Type Inference Builds a Short Cut to Deforestation

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. 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. Warm fusion was proposed to automatically transform functions into this form. Unfortunately, it is costly and hard to implement. 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. Typability for the second-order typed λ-calculus is undecidable. However, we present a linear-time algorithm that solves a partial type inference problem and that, together with controlled inlining and polymorphic type instantiation, suffices for deforestation. The resulting new short cut deforestation algorithm is efficient and removes more intermediate lists than the original.

1 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)
```


1999 ACM SIGPLAN International Conference on Functional Programming (ICFP '99)

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 `(||)`.

John Hughes expounds in his influential paper [Hug89] the relevance of the obtained modularity and points out that lazy evaluation makes this programming style practicable. In contrast to eager evaluation, 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. Thus the function `any` runs in constant space. Furthermore, when `or` comes across the value `True`, the production of the list is aborted.

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.

```
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*. Although there is an extensive literature on various deforestation methods, their implementation in real compilers proved to be difficult (see Section 7).

1.1 Short Cut Deforestation

In response to these problems short cut deforestation was developed [GLP93, Gil96], which is based on the single, local transformation

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

where `build` is a new function with a second-order type:

```
build :: ∀α.(∀γ.(α -> γ -> γ) -> γ -> γ) -> [α]
build g = g (:) []
```

Short cut deforestation removes an intermediate list that is produced by the expression `build` g and consumed by the function `foldr`. Lists are the most common intermediate data structures in functional programs. The compiler writer defines all list-manipulating functions in the standard library, which are used extensively by programmers, in terms of `build` and `foldr`. For example the definition of `map` is:

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

Gill implemented short cut deforestation in the Glasgow Haskell compiler [GHC] and measured speed ups of 43 % for the 10 queens program and of 3 % on average for a large number of programs that were programmed without deforestation in mind [Gil96].

1.2 Warm Fusion

Originally, the second-order type of `build` confined deforestation to producers that are defined in terms of list-producing functions from the standard library. Today, the Glasgow Haskell compiler has an extended type system which permits the programmer to use functions like `build`. However, asking the programmer to supply list producers in `build` form runs contrary to the aim of writing clear and concise programs. Whereas using `foldr` for defining list consumers is generally considered as good, modular programming style, `build` is only a crutch to enable deforestation.

Hence warm fusion [LS95] was developed to automatically derive a `build`/`foldr` form from generally recursive definitions. Its basic idea for transforming an arbitrary list producer e into `build` form is to rewrite it as

```
build (\c n -> foldr c n e)
```

and push the `foldr` into the e by term rewriting, hoping that it cancels with `build`s in there. When it does, the expression is in `build` form. Otherwise the transformation is abandoned. This method is rather expensive and poses substantial problems for an implementation [NP98].

1.3 Derivation of Build through Type Inference

In this paper we present a different, both efficient and powerful method for transforming an arbitrary list-producing expression into `build` form which can then be fused with a `foldr` consumer.

We were inspired by the fact that short cut deforestation is based on a parametricity theorem of the second-order typed λ-calculus. So we reduce the problem of deriving a `build` form of the producer to a type inference problem. At first, the fact that typability for the second-order typed λ-calculus is undecidable [Wel94] seems to pose a problem, but we have developed a linear-time *partial* type inference algorithm that makes good use of the type annotations in a second-order typed program and that suffices for our purposes.

The type inference algorithm indicates clearly where inlining is needed for deforestation. Hence in contrast to warm fusion we only need to derive a `build` form where necessary. Thus we reduce transformation time and avoid the potential inefficiency of `build` forms.

We formally present our improved short cut deforestation for a small functional language with second-order types, which is similar to the intermediate language Core used inside the Glasgow Haskell compiler.

The paper is structured as follows. In the next section we informally present our idea of using type inference for adapting a producer for short cut deforestation. In Section 3 we present a type inference algorithm for a simply typed functional language and demonstrate how our method works. Furthermore, we discuss inlining of definitions of list functions used by the producer. In Section 4 we extend our type inference algorithm to a second-order typed language and present the whole `build` derivation algorithm. We see in particular that we need to instantiate some polymorphic functions. Afterwards we discuss in Section 5 how an entire program is deforested. In Section 6 we present two extensions of our deforestation method. In Section 7 we compare our approach to related ones and we conclude in Section 8.

2 The Idea

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`:[1]

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

Hence, if `foldr` just replaces all list constructors in a list that is produced by an expression e, an eye-catching optimisation is to replace all list constructors in e already at compile time:

$$\texttt{foldr } e_{(:)}\ e_{[]}\ e \quad \rightsquigarrow \quad e\,[e_{(:)}/\texttt{(:)}][e_{[]}/\texttt{[]}]$$

Unfortunately this rule is wrong:

```
   foldr (||) False (map p [1,2])
≠  map p (1 || (2 || False))
```

In this example the constructors in the definition of `map`, which is not part of the expression, needed to be replaced and not those in `[1,2]`.

To solve the problem we distinguish between the constructors that build the intermediate list and all the other list constructors. Let e' be the producer e where the former constructors are replaced by variables `c` and `n`. Let the new transformation rule be

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

We observe that according to definition $e'\,[\texttt{(:)}/\texttt{c}][\texttt{[]}/\texttt{n}] = e$ and that generally $e_{(:)}$ and $e_{[]}$ have different types from `(:)` and `[]`. Hence just for this transformation to be type correct, e' must be polymorphic, that is, if e has type $[\tau]$, then e' `::` γ must hold under the assumption `c ::` τ `->` γ `->` γ and `n ::` γ for some type variable γ.

To express these type conditions in the transformation rule, the function `build` is introduced:

```
    foldr e(:) e[] (build (\c n -> e'))
⇝   (\c n -> e') e(:) e[]

build :: ∀α.(∀γ.(α -> γ -> γ) -> γ -> γ) -> [α]
build g = g (:) []
```

The function `build` has the additional effect that computing the substitution $[e_{(:)}/\texttt{c}][e_{[]}/\texttt{n}]$ is factored out and short cut deforestation is hence a very simple transformation.

Strikingly, the polymorphic type of e' also guarantees the semantic correctness of the transformation. Intuitively, e' can only manufacture its value of type γ from `c` and `n`, because only these have the right types. Formally, the transformation is an instance of a parametricity or free theorem [Wad89, Pit98]. The validity of the `foldr`/`build` rule is proved in [GLP93, Gil96].

