# Type-Inference Based Short Cut Deforestation (nearly) without Inlining
## — Work in Progress —

Olaf Chitil

Lehrstuhl für Informatik II, RWTH Aachen,
52056 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. In [Chi99] we presented a type-inference based deforestation algorithm which performs extensive inlining. However, across module boundaries only limited inlining is practically feasible. Furthermore, inlining is a non-trivial transformation which is therefore best implemented as a separate optimisation pass. To perform short cut deforestation (nearly) without inlining, Gill suggested to split definitions into workers and wrappers and inline only the small wrappers, which transfer the information needed for deforestation. We show that Gill's use of a function `build` limits deforestation and note that his reasons for using `build` do not apply to our approach. Hence we develop a more general worker/wrapper scheme without `build`. We give a type-inference based algorithm which splits definitions into workers and wrappers. Finally, we show that we can deforest more expressions with the worker/wrapper scheme than the algorithm with inlining.

## 1 Type-Inference-Based Short Cut Deforestation

In lazy functional programs two functions are often 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 [PH+99]:

```
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 `(||)`.

Although lazy evaluation makes this modular programming style practicable [Hug89], it 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.

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

It is the aim of *deforestation* algorithms to automatically transform a functional program into another one that does not create such intermediate data structures. We say that the producer and the consumer of the data structure are *fused* .

## 1.1 Short Cut Deforestation

The fundamental idea of short cut deforestation is to restrict deforestation to intermediate lists that are consumed by the function `foldr`. This higher-order function uniformly replaces the constructors `(:)` in a list by a given function `c` and the empty list constructor `[]` by a constant `n`:

$$\texttt{foldr c n } [x_1, \ldots, x_k] = x_1 \text{ `c` } (x_2 \text{ `c` } (x_3 \text{ `c` } (\ldots (x_k \text{ `c` n}) \ldots)))$$

So if `foldr` consumes a list that is produced by an expression $e$, short cut deforestation replaces the list constructors already at compile time. We cannot, however, simply replace all list constructors in $e$. Consider $e =$ `(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 $e$.

Therefore we need the producer $e$ in a form such that the list constructors that build the intermediate list are explicit and can easily be replaced. The convenient solution is to have the producer in the form `(\c n ->` $e'$`) (:) []` where the abstracted variables `c` and `n` mark the intermediate list constructors `(:)` and `[]`. Then fusion is performed by the simple rule:

$$\texttt{foldr } e_{(:)}\ e_{[]}\ \texttt{((\c n ->}\ e'\texttt{) (:) [])} \quad \rightsquigarrow \quad \texttt{(\c n ->}\ e'\texttt{)}\ e_{(:)}\ e_{[]}$$

The rule removes the intermediate list constructors. A subsequent $\beta$-reduction puts the consumer components $e_{(:)}$ and $e_{[]}$ into the places that were before hold by the list constructors.

We observe that generally $e_{(:)}$ and $e_{[]}$ have different types from `(:)` and `[]`. Hence for this transformation to be type correct the function in the producer must be polymorphic. This can be expressed in Haskell with the help of a special function `build` with a second-order type:

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

$$\texttt{foldr } e_{(:)}\ e_{[]}\ \texttt{(build } e_{\mathrm{p}}\texttt{)} \quad \rightsquigarrow \quad e_{\mathrm{p}}\ e_{(:)}\ e_{[]}$$

Here $e_{\mathrm{p}}$ will always have the form `\c n ->` $e'$, but this is not necessary for the semantic correctness of the transformation. Strikingly, the polymorphic type of $e_{\mathrm{p}}$ already guarantees the correctness. Intuitively, $e_{\mathrm{p}}$ can only build its value of type `b` from its two term arguments, because only these have the right types. The correctness has been proved in [GLP93, Gil96].

## 1.2 Derivation of Producers through Type Inference

Whereas using `foldr` for defining list consumers is generally considered as good, modular programming style, programmers can hardly be demanded to use `build`. The idea of the first works on short cut deforestation is that all list-manipulating functions in the standard libraries are defined in terms of `foldr` and `build`. However, thus deforestation is confined to producers that are defined in terms of these standard list functions.

On the other hand we see that, if we can transform a producer $e$ of type `[`$\tau$`]` into the form `build (\c n ->` $e'$`)`, then the type system guarantees that we have abstracted exactly those

list constructors that build the intermediate list. Based on this observation we presented in [Chi99] a type-inference based algorithm which abstracts the intermediate list type and its constructors from a producer to obtain a `build` form.

For the producer `map p [1,2]` for example, this algorithm observes, that the intermediate list is constructed by the function `map`. Therefore it inlines the body of `map` to be able to proceed. Afterwards the algorithm decides that the list constructors in the body of `map` have to be abstracted whereas the list constructors in `[1,2]` remain unchanged. With this result the algorithm terminates successfully. In general, the algorithm recursively inlines all functions that are needed to be able to derive the desired form of the producer, only bounded by an arbitrary code size limit. We recapitulate the algorithm in more detail in Section 3.

