} \\ ts & ::= & t_1 \ldots t_n \\ xs & ::= & x_1 \ldots x_n \end{array}$$

In the sequent calculus language, application takes the form:

$$\textbf{apply}\ z\ \textbf{to}\ t\ \textbf{as}\ y\ \textbf{in}\ v$$

The above term is similar in meaning to the natural deduction term $(\lambda y.\, v)\ (z\ \ t)$. The difference is that the sequent calculus term is always strict in z, whereas the natural deduction term may not be (for example, if y does not appear in v). Put more precisely, if the value of z in the sequent calculus form of application is $\bot$, then the value of the

$$Id\ \frac{}{\Gamma,\ x : A \vdash x : A}$$

$$\to\mathrm{R}\ \frac{\Gamma,\ x : A \vdash u : B}{\Gamma \vdash \lambda x.\, u : A \to B} \qquad \to\mathrm{L}\ \frac{\Gamma \vdash t : A \quad \Gamma,\ y : B \vdash v : B'}{\Gamma,\ z : A \to B \vdash \mathbf{apply}\ z\ \mathbf{to}\ t\ \mathbf{as}\ y\ \mathbf{in}\ v : B'}$$

$$\text{C-R}\ \frac{\Gamma \vdash ts : As_j}{\Gamma \vdash C_j\ ts : \{C_1\ As_1; \ldots; C_n\ As_n\}}\ \text{some } j \in [1..n]$$

$$\text{C-L}\ \frac{\Gamma,\ xs_1 : As_1 \vdash v_1 : B \quad \ldots \quad \Gamma,\ xs_n : As_n \vdash v_n : B}{\Gamma,\ z : \{C_1\ As_1; \ldots; C_n\ As_n\} \vdash \mathbf{case}\ z\ \mathbf{of}\ \{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\} : B}$$

$$Cut\ \frac{\Gamma \vdash t : A \quad \Gamma,\ x : A \vdash u : B}{\Gamma \vdash \mathbf{cut}\ x = t\ \mathbf{in}\ u : B}$$

$$\text{Multi}\ \frac{\Gamma \vdash t_1 : A_1 \quad \ldots \quad \Gamma \vdash t_n : A_n}{\Gamma \vdash t_1 \ldots t_n : A_1 \ldots A_n}$$

Figure 3.2: Sequent Calculus

term as a whole is always $\bot$. Of course, this is only true if the semantics of our language includes $\bot$ (which ours does).

The grammar for sequent calculus describes terms in normal form if the **cut** term form is removed. This is an attractive representation of terms for our purposes, since it gives complete transparency to the normalisation process: normalising a term in the language of sequent calculus takes a term involving **cuts** and yields a term containing no **cuts**. Equivalently, a proof involving cuts becomes a proof with no cuts. Full explanations of cut elimination can be found in Girard, Lafont and Taylor's book [GLT89] and Gallier's tutorial [Gal93].

To a logician, the important property of sequent calculus is that in all the rules in the logic except cut, the formulas appearing above the line are proper subformulas of those below it. While natural deduction is more intuitive as a proof methodology, sequent calculus is more interesting from a proof theoretic point of view. From our point of view, however,

we wish to have the best of both worlds: the transparency of cut elimination applied to the familiar language of the lambda calculus.

3.4 Cut Elimination

In this section we present our hybrid language. We then give an algorithm for cut elimination, and prove termination for it. Our eventual goal, an algorithm to remove intermediate data structures from programs written in a lambda-calculus based programming language, leads us to formulate cut elimination in a way that has some important differences from that of Gentzen.

As noted above, the language that corresponds to sequent calculus has a different syntax for application than the lambda calculus. An applicative expression in a normal form lambda calculus term takes the form $z\ t_1 \ldots t_n$, a variable applied to any number of other terms in normal form.

There is a simple translation between the lambda calculus form of application and the sequent calculus form. The term equivalent to $z\ t_1 \ldots t_n$ looks like this:

$$\begin{aligned}
&\textbf{apply } z \textbf{ to } t_1 \textbf{ as } z_1 \textbf{ in} \\
&\textbf{apply } z_1 \textbf{ to } t_2 \textbf{ as } z_2 \textbf{ in} \\
&\ldots \\
&\textbf{apply } z_{n-1} \textbf{ to } t_n \textbf{ as } z_n \textbf{ in } z_n
\end{aligned}$$

However, there is no translation in the other direction in general. The closest we can get is to translate **apply** z **to** t **as** y **in** v as $(\lambda y.\, v)\ (z\ t)$, but the first is strict in z while the second is not.

The strictness property means that the sequent calculus form of application is in some sense more expressive than the natural deduction form. This extra flexibility is however superfluous to our needs, and in fact this turns out to be important when we add recursion to the language.

The second difference between the cut elimination of Gentzen and that given here is that we present the algorithm as the transformation of terms in the programming language rather

t, u, v	::=	$\lambda x.\, t$	lambda abstraction
	|	$C\ ts$	constructor application
	|	$z\ ts$	application
	|	**case** $z\ ts$ **of** $alts$	case expression
ts	::=	$t_1 \ldots t_n$	sequence of terms
xs	::=	$x_1 \ldots x_n$	sequence of variables
$alts$	::=	$\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\}$	case alternatives

Figure 3.3: The Syntax of Terms

than the manipulation of proofs in the logic. Since each rule in the logic corresponds to a term form in the programming language the two styles are necessarily dual. However, we chose the program transformation style because it more clearly shows the relationship to deforestation.

The third and final difference is that our cut elimination algorithm removes an arbitrary number of cuts simultaneously; it is a *multi*-cut elimination algorithm. This extension occurs naturally as a consequence of using the **case** construct to examine data structures: the **case** reduction step gives rise to several cuts, which may all be removed simultaneously. Of course, being able to remove multiple cuts in a single pass also leads to a more efficient implementation. There is one restriction, however: the cuts cannot be interdependent. That is, the intermediate values being removed cannot depend on each other. This is not a serious restriction, since we can always order a set of cuts according to dependence and recursively remove the inner ones first.

In our cut elimination algorithm we do not make use of explicit **cut** terms; the algorithm begins with a term in normal form and a list of variable bindings in the form of an environment, and yields a new term in normal form.

3.4.1 The Hybrid Language

The grammar of types is as in Section 3.2.

$$\rightarrow\text{I}\ \frac{\Gamma,\ x : A \vdash u : B}{\Gamma \vdash \lambda x.\, u : A \rightarrow B} \qquad \rightarrow\text{E}\ \frac{\Gamma \vdash ts : As}{\Gamma,\ z : As \rightarrow B \vdash z\ ts : B}$$

$$\text{C-I}\ \frac{\Gamma \vdash ts : As_j}{\Gamma \vdash C_j\ ts : \{C_1\ As_1; \ldots; C_n\ As_n\}}\ \text{some } j \in [1..n]$$

$$\text{C-E}\ \frac{\begin{array}{c}\Gamma \vdash z\ ts : \{C_1\ As_1; \ldots; C_n\ As_n\} \\ \Gamma,\ xs_1 : As_1 \vdash v_1 : B \quad \ldots \quad \Gamma,\ xs_n : As_n \vdash v_n : B\end{array}}{\Gamma \vdash \textbf{case}\ z\ ts\ \textbf{of}\ \{C_1\ xs_1 \rightarrow v_1; \ldots; C_n\ xs_n \rightarrow v_n\} : B}$$

$$\text{Multi}\ \frac{\Gamma \vdash t_1 : A_1 \quad \ldots \quad \Gamma \vdash t_n : A_n}{\Gamma \vdash t_1 \ldots t_n : A_1 \ldots A_n}$$

Figure 3.4: Type System

In our treatment of types we use a syntactic abbreviation for long function types. If As is a sequence of types $A_1 \ldots A_n$ then $As \rightarrow B$ stands for $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$.

The language around which the cut elimination algorithm is based (Figure 3.3) is a lambda calculus based language in which only terms in normal form can be represented.

We assume a straightforward non-strict semantics for the language, given by an evaluator, $\mathcal{E}$. The evaluator takes a term t and a value environment η and returns the value of t. The value environment must be complete; that is, it must map all the free variables of t.

The type system for the language is given in Figure 3.4. There is one rule associated with each term form, as we would expect. By the Curry-Howard isomorphism, removing the terms (everything to the left of a ':') from the type system gives us a natural deduction logic, albeit only for proofs in normal form.

3.4.2 Algorithm

The algorithm for multi-cut elimination is shown in Figure 3.5. It consists of three mutually recursive operations, $\mathcal{GT}$, $\mathcal{GA}$, and $\mathcal{GC}$. Both $\mathcal{GT}$ and $\mathcal{GC}$ require an environment ρ, which

$$
\begin{array}{lll}
\mathcal{GT}\ \rho\ (\lambda x.\, u) & = \lambda x.\, \mathcal{GT}\ \rho\ u & (1)\\
\mathcal{GT}\ \rho\ (C\ ts) & = C\ (\mathcal{GTs}\ \rho\ ts) & (2)\\
\mathcal{GT}\ \rho\ (z\ ts) & = \mathcal{GA}\ \rho(z)\ (\mathcal{GTs}\ \rho\ ts) & (3)\\
\mathcal{GT}\ \rho\ (\textbf{case}\ z\ ts\ \textbf{of}\ alts) & = \mathcal{GC}\ \rho\ (\mathcal{GA}\ \rho(z)\ (\mathcal{GTs}\ \rho\ ts))\ alts & (4)\\
 & & \\
\mathcal{GTs}\ \rho\ (t_1 \ldots t_n) & = (\mathcal{GT}\ \rho\ t_1) \ldots (\mathcal{GT}\ \rho\ t_n) & (5)\\
 & & \\
\mathcal{GA}\ t\ [] & = t & (6)\\
\mathcal{GA}\ (\lambda x.\, u)\ (t : ts) & = \mathcal{GA}\ (\mathcal{GT}\ [t/x]\ u)\ ts & (7)\\
\mathcal{GA}\ (z\ ts)\ ts' & = z\ (ts +\!\!+ ts') & (8)\\
\mathcal{GA}\ (\textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ xs_i \to v_i\})\ ts' & = \textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ ys_i \to \mathcal{GA}\ v_i\ ts'\} & (9)\\
 & & \\
\mathcal{GC}\ \rho\ (C\ ts)\ \{\ldots; C\ xs \to v; \ldots\} & = \mathcal{GT}\ \rho[ts/xs]\ v & (10)\\
\mathcal{GC}\ \rho\ (z\ ts)\ \{C_i\ xs_i \to v_i\} & = \textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ ys_i \to \mathcal{GT}\ \rho\ v_i\} & (11)\\
\mathcal{GC}\ \rho\ (\textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ xs_i \to v_i\})\ alts & = \textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ ys_i \to \mathcal{GC}\ \rho\ v_i\ alts\} & (12)
\end{array}
$$

Figure 3.5: Multi Cut Elimination

is an idempotent mapping from variables to terms. Idempotence in this context means that none of the terms in ρ refer to variables in the domain of ρ. The assumption that the environment is idempotent is pervasive in the cut-elimination algorithm; we never attempt to substitute within terms that originate in the environment.

We use a substitution syntax for environments. An environment is a list of the form $[t_1/x_1, \ldots, t_n/x_n]$ where each of the elements t_i/x_i represents a mapping from x_i to t_i. Write $\rho(x)$ for the term associated with x in the environment ρ. The notation $\rho[t/x]$ refers to an environment that maps x to t and maps y (where $y \neq x$) to $\rho(y)$.

We also use some abbreviations related to environments for compactness in the representation of the algorithm. If $xs = x_1 \ldots x_n$, $ys = y_1 \ldots y_n$, and $ts = t_1 \ldots t_n$, then the notation $[ts/xs]$ is short for $[t_1/x_1, \ldots, t_n/x_n]$, and the notation $[\rho(xs)/ys]$ is short

for $[\rho(x_1)/y_1, \ldots, \rho(x_n)/y_n]$.

The meanings of each of the three operations can be expressed in terms of the evaluator, $\mathcal{E}$:

$$\begin{aligned}
\mathcal{E}\ (\mathcal{GT}\ \rho\ t)\ \eta &= \mathcal{E}\ t\ (\eta + \rho) \\
\mathcal{E}\ (\mathcal{GA}\ t\ ts)\ \eta &= \mathcal{E}\ (z\ ts)\ (\eta[(\mathcal{E}\ t\ \eta)/z] \\
\mathcal{E}\ (\mathcal{GC}\ \rho\ t\ alts)\ \eta &= \mathcal{E}\ (\textbf{case}\ z\ \textbf{of}\ alts)\ (\eta[(\mathcal{E}\ t\ \eta)/z] + \rho)
\end{aligned}$$

where the notation $\eta + \rho$ is shorthand for $[v_1, \ldots, v_n, (\mathcal{E}\ t_1\ \eta)/z_1, \ldots, (\mathcal{E}\ t_n\ \eta)/z_n]$ if $\eta = [v_1, \ldots v_n]$ and $\rho = [t_1/z_1, \ldots, t_n/z_n]$.

As an aside, it is possible to merge the functionality of $\mathcal{GA}$ into $\mathcal{GC}$, enabling the call to $\mathcal{GA}$ in rule 4 to be omitted. This takes the recursive calls in rule 4 from being triple-nested to being double-nested in exchange for some additional complexity in $\mathcal{GC}$. However, the nesting turns out not to be a factor in the proof of termination for this rule, so we opted for the clearer form.

3.4.3 Proof of termination

Lemma 10 Given a typed term t and an environment ρ binding free variables of t to typed terms, the call $\mathcal{GT}\ \rho\ t$ terminates.

Proof. To prove termination of the cut elimination algorithm, we make use of the multiset ordering. A multiset is a finite set of tokens. The tokens support a well-founded partial ordering $<$, and a equivalence $=$. A multiset can contain multiple copies of the same token, so it is not a set in the strict sense of the word. The partial ordering on tokens is extended to multisets as follows:

$$\text{if}\ \ y_1 < x, \ldots, y_m < x\ \ \text{then}\ \ \{y_1, \ldots, y_m, x_1, \ldots, x_n\} < \{x, x_1, \ldots, x_n\}$$

Informally, one multiset is smaller than another if it can be derived from the first by repeatedly removing a token and replacing it with any number of smaller tokens. It is

this property of multisets that is the key to proving termination of our algorithm: in two cases (namely, beta reduction and **case** reduction) the transformation exchanges one large object for a number of smaller objects. By the ordering we have defined, this amounts to a decrease in the size of the multiset.

The size of a term, written $\mathrm{siz}(t)$, is defined as one for a variable and one plus the sum of the sizes of its subterms otherwise. The size of a type, written $\mathrm{siz}(A)$ is defined as one for a type variables and one plus the sum of the sizes of its subtypes otherwise. The operation $\mathrm{occ}(x, t)$ gives the number of occurrences of the variable x in the term t. The type of a term t is given by $\mathrm{typ}(t)$. The set of variables in the domain of an environment ρ is written $\mathrm{dom}(\rho)$.

We now define a multiset index for each of the operations $\mathcal{GT}$, $\mathcal{GC}$ and $\mathcal{GA}$. The index for an operation is pair of multisets; the first multiset contains the types of some of the expressions in the call, and the second multiset contains the sizes of some of the expressions in the call. The partial order on types is 'subtype of' and on sizes is 'less than'. The notation $n \times t$ is shorthand for n copies of t.

$$
\begin{aligned}
\mathrm{index}(\mathcal{GT}\ \rho\ t) &= (\{\mathrm{occ}(x,t) \times \mathrm{typ}(\rho(x)) \mid x \in \mathrm{dom}(\rho)\}, \\
&\quad \{\mathrm{occ}(x,t) \times \mathrm{siz}(\rho(x)) \mid x \in \mathrm{dom}(\rho)\} \cup \{\mathrm{siz}(t)\}) \\
\mathrm{index}(\mathcal{GC}\ \rho\ t\ \{C_i\ xs_i \to v_i\}) &= (\{\mathrm{occ}(x,v) \times \mathrm{typ}(\rho(x)) \mid x \in \mathrm{dom}(\rho)\} \cup \{\mathrm{typ}(t)\}, \\
&\quad \{\mathrm{occ}(x,t) \times \mathrm{siz}(\rho(x)) \mid x \in \mathrm{dom}(\rho)\} \cup \{\mathrm{siz}(t)\} \\
&\quad \cup \bigcup(\mathrm{siz}(v_i))) \\
\mathrm{index}(\mathcal{GA}\ t\ ts) &= (\{\mathrm{typ}(t)\}, \{\mathrm{siz}(t)\})
\end{aligned}
$$

Indices are ordered lexicographically: one index is smaller than another if either the type index (the first component) is smaller, or the type index remains the same while the size index (the second component) is smaller. Note that provided the type index gets smaller, the size index may be larger and the ordering still holds.

Our goal in this proof is to show that, for each rule in the cut elimination algorithm, the index of each recursive call on the right is smaller than the index of the call on the left. By this argument, the algorithm is guaranteed to terminate because, by the well-foundedness of the ordering, the indices of recursive calls cannot continue to decrease indefinitely.

We now examine each rule in the algorithm that has recursive calls and show that the indices are decreasing as required.

Rules 1 and 2. The size index of each recursive call is smaller (since each call is on a subterm of the original), and because each recursive call has the same environment ρ as the original the type indices are equal.

Rule 3. The inner calls to $\mathcal{GT}$ each have smaller size indices and equal type indices. The call to $\mathcal{GA}$ has a smaller size index (since the size of $\rho(z)$ is one element of the size index for the original call), and the type index is no larger (because the mapping for z may be the only mapping in ρ, otherwise the type index is smaller).

Rule 4. The inner calls to $\mathcal{GT}$ are all on subterms, and thus have identical type indices and smaller size indices. The call of $\mathcal{GA}$ has a type index that is definitely no larger than the original (and will be equal if the only binding in ρ is for z), and a size index that is smaller. The call to $\mathcal{GC}$ has a smaller type index: the type $z : As \to B$ is replaced by a term of type B (the result of the $\mathcal{GA}$ call).

Rule 7. Suppose the expression $\lambda x.\, u$ has the type $A \to B$, this is the only element in the type index for the call on the left. The inner call to $\mathcal{GT}$ has a type index that contains the element A (possibly many times). Because of the multiset ordering, this index is smaller than the original. The call to $\mathcal{GA}$ has a type index containing only the element B, which is smaller than $A \to B$. Note that in both cases the size index could be larger, but thanks to our ordering on indices this is irrelevant. Also note that if the type $A \to B$ were recursive, we could not guarantee that the type indices here would be smaller, which is one reason why we insist on the input to the algorithm being well-typed.

Rule 9. The recursive call is on a subterm, therefore the size index is smaller. The type index remains the same.

Rule 10. The size index is smaller due to the multiset ordering. The size of $C\ t_1 \ldots t_n$ is replaced by possibly multiple elements each with the size of one of the terms $t_1 \ldots t_n$. The type index gets smaller by the multiset ordering, since we replace the type $\{C_i\ As_i\}$ on the left by possibly multiple items of the types As_j.

Rules 11 and 12. Each recursive call is on a subterm, and therefore has an identical type index and smaller size index. □

Note that we cannot allow any recursively typed terms, for doing so would mean that the reasoning for rules 7 and 10 would be invalid. However, since one can embed positive recursive datatypes in the polymorphic lambda calculus, which is known to be strongly normalising, we believe there is a proof of termination for this algorithm that enables the sum-of-product types to be positively recursive.

Another proof of termination for cut elimination provided by Girard, Lafont and Taylor [GLT89].

3.5 Interlude: Recursion

In this Section we describe two methods for adding recursion to a typed lambda calculus based language. The goal is to find a suitable framework for extending cut elimination to recursive programs.

3.5.1 Cyclic Terms

The first method is strikingly simple: we just allow terms to be infinite in size. Now, infinite terms have a number of advantages over other methods of introducing recursion into a language.

- There are no recursive types involved. One way to add recursion to a language is to allow recursively typed expressions; this permits the definition of the fixpoint combinator, Y, which can then be used to define arbitrary recursive functions.
- Few modifications are required to a transformation algorithm in order for it to operate on infinite terms. The tricky bit is when you try to make the algorithm terminate, but we will come to this later.
- Infinite terms can have a regular structure, and those that do have a finite representation. It is only infinite terms with a regular structure that we are interested in, since (obviously) we cannot guarantee termination of our transformations if the input is infinite and irregular.

One way to represent regular infinite terms (or *cyclic terms*) is to use a system of labels. We add two extra term forms to the grammar of the language: a labelled cyclic term and a label reference. A labelled cyclic term takes the form $L : t$, where L is a label name and t is a term. The term t may then contain label references to the label L. For example, here is how the function *map* would be represented as a cyclic term:

$$\begin{aligned} map = \lambda f.\, \lambda xs.\, L : \textbf{case } xs \textbf{ of} \\ \quad Nil \quad\quad\quad &\rightarrow Nil \\ \quad Cons\ x\ xs &\rightarrow Cons\ (f\ x)\ L \end{aligned}$$

Looking closely at this definition, we realise that there is something slightly strange going on: some of the variables are being rebound in the cycle, in this case x and xs, while some variables (just f in this example) scope over the entire infinite term. In fact, the cyclic term bears a striking resemblance to the **for** loop construct of many imperative languages: some variables are updated each time around the loop, while others remain constant. With the labelled cyclic term syntax, we are making use of shadowing to provide an infinite number of different variables each with the same name.

The nature of the variable rebinding in a cyclic term is not just a curiosity, though; it actually complicates matters for substitution. Imagine taking the right hand side of the *map* definition above and substituting for f. That is easy, a textual substitution works fine. However, now try substituting for xs. If we do a textual substitution (substituting t for xs), we get:

$$\begin{aligned} map = \lambda f.\, \lambda xs.\, L : \textbf{case } t \textbf{ of} \\ \quad Nil \quad\quad\quad &\rightarrow Nil \\ \quad Cons\ x\ xs &\rightarrow Cons\ (f\ x)\ L \end{aligned}$$

Something is clearly wrong. Now, each time around the loop the **case** analyses t, which is not what we intended. The correct answer is to "unroll" the loop once:

$$
\begin{array}{l}
map = \lambda f.\, \lambda xs.\, \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad Nil \quad\quad\ \ \rightarrow Nil \\
\qquad\qquad Cons\ x\ xs \rightarrow Cons\ (f\ x)\ (L : \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Nil \quad\quad\ \ \rightarrow Nil \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad Cons\ x\ xs \rightarrow Cons\ (f\ x)\ L)
\end{array}
$$

Another way to write down a cyclic term is as a recursive system of equations, each one defining part of the term. This method was proposed by Mark Hopkins [Hop94]. Hopkins showed that free-variable and bound-variable operators could be defined for regular infinite terms by taking the least fixed point of the operator for the equation system defining the term. Hopkins then formalised the idea of substitution for regular infinite terms and finally proved that regular infinite terms were related to ordinary lambda terms by beta-equivalence. In Hopkins' system, the definition of *map* would look like this:

$$
\begin{array}{ll}
map & = \lambda f.\, \lambda xs.\, map' \\
map' & = \textbf{case}\ xs\ \textbf{of} \\
& \qquad Nil \quad\quad\ \ \rightarrow Nil \\
& \qquad Cons\ x\ xs \rightarrow Cons\ (f\ x)\ map'
\end{array}
$$

It is of course possible to define a cyclic term that has an infinite type. For example $L : \lambda x.\, L$ has a positive recursive type and $L : f\ L$ has a negative recursive type.