There is a straightforward method to obtain e' from e. Let us replace in e the constructor `(:)` at some positions

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

$$
\begin{array}{rcll}
\text{Type constants} & T & ::= & \texttt{[Int]}, \texttt{Int}, \ldots \\
\text{Type inference variables} & \gamma, \delta, \delta_1, \delta_2, \ldots & & \\
\text{Types} & \tau, \rho & ::= & T \mid \beta \mid \tau \to \rho \\
\text{Term variables} & x, c & & \\
\text{Terms} & e & ::= & x \mid \lambda x : \tau . e \mid e_1\, e_2 \mid \texttt{case}\ e\ \texttt{of}\ \{c_i\, \overline{x}_i \to e_i\}_{i=1}^{k} \mid \texttt{let}\ \{x_i : \tau_i = e_i\}_{i=1}^{k}\ \texttt{in}\ e
\end{array}
$$

Figure 1: Terms and types of the simply typed language

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

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

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

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

Figure 2: Type system of the simply typed language

by `c` and the constructor `[]` at some positions by `n`. Next we type check the expression. If it has type γ, then we have found e'. 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 generate-and-test approach is prohibitively expensive. Fortunately, we can determine the list constructors that need to be replaced in one pass, if we use an algorithm that infers a principal typing. We just replace every occurrence of `(:)` by a new variable c_i and every occurrence of `[]` by a new variable n_i. If the principal type of the modified producer is a type variable γ, then deforestation is possible. Subsequently we just have to look at the types of the c_i and n_i in the typing. Those with types τ `->` `[`τ`]` `->` `[`τ`]`, respectively `[`τ`]`, are turned back into list constructors. Those with types τ `->` γ `->` γ, respectively γ, are replaced by `c` and `n`. We obtain e' and thus have derived the desired form `build (\c n ->` e'`)` of the list producer e.

3 Abstraction of List Constructors in the Simply Typed Language

The syntax of our simply typed language is defined in Figure 1 and the type system is given in Figure 2. The language is essentially the simply typed λ-calculus augmented with `let` for arbitrary mutual recursion and `case` for decomposition of algebraic data structures. We view data constructors c just as special term variables. We introduce type inference variables only for using them in the type inference algorithm. Nonetheless they may be used inside types like type constants. Because we have a monomorphic language, we only have lists with elements of type `Int`. We view a type environment Γ 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 $\Gamma + x : \tau$. The language does not have explicit definitions of algebraic data types like `data` T `=`$c_1\, \overline{\tau}_1 \mid \ldots \mid c_k\, \overline{\tau}_k$. Such a definition is implicitly expressed by having the data constructors in the type environment: $\Gamma(c_i) = \tau_{1,i} \to \ldots \to \tau_{n_i,i} \to T\ = \overline{\tau}_i \to T$.

We demand that every variable binding construct binds a different term variable. This condition avoids many complications with scope rules and is usually enforced inside compilers by a preliminary renaming pass.

3.1 Type Inference

Type inference algorithms for the simply typed λ-calculus usually take an untyped λ-term as input. However, the input to deforestation is already a typed program and we want to take advantage of this property. Hence we base our type inference algorithm on the algorithm $\mathcal{M}$ [LY98], which is for the Hindley-Milner type system. Algorithm $\mathcal{M}$ takes a type environment Γ, an expression e and a type τ as input, all of which may contain type variables, and returns a principal typing for (Γ, e, τ). A type substitution σ is a *typing* for (Γ, e, τ) iff $\Gamma\sigma \vdash e\sigma : \tau\sigma$. Furthermore, $\tilde{\sigma}$ is a *principal typing* for (Γ, e, τ) iff it is a typing for (Γ, e, τ) and for every typing σ' for (Γ, e, τ) there exists a substitution $\hat{\sigma}$ such that $\Gamma\sigma' = \Gamma\tilde{\sigma}\hat{\sigma}$, $e\sigma' = e\tilde{\sigma}\hat{\sigma}$ and $\tau\sigma' = \tau\tilde{\sigma}\hat{\sigma}$. Our type inference algorithm $\mathcal{T}$ is given in Figure 3. It has an initial substitution as additional argument. We say that a substitution σ' is an extension of a substitution σ, written $\sigma \preceq \sigma'$, if there exists a substitution $\hat{\sigma}$ such that $\sigma' = \sigma\hat{\sigma}$. We say that $\tilde{\sigma}$ is a *principal typing with respect to* σ for (Γ, e, τ) iff it is a typing for (Γ, e, τ) and for every typing σ' for (Γ, e, τ) with $\sigma \preceq \sigma'$ there exists a substitution $\hat{\sigma}$ such that $\Gamma\sigma' = \Gamma\tilde{\sigma}\hat{\sigma}$, $e\sigma' = e\tilde{\sigma}\hat{\sigma}$ and $\tau\sigma' = \tau\tilde{\sigma}\hat{\sigma}$.

Theorem 3.1 *Let Γ be an environment, e an expression and τ a type. If $(\Gamma\sigma, e\sigma, \tau\sigma)$ has a typing, then $\mathcal{T}(\Gamma, e, \tau, \sigma)$*

$$
\begin{aligned}
\mathcal{T}(\Gamma, x, \tau, \sigma) &= \mathcal{U}(\Gamma(x) \doteq \tau, \sigma) \\
\mathcal{T}(\Gamma, \lambda(x:\tau_1).e, \tau, \sigma) &= \mathcal{T}(\Gamma + x:\tau_1, e, \delta, \mathcal{U}(\tau_1 \to \delta \doteq \tau, \sigma)) \\
&\quad \text{where } \delta \text{ a new type inference variable} \\
\mathcal{T}(\Gamma, e_1\, e_2, \tau, \sigma) &= \mathcal{T}(\Gamma, e_2, \delta, \mathcal{T}(\Gamma, e_1, \delta \to \tau, \sigma)) \\
&\quad \text{where } \delta \text{ a new type inference variable} \\
\mathcal{T}(\Gamma, \texttt{let } \{x_i:\tau_i = e_i\}_{i=1}^k \texttt{ in } e, \tau, \sigma) &= \sigma_k \\
&\quad \text{where} \\
&\quad \sigma_0 = \mathcal{T}(\Gamma + \{x_i:\tau_i\}, e, \tau, \sigma) \\
&\quad \sigma_1 = \mathcal{T}(\Gamma + \{x_i:\tau_i\}, e_1, \tau_1, \sigma_0) \\
&\quad \vdots \qquad\quad \vdots \\
&\quad \sigma_k = \mathcal{T}(\Gamma + \{x_i:\tau_i\}, e_k, \tau_k, \sigma_{k-1}) \\
\mathcal{T}(\Gamma, \texttt{case } e \texttt{ of } \{c_i\, \overline{x}_i \to e_i\}_{i=1}^k, \tau, \sigma) &= \sigma_k \\
&\quad \text{where} \\
&\quad \Gamma(c_i) = \overline{\rho}_i \to T \quad \forall i = 1..k \\
&\quad \sigma_0 = \mathcal{T}(\Gamma, e, T, \sigma) \\
&\quad \sigma_1 = \mathcal{T}(\Gamma + \overline{x}_1 : \overline{\rho}_1, e_1, \tau, \sigma_0) \\
&\quad \vdots \qquad\quad \vdots \\
&\quad \sigma_k = \mathcal{T}(\Gamma + \overline{x}_k : \overline{\rho}_k, e_k, \tau, \sigma_{k-1})
\end{aligned}
$$

Figure 3: Type inference algorithm for the simply typed language

yields a principal typing $\tilde{\sigma}$ with respect to σ for (Γ, e, τ). Otherwise it fails. □

PROOF Similar to the soundness and completeness of $\mathcal{M}$ proved in [LY98]. ■

For our type inference algorithm we assume the existence of a unification algorithm. A substitution σ is a *unifier* for an equation $\tau_1 \doteq \tau_2$ iff $\tau_1\sigma = \tau_2\sigma$ Moreover, $\tilde{\sigma}$ is a *most general unifier* for $\tau_1 \doteq \tau_2$ iff it is a unifier for $\tau_1 \doteq \tau_2$ and for every unifier σ' for $\tau_1 \doteq \tau_2$ we have $\sigma \preceq \sigma'$.