### 1.3 The Problem of Inlining

It is neat that the algorithm determines exactly those functions which need to be inlined, but nonetheless it causes problems in practise. Inlining across module boundaries is usually implemented by saving some small function definitions in a file when a module is compiled. This file is read and used when a module is compiled which imports the former module. Extensive inlining across module boundaries would defeat the idea of separate compilation. In general, inlining, although trivial in principal, is in practise "a black art, full of delicate compromises that work together to give good performance without unnecessary code bloat" [PM99]. 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.

Gill already suggested a method to separate short cut deforestation from inlining[Gil96]: Each list-producing function definition is split into a worker and a wrapper. The latter is small enough to be inlined everywhere, also across module boundaries, and transfers enough information to permit short cut deforestation. Inlining can be performed subsequently to improve the result of deforestation further. However, Gill's use of `build` in wrappers limits the expressibility of his worker/wrapper scheme. There are functions that cannot be defined in terms of `build` and hence cannot be deforested within Gill's worker/wrapper scheme, but that can be deforested by our type-inference based algorithm with inlining. Fortunately, the function `build` turns out to be unnecessary for our type-inference based deforestation algorithm. Hence we developed a more general worker/wrapper scheme without `build`. We present it in Section 4.

Afterwards we show in Section 5 how the list-producing function definitions of a program are split into wrappers and workers by an algorithm that is based on our type-inference based algorithm without using inlining.

In Section 6 we present a class of recursively defined functions which cannot be deforested by the algorithm with inlining but can be deforested within the worker/wrapper scheme. However, to be able to split these function definitions into the required workers and wrappers we need to extend our worker/wrapper split algorithm. As basis we use Mycroft's extension of the Damas-Milner type inference algorithm by polymorphic recursion [Myc84].

## 2 The second-order typed language

We use a small functional language with second-order types, which is similar to the intermediate language Core used inside the Glasgow Haskell compiler [GHC]. The syntax is defined in Figure 1 and the type system in Figure 2. The language is essentially the second-order

$$
\begin{array}{rrll}
\text{Type constructors} & C & ::= & \texttt{[]} \mid \texttt{Int} \mid \ldots \\
\text{Type variables} & \alpha, \beta, \gamma & & \\
\text{Types} & \tau & ::= & C\,\overline{\tau} \mid \alpha \mid \tau_1 \to \tau_2 \mid \forall \alpha.\tau \\
\text{Term variables} & x, c & & \\
\text{Terms} & e & ::= & x \mid \lambda x : \tau.e \mid e_1\, e_2 \mid \texttt{case}\ e\ \texttt{of}\ \{c_i\,\overline{x}_i \to e_i\}_{i=1}^{k} \mid \\
 & & & \texttt{let}\ \{x_i : \tau_i = e_i\}_{i=1}^{k}\ \texttt{in}\ e \mid \lambda \alpha.e \mid e\,\tau
\end{array}
$$

Figure 1: Terms and types of the language

$$
\frac{}{\Gamma + x : \tau \vdash x : \tau}\ \text{VAR}
$$

$$
\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2}\ \text{TERM ABS} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau}\ \text{TERM APP}
$$

$$
\frac{\forall i = 1..k \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash e_i : \tau_i \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash e : \tau}{\Gamma \vdash \texttt{let}\ \{x_i : \tau_i = e_i\}_{i=1}^{k}\ \texttt{in}\ e : \tau}\ \text{LET}
$$

$$
\frac{\Gamma \vdash e : C\,\overline{\rho} \quad \Gamma(c_i) = \forall \overline{\alpha}.\overline{\rho}_i \to C\,\overline{\alpha} \quad \Gamma + \{\overline{x}_i : \overline{\rho}_i[\overline{\rho}/\overline{\alpha}]\} \vdash e_i : \tau \quad \forall i = 1..k}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto e_i\}_{i=1}^{k} : \tau}\ \text{CASE}
$$

$$
\frac{\Gamma \vdash e : \tau \quad \alpha \notin \text{freeTyVar}(\Gamma)}{\Gamma \vdash \lambda \alpha.e : \forall \alpha.\tau}\ \text{TYPE ABS} \qquad \frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e\,\rho : \tau[\rho/\alpha]}\ \text{TYPE APP}
$$

Figure 2: Type system

typed $\lambda$-calculus augmented with `let` for arbitrary mutual recursion and `case` for decomposition of algebraic data structures. We view a type environment $\Gamma$ as both a mapping from variables to types and a set of tuples $x : \tau$. The operator $+$ combines two type environments under the assumption that their domain is disjunct. We abbreviate $\Gamma + \{x : \tau\}$ by $\Gamma + x : \tau$. Data constructors $c$ are just special term variables. The language does not have explicit definitions of algebraic data types like `data` $C\,\overline{\alpha}$ `=` $c_1\,\overline{\tau}_1 \mid \ldots \mid c_k\,\overline{\tau}_k$. Such a definition is implicitly expressed by having the data constructors in the type environment: $\Gamma(c_i) = \tau_{1,i} \to \ldots \to \tau_{n_i,i} \to C\,\overline{\alpha} = \overline{\tau}_i \to C\,\overline{\alpha}$. Hence for the polymorphic type list, which we write `[`$\alpha$`]` instead of `[]` $\alpha$, we have $\Gamma(\texttt{(:)}) = \forall \alpha.\alpha \to \texttt{[}\alpha\texttt{]} \to \texttt{[}\alpha\texttt{]}$ and $\Gamma(\texttt{[]}) = \forall \alpha.\texttt{[}\alpha\texttt{]}$. The functions `foldr` and `build` are defined as follows

