Deforestation of Functional Programs through Type Inference

Olaf Chitil

Lehrstuhl für Informatik II, RWTH Aachen, Germany
chitil@informatik.rwth-aachen.de

Abstract. Deforestation optimises a functional program by transforming it into another one that does not create certain intermediate data structures. Short cut deforestation is a deforestation method which is based on a single, local transformation rule. In return, short cut deforestation expects both producer and consumer of the intermediate structure in a certain form. Starting from the fact that short cut deforestation is based on a parametricity theorem of the second-order typed λ-calculus, we show how the required form of a list producer can be derived through the use of type inference. Type inference can also indicates which function definitions need to be inlined. Because only limited inlining across module boundaries is practically feasible, we develop a scheme for splitting a function definition into a worker definition and a wrapper definition. For deforestation we only need to inline the small wrapper definition.

1 Introduction

In functional programming modularity is often achieved through intermediate data structures. Two separately defined functions can be glued together by an intermediate data structure that is produced by one function and consumed by the other. For example, the function `any`, which tests whether any element of a list `xs` satisfies a given predicate `p`, may be defined as follows in Haskell [10]:

```
any :: (a -> Bool) -> [a] -> Bool

any p xs = or (map p xs)
```

The function `map` applies `p` to all elements of `xs` yielding a list of boolean values. The function `or` combines these boolean values with the logical or operation (`||`). So we defined `any` by glueing together the **producer** `map p xs` and the **consumer** `or` by an intermediate list.

In [8] John Hughes points out that lazy functional languages make modularity through intermediate data structures practicable. Considering our definition of `any` we note that with eager evaluation the whole intermediate boolean list is produced before it is consumed by the function `or`. Hence modularity through intermediate data structures is seldomly used in languages with eager evaluation.

In contrast, lazy evaluation ensures that the boolean list is produced one cell at a time. Such a cell is immediately consumed by `or` and becomes garbage, which can be reclaimed automatically. Hence the function `any` runs in constant space. Furthermore, when `or` comes across the value `True`, the production of the list is aborted. Thus the termination condition is separated from the "loop body".

```
   any (> 2) [1,2,3,4,5,6,7,8,9]
⇝  or (map (> 2) [1,2,3,4,5,6,7,8,9])
⇝  or (False : (map (> 2) [2,3,4,5,6,7,8,9]))
⇝  or (map (> 2) [2,3,4,5,6,7,8,9])
⇝  or (False : (map (> 2) [3,4,5,6,7,8,9]))
⇝  or (map (> 2) [3,4,5,6,7,8,9])
⇝  or (True : (map (> 2) [4,5,6,7,8,9]))
⇝  True
```

We can see from the discussion why the use of intermediate data structures as glue has become popular in lazy functional programming.

Deforestation. Nonetheless this modular programming style does not come for free. Each list cell has to be allocated, filled, taken apart and finally garbage collected. The following monolithic definition of `any` is more efficient than the modular one, because it does not construct an intermediate list.

```
any p []     = False
any p (x:xs) = p x || any p xs
```

It is the aim of **deforestation** algorithms [13] to transform automatically a modular functional program which uses intermediate data structures as glue into another one which does not produce these intermediate data structures. We say that the producer and the consumer of a data structure are **fused**. Besides removing the costs of an intermediate data structure, deforestation also brings together subterms of producer and consumer which previously were separated by the intermediate data structure. Thus new opportunities for further optimising transformations arise.

Short Cut Deforestation. Despite the extensive literature on various deforestation methods, their implementation in real compilers proved to be difficult. Hence, a simple deforestation method, called **cheap** or **short cut deforestation**, was developed [6, 7].