We require $\hat{\mathcal{U}}(\tau_1 \doteq \tau_2)$ to return a most general unifier $\tilde{\sigma}$ for $\tau_1 \doteq \tau_2$, if a unifier exists, and fail otherwise. For convenience we define $\mathcal{U}(\tau_1 \doteq \tau_2, \sigma) := \sigma\hat{\mathcal{U}}(\tau_1\sigma \doteq \tau_2\sigma)$. Hence $\mathcal{U}(\tau_1 \doteq \tau_2, \sigma)$ returns a most general unifier $\tilde{\sigma}$ for $\tau_1\sigma \doteq \tau_2\sigma$ with $\sigma \preceq \tilde{\sigma}$, if a unifier exists, and fails otherwise.

We choose to formulate $\mathcal{T}$ with the additional substitution, because it makes the presentation of the algorithm more readable and it already indicates an efficient implementation method.

3.2 Abstraction of List Constructors

We present our list constructor abstraction algorithm at the hand of an example. The following well-typed expression produces a list.

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

Our algorithm replaces every list constructor `(:)`, respectively `[]`, by a different variable c_i, respectively n_i. To use the existing type annotations as far as possible, we just replace every type `[Int]` in the expression by a new type inference variable. Furthermore, we add $c_i : \tau \to \delta_i \to \delta_i$, respectively $n_i : \delta_i$, to the type environment, where δ_i is a new type inference variable for every variable c_i, respectively n_i. These type schemes assure that the inferred types will be appropriate for `(:)` or the consumer argument $e_{(:)}$, respectively `[]` or $e_{[]}$. As remaining arguments we give $\mathcal{T}$ a type inference variable γ and the identity substitution. For our example, input and output of $\mathcal{T}$ look as follows

```
T({g : Int→Int, 1 : Int, 2 : Int,
   (:) : Int→[Int]→[Int], [] : [Int],
   c1 : Int→ δ1 → δ1, c2 : Int→ δ2 → δ2,
   c3 : Int→ δ3 → δ3, n1 : δ4, n2 : δ5}
 , let map : (Int→Int)→ δ6 → δ7
           = λf:Int→Int. λxs:δ8.
               case xs of {
                 []   → n1,
                 y:ys → c1 y (map f ys)}
   in map g (c2 1 (c3 2 n2))
 ,γ, id )
=  [γ/δ1][[Int]/δ2][[Int],δ3][γ/δ4][[Int]/δ5]
   [[Int]/δ6][γ/δ7][[Int]/δ8]
```

The inferred principal typing is easier to read when we apply the substitution to the input:

```
{g : Int→Int, 1 : Int, 2 : Int,
 (:) : Int→[Int]→[Int], [] : [Int],
 c1 : Int→ γ → γ, c2 : Int→[Int]→[Int],
 c3 : Int→[Int]→[Int], n1 : γ, n2 : [Int]}
⊢ let map : (Int→Int)→[Int]→ γ
          = λf:Int→Int. λxs:[Int].
```

$$
\begin{array}{rcll}
\text{Type constructors} & C & := & \texttt{[]} \mid \texttt{Int} \mid \ldots \\
\text{Type variables} & \alpha, \beta & & \\
\text{Type inference variables} & \gamma, \delta, \delta_1, \delta_2, \ldots & & \\
\text{Types} & \tau, \rho & ::= & C\,\overline{\tau} \mid \alpha \mid \beta \mid \tau \to \rho \mid \forall\alpha.\tau \\
\text{Terms} & e & ::= & \ldots \mid \lambda\alpha.e \mid e\,\tau
\end{array}
$$

Figure 4: Additional terms and types of the second-order typed language

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

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

Figure 5: New type rules for the second-order typed language

```
                case xs of {
                 []   → n1,
                 y:ys → c1 y (map f ys)}
  in map g (c2 1 (c3 2 n2)) : γ
```

The type environment tells us that $\texttt{c}_1$ and $\texttt{n}_1$ construct the result of the expression whereas $\texttt{c}_2$, $\texttt{c}_3$, and $\texttt{n}_2$ have to construct lists that are internal to the expression.

Because `build` has a polymorphic type, we cannot express the producer in `build` form in this section. Instead we directly consider the final fusion step. For a consumer `foldr` $e_{(:)}$ $e_{[]}$ the variables $\texttt{c}_1$ and $\texttt{n}_1$ are replaced by $e_{(:)}$ and $e_{[]}$. The other variables are turned back into list constructors. For example, fusion of our producer with the consumer `foldr (+) 0` gives:

```
{g : Int→Int, 1 : Int, 2 : Int,
 (:) : Int→[Int]→[Int], [] : [Int],
 (+) : Int→Int→Int, 0 : Int}
⊢ let map : (Int→Int)→[Int]→Int
          = λf:Int→Int. λxs:[Int].
              case xs of {
               []   → 0,
               y:ys → y + (map f ys)}
  in map g [1,2] : Int
```

In general, the result of type inference may still contain other type inference variables than γ. Subexpressions of type inference variable type are not relevant for the result and hence the type inference variables can safely be replaced by `[Int]`. If type inference fails, no fusion is possible.

As an optimisation to make $\mathcal{T}$ fail early if the desired polymorphic type cannot be inferred, we let the unification algorithm treat γ like a constant, not like a type inference variable that can be replaced.

3.3 Inlining of External Definitions

The previous example is rather artificial, because usually the definition of `map` will not be part of the producer. Deforestation requires inlining. The example also demonstrates that it is important not just to inline the right hand side of the definition but the whole recursive definition. Experiences with the implementation of other deforestation algorithms [Mar95, NP98] have shown that leaving inlining to a separate transformation pass gives unsatisfactory results. We can in fact use the type environment of a principal typing of the producer to determine the variables whose definitions need to be inlined.

Determining which definitions can be inlined without duplicating computations is a separate art not discussed here (see for example [San95, MOTW95, WP99]). At least it is always safe to inline functions defined by a λ-abstraction.

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

As example we consider the producer `map g [1,2]`. After replacement of the list constructors and types `[Int]` the type inference algorithm gives us the typing

```
{map : (Int→Int)→ δ → γ,
 g : Int→Int, 1 : Int, 2 : Int,
 n1 : δ, c1 : Int→ δ → δ, c2 : Int→ δ → δ}
⊢ map g (c1 1 (c2 2 n1)) : γ
```

The special type inference variable γ in the type of `map` signifies that the definition of `map` needs to be inlined. We do not need to repeat the whole process for an extended producer. Instead we continue processing the right hand side of map with the type substitution we already have. We inline definitions and process them until γ appears in the type of no further variable of the type environment, except those of the c_i and n_i of course. Subsequently, all processed (potentially mutually recursive) definitions are put into a single let binding. Furthermore, the bound variables need to be renamed to preserve our invariant that every binding construct binds a different variable. For our example the output coincides modulo variable names with the output from Section 3.2. The actual fusion is performed as described there. In contrived cases our algorithm may inline all list functions. Although we do not expect such a code explosion to occur in practise, the algorithm will already abandon list constructor abstraction when the inlined code exceeds a predefined size.