```
foldr : ∀α.∀β.(α → β → β) → β → [α] → β
      = λα. λβ. λc:α → β → β. λn:β. λxs:[α]. case xs of {
          []     → n
          y:ys → c y (foldr α β c n ys)}

build : ∀α.(∀β.(α → β → β) → β → β) → [α]
      = λα. λg:∀β.(α → β → β) → β → β. g [α] ((:) α) ([] α)
```

and the `foldr/build` rule takes the form:

$$\texttt{foldr}\ \tau_1\ \tau_2\ e_{(:)}\ e_{[]}\ (\texttt{build}\ \tau_1\ e_{\mathtt{p}}) \quad \rightsquigarrow \quad e_{\mathtt{p}}\ \tau_2\ e_{(:)}\ e_{[]}$$

# 3 List Abstraction through Type Inference

To understand the mode of operation of the list abstraction algorithm of [Chi99] we study an example. We have to start with the typing of the producer:[1]

$$\begin{array}{l}\{\texttt{map'}:\texttt{(Int}\to\texttt{Int)}\to\texttt{[Int]}\to\texttt{[Int]},\ \texttt{g}:\texttt{Int}\to\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int},\\ \ \texttt{(:)}:\forall\alpha.\alpha\to\texttt{[}\alpha\texttt{]}\to\texttt{[}\alpha\texttt{]},\ \texttt{[]}:\forall\alpha.\texttt{[}\alpha\texttt{]}\}\\ \vdash \texttt{map' g ((:) Int 1 ((:) Int 2 ([] Int))) : [Int]}\end{array}$$

The algorithm replaces every list constructor application `(:)` $\tau$, respectively `[]` $\tau$, by a different variable $c_i$, respectively $n_i$. To use the existing type annotations as far as possible, we just replace every list type[2] in the expression and in the type environment by a new type inference variable. Furthermore, we add $c_i : \tau \to \gamma_i \to \gamma_i$, respectively $n_i : \gamma_i$, to the type environment, where $\gamma_i$ is a new type inference variable for every variable $c_i$, respectively $n_i$.

$$\begin{array}{l}\{\texttt{map'}:\texttt{(Int}\to\texttt{Int)}\to\gamma_1\to\gamma_2,\ \texttt{g}:\texttt{Int}\to\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int},\\ \ \texttt{n}_1:\gamma_3,\ \texttt{c}_1:\texttt{Int}\to\gamma_4\to\gamma_4,\ \texttt{c}_2:\texttt{Int}\to\gamma_5\to\gamma_5\}\\ \vdash \texttt{map' g (c}_1\ \texttt{1 (c}_2\ \texttt{2 n}_1\texttt{)) : }\gamma\end{array}$$

This typing with type variables is the input to a modified version of the Damas-Milner type inference algorithm [DM82, LY98]. On the one hand the algorithm was extended to cope with explicit type abstraction and application. On the other hand the type generalisation step (type closure) at `let` bindings was dropped. The type inference algorithm replaces some of the type variables so that the typing is again derivable from the type inference rules, that is, the expression is well-typed in the type environment. Note that type inference cannot fail, because the typing we start with is derivable. We just try to find a more general typing.

$$\begin{array}{l}\{\texttt{map'}:\texttt{(Int}\to\texttt{Int)}\to\gamma_1\to\gamma,\ \texttt{g}:\texttt{Int}\to\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int},\\ \ \texttt{n}_1:\gamma_1,\ \texttt{c}_1:\texttt{Int}\to\gamma_1\to\gamma_1,\ \texttt{c}_2:\texttt{Int}\to\gamma_1\to\gamma_1\}\\ \vdash \texttt{map' g (c}_1\ \texttt{1 (c}_2\ \texttt{2 n}_1\texttt{)) : }\gamma\end{array}$$

The type of the expression is a type variable which can be abstracted, but this type variable also appears in the type of the function `map'`. So the the definition of `map'` has to be inlined, all lists types and list constructors be replaced by new variables and type inference be continued.

$$\begin{array}{l}\{\texttt{g}:\texttt{Int}\to\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int},\ \texttt{n}_1:\texttt{[Int]},\ \texttt{n}_2:\gamma,\\ \ \texttt{c}_1:\texttt{Int}\to\texttt{[Int]}\to\texttt{[Int]},\ \texttt{c}_2:\texttt{Int}\to\texttt{[Int]}\to\texttt{[Int]},\ \texttt{c}_3:\texttt{Int}\to\gamma\to\gamma\}\\ \vdash \texttt{let map' : (Int}\to\texttt{Int)}\to\texttt{[Int]}\to\gamma\\ \qquad\qquad \texttt{= }\lambda\texttt{f}:\texttt{Int}\to\texttt{Int.foldr Int }\gamma\ (\lambda\texttt{v}:\texttt{Int}.\,\lambda\texttt{w}:\gamma.\,\texttt{c}_3\ \texttt{(f\ \ v) w)\ n}_3\\ \quad \texttt{in map' g (c}_2\ \texttt{1 (c}_3\ \texttt{2 n}_2\texttt{)) : }\gamma\end{array}$$

[1] We only consider a monomorphic version of `map`. Polymorphic functions that may be inlined require an additional instantiation step which we skip here (see [Chi99], Section 4.2).

[2] We only need to replace the type `[Int]`, because it is the type to be abstracted from. We are not permitted to replace list types which contain locally bound type variables.

Now the type of the expression is still a type variable that, however, does not occur in the type environment except in the types of the $c_i$ and $n_i$. Hence the algorithm terminates successfully. The type environment tells us that $\mathtt{c}_1$ and $\mathtt{n}_1$ construct the result of the producer whereas $\mathtt{c}_2$, $\mathtt{c}_3$, and $\mathtt{n}_2$ have to construct lists that are internal to the producer. So the producer can be expressed with abstracted list type and list constructors as follows, suitable as argument for `build`:

```
λγ. λc:Int → γ → γ. λn:γ.
  let map' : (Int→Int)→[Int]→ γ
           = λf:Int→Int. foldr Int γ (λv:Int. λw:γ. c (f v) w) n
  in map' g ((:) Int 1 ((:) Int 2 ([] Int)))
```

# 4 The Worker/Wrapper Scheme

To apply the `foldr/build` rule we do not need to fully inspect the consumer and the producer. We only have to see that the consumer is in `foldr` form and the producer is in `build` form. The other arguments of `foldr` and the argument of `build` are of no interest, they are just rearranged by the transformation. This observation is the basis for obtaining producers in `build` form without inlining of large expressions.

## 4.1 Gill's Worker/Wrapper Scheme

Gill ([Gil96], Section 7.4) takes the definition of a list producer function from which the list constructors are abstracted

```
f x1 ... xn = build (\c n -> e)
```

and splits it into a definition of two functions, a worker `fW` and a wrapper `f`:

```
fW x1 ... xn c n = e
f x1 ... xn = build (fW x1 ... xn)
```

The wrapper just contains the `build` and a call to the worker. It is small enough to be inlined at all its call sites. For example, the definition

```
map f xs = build (\c n -> foldr (c . f) n xs)
```

is split up as follows:

```
mapW :: (a -> b) -> [a] -> (b -> c -> c) -> c -> c
mapW f xs c n = foldr (c . f) n xs

map f xs = build (mapW f xs)
```

Consider deforestation of the definition body of `any` that was given first in the introduction:

```
    or (map p xs)
⇝   foldr (||) True (build (mapW f xs))
⇝   mapW f xs (||) True
```

First `or` and `map` are inlined. Then fusion is performed by the `foldr/build` rule. Afterwards it is left to the standard inliner, if `mapW` is inlined. This will be the case for this simple example, but not in general. In any case the expression has been deforested.

## 4.2 The Limitations of `build`

The use of `build` unfortunately limits the scope of Gill's worker/wrapper scheme. In [Chi99] we showed that the type-inference based algorithm can abstract the list constructors from the producer `fst (unzip zs)`, where the function `unzip` maps a list of pairs to a pair of lists. To abstract the list the algorithm inlines the definition of `unzip`. The function `unzip` cannot be expressed in terms of `build`, because `build` can only wrap a producer that returns a single list. So if we replace the inlining of the type abstraction algorithm by using Gill's worker/wrapper scheme we loose the ability to fuse a consumer with `fst (unzip zs)`.

There is also a second problem which is related to our list abstraction algorithm. We still want to use it — without inlining — to abstract the intermediate list constructors from the producer, because not all producers will already be in `build` form after inlining of wrappers. However, a `build` in the producer hinders the type-inference algorithm. For example from the producer `build (mapW f xs)` no list constructors can be abstracted, because they are hidden by `build`. We have to inline `build` to proceed with list constructor abstraction.

So, because `build` is both inflexible and is not needed but even disturbs the abstraction of list constructors, we draw the obvious conclusion to do without it.

## 4.3 Short Cut Deforestation without `build`

The purpose of `build` is to express in Haskell that the producer has the required polymorphic type and to prevent this special form from being destroyed by another compiler optimisation. Because we perform our transformation in a second-order typed language with explicit type abstraction and application we can enforce the type requirement directly. Instead of transforming a producer into the form

$$\texttt{build}\ \tau\ (\lambda\beta.\,\lambda\texttt{c}{:}\tau \to \beta \to \beta.\,\lambda\texttt{n}{:}\beta.\,e')$$

we transform it into

$$(\lambda\beta.\,\lambda\texttt{c}{:}\tau \to \beta \to \beta.\,\lambda\texttt{n}{:}\beta.\,e')\ [\tau]\ ((:)\ \tau)\ ([]\ \tau)$$

The short cut fusion rule now looks as follows:

$$\text{for all } e_{(:)}{:}\tau \to \tau' \to \tau',\quad e_{[]}{:}\tau',\quad e_{\texttt{p}}{:}\forall\gamma.(\tau \to \gamma \to \gamma) \to \gamma \to \gamma$$
$$\texttt{foldr}\ \tau\ \tau'\ e_{(:)}\ e_{[]}\ (e_{\texttt{p}}\ [\tau]\ ((:)\ \tau)\ ([]\ \tau)) \quad \rightsquigarrow \quad e_{\texttt{p}}\ \tau'\ e_{(:)}\ e_{[]}$$

The `foldr/build` rule looks more simple, because `build` is easier to recognise than the producer form of the new rule. However, we neither have to construct nor to search for such a producer form. Our short cut deforestation algorithm searches for occurrences of `foldr`, abstracts the result list from the producer and then directly applies the short cut fusion rule. Therefore we also do not have to worry that another compiler optimisation might $\beta$-reduce the special producer form.

## 4.4 Wrappers without `build`

Similarly we do not use `build` in wrappers. As an example for our new worker/wrapper scheme consider the worker and wrapper for the function `map`:

$$\texttt{mapW} : \forall\alpha.\forall\beta.\forall\gamma.\,(\beta \to \gamma \to \gamma) \to \gamma \to\ (\alpha \to \beta) \to [\alpha] \to \gamma$$
$$= \lambda\alpha.\,\lambda\beta.\,\lambda\gamma.\,\lambda \texttt{c}{:}\beta \to \gamma \to \gamma.\,\lambda \texttt{n}{:}\gamma.\,\lambda \texttt{f}{:}\alpha \to \beta.$$
$$\texttt{foldr}\ \alpha\ \gamma\ (\lambda \texttt{v}{:}\alpha.\,\lambda \texttt{w}{:}\gamma.\ \texttt{c (f v) w)\ n}$$

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

The wrapper calls the worker directly with a list type and respective list constructors as arguments. The worker is almost identical to Gill's worker. Note that we insert the abstraction from list type and its respective list constructors between the type abstractions and the term abstractions. We cannot insert the new term abstractions in front of the original type abstractions, because the list type $[\beta]$, from which we abstract, contains a type variable $\beta$ which is bound in the type of the function. To insert the new abstractions before the original term abstractions has several minor advantages. First, we thus do not require that all term arguments are $\lambda$-abstracted at the top of the definition body, the definition may also use partial application. Second, the wrapper can be inlined and $\beta$-reduced at all call sites, even where it is only partially applied, because we can use partial application in the definition of the wrapper.

To see the gained expressiveness consider the function `unzip`, which produces two lists. The worker abstracts from both list types and their respective sets of list constructors.

$$\texttt{unzipW} : \forall\alpha.\forall\beta.\forall\gamma_1.\forall\gamma_2.(\alpha \to \gamma_1 \to \gamma_1) \to \gamma_1 \to (\beta \to \gamma_2 \to \gamma_2) \to \gamma_2 \to [(\alpha,\beta)] \to (\gamma_1,\gamma_2)$$
$$= \lambda\alpha.\,\lambda\beta.\,\lambda\gamma_1.\,\lambda\gamma_2.\,\lambda \texttt{c}_1{:}\alpha \to \gamma_1 \to \gamma_1.\,\lambda \texttt{n}_1{:}\gamma_1.\,\lambda \texttt{c}_2{:}\beta \to \gamma_2 \to \gamma_2.\,\lambda \texttt{n}_2{:}\gamma_2.$$
$$\texttt{foldr}\ (\alpha,\beta)\ (\gamma_1,\gamma_2)$$
$$(\lambda \texttt{y}{:}(\alpha,\beta).\,\lambda \texttt{u}{:}(\gamma_1,\gamma_2).\texttt{case y of}\,\{\texttt{(v,w)} \to \texttt{case u of}\,\{\texttt{(vs,ws)} \to$$
$$\texttt{(,)}\ \gamma_1\ \gamma_2\ (\texttt{c}_1\ \texttt{v vs})\ (\texttt{c}_2\ \texttt{w ws})\ \}\})$$
$$(\texttt{(,)}\ \gamma_1\ \gamma_2\ \texttt{n}_1\ \texttt{n}_2)$$

$$\texttt{unzip} : \forall\alpha.\forall\beta.[(\alpha,\beta)] \to ([\alpha],[\beta])$$
$$= \lambda\alpha.\lambda\beta.\ \texttt{unzipW}\ \alpha\ \beta\ [\alpha]\ [\beta]\ ((:)\ \alpha)\ ([]\ \alpha)\ ((:)\ \beta)\ ([]\ \beta)$$

## 4.5 Deforestation (nearly) without Inlining

The new wrappers transfer the information needed for the list abstraction algorithm. Let us reconsider the expression `fst` $[\tau_1]$ $[\tau_2]$ `(unzip` $\tau_1$ $\tau_2$ `zs)`. After the wrapper `unzip` is inlined

$$\texttt{fst}\ [\tau_1]\ [\tau_2]\ (\texttt{unzipW}\ \tau_1\ \tau_2\ [\tau_1]\ [\tau_2]\ ((:)\ \tau_1)\ ([]\ \tau_1)\ ((:)\ \tau_2)\ ([]\ \tau_2)\ \texttt{zs})$$

all result list types and constructors are exposed and the type-inference based algorithm can abstract them without any further inlining:

$$\lambda\gamma.\lambda \texttt{c}{:}\tau_1 \to \gamma \to \gamma.\lambda \texttt{n}{:}\gamma.\,\texttt{fst}\,\gamma\,[\tau_2]\,(\texttt{unzipW}\,\tau_1\,\tau_2\,\gamma\,[\tau_2]\,\texttt{c n}\,((:)\,\tau_2)\,([]\,\tau_2)\,\texttt{zs})$$

So fusion with a consumer `foldr` $\tau_1\ \tau_3\ e_{(:)}\ e_{[]}$ results in

$$(\lambda\gamma.\lambda \texttt{c}{:}\tau_1 \to \gamma \to \gamma.\lambda \texttt{n}{:}\gamma.\,\texttt{fst}\,\gamma\,[\tau_2]\,(\texttt{unzipW}\,\tau_1\,\tau_2\,\gamma\,[\tau_2]\,\texttt{c n}\,((:)\,\tau_2)\,([]\,\tau_2)\,\texttt{zs}))\,\tau_3\,e_{(:)}\,e_{[]}$$

## 4.6 More Wrappers

The list append function `(++)` is notorious for being difficult to fuse with. The expression `(++)` $\tau$ `xs ys` does not produce the whole result list itself, because only `xs` is copied but not `ys`. Therefore `(++)` cannot be defined in terms of `build`. Gill defines a further second-order typed function `augment` to solve this problem. However, we can easily define a worker for `(++)` by abstracting not just from the result list but simultaneously from the type of the second argument:

```
appW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ → γ
     = λα. λγ. λc:α → γ → γ. λn:γ. λxs:[α]. λys:γ.
         foldr α γ c ys xs

(++) : ∀α.[α] → [α] → [α]
     = λα. appW α [α] ((:) α) ([] α)
```

The type of `appW` implies, that we can only abstract from the result list constructors of an application of `(++)`, if we can abstract from the result list constructors of its second argument. We believe that this will seldom restrict deforestation in practise. For example the definition

```
concat : ∀α.[[α]] → [α]
       = λα. foldr [α] [α] ((++) α) ([] α)
```

can be split into worker and wrapper thanks to the wrapper `appW`:

```
concatW : ∀α.∀γ.(α → γ → γ) → γ → [[α]] → γ
        = λα. λγ. λc:α → γ → γ. λn:γ. foldr [α] γ (appW α γ c n) n

concat : ∀α.[[α]] → [α]
       = λα. concatW α [α] ((:) α) ([] α)
```

None of the previous function definitions was recursive. The recursion was hidden by `foldr`. The recursively defined function `tails` returns the list of all final segments of xs, longest first:

```
tails : ∀α.[α] → [[α]]
      = λα. λxs:[α]. case xs of {
          []   → (:) [α] ([] α) ([] [α])
          y:ys → (:) [α] xs (tails α ys) }
```

A list can also be abstracted from this recursive definition:

```
tailsW : ∀α.∀γ.[α] → γ
       = λα. λγ. λc:[α] → γ → γ. λn:γ. λxs:[α]. case xs of {
           []   → c ([] α) n
           y:ys → c xs (tailsW α γ c n ys) }