The fundamental idea of short cut deforestation is to restrict deforestation to intermediate lists which are consumed by the function `foldr`. Lists are the most common intermediate data structures in functional programs. The higher-order function `foldr` uniformly replaces in a list all occurrences of the constructor `(:)` by a given function $\oplus$ and the empty list constructor `[]` by a given constant `n`:[1]

[1] Note that the term `[`$x_1, x_2, x_3, \ldots, x_k$`]` is only syntactic sugar for x_1`:(`x_2`:(`x_3`:(`$\ldots$`(`x_k`:[]))`$\ldots$`)`.

$$\texttt{foldr}\ (\oplus)\ \texttt{n}\ [x_1, x_2, x_3, \ldots, x_k] \;=\; x_1 \oplus (x_2 \oplus (x_3 \oplus (\ldots (x_k \oplus n)\ldots)))$$

So if `foldr` replaces the list constructors in a list which is produced by a term $M^{\mathtt{P}}$ at runtime, then short cut deforestation simply replaces the list constructors already at compile time. However, the naïve transformation rule

$$\texttt{foldr}\ M^{\texttt{(:)}}\ M^{\texttt{[]}}\ M^{\mathtt{P}} \quad\rightsquigarrow\quad M^{\mathtt{P}}[M^{\texttt{(:)}}/\texttt{(:)}, M^{\texttt{[]}}/\texttt{[]}]$$

which replaces all list constructors in $M^{\mathtt{P}}$ is *wrong*. Consider $M^{\mathtt{P}} =$ `(map p [1,2])`. Here the constructors in `[1,2]` are not to be replaced but those in the definition of `map`, which is not even part of $M^{\mathtt{P}}$.

Therefore, we need the producer $M^{\mathtt{P}}$ in a form such that the constructors which construct the intermediate list are explicit and can be replaced easily. The convenient solution is to have the producer in the form $(\backslash\texttt{v}^{\texttt{(:)}}\ \texttt{v}^{\texttt{[]}}\ \texttt{->}\ M')\ \texttt{(:)}\ \texttt{[]}$ where the abstracted variables $\texttt{v}^{\texttt{(:)}}$ and $\texttt{v}^{\texttt{[]}}$ mark the positions of the intermediate list constructors `(:)` and `[]`. Then fusion is performed by the simple rule:

$$\begin{array}{ll} & \texttt{foldr}\ M^{\texttt{(:)}}\ M^{\texttt{[]}}\ ((\backslash\texttt{v}^{\texttt{(:)}}\ \texttt{v}^{\texttt{[]}}\ \texttt{->}\ M')\ \texttt{(:)}\ \texttt{[]}) \\ \rightsquigarrow & (\backslash\texttt{v}^{\texttt{(:)}}\ \texttt{v}^{\texttt{[]}}\ \texttt{->}\ M')\ \ M^{\texttt{(:)}}\ M^{\texttt{[]}} \end{array}$$

The rule removes the intermediate list constructors. Subsequent reduction steps put the consumer components $M^{\texttt{(:)}}$ and $M^{\texttt{[]}}$ into the places which were before held by the constructors. We call $\backslash\texttt{v}^{\texttt{(:)}}\ \texttt{v}^{\texttt{[]}}\ \texttt{->}\ M'$ **producer skeleton**, because it is equal to the producer $M^{\mathtt{P}}$ except that the constructors of the result list are abstracted.

We observe that in general the types of $M^{\texttt{(:)}}$ and $M^{\texttt{[]}}$ are different from the types of `(:)` and `[]`. Hence, for this transformation to be type correct, the producer skeleton must be polymorphic. So we finally formulate short cut fusion as follows: Let A be a type and c be a type variable. If the term P has the type $(A\ \texttt{->}\ c\ \texttt{->}\ c)\ \texttt{->}\ c\ \texttt{->}\ c$, then we may apply the transformation

$$\texttt{foldr}\ M^{\texttt{(:)}}\ M^{\texttt{[]}}\ (P\ \texttt{(:)}\ \texttt{[]}) \quad\rightsquigarrow\quad P\ \ M^{\texttt{(:)}}\ M^{\texttt{[]}}$$

Usually, the producer skeleton P has the form $\backslash\texttt{v}^{\texttt{(:)}}\ \texttt{v}^{\texttt{[]}}\ \texttt{->}\ M'$, but this is not required for the semantic correctness of the transformation. Strikingly, the polymorphic type of P already guarantees the correctness. Intuitively, P can only construct its result of type c from its two arguments, because only these have matching types. Formally, the transformation is an instance of a parametricity or free theorem [12].