$$
\begin{aligned}
\mathcal{T}(\Gamma, \texttt{case}\ e\ \texttt{of}\ \{c_i\,\overline{x}_i \to e_i\}_{i=1}^{k}, \tau, \sigma) &= \sigma_k \\
&\quad \text{where} \\
&\quad \Gamma(c_i) = \forall\overline{\alpha}.\overline{\rho}_i \to C\,\overline{\alpha} \quad \forall i = 1..k \\
&\quad \overline{\delta} \text{ new type inference variables} \\
&\quad \sigma_0 = \mathcal{T}(\Gamma, e, C\,\overline{\delta}, \sigma) \\
&\quad \sigma_1 = \mathcal{T}(\Gamma + \overline{x}_1 : \overline{\rho}_1[\overline{\delta}/\overline{\alpha}], e_1, \tau, \sigma_1) \\
&\quad \vdots \qquad\qquad \vdots \\
&\quad \sigma_k = \mathcal{T}(\Gamma + \overline{x}_k : \overline{\rho}_k[\overline{\delta}/\overline{\alpha}], e_k, \tau, \sigma_{k-1}) \\
\mathcal{T}(\Gamma, \lambda\alpha.e, \tau, \sigma) &= \mathcal{T}(\Gamma, e, \delta, \mathcal{U}(\tau \doteq \forall\alpha.\delta, \sigma)) \\
&\quad \text{where} \\
&\quad \delta \text{ a new type inference variable} \\
\mathcal{T}(\Gamma, e\,\rho, \tau, \sigma) &= \sigma_0\hat{\mathcal{U}}(\tau\sigma_0 \doteq \tau_0[\rho\sigma_0/\alpha]) \\
&\quad \text{where} \\
&\quad \delta \text{ a new type inference variable} \\
&\quad \sigma_0 = \mathcal{T}(\Gamma, e, \delta, \sigma) \\
&\quad \forall\alpha.\tau_0 = \delta\sigma_0 \text{ for some } \alpha \text{ and } \tau_0
\end{aligned}
$$

Figure 6: Modification of the type inference algorithm for the second-order typed language

4 Build Derivation in the Polymorphic Language

We extend the terms of our simply typed language by type abstraction and type application and the types by polymorphic algebraic data types and universally quantified types. The additional syntax is given in Figure 4 and the new type rules in Figure 5. We distinguish between type variables which may be bound in types and terms, and type inference variables which only occur freely. Because of the polymorphic algebraic data types with polymorphic constructors we need a more general type rule for `case`. Note that we write `[`α`]` instead of `[]` α for the polymorphic list type and that $\Gamma($`(:)`$) = \forall\alpha.\alpha \to$`[`$\alpha$`]`$\to$`[`$\alpha$`]` and $\Gamma($`[]`$) = \forall\alpha.$`[`α`]`. The functions `build` and `foldr` are defined as follows

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

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) }
```

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

`foldr` $\tau_1\ \tau_2\ e_{(:)}\ e_{[]}$ `(build` $\tau_1\ g$`)` $\quad\rightsquigarrow\quad g\ \tau_2\ e_{(:)}\ e_{[]}$

As for the simply typed language we demand that every variable binding construct binds a different variable and that bound variables are disjunct from free variables, both for term variables and for type variables. However, for better readability we are more liberal with variable names in our examples.

4.1 Type Inference

The extension of $\mathcal{T}$ to the second-order typed language is given in Figure 6. It is of central importance that we require $\mathcal{U}$ to treat type variables like constants, only type inference variables may be replaced. The only place where we replace a type variable is in the typing of $e\,\rho$, but this substitution is not part of the result substitution of $\mathcal{T}$. All σ, $\tilde{\sigma}$, σ_0, etc. denote substitutions that only replace type inference variables.

Note that in general $\mathcal{T}$ is neither sound nor complete. In the case of $\lambda\alpha.e$, $\mathcal{T}$ performs no test to assure that a bound type variable does not leave its scope. In the case of $e\,\rho$ we expect $\mathcal{T}(\Gamma, e, \delta, \sigma)$ to return a universally quantified type. Hence $\mathcal{T}(\{x : \delta\}, x\,\rho, \texttt{Int}, \mathrm{id})$ fails in finding a typing.

Our algorithm $\mathcal{T}$ relies on the fact that the original producer is typable and that not only all places of type abstraction and type application are known, but that we only replace list types by type inference variables. Hence a principal typing exists and only list types need to be substituted for the type inference variables in the type environment and the expression. Therefore the following limited completeness of $\mathcal{T}$ suffices for our fusion method.

Theorem 4.1 *Let Γ be an environment, e an expression and τ and ρ types. If there exists a substitution σ' with $\delta\sigma' = [\rho]$ for all $\delta \in \mathrm{freeTyVar}(\Gamma\sigma, e\sigma)$ such that σ' is a typing for $(\Gamma\sigma, e\sigma, \tau\sigma)$, then $\mathcal{T}(\Gamma, e, \tau, \sigma)$ yields a principal typing $\tilde{\sigma}$ with respect to σ for (Γ, e, τ).* □

Proof The theorem is an instance of Theorem A.1, which is given and proved in Appendix A. ■

The extended algorithm $\mathcal{T}$ can be used for abstracting list constructors in second-order typed programs just as described in Section 3.2 for simply typed programs. The only difference is that we do not replace `(:)`, `[]` and `[Int]`. Instead, if `[`τ`]` is the type of the intermediate list producer, we replace every `(:)` τ, respectively `[]` τ, by a new c_i, respectively n_i, and replace every type `[`τ`]` by a new type inference variable.

4.2 Instantiation of Polymorphism

There is however still a problem left. Functions that manipulate polymorphic lists, such as for example `map` : $\forall\alpha.\forall\beta.(\alpha \to \beta)\to$`[`$\alpha$`]`$\to$`[`$\beta$`]`, prevent list constructor abstraction, because the type annotations contain polymorphic list types like `[`β`]` instead of `[`τ`]`.

Hence we replace all occurrences of a type application $f\ \overline{\rho}$, where f is an inlineable variable with $f : \forall\overline{\alpha}.\tau = \lambda\overline{\alpha}.e$ and τ contains a polymorphic list type `[`α`]` with $\alpha \in \overline{\alpha}$, by a new inlineable variable $f_{\overline{\rho}}$ that is defined through $f_{\overline{\rho}} : \tau[\overline{\rho}/\overline{\alpha}] = e[\overline{\rho}/\overline{\alpha}]$. This type instantiation is performed before the list types `[`τ`]` are replaced by type inference variables. To avoid code explosion we inline at most one such instantiated definition for every polymorphic variable f.

As example consider the following list producer:

```
{fst : ∀α.∀β.(α,β) → α, zs : [(Bool, Char)],
 unzip : ∀α.∀β.[(α,β)] → ([α],[β])}
⊢ fst [Bool] [Char] (unzip Bool Char zs) : [Bool]
```

Our instantiation of polymorphism replaces all occurrences of `unzip Bool Char` by a monomorphic `unzip'`:

```
{fst : ∀α.∀β.(α,β) → α, zs : [(Bool, Char)],
 unzip' : [(Bool,Char)]→([Bool],[Char])}
⊢ fst [Bool] [Char] (unzip' zs) : [Bool]
```

Then we replace all occurrences of `[Bool]` by new type inference variables and replace all (here not existing) list constructors. The input and the result of type inference look as follows.