Cyclic terms are a simple and elegant way to introduce recursion into the lambda calculus. However the subtleties introduced by the rebinding/shadowing technique used to define them can mean practical problems for an implementation. After some experience with an implementation of cut elimination for cyclic terms, we rejected the method in favour of recursion equations.

3.5.2 Recursion equations

The more normal approach is to specify a recursive function using recursion equations, in the style of Burstall and Darlington in their work on program transformation [BD77]. The definition of *map* is the more familiar:

$$\begin{aligned} map\ f\ xs = \textbf{case}\ t\ \textbf{of}& \\ & Nil \rightarrow Nil \\ & Cons\ x\ xs \rightarrow Cons\ (f\ x)\ (map\ f\ xs) \end{aligned}$$

The rebinding is now explicit, in the form of arguments to the function.

We will use first-order recursion equations, in which all recursive function calls must be fully applied, to define a simple first-order deforestation algorithm. Then, by defining a language based on the lambda calculus language of Section 3.4.1 with first-order recursion equations, we will merge the simple deforestation algorithm and cut elimination to yield higher-order deforestation.

3.6 First-Order Deforestation

We now present a simple first-order deforestation algorithm. The algorithm is equivalent to the first-order deforestation of Wadler [Wad88, Wad90b], in the sense that given identical inputs both algorithms will produce identical output. However, the formulation is somewhat different. As we shall see, the deforestation algorithm here has been restructured to fit into the framework of cut-elimination, and is in some ways simpler than Wadler's algorithm.

3.6.1 Terms

Treeless terms are described by the language in Figure 3.6. We do not need a definition for non-treeless terms, because these never occur in the algorithm that follows. We have included residual **let** expressions in the language - these denote intermediate values that

t, u, v	::=	x	variable
	|	$C\ ts$	constructor application
	|	**case** x **of** $alts$	case expression
	|	**let** $x = t$ **in** u	let expression
	|	$f\ xs$	recursive function call
ts	::=	$t_1 \ldots t_n$	sequence of terms
xs	::=	$x_1 \ldots x_n$	sequence of variables
$alts$	::=	$\{C_1\ xs_1 \rightarrow v_1; \ldots; C_n\ xs_n \rightarrow v_n\}$	case alternatives

Figure 3.6: Treeless Terms for the Simple Deforester

will not be removed by the algorithm, in the same way that **let** was used in Chapter 2. By the addition of this term form, it is possible to represent any first-order recursive function in this language. An important point here is that the term treeless here does not correspond to normal form: this is thanks to the addition of the **let** construct. We shall tackle the problem of optimal conversion to treeless form in a later chapter.

Implicit with any operation on a term is a global set of recursive function definitions D. Each definition is of the form $f\ xs = t$ where the free variables of t are a subset of xs.

We assume a straight-forward semantics for the language, embodied by an evaluator, $\mathcal{E}$. The evaluator takes a term t and a value environment η, and returns the value of t where its free variables are given by η. The evaluator must of course also depend on D, but this value is global so we opt to omit it in most places. When we wish to be explicit about D, we will use a subscript as in $\mathcal{E}_D$.

3.6.2 Algorithm

The algorithm for first-order deforestation is shown in Figure 3.7. The $\mathcal{DT}$ operation takes a term t and a value environment ρ and returns a term u. The environment ρ is an idempotent mapping from free variables of t to terms. Idempotence for an environment is defined in the same way as before (Section 3.4.2). Using the evaluator $\mathcal{E}$, the semantics of

$$
\begin{array}{ll}
\mathcal{DT}\ x\ \rho & = \rho(x) \\
\mathcal{DT}\ (f\ xs)\ \rho & = \mathcal{DT}\ u\ [\rho(xs)/ys] \text{where } (f\ ys = u) \in D \\
\mathcal{DT}\ (C\ ts)\ \rho & = C\ (\mathcal{DTs}\ ts\ \rho) \\
\mathcal{DT}\ (\textbf{case}\ x\ \textbf{of}\ alts)\ \rho & = \mathcal{DC}\ (\rho(x))\ alts\ \rho \\
\mathcal{DT}\ (\textbf{let}\ x = t\ \textbf{in}\ u)\ \rho & = \textbf{let}\ x = (\mathcal{DT}\ t\ \rho)\ \textbf{in}\ (\mathcal{DT}\ u\ \rho) \\
 & \\
\mathcal{DTs}\ t_1 \ldots t_n\ \rho & = \mathcal{DT}\ t_1\ \rho \ldots \mathcal{DT}\ t_n\ \rho \\
 & \\
\mathcal{DC}\ x\ \{C_i\ xs_i \rightarrow v_i\}\ \rho & = \textbf{case}\ x\ \textbf{of}\ \{C_i\ xs_i \rightarrow \mathcal{DT}\ v_i\ \rho\} \\
\mathcal{DC}\ (f\ xs)\ alts\ \rho & = \mathcal{DC}\ u[xs/ys]\ alts\ \rho \\
 & \quad \text{where } f\ ys = u \\
\mathcal{DC}\ (C\ ts)\ \{\ldots; C\ xs \rightarrow v; \ldots\}\ \rho & = \mathcal{DT}\ v\ \rho[ts/xs] \\
\mathcal{DC}\ (\textbf{case}\ x\ \textbf{of}\ \{C_i\ xs_i \rightarrow v_i\})\ alts\ \rho & = \textbf{case}\ x\ \textbf{of}\ \{C_i\ xs_i \rightarrow \mathcal{DC}\ v_i\ alts\ \rho\} \\
\mathcal{DC}\ (\textbf{let}\ x = t\ \textbf{in}\ u)\ alts\ \rho & = \textbf{let}\ x = t\ \textbf{in}\ (\mathcal{DC}\ u\ alts\ \rho)
\end{array}
$$

Figure 3.7: Simple Deforester

$\mathcal{DT}$ can be expressed thus:

$$\mathcal{E}_D\ (\mathcal{DT}_D\ t\ \rho)\ \eta = \mathcal{E}_D\ t\ (\rho + \eta)$$

where $\rho + \eta$ is defined in the same way as before (Section 3.4.2).

Because ρ is restricted to being idempotent, each invocation of $\mathcal{DT}$ performs one level of substitution. For example, to simplify the expression $f\ (g\ (h\ x))$ two invocations of $\mathcal{DT}$ are required: $\mathcal{DT}\ (fy)\ [y/(\mathcal{DT}\ (g\ z)\ [z/(h\ x)])]$. This is a major difference between the algorithm presented here and the original deforestation algorithm of Wadler. This formulation is justified by the analogy to cut elimination, since our goal is to merge deforestation of recursive functions with cut elimination itself.

Termination of this algorithm is straightforward; a possible proof methodology is co-

induction. The proof is not included here.

3.7 Higher-Order Deforestation

So far we have an algorithm that removes intermediate data structures from arbitrary typed lambda calculus terms, and an algorithm that removes intermediate data structures from compositions of calls to first-order recursive functions. By combining the two, we will have a higher-order deforestation algorithm.

The higher-order deforestation algorithm is a conservative extension of the first-order algorithm; that is, it yields the same results for first order terms and functions as the first-order algorithm.

By applying cut elimination to recursive programs we lose the strongly normalising property of cut elimination, but this should be replaced by a general termination property for the combined algorithm. We have so far been unable to prove that the resulting algorithm terminates, but in Section 3.7.4 we give some necessary conditions for termination and conjecture that these are sufficient.

3.7.1 Syntax

The grammar of treeless terms in our hybrid language is given in Figure 3.8. Note that the domain of recursive function names is separate from the domain of variable names, and that recursive functions must be fully applied. This restriction is not serious, however, since it is now possible to write a recursive function that can be partially applied by using a lambda expression, for example:

$$\lambda x.\, \lambda y.\, \lambda z.\, f\ x\ y\ z$$

As before, we note that the grammar describes terms in normal form, save for the **let** construct. The intermediate structures to be removed are those present in the environment given to the transformation process.

t, u, v	::=	$\lambda x.\, a$	lambda abstraction
	|	$C\ t_1 \ldots t_n$	constructor application
	|	$z\ ts$	function application
	|	**case** $z\ ts$ **of** $alts$	case expression
	|	**let** $x = t$ **in** u	let expression
	|	$f\ xs$	recursive function call
ts	::=	$t_1 \ldots t_n$	sequence of terms
xs	::=	$x_1 \ldots x_n$	sequence of variables
$alts$	::=	$\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\}$	case alternatives

Figure 3.8: Treeless Terms for the Higher Order Deforester

As in the simple deforestation algorithm we assume a global set of recursive function definitions, D, in which each definition has the form $f\ xs = t$. We also assume an evaluation function for the language, $\mathcal{E}$.

3.7.2 Algorithm

The algorithm combining cut elimination with deforestation is shown in Figure 3.9. The environment ρ must be idempotent, as in the earlier algorithms.

The meanings of the three constituent operations can be defined in terms of our evaluation function, $\mathcal{E}$, as follows:

$$\begin{aligned}
\mathcal{E}\ (\mathcal{GT}\ \rho\ t)\ \eta &= \mathcal{E}\ t\ (\eta + \rho) \\
\mathcal{E}\ (\mathcal{GA}\ t\ ts)\ \eta &= \mathcal{E}\ (z\ ts)\ (\eta[(\mathcal{E}\ t\ \eta)/z] \\
\mathcal{E}\ (\mathcal{GC}\ \rho\ t\ alts)\ \eta &= \mathcal{E}\ (\textbf{case}\ z\ \textbf{of}\ alts)\ (\eta[(\mathcal{E}\ t\ \eta)/z] + \rho)
\end{aligned}$$

3.7.3 Knot Tying

Knot-tying for this algorithm can be achieved in two ways: knot-tying can be merged into the transformation algorithm itself, or we can adopt a two stage process where the trans-

$$
\begin{array}{ll}
\mathcal{GT}\ \rho\ (\lambda x.\, u) & = \lambda x.\, \mathcal{GT}\ \rho\ u \\
\mathcal{GT}\ \rho\ (C\ ts) & = C\ (\mathcal{GTs}\ \rho\ ts) \\
\mathcal{GT}\ \rho\ (z\ ts) & = \mathcal{GA}\ \rho(z)\ (\mathcal{GTs}\ \rho\ ts) \\
\mathcal{GT}\ \rho\ (\textbf{case}\ z\ ts\ \textbf{of}\ alts) & = \mathcal{GC}\ \rho\ (\mathcal{GA}\ \rho(z)\ (\mathcal{GTs}\ \rho\ ts))\ alts \\
\mathcal{GT}\ \rho\ (\textbf{let}\ x = t\ \textbf{in}\ u) & = \textbf{let}\ x = \mathcal{GT}\ \rho\ t\ \textbf{in}\ \mathcal{GT}\ \rho\ u \\
\mathcal{GT}\ \rho\ (f\ xs) & = \mathcal{GT}\ [\rho(xs)/ys]\ u \\
 & \quad \text{where } (f\ ys = u) \in D \\
 & \\
\mathcal{GTs}\ \rho\ t_1 \ldots t_n & = \mathcal{GT}\ \rho\ t_1 \ldots \mathcal{GT}\ \rho\ t_n \\
 & \\
\mathcal{GA}\ t\ [] & = t \\
\mathcal{GA}\ (\lambda x.\, u)\ (t : ts) & = \mathcal{GA}\ (\mathcal{GT}\ [t/x]\ u)\ ts \\
\mathcal{GA}\ (z\ ts)\ ts' & = z\ (ts +\!\!+ ts') \\
\mathcal{GA}\ (\textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ xs_i \to v_i\})\ ts' & = \textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ ys_i \to \mathcal{GA}\ v_i\ ts'\} \\
\mathcal{GA}\ (\textbf{let}\ x = t\ \textbf{in}\ u)\ ts & = \textbf{let}\ x = t\ \textbf{in}\ \mathcal{GA}\ u\ ts \\
\mathcal{GA}\ (f\ xs)\ ts & = \mathcal{GA}\ u[xs/ys]\ ts \\
 & \quad \text{where } (f\ ys = u) \in D \\
 & \\
\mathcal{GC}\ \rho\ (C\ ts)\ \{\ldots;\ C\ ys \to v;\ \ldots\} & = \mathcal{GT}\ \rho[ts/ys]\ v \\
\mathcal{GC}\ \rho\ (z\ ts)\ \{C_i\ xs_i \to v_i\} & = \textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ ys_i \to \mathcal{GT}\ \rho\ v_i\} \\
\mathcal{GC}\ \rho\ (\textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ xs_i \to v_i\})\ alts & = \textbf{case}\ z\ ts\ \textbf{of}\ \{C_i\ ys_i \to \mathcal{GC}\ \rho\ v_i\ alts\} \\
\mathcal{GC}\ \rho\ (\textbf{let}\ x = t\ \textbf{in}\ u)\ alts & = \textbf{let}\ x = t\ \textbf{in}\ \mathcal{GC}\ \rho\ u\ alts \\
\mathcal{GC}\ \rho\ (f\ xs)\ alts & = \mathcal{GC}\ \rho\ u[xs/ys]\ alts \\
 & \quad \text{where } (f\ ys = u) \in D
\end{array}
$$

Figure 3.9: Multi Cut Elimination With Loops

formation generates labelled expressions and the subsequent knot-tying process analyses the labels and makes new function definitions appropriately, as in the algorithm presented in Chapter 2 and indeed in the algorithm we implemented, Chapter 5. We believe the first method may better facilitate a proof of termination.

3.7.4 Proposition of termination

Our claim is that, given terms with finite application depth and no negative recursive types, the above algorithm will terminate. We will now give arguments as to why these two restrictions should be necessary to guarantee termination.

The guiding principle that we use in formulating the restrictions on the input is that the user should not be able to define a general fixpoint operator and present it as input to the deforestation algorithm. To allow this would surely invalidate the termination property, since the user could define arbitrary recursive functions using only the fixpoint operator applied to non-recursive lambda expressions. Since it cannot be possible to guarantee termination of the algorithm given arbitrary recursive functions, it seems reasonable to restrict the input to the algorithm in such a way that a general fixpoint combinator cannot be formulated. It will still be possible to define the Y combinator, but the resulting definition will involve **let** and therefore residual data structures; because of this it will be safe to transform.

There are two ways in which the fixpoint operator may be defined. Firstly, using a recursively typed expression:

$$fix = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

This definition is already outlawed according to the grammar we gave for the input to the algorithm, but it can be reformulated using **let**. The property of the definition we are interested in is the type of x: it has a cyclic type given by $\mu l.l \rightarrow A$. This is known as a negative recursive type. The *rank* of a cyclic type is given by the number of times the cycle passes through the argument position of an arrow. If the rank is even, the type is positive, otherwise it is negative. Terms involving only positive types are known to be

strongly normalising, but those involving negative types do not possess this property. This is supported by the fact that negative types are required to define a fixpoint combinator, as above (the type of x has a type with rank 1). It is insufficient to just Dis-sallow recursive types unless the cycle passes through a sum-of-products type (a common practice in the Hindley Minler type systems used in modern programming languages such as Haskell and ML), since this still allows a fixpoint combinator to be defined. Instead, we simply outlaw all negatively recursive types.

The second way to define a fixpoint combinator is to make use of the recursion mechanism provided by the language:

$$\mathit{fix} = \lambda f.f\ (\mathit{fix}\ f)$$

which doesn't involve any recursive types. However, the definition does possess *infinite application depth*, which means that if the definition were fully expanded, a descent of the expression tree would pass through an infinite number of application nodes. We outlaw this by requiring that all terms have a finite application depth. It is straightforward to determine whether a given set of recursive functions has infinite application depth by descending the expression tree for each function, expanding function calls, until a call to a function that has previously been expanded is encountered. If the descent required passing through an application node then the application depth is infinite.

Note that it seems sensible to impose a slightly stronger restriction, namely that recursive calls cannot appear in the argument to an application. This would appear to be a useful property on which to base a termination proof, but we have so far been unsuccessful in finding such a proof, although we believe one exists. In any case, there seems no reason why the more lenient criteria given above should not be sufficient to guarantee termination.

We have provided two restrictions on the input to the deforestation algorithm, and argued that they are necessary to ensure termination. Why, however, should these two restrictions alone be *sufficient* for termination? We have not been able to find a counter example, nor have we succeeded in finding a proof. This is a topic for future work.

Chapter 4

Other Issues

4.1 Conversion to Treeless Form

The deforestation algorithm presented in the previous chapter took as input recursive function definitions in treeless form. Treeless form includes the **let** construct, which renders it complete in the sense that any function can be represented in this language, by the addition of appropriate **lets**. However, the **let** construct acts as a barrier to deforestation: it introduces a *residual* data structure, one that will not be removed by the transformation. The danger in converting an arbitrary function into treeless form is that data structures which are eliminable will be rendered residual by the addition of a superfluous **let**.

For example, the fundamental list processing function *foldr* can be written like this:

$$\begin{aligned} \mathit{foldr}\ f\ c\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}& \\ \mathit{Nil} \quad &\rightarrow c \\ \mathit{Cons}\ x\ \mathit{xs}' &\rightarrow f\ x\ (\mathit{foldr}\ f\ c\ \mathit{xs}') \end{aligned}$$

This term has infinite application depth, so the addition of a **let** is required to represent it in treeless form:

$$
\begin{aligned}
\mathit{foldr}\ f\ c\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of}& \\
\mathit{Nil} &\rightarrow c \\
\mathit{Cons}\ x\ \mathit{xs}' &\rightarrow \mathbf{let}\ y = \mathit{foldr}\ f\ c\ \mathit{xs}'\ \mathbf{in}\ f\ x\ y
\end{aligned}
$$

which enables elimination of the input list xs, but makes the result of the recursive call to *foldr* residual. However, by the insertion of an extra **let**, we can make the input list residual too, preventing any removal of this data structure:

$$
\begin{aligned}
\mathit{foldr}\ f\ c\ \mathit{xs} = \mathbf{let}\ \mathit{ys} = \mathit{xs}\ \mathbf{in\ case}\ \mathit{ys}\ \mathbf{of}& \\
\mathit{Nil} &\rightarrow c \\
\mathit{Cons}\ x\ \mathit{xs}' &\rightarrow \mathbf{let}\ y = \mathit{foldr}\ f\ c\ \mathit{xs}'\ \mathbf{in}\ f\ x\ y
\end{aligned}
$$

In this section we show that there is an optimal conversion from an arbitrary expression in our extended lambda calculus language into the treeless form of the previous chapter. By "optimal" we mean that the largest subset of data structures in the resulting expression will be eliminable by the deforestation algorithm.

As an aside, note that if the definition of treeless form is relaxed, in some cases more data structures can be removed by the deforestation algorithm, but in other cases the algorithm may fail to terminate. For instance, if the original definition of *foldr* above is admitted as treeless form (and the deforestation algorithm adjusted accordingly—this is straightforward), then the call $\mathit{foldr}\ f\ c\ \mathit{xs}$ where f is bound to $\lambda x.\ \lambda \mathit{xs}.\ \mathit{Cons}\ x\ \mathit{xs}$ yields a list that is eliminable. This is the standard *append* function, which is an instance of *foldr*. However, if f is bound to $\lambda x.\ \lambda \mathit{xs}.\ \mathit{foldr}\ g\ x\ \mathit{xs}$, then the algorithm will fail to terminate as it encounters an ever increasing nesting of calls to *foldr*.

A somewhat lower-level description of a conversion-to-treeless-form algorithm can be found in Section 5.5 where we apply the principles from this section to the language used in our deforestation implementation.

4.1.1 Languages

The grammar for the full language is shown in Figure 4.1, and the grammar for the treeless subset of this language is shown in Figure 4.2.

The domains of function symbols and ordinary variables are distinct, and we use x, y, z to range over variables and f, g, h to range over recursive functions. As in the previous chapter, we assume a global set of recursive function definitions D. In the full language, there is no restriction on how a recursive function symbol is used, but in treeless form it may only be applied to a sequence of variables.

The grammar of treeless form also prohibits terms with infinite application depth. In fact, it is even more restrictive than this: a recursive function application cannot appear to the right of an application. The restriction is formulated in this way so that treeless form can be easily specified as a grammar.

Strictly speaking, because of this difference in the way treeless form is defined in this section, the algorithm here is not optimal with respect to the definition of treeless form in Chapter 3. However, the additional restriction is purely to enable a clearer presentation, and the algorithm presented can easily be extended to the treeless form of Chapter 3 (and indeed restricted to the treeless form of Chapter 2).

4.1.2 Algorithm

The algorithm to translate an expression in the full language into treeless form consists of two stages. Firstly, the expression is translated into normal form (such that applicative expressions have only variables at the head, and case expressions have only applicative expressions as the selector). This stage involves no addition of **let** terms. The second stage enforces the rules concerning recursive function calls, namely that calls must have variable-only arguments and not appear in the argument of an application or in the selector of a **case** expression. During this stage, certain expressions may be made residual.

Terms in the full language can be converted to normal form using the cut-elimination algorithm presented in the previous chapter, and this constitutes the first stage of conversion to treeless form.

t, u, v	$::=$	x	variable
	$\mid$	f	recursive function identifier
	$\mid$	$\lambda x.\, t$	lambda abstraction
	$\mid$	$C\ ts$	constructor application
	$\mid$	$t\ ts$	application
	$\mid$	**case** t **of** $alts$	case expression
	$\mid$	**let** $x = t$ **in** u	residual let expression
ts	$::=$	$t_1, \ldots, t_n$	sequence of terms
xs	$::=$	$x_1, \ldots, x_n$	sequence of variables
$alts$	$::=$	$\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\}$	case alternatives

Figure 4.1: The Full Language

t, u, v	$::=$	$\lambda x.\, t$	lambda abstraction
	$\mid$	$C\ ts$	constructor application
	$\mid$	$z\ ts'$	application
	$\mid$	$f\ xs$	recursive function application
	$\mid$	**case** $z\ ts'$ **of** $alts$	case expression
	$\mid$	**let** $x = t$ **in** u	residual let expression
t', u', v'	$::=$	$\lambda x.\, t'$	lambda abstraction
	$\mid$	$C\ ts'$	constructor application
	$\mid$	$z\ ts'$	application
	$\mid$	**case** $z\ ts'$ **of** $alts'$	case expression
	$\mid$	**let** $x = t'$ **in** u'	residual let expression
ts	$::=$	$t_1, \ldots, t_n$	sequence of terms
ts'	$::=$	$t'_1, \ldots, t'_n$	sequence of terms
xs	$::=$	$x_1, \ldots, x_n$	sequence of variables
$alts$	$::=$	$\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n\}$	case alternatives
$alts'$	$::=$	$\{C_1\ xs_1 \to v'_1; \ldots; C_n\ xs_n \to v'_n\}$	case alternatives

Figure 4.2: Treeless Form

The second stage consists of three key transformations. These transformations are applied to the term repeatedly until it is in treeless form, and they may be applied in any order.