The original short cut deforestation method requires a list producer to be defined explicitly in terms of a polymorphic producer skeleton. To easily recognise the producer skeleton and to ensure that it has the required polymorphic type, a special function `build` with a second-order type is introduced:

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

The local quantification of the type variable `b` in the type of `build` ensures that the argument of `build` is of the desired polymorphic type. Standard HASKELL does not have second-order types. Hence, a compiler needs to be extended to support the function `build`. With `build` the short cut fusion rule can be written as follows:

$$\texttt{foldr}\ M^{\texttt{(:)}}\ M^{\texttt{[]}}\ \texttt{(build}\ P\texttt{)} \quad \rightsquigarrow \quad P\ M^{\texttt{(:)}}\ M^{\texttt{[]}}$$

This transformation rule can easily be implemented in a compiler, hence the names cheap or short cut deforestation. The idea is that the compiler writer defines all list-manipulating functions in the standard libraries, which are used extensively by programmers, in terms of `build` and `foldr`.

Type Inference Identifies List Constructors. Originally, the second-order type of `build` confined deforestation to producers which are defined in terms of list-producing functions from the standard library. Today, the Glasgow Haskell compiler [5] has an extended type system which permits the programmer to use functions such as `build`. However, asking the programmer to supply list producers in `build` form runs contrary to the aim of writing clear and concise programs. The simplicity of a list producer would be lost as the following definition of the function `map` demonstrates:

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

Whereas `foldr` abstracts a common recursion scheme and hence its use for defining list consumers is generally considered as good, modular programming style, `build` is only a crutch to enable deforestation.

The starting-point for making `build` superfluous is the observation that the correctness of the short cut fusion rule solely depends on the polymorphic type of the producer skeleton. So we reduce the problem of transforming an arbitrary list-producing term into the required form to a type inference problem.

We can obtain a polymorphic producer skeleton from a producer $M^{\texttt{P}}$ by the following generate-and-test method: First, we replace in $M^{\texttt{P}}$ some occurrences of the constructor `(:)` by a variable $\texttt{v}^{\texttt{(:)}}$ and some occurrences of the constructor `[]` by a variable $\texttt{v}^{\texttt{[]}}$. We obtain a term M'. Next we type check the term $\texttt{\\v}^{\texttt{(:)}}\,\texttt{v}^{\texttt{[]}}$ `->` M'. If it has the polymorphic type $(A$ `->` c `->` $c)$ `->` c `->` c for some type A and type variable c, then we have abstracted exactly those constructors which construct the result list and thus we have found the producer skeleton. Otherwise, we try a different replacement of `(:)`s and `[]`s. If no replacement gives the desired type, then no short cut deforestation takes place.

Obviously, this method is prohibitively expensive. Fortunately, we can determine the list constructors that need to be replaced in one pass, if we use an algorithm which infers a most general, a principal typing. We call the transformation which obtains the polymorphic producer skeleton from an arbitrary producer **list abstraction**.

2 List Abstraction through Type Inference

In the following we use a small second-order typed functional language with explicit type abstraction and type application (cf. [1]). The explicit handling of types makes clearer, how terms are transformed. Additionally, this language is only a slightly simplified version of the intermediate language used inside the Glasgow Haskell compiler [5].

The following term produces a list of type `[Int]`:

```
let mapInt : (Int→Int)→[Int]→[Int]
           = λf:Int→Int.λxs:[Int].case xs of
                 []   ↦ []
                 y:ys ↦ (f y) : (mapInt f ys)
in mapInt inc [1,2]
```

For the moment we only consider monomorphic list constructors, that is, `(:)` has type `Int→[Int]→[Int]` and `[]` has type `[Int]`, and a monomorphic version of `map` for type `Int`. Furthermore we assume that the definition of `mapInt` is part of the producer. We will later lift these restrictions step by step. We start list abstraction with the typing of the producer:

```
{inc:Int→Int}
⊢ let mapInt : (Int→Int)→[Int]→[Int]
             = λf:Int→Int.λxs:[Int].case xs of
                   []   ↦ []
                   y:ys ↦ (:) (f y) (mapInt f ys)
  in mapInt inc ((:) 1 ((:) 2 []))
: [Int]
```

The typing environment, given before $\vdash$, assigns types to all free variables of the producer. With respect to this typing environment the producer has the type given after the colon.