```
T({fst : ∀α.∀β.(α,β) → α, zs : [(Bool, Char)],
   unzip' : [(Bool,Char)]→(δ1,[Char])}
  , fst δ2 [Char] (unzip' zs) , γ, id)
= [γ/δ1][γ/δ2]
```

The γ in the inferred type of `unzip'` indicates that its definition must be inlined. The original right hand side of the definition of `unzip` is:

```
{unzip : ∀α.∀β.[(α,β)] → ([α],[β]),
 foldr : ∀α.∀β.(α → β → β) → β → [α] → β,
 (:) : ∀α.α → [α] → [α], [] : ∀α.[α],
 (,) : ∀α.∀β.α → β →(α,β)}
⊢ λα. λβ. foldr (α,β) ([α],[β])
          (λy:(α,β). λu:([α],[β]). case y of {
           (v,w) → case u of {
            (vs,ws) → (,) [α] [β] ((:) α v vs)
                          ((:) β w ws)}})
          ((,) [α] [β] ([] α) ([] β))
: ∀α.∀β.[(α,β)] → ([α],[β])
```

For processing `unzip'` the input and the result of the type inference algorithm is as follows. Note that we start with the type substitution that we obtained from previous type inference.

```
T({unzip' : [(Bool,Char)]→(δ1,[Char]),
   foldr : ∀α.∀β.(α → β → β) → β → [α] → β,
   (:) : ∀α.α → [α] → [α], [] : ∀α.[α],
   (,) : ∀α.∀β.α → β →(α,β),
   c1 : Bool→ δ3 → δ3, n1 : δ4}
  , foldr (Bool,Char) (δ5,[Char])
    (λy:(Bool,Char). λu:(δ6,[Char]). case y of {
     (v,w) → case u of {
      (vs,ws) → (,) δ7 [Char] (c1 v vs)
                 ((:) Char w ws)}})
    ((,) δ8 [Char] n1 ([] Char))
  , [(Bool,Char)]→(δ1,[Char]), [γ/δ1][γ/δ2])
= [γ/δ1][γ/δ2][γ/δ3][γ/δ4][γ/δ5][γ/δ6][γ/δ7][γ/δ8]
```

There is no γ in the type of any further variable of the type environment. Hence list constructor abstraction has terminated successfully.

We can now express the producer in `build` form as follows:

```
{build : ∀α.(∀β.(α → β → β) → β → β) → [α],
 fst : ∀α.∀β.(α,β) → α, zs : [(Bool, Char)],
 foldr : ∀α.∀β.(α → β → β) → β → [α] → β,
 (:) : ∀α.α → [α] → [α], [] : ∀α.[α],
 (,) : ∀α.∀β.α → β →(α,β)}
⊢ build Bool (λβ. λc:Bool→ β → β. λn:β.
   let unzip' : [(Bool,Char)]→(β,[Char])
              = foldr (Bool,Char) (β,[Char])
                 (λy:(Bool,Char). λu:(β,[Char]).
                  case y of {(v,w) →
                   case u of {(vs,ws) →
                    (,) β [Char] (c v vs)
                     ((:) Char w ws)}})
                 ((,) β [Char] n ([] Char))
   in fst β [Char] (unzip' zs) )
: [Bool]
```

Note that the producer `fst [Bool] [Char] (unzip Bool Char zs)` cannot be fused by standard short cut deforestation, because neither `fst` nor `unzip` can individually be expressed in `build` form. Note also that the definition of `fst` is not needed for deforestation and hence is not inlined.

In practise we finally have to rename all inlined variables and their types to preserve the invariant that every binding construct binds a different variable. In general we also may have instantiated polymorphic variables that we did not inline later. These instantiations have to be undone.

The instantiation method may seem restrictive, but note that the translation of a Hindley-Milner typed program into our second-order typed language yields a program where only let-bound expressions are type-abstracted and polymorphic variables occur only in type applications. Programs in the intermediate language Core of the Glasgow Haskell compiler are nearly completely in this form, because the Haskell type systems is based on the Hindley-Milner type system.

5 Deforestation of a Program

To deforest a program we traverse it twice.

During the first traversal we search for potentially fusible expressions `foldr` $\tau_1\ \tau_2\ e_{(:)}\ e_{[]}\ e$. Furthermore, we collect all definitions from `let` bindings that may be inlined. Note that we only need to collect the definitions of variables that return lists, that is, have a list type after the last function arrow $\to$ in their type.

Suppose we find the following potentially fusible expression:

```
...
 foldr Bool Bool (&&) True
  (fst [Bool] [Char] (unzip Bool Char zs))
...
```

For every such expression we try to convert its producer into `build` form.

```
...
 foldr Bool Bool (&&) True
  (build Bool (λβ.λc:Bool→ β → β.λn:β. ...))
...
```

After this `build` derivation pass we fuse the `foldr/build` pairs in the second, actual deforestation pass.

```
...
 (λβ.λc:Bool→ β → β.λn:β. ...) Bool (&&) True
...
```

Finally, a few β-reductions put the arguments $e_{(:)}$ and $e_{[]}$ of the consumer in the former places of the list constructors.

```
...
 let unzip' : [(Bool,Char)]→(Bool,[Char])
           = foldr (Bool,Char) (Bool,[Char])
              (λy:(Bool,Char).λys:(Bool,[Char]).
               case y of {(v,w) →
                case (unzip' ys) of {(vs,ws) →
                 (,) Bool [Char] (v && vs)
                  ((:) Char w ws)}})
              ((,) Bool [Char] True ([] Char))
 in fst Bool [Char] (unzip' zs)
...
```

The idea immediately suggests itself that we could directly replace the right list constructors by $e_{(:)}$ and $e_{[]}$ during the `build` derivation pass, thereby avoiding completely the explicit construction of the `build` form. Indeed we did so in Section 3. However, we want to stress that finding the right list constructors in a list producer and using this information are two separate issues. Keeping them separate increases the potential application area of our `build` derivation method. We can, for example, derive `build` wrappers of list producing functions for a wrapper/worker scheme as described in [LS95, Gil96]. Furthermore, the `foldr/build` rule and various simple transformations like β-reduction are already implemented in the Glasgow Haskell compiler.

Note however, that we only derive a `build` form when it is needed for deforestation. Thus we avoid the potential inefficiencies introduced by superfluous `build` forms which were observed in [Gil96].

Besides removing the costs of an intermediate data structure, deforestation also brings together subexpressions of producer and consumer which previously were separated by the intermediate data structure. Thus new opportunities for optimising transformations arise, even for deforestation itself. Consider for example the expression