The first transformation enforces the variable-only argument rule for recursive function calls, and consists of two rules, the first of which applies to function calls where one or more arguments are non-variables:

$$\begin{array}{ll} f\ ts & \Rightarrow \textbf{let}\ x_1 = t_1\ \textbf{in}\ \ldots \textbf{let}\ x_n = t_n\ \textbf{in}\ f\ xs \\ \textbf{let}\ x = y\ \textbf{in}\ t & \Rightarrow t[y/x], \text{if } x \text{ occurs at most once in } t \end{array}$$

Here we extract all the non-variable arguments to a recursive function application and make them residual. This transformation is optimal in the sense that there is no alternative way to translate the call into treeless form that would introduce fewer residual expressions. The reason for the side condition on the second rule above is that **let** expressions binding variables that occur multiple times are used to enforce linearity constraints, see Section 4.2.

The remaining two transformation rules extract recursive function calls that appear in the argument of an application. The grammar for treeless terms provides two places where applicative terms may appear: at the ground level, or in the selector of a **case expression**. The following rule extracts recursive function calls from ground level applicative terms:

$$x\ t_1 \ldots t_n \Rightarrow \textbf{let}\ x_1 = u_1\ \textbf{in}\ \ldots \textbf{let}\ x_k = u_k\ \textbf{in}\ x\ t'_1 \ldots t'_n$$

where $\{u_1, \ldots, u_k\}$ are the minimal free expressions of $t_1 \ldots t_n$ containing recursive calls.

s is the substitution $[u_1/x_1, \ldots, u_k/x_k]$

$t'_1 \ldots t'_n$ is the sequence of expressions such that $s(t'_1) = t_1, \ldots, s(t'_n) = t_n$

where the "minimal free expressions containing recursive calls" is the set of smallest subexpressions of an expression that contain recursive calls, with the additional constraint that each expression in the set cannot contain free instances of any variables bound within the original expression. The reason for extracting this set is that we wish to remove all the

recursive calls from the arguments to the application, and bind them outside the application with **let** expressions. Hence, the terms we extract cannot contain any free instances of variables that will become unbound once they are lifted from the application.

More formally, the minimal free expressions containing recursive calls for an expression t is the set of the smallest subexpressions of t satisfying the following criteria:

- Each subexpression contains at least one recursive call.
- No expression in the set contains any other as a subexpression.
- The set of free variables of each expression in the set is a subset of the free variables of t.

For example, for each of the following expressions we give the minimal free sets:

$f\ ts$	$\{f\ ts\}$
$f\ (g\ ts)$	$\{f\ (g\ ts)\}$
$\lambda z.f\ x$	$\{f\ x\}$
$\lambda x.f\ x$	$\{\lambda x.f\ x\}$
$\mathbf{let}\ z = f\ x\ \mathbf{in}\ g\ z$	$\{\mathbf{let}\ z = f\ x\ \mathbf{in}\ g\ z\}$
$\mathbf{let}\ z = f\ x\ \mathbf{in}\ g\ y$	$\{(f\ x), (g\ y)\}$
$(\lambda x.f\ x)\ (g\ us)$	$\{(\lambda x.f\ x), (g\ us)\}$

By extracting only the minimal free expressions, instead of simply extracting each argument to an application that contains recursive calls, we ensure that the maximum possible number of reductions can be performed during deforestation. For example, if we translated $(x\ (Cons\ y\ (f\ z)))$ as $(\mathbf{let}\ x' = Cons\ y\ (f\ z)\ \mathbf{in}\ x\ x')$ instead of $(\mathbf{let}\ x' = f\ z\ \mathbf{in}\ x\ (Cons\ y\ x'))$, we would inhibit the elimination of the *Cons* should x be instantiated by a **case** expression during deforestation.

The third and final transformation is applied to **case** selectors and is similar to the second transformation:

$$\textbf{case } x\ t_1 \ldots t_n \textbf{ of } alts \Rightarrow \textbf{let } x_1 = u_1 \textbf{ in } \ldots \textbf{let } x_k = u_k \textbf{ in case } x\ t'_1 \ldots t'_n \textbf{ of } alts$$

where $u_1 \ldots u_k$ are the minimal free expressions of $t_1 \ldots t_n$ containing recursive calls

$t'_1 \ldots t'_n$ are the expressions $t_1 \ldots t_n$ with $u_1 \ldots u_k$ replaced by $x_1 \ldots x_n$

Here we extract the maximal free expressions from the arguments to the application as before, but they are placed outside the **case** selector.

Applying the conversion to normal form followed by repeated application of the three rules above yields an optimal translation to treeless form. The algorithm we use to perform automatic conversion to treeless form in our deforestation implementation is described in Section 5.5.

4.2 Linearity

Blindly applying the deforestation algorithm to arbitrary programs carries an inherent risk: the resulting program may be *less* efficient than the original. This surprising result is due to non-linear features of the input program. We use the terms linear and non-linear here informally, with no direct connection to linear logic.

We shall discuss three problems that can cause degradation of performance. In each case, we give a solution to the problem involving either the addition of **let** bindings to the expression or the use of some additional transformations to recover sharing. We conjecture that by following the rules given in this section, no loss of performance will be incurred by applying the deforestation algorithm to a given program (defining performance to be 'number of reductions performed').

The techniques presented here are relevant to the deforestation algorithms presented in the previous two chapters, and also the algorithm which we have implemented, Section 5.6.

4.2.1 Duplication of work

A loss of efficiency can arise as the result of an expression being duplicated during transformation. Expressions can be duplicated when applying one of the reduction rules in the algorithm: **case** reduction, β-reduction, and recursive function unfolding. If a variable in a **case** pattern, a recursive function argument, or a lambda-abstracted variable appears more than once in the body of the construct then we say that the variable is *non-linear*. Non-linear variables are a potential source of duplication of expressions, and thus duplication of work when the resulting program is executed.

Perhaps the most obvious example of a function with a non-linear argument is *square*:

$$square\ x = x \times x$$

Deforesting the expression *square* t, where t is a possibly large and expensive-to-compute term, would yield the result $t \times t$. The problem is now evident: the time taken to evaluate the transformed expression is roughly double the time taken to evaluate the original, because the result of evaluating t is no longer shared.

Now consider this alternative definition of *square*:

$$square'\ x = \mathbf{let}\ x' = x\ \mathbf{in}\ x' \times x'$$

Now, deforesting the expression *square'* t, we get $\mathbf{let}\ x' = t\ \mathbf{in}\ x' \times x'$, which is no worse (or better) than the original. The difference is that the non-linear argument x has been *protected* by a residual **let** expression. After unfolding, the argument to the function was captured by the **let**, and not allowed to duplicate itself.

Note however that even if it was previously possible to eliminate the data structure represented by t when passed to a non-linear function, by forcing it to be residual this opportunity has been lost. It is not possible to determine statically when this is the case, so we must assume the worst and make all non-linear variables residual.

So, we have established that to prevent expressions being duplicated we must add a **let** construct for each non-linear variable in a function definition. Unfortunately, this is not

sufficient to guarantee that the result of deforesting a program is no less efficient than the original.

4.2.2 Full Laziness

When a program is executed using a fully-lazy evaluation strategy [Hug83, Pey87], all expressions are updated with their value once computed. This ensures that an expression whose value is required several times will only be evaluated once, the expression being replaced by its value after the first evaluation. To illustrate the difference between full laziness and the semi-lazy evaluation strategy of most modern compilers (including the Glasgow Haskell Compiler), consider the following simple function:

$$f = \lambda x.\, x + square\ z$$

each time the function f is applied, the value of the expression *square* z is required. Since this value does not depend on the argument x, its value could be evaluated once and cached for all future calls to f. This behaviour is characteristic of a fully-lazy evaluation strategy, but ordinary laziness loses this sharing.

With a non-fully-lazy implementation, every time a function is applied its definition is copied to the call site, replicating any subexpressions that may in fact be independent of the argument that the function was applied to.

However, a transformation scheme called the *full-laziness transformation* can transform a program such that it will have fully-lazy operational semantics, even though the evaluation strategy is not fully lazy. The full-laziness transformation modifies a program such that maximum sharing occurs during execution. The process consists of recursively extracting from each lambda binding the maximally free expressions, i.e. the largest subexpressions that contain no variables referred to by the binding.

4.2.3 Losing opportunities for full laziness

It is possible that performing the deforestation optimisation on a program can affect the behaviour of a subsequent full-laziness transformation. This is a serious problem, since many optimising compilers perform the full-laziness transformation as part of the optimisation process, and removing opportunities for full-laziness would impair the transparency of the full-laziness transformation.

The following program exhibits the problem described:

$$\textbf{let } f = \lambda g.\, map\ g\ (map\ h\ xs) \textbf{ in } sum\ (f\ h1) + sum\ (f\ h2)$$

The program contains a function f which takes an argument g, and returns the result of applying h followed by g to every element of the list xs. An intermediate data structure exists between the nested calls to *map* in the definition of f, which can be removed by deforestation. However, the expression $map\ h\ xs$ is independent of g, and could therefore be shared across multiple calls to f. The full-laziness transformation would yield

$$\textbf{let } ys = map\ h\ xs \textbf{ in let } f = \lambda g.\, map\ g\ ys \textbf{ in } sum\ (f\ h1) + sum\ (f\ h2)$$

If h is an expensive function to apply, then this transformation can have dramatic effects on the execution time of the program for a non-fully-lazy compiler.

However, if we perform deforestation before the full-laziness transformation, the result would be

$$\begin{array}{l}\textbf{let } f = \lambda g.\, f'\ g\ h\ xs \textbf{ in } sum\ (f\ h1) + sum\ (f\ h2)\\ \text{where}\\ f' = \lambda g.\, \lambda h.\, \lambda xs.\, \textbf{case } xs \textbf{ of}\\ \qquad\qquad Nil \rightarrow Nil\\ \qquad\qquad Cons\ x\ xs \rightarrow Cons\ (g\ (h\ x))\ (f'\ g\ h\ xs)\end{array}$$

The opportunity to extract the second call to *map* from the definition of f has been lost,

as the two calls to *map* are merged to remove the intermediate list.

Although removing the intermediate list provides some benefit, this is limited compared to the benefit of sharing this list between calls to f.

The following table compares the effect of deforestation with that of full laziness, taking n to be the number of times that f is called, and m to be the length of the list xs.

Deforestation	**Full laziness**
Removes construction of n lists.	Replaces construction of n lists with one list.
Removes deconstruction of n lists.	Has no effect on the deconstruction.
Has no effect on the calls to h.	Replaces $n \times m$ calls to h with m calls.

If we assume that calls to h are arbitrarily expensive, then deforestation can only be beneficial in the case where n is one. For all other values of n, full laziness is better. Since we cannot decide in general what the exact value of n is, the safest option is to perform the full-laziness transformation before deforestation.

4.2.4 Loss of laziness

Not only can deforestation remove opportunities for full laziness to provide a benefit, it can also remove sharing opportunities for ordinary laziness. This phenomenon is called *loss of laziness.*

To illustrate the problem, suppose the input to deforestation contains the following:

$$\begin{aligned} plus &= \lambda x.\, \lambda y.\, x + y \\ f &= \lambda g.\, \mathbf{let}\ g' = g\ \mathbf{in}\ \lambda x.\, \lambda y.\, g'\ x + g'\ y \end{aligned}$$

Notice that the non-linear argument to f, namely g, has been protected. If we now deforest the expression f $(plus\ t)$ 1 2 (where t is another arbitrarily large term), we get the following term:

$$\textbf{let } g' = \lambda y.\, t + y \textbf{ in } g'\ 1 + g'\ 2 \qquad (1)$$

In the original expression, t would have been updated with its value, once computed, and the second demand of t would have yielded the cached result. With the expression after deforestation, this is no longer the case.

How could we have detected the problem? Well, the cause stems from pushing an expression (namely t) inside a lambda term, which happened in this example when the argument x to *plus* was substituted. The result of this substitution, $\lambda y.\, t + y$ is now a normal form, and t will be recalculated every time it is applied. One sure way to avoid the problem is never to substitute expressions inside a lambda term; however this would hamper the process of deforestation, since we might miss some opportunities to remove intermediate data structures.

Can we force arguments to be residual, as before? If we inserted a **let**:

$$plus = \lambda x.\, \textbf{let } x' = x \textbf{ in } \lambda y.\, x + y$$

this captures t before it is pushed inside the lambda. However, there seems to be no justification for such a change, since the function *plus* might be used in a non-linear fashion elsewhere in the program, where there is no danger of losing laziness, and we do not wish to introduce residual data structures unnecessarily. In fact, the correct place to insert the **let** is in the original application of *plus*: replacing *plus* t with **let** $x = t$ **in** *plus* x. The problem is how to decide in general

- that a particular expression/function can cause loss of laziness, and
- where to place the **let** to ensure that updatability for the expression in danger is maintained.

The answer is that we don't have to. Notice that a valid transformation of the deforestation result (1) above is the following:

$$\textbf{let } x = t \textbf{ in let } g' = \lambda y.\, x + y \textbf{ in } g'\ 1 + g'\ 2$$

since t cannot contain y or g' as free variables. This is just the full-laziness transformation.

Any expression that is unfolded inside a lambda will automatically be a candidate for the full-laziness transformation. So, provided our compiler implements full laziness then our transformations are safe, because any loss of laziness that results from substitution will be reversed by the full-laziness transformation.

Combining this result with the conclusion of the previous section, we discover that in order to guarantee that deforestation does not impair the efficiency of the program, we must perform the full-laziness transformation on the program both before and after deforestation. We have used this strategy successfully in our implementation of deforestation described in Chapter 5.

4.2.5 Static argument transformation

To enable the full-laziness transformation to extract all the free expressions in the result from deforestation, it is sometimes necessary to employ an extra transformation stage.

Consider the following example, involving the function *map*:

$$\begin{aligned} map \ = \lambda f'.\, &\textbf{let } f = f' \textbf{ in } \lambda xs. \\ &\textbf{case } xs \textbf{ of} \\ &\quad Nil \ \rightarrow Nil \\ &\quad Cons\ x\ xs \rightarrow Cons\ (f\ x)\ (map\ f\ xs) \end{aligned}$$

The function f is applied to each element of the list. Thus, any computation which is performed when f is applied will be duplicated for each element of the list that is subsequently demanded. If we deforest the expression $map\ (plus\ t)\ xs$ where t is an expensive expression to compute, and using the definition of *plus* from above, we obtain the following result:

$$
\begin{array}{l}
h\ x_1 \ldots x_n\ xs \\
\text{where} \\
h\ = \lambda x_1. \ldots \lambda x_n. \textbf{let}\ f = \lambda y.\ t + y\ \textbf{in}\ \lambda xs. \\
\qquad \textbf{case}\ xs\ \textbf{of} \\
\qquad\quad \mathit{Nil} \qquad\quad \rightarrow \mathit{Nil} \\
\qquad\quad \mathit{Cons}\ x\ xs \rightarrow \mathit{Cons}\ (f\ x)\ (h\ x_1 \ldots x_n\ xs)
\end{array}
$$

(where $x_1 \ldots x_n$ are the free variables of t). Now, the expression t will be recomputed for each element of the list whereas previously the value of t would have been cached on the first evaluation. However, we can reverse this state of affairs by using some well-known transformation techniques.

The first task is to remove the arguments $x_1 \ldots x_n$ to h that have been introduced by the knot-tying process. These arguments prevent the expression t from being extracted by the full-laziness transformation. Fortunately, these arguments are the same every time h is called, and h is therefore subject to the *static argument transformation* [San95c]. Applying this optimisation to h, we get

$$
\begin{array}{l}
h\ = \lambda x_1. \ldots \lambda x_n. \textbf{let}\ f = \lambda y.\ t + y\ \textbf{in} \\
\quad \textbf{letrec} \\
\qquad h'\ = \lambda xs. \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\quad \mathit{Nil} \qquad\quad \rightarrow \mathit{Nil} \\
\qquad\qquad\qquad\quad \mathit{Cons}\ x\ xs \rightarrow \mathit{Cons}\ (f\ x)\ (h'\ xs) \\
\quad \textbf{in}\ h'
\end{array}
$$

Now, by inlining the function h at the call site (it can only occur once), we have

$$
\begin{array}{l}
\textbf{let } f = \lambda y.\, t + y \textbf{ in letrec} \\
\qquad h' \;\; = \lambda xs.\, \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathit{Nil} \qquad\quad \rightarrow \mathit{Nil} \\
\qquad\qquad\qquad\qquad \mathit{Cons}\ x\ xs \rightarrow \mathit{Cons}\ (f\ x)\ (h'\ xs) \\
\qquad \textbf{in } h'
\end{array}
$$

Next, we can apply the full-laziness transformation to extract the expression t from the function f, since it cannot contain instances of the bound variable y:

$$
\begin{array}{l}
\textbf{let } z = t \textbf{ in} \\
\textbf{let } f = \lambda y.\, z + y \textbf{ in} \\
\textbf{letrec} \\
\quad h' \;\; = \lambda xs.\, \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad\quad \mathit{Nil} \qquad\quad \rightarrow \mathit{Nil} \\
\qquad\qquad\qquad\quad \mathit{Cons}\ x\ xs \rightarrow \mathit{Cons}\ (f\ x)\ (h'\ xs) \\
\textbf{in } h'\ xs
\end{array}
$$

Now t is evaluated exactly once each time this expression is evaluated, which is identical to the situation before deforestation was applied.

4.2.6 Summary

To summarise, the following rules should be applied to ensure that deforestation does not impair the efficiency of a program.

- To avoid duplication of expressions, non-linear bindings must be protected by **let**.
- The full-laziness transformation must be applied to the program prior to deforestation, so that opportunities to increase sharing are not lost during transformation.
- After deforestation, the static argument transformation should be applied to all new function definitions to enable free expressions to be extracted.

- The full-laziness transformation should then be performed again, to recover any sharing that was lost by inlining expressions inside bindings.

We conjecture that by following these rules, the deforestation algorithm will not make a program less efficient than before.

An alternative and perhaps simpler solution to the problem is to make use of a linear type system. Traditional linear type systems are not suitable for two reasons: they consider expressions which are not required at all to be non-linear, and they also assign non-linear types to expressions which are required more than once, but will be shared by lazy evaluation. However, the type system proposed by Turner/Wadler/Mossin [TWM95] solves both of these problems, and could provide a suitable framework on which to base a deforestation algorithm that guarantees not to lose efficiency in the transformed program. Investigation of this alternative technique is planned as future work.

4.3 Transparency

Much of the motivation for formulating the deforestation algorithm from cut-elimination in the previous chapter came from the desire for the optimisation to be transparent. This we have achieved: there is a syntactic constraint on the program that guarantees that all intermediate data structures will be removed. Furthermore, we have a construct that enables the program to contain residual data structures, thus making treeless form expressive enough such that any function can be represented. The effects of transformation (that is, which data structures will be eliminated by the algorithm) are obvious by simply examining the source program.

The construct that makes this possible is residual **let**. By binding an expression with **let** we put in place a barrier to deforestation; no elimination of the structure represented by the expression bound by the **let** can take place. In some cases, it is obvious what effect this will have on deforestation. For example, consider the following standard definitions of *map* and *sum*:

$$
\begin{aligned}
sum &= \lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
&\qquad\quad Nil \rightarrow 0 \\
&\qquad\quad Cons\ x\ xs' \rightarrow \textbf{let}\ y = sum\ xs'\ \textbf{in}\ x + y \\
map &= \lambda f.\,\lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
&\qquad\quad Nil \rightarrow Nil \\
&\qquad\quad Cons\ x\ xs' \rightarrow Cons\ (f\ x)\ (map\ f\ xs')
\end{aligned}
$$

We can see by observation that the only residual data structure in these definitions is the integer result of the recursive call to *sum*, so we know that all other structures are eliminable. For example, we can deduce that the intermediate list in $sum\ (map\ f\ xs)$ will be removed by deforestation.

However, we can prevent removal of this list by insertion of a **let**, like this:

$$\textbf{let}\ ys = map\ f\ xs\ \textbf{in}\ sum\ ys$$

It is clear from the placement of the **let** that the data structure represented by $map\ f\ xs$ is residual. In fact, when the deforestation algorithm is applied to this new expression, the result will be simply a renaming of the original program.

In some cases, it can be more difficult to assess the impact of inserting a residual **let** in a program. Consider the following altered definition of *map*:

$$
\begin{aligned}
map &= \lambda f.\,\lambda xs.\,\textbf{case}\ xs\ \textbf{of} \\
&\qquad\quad Nil \rightarrow Nil \\
&\qquad\quad Cons\ x\ xs' \rightarrow \textbf{let}\ ys = map\ f\ xs'\ \textbf{in}\ Cons\ (f\ x)\ ys
\end{aligned}
$$

Which data structure is being made residual here? Well, if we expand the result of a call to this version of *map*, we get

$$\textbf{let}\ ys_1 = (\textbf{let}\ ys_2 = \ldots\ \textbf{in}\ Cons\ (f\ x_2)\ ys_2)\ \textbf{in}\ Cons\ (f\ x_1)\ ys_1$$

From this representation it is clear that all elements of the result list are residual, except for the head. Now, if we deforest the expression *sum* (*map* f xs) using this definition of *map*, the head element of the intermediate list between the applications of *sum* and *map* is removed, but the rest of the list is left in place. The result is:

$$
\begin{array}{l}
\textbf{case } xs \textbf{ of} \\
\quad \mathit{Nil} \quad\quad\quad \rightarrow 0 \\
\quad \mathit{Cons}\ x\ xs' \rightarrow \textbf{let } ys = g\ f\ xs' \textbf{ in } (f\ x) + (h\ ys) \\
\text{where } g \;\; = \lambda f.\, \lambda xs.\, \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathit{Nil} \quad\quad\quad \rightarrow \mathit{Nil} \\
\qquad\qquad\qquad\qquad \mathit{Cons}\ x\ xs' \rightarrow \textbf{let } ys = g\ f\ xs' \textbf{ in } \mathit{Cons}\ (f\ x)\ ys \\
\qquad h \;\; = \lambda xs.\, \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad\quad \mathit{Nil} \quad\quad\quad \rightarrow 0 \\
\qquad\qquad\qquad\quad \mathit{Cons}\ x\ xs' \rightarrow x + (h\ xs')
\end{array}
$$

Knowing the effect of adding a **let** construct to a function definition is useful when it is necessary to convert a function to treeless form, or to assess the result of an automatic conversion (Section 4.1). For example, the treeless form version of the *foldr* function looks like this:

$$
\begin{array}{l}
\mathit{foldr}\ f\ c\ xs = \textbf{case } xs \textbf{ of} \\
\qquad\qquad\qquad\quad \mathit{Nil} \quad\quad\quad \rightarrow c \\
\qquad\qquad\qquad\quad \mathit{Cons}\ x\ xs' \rightarrow \textbf{let } y = \mathit{foldr}\ f\ c\ xs' \textbf{ in } f\ x\ y
\end{array}
$$

Here the result of the recursive call to *foldr* is residual. Thus, if the result of this call is a data structure of some kind, then it cannot be removed during deforestation. An example of this is when the argument f to *foldr* is $\lambda x.\, \lambda xs.\, \mathit{Cons}\ x\ xs$, yielding the list append function. Because the recursive call to *foldr* is residual, only the head of the resulting list could be eliminated.