We replace the list constructor `(:)`, respectively `[]`, at every occurrence by a new variable $v^{(:)}$, respectively $v^{[]}$ (except for patterns in `case` constructs, because these do not construct but destruct a list). Furthermore, the types in the term and in the typing environment have to be modified. To use the existing ones as far as possible, we only replace the list type `[Int]` at every occurrence by a new type variable γ. Furthermore, we add $v^{(:)} : \texttt{Int} \to \gamma \to \gamma$, respectively $v^{[]} : \gamma$, to the typing environment, where γ is a new type variable for every variable $v^{(:)}$, respectively $v^{[]}$.

$$
\begin{array}{l}
\{\texttt{inc}:\texttt{Int}\to\texttt{Int},\ \texttt{v}_1^{(:)}:\texttt{Int}\to\gamma_1\to\gamma_1,\ \texttt{v}_2^{(:)}:\texttt{Int}\to\gamma_2\to\gamma_2,\\
\ \texttt{v}_3^{(:)}:\texttt{Int}\to\gamma_3\to\gamma_3,\ \texttt{v}_1^{[]}:\gamma_4,\ \texttt{v}_2^{[]}:\gamma_5\}\\
\vdash \texttt{let mapInt : (Int}\to\texttt{Int)}\to\gamma_6\to\gamma_7\\
\qquad\qquad\quad = \lambda\texttt{f:Int}\to\texttt{Int}.\lambda\texttt{xs}:\gamma_8.\texttt{case xs of}\\
\qquad\qquad\qquad\quad \texttt{[]}\ \ \ \ \mapsto \texttt{v}_1^{[]}\\
\qquad\qquad\qquad\quad \texttt{y:ys} \mapsto \texttt{v}_1^{(:)}\ \texttt{(f y) (mapInt f ys)}\\
\quad \texttt{in mapInt inc (}\texttt{v}_2^{(:)}\ \texttt{1 (}\texttt{v}_3^{(:)}\ \texttt{2 }\texttt{v}_2^{[]}\texttt{))}
\end{array}
$$

This typing environment and term with type variables do not form a valid typing for any type. They are the input to a type inference algorithm. The type inference algorithm replaces some of the new type variables $\gamma_1, \ldots, \gamma_8$ and determines a type to obtain again a valid typing. More precisely, the type inference algorithm determines a principal typing, that is, the most general instance of the input that gives a valid typing. Note that type inference cannot fail, because the typing we start with is valid. In the worst case the type inference algorithm yields the typing of the original producer. We just try to find a more general typing. For our example the type inference algorithm yields the valid typing:

$$
\begin{array}{l}
\{\texttt{inc}:\texttt{Int}\to\texttt{Int},\, \texttt{v}_1^{(:)}:\texttt{Int}\to\gamma\to\gamma,\, \texttt{v}_2^{(:)}:\texttt{Int}\to\texttt{[Int]}\to\texttt{[Int]},\\
\;\texttt{v}_3^{(:)}:\texttt{Int}\to\texttt{[Int]}\to\texttt{[Int]},\, \texttt{v}_1^{[]}:\gamma,\, \texttt{v}_2^{[]}:\texttt{[Int]}\}\\
\vdash \texttt{let mapInt : (Int}\to\texttt{Int)}\to\texttt{[Int]}\to\gamma\\
\qquad\qquad\quad \texttt{=}\,\lambda\texttt{f:Int}\to\texttt{Int}.\,\lambda\texttt{xs:[Int].case xs of}\\
\qquad\qquad\qquad\quad \texttt{[]}\quad\mapsto \texttt{v}_1^{[]}\\
\qquad\qquad\qquad\quad \texttt{y:ys}\mapsto \texttt{v}_1^{(:)}\ \texttt{(f y) (mapInt f ys)}\\
\quad\texttt{in mapInt inc (}\texttt{v}_2^{(:)}\ \texttt{1 (}\texttt{v}_3^{(:)}\ \texttt{2 }\texttt{v}_2^{[]}\texttt{))}\\
:\gamma
\end{array}
$$