```
map sum (inits [1..])
```

The function `inits` returns the list of all prefixes of its argument, that is, `[[],[1],[1,2],...]`. A short cut deforestation pass fuses `inits` with `[1..]` and `map sum` with the result of the former fusion. Afterwards `sum` is positioned next to the production of the inner lists `[],[1],[1,2],...` and can be fused with it in a second deforestation pass.

6 Extensions

6.1 Fusion with Partial Producers

The list append function `(++)` poses a problem for the `foldr/build` rule. The expression `(++)` τ `xs ys` does not produce the whole resulting list itself, because only `xs` is copied but not `ys`. Modifying the definition of `(++)` to copy `ys` as well, as done in [GLP93, LS95, TM95], runs contrary to our aim of eliminating data structures. Fortunately, Gill [Gil96] discovered a generalisation of `build`

$$\begin{aligned}\texttt{augment} &: \forall\alpha.(\forall\beta.(\alpha\to\beta\to\beta)\to\beta\to\beta)\to[\alpha]\to[\alpha]\\ &= \lambda\alpha.\ \lambda g{:}\forall\beta.(\alpha\to\beta\to\beta)\to\beta\to\beta).\ \lambda xs{:}[\alpha].\\ &\quad\ g\ [\alpha]\ ((:)\ \alpha)\ xs\end{aligned}$$

and the `foldr/build` rule

$$\begin{aligned}&\texttt{foldr}\ \tau_1\ \tau_2\ e_{(:)}\ e_{[]}\ (\texttt{augment}\ \tau_1\ g\ xs)\\ \rightsquigarrow\ &g\ \tau_2\ e_{(:)}\ (\texttt{foldr}\ \tau_1\ \tau_2\ e_{(:)}\ e_{[]}\ xs)\end{aligned}$$

The `foldr/augment` rule does not eliminate the whole intermediate list, but at least the part produced by the partial producer.

Let us come back to our example `(++)` τ `xs ys`. Type inference indicates that the definition of $(\texttt{++})_\tau$ needs to be inlined. After type inference of the definition, the type environment contains `ys`: γ but `ys` may not be inlineable, because it is shared or λ-bound. Then this non-inlineable variable becomes the list argument of `augment`. So the `augment` form of our example is:

```
augment τ ( λβ. λc:τ → β → β. λn:β.
 let (++)_τ : [τ]→ β → β
           = λvs:[τ]. λws:β. case vs of {
              []   → ws,
              z:zs → c z ((++)_τ zs ws) }
 in (++)_τ xs n ) ys
```

The `foldr/build` rule is a special instance of the `foldr/augment` rule, we only have to replace `xs` by `[]` and β-reduce to obtain the former from the latter. So we do not need `build` at all. We prefer to use `build` in this paper, because it is more intuitive and more wildly known.

6.2 Other Algebraic Data Types than Lists

Our type-based method for transforming a producer into `build` form is not specific to lists but can be used for most other algebraic data types. Consider for example the type of arbitrarily branching trees:

```
data RoseTree a = Node a [RoseTree a]
```

The `build` function of the type is determined by the the data constructor(s) and its type(s):

$$\texttt{Node} : \forall\alpha.\alpha\to[\texttt{RoseTree}\ \alpha]\to(\texttt{RoseTree}\ \alpha)$$

$$\begin{aligned}\texttt{buildRT} &: \forall\alpha.(\forall\beta.(\alpha\to[\beta]\to\beta)\to\beta)\to(\texttt{RoseTree}\,\alpha)\\ &= \lambda\alpha.\ \lambda g{:}\forall\beta.(\alpha\to[\beta]\to\beta)\to\beta.\\ &\quad\ g\ (\texttt{RoseTree}\ \alpha)\ (\texttt{Node}\ \alpha)\end{aligned}$$

Note that $\forall\beta.(\alpha\to[\beta]\to\beta)\to\beta$ is the canonical encoding of the data type `RoseTree` α in the second-order λ-calculus and `buildRT` α is the isomorphism from the former type to the latter, just as $\forall\beta.(\alpha\to\beta\to\beta)\to\beta\to\beta$ is the canonical encoding of the data type $[\alpha]$ and `build` α is the isomorphism from the former type to the latter (cf. [Pit98]).

Furthermore, an algebraic data type comes with a catamorphism which consumes its elements in a regular way (see e.g. [MH95] for how a catamorphism is defined in general):

```
foldRT : ∀α∀β.(α → [β] → β) → (RoseTree α) → β
       = λα. λβ. λn:α → [β] → β. λt:RoseTree α.
         case t of { Node x ts →
          n x (map (RoseTree α) β
                (foldRT α β n) ts) }
```

The `foldRT/buildRT` rule is an instance of the parametricity theorem of the type of `buildRT`:

$$\texttt{foldRT}\ \tau_1\ \tau_2\ e_{\texttt{Node}}\ (\texttt{buildRT}\ \tau_1\ g) \quad \rightsquigarrow \quad g\ \tau_2\ e_{\texttt{Node}}$$

Obviously, we can derive a `buildRT` form from an expression of type `RoseTree` τ just as easily as a `build` form from a list-typed expression. For deforestation we additionally need a consumer to be defined in terms of the catamorphism of the data type. The catamorphism could be defined and used explicitly by the user. The compiler would learn the name of the catamorphism through a directive. Alternatively, a consumer could automatically be transformed into catamorphism form by another algorithm like warm fusion.

We can even handle mutually recursive data types. The catamorphisms of a set of mutually recursive data types are mutually recursive. Because the last argument of every `build` is polymorphic in as many type variables as there are types in the set, we need several special type inference variables γ_i for `build` derivation.

However, all the above is limited to regular, covariant algebraic data types. An algebraic data type is called regular, if all recursive calls in the body are of the form of the head of the definition. A counter-example is

```
data Twist a b = Nil | Cons a (Twist b a)
```

A data type is called contravariant, if some recursive call appears in a contravariant position of the body, that is, more or less as the first argument of the function type constructor:

```
data Infinite a = I (Infinite a -> a)
```

Catamorphisms for contravariant and mixedvariant types are more complex than for covariant ones and for non-regular types a general definition is unknown [MH95]. Furthermore, the validity of parametricity theorems has only been proved for regular, covariant algebraic data types [Pit98]. Nonetheless these limitations are mild, because nearly all types that appear in practise are regular and covariant.

7 Related Work

Wadler [Wad90] coined the term deforestation. His method is based on fold-unfold transformations; see [San96] for the first proof of correctness and a recent presentation of this approach.

On the other hand, there are methods like short cut deforestation that abandon the treatment of generally recursive definitions in favour of some "regular" form of recursion. These methods use fusion laws based on parametricity theorems. Authors take different points of view on how the "regular" forms of recursion are obtained, if they need to be provided by the programmer or are automatically inferred from arbitrary recursive programs by another transformation.

[MFP91] gives an overview over several such fusion laws. Hylomorphisms [TM95] enable the description of "regular" producers besides "regular" consumers in a single form. [HIT96] presents an algorithm for transforming generally recursive definitions into hylomorphisms. The algorithm can transform most definitions of real world programs [Tüf98]. [Feg96] describes an even more ambitious method of deriving a recursion skeleton from a definition and using its parametricity theorem for fusion. The sketched algorithms are still far from an implementation.

Generally, implementations of deforestation have problems with controlling inlining to avoid code explosion [Mar95, NP98]. Unlike the `foldr/build` rule, most fusion laws that are based on "regular" producers or consumers still require some transformation of arbitrary expressions. All methods for automatically transforming programs into some "regular" form are purely syntax-directed and raise the question of how generally they are applicable. Finally, the question of how such a transformation changes the efficiency of a program is usually not answered.

8 Conclusions

In this paper we presented a type-inference-based method for adapting producers of intermediate data structures for short cut deforestation. We showed how the problem of abstracting the data constructors of an arbitrary producer can be reduced to a partial type inference problem. We presented a linear-time algorithm that solves this type inference problem. Together with inlining and polymorphic type instantiation it transforms a producer into `build` form which is suitable for fusion with a `foldr` consumer.

Our method is efficient and easier to implement than the warm fusion method presented in [LS95]. We also believe it to be able to transform more producers into the desired form. This claim is difficult to prove; a complete definition of the rewriting performed by warm fusion would be required. However, the example in Section 4.2 demonstrates that the warm fusion technique of transforming list-manipulating functions into `build` form on a per function basis is insufficient. The functions which comprise the producer, `fst` and `unzip`, cannot be expressed in `build` form.

Furthermore, our type based method clearly indicates which definitions need to be inlined, a control that has been much searched for [Mar95, NP98]. An optimising compiler of a high-level programming language has to undo the abstractions of a program. Hence it transforms a modular, concise program into efficient, longer code. Nonetheless it has to avoid unnecessary inlining as well.

Because our method basically just replaces list constructors in the producer and performs no complex transformation, it is also *transparent* [Mar95], that is, the programmer can easily determine from the source program where deforestation will be applicable. Furthermore, the effect of our transformation on efficiency is more calculable. We also showed that our short cut deforestation method is not specific to lists but can be used for most algebraic data types. We only require some additional mechanism that ensures that the consumer of a data structure is defined in terms of the respective catamorphism.

With our algorithm we adapted by hand a large number of list producers for fusion. We are currently working on an implementation of our short cut deforestation algorithm for the Glasgow Haskell compiler. This compiler was designed for being easily extendible by further compiler optimisations [Chi97].

Finally, we believe that the idea of using type inference

algorithms could also be fruitful for other transformations based on parametricity theorems.

Acknowledgements

I thank Frank Huch and Simon Peyton Jones for fruitful discussions and many suggestions for improving this paper.

References

[Chi97] Olaf Chitil. Adding an optimisation pass to the Glasgow Haskell compiler. Available from `http://www-i2.informatik.rwth-aachen.de/~chitil`, November 1997.

[Feg96] Leonidas Fegaras. Fusion for free! Technical Report CSE-96-001, Oregon Graduate Institute of Science and Technology, January 8, 1996.

[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, Copenhagen, Denmark, June 9–11, 1993. ACM Press.

[HIT96] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings 1st ACM SIGPLAN Intl. Conf. on Functional Programming, ICFP'96, Philadelphia, PA, USA, 24–26 May 1996*, pages 73–82, New York, 1996. ACM Press.

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

[LS95] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 314–323. ACM Press, New York, 1995.

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

[Mar95] Simon Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Glasgow University, 1995.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Verlag, June 1991.

[MH95] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 324–333, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[MOTW95] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need, and the linear lambda calculus. In *11'th International Conference on the Mathematical Foundations of Programming Semantics*, New Orleans, Lousiana, March–April 1995.

[NP98] Lázló Németh and Simon Peyton Jones. A design for warm fusion. In *Proceedings of the 10th International Workshop on Implementation of Functional Languages*, pages 381–393, 1998.

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

[Pit98] A. M. Pitts. Parametric polymorphism and operational equivalence. Technical Report 453, Cambridge University Computer Laboratory, 1998. A preliminary version appeared in *Proceedings, Second Workshop on Higher Order Operational Techniques in Semantics (HOOTS II), Stanford CA, December 1997*, Electronic Notes in Theoretical Computer Science 10, 1998.

[San95] André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1995.

[San96] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1–2):193–233, October 1996.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 306–313, La Jolla, California, June 25–28, 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[Tüf98] Christian Tüffers. Erweiterung des Glasgow-Haskell-Compilers um die Erkennung von Hylomorphismen. Master's thesis, RWTH Aachen, 1998.

[Wad89] Philip Wadler. Theorems for free. In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[Wad90] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990.

[Wel94] J. B. Wells. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 176–185, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[WP99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Conference Record of POPL '99: The 26nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

A Proof of Theorem 4.1