tails : ∀α.[α] → [[α]]
      = λα. tailsW α [[α]] ((:) [α]) ([] [α])
```

Note that to abstract the list the recursive calls in the definition must be to the worker itself. It is not possible to abstract the list first and to inline calls to the wrapper in the definition body later.

### 4.7 Effects on Performance

As Gill already noticed for his worker/wrapper scheme, there is a substantial performance difference between calling a function as originally defined (`map` $\tau'$ $\tau$) and calling the worker with list constructors as arguments (`mapW` $\tau'$ $\tau$ `[`$\tau$`]` `((:)` $\tau$`)` `([]` $\tau$`)`). Constructing a list with list constructors that are passed as arguments is more expensive than constructing the list directly. After deforestation all calls of workers that were not needed still have list constructors as arguments. So, as Gill suggested, we must have for each worker a version which is specialised to the list constructors and replace the call of each unused worker by a call of its specialised version. We could use the original, unsplit definition of the function, but by specialising the worker we can profit from any optimisations, especially deforestation, which were performed inside the definition of the worker. Note that we only derive one specialised definition for every worker definition.

The worker/wrapper scheme increases code size through the introduction of wrappers and specialised workers. However, this increase is clearly limited in contrast to the code increase that is caused by our original list abstraction algorithm with inlining. An implementation will show if the code size increase is acceptable. Note that the definition of workers which are not needed for deforestation can be removed by standard dead code elimination after worker specialisation has been performed.

## 5 The Worker/Wrapper Split Algorithm

For the worker/wrapper scheme each list-producing function definition has to be split into a worker and a wrapper definition.

### 5.1 Derivation of Workers through Type Inference

A worker is easily derived from a non-recursive function definition by application of the list abstraction algorithm. Consider the `foldr` definition of `map`. Only the preceding type abstractions have to be removed to form the input for the list abstraction algorithm:

$$\{\texttt{foldr}:\forall\alpha.\forall\beta.(\alpha\to\beta\to\beta)\to\beta\to[\alpha]\to\beta,\ \texttt{(:)}:\forall\alpha.\alpha\to[\alpha]\to[\alpha],\ \texttt{[]}:\forall\alpha.[\alpha]\}$$
$$\vdash \lambda\texttt{f}:\alpha\to\beta.\,\texttt{foldr}\ \alpha\ [\beta]\ (\lambda\texttt{v}:\alpha.\lambda\texttt{w}:[\beta].\texttt{(:)}\ \beta\ \texttt{(f v) w)}\ \texttt{([]}\ \beta\texttt{)}$$

The algorithm returns:

$$\lambda\gamma.\,\lambda\texttt{c}:\beta\to\gamma\to\gamma.\,\lambda\texttt{n}:\gamma.\,\lambda\texttt{f}:\alpha\to\beta.\,\texttt{foldr}\ \alpha\ \gamma\ \texttt{(c (f v) w) n}$$

So the result list can be abstracted. The readdition of the abstraction from $\alpha$ to obtain the worker and the construction of the wrapper is straightforward. In the case that no list can be abstracted, no worker/wrapper split takes place.

Because all list types in the the type of the processed function are replaced by type variables — not only the list types in the result — also the worker of `(++)` is derived by this algorithm. The derivation of the workers `unzipW` and `concatW` poses no problem either.

### 5.2 Derivation of Workers of Recursively Defined Functions

For recursive definitions we have to modify the list abstraction algorithm slightly. Consider the definition of `tails` given in Section 4.6. The input typing for the type inference algorithm

must contain a type assignment for the recursive call(s). Because we remove the abstraction from the type variable $\alpha$, we have to replace `tails` $\alpha$ by a new identifier `tails'`. The type environment assigns the same type to this identifier as is given for the whole definition body. The latter corresponds to the processing of recursive `let`s in the Damas-Milner type inference algorithm.

$$\begin{array}{l}\{\texttt{tails'} : \gamma_1 \to \gamma_2,\ \texttt{n}_1 : \gamma_3,\ \texttt{n}_2 : \gamma_4,\ \texttt{c}_1 : \gamma_5 \to \gamma_6 \to \gamma_6,\ \texttt{c}_2 : \gamma_7 \to \gamma_8 \to \gamma_8\} \\ \vdash \lambda\texttt{xs}:[\alpha].\ \texttt{case xs of } \{ \\ \qquad\qquad \texttt{[]} \quad \to \texttt{c}_1\ \texttt{n}_1\ \texttt{n}_2 \\ \qquad\qquad \texttt{y:ys} \to \texttt{c}_2\ \texttt{xs (tails' ys) } \} \ :\ \gamma_1 \to \gamma_2\end{array}$$

Type inference yields:

$$\begin{array}{l}\{\texttt{tails'} : [\alpha] \to \gamma,\ \texttt{n}_1 : [\alpha],\ \texttt{n}_2 : \gamma,\ \texttt{c}_1 : [\alpha] \to \gamma \to \gamma,\ \texttt{c}_2 : [\alpha] \to \gamma \to \gamma\} \\ \vdash \lambda\texttt{xs}:[\alpha].\ \texttt{case xs of } \{ \\ \qquad\qquad \texttt{[]} \quad \to \texttt{c}_1\ \texttt{n}_1\ \texttt{n}_2 \\ \qquad\qquad \texttt{y:ys} \to \texttt{c}_2\ \texttt{xs (tails' ys) } \} \ :\ [\alpha] \to \gamma\end{array}$$

The remaining construction of the worker and wrapper is again straightforward.

For the worker/wrapper split of several mutually recursive definitions the type inference algorithm processes all definitions together.

### 5.3 Traversal Order

The worker/wrapper split algorithm splits each `let` defined function individually. The example of `concat` in Section 4.6 shows us, that the list could only be abstracted after the wrapper of `(++)` had been inlined. Hence the split algorithm must traverse the program in top-down order and inline wrappers in the remaining program directly after they were derived.

Additionally, definitions can be nested, that is, the right-hand-side of a `let` binding can contain another `let` binding. Here the inner definition has to be split first. Its wrapper can then be inlined in the body of the outer definition and thus enable the abstraction of more lists from the outer definition.

## 6 Functions that Consume their own Result

There are list functions which consume their own result. The most simple example is the definition of the function that reverses a list in quadratic time:

$$\begin{array}{l}\texttt{reverse} : \forall\alpha.[\alpha] \to [\alpha] \\ \qquad\quad = \lambda\alpha.\ \lambda\texttt{xs}:[\alpha]\,.\ \texttt{case xs of } \{ \\ \qquad\qquad \texttt{[]} \quad \to \texttt{[]}\ \alpha \\ \qquad\qquad \texttt{y:ys} \to \texttt{(++)}\ \alpha\ \texttt{(reverse}\ \alpha\ \texttt{ys)}\ \texttt{((:)}\ \alpha\ \texttt{y ([]}\ \alpha\texttt{))}\ \}\end{array}$$

This definition can be split into the following worker and wrapper:

$$\begin{array}{l}\texttt{reverseW} : \forall\alpha.\forall\gamma.(\alpha \to \gamma \to \gamma) \to \gamma \to [\alpha] \to \gamma \\ \qquad\quad = \lambda\alpha.\ \lambda\gamma.\ \lambda\texttt{c}:\alpha \to \gamma \to \gamma.\ \lambda\texttt{n}:\gamma.\ \lambda\texttt{xs}:[\alpha]\,.\ \texttt{case xs of } \{ \\ \qquad\qquad \texttt{[]} \quad \to \texttt{n} \\ \qquad\qquad \texttt{y:ys} \to \texttt{appW}\ \alpha\ \gamma\ \texttt{c n (reverseW}\,\alpha\,[\alpha]\,\texttt{((:)}\,\alpha\texttt{)}\,\texttt{([]}\,\alpha\texttt{)}\,\texttt{ys)}\ \texttt{(c y n)}\ \}\end{array}$$

```
reverse : ∀α.[α] → [α]
        = λα.reverseW α [α] ((:) α) ([] α)
```

In the definition of `reverseW` the worker `appW` can be inlined. Then short cut fusion yields:

```
reverseW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
         = λα. λγ. λc:α → γ → γ. λn:γ. λxs:[α]. case xs of {
             []   → n
             y:ys → reverseW α γ c (c y n) ys }
```

The deforested version performs list reversal in linear time. The worker argument for the abstraction from the list constructor `[]` is used as an accumulator.

The list abstraction algorithm with inlining cannot achieve this transformation of the quadratic version into the linear version. To abstract the intermediate list, that algorithm would inline the definition of `reverse`. Then the intermediate list would be eliminated successfully, but the inlined definition of `reverse` would contain a new starting point for deforestation which would lead to new inlining of `reverse` ... The quadratic version creates at run time an intermediate list between each recursive call. To remove all these intermediate lists through a finite amount of transformation the worker/wrapper scheme is required.

## 6.1 Worker Derivation with Polymorphic Recursion

Unfortunately, the worker `reverseW` cannot be derived by the algorithm described in Section 5. Compare the recursive definition of `reverseW` with the recursive definition of `tailsW`. The former is polymorphically recursive, that is, a recursive call uses type arguments different from the abstracted type variables. Obviously, functions that consume their own result need such polymorphically recursive workers.

Typability in the Damas-Milner type system with polymorphic recursion is semi-decidable, that is, there are algorithms which do infer the most general type of an expression within the Damas-Milner type system with polymorphic recursion if it is typable and may diverge otherwise. Fortunately, the input of the worker/wrapper split algorithm is typable, we only try to find a more general type than we have.

To derive a possibly polymorphically recursive worker definition, we build on Mycroft's extension of the Damas-Milner type inference algorithm [Myc84]. We start with the most general worker type possible, which is obtained from the original type by replacing every list type by a new type variable and abstracting from it and respective list constructors.

```
{reverseW : ∀α.∀γ1.∀γ2.(α → γ1 → γ1) → γ1 → (α → γ2 → γ2) → γ2 → (γ1 → γ2),
 n1 : γ3, n2 : γ4, n3 : γ5, n4 : γ6, n5 : γ7,
 c1 : α → γ8 → γ8, c2 : α → γ9 → γ9, c3 : α → γ10 → γ10, c4 : α → γ11 → γ11}
⊢ λxs:γ12. case xs of {
    []   → n1
    y:ys → appW α γ13 c1 n2 (reverseW α γ14 γ15 c2 n3 c3 n4 ys) (c4 y n5) }
: γ16
```

We perform type inference to obtain a first approximation of the type of the worker:

$\{\texttt{reverseW} : \forall\alpha.\forall\gamma_1.\forall\gamma_2.(\alpha\to\gamma_1\to\gamma_1)\to\gamma_1\to(\alpha\to\gamma_2\to\gamma_2)\to\gamma_2\to(\gamma_1\to\gamma_2),$
$\texttt{n}_1:\gamma,\ \texttt{n}_2:\gamma,\ \texttt{n}_3:[\alpha],\ \texttt{n}_4:[\alpha],\ \texttt{n}_5:\gamma,$
$\texttt{c}_1:\alpha\to\gamma\to\gamma,\ \texttt{c}_2:\alpha\to[\alpha]\to[\alpha],\ \texttt{c}_3:\alpha\to[\alpha]\to[\alpha],\ \texttt{c}_4:\alpha\to\gamma\to\gamma\}$
$\vdash \lambda\texttt{xs}:[\alpha].\ \texttt{case xs of}\ \{$
$\quad [] \to \texttt{n}_1$
$\quad \texttt{y:ys} \to \texttt{appW}\ \alpha\ \gamma\ \texttt{c}_1\ \texttt{n}_2\ (\texttt{reverseW}\ \alpha\ [\alpha]\ [\alpha]\ \texttt{c}_2\ \texttt{n}_3\ \texttt{c}_3\ \texttt{n}_4\ \texttt{ys})\ (\texttt{c}_4\ \texttt{y}\ \texttt{n}_5)\ \}$
$: [\alpha]\to\gamma$

Subsequently we infer anew the type of the definiton body, this time under the assumption that `reverseW` has the type $\forall\alpha.\forall\gamma.(\alpha\to\gamma\to\gamma)\to\gamma\to[\alpha]\to\gamma$, the result of the first type inference. This process iterates until the inferred type is stable, that is input and output type are identical. For our example the second iteration already shows that the result of the first iteration is correct. In general, worker derivation stops latest after $n+1$ iterations, where $n$ is the number of list types in the type of the original function.

## 6.2 Further Workers with Polymorphic Recursion

Similar to the example `reverse` are definitions of functions which traverse a tree to collect all node entries in a list. A straightforward quadratic time definition can be split into a polymorphically recursive worker and a wrapper and then be deforested to obtain a linear time definition which uses an accumulating argument.

A different, fascinating example is the definition of the function `inits`, which determines the list of initial segments of a list with the shortest first.

```
inits: ∀α.[α] → [[α]]
     = λα. λxs:[α].
         case xs of {
           []   → (:) [α] ([] α) ([] [α])
           y:ys → (:) [α] ([] α) (map [α] [α] ((:) α y) (inits α ys)) }
```

It is split into the following polymorphically recursive worker and wrapper:

```
initsW: ∀α.∀γ1.∀γ2.(α → γ1 → γ1) → γ1 → (γ1 → γ2 → γ2) → γ2 → [α] → γ2
      = λα. λγ1. λγ2. λc1:α → γ1 → γ1. λn1:γ1. λc2:γ1 → γ2 → γ2. λn2:γ2. λxs:[α].
          case xs of {
            []   → c2 n1 n2
            y:ys → c2 n1 (mapW γ1 γ1 γ2 c2 n2 (c1 y)
                           (initsW α γ1 [γ1] c1 n1 ((:) γ1) ([] γ1) ys)) }

inits: ∀α.[α] → [[α]]
     = λα. initsW α [α] [[α]] ((:) α) ([] α) ((:) [α]) ([] [α])
```

Note the abstraction from both (nested) result lists, which cannot be expressed with `build`. Fusion can be performed in the definition body of `initsW`:

```
initsW: ∀α.∀γ1.∀γ2.(α → γ1 → γ1) → γ1 → (γ1 → γ2 → γ2) → γ2 → [α] → γ2
      = λα. λγ1. λγ2. λc1:α → γ1 → γ1. λn1:γ1. λc2:γ1 → γ2 → γ2. λn2:γ2. λxs:[α].
          case xs of {
            []   → c2 n1 n2
            y:ys → c2 n1 (initsW α γ1 γ2 (λv:γ1. λw:γ2. c2 (c1 y v) w) n2 ys)}
```

The $n$-queens function as defined in [Gil96], Section 5.1, is another example in the same spirit.

### 6.3 Inaccessible Recursive Arguments

Unfortunately, a function may consume its own result but not be defined recursively. For example, the function `reverse` should actually be defined in terms of `foldr`, to enable short cut deforestation with `reverse` as consumer.