The type of the term is a type variable γ. Hence list abstraction is possible. The typing environment tells us that $\texttt{v}_1^{(:)}$ and $\texttt{v}_1^{[]}$ construct values of type γ, so they construct the result of the producer. In contrast $\texttt{v}_2^{(:)}$, $\texttt{v}_3^{(:)}$, and $\texttt{v}_2^{[]}$ have the types of normal list constructors. Hence they construct lists that are internal to the producer. So we reinstantiate $\texttt{v}_2^{(:)}$, $\texttt{v}_3^{(:)}$, and $\texttt{v}_2^{[]}$ to normal list constructors and abstract the type variable γ and the variables $\texttt{v}_1^{(:)}$ and $\texttt{v}_1^{[]}$ to obtain the producer skeleton of the required type:

$$
\begin{array}{l}
\{\texttt{inc}:\texttt{Int}\to\texttt{Int}\}\\
\vdash \lambda\gamma.\,\lambda\texttt{v}_1^{(:)}:\texttt{Int}\to\gamma\to\gamma.\,\lambda\texttt{v}_1^{[]}:\gamma.\\
\qquad\texttt{let mapInt : (Int}\to\texttt{Int)}\to\texttt{[Int]}\to\gamma\\
\qquad\qquad\qquad \texttt{=}\,\lambda\texttt{f:Int}\to\texttt{Int}.\,\lambda\texttt{xs:[Int].case xs of}\\
\qquad\qquad\qquad\quad \texttt{[]}\quad\mapsto \texttt{v}_1^{[]}\\
\qquad\qquad\qquad\quad \texttt{y:ys}\mapsto \texttt{v}_1^{(:)}\ \texttt{(f y) (mapInt f ys)}\\
\qquad\texttt{in mapInt inc ((:) 1 ((:) 2 []))}\\
:\forall\gamma.(\texttt{Int}\to\gamma\to\gamma)\to\gamma\to\gamma
\end{array}
$$

The complete producer can be written as

$$(\lambda\gamma.\,\lambda\texttt{v}_1^{(:)}:\texttt{Int}\to\gamma\to\gamma.\,\lambda\texttt{v}_1^{[]}:\gamma.\,\texttt{let}\ \ldots\ \texttt{in}\ \ldots)\ \texttt{[Int] (:) []}$$

In this list abstracted form it is suitable for short cut fusion with a `foldr` consumer.

Until now we assumed that we only have lists over `Int`s. In reality lists are polymorphic, that is, `(:)` has type $\forall\alpha.\alpha\to[\alpha]\to[\alpha]$ and `[]` has type $\forall\alpha.[\alpha]$. So the typing we start with looks as follows:

```
{inc:Int→Int}
⊢ let mapInt : (Int→Int)→[Int]→[Int]
             = λf:Int→Int.λxs:[Int].case xs of
                 []    ↦ [] Int
                 y:ys ↦ (:) Int (f y) (mapInt f ys)
  in mapInt inc ((:) Int 1 ((:) Int 2 ([] Int)))
: [Int]
```

We want to abstract the list of type `[Int]`. Therefore we replaces every list constructor *application* `(:) Int`, respectively `[] Int`, by a different variable $v^{(:)}$, respectively $v^{[]}$. Then we continue just as described in the previous section.