Another function which requires addition of a **let** to render it in treeless form is *iterate*:

$$\mathit{iterate} = \lambda f.\, \lambda x.\, \mathit{Cons}\ x\ (\mathit{iterate}\ f\ (f\ x))$$

which can be written in treeless form as follows:

$$iterate = \lambda f.\, \lambda x.\, Cons\ x\ (\textbf{let}\ y = f\ x\ \textbf{in}\ iterate\ f\ y)$$

Here the elements of the result list are residual, although the list itself is not.

To summarise, the residual **let** construct gives the programmer precise control over which data structures in the program should be removed by deforestation. With some thought, it is also possible to examine a function definition that has been converted to treeless form and decide which parts of the result are residual.

Chapter 5

Implementing Deforestation

This chapter will present an approach to implementing higher order deforestation in a functional language compiler. We cover the design of the optimisation pass, showing how the higher order deforestation algorithm works in the context of a real compiler.

We chose the Glasgow Haskell Compiler [Pey93] as the basis for our implementation, since it is designed in such a way that implementors can 'plug in' their own optimisation passes. It also provides a number of features that we found particularly helpful, including the ability to pass compiler specific optimisation details between the modules of a program in a separate-compilation situation. However, many of the techniques that are described in this chapter are not specific to the Glasgow Haskell Compiler, or even to Haskell. An implementation of deforestation can be constructed along these lines for any functional language compiler.

The next section describes the design goals for the implementation of deforestation, and is entirely language-independent. Section 5.2 describes how we resolved the design goals into a real deforestation model, and contains some details of how we mapped the design on to Haskell and the Glasgow Haskell Compiler. From Section 5.3 we go into more detail about the structure of the implementation, describing the algorithm used and some solutions to the problems facing a practical realisation of deforestation.

5.1 Design Goals

The main concern in the design of the optimisation pass is transparency. How much control should we give to the user? With the higher order deforestation algorithm, we have the ability to provide the programmer with complete control, by allowing him or her to augment the program with **lets** to indicate residual data structures. Of course, the programmer is then burdened with the task of making sure that recursive functions are in treeless form. This is often an unwelcome burden.

In many cases, the programmer will want to simply instruct the compiler to perform deforestation during the compilation process, and have the compiler remove as many intermediate data structures as possible. It would also be useful if we could deforest existing programs without needing to alter the code in any way.

We have already shown that there is an optimal conversion from an arbitrary higher order language into treeless form, so this stage can be automated. In fact, there is no problem in theory with applying the deforestation algorithm to all parts of the program that meet the criteria for a legal input to the algorithm.

However, there is a pragmatic problem with the fully automatic scheme. If the deforestation algorithm is allowed to run unchecked on the whole program it can create code explosion (an ungainly increase in code size). Informally, the reason for this is that recursive functions are always expanded at the call site, creating one copy of the function for each call. In some cases, the unfolded function will be subject to deforestation, combining with other unfolded functions to produce new specialised functions that use less intermediate store. In other cases, the function call will not be in a suitable position for deforestation to occur, and the result will be a copy of the original function at the call site (possibly specialised with respect to its higher order arguments - a small additional benefit). Although the unnecessary unfolding of functions does not in theory adversely affect the efficiency of the program, in practice the increase in code size can be a problem—both at compilation and run-time. Later stages of the compiler will have to deal with an unduly large program, resulting in longer compilation times. An unreasonably large executable can put extra strain on the virtual memory system of the computer, leading to large increases in the perceived execution time of the program.

So, while the fully automatic scheme is desirable, compromises will have to be made in order for it to be practical. Section 5.2 will present one solution to this problem.

5.2 A Model for a Deforestation Compiler Pass

In this section we detail the decisions that were made before implementing the deforestation optimisation in the Glasgow Haskell Compiler, including our solutions to the problems outlined in the previous section.

5.2.1 User annotations vs. Automatic compiler annotations - A compromise

The solution we chose to this problem allows existing programs to be deforested (to a certain extent) without programmer intervention, but also allows the programmer full control over the deforestation process if he is prepared to provide annotations to guide the deforester.

The choice of which recursive functions to unfold is left to the programmer. To indicate to the deforester that a particular function is to be unfolded during deforestation, the programmer places a special annotation, or *pragma*, in the program. For example,

```
{-# DEFOREST f #-}
```

indicates that the function f is to be unfolded. Since the syntax `{-...-}` normally denotes a comment in Haskell, a compiler that does not understand this form of annotation will ignore it.

Converting function definitions to treeless form is done automatically, with the programmer having the option of forcing some expressions to be *residual*. A residual expression will be converted into a **let**-expression at transformation time [1], and the intermediate data structure that it represents will not be removed by deforestation. To indicate that an expression is to be residual, the programmer applies the pseudo-function `_residual` to it.

[1] The **let** term form has a special meaning to the deforestation algorithm, in that it always represents a residual data structure. See Section 5.4

The Standard Prelude

It appears from the description above that, in order for any deforestation to be performed, the programmer must place some annotations in his program. However, this is not the case: in Haskell, there are a large number of functions that are built in to the language, and must be available to all programs. This set of functions is collectively called the Standard Prelude.

The Standard Prelude (or Prelude for short) includes a wide range of list-processing functions, such as *map*, *filter*, *sum*, and *foldr*. Since the definitions of these functions are invisible to the programmer (although their semantics are specified by the language definition), we as compiler writers are free to annotate them for deforestation.

The approach taken in our implementation is to annotate for unfolding all Prelude list-processing functions that can be successfully used during deforestation. This has been found to yield a modest increase in code size (see Chapter 6) while allowing the maximum amount of intermediate data to be removed without user annotations.

Enumerated Lists

Enumerated lists in Haskell take four forms: `[n..]`, `[n..m]`, `[n,m..]`, and `[n,m..o]`. At compile time, these enumerated list constructs are converted into calls to the Prelude list generating functions *enumFrom*, *enumFromTo*, *enumFromThen* and *enumFromThenTo* respectively. Accordingly, all we need to do to make these lists deforestable is to annotate the corresponding functions in the Prelude for deforestation.

List Comprehensions

There are a number of conversions from list comprehension syntax into an equivalent expression without list comprehensions [Pey87, HPW+92, GLJ93]. The Glasgow Haskell Compiler uses two: the conversion into recursive function definitions of Wadler [Pey87] and the `foldr/build` conversion of Gill/Launchbury/Peyton-Jones [GLJ93]. The latter strategy permits deforestation based on the `foldr/build` method.

We use the former method, and automatically annotate any function definitions introduced to be unfolded during deforestation. This enables any lists consumed or produced by a list comprehension to be removed by deforestation.

For instance, we can always remove the intermediate list in expressions of the form

```
[ f x | x <- [n..m] ]
```

because the translation inside the compiler produces an expression similar to

```
letrec
        h = \xs -> case xs of
                        Nil -> Nil
                        Cons x xs -> Cons (f x) (h xs)
in
        h (enumFromTo n m)
```

and since the function h is annotated as deforestable, the intermediate enumerated list can be removed during deforestation. Note that the function generated is in treeless form; this is true for all functions produced by the list comprehension translator.

Other common uses of list comprehensions which are a source of intermediate lists include expressions like

```
[ f x y | (x,y) <- zip xs ys ]
```

where we can eliminate the list of pairs produced by *zip*.

Arrays

In Haskell, an array is built using the *array* function, which has the following type:

```
array :: (Ix a) => (a,a) -> [Assoc a b] -> Array a b
```

The construction of an array requires an upper and lower bound for the index (the type of the index can be any member of the `Ix` class), and a list of association pairs to initialise the array. The association pairs are defined by the `Assoc` datatype:

```
data Assoc a b = a := b
```

Each association pair in the list passed to `array` initialises one element of the array, by placing value `b` at index `a`.

For example, suppose we would like to construct a lookup table for the *sin* operation. We will construct an array with indices $0 \ldots 180$, where the value at each index x is $sin(\pi x/180)$. The following Haskell expression builds such an array:

```
array (0,180) [ x := sin (pi * x / 180) | x <- [0..180] ]
```

The above definition, while clear and abstract, suffers from some serious inefficiencies. There are two intermediate lists:

- The list `[0..180]` is built and immediately consumed by the list comprehension. The previous Section describes how this list will be removed by deforestation.
- The list of association pairs produced by the list comprehension is consumed by `array` during construction of the final array.

The second list can be removed by unfolding the definition of `array` during deforestation. This may present problems with some implementations of Haskell, since `array` is often a primitive operation without its own Haskell definition. However, in Glasgow Haskell, the `array` function is defined in terms of lower level primitive array operations, and we can unfold its definition as normal.

5.2.2 The Module System

Separate compilation often presents a barrier to global optimisation techniques. Without a means by which information can be propagated between modules, many optimisations can

be lost. For example, it is desirable to have available information regarding the strictness of functions in one module whilst compiling a separate module that refers to those functions.

The Haskell module system defines *interface* files that allow information to be passed between modules of a program. Each module has an associated interface which gives the types of functions defined in that module. The interface for a module is consulted when compiling further modules that use functions from the first module. Interface files may be written by the programmer (in fact, this is required when modules are mutually recursive), but more commonly the interface for a module is generated automatically whilst compiling the module itself.

The Glasgow Haskell Compiler makes additional use of this form of inter-module communication by placing optimisation information in the interface file, in the form of pragmas. There is normally one pragma for each function defined in the module, and it takes the following form:

```
{-# GHC_PRAGMA ... #-}
```

where the ... is replaced by a large amount of optimisation information, including the strictness, arity, and update behaviour of the function.

In some cases, the pragma can contain the definition of the function itself, so that importing modules can inline the function at call sites. This is typically used for small functions, where the speed advantage outweighs the small increase in code size that can result from inlining.

Our deforestation implementation also makes use of this facility by forcing the definition of any function annotated for deforestation to be placed in the interface pragma for that function. Thus, any module that imports an annotated function will have its definition available during the deforestation phase, and compositions involving the imported function will be deforested as normal. This technique essentially bridges the module gap, such that there is no penalty for spreading a program across several modules.

Since the Prelude itself is just a collection of Haskell modules (bar some primitive objects that are built-in to the compiler), the definitions for our annotated functions appear in the interface for the Prelude, and are available to the compiler when compiling any user program.

5.2.3 Summary

We have shown how our implementation of deforestation will work from the user's point of view, including which standard Haskell functions and constructs will be deforested without the need for special annotations.

The advantages of the scheme we have outlined can be summarised as follows:

- The programmer has full control over the deforestation of his program through the use of optional annotations in his program.
- If the annotations are omitted, some removal of intermediate lists may still take place where the programmer has made use of enumerated lists, list comprehensions, array constructions and Prelude list-processing functions.
- There are no compromises to be made in the deforestation transformation itself—it simply picks expressions that fit the input criteria and transforms them, unfolding only functions that have been annotated beforehand.

The disadvantages are:

- No deforestation of user-written functions and data structures will take place unless the user annotates his program. For example, if the user writes tree-processing functions that he wishes to be deforestable, they must be annotated as such. Probably the most common problem occurs when the user writes his own list-processing functions (or his own versions of Prelude list-processing functions), which will not be unfolded by the deforester unless specifically annotated.
- The deforestation pass will unfold all annotated functions, whether any intermediate structures can be removed or not. This can result in an increase in the size of the object code, but we have typically found this increase to be small (this is quantified in Chapter 6). In fact, the Glasgow Haskell Compiler already inlines a number of functions for speed, including *map*, *append*, and *foldr*.

The scheme is somewhat limited, in that we only attempt to remove intermediate lists and ignore all user-defined data structures, but it is an effective compromise. After all, the programmer is still free to annotate his program to gain maximum benefit from deforestation.

5.3 Structure of the Deforestation Implementation

The algorithm used in our implementation is a variant of the higher-order deforestation algorithm described in Chapter 2. This algorithm was chosen over the improved algorithm in Chapter 3 for two reasons:

- It has a termination proof, which is essential for an optimisation to be included in a production compiler.
- It is more efficient than the cut-elimination algorithm. This is due to there being no nested calls to the transformer (the algorithm never transforms code twice).

The disadvantage of using this algorithm is that we have to obey the slightly more restrictive definition of treeless form, which does not correspond exactly to the normal form of our language. We avoid the problem of the extra introduction of **let** bindings (Section 2.7.1), by requiring only that the result is in normal form, not treeless form. This may sound like we are abandoning transparency, but in fact as noted in Chapter 2, the intermediate structures which would be rebound with **let** during transformation are not in fact removable.

Opting for a separate knot-tying pass requires that the implementation of the transformation system is *lazy*, which in itself presents some problems for avoiding name capture. The difficulties and our solutions are described in Section 5.9.

The next section describes the Core language. This is the small functional language used internally by the Glasgow Haskell Compiler. The Haskell source code passes through a dependency analyser and type checker before being translated into Core. All the optimisation passes in the compiler take a program in Core and yield a result in Core; these are the so-called Core-to-Core passes, of which our deforestation implementation is one.

Section 5.4.1 describes the subset of Core which we define as treeless form. As noted above, we use the definition of treeless form from Chapter 2, but extended with the extra constructs found in the Core language.

Section 5.5 gives our algorithm for automatic conversion of arbitrary Core expressions into treeless form. We use a conversion process which is a simplification of the algorithm

described in Section 4.1, adapted for the treeless-form definition in Section 5.4.1. This is applied to all functions which are annotated by the user for deforestation, before they are used in transformation.

Section 5.6 gives the algorithm used by our deforestation implementation in three parts: the transformation scheme for Core expressions (except **letrec**), the transformation scheme for Core programs, and the transformation scheme for **letrec** expressions.

Section 5.9 describes our method for avoiding name capture during the transformation process. Section 5.7 gives the algorithm for knot-tying.

5.4 Glasgow Haskell Core Language

The Glasgow Haskell Core language which forms the input to our deforestation implementation is shown in Figure 5.1. The Core language is derived from the second-order polymorphic lambda calculus, with additional constructs for manipulation of primitive values (integers, floating-point numbers etc.), and algebraic data types similar to those used in earlier chapters.

We use x, y, z, f, g, h to range over variables, l to range over literals, t, u, v to range over terms, xs, ys, zs to range over sequences of variables, ts, us, vs to range over sequences of terms, *alts* to range over lists of algebraic or primitive case alternatives, and *defs* to range over lists of recursive definitions.

There are two types of variables: those that are annotated as deforestable (i.e. have an associated function definition that was annotated as deforestable by either the user or the compiler) and those that are not. When we wish to distinguish the two types, we will use f, g, h to represent deforestable variables and x, y, z for all others.

The language includes explicit type abstraction and application. Additionally, each variable in a Core program (variable instances and bindings) has an associated type, which is stored along with the program. All this type information must be kept consistent during transformation through type substitutions.

We use T and U to range over types and Ts to range over sequences of types. The actual grammar of types is not given here because it has little bearing on the algorithm to be

t, u, v	::=	l	literal
	|	x	variable
	|	$\lambda x.\, u$	lambda abstraction
	|	$\Lambda X.\, u$	type abstraction
	|	$C\ ts$	constructor application
	|	$\oplus\ ts$	primitive application
	|	$t\ ts$	application
	|	$t\ Ts$	type application
	|	**case** t **of** *alts*	case expression
	|	**let** $x = t$ **in** u	residual let expression
	|	**letrec** *defs* **in** t	letrec expression
ts	::=	$\langle t_1, \ldots, t_n \rangle$	sequence of terms
Ts	::=	$\langle T_1, \ldots, T_n \rangle$	sequence of types
xs	::=	$\langle x_1, \ldots, x_n \rangle$	sequence of variables
alts	::=	$\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n; x \to v\}$	algebraic case alternatives
	|	$\{l_1 \to v_1; \ldots; l_n \to v_n; x \to v\}$	primitive case alternatives
defs	::=	$\langle x_1 = t_1, \ldots, x_n = t_n \rangle$	recursive function definitions

Figure 5.1: The Core Language

presented; however, we do assume two operations involving types: $U[T/X]$ denotes the type given by substituting the type T for the type variable X in the type U, and $t[T/X]$ denotes the term given by substituting the type T for the variable X in all the types contained in the term t, including those associated with each variable instance.

Constructor applications and primitive applications in Core are always saturated; constructor applications always yield an algebraic type, and primitive applications yield a primitive type.

There are two kinds of **case** expression in Core: one for algebraic datatypes and one for primitive types. These **case** constructs are slightly different than those we have encountered before: it is not necessary for a complete set of alternatives to be present, but if the list is incomplete then a *default case* must be included. The default case takes the form $x \to v$

and binds the value of the selector expression to x in the term v, if none of the other cases match. The default case may be omitted for algebraic case expressions when alternatives for all of the constructor in the datatype are provided, but it may not be omitted for primitive case expressions. It is worth noting that the Core expression

$$\textbf{case } t \textbf{ of } \{x \to v\}$$

is not equivalent to a Haskell **case** expression of the same form, because the Core version has a strict semantics. That is, if the value of t is $\perp$, then the value of the expression as a whole is $\perp$, but this is not necessarily true for the equivalent Haskell expression. The Core expression equivalent to a Haskell expression in the above form is **let** $x = t$ **in** v.

In the following chapter we will use the abbreviation $\{C_i\ xs_i \to v_i; x \to v\}$ for $\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n; x \to v\}$, and $\{l_i \to v_i; x \to v\}$ for $\{l_1 \to v_1; \ldots; l_n \to v_n; x \to v\}$.

With regard to the typing of Core expressions, we require that all terms in the input to the deforestation algorithm be well-typed according to the Hindley/Milner type system. This is enforced by the type checker in the compiler, so any code that is passed to the deforestation algorithm automatically satisfies the restriction.

The Core language given here differs slightly from the Core language used in the rest of the Glasgow Haskell Compiler—for other transformations in the compiler it was found more convenient to restrict expressions in the argument position of an application to be atomic (variables and primitive values only). In our implementation we use a simple transformation between standard Glasgow Haskell Core and our modified form, which involves the inlining of certain **let** expressions. The transformation is careful not to duplicate expressions or expand a **let** expression where this would cause duplication of work at run-time.

We assume a straightforward non-strict semantics for Core embodied by an evaluator $\mathcal{E}$. The evaluator takes a Core term t and an environment η mapping free variables of t to values, and returns the value of the term. The exact domain of values is not important to us, but suffice to say it includes all the base types in Haskell together with the algebraic datatypes defined in the program and the Standard Prelude.

t, u, v	$::=$	l	literal
	$\mid$	$\lambda x.\, u$	lambda abstraction
	$\mid$	$\Lambda X.\, u$	type abstraction
	$\mid$	$C\ ts$	constructor application
	$\mid$	$\oplus\ ts$	primitive application
	$\mid$	$f\ ps$	deforestable function application
	$\mid$	$z\ ps$	normal function application
	$\mid$	**case** z **of** *alts*	case expression
	$\mid$	**let** $x = t$ **in** u	residual let expression
	$\mid$	**letrec** *defs* **in** t	letrec expression
a, b, c	$::=$	t	term argument
	$\mid$	T	type argument
p	$::=$	x	variable argument
	$\mid$	T	type argument
ps	$::=$	$\langle p_1, \ldots, p_n \rangle$	sequence of variable arguments
ts	$::=$	$\langle t_1, \ldots, t_n \rangle$	sequence of terms
xs	$::=$	$\langle x_1, \ldots, x_n \rangle$	sequence of variables
alts	$::=$	$\{C_1\ xs_1 \to v_1; \ldots; C_n\ xs_n \to v_n; x \to v\}$	algebraic case alternatives
	$\mid$	$\{l_1 \to v_1; \ldots; l_n \to v_n; x \to v\}$	primitive case alternatives
defs	$::=$	$\langle x_1 = t_1, \ldots, x_n = t_n \rangle$	recursive function definitions

Figure 5.2: Core Treeless Terms

5.4.1 Treeless form

The grammar for Core terms in treeless form is given in Figure 5.2. It is essentially the subset of terms in Core that are in normal form, with the exceptions that we use **let** to indicate residual data structures, function arguments must be variables or types, and **case** selectors must be variables.

5.4.2 Labelled terms

The transformation scheme we will present shortly takes a Core term and some treeless recursive Core functions and yields an infinite labelled term. The knot-tying process takes the infinite result of transformation and produces a finite treeless term and some new treeless recursive functions. The intermediate language is Core augmented with a **label** construct, which takes the form **label** t u. There are four restrictions on terms of this form:

- Semantically, the two subterms must be identical.
- u is a labelled term and t is an unlabelled term.
- t may contain references to deforestable functions, but u may not (except in the first component of **label** subterms).
- The set of free variables in u is always a subset of the free variables in t.