```
reverse: ∀α.[α] → [α]
       = λα.foldr α [α] (λy:α.λr:[α].(++) α r ((:) α y ([] α))) ([] α)
```

Here type inference with polymorphic recursion cannot help. To enable list abstraction we rewrite the definition as follows:

```
reverse: ∀α.[α] → [α]
       = λα.foldr α [α]
           (λy:α.λr:(α→[α]→[α])→[α]→[α].
             (++) α (r((:)α)([]α)) ((:)α y ([]α)))
           (λc:α → [α] → [α].λn:[α].n)
           ((:) α)
           ([] α)
```

It is, however, unclear when and how such a lifting of the result type of a function that encapsulates recursion can be done in general.

### 6.4 Deforestation Changes Complexity

Deforestation of the definition of `reverse` changes its complexity from quadratic to linear time. In case of the definition of `inits`, the change of complexity is more subtle. Both the original definition and the deforested definition take quadratic time to produce their complete result. However, to produce only the outer list of the result, with computation of the list elements still suspended, the original definition still takes quadratic time whereas the deforested version only needs linear time.

A polymorphically recursive worker will nearly always enable deforestation which changes the asymptotic time complexity of a function definition. This power is, however, a double-edged sword. A small syntactic change of a program (cf. previous subsection) may cause deforestation to be no longer applicable, and thus change the asymptotic complexity of the program. It can hence be argued that such far-reaching modifications should be left to the programmer.

## 7 Summary and Future Work

In this paper we presented an expressive worker/wrapper scheme to perform short cut deforestation (nearly) without inlining. An algorithm which is based on our type abstraction algorithm [Chi99] splits all definitions of list-producing functions of a program into workers and wrappers. The wrappers are small enough to be inlined unconditionally everywhere, also across module boundaries. They transfer the information needed for type abstraction in the split algorithm and the actual deforestation algorithm.

The actual deforestation algorithm itself searches for occurrences of `foldr`, abstracts the result list from the producer and then directly applies the short cut fusion rule. Further optimisations may be obtained by a subsequent standard inlining pass.

The deforestation algorithm is separate from the worker/wrapper split algorithm. The algorithms may be integrated, but we intend to perform deforestation after worker/wrapper splitting. Because deforestation and other optimisations may lead to new deforestation opportunities, it is useful to repeat deforestation several times.

Finally, we studied functions which consume their own result. Their definitions can automatically be deforested if the split algorithm is extended on the basis of Mycroft's extension of Damas-Milner type inference to polymorphic recursion. Nonetheless they still raise interesting questions.

**A Worker/Wrapper Scheme for Consumers** We focused on how to derive a producer for short cut deforestation without requiring large-scale inlining. Dually the consumer must be a `foldr` and hence sufficient inlining must be performed in the consumer to expose the `foldr`. If the arguments of the `foldr` are large expressions, the standard inliner will refuse to inline the `foldr` expression. So it seems reasonable to also split consumers into `foldr` wrappers and separate workers for the arguments of `foldr`. This transformation, however, does not require any (possibly type-based) analysis but can be performed directly on the syntactic structure.

**Other Data Types** The wrapper/worker scheme and the type-inference based derivation of workers is not specific to lists but can be used for arbitrary algebraic data types. In fact, a single worker can abstract from the data constructors of several different data types simultaneously and type inference can derive all abstractions simultaneously as well. For fusion we additionally need a catamorphism, that is a `foldr`, for the intermediate data type. This catamorphism could be provided by the user through a compiler directive or be inferred automatically by other algorithms.

**Integration with Type Inference Algorithm** The worker/wrapper split algorithm is not as efficient as it could be. The list abstraction algorithm traverses a whole definition body once. Even if we ignore polymorphic recursion, if $n$ `let` bindings are nested, then the body of the inner definition is traversed $n$ times.

However, as stated in Section 2, the list abstraction algorithm uses a modified version of the Damas-Milner type inference algorithm. The abstraction from type variables that replace list types corresponds to the generalisation step of the Damas-Milner algorithm. The list abstraction algorithm just additionally abstracts from term variables that replace list constructors and inserts both type and term abstractions into the program. The Damas-Milner algorithm recursively traverses a program only once. So we plan to integrate explicit type and term abstraction at `let` bindings into this type inference algorithm to obtain a single pass split algorithm. To deal with polymorphic recursion as well, the type inference algorithm of Emms and Leiß, which integrates semiunification into the Damas-Milner algorithm, may provide a good basis [EL99].

**Implementation** We have a working prototype of the list abstraction algorithm with inlining. On this basis we are implementing a simple worker/wrapper split algorithm. The final goal is an implementation in the Glasgow Haskell compiler to apply type-inference based short cut deforestation to real-world programs.

## Acknowledgements

I thank Simon Peyton Jones for several comments that inspired this paper. Especially, he drew my attention to producers that consume their own result.

## References

[Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. In *ICFP'99, International Conference on Functional Programming*. ACM Press, 1999.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, January 1982.

[EL99] Martin Emms and Hans Leiß. Extending the type checker of Standard ML by polymorphic recursion. *Theoretical Computer Science*, 212(1–2):157–181, February 1999.

[GHC] The Glasgow Haskell compiler. Availabel from `http://research.microsoft.com/users/t-simonm/ghc/`.

[Gil96] Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, 1996.

[GLP93] 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.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

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

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, LNCS 167, pages 217–228, Toulouse, France, April 1984. Springer.

[PH+99] Simon L. Peyton Jones, John Hughes, et al. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, February 1999.

[PM99] Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell compiler inliner. Submitted to IDL'99, July 1999.