After type inference we naturally have to replace again those variables $v^{(:)}$ and $v^{[]}$ that are not abstracted by list constructor applications `(:) Int`, respectively `[] Int`. We obtain a producer skeleton of the required type

```
{inc:Int→Int}
⊢ λγ. λv_1^(:):Int→ γ → γ. λv_1^[]:γ.
    let mapInt : (Int→Int)→[Int]→ γ
               = λf:Int→Int.λxs:[Int].case xs of
                   []    ↦ v_1^[]
                   y:ys ↦ v_1^(:)  (f y) (mapInt f ys)
    in mapInt inc ((:) Int 1 ((:) Int 2 ([] Int)))
: ∀γ.(Int→ γ → γ) → γ → γ
```

and the complete producer looks as follows:

```
(λγ. λv_1^(:):Int→ γ → γ. λv_1^[]:γ. let ...) [Int] ((:) Int) ([] Int)
```

Note that in contrast to the list constructors `(:)` and `[]` the abstracted variables $\mathtt{v}_1^{(:)}$ and $\mathtt{v}_1^{[]}$ must have a monomorphic type, because the terms $M^{(:)}$ and $M^{[]}$ of a consumer `foldr` τ_1 τ_2 $M^{(:)}$ $M^{[]}$ are monomorphic.

3 Inlining of Definitions

In practise the definition of `mapInt` will not be part of the producer. The producer will just be `mapInt inc [1,2]`, from which it is impossible to abstract the list constructors that construct the result list, because they are not part of the term. Hence we may have to inline definitions of variables such as `mapInt`, that is, make them part of the producer.

Generally, implementations of deforestation have problems with controlling the necessary inlining to avoid code explosion [9]. Rather nicely, instead of having to use some heuristics for inlining, we can use the typing environment of a principal typing to determine exactly those variables whose definitions need to be inlined. Note that it is important not just to inline the right hand side of a recursive definition but the whole recursive definition.

We consider the producer `mapInt inc [1,2]`. So we start with its typing:

$$\{\texttt{inc}:\texttt{Int}\to\texttt{Int}, \texttt{mapInt}:(\texttt{Int}\to\texttt{Int})\to\texttt{[Int]}\to\texttt{[Int]}\}$$
$$\vdash \texttt{mapInt inc ((:) Int 1 ((:) Int 2 ([] Int)))}$$
$$: \texttt{[Int]}$$

Before type inference we replace the type [Int] at every occurrence by a new type variable, not only in the term but also in the types of all inlineable variables in the typing environment.

$$\{\texttt{inc}:\texttt{Int}\to\texttt{Int}, \texttt{mapInt}:(\texttt{Int}\to\texttt{Int})\to\gamma_1\to\gamma_2,$$
$$\texttt{v}_1^{(:)}:\texttt{Int}\to\gamma_3\to\gamma_3, \texttt{v}_2^{(:)}:\texttt{Int}\to\gamma_4\to\gamma_4, \texttt{v}_1^{[]}:\gamma_5\}$$
$$\vdash \texttt{mapInt inc (}\texttt{v}_1^{(:)}\texttt{ 1 (}\texttt{v}_2^{(:)}\texttt{ 2 }\texttt{v}_1^{[]}\texttt{))}$$

The type inference algorithm gives us the principal typing

$$\{\texttt{inc}:\texttt{Int}\to\texttt{Int}, \texttt{mapInt}:(\texttt{Int}\to\texttt{Int})\to\gamma_1\to\gamma_2,$$
$$\texttt{v}_1^{(:)}:\texttt{Int}\to\gamma_1\to\gamma_1, \texttt{v}_2^{(:)}:\texttt{Int}\to\gamma_1\to\gamma_1, \texttt{v}_1^{[]}:\gamma_1\}$$
$$\vdash \texttt{mapInt inc (}\texttt{v}_1^{(:)}\texttt{ 1 (}\texttt{v}_2^{(:)}\texttt{ 2 }\texttt{v}_1^{[]}\texttt{))}$$
$$: \gamma_2$$

The type of the term is a type variable γ_2. However, we cannot abstract γ_2, because it appears in the typing environment in the type of mapInt. So this occurrence of γ_2 signifies that the definition of mapInt needs to be inlined.

In practise a producer will be defined in terms of the the polymorphic function map instead of mapInt. We have to instantiate uses of map as far as possible before applying the list abstraction algorithm. In the definition of map we drop the abstraction from the two type variables and thus create a new definition for a function $\texttt{map}_{\texttt{Int Int}}$. In the producer we replace map Int Int by $\texttt{map}_{\texttt{Int Int}}$. Then we can abstract the produced list as before.