5.5 Conversion to Treeless Form

Before function definitions are used for deforestation, they are converted from Core to treeless form. This is a two stage process. The first stage is to convert the Core term to normal form (i.e. applicative expressions are of the form variable applied to list of terms/types, and case selectors are applicative expressions). An arbitrary Core term can be reduced to this form by applying the deforestation transformation (Section 5.6.1) with σ equal to $\langle\rangle$.

$$
\begin{array}{lcl}
\mathcal{F}\ fs\ l & = & l \\
\mathcal{F}\ fs\ (\lambda x.\, u) & = & \lambda x.\, \mathcal{F}\ fs\ u \\
\mathcal{F}\ fs\ (\Lambda X.\, t) & = & \Lambda X.\, \mathcal{F}\ fs\ t \\
\mathcal{F}\ fs\ (C\ ts) & = & C\ (\mathcal{F}s\ fs\ ts) \\
\mathcal{F}\ fs\ (\oplus\ ts) & = & \oplus\ (\mathcal{F}s\ fs\ ts) \\
\mathcal{F}\ fs\ (x\ as) & = & \mathcal{F}_{app}\ fs\ x\ as\ \langle\rangle \\
\mathcal{F}\ fs\ (\textbf{case}\ x\ as\ \textbf{of}\ alts) & = & \textbf{let}\ z = \mathcal{F}_{app}\ fs\ x\ as\ \langle\rangle\ \textbf{in case}\ z\ \textbf{of}\ (\mathcal{F}_{alts}\ fs\ alts) \\
\mathcal{F}\ fs\ (\textbf{let}\ x = t\ \textbf{in}\ u) & = & \textbf{let}\ x = \mathcal{F}\ fs\ t\ \textbf{in}\ \mathcal{F}\ fs\ u \\
 & & \\
\mathcal{F}s\ fs\ \langle t_1, \ldots, t_n \rangle & = & \langle \mathcal{F}\ fs\ t_1, \ldots, \mathcal{F}\ fs\ t_n \rangle \\
 & & \\
\mathcal{F}_{app}\ fs\ x\ \langle\rangle\ bs & = & x\ bs \\
\mathcal{F}_{app}\ fs\ x\ (T : as)\ bs & = & \mathcal{F}_{app}\ fs\ x\ as\ (bs +\!\!+ \langle T \rangle) \\
\mathcal{F}_{app}\ fs\ x\ (z : as)\ bs & = & \mathcal{F}_{app}\ fs\ x\ as\ (bs +\!\!+ \langle z \rangle) \\
\mathcal{F}_{app}\ fs\ x\ (t : as)\ bs & = & \textbf{let}\ z = \mathcal{F}\ fs\ t\ \textbf{in}\ \mathcal{F}_{app}\ fs\ x\ as\ (bs +\!\!+ \langle z \rangle) \\
 & & \\
\mathcal{F}_{alts}\ fs\ \{C_i\ xs_i \to v_i; x \to v\} & = & \{C_i\ xs_i \to \mathcal{F}\ fs\ v_i;\ x \to \mathcal{F}\ fs\ v\} \\
\mathcal{F}_{alts}\ fs\ \{l_i \to v_i; x \to v\} & = & \{l_i \to \mathcal{F}\ fs\ v_i;\ x \to \mathcal{F}\ fs\ as\}
\end{array}
$$

Figure 5.3: Conversion to Treeless Form

The second stage converts the normal form term to treeless form, taking into account the set of variables which will be unfolded in the subsequent deforestation transformation. The algorithm for this stage is given in Figure 5.3. The set of variables which will be unfolded changes during deforestation, so the conversion procedure takes as an argument the list of variables currently considered to be deforestable (fs).

The conversion to treeless form given here is optimal, in the sense that there is no other way to convert a term to treeless form that would allow more intermediate structures to be removed (see Section 4.1).

5.6 Algorithm

We now describe the deforestation algorithm implemented in the Glasgow Haskell Compiler. In Section 5.6.1 we give the basic transformation scheme for Core expressions without **letrec**. Section 5.6.2 gives the transformation scheme for **letrec** expressions, and Section 5.6.3 describes how the deforestation algorithm is applied to whole programs. The knot-tying algorithm (which is required by both the **letrec** and the top-level transformation schemes) is described in detail in Section 5.7.

5.6.1 Transformation for expressions

The transformation scheme for Core expressions (except **letrec**) is shown in Figures 5.4 and 5.5. It consists of two mutually-recursive operations, $\mathcal{T}$ and $\mathcal{C}$, and two support operations, $\mathcal{T}s$ and $\mathcal{T}_{alts}$. The $\mathcal{T}$ operation transforms a term applied to a number of arguments (where the list of arguments may be empty, and can contain both terms and types), and $\mathcal{C}$ transforms a **case** expression applied to a list of arguments, with a selector that is also applied to a list of arguments. The $\mathcal{T}s$ operation transforms a sequence of terms, and $\mathcal{T}_{alts}$ transforms lists of case alternatives.

Note that the transformation scheme is split into two functions for clarity alone; it is certainly possible to have a single function that performs all the transformation, but to avoid duplicating **case** terms on the left we opted for the two-operation approach.

$$
\begin{array}{ll}
\mathcal{T}\ \sigma\ l\ \langle\rangle & = l \\
\mathcal{T}\ \sigma\ x\ as & = x\ (\mathcal{T}s\ \sigma\ as) \\
\mathcal{T}\ \sigma\ f\ as & = \mathbf{label}\ (f\ as)\ (\mathcal{T}\ \sigma\ (\sigma(f))\ as) \\
\mathcal{T}\ \sigma\ (\lambda x.\, u)\ \langle\rangle & = \lambda x.\, \mathcal{T}\ \sigma\ u\ \langle\rangle \\
\mathcal{T}\ \sigma\ (\lambda x.\, u)\ (t : as) & = \mathcal{T}\ \sigma\ u[t/x]\ as \\
\mathcal{T}\ \sigma\ (\Lambda X.\, t)\ \langle\rangle & = \Lambda X.\, (\mathcal{T}\ \sigma\ t\ \langle\rangle) \\
\mathcal{T}\ \sigma\ (\Lambda X.\, t)\ (T : as) & = \mathcal{T}\ \sigma\ t[T/X]\ as \\
\mathcal{T}\ \sigma\ (C\ ts)\ \langle\rangle & = C\ (\mathcal{T}s\ \sigma\ ts) \\
\mathcal{T}\ \sigma\ (\oplus\ ts)\ \langle\rangle & = \oplus\ (\mathcal{T}s\ \sigma\ ts) \\
\mathcal{T}\ \sigma\ (t\ ts)\ as & = \mathcal{T}\ \sigma\ t\ (ts +\!\!+ as) \\
\mathcal{T}\ \sigma\ (t\ Ts)\ as & = \mathcal{T}\ \sigma\ t\ (Ts +\!\!+ as) \\
\mathcal{T}\ \sigma\ (\mathbf{case}\ t\ \mathbf{of}\ alts)\ as & = \mathcal{C}\ \sigma\ t\ \langle\rangle\ alts\ as \\
\mathcal{T}\ \sigma\ (\mathbf{let}\ x = t\ \mathbf{in}\ u)\ as & = \mathbf{let}\ x = (\mathcal{T}\ \sigma\ t\ \langle\rangle)\ \mathbf{in}\ (\mathcal{T}\ \sigma\ u\ as) \\
 & \\
\mathcal{T}s\ \sigma\ \langle t_1, \ldots, t_n \rangle & = \langle \mathcal{T}\ \sigma\ t_1\ \langle\rangle, \ldots, \mathcal{T}\ \sigma\ t_n\ \langle\rangle \rangle
\end{array}
$$

Figure 5.4: Transformation for Core Expressions

$$
\begin{array}{ll}
\mathcal{C}\ \sigma\ l\ \langle\rangle\ alts\ as & = \textbf{case}\ l\ \textbf{of}\ (\mathcal{T}_{alts}\ \sigma\ alts\ as) \\
\mathcal{C}\ \sigma\ x\ as\ alts\ bs & = \textbf{case}\ x\ (\mathcal{T}s\ \sigma\ as)\ \textbf{of}\ (\mathcal{T}_{alts}\ \sigma\ alts\ bs) \\
\mathcal{C}\ \sigma\ f\ as\ alts\ bs & = \textbf{label}\ (\textbf{case}\ f\ as\ \textbf{of}\ alts)\ (\mathcal{C}\ \sigma\ \sigma(f)\ as\ alts\ bs) \\
\mathcal{C}\ \sigma\ (\lambda x.\,u)\ (t : as)\ alts\ bs & = \mathcal{C}\ \sigma\ u[t/x]\ as\ alts\ bs \\
\mathcal{C}\ \sigma\ (\Lambda X.\,t)\ (T : as)\ alts\ bs & = \mathcal{C}\ \sigma\ t[T/X]\ as\ alts\ bs \\
\mathcal{C}\ \sigma\ (C\ ts)\ \langle\rangle\ \{\ldots; C\ xs \to v; \ldots\}\ as & \\
\qquad = \mathcal{T}\ \sigma\ v[ts/xs]\ as & \\
\mathcal{C}\ \sigma\ (C\ ts)\ \langle\rangle\ \{\ldots; x \to v\}\ as & = \mathcal{T}\ \sigma\ v[(C\ ts)/x]\ as \\
\mathcal{C}\ \sigma\ (\oplus\ ts)\ \langle\rangle\ alts\ as & = \textbf{case}\ (\oplus\ (\mathcal{T}s\ \sigma\ ts))\ \textbf{of}\ (\mathcal{T}_{alts}\ \sigma\ alts\ as) \\
\mathcal{C}\ \sigma\ (t\ ts)\ as\ alts\ bs & = \mathcal{C}\ \sigma\ t\ (ts +\!\!+ as)\ alts\ bs \\
\mathcal{C}\ \sigma\ (t\ Ts)\ as\ alts\ bs & = \mathcal{C}\ \sigma\ t\ (Ts +\!\!+ as)\ alts\ bs \\
\mathcal{C}\ \sigma\ (\textbf{case}\ t\ \textbf{of}\ \{C_i\ xs_i \to v_i; x \to v\})\ as\ alts\ bs & \\
\qquad = \mathcal{C}\ \sigma\ t\ \langle\rangle \left\{ \begin{array}{ll} C_i\ xs_i & \to (\textbf{case}\ v_i\ \textbf{of}\ alts)\ as; \\ x & \to (\textbf{case}\ v\ \textbf{of}\ alts)\ as \end{array} \right\}\ bs & \\
\mathcal{C}\ \sigma\ (\textbf{case}\ t\ \textbf{of}\ \{l_i \to v_i; x \to v\})\ as\ alts\ bs & \\
\qquad = \mathcal{C}\ \sigma\ t\ \langle\rangle \left\{ \begin{array}{ll} l_i & \to (\textbf{case}\ v_i\ \textbf{of}\ alts)\ as; \\ x & \to (\textbf{case}\ v\ \textbf{of}\ alts)\ as \end{array} \right\}\ bs & \\
\mathcal{C}\ \sigma\ (\textbf{let}\ x = t\ \textbf{in}\ u)\ as\ alts\ bs & = \textbf{let}\ x = (\mathcal{T}\ \sigma\ t\ \langle\rangle)\ \textbf{in}\ (\mathcal{C}\ \sigma\ u\ as\ alts\ bs) \\
\\
\mathcal{T}_{alts}\ \sigma\ \{C_i\ xs_i \to v_i; x \to v\}\ as & = \{C_i\ xs_i \to \mathcal{T}\ \sigma\ v_i\ as;\ x \to \mathcal{T}\ \sigma\ v\ as\} \\
\mathcal{T}_{alts}\ \sigma\ \{l_i \to v_i; x \to v\}\ as & = \{l_i \to \mathcal{T}\ \sigma\ v_i\ as;\ x \to \mathcal{T}\ \sigma\ v\ as\}
\end{array}
$$

Figure 5.5: Transformation for **case** expressions

Each of the transformation operations depends on an environment σ which contains the definitions of all the functions which are to be unfolded during transformation. The environment σ is only updated when a **letrec** subterm is transformed (see Section 5.6.2).

The transformation scheme has no nested recursive calls to the transformation operations, although it does have explicit substitution in arguments to recursive calls. The substitution can be implemented by an extra environment argument to the transformation operations, which yields a transformation scheme with linear complexity.

The meanings of the transformation operations can be expressed in terms of the Core evaluator, $\mathcal{E}$, as follows:

$$\begin{aligned}
\mathcal{E}\ (\mathcal{T}\ \sigma\ t\ as)\ \rho &= \mathcal{E}\ (\textbf{letrec}\ \sigma\ \textbf{in}\ t\ as)\ \rho \\
\mathcal{E}\ (\mathcal{C}\ \sigma\ t\ as\ alts\ bs)\ \rho &= \mathcal{E}\ (\textbf{letrec}\ \sigma\ \textbf{in}\ ((\textbf{case}\ (t\ as)\ \textbf{of}\ alts)\ bs))\ \rho
\end{aligned}$$

5.6.2 Nested letrec expressions

In this section we deal with **letrec** expressions present in the the expression being deforested. This is the first time we have presented a solution to this problem, although the techniques described here can be adapted without much difficulty to the algorithms of Chapters 2 and 3.

The transformation for **letrec** expressions requires the knot-tying function $\mathcal{K}$, which we will describe in Section 5.7. An invocation of $\mathcal{K}$ takes the form $\mathcal{K}\ t$, where t is a labelled Core term. The result is of the form $(t', defs)$, where t' is a treeless term and *defs* is a sequence of treeless definitions extracted from t.

Any of the functions defined by a **letrec** may be annotated as deforestable; this can occur as the result of automatic recursive function generation and tagging by the compiler, for example when list comprehension expressions are translated into Core (Section 5.2.1). A nested list comprehension is translated into deforestable recursive functions which themselves contain **letrec**s defining deforestable recursive functions.

Since the recursive functions themselves may contain expressions that we wish to deforest, it is desirable to apply the transformation algorithm and knot-tyer to each recursive function definition before it is placed in the environment σ for further unfolding during

$$
\begin{array}{l}
\mathcal{T}\ \sigma\ (\textbf{letrec}\ (defs +\!\!+ defs')\ \textbf{in}\ t)\ as = \textbf{letrec}\ defs''\ \textbf{in}\ (\mathcal{T}\ \sigma'\ t\ as) \\
\quad \begin{array}{lll}
\text{where} & \langle f_1 = u_1, \ldots, f_n = u_n \rangle & = defs \\
& \langle x_1 = v_1, \ldots, x_n = v_n \rangle & = defs' \\
& \langle v_1', \ldots, v_n' \rangle & = \langle \mathcal{T}\ \sigma\ v_1\ \langle\rangle, \ldots, \mathcal{T}\ \sigma\ v_n\ \langle\rangle \rangle \\
& defs'' & = \langle x_1 = v_1', \ldots, x_n = v_n' \rangle \\
& \langle (u_1', gs_1), \ldots, (u_n', gs_n) \rangle & = \langle \mathcal{K}\ (\mathcal{T}\ \sigma\ u_1\ \langle\rangle), \ldots, \mathcal{K}\ (\mathcal{T}\ \sigma\ u_n\ \langle\rangle) \rangle \\
& hs & = \mathrm{dom}(\sigma) +\!\!+ \langle f_1, \ldots, f_n \rangle \\
& \langle u_1'', \ldots, u_n'' \rangle & = \langle \mathcal{F}\ hs\ u_1', \ldots, \mathcal{F}\ hs\ u_n' \rangle \\
& \sigma' & = \sigma +\!\!+ \langle f_1 = u_1'', \ldots, f_n = u_n'' \rangle +\!\!+ gs_1 +\!\!+ \ldots +\!\!+ gs_n
\end{array}
\end{array}
$$

Figure 5.6: Transformation scheme for **letrec** expressions

transformation of the rest of the program. The right hand side of each recursive definition is therefore transformed using the current environment, σ. This has the side-effect of leaving the definition in normal form, which makes the job of converting to treeless form somewhat easier.

The transformation scheme for **letrec** functions is shown in Figure 5.6. First, we split the definitions into deforestable and normal functions, and transform the right hand side of each definition. We then apply the knot-tyer to the transformed right hand sides of the deforestable functions, and collect all the new definitions that result. The new definitions are converted to treeless form with respect to a new σ, which is built from the original σ augmented with the extracted functions from knot-tying along with the new deforestable functions from the **letrec** expression. The body of the **letrec** is then transformed in this new environment. We leave behind a **letrec** binding just the non-tagged function definitions.

Using this scheme, function definitions in the environment may contain **letrec** expressions, but they will contain no deforestable functions, thus avoiding the possibility of the transformation getting into an infinite loop processing deforestable functions.

$$
\begin{array}{lll}
\mathcal{TP}\ \sigma\ \langle\rangle & = \langle\rangle & \\
\mathcal{TP}\ \sigma\ ((defs +\!\!+ defs') : defss) & = defs'' : \mathcal{TP}\ \sigma'\ defss & \\
\quad \text{where } \langle f_1 = u_1, \ldots, f_n = u_n\rangle & = defs & \\
\quad\quad \langle x_1 = v_1, \ldots, x_n = v_n\rangle & = defs' & \\
\quad\quad \langle (u'_1, gs_1), \ldots, (u'_n, gs_n)\rangle & = \langle \mathcal{K}\ (\mathcal{T}\ \sigma\ u_1\ \langle\rangle), \ldots, \mathcal{K}\ (\mathcal{T}\ \sigma\ u_n\ \langle\rangle)\rangle & \\
\quad\quad \langle (v'_1, hs_1), \ldots, (v'_n, hs_n)\rangle & = \langle \mathcal{K}\ (\mathcal{T}\ \sigma\ v_1\ \langle\rangle), \ldots, \mathcal{K}\ (\mathcal{T}\ \sigma\ v_n\ \langle\rangle)\rangle & \\
\quad\quad hs & = \mathrm{dom}(\sigma) +\!\!+ \langle f_1, \ldots, f_n\rangle & \\
\quad\quad \langle u''_1, \ldots, u''_n\rangle & = \langle \mathcal{F}\ hs\ u'_1, \ldots, \mathcal{F}\ hs\ u'_n\rangle & \\
\quad\quad defs'' & = \langle x_1 = v'_1, \ldots, x_n = v'_n\rangle +\!\!+ & \\
 & \quad \langle f_1 = u''_1, \ldots, f_n = u''_n\rangle +\!\!+ & \\
 & \quad gs_1 +\!\!+ \ldots +\!\!+ gs_n +\!\!+ hs_1 +\!\!+ \ldots +\!\!+ hs_n & \\
\quad\quad \sigma' & = \sigma +\!\!+ \langle f_1 = u''_1, \ldots, f_n = u''_n\rangle +\!\!+ & \\
\quad\quad gs_1 +\!\!+ \ldots +\!\!+ gs_n & &
\end{array}
$$

Figure 5.7: Top-Level Transformation Scheme

5.6.3 Top-level transformation

A Core program is a sequence of mutually-recursive function groups arranged in dependency order, such that earlier function groups do not refer to later ones. The top-level transformation scheme simply treats the program as a nested **letrec** expression with no body, see Figure 5.7. The transformation scheme for each group of definitions is similar to that for a **letrec** expression, and we split the definitions into deforestable and non-deforestable as before. The transformed deforestable functions and the extracted functions from the knot-tyer are placed in the new environment for transforming the rest of the program as before, but we also keep all the transformed definitions in case any of them are exported from the module.

5.7 Knot-tyer

The knot-tying process takes the infinite labelled output from transformation and yields a finite treeless term and some treeless recursive functions. In order to do this it must search

$$
\begin{array}{l}
\mathcal{K}\ t = (t', defs) \\
\quad \text{where } (t', used, defs) = \mathcal{KT}\ \langle\rangle\ t \\
\\
\mathcal{KT}\ ls\ (\textbf{label}\ t\ u) = \\
\qquad \text{if } loops = \langle\rangle \\
\qquad\quad \text{then if } f \in used \\
\qquad\qquad\quad \text{then } (f\ (Xs +\!\!+ xs), used, \langle f = \Lambda Xs.\, \lambda xs.\, u' \rangle) \\
\qquad\qquad\quad \text{else } (u', used, defs) \\
\qquad\qquad \text{where } f \text{ is fresh, tagged deforestable} \\
\qquad\qquad\qquad xs = \text{freeVars}(t) \\
\qquad\qquad\qquad Xs = \text{freeTypeVars}(t) \\
\qquad\qquad\qquad (u', used, defs) = \mathcal{KT}\ ((t, f, xs, Xs) : ls)\ u \\
\qquad\quad \text{else } (f\ (Xs +\!\!+ s(xs)), \langle f \rangle, \langle\rangle) \\
\qquad\qquad \text{where } ((f, xs, Xs, s) : \ldots) = loops \\
\quad \text{where } loops = \{(f, xs, Xs, s) \mid (t', f, xs, Xs) \leftarrow ls, s(t') = t\}
\end{array}
$$

Figure 5.8: Knot-Tyer

for loops in the input term.

A loop is defined as follows: the term **label** t u is a looping term if it is a subterm of **label** t' u' and t is a strict renaming of t'. One term is a strict renaming of another if only the *untagged* variables (i.e. the non-deforestable variables) are renamed. Renamings must also be consistent; for example, the term $(x\ y)$ is not a renaming of the term $(z\ z)$, but the reverse is true.

Our algorithm for knot-tying is shown in Figure 5.8. Given an infinite labelled term, the algorithm $\mathcal{K}$ will return a finite term and a list of new function definitions. The new definitions are closed; that is, they have no free variables, provided that the free variable property of labels is true for the input (Section 5.4.2).

The algorithm makes use of two auxiliary functions: freeVars(t) is the set of free variables in t, freeTypeVars(t) is the set of free type variables in the type of t. If s is a mapping from variables to variables, then $s(t)$ is the term t with the mapping s applied to all its free variables. Hence if $s(u) = t$ then the term t is a consistent strict renaming of the term

u, and the substitution that expresses that renaming is s.

The function $\mathcal{KT}$ forms the heart of the knot-tying process. It takes a list of tuples and the current term. There is one tuple in the list for each **label** term of which the current term is a subterm of the second component. Each tuple takes the form (t, f, xs, Xs), where

- t is the first component of the **label** term,
- f is a function name allocated to this **label**,
- xs is the list of free variables of the **label**, and
- Xs is the list of free type variables in the type of the **label**.

A call to $\mathcal{KT}$ returns a tuple of the form $(t', used, defs)$, where

- t' is the result term,
- *used* is the list of function names for which we found loops,
- *defs* is the list of new function definitions referred to in the result.

When a term of the form **label** t u is encountered in the input, it is checked against all the elements of the list of tuples ls. For each of the elements of ls (of the form (t', f, xs, Xs)), if t is a renaming of t' then we have found a loop. A call to the function f is generated at this point, with $Xs +\!\!+ s(xs)$ as the argument list, where s is a substitution such that $s(t') = t$. To indicate that a loop was found for this particular **label**, we return the list $\langle f \rangle$ in the second component of the result.

If t is not a renaming of any t', then we construct a new tuple (t, f, xs, Xs), where f is a fresh function name, xs is the list of free variables of t with duplicates removed, and Xs is the list of free type variables in the type of t. The new tuple is added to ls before invoking $\mathcal{KT}$ on u. If a match for this **label** is found in a subterm of u (indicated by f being present in the second component of the result of the recursive call), then a new definition needs to be extracted at this point. The right hand side of the definition is simply the result term

from applying $\mathcal{KT}$ to u. We then construct a call to the new function and return this as the result.