4 The Worker/Wrapper Scheme

It is neat that the algorithm determines exactly the functions that need to be inlined, but nonetheless inlining causes problems in practise. Extensive inlining across module boundaries would defeat the idea of separate compilation. Furthermore, in practise “inlining is a black art, full of delicate compromises that work together to give good performance without unnecessary code bloat” [11]. It is best implemented as a separate optimisation pass. Consequently, we would like to use our list abstraction algorithm without it having to perform inlining itself.

To be able to abstract the result list from a producer without using inlining, all list constructors that construct the result list already have to be present in the producer. Therefore we use a so called worker/wrapper scheme as it has already been proposed by Gill for short cut deforestation [7]. We split every definition of a function that produces a list into a definition of a worker and a definition of a wrapper. The definition of the worker is obtained from the original definition by abstracting the result list type and its list constructors. The definition of the wrapper, which calls the worker, contains all the list constructors that construct the result list. For example, we split the definition of the function mapInt

$$\begin{array}{l}\texttt{mapInt} : (\texttt{Int}\to\texttt{Int})\to[\texttt{Int}]\to[\texttt{Int}]\\ \quad = \lambda\texttt{f}:\texttt{Int}\to\texttt{Int}.\texttt{foldr Int [Int]}\\ \qquad (\lambda\texttt{u}:\texttt{Int}.\lambda\texttt{w}:[\texttt{Int}].\texttt{(:) Int (f u) w) ([] Int)}\end{array}$$

into definitions of a worker `mapIntW` and a wrapper `mapInt`:

$$\begin{array}{l}\texttt{mapIntW} : \forall\gamma.\,(\texttt{Int}\to\gamma\to\gamma)\to\gamma\to(\texttt{Int}\to\texttt{Int})\to[\texttt{Int}]\to\gamma\\ \quad = \lambda\gamma.\,\lambda\texttt{v}^{(:)}:\texttt{Int}\to\gamma\to\gamma.\,\lambda\texttt{v}^{[]}:\gamma.\,\lambda\texttt{f}:\texttt{Int}\to\texttt{Int}.\\ \qquad \texttt{foldr Int}\ \gamma\ (\lambda\texttt{u}:\texttt{Int}.\lambda\texttt{w}:\gamma.\ \texttt{v}^{(:)}\ \texttt{(f u) w})\ \texttt{v}^{[]}\end{array}$$

$$\begin{array}{l}\texttt{mapInt} : (\texttt{Int}\to\texttt{Int})\to[\texttt{Int}]\to[\texttt{Int}]\\ \quad = \texttt{mapW [Int] ((:) Int) ([] Int)}\end{array}$$

Just as easily we split the definition of the polymorphic function `map`

$$\begin{array}{l}\texttt{map} : \forall\alpha.\forall\beta.\,(\alpha\to\beta)\to[\alpha]\to[\beta]\\ \quad = \lambda\alpha.\,\lambda\beta.\,\lambda\texttt{f}:\alpha\to\beta.\\ \qquad \texttt{foldr}\ \alpha\ [\beta]\ (\lambda\texttt{u}:\alpha.\lambda\texttt{w}:[\beta].\texttt{(:)}\ \beta\ \texttt{(f u) w)}\ \texttt{([]}\ \beta)\end{array}$$

into definitions of a worker `mapW` and a wrapper `map`:

$$\begin{array}{l}\texttt{mapW} : \forall\alpha.\forall\beta.\forall\gamma.\,(\beta\to\gamma\to\gamma)\to\gamma\to(\alpha\to\beta)\to[\alpha]\to\gamma\\ \quad = \lambda\alpha.\,\lambda\beta.\,\lambda\gamma.\,\lambda\texttt{v}^{(:)}:\beta\to\gamma\to\gamma.\,\lambda\texttt{v}^{[]}:\gamma.\,\lambda\texttt{f}:\alpha\to\beta.\\ \qquad \texttt{foldr}\ \alpha\ \gamma\ (\lambda\texttt{u}:\alpha.\lambda\texttt{w}:\gamma.\ \texttt{v}^{(:)}\ \texttt{(f u) w})\ \texttt{v}^{[]}\end{array}$$

$$\begin{array}{l}\texttt{map} : \forall\alpha.\forall\beta.\,(\alpha\to\beta)\to[\alpha]\to[\beta]\\ \quad = \lambda\alpha.\,\lambda\beta.\texttt{mapW}\ \alpha\ \beta\ [\beta]\ \texttt{((:)}\ \beta)\ \texttt{([]}\ \beta)\end{array}$$