We have described only the case of $\mathcal{KT}$ for **label** terms here, as the rest of the cases form a straightforward traversal of the input term. Where a term form has more than one subterm, the lists of bindings and used function names from applying $\mathcal{KT}$ to the subterms are concatenated to produce the result of knot-tying the whole term.

When generating new function definitions, the knot-tyer must abstract both the free variables and the free type variables from the function, in order to keep the type information in the program consistent. Note also that any functions generated by the knot-tyer are themselves marked deforestable: there is no reason not to do this, provided that we also convert the definitions to treeless form before using them. The conversion to treeless form is necessary because the definition of treeless form depends on the set of functions in the domain of σ; thus when a new set of function definitions are added to σ, they must be converted to treeless form with respect to the new σ.

5.8 Improving the Knot-Tyer

We now consider a number of improvements to the basic knot-tying algorithm given in the previous section. The improvements are all related to the quality of the output from the deforestation algorithm: in a number of cases, we have found by experimentation that the simple knot-tyer is inefficient in finding loops, which leads to an increase in the size of the code being generated. Since the output from the knot-tyer is used as further input to deforestation (for example when transforming deforestable functions before they are unfolded), then any unnecessary increase in code size is amplified when the code is re-transformed.

We consider several cases where unnecessary code explosion occurs, and propose solutions to each one. Most of these techniques have been implemented, as noted below.

Finally, in Section 5.8.5 we consider a technique for improving the performance of the knot-tyer, which is by definition an expensive computation. The technique described reduces the complexity of the average case, leading to important improvements in the efficiency of

the deforestation optimisation.

5.8.1 Back Loops

Through experimentation with the knot-tyer as described above, we discovered that in certain cases the output was larger than expected. In particular, the knot-tyer appeared to have skipped a few labels before finding a loop and generating the new function. While the intermediate structure had been removed as expected, the resulting program was much larger than necessary.

The problem can be illustrated with the *append* function, defined as follows:

$$
\begin{array}{lll}
append & = \lambda xs.\, \lambda ys.\, \textbf{case}\ xs\ \textbf{of} & \\
 & \quad Nil & \rightarrow ys \\
 & \quad Cons\ x\ xs' & \rightarrow Cons\ x\ (append\ xs'\ ys)
\end{array}
$$

Suppose the deforestation algorithm is required to transform the expression $f\ zs\ zs$ (there are no intermediate data structures here, but we would expect the transformation/knot-tyer to yield a function isomorphic to *append*).

The output from $\mathcal{T}\ \sigma\ (append\ zs\ zs)$ will be:

$$
\begin{array}{l}
\textbf{label}\ (append\ zs\ zs) \\
\quad (\textbf{case}\ zs\ \textbf{of} \\
\quad\quad Nil \rightarrow zs \\
\quad\quad Cons\ x\ xs' \rightarrow \textbf{label}\ (append\ xs'\ zs) \\
\quad\quad\quad\quad\quad\quad\quad\quad (\textbf{case}\ xs'\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Nil \rightarrow zs \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Cons\ x'\ xs'' \rightarrow \textbf{label}\ (append\ xs''\ zs) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\ldots)))
\end{array}
$$

The first attempt at finding a renaming will attempt to match the label $(append\ xs'\ zs)$ against $(append\ zs\ zs)$. This is an invalid renaming, because it is inconsistent: zs is

renamed to both xs' and zs. However, the next match does succeed: $(append\ xs''\ zs)$ is a valid renaming of $(append\ xs'\ zs)$ and a new function is generated at the point of this label. The result is therefore (ignoring free type variables):

$$
\begin{array}{l}
\textbf{case}\ zs\ \textbf{of} \\
\quad Nil \rightarrow zs \\
\quad Cons\ x\ xs' \rightarrow g\ xs'\ zs \\
\textbf{where}\ \ g = \lambda xs'.\, \lambda zs.\, \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \rightarrow zs \\
\qquad\qquad\qquad\qquad Cons\ x'\ xs'' \rightarrow g\ xs''\ zs
\end{array}
$$

Although no loss of efficiency has occurred in terms of extra computation, the size of the code is needlessly larger than we would expect, and this imposes its own penalty on the execution speed of the program.

The above observations led us to develop a simple principle that allows the extra code in examples such as the above to be eliminated. The key idea is that if the term t is an inconsistent renaming of u, it may be the case that u is a valid renaming of t. This situation is called a *back-loop*. When a call to $\mathcal{KT}$ finds a back loop, the original label in the outer call can be replaced by the result of the inner call, after applying the substitution that arises from the renaming.

Operationally, when a call to $\mathcal{KT}$ finds a previous expression that is a renaming of the current expression, it applies the substitution to the result of the current call and returns the new expression as a back-loop (this requires an extra field in the tuple returned from $\mathcal{KT}$, containing a list of function name/back-loop expression pairs). The outer call checks for any back-loops in the result of the recursive call, and returns the corresponding expression if one is found.

We found that the extra complexity this adds to the knot-tyer is compensated for by the reduction in code size that results from the implementation of back-loops.

5.8.2 Boring expressions

Another source of superfluous code in the output from the knot-tyer is failure to detect a loop early enough when the original expression in the input contains expressions where future labels have variables. For example, consider an input to the deforester of the form *map f t*, where *t* represents a data structure that cannot be eliminated (such as a call to an external function). With *map* defined in the usual way, the output from the transformation will be

```
label (map f t)
      (case t of
         Nil       → Nil
         Cons x xs →
           Cons (f x) (label (map f xs)
                             (case xs of
                                Nil       → Nil
                                Cons x' xs' →
                                  Cons (f x') (label (map f xs')
                                                     (...)))))
```

The first attempt to find a loop tries to match *map f t* with *map f xs*, which is not a renaming. The loop is found on the second iteration, however, because *map f xs'* is a renaming of *map f xs*, yielding the result

```
case t of
  Nil       → Nil
  Cons x xs → Cons (f x) (g xs)
where g = λxs. case xs of
                 Nil       → Nil
                 Cons x' xs' → Cons (f x) (g xs')
```

As in the back-loop example above, the body of the function appears to have been unrolled once, effectively doubling the size of the result.

m	::=	x	variable (non-deforestable)
	|	$m\ as$	application
	|	**case** m **of** *balts*	case expression
balts	::=	$\{C_1\ xs_1 \to m_1; \ldots; C_n\ xs_n \to v_m; x \to m\}$	algebraic case alternatives
	|	$\{l_1 \to m_1; \ldots; l_n \to v_m; x \to m\}$	primitive case alternatives

Figure 5.9: Boring Expressions

The solution to this problem that we have adopted in our implementation is to extract any expressions that cannot be involved in the removal of intermediate data structures, and rebind them using the **let** construct. For instance, the term *map f t* in the above example becomes **let** $x = t$ **in** *map f x*, and the knot-tying will succeed on the first iteration.

The subclass of Core terms that are *boring*, i.e. cannot represent eliminable data structures, is given by the grammar in Figure 5.9. Although we can also consider **let** expressions to be boring, we do not include them in the grammar. Instead, they are dealt with differently, as described in the following section.

The extraction can be implemented simply by adding two new transformation rules dealing with application:

$$\begin{aligned} \mathcal{T}\ \sigma\ (t\ (ts +\!\!+ \langle m \rangle))\ as &= \textbf{let}\ x = m\ \textbf{in}\ \mathcal{T}\ \sigma\ t\ (x : as) \\ \mathcal{C}\ \sigma\ (t\ (ts +\!\!+ \langle m \rangle))\ as\ alts\ bs &= \textbf{let}\ x = m\ \textbf{in}\ \mathcal{C}\ \sigma\ t\ (x : as)\ alts\ bs \end{aligned}$$

Because the boring expressions are extracted before the application is transformed (when the head is examined), they do not appear in any **label** expressions that may be generated as a result of unfolding the head of the application, and thus do not impair the knot-tying process from identifying renamings on the first iteration.

Extracting expressions using **let** may introduce residual data structures and hence impair transparency. This could happen if we are transforming the body of a deforestable recursive definition in order to unfold it later—the expressions that we are extracting are boring with

respect to this transformation, but on unfolding later they may represent eliminable data structures.

The solution to this problem is to remove the **lets** we have generated after transformation, and before the conversion to treeless form. Each expression that has been extracted in this way can be safely inlined after transformation, because there is no risk of duplicating the expression provided that the linearity rules given in Section 4.2 have been adhered to. In order to remember which **let** expressions appear in the output as a result of extraction, we simply use a special class of variable names when generating the new binding.

5.8.3 Extracting lets

Extracting the boring expressions is often not enough to make labels match: another situation that arises is a label of the form $(t\ (\mathbf{let}\ x = u\ \mathbf{in}\ v))$, which is required to match against $(t'\ v')$, where t is a renaming of t' and v of v'. If v is not a boring expression, we cannot use the extraction technique described above. However, the following transformation is valid, and achieves the required effect:

$$\mathcal{T}\ \sigma\ (t\ (ts +\!\!+ \langle \mathbf{let}\ x = u\ \mathbf{in}\ v\rangle))\ as\ \ = \mathbf{let}\ x = \mathcal{T}\ \sigma\ u\ \ \mathbf{in}\ \mathcal{T}\ \sigma\ (ts +\!\!+ \langle v\rangle)\ as$$

This transformation alone is not sufficient to extract all the **let** expressions from the argument to an application. There may be more deeply nested **let** terms in v, which we also wish to extract. We found by experimentation that it is not necessary to extract as many **lets** as possible from the argument, only those that are found by descending the term through the left side of applications and into the selector of **case** expressions. Further **lets** occurring in the argument will be extracted later in the transformation process.

We define the more general **let** extraction scheme as follows. The idea is to float **let** terms towards the top level, so they may be extracted using a similar rule to that given above. In order to do this, we use a term-rewriting scheme, and define precisely the subterms of the subject term to which the rewrite rules are applicable.

An *extraction context* (an expression with a hole, represented by []) is given by the definition

$$
\begin{array}{rcl}
E & ::= & [] \\
& | & E\ ts \\
& | & \textbf{case}\ E\ \textbf{of}\ alts
\end{array}
$$

A context E with the hole replaced by a term t is written $E[t]$. Three rewrite rules now define the **let** extraction:

$$
\begin{array}{lll}
E[t] & \rightarrow E[t'] & \text{if } t \rightarrow t' \\
(\textbf{let}\ x = u\ \textbf{in}\ t)\ ts & \rightarrow \textbf{let}\ x = u\ \textbf{in}\ t\ ts & \\
\textbf{case}\ (\textbf{let}\ x = u\ \textbf{in}\ t)\ \textbf{of}\ alts & \rightarrow \textbf{let}\ x = u\ \textbf{in case}\ t\ \textbf{of}\ alts &
\end{array}
$$

The above rules are applied as much as possible to each application argument before it is transformed. If we call this transformation $\mathcal{E}$, then the new rule expressing **let** extraction during transformation is given by

$$
\begin{array}{rl}
\mathcal{T}\ \sigma\ (t\ (ts +\!\!+ \langle v \rangle))\ as & = \textbf{let}\ x_1 = \mathcal{T}\ \sigma\ u_1\ \textbf{in}\ \ldots \textbf{let}\ x_n = \mathcal{T}\ \sigma\ u_n\ \textbf{in}\ \mathcal{T}\ \sigma\ ts\ (u : as) \\
& \quad \textbf{where let}\ x_1 = u_1\ \textbf{in}\ \ldots \textbf{let}\ x_n = u_n\ \textbf{in}\ u = \mathcal{E}\ v
\end{array}
$$

Note that **let** extraction cannot be done in advance, it must be done during transformation. This is because substitution operations can cause **lets** to appear in the argument position of applicative terms, where there were previously none.

These extensions to the transformation algorithm have been found to improve the compactness of the code generated by our deforestation implementation, and also improve its efficiency.

5.8.4 Loop merging

It is common after deforestation of a module to have several new functions that are essentially identical modulo renaming. This can occur, for example, when the program has several calls to the function *map*, none of which result in the elimination of an intermediate

list. The knot-tying process will extract a new function in each case that is identical to *map*. It is beneficial in these cases to merge the definitions into one, updating all calls to the old functions to point to the new one.

This is achieved after deforesting the module, by comparing the definitions of all the newly created functions to find duplicates, and merging them as described above. Although we have not implemented this technique as yet, we believe it would further reduce the code-size penalty for deforestation in certain cases.

5.8.5 Improving the performance of the knot-tyer

The knot-tying algorithm as presented is quadratic in the number of labels deep the search must progress before a loop is found, because each new label is compared with all previous labels. A full label comparison is linear in the size of the labels being compared. We can reduce the complexity of the average case by improving the method by which labels are compared. The standard approach to comparing many treelike data structures is to apply a hash function to each tree, and compare the hash values, which is a constant time operation. If the hash values are identical, then a full comparison of the trees is required to establish equivalence.

The approach we take is to apply a hash function to each label placed in the history. As a new label is found, it is hashed and the value compared to the hash values of each of the previous labels. When a match is found, the labels are compared as before. In the worst case, all the hash values will be identical and the complexity is unchanged. However, in the average case, all the comparisons will be constant time hash-value comparisons until a match is found.

Implementing this optimisation has a dramatic effect on the efficiency of the knot-tying algorithm in expressions where many label comparisons are required to find the loop.

5.9 Avoiding Name-Capture

Ensuring that name capture can never occur presents real problems for an implementation of the deforestation algorithm. The goal is to make sure that no variable becomes acci-

dentally 'captured' by a binding of the same name. For example, suppose the following subexpression occurs during transformation:

$$(\textbf{let } x = t \textbf{ in } u)\ x$$

Deforestation transforms this into

$$\textbf{let } x = t \textbf{ in } (u\ x)$$

where the variable x has moved inside the **let** binding of the same name. This is clearly an invalid transformation, and cannot be allowed to take place.

We have so far circumvented the problem of name capture in our treatment of deforestation by making the assumption that all variable names are unique. This approach is fine for a theoretical treatment, but ensuring that the assumption holds at all times is problematic for an implementation. It is not enough to rename all the variables in the original program to be distinct: deforestation replaces function calls with their definitions, so if a function is unfolded multiple times several bindings that refer to variables of the same name can arise.

Several methods for avoiding name capture will now be described, leading to our final solution which is implemented in the Glasgow Haskell Compiler.

5.9.1 Unique Name Supplies

The simplest programming technique to avoid name capture involves the use of a *name supply*. The name supply is an object that represents an infinite sequence of unique variable names. The name supply must be used in a single-threaded manner throughout the program (i.e. it must never be duplicated) so that the variables generated are always guaranteed to be unique.

Transformation then proceeds as follows. Begin with an expression in which no two bindings refer to variables of the same name. Each time an expression is duplicated, replace

all the bound variables within it with new unique names. Thus, name capture can never occur.

A simple name supply can be implemented as an integer that is increased by one for each fresh variable that is requested. For example, if we are representing variables as strings:

```
type NameSupply          =  Integer

initialNameSupply        :: NameSupply
initialNameSupply        =  0

freshVariable            :: NameSupply -> (String,NameSupply)
freshVariable ns         =  ("_" ++ show ns, ns + 1)
```

Fresh variables are of the form _n where n is an integer, different for each new variable.

A minor problem with this implementation is that all variables become _1, _2 etc. and the original mnemonic names are lost. This could be avoided by keeping track of the original names and including them after the "_".

The real problem with this technique becomes clear when we try to adapt the deforestation algorithm to include the name supply. Suppose that we implement the deforestation transformation with a function `trans` that has the following type:

```
trans   ::   Expression -> Expression
```

In order to make the straightforward translation from the specification of the algorithm, $\mathcal{T}$, into a function of this type, we must use a language with lazy evaluation. The reason is that the transformation sequence is likely to be infinite, but we would like to be able to examine the result and prune it to produce a finite expression using a separate knot-tying process.

Unfortunately, when a name supply is introduced, the fragile balance of laziness is upset. The type of the `trans` function would become:

```
trans   ::   Expression -> NameSupply -> (Expression,NameSupply)
```

In order to return the name supply, the `trans` function must have completed transformation of its expression argument. Because any call to `trans` is likely to generate an infinite expression as output, which will require an infinite number of fresh variables, the name supply will never be returned. To see why this is a problem, consider the following fragment of a hypothetical implementation of `trans`:

```
trans ns (Let (x,t) u)
        =  (Let (x,t') u',ns'')
        where  (t',ns' ) = trans ns  t
               (u',ns'') = trans ns' u
```

To transform a **let** expression, we recursively transform each of the branches. Transformation of the body of the **let**, `u`, requires the name supply `ns'` returned by transformation of the right-hand side of the **let** binding, `t`. Suppose that transformation of `t` yields an infinite expression, then `ns'` will be $\perp$, and the second recursive call to `trans` will be `trans` $\perp$ `u`. As soon as this call to `trans` requires the name supply, the result will be $\perp$, and hence the result of the transformation as a whole will be an expression in which some branches are $\perp$.

To avoid this problem, we could merge the knot-tying process with the transformation such that the result of all calls to `trans` were finite. In fact, this would be essential if we were working in a language with strict evaluation. However, we would like to retain the modularity provided by splitting the algorithm in this way, so it is desirable to find another way to avoid name capture without introducing errant strictness into the implementation.

5.9.2 Debruijn Numbers

The Debruijn numbering method is an alternative naming scheme for the lambda calculus that inherently prevents name capture from occurring. The idea is simple: replace each variable name with a number obtained by counting the intervening bindings between the variable occurrence and the binding that it refers to. For example, the expression

$\lambda f.\,\lambda g.\,\lambda x.\,f\ (g\ x)$ turns into $\lambda.\lambda.\lambda.2\ (1\ 0)$. The scheme can be extended to include other constructs such as **case** and **let** in the natural way.

The Debruijn numbering scheme is a simple and elegant way to avoid name capture during transformation, but it presents its own problems. The main disadvantage is that substitution becomes much more complex: the term being substituted must be kept consistent with the term in which substitution is taking place. That is, the free variables of the term being substituted must be increased by the number of bindings between the substitution site and the original instance of the term. In addition, if the act of substitution removes a binding (a common occurrence during beta- or case-reduction), then the free variables of any subterms must be adjusted accordingly. In fact, with these additional manipulations it emerges that using the Debruijn numbering method for our purposes creates as many problems as it solves. For this reason, it was rejected as a method for avoiding name capture.

5.9.3 Splitting Name Supplies

A splitting name supply is a modification to the original name supply idea which copes neatly with the strictness issues introduced. The idea is to treat a name supply as a single threaded object with two operations. Firstly, a name supply can produce a new unique name. Secondly, a name supply can split itself into two new name supply objects. An easy way to understand a splitting name supply is to associate a (possibly infinite) set with each supply, from which unique names are drawn. A splitting operation on the name supply also splits this set, yielding two disjoint sets. Each split always yields disjoint name supplies, so the new supplies can be safely used by different parts of the transformation with no danger of name clashes. The behaviour of a splitting name supply can be specified thus: the names produced by any individual supply are different from the names produced by all supplies that have a common ancestor.

To simplify matters, we can restrict the number of names produced by any given supply to one. This is not a serious restriction since we can perform a number of splits to generate several unique names from one supply. A simple splitting name supply of this type can be implemented by representing names as lists of booleans:

```
type NameSupply          =  [ Bool ]

initialNameSupply        :: NameSupply
initialNameSupply        =  [ ]

freshVariable            :: NameSupply -> String
freshVariable ns         =  showName ns

split                    :: NameSupply -> (NameSupply,NameSupply)
split ns                 =  (False:ns, True:ns)
```

This implementation simulates a binary tree: the splitting operation prepends a new item onto the current supply, `False` for the left branch and `True` for the right branch. The resulting names can be interpreted as binary numbers for the purpose of printing names.

The small excerpt of code from the transformation process in Section 5.9.1 now becomes:

```
trans :: NameSupply -> Expression -> Expression

trans ns (Let (x,t) u)
        =  Let (x,t') u'
        where  t'         = trans ns1 t
               u'         = trans ns2 u
               (ns1,ns2)  = split ns
```

Because of the name supply split, the dependency of the second recursive call to `trans` on the first has been removed.

One disadvantage of this technique is that the names generated by a splittable name supply can become large, because splits are performed often and not all the names are used. The names do not often map easily into strings, because even treating the names as binary numbers we found can yield extremely large numbers. One solution is to rename the program after transformation, another is to use imperative techniques in the implementation of the name supply as we shall see in the next Section.

5.9.4 Monadic Name Supplies

Since a name supply must be used in a single-threaded manner, it makes sense to force this through the use of a monad. The use of monads [Mog89], a concept from category theory, is becoming popular in functional programming [Wad90a, Wad92] because they allow a wide variety of programming paradigms to be expressed within a single framework.

A monad encapsulates a particular kind of computation. The kind we are interested in here is the single-threaded use of an object, that is the object is to be treated as *state*. The *state monad* provides exactly the capabilities we require: the state (in this case the name supply) is held in the monad, and can only be manipulated by monadic operations. The monad itself enforces single-threaded access to the state. The following is a sample implementation of a monadic name supply, using the basic name supply type and functions given in Section 5.9.1:

```
type NS a       = NameSupply -> (NameSupply,a)

returnNS        :: a -> NS a
returnNS a ns   =  (a,ns)

thenNS          :: NS a -> (a -> NS b) -> NS b
thenNS a k ns   =  case a ns of
                        (a',ns') -> k a' ns'

getNameNS       :: NS String
getNameNS       =  freshVariable
```

Using the same techniques as the IO system of the Glasgow Haskell Compiler (which is based on monadic IO [PW93]) we can represent the name supply as a global integer variable, which yields an increase in performance.

However, as we explained before, a single name supply is not good enough for the Deforestation algorithm; we need a splittable name supply in order to retain sufficient laziness for the algorithm to terminate. Fortunately, the splittable name supply can also be embodied in a monad. The idea is that the bind (or *then*) operation of the monad also performs

a split operation on the name supply, and a new name supply is passed to each argument of the bind. For example:

```
type NS a       = NameSupply -> a

returnNS        :: a -> NS a
returnNS a ns   =  a

thenNS          :: NS a -> (a -> NS b) -> NS b
thenNS a k ns   =  k (a ns1) ns2
                   where (ns1, ns2) = split ns

getNameNS       :: NS String
getNameNS ns    =  freshVariable ns
```

There is no danger of ever duplicating a name supply, since multiple computations must be joined by `thenNS` and thus receive different name supplies as a result of the call to `split` in the definition of `thenNS`. There is also no danger of two calls to `getNameNS` ever returning the same name, since the two calls must be separated by a `thenNS` and therefore receive different name supplies.

With some clever programming we can still represent the name supply as a global integer variable. The idea (due to Augustsson, Rittri and Synek [ARS94], and described in some detail by Launchbury and Peyton Jones [LJ95]) is to implement the underlying name supply as follows:

```
data NameSupply = Splittable Name NameSupply NameSupply

split (Splittable n s1 s2) = (s1, s2)

freshVariable (Splittable n s1 s2) = n

initialNameSupply = mk_supply
```

```
        where
                mk_supply
                      = unsafeInterleavePrimIO (
                        _ccall_ genName  `thenPrimIO` \ n ->
                        mk_supply        `thenPrimIO` \ s1 ->
                        mk_supply        `thenPrimIO` \ s2 ->
                        returnPrimIO (MkSplitUniqSupply n s1 s2)
                        )
```

This code (in non-standard Glasgow Haskell) calls a C function which increments a global variable each time a new name supply is generated (i.e. each time the supply is split). To change this such that new names are generated only when needed, it is possible to wrap the C call itself in a call to `unsafeInterleavePrimIO` so that the call will only be made when the name is required. This code is only referentially transparent because we are using it within the monad; it is possible by nefarious means to use the Glasgow Haskell extensions to write code that is not referentially transparent.

Another advantage of placing the name supply under monadic control is that our code is likely to be written in a monadic style anyway—the book-keeping required for the name supply will therefore have very little impact on the rest of the program, which is an improvement over the rather cluttered style of the example in Section 5.9.1.

Chapter 6

Results and Analysis

In this chapter, we examine the effect of applying the deforestation implementation described in the previous chapter to several small functional programs. In each case, we show that by examining the source program it is possible to predict which intermediate data structures will be eliminated by deforestation. We also give some evidence that these data structures have been eliminated, by giving the output from the deforestation transformation, and some measurements of memory usage and execution time made by running the transformed program. We also compare the code size of the deforested program against the original to assess the impact of unfolding functions during deforestation.

The examples programs are chosen to illustrate several situations in which deforestation can be effective.

6.1 Description of Measurements

We compiled each example program twice using our modified version of the Glasgow Haskell Compiler derived from version 0.23. The first compilation was made with the default set of optimisations, by requesting normal optimisation from the compiler; the alternatives are no optimisation and extra optimisation. The second compilation was made with additional deforestation and full-laziness passes inserted at appropriate points in the compilation cycle.

The set of normal Core-to-Core optimisations includes the following:

- General simplification. This consists of a large number of small local transformations designed to improve the overall quality of the code [San95c]. Examples are β-reduction and **case**-reduction. The simplification pass is typically invoked at several stages during the compilation cycle, to clean up after each major optimisation pass.

- Specialisation. This optimisation generates versions of overloaded functions instantiated at a particular type.

- Strictness Analysis. The strictness analyser in the Glasgow Haskell Compiler [PJ93] is primarily useful for removing the repeated boxing and unboxing of primitive data types (such as integers and characters) in recursive functions. It also applies to other product (single-constructor) datatypes, but its usefulness for sum (multi-constructor) datatypes is limited, due to the difficulty in utilising strictness information for these types.

- Full-laziness. The full-laziness pass extracts maximally free expressions by a technique known as **let**-floating [San95c]. It is required after deforestation as described in Section 4.2.2.

- The Static Argument Transformation, which is the opposite of lambda-lifting. This optimisation transforms a function with a number of arguments which are identical in each recursive call into a local function definition with these arguments as free variables. These functions are implemented more efficiently by the Glasgow Haskell Compiler than their lambda-lifted equivalents, as the Spineless Tagless G-Machine (the abstract machine on which GHC is based) doesn't require lambda-lifted input. Functions which respond to the static argument transformation occur frequently in the output from deforestation.

The exact order in which these passes are performed is critical, due to subtle interactions between different optimisations. In the current version of the Glasgow Haskell Compiler, they are performed in the following order:

1. Specialisation
2. Simplification
3. Full-laziness
4. Strictness Analysis
5. Simplification
6. Static Argument Transformation

When we insert deforestation into the compilation cycle, the order of passes is as follows:

1. Specialisation
2. Simplification
3. Full-laziness
4. Deforestation
5. Simplification
6. Full-laziness
7. Strictness Analysis
8. Simplification
9. Static Argument Transformation

The full-laziness optimisation is actually performed twice when we have deforestation, once before the deforestation pass and once after. The reason for this is that applying deforestation to the program before full-laziness can remove opportunities for full-laziness, which decreases the efficiency of the deforested program as compared to the version compiled without deforestation.

The measurements were made on an unloaded SparcStation 10. For each program we measured the wall-clock execution time, the total heap allocation, and the code-size of the

object program. The most useful of these is ultimately the execution time, since the goal of any optimisation strategy is to reduce this figure. By measuring the actual elapsed time we implicitly take into account several factors:

- Mutator time, which is the time spent in actual computation by the program. This includes heap allocation, updates, and all other elements of the execution of lazy functional programs. The mutator time is reduced by deforestation because fewer references to the heap are made, both for allocation and examination of existing data. Reducing heap accesses has a significant impact on modern architectures where memory references impose a larger penalty on execution time than purely register-based computation.

- Garbage collection time. The number of garbage collections performed during execution is dependent not only on the amount of heap allocation, but also on the amount of live data present in the heap at garbage collection time. The time spent in each garbage collection is typically dependent only on the amount of live data, since the garbage collector must traverse the tree of reachable data on each collection to determine what to keep. Deforestation tends to reduce the total heap allocation made by the program, but the data it eliminates is short-lived in the case of simple list-processing computations, which constitute the majority of situations in which deforestation is effective.

- Input/Output, and other system operations such as paging. Because we are measuring the execution time of both versions of the program with a fixed heap size, which fits in the real memory of the hardware, paging is not a factor. Deforestation can of course have no effect on the time spent performing input/output.

The relationship between the amount of computation performed by the program (where computation can be defined as the actual number of instructions executed) and the wall-clock execution time is tenuous at best, due to memory cache effects [HBH93]. It has been argued that to provide realistic comparisons of functional programs one should first find the optimal heap size for the program, to maximise the cache hits and minimise cache misses caused by interaction between heap and stack accesses. We did not do this in

our measurements, due to the large number of timed runs required to find the optimal configuration.

The elapsed execution time for each program was measured by taking the best result from several runs. The total heap allocation is provided by the Glasgow Haskell run-time system on request. The code size for each version is actually the size of the compiled object produced, before linking takes place—this is to eliminate the large constant factor of prelude/library code linked in to the final program.

We did not include the time taken to compile each program in the results, since the compilation time was not affected significantly by the inclusion of deforestation in the compilation process.

6.2 Queens

Our first example is the traditional 10-queens problem: find the number of ways in which 10 queens can be placed on a $10x10$ chessboard such that no queen is on the same row, column, or diagonal as any other queen.

The program (given in Figure 6.1) is written in a *listful* style: it makes heavy use of lists and standard list operators as convenient programming tools to express the problem.

The algorithm itself is implemented using backtracking, by means of a common lazy functional programming technique known as the *list of solutions* technique [BW88]. Backtracking is achieved by defining the list of all solutions to the problem, recursively finding all the solutions to n-queens starting with $n = 1$, and expanding each solution to $n + 1$ by appending all the possible positions for another queen and filtering out all the resulting positions which aren't valid. Because the list of solutions is evaluated lazily, a demand for the first element of the list will search the solution space depth-first, backtracking until it finds a valid solution. The backtracking aspect is not important here, however, since we require the total number of solutions which necessitates evaluation of the entire list.

There are several intermediate lists which can be removed by deforestation, without any additional annotations to the program.

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

queens :: Int -> [[Int]]
queens 0 = [[]]
queens m = [ p ++ [n] | p <- queens (m-1),
                        n <- [1..10],
                        safe p n ]

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

Figure 6.1: Haskell code for n-queens

- The enumerated list `[1..10]` in the `queens` function is consumed by a list comprehension, so this will be removed automatically be deforestation.
- The enumerated list `[1..]` in the `safe` function is consumed by the standard prelude list processing function `zip`, and this can be eliminated.
- The list of pairs produced by `zip` is consumed by a list comprehension, so this can be removed.
- The list of booleans produced by the list comprehension in the `safe` function is consumed by the prelude function `and`, this can also be removed.

The intermediate Core program generated by the Glasgow Haskell compiler, for the 10-queens program is given in Appendix A.1, along with the Core output from the Deforestation transformation. As can be seen by examining the code, the four intermediate lists above have all been eliminated.

The code shown is the direct output from deforestation, before any further simplifications

	Code size (bytes)	Heap Allocations (bytes)	Execution time (s)
Before	unknown	140,522,924	16.51
After	unknown	20,337,924	6.16

Figure 6.2: Deforestation results for Queens

have been applied. There are several optimisations that will be applied by the compiler post-deforestation. These include:

- The removal of multiple unboxing of integer variables. For example, the variable `i` in the deforested `safe` function. This is performed by the general simplifier.
- Strictness analysis will remove the boxing and unboxing of integer arguments to functions. In the queens example, both `safe` and the local recursive functions in `queens` recurse over integer variables, and strictness analysis can have a profound effect on the efficiency of these functions.
- The static argument transformation is useful in removing the static arguments `n` and `m` to the local recursive function `f` in `safe`, and the argument `j` to `h` in `queens`.

6.2.1 Results

The results from applying deforestation to 10-queens (we performed the tests with $n = 10$) are shown in Figure 6.2. Looking at the results, we see a large reduction in total memory allocations made by the program: the heap usage of the transformed program is roughly one seventh of the original. This is reflected in the time taken to execute the program, the transformed program executes in about 40% of the time of the original. In general, there is no direct relationship between reduction in heap usage and increase in efficiency; this depends on how much time is spent by the program doing non-heap-intensive computation.

One intermediate list in the queens program that is not eliminable by our deforestation technique is the list generated and consumed by the `queens` function itself. A technique

termed *Worker-wrapper deforestation*, developed by Gill as part of his *foldr/build* deforestation scheme, is capable of removing this type of intermediate structure. Adapting this technique to our deforestation scheme is planned as future work.

This example is somewhat pathological, in that we can never expect to see such dramatic improvements in larger programs, unless the program spends a great deal of time working with intermediate structures which can be eliminated by deforestation, which is unlikely. However, the example serves to illustrate the upper bound on the improvements that can be achieved, and the relation of memory allocation behaviour to overall execution time.

6.3 Life

In this section we examine a Haskell implementation of Conway's Life written by John Launchbury, taken from the nofib benchmark suite [Par92]. The game of Life is very simple: played on a fixed grid, each cell begins as either alive or dead. Each successive generation is evolved from the previous using one simple rule: if a cell has 2 or 3 live neighbours it is alive in the next generation, otherwise it is dead.

This implementation of life has a fixed starting position, and evolves the pattern until a stable configuration is reached. Each generation is printed on the standard output stream with live cells represented by `o`, and dead cells by a blank space.

The program (given in Figure 6.3) is written in a listful style, as before. However, this time we will have to make use of explicit programmer annotations to obtain the maximum benefit from deforestation. The reason annotations are required is that a trade-off between code size and speed is involved, and the compiler leaves these decisions up to the programmer.

6.3.1 Deforesting Life

By examining the code (and the definitions of the prelude functions used therein), we can see that the following intermediate structures will be removed by deforestation:

- In function `disp`:

```
module Main where

main inp = [AppendChan stdout ("\FF" ++ life 30 start)]

start :: [[Int]]
start = [[],[],[],[],[],[],[],[],[],[],[],[],[],[],
         [0,0,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,0]]

gen n board = map row (shift (copy n 0) board)
  where
    row (last,this,next)
      = zipWith3 elt (shift 0 last) (shift 0 this) (shift 0 next)
      where
      elt (a,b,c) (d,e,f) (g,h,i) | tot < 2 || tot > 3 = 0
                                  | tot == 3           = 1
                                  | otherwise          = e
        where tot = a+b+c+d+f+g+h+i

shiftr x xs = append [x] (init xs)
shiftl x xs = append (tail xs) [x]
shift x xs = zip3 (shiftr x xs) xs (shiftl x xs)

copy :: Int -> a -> [a]
copy 0 x = []
copy n x = x : copy (n-1) x

disp (gen,xss) =
   gen ++ "\n\n" ++ (foldr (glue "\n") "" . map (concat . map star)) xss

star 0 = "  "
star 1 = " o"
glue s xs ys = xs ++ s ++ ys

life n = foldr1 (glue (copy (n+2) '\VT')) . map disp
          . zip (map show [0..]) . limit . iterate (gen n) . initial n

initial n xss = take n (map (take n.(++(copy n 0))) xss ++
                  copy n (copy n 0))

limit (x:y:xs) | x==y      = [x]
               | otherwise = x : limit (y:xs)
```

Figure 6.3: Haskell implementation of Conway's Life

  - The list between `concat` and the second call to `map`
  - The list between `foldr` and the first call to `map`
  - The constant string `"\n\n"` (it is consumed by `++`)

- In function `life`:
  - The list between `foldr1` and `map`
  - The list between `map disp` and `zip`
  - The list between `zip` and `map show [0..]`
  - The list between `map` and `[0..]`

- In function `initial`:
  - The list between the first call of `take` and `map`
  - The list between the second call of `take` and `++`

Although we can remove a large number of intermediate lists without any annotations at all, none of these lists are part of the *inner loop* of the program (the part of the program that performs the majority of computation during a run). The inner loop in this case is the function `gen`, which computes the next generation of the Life universe. We can see the somewhat disappointing results from performing deforestation on this program without annotations in Figure 6.4 (under Ver.1 with and without deforestation).

Looking again at the program, we can see a number of opportunities for deforestation which are being missed, because the intermediate structure in question arises as the result of a top-level function. For example, `shiftr` and `shiftl` both produce deforestable lists, which are consumed by the `shift` function. If `shiftr` and `shiftl` were inlined at the call-site, the calls to `append` would be fused with the call to `zip3`, removing the intermediate lists.

There are two ways in which top-level functions may be inlined at the call-site:

- The compiler does this automatically, provided:
  1. The function is not recursive,
  2. it is called once only,

3. inlining it cannot result in duplication of work, and
4. the function is not exported from the module.

- By providing a `DEFOREST` annotation for a function, we instruct the deforestation phase of the compiler to unfold the definition at each place where it is called. This option should be used for functions which are called multiple times, but where we believe the benefit of deforestation will outweigh the increase in code-size that results from unfolding the definition several times.

In the Life program as given above, all the functions are by default exported, so none of them satisfy the criteria for automatic inlining. However, since this is a single module program we can safely change the module header to

```
module Main(main) where
```

so that only the function `main` is exported. This gives the compiler the freedom to inline the following functions: `start`, `gen`, `shiftr`, `shiftl`, `disp`, `initial`, and `star`. The following intermediate structures are now removed by deforestation:

- The lists generated by `shiftr` and `shiftl`
- The pairs in the list between `map disp` and `zip` in `life`.

The results from applying deforestation to the modified program are given in Figure 6.4 (under Ver.2 with and without deforestation). We have now removed some important lists: the lists generated by `shiftr` and `shiftl` were in the inner loop.

We can do still better than this: by annotating some of the functions for deforestation we can remove all the important intermediate lists and greatly improve the space consumption of the program.

Firstly, we can annotate `limit`, and thus remove two more lists from the function `life`:

- The list generated by `iterate` and consumed by `limit`

- The list generated by `limit` and consumed by `zip`

Next, we can annotate `shift`, the function most heavily used in calculating the next generation, and remove the following structures:

- In `row`, the lists of triples produced by all three calls to `shift` and consumed by `zipWith3 elt`.
- In `gen`, the list of triples produced by `shift` and consumed by `map row`.

Finally, by annotating `glue` and `copy`, we can remove the following lists:

- The lists generated by `disp` and `copy`, and consumed by `glue` in `life`
- The lists generated by `copy` in `initial`.

With these annotations alone, the deforestation process can have a more dramatic effect on the total heap allocation made by the Life program during execution (See Figure 6.4, Ver.3). We do not provide the Core output from deforestation here as it is too large, but we have verified that the structures which we have indicated above as removable are in fact removed. In particular, the resulting program contains no references to 3-tuples at all, although the original program uses them extensively during the calculation of each generation.

It is worth mentioning that the Life program demonstrates some of the advantages of our higher order deforestation algorithm over short-cut deforestation schemes such as foldr/build [GLJ93]. Short-cut deforestation cannot remove all the intermediate lists in functions which consume multiple lists simultaneously, such as `zip`. It also requires the programmer to rewrite recursive list-processing functions in terms of the combinators `foldr` and `build`; this technique would need to be used to remove the lists involving `copy` and `limit` in the Life program, which we eliminated by simply annotating these functions for deforestation.

	Code size (bytes)	Heap Allocations (bytes)	Run time (s)
Ver.1 (no deforestation)	303104	254,647,484	78
Ver.1 (w/ deforestation)	311296	238,684,128	77
Ver.2 (no deforestation)	278528	253,555,692	76
Ver.2 (w/ deforestation)	286720	201,633,604	73
Ver.3 (w/ deforestation)	442368	157,128,460	61

Figure 6.4: Deforestation results for Life

6.3.2 Results

The final results from applying deforestation to the annotated Life program are given in Figure 6.4. The heap usage of the program is about 60% of the original and the execution time has decreased by about 20%.

The program still uses a large amount of heap. This is due to two factors:

- The list of lists produced for each generation is built from closures which contain a large number of free variables. Consider for example, the closure for each new cell, which must contain at least eight free variables, one for each neighbour. These closures are evaluated immediately as the generation is printed to the output device. A powerful strictness analyser would be able to detect the strictness in the program and avoid creating these large closures, although the strictness analyser in the Glasgow Haskell Compiler is not currently able to detect strictness in lists.
- The program prints out each generation to the standard output, after converting it to printable form. Converting the internal representation of the generation to a string consumes a large amount of heap, and printing it takes time. We cannot expect deforestation to remove the intermediate representation, because the internal form is used to determine when a stable state has been reached.

We also notice that the code size of Version 3 is much higher than the others. This is due to a code explosion created by the deforester while it was deforesting `gen` (which involved unfolding `shift` four times). This is partly to be expected, since `shift` is by no means a

trivial function, but the code size is still larger than necessary (inspection of the output from deforestation revealed some duplicated code). It seems that further improvements to the algorithm are required to reduce code explosions of this kind, along the lines of the enhancements made in Chapter 5 (back loops and the boring expression transformation).

6.4 Pattern Matching

To illustrate the usefulness of deforestation for eliminating data structures other than lists and tuples, we chose a program for performing simple pattern matching on strings.

The pattern matcher (Figure 6.5) is an implementation of Unix^{TM} filename matching, in which a pattern is a sequence of pattern elements. A pattern element is of one of the following forms:

- $[c_1 \ldots c_n]$, where the c_i are characters, matches a single instance of any one of the c_i.
- ? matches any single character.
- $*$ matches a sequence of zero or more characters.

Single character exact matches can be represented by $[c]$, where c is the character to match.

The pattern matching function `match` takes a pattern and a string and returns `True` if the pattern matches the string, and `False` otherwise.

By deforesting an application of the pattern matcher to a known pattern, the intermediate structure representing the pattern will be removed and a version of the pattern matcher specialised to the given pattern will be generated. In this guise, the deforestation algorithm is acting as a restricted form of partial evaluation, albeit one that always terminates and will always eliminate the static pattern argument.

This is also an example where eliminating a small intermediate data structure can have profound effects on the execution time of the program. This is because we are eliminating a structure that is repeatedly deconstructed and examined. Deforestation performs this deconstruction work once and for all at compile time.

```
module Main (main) where

data Pat
        = PatChars [Char]
        | PatAny
        | PatStar

match [] []         = True
match [] (c:cs)     = False
match ps@(p:ps') cs = case p of
        PatChars chars ->
            case cs of
                []      -> False
                (c:cs') -> c `elem` chars && match ps' cs'
        PatAny ->
            case cs of
                []      -> False
                (c:cs') -> match ps' cs'
        PatStar ->
            case cs of
                []      -> match ps' []
                (c:cs') -> if match ps' cs then True else match ps cs'

main =  readFile "/usr/dict/words" abort  $ \datafile ->
        print (length (filter (match pat) (lines datafile)))

pat =   [PatStar, PatChars "abc", PatStar, PatChars "def", PatStar,
         PatChars "ghi" ]
```

Figure 6.5: Haskell implementation of Unix filename matching

For our experiments, we chose the sufficiently complicated pattern $*[abc] * [def] * [ghi]*$, and counted the number of matching lines in a dictionary containing some 200,000 words.

6.4.1 Results

In order to deforest the application of `match` to the static pattern `pat`, we must annotate both of these definitions as deforestable. Therefore, we added the following two annotations to the program:

```
{-# DEFOREST match #-}
{-# DEFOREST pat   #-}
```

With these annotations, the deforestation algorithm transforms the call `match pat` into a specialised matcher for `pat`.

After simplifications, the new matching function consists of four mutually recursive functions, one for each occurrence of $*$ in the pattern. Recursive functions are generated because one of the calls to `match` in the case for `PatStar` is passed the input pattern, so termination in this case is via the knot-tying process discovering a renaming, and generating a new recursive definition. All the other recursive calls in `match` are passed a subexpression of the original pattern, so they are simply expanded until the end of the pattern is reached.

It is also worth noting that because the `if` expression in the case for `PatStar` in `match` is equivalent to a boolean **case** expression with a call to `match` as its selector, this expression is not in treeless form:

```
case match ps' cs of
    True -> True
    False -> match ps cs'
```

The automatic conversion process extracts this call to `match` making its result, and hence the whole `if` expression, residual:

	Code size (bytes)	Heap Allocations (bytes)	Run time (s)
Before	8640	33,470,720	6.6
After	7888	33,470,796	5.8

Figure 6.6: Deforestation results for Match

```
let z = match ps' cs in
case z of
    True -> True
    False -> match ps cs'
```

Because of the treeless form restriction we are prevented from generating the optimal specialised pattern matcher, although the transformer will eliminate the static pattern data structure.

The results are given in Figure 6.6. We can see that the execution time of the program is improved by 12%, accompanied by a small reduction in code size and a miniscule increase in heap usage. We attribute the relatively small improvement in efficiency to the slow character input/output in Haskell.

This example demonstrates that the advantages of deforestation are not always through the removal of as much intermediate structure as possible.

Some similarity can be drawn between this example and the work of Sørenson, Glück and Jones [SGJ94], who extend deforestation in order to be able to derive optimal Knuth-Morris-Pratt [KMP77] specialised pattern matchers from a general matching algorithm.

Chapter 7

Conclusion

7.1 Summary

7.1.1 Deforestation Algorithm

In Chapter 2 we presented a deforestation algorithm for a higher-order language. This is a significant generalisation of Wadler's first-order deforestation scheme, not only because it applies to higher-order programs and is therefore applicable to the majority of modern programming languages based on the lambda calculus, but also because it incorporates a mechanism whereby residual data structures can be present in the program to be deforested. This enables any function to be presented as input to the deforestation algorithm, given a suitable translation of the function to treeless form.