For deforestation we only need to inline the wrapper. Consider for example deforestation of the body of the definition of `any` as defined in the introduction:

$$\begin{array}{ll} & \texttt{or (map}\ \tau\ \texttt{Bool p xs)}\\ \rightsquigarrow & \{\text{inlining of } \texttt{or} \text{ and } \texttt{map}\}\\ & \texttt{foldr Bool Bool (||) False}\\ & \quad \texttt{(mapW}\ \tau\ \texttt{Bool [Bool] ((:) Bool) ([] Bool) p xs)}\\ \rightsquigarrow & \{\text{list abstraction of the producer}\}\\ & \texttt{foldr Bool Bool (||) False}\\ & \quad ((\lambda\gamma.\,\lambda\texttt{v}^{(:)}:\beta\to\gamma\to\gamma.\,\lambda\texttt{v}^{[]}:\gamma.\,\texttt{mapW}\ \tau\ \texttt{Bool}\ \gamma\ \texttt{v}^{(:)}\ \texttt{v}^{[]}\ \texttt{p xs})\\ & \quad \texttt{[Bool] ((:) Bool) ([] Bool))}\\ \rightsquigarrow & \{\text{fusion and subsequent } \beta\text{-reduction}\}\\ & \texttt{mapW}\ \tau\ \texttt{Bool Bool (||) False p xs}\end{array}$$

It is left to the standard inliner, if `mapW` is inlined. Across module boundaries or if its definition is large, a worker may not be inlined. This does not influence deforestation.

5 Conclusions

We described how a type inference algorithm enables automatic transformation of a nearly arbitrary producer term into the form required by short cut deforestation. The new deforestation algorithm searches for terms of the form $\texttt{foldr}\ \tau_1\ \tau_2\ M^{(:)}\ M^{[]}\ M^{\mathtt{P}}$, transforms the producer term $M^{\mathtt{P}}$ into the form $P\ \texttt{[}\tau_1\texttt{]}\ \texttt{((:)}\,\tau_1\texttt{)}\ \texttt{([]}\,\tau_1\texttt{)}$ with P of type $\forall\gamma.(\tau_1 \rightarrow \gamma \rightarrow \gamma) \rightarrow \gamma \rightarrow \gamma$ and subsequently applies the short cut fusion rule. The method also indicates where inlining is required and permits deforestation across module boundaries, requiring only inlining of very small wrapper definitions.

Here we only outlined the idea of using type inference for deforestation. Algorithms, formal descriptions and proofs are given in [2, 4]. Shortly a detailed presentation of the whole approach will appear in [3].

References

1. Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.
2. Olaf Chitil. Type inference builds a short cut to deforestation. *ACM SIGPLAN Notices*, 34(9):249–260, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).
3. Olaf Chitil. *Type-Inference Based Deforestation of Functional Programs*. PhD thesis, RWTH Aachen, 2000. to appear.
4. Olaf Chitil. Type-inference based short cut deforestation (nearly) without inlining. In *Proceedings of the 11th International Workshop on Implementation of Functional Languages 1999*, LNCS. Springer, 2000. to appear.
5. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
6. Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM Press, 1993.
7. Andrew J. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, 1996.
8. John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
9. Simon Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Glasgow University, 1995.
10. Simon L. Peyton Jones, John Hughes, et al. Haskell 98: A non-strict, purely functional language. http://www.haskell.org, February 1999.
11. Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell compiler inliner. IDL '99, http://www.binnetcorp.com/wshops/IDL99.html, 1999.
12. Philip Wadler. Theorems for free! In *4th International Conference on Functional Programming Languages and Computer Architectures*, pages 347–359. ACM Press, 1989.
13. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.