One of our goals was to formulate higher-order deforestation as a *transparent* optimisation strategy. We had to ensure that if the optimisation is not applicable everywhere in the program (as indeed it isn't; we cannot expect to remove all intermediate data structures from the program) then the programmer is aware, by means of a syntactic specification, exactly where deforestation *is* applicable. This was achieved by introducing the **let** construct as a means of indicating residual data structures, those which will be left in place by the deforestation process.

The **let** construct is also used to ensure that expressions are not duplicated by deforesta-

tion, by protecting non-linear bindings. To further ensure that a deforested program is no less efficient than the original, we employed the full-laziness transformation which extracts expressions by rebinding them, again with **let**. The **let** construct turns out to be a universally useful way to indicate residual structures, both to ensure termination of the algorithm and to prevent a loss of efficiency in the transformed program.

Having formulated the original algorithm we recognised a deficiency, namely that treeless form did not correspond exactly to the normal form of the language, and consequently the algorithm had to insert **lets** in the output to keep it in treeless form. This is counter-intuitive, and overly restrictive in most cases. We wanted to generalise treeless form so that it corresponded to normal form, modulo calls to the recursive functions that we were using for the transformation, since extra restrictions must be placed on these to ensure termination.

To this end, we used principles from logic to formulate a new deforestation algorithm. The logic of sequent calculus has an important property, that of cut elimination. Cut elimination is a reduction process for proof-trees that renders the proof in normal form by eliminating applications of one rule: the cut rule. If the logic is translated into a programming language, we have a language that is in normal form save for a single construct, which we call the **cut** construct (identical in meaning to **let**, but we used **cut** to avoid confusion). This situation is exactly what is required for deforestation, which also reduces terms to normal form (albeit using recursive functions, which are not covered by cut-elimination). We merged the languages of natural deduction and sequent calculus to obtain a new normal-form language, and defined cut-elimination for this language, and proved that the algorithm terminated. We also defined a simple first-order language with recursion and formulated a deforestation algorithm for it. By merging the two, we have a transparent deforestation algorithm with a sound logical basis and a generalised treeless form.

Unfortunately, our efforts to find a termination proof for this algorithm were unsuccessful, although we provide some convincing arguments for termination given certain restrictions on the input terms. The restrictions are less intrusive than those imposed on treeless form terms in the first algorithm, because arbitrary terms are allowed to the right of an application in many cases.

In Chapter 4 we tackled several issues closely related to the deforestation algorithm itself. Firstly, we examined the subject of automatic conversion of general terms to treeless form, and found that this can be performed optimally for both definitions of treeless form, and gave an algorithm for this process.

Secondly, the topic of linearity was covered and we showed that there are several situations in which the basic deforestation scheme could impair the efficiency of the program. To remedy the matter, we provide a set of rules which must be adhered to to ensure safety, and conjecture that they are sufficient.

Finally, we discussed the transparency property of deforestation, and demonstrated that it is possible to predetermine the effects of deforestation by examining the subject program. Because conversion to treeless form is a syntax-directed (and therefore transparent) process, the programmer can determine even by examining the original definitions of recursive functions to be used during deforestation which data structures will be removed and which will be residual.

7.1.2 Implementation

Having found a suitable deforestation algorithm we constructed a prototype in the Glasgow Haskell Compiler, to demonstrate that our algorithm fits into a real compilation setting.

We designed the implementation such that it can be used with varying amounts of programmer intervention. With no intervention at all, the algorithm will attempt to perform as much deforestation as possible without making any dangerous trade-offs between code-size and speed or memory usage. This was found to be a worthwhile approach, since there are a large number of functions in the standard Haskell prelude which operate on lists, as well as features such as list comprehensions which can all be used for deforestation transparently to the programmer. This involves a modest increase in code size as the functions involved are all unfolded at the call site, but the benefit in terms of reduced memory usage was found to outweigh the code bloat for the small functions involved.

If the programmer wishes to intervene in the deforestation process, then a greater amount of flexibility is available, and a more potent optimisation strategy is available. The major

technique here is the annotation of functions to be used during deforestation. By annotating a function, the programmer indicates that it should be unfolded at each call site and any intermediate structures which it creates or consumes should be eliminated if possible. This enables not only user-defined list processing functions to be applicable to intermediate list removal, but also functions that manipulate other datatypes such as trees.

We also devised a scheme whereby annotated function definitions can be communicated between Haskell modules at compilation time. This enables deforestation to be performed using function definitions from another module in the same program, or the standard prelude.

The deforestation algorithm of Chapter 2 was extended to use the Glasgow Haskell Core language. This required modifying the algorithm to transform expressions that involve arbitrarily nested **letrecs**. We also incorporated a version of the treeless form conversion algorithm described in Chapter 4. We also gave a full description of the knot-tying algorithm used.

In experimenting with earlier versions of the implementation we discovered several situations where the simple knot-tying algorithm could be improved in order to reduce the size of the generated code. These techniques, which are described in Chapter 5, turned out to be vitally important when we applied the algorithm to larger examples. We found that without the additional techniques described, the code size could explode dramatically, increasing both the compilation time and the size of the resulting binary. In extreme cases this could hamper the execution time of the program, cancelling the beneficial effects of deforestation, even though intermediate structures had been removed.

7.1.3 Results

In Chapter 6 we applied the prototype deforestation implementation to several non-trivial example programs. We found that the algorithm would remove all the intermediate data structures that were predicted as removable by the transparency property. The effects on the memory usage and run time of the program were sometimes dramatic, especially when the intermediate structure removed was part of the inner loop.

We also found that when the functions and expressions being deforested were complex,

the resulting code size was larger than expected. This prevented any larger examples from being deforested successfully, and we concluded that further techniques along the lines of those presented in Chapter 5 were necessary to reduce the size of the output from deforestation.

7.2 Future Research

7.2.1 Deforestation Algorithm

We intend to continue to generalise the deforestation algorithm. This will begin with further investigation of the termination properties of the improved algorithm of Chapter 3, hopefully leading to a termination proof. This would be a significant improvement, allowing our implementation to use the new algorithm and generalised definition of treeless form.

We also aim to investigate the linearity restrictions on deforestation, and develop a scheme where they may be relaxed under certain conditions. One possible avenue for research lies with linear type systems such as that of Turner/Wadler/Mossin [TWM95], which would assign linearity annotations to bound variables providing more information to deforestation about which expressions are in danger of being duplicated.

7.2.2 Relationship to `foldr/build` deforestation

In Section 1.4.2 we outlined the combinator deforestation of Gill/Launchbury/Peyton Jones [GLJ93, Gil95]. In this section we will give a detailed description of the relative power of the two systems, and propose a possible avenue for future research based on our findings.

We searched for a long time to find a clear relationship between our deforestation scheme and the `foldr/build` rule, believing deforestation to be a superset of this scheme. It appears that this is not the case, and there is no obvious theorem relating the two [GM95]. We can, however, informally describe the differences:

- `foldr/build` deforestation cannot express the removal of intermediate lists between a consumer and several producers. The reason for this is that the translation into

`foldr` only allows a single argument to be deforested using the `foldr/build` rule. The *zip* function is a classic example:

$$\begin{aligned} zip\ xs\ ys &= foldr\ f\ (\lambda\_.\,[])\ xs\ ys \\ &\quad \text{where}\quad f\ x\ g\ [] &&= [] \\ &\qquad\quad f\ x\ g\ (y : ys) &&= (x, y)\ :\ g\ ys \end{aligned}$$

Our deforestation scheme can remove intermediate lists between a multiple-list consumer and several producers without problems.

- It is cumbersome to express irregular list consumers using `foldr`. The term irregular is somewhat subjective, but in general it applies to functions that treat some elements of a list differently from others. Examples are `foldr1`, `tail`, and `halve` (the function which makes a new list consisting of every other element of its input). Deforestation, however, has no difficulty eliminating intermediate lists consumed by these functions.

- Some list consuming functions are naturally expressed using `foldl` rather than `foldr`. Examples include `sum` and `reverse`. The translation of `foldl` into `foldr` necessary for `foldr/build` introduces some inefficiency, which must be removed by additional transformations once the intermediate lists have been eliminated.

- In `foldr/build` a list producer is represented by simply abstracting an existing function over Cons and Nil, whereas in deforestation we must adhere to treeless form for our list producers. This facilitates the removal of some intermediate lists using `foldr/build` that are residual with respect to deforestation. The classic example is `reverse`, where treeless form forces the list produced to be residual:

$$\begin{aligned} reverse = \lambda xs.\,\lambda ys.\,&\textbf{case}\ xs\ \textbf{of} \\ &\quad Nil \rightarrow ys \\ &\quad Cons\ x\ xs \rightarrow \textbf{let}\ z = Cons\ x\ ys\ \textbf{in}\ reverse\ xs\ z \end{aligned}$$

The reverse function can instead be defined using `build` by simply abstracting over the Cons and Nil used to build the output list, incurring no penalty and allowing this list to be removed by `foldr/build` deforestation.

- Deforestation natively handles arbitrary data structures, whereas `foldr/build` is defined only over lists. We believe, however, that equivalent `foldr` and `build` definitions and the corresponding combination rules can be defined in a straightforward way for some other algebraic data structures. To our knowledge, this technique has not been implemented.

To summarise the important points: `foldr/build` has problems with irregular list consumers and functions that consume multiple lists, whereas deforestation has problems with some irregular list producers.

We plan to assess the possibility of a hybrid scheme that would draw on both deforestation methods, to remove more intermediate data structures than is possible with either system alone.

7.2.3 Implementation

The higher-order deforestation prototype is already distributed along with the Glasgow Haskell Compiler, but it requires a number of improvements before it can be used in production situations.

Code size

There is no inherent bound on the size of the code generated by the deforestation transformation, and this has turned out to be a major stumbling block. Although the system will always keep its promises, in that intermediate structures which are indicated as removable by the transparency property will be removed, the resulting code can be far larger than necessary in cases where several complex functions are being combined. This is illustrated in the Life example of Chapter 6, where we removed as much intermediate structure as possible through annotations, and ended up with a program that was nearly twice as large as the original. This code explosion also has a significant impact on compilation time and the perceived usability of the system.

We incorporated several additional techniques to address the code size issue in Chapter 5, in the form of extra rules for the transformation system and improvements to the knot-

tying process. Although these techniques were essential, it appears that more investigation into the problem is required to enable deforestation to be used in practice.

A possible approach is to define the maximum size of the resulting code given the initial function definitions and expression to be deforested. This is a known quantity since we have a termination proof for the algorithm, and we can take the worst case. However, we believe the worst case would be far too large to be of any use in practice. For example, the maximum depth of the result of transformation could be found by determining the maximum depth between labels and the number of possible permutations of label expressions. In most examples the algorithm terminates much earlier than this, but there is no guarantee. Techniques such as those in Chapter 5 help to reduce the number of permutations (extracting **lets** from applicative terms, for example), and it may be the case that more sophisticated techniques could reduce the maximum code size to an acceptable level.

Another minor code size issue is that of unfolding each deforestable function at the call site. This is wasteful in situations where no deforestation can be performed. Although this is impossible to determine in general, there are several specialised cases where a useless unfolding can be detected and prevented.

More automation

In order to perform more deforestation than simple intermediate list removal, the programmer must employ explicit annotations to direct the transformation. This situation could be improved, and it should be possible to derive safe annotations in several circumstances.

Functions that are useful for deforestation can be identified using some simple criteria: they have non-residual inputs and/or outputs, and operate recursively on inputs having recursive datatypes. It would also be necessary to impose a size restriction on functions to be annotated, to avoid dangerous code size/speed tradeoffs.

Some work has been performed for short-cut deforestation on identifying functions which can be translated automatically into applications of the combinators *foldr* and *build* [LS95]. Similar techniques can be applied in the deforestation setting, although no translation would be required, just an annotation of suitable functions.

Appendix A

Code Examples

A.1 Queens

The following code is the Glasgow Haskell Core intermediate output from the n-queens program given in Section 6.2.

```
safe = \ p n ->
   let {
     m = case (length Int p) of
           I# u# -> case (plusInt#! u# 1#) of
                       v# -> I# v#
   } in
   let { a =
     letrec {
       f = \ ijs ->
          case ijs of {
            Nil -> Nil
            (:) ij  ijs' ->
                case ij of {
                  Tup2 i j ->
                    let {
                      b =
                        case j of
                          I# j# ->
                        case n of
                          I# n# ->
```

```
                        case (neInt#! j# n#) of {
                          True ->
                            case i of
                              I# i# ->
                            case (plusInt#! i# j#) of
                              k# ->
                            case m of
                              I# m# ->
                            case (plusInt#! m# n#) of
                              l# ->
                            case (neInt#! k# l#) of {
                              True ->
                                  case (minusInt#! i# j#) of
                                    u# ->
                                        case (minusInt#! m# n#) of
                                          v# ->
                                              neInt#! u# v#
                              False -> False
                            }
                          False -> False
                        }
                        } in
                    Cons b (f ijs')
                }
          }
      } in
        f (zip Int Int (enumFrom (I# 1#)) p)
   } in
     and a

queens = \ n ->
   case n of {
     I# n# ->
         case n# of {
           0# -> Cons Nil Nil
           _  ->
               letrec {
                 g = \ ps ->
                   case ps of {
                     Nil -> Nil
                     Cons p ps' ->
                         letrec {
```

```
                        h = \ is ->
                          case is of {
                            Nil -> g ps'
                            Cons i is' ->
                                case (safe p i)
                                of {
                                  True ->
                                        Cons
                                            (++ Int p (Cons i Nil))
                                            (h is')
                                  False -> h is'
                                }
                          }
                        } in h (enumFromTo (I# 1#) (I# 10#))
                    }
                } in
                  let {
                    n' =
                        case (minusInt#! n# 1#) of {
                          u# -> queens (I# u#)
                        }
                  } in  g n'
        }
    }

main = print Int dfun.Text.Int (sum.Int (concat Int (queens (I# 10#))))

safe =
    letrec {
      f = \ i p n m ->
        case p of {
          Cons j p' ->
              case j of
                I# j# ->
              case n of
                I# n# ->
              case (neInt#! j# n#) of {
                True ->
                    case i of
                      I# i# ->
                    case (plusInt#! i# j#) of
                      k# ->
```

```
                    case m of
                      I# m# ->
                    case (plusInt#! m# n#) of
                      1# ->
                    case (neInt#! k# 1#) of {
                      True ->
                          case (minusInt#! i# j#) of {
                            u# ->
                          case (minusInt#! m# n#) of {
                            v# ->
                              case (neInt#! u# v#) of {
                                False -> False
                                True ->
                                  let {
                                    w = case i of
                                          I# i# ->
                                            case (plusInt#! i# 1#) of
                                              x# -> I# x#
                                  } in
                                    f w p' n m
                              }
                      False ->
                          False
                    }
                False -> False
              }
        Nil -> True
      }
  } in
    \ p n ->
      let {
        m = case (length Int p) of {
              I# u# ->
                  case (plusInt#! u# 1#) of {
                    v# -> I# v#
                  }
            }
      } in  f (I# 1#) p n m

queens =
    letrec {
      g = \ ps ->
```

```
        case ps of {
          Nil -> Nil
          Cons p ps' ->
              h (I# 1#) (I# 10#) ps' p
        }
    h = \ i j ps p ->
        case i of
          I# i# ->
        case j of
          I# j# ->
        case (gtInt#! i# j#) of {
          True -> g ps
          False ->
              case (safe p i) of {
                True ->
                    let { a = (++ Int p (Cons i Nil)
                    } in
                    let { b =
                        case (plusInt#! i# 1#) of
                          u# -> h (I# u#) j ps p
                    } in
                    Cons a b
                False ->
                      case (plusInt#! i# 1#) of
                        u# -> h (I# u#) j ps p
              }
      }
  } in
    \ n ->
        case n of
          I# n# ->
        case n# of
          0# -> Cons Nil Nil
          _  ->
              let {
                ps = case (minusInt#! n# 1#) of
                       u# -> queens (I# u#)
              } in  g ps

main = print Int dfun.Text.Int (sum.Int (concat Int (queens (I# 10#)))
```

Bibliography

[ARS94] L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, Jan 1994.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology, S–412 96 Göteborg, November 1987.

[BD77] R. M. Burstall and J. Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bur77] W. H. Burge. Examples of program optimisation. Technical report, RC 6531, IBM Thomas J Watson Research Centre, Oct 1977.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice-Hall, 1988.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[Chi90] W. N. Chin. *Automatic Methods for Program Transformation.* PhD thesis, University of London, March 1990.

[Fea79] M. S. Feather. *A System for Developing Programs by Transformations.* PhD thesis, University of Edinburgh, 1979.

[Fea82] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, Jan 1982.

[FW89] A. B. Ferguson and P. Wadler. When will deforestation stop. In *Functional Programming, Glasgow*, Workshops in Computing. Springer-Verlag, August 1989.

[Gal93] J. Gallier. Constructive logics part i: A tutorial on proof systems and typed λ-calculi. *Theoretical Computer Science*, 110(2):249–339, March 1993.

[Gen35] G. Gentzen. Investigations into logical deduction. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935.

[Gil95] Andrew Gill. *Cheap Deforestation for Non-Strict Functional Languages.* PhD thesis, Department of Computing Science, Glasgow University, September 1995.

[GLJ93] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, pages 223–232. ACM, 1993.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[GM95] Andrew Gill and Simon Marlow. Personal communication. Curlers, Byres Road, Glasgow, 1995.

[Ham93] G. W. Hamilton. *Compile-Time Optimisation of Store Usage in Lazy Functional Programs.* PhD thesis, University of Stirling, October 1993.

[Ham95] G. W. Hamilton. Higher order deforestation. Technical Report 95-07, University of Keele, 1995.

[HBH93] K. Hammond, G.L. Burn, and D.B. Howe. Spiking your caches. In *Functional Programming, Glasgow*, Ayr, Scotland, 1993. Springer-Verlag.

[HG85] P. Hudak and B. Goldberg. Serial combinators - optimal grains of parallelism. In *Proc IFIP Conf on Functional Programming and Computer Architecture*, volume 201 of *LNCS*, pages 382–389, Nancy, Aug 1985.

[Hop94] Mark Hopkins. The regular infinite lambda calculus. Note distributed to the usenet group comp.lang.functional, October 1994.

[How80] W. Howard. The formulas-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HPW+92] P. Hudak, S. L. Peyton Jones, P. Wadler, et al. Report on the functional programming language haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[Hug83] J. Hughes. *The design and implementation of programming languages.* PhD thesis, Programming Research Group, Oxford, July 1983.

[Hug84] R. J. M. Hughes. A novel representation of lists and its application to the function "reverse". Technical report, Programming Methodology Group, Chalmers Inst, Sweden, 1984.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[Kot78] L. Kott. About a transformation system: A theoretical study. In *Proc. 3rd Symposium on Programming*, pages 232–267, Paris, 1978.

[LJ95] J. Launchbury and S. L. Peyton Jones. State in haskell. In *Lisp and Symbolic Computation*, volume 8, pages 293–342, Dec 1995.

[LS95] John Launchbury and T. Sheard. Warm fusion. In Simon L. Peyton Jones, editor, *Functional Programming Languages and Computer Architecture*, San Diego, California, June 1995. ACM SIGPLAN/SIGARCH, ACM.

[Mar93] Simon Marlow. Update avoidance analysis by abstract interpretation. In *Functional Programming, Glasgow*, Ayr, Scotland, 1993. Springer-Verlag.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar, California, June 1989. IEEE.

[MW92] S. Marlow and P. Wadler. Deforestation for higher order functions. In *Functional Programming, Glasgow*, Workshops in Computing, Ayr, Scotland, 1992. Springer Verlag, Workshops in Computing.

[Par92] W. D. Partain. The nofib benchmarking suite. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*, Ayr, Scotland, 1992. Springer Verlag, Workshops in Computing.

[Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Pey93] S. L. Peyton Jones. The glasgow haskell compiler: a technical overview. In *Joint Framework for Information Technology Conference*, Keele, 1993.

[PJ93] Will Partain and Simon Peyton Jones. On the effectiveness of a simple strictness analyser. In *Functional Programming, Glasgow*, Ayr, Scotland, 1993. Springer-Verlag.

[Pra65] D. Prawitz. *Natural Deduction, A Proof-Theoretical Study.* Almquist & Wisell, Stockholm, 1965.

[PW93] S. L. Peyton Jones and P. L. Wadler. Imperative functional programming. In *Symposium on Principles of Programming Languages*, pages 71–84, Charleston, Jan 1993.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, volume 83, pages 513–523. North-Holland, 1983.

[San95a] D. Sands. Proving the correctness of recursion-based automatic program transformations. In P. Mosses, M. Nielsen, and M. Schwartzbach, editors, *Sixth International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 915 of *Lecture Notes in Computer Science*, pages 681–695. Springer-Verlag, 1995.

[San95b] D. Sands. Total correctness by local improvement in program transformation. In *Symposium on Principles of Programming Languages*, San Francisco, California, January 1995.

[San95c] Andre Santos. *Compilation by Transformation in Non-Strict Functional Languages.* PhD thesis, Department of Computing Science, Glasgow University, 1995.

[Sch80] W. L. Scherlis. *Expression Procedures and Program Derivations.* PhD thesis, Stanford University, Aug 1980.

[SGJ94] M. H. Sørenson, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation and gpc. In *Programming Languages and Systems*, volume 788 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, April 1994.

[Tur82] D.A. Turner. Recursion equations as a programming language. In Darlington, Henderson, and Turner, editors, *Functional programming and its applications.* Cambridge University Press, 1982.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In Simon L. Peyton Jones, editor, *Functional Programming and Computer Architecture*, San Diego, California, June 1995. ACM SIGPLAN/SIGARCH, ACM.

[Wad81] P. Wadler. Applicative style programming, program transformation and list operators, 1981.

[Wad84] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In *ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1984.

[Wad85] P. Wadler. Listlessness is better than laziness II: composing listless functions. In *Proc. Workshop on Programs as Data Objects*, volume 217 of *LNCS*, Copenhagen, 1985. Springer-Verlag.

[Wad87] P. Wadler. The concatenate vanishes. Technical report, Dept of Computing Science, Glasgow University, 1987.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, volume 300 of *LNCS*, Nancy, 1988.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture.* Addison Wesley, 1989.

[Wad90a] P. Wadler. Comprehending monads. In *ACM Symposium on Lisp and Functional Programming*, Nice, June 1990. Later version to appear in Mathematical Structures in Computer Science.

[Wad90b] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wad92] P. Wadler. The essence of functional programming. In *Symposium on Principles of Programming Languages*, 1992.