The theorem we prove here is slightly stronger than Theorem 4.1. This is because the property of principality of a typing $\tilde{\sigma}$, that is, that for any typing σ' for (Γ, e, τ) there exists $\hat{\sigma}$ such that $\Gamma\sigma' = \Gamma\tilde{\sigma}\hat{\sigma}$, $e\sigma' = e\tilde{\sigma}\hat{\sigma}$ and $\tau\sigma' = \tau\tilde{\sigma}\hat{\sigma}$, is too weak for an induction. On the other hand, it is not true for the result $\tilde{\sigma}$ of $\mathcal{T}$, that there exists $\hat{\sigma}$ such that $\sigma' = \tilde{\sigma}\hat{\sigma}$, because the call $\mathcal{T}(\Gamma, e, \tau, \sigma)$ generally creates new variables which σ' may substitute differently from $\tilde{\sigma}$. However, we can prove that $\delta\sigma' = \delta\sigma\hat{\sigma}$ for all type inference variables δ except the newly created ones.

Hence we define the following: Let N be a set of type inference variables. We write $\sigma =_{|N} \sigma'$ iff $\delta\sigma = \delta\sigma'$ for all type inference variables $\delta \notin N$. We write $\sigma \preceq_N \sigma'$ iff there exists $\hat{\sigma}$ with $\sigma' =_{|N} \sigma\hat{\sigma}$.

A *non-$\forall$-type* is a type that does not have a $\forall$ at the outermost level. We write $\sigma \preceq_N^M \sigma'$ iff there exists $\hat{\sigma}$ with $\sigma' =_{|N} \sigma\hat{\sigma}$ and for all $\delta \in \text{freeTyVar}(M\sigma)$ the type $\delta\hat{\sigma}$ is a non-$\forall$-type. The idea is that for extending σ to obtain σ' the variables in $M\sigma$ need only be replaced by non-$\forall$-types.

Remember that $\mathcal{U}$ treats type variables like constants, only type inference variables may be replaced. Hence σ, $\tilde{\sigma}$, σ_0, etc. denote substitutions that only replace type inference variables.

Theorem A.1 (Soundness and Completeness of $\mathcal{T}$)
Let σ' be typing for (Γ, e, τ) with $\sigma \preceq_N^{(\Gamma,e)} \sigma'$ for some variables N not appearing in Γ, e or τ. Then $\mathcal{T}(\Gamma, e, \tau, \sigma)$ creates new variables $\tilde{N}$ and returns $\tilde{\sigma}$ with $\sigma \preceq \tilde{\sigma}$, $\tilde{\sigma} \preceq_{N\cup\tilde{N}}^{(\Gamma,e,\tau)} \sigma'$ and $\Gamma\tilde{\sigma} \vdash e\tilde{\sigma} : \tau\tilde{\sigma}$. □

PROOF Union of Lemma A.6 and Lemma A.7. ■

The following lemmas are used in the proof of the main lemma.

Lemma A.2 (Substitution in a typing)
If $\Gamma \vdash e : \tau$ then $\Gamma\sigma \vdash e\sigma : \tau\sigma$ for any substitution σ. □

PROOF By induction on the derivation of $\Gamma \vdash e : \tau$. ■

Lemma A.3 *Let τ_1, τ_2, α and σ such that $\alpha \notin \text{freeTyVar}(\tau_1\sigma)$. Then $\tau_1\sigma[\tau_2\sigma/\alpha] = (\tau_1[\tau_2/\alpha])\sigma$.* □

PROOF By induction on τ_1, using that σ replaces no type variable α. ■

Lemma A.4 *If $\sigma_1 \preceq_N^M \sigma_3$, $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \preceq_{N\cup N'} \sigma_3$, then $\sigma_2 \preceq_{N\cup N'}^M \sigma_3$.* □

PROOF Directly from the definition of $\preceq_N^M$. ■

Lemma A.5 (Properties of $\mathcal{U}$)

1. *If $\mathcal{U}(\tau_1 \doteq \tau_2, \sigma)$ returns $\tilde{\sigma}$, then $\sigma \preceq \tilde{\sigma}$.*
2. *If $\mathcal{U}(\tau_1 \doteq \tau_2, \sigma)$ returns $\tilde{\sigma}$, then $\tau_1\tilde{\sigma} = \tau_2\tilde{\sigma}$.*
3. *If $\tau_1\sigma' = \tau_2\sigma'$ and $\sigma \preceq_N^M \sigma'$ then $\mathcal{U}(\tau_1 \doteq \tau_2, \sigma)$ returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_N^M \sigma'$.* □

PROOF Follows from the prerequisite that $\hat{\mathcal{U}}(\tau_1 \doteq \tau_2)$ returns a most general unifier if a unifier exists. ■

Note that algorithm $\mathcal{T}$ is defined recursively on the structure of the term argument, but not the structure of embedded types. Because substitutions only replace type inference variables, they do not increase the size of the actual term structure. Hence we can prove properties of $\mathcal{T}$ by structural induction on the term argument, applying the induction hypothesis to the recursive calls of $\mathcal{T}$.

Lemma A.6 ($\mathcal{T}$ extends substitutions)
If $\mathcal{T}(\Gamma, e, \tau, \sigma)$ returns $\tilde{\sigma}$, then $\sigma \preceq \tilde{\sigma}$. □

PROOF By induction on e. ■

We do not separate the main proof into a proof of soundness and a proof of completeness, because the proof of soundness would repeat large parts of the proof of completeness, because in the case of $\forall\alpha.e$ soundness requires the existence of a typing. We could avoid this complication by adding a test to $\mathcal{T}$ which corresponds to the condition $\alpha \notin \text{freeTyVar}(\Gamma)$ in TYPE ABS. However, we want to prove that for our purpose such a test is not necessary.

Lemma A.7 *Let σ' be typing for (Γ, e, τ) with $\sigma \preceq_N^{(\Gamma,e)} \sigma'$ for some variables N not appearing in Γ, e or τ. Then $\mathcal{T}(\Gamma, e, \tau, \sigma)$ creates new variables $\tilde{N}$ and returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_{N\cup\tilde{N}}^{(\Gamma,e,\tau)} \sigma'$ and $\Gamma\tilde{\sigma} \vdash e\tilde{\sigma} : \tau\tilde{\sigma}$.* □

PROOF Structural induction on the term e.

Case x.

Because $x\sigma' = x$, the typing $\Gamma\sigma' \vdash x\sigma' : \tau\sigma'$ must be derived by VAR. Hence $\Gamma\sigma'(x) = \tau\sigma'$. Furthermore $\sigma \preceq_N^{(\Gamma,x)} \sigma'$. According to Lemma A.5 $\mathcal{U}(\Gamma(x) \doteq \tau, \sigma)$ returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_N^{(\Gamma,x)} \sigma'$ and $\Gamma(x)\tilde{\sigma} = \tau\tilde{\sigma}$. Hence $\tilde{\sigma} \preceq_N^{(\Gamma,x,\tau)} \sigma'$. Furthermore with rule VAR follows $\Gamma\tilde{\sigma} \vdash x\tilde{\sigma} : \tau\tilde{\sigma}$.

Case $\lambda(x : \tau_1).e$.

The typing $\Gamma\sigma' \vdash (\lambda(x : \tau_1).e)\sigma' : \tau\sigma'$ must be derived by TERM ABS. Hence exists $\hat{\tau}$ with $\tau\sigma' = \tau_1\sigma' \to \hat{\tau}$ and $\Gamma\sigma' + x : \tau_1\sigma' \vdash e\sigma' : \hat{\tau}$, that is, $\tau\sigma'' = (\tau_1 \to \delta)\sigma''$ and $(\Gamma + x : \tau_1)\sigma'' \vdash e\sigma'' : \delta\sigma''$ for a new variable δ and $\sigma'' := \sigma'[\hat{\tau}/\delta]$. From $\sigma \preceq_N^{(\Gamma,\lambda(x:\tau_1).e)} \sigma'$ follows $\sigma \preceq_N^{(\Gamma,\lambda(x:\tau_1).e)} \sigma''$. According to Lemma A.5 $\mathcal{U}(\tau_1 \to$

$\delta \doteq \tau, \sigma)$ returns σ_0 with $\sigma_0 \preceq_N^{(\Gamma, \lambda(x:\tau_1).e)} \sigma''$ and $(\tau_1 \to \delta)\sigma_0 = \tau\sigma_0$.

It follows that $\sigma_0 \preceq_N^{(\Gamma + x:\tau_1, e)} \sigma''$. According to the induction hypothesis $\mathcal{T}(\Gamma + x : \tau_1, e, \delta, \sigma_0)$ creates new variables N_1 and returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_{N \cup N_1}^{(\Gamma + x:\tau_1, e, \delta)} \sigma''$ and $(\Gamma + x : \tau_1)\tilde{\sigma} \vdash e\tilde{\sigma} : \delta\tilde{\sigma}$.

According to Lemma A.6 $\sigma_0 \preceq \tilde{\sigma}$. Hence $(\tau_1 \to \delta)\tilde{\sigma} = \tau\tilde{\sigma}$. It follows that $\tilde{\sigma} \preceq_{N \cup N_1}^{(\Gamma, \lambda(x:\tau_1).e, \tau)} \sigma''$. Hence $\mathcal{T}(\Gamma, \lambda(x : \tau_1).e, \tau, \sigma)$ creates new variables $\tilde{N} := N_1 \cup \{\delta\}$ and returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_{N \cup \tilde{N}}^{(\Gamma, \lambda(x:\tau_1).e, \tau)} \sigma'$. Furthermore we can apply TERM ABS to obtain $\Gamma\tilde{\sigma} \vdash (\lambda(x : \tau_1).e)\tilde{\sigma} : \tau\tilde{\sigma}$.

Case $e_1\, e_2$.

The typing $\Gamma\sigma' \vdash (e_1\, e_2)\sigma' : \tau\sigma'$ must be derived by rule TERM APP. Hence exists $\hat{\tau}_2$ with $\Gamma\sigma' \vdash e_1\sigma' : \hat{\tau}_2 \to \tau\sigma'$ and $\Gamma\sigma' \vdash e_2\sigma' : \hat{\tau}_2$, that is, $\Gamma\sigma'' \vdash e_1\sigma'' : (\delta \to \tau)\sigma''$ and $\Gamma\sigma'' \vdash e_2\sigma'' : \delta\sigma''$ for a new variable δ and $\sigma'' := \sigma'[\hat{\tau}_2/\delta]$. From $\sigma \preceq_N^{(\Gamma, e_1\, e_2)} \sigma'$ follows $\sigma \preceq_N^{(\Gamma, e_1)} \sigma''$. So according to the induction hypothesis $\mathcal{T}(\Gamma, e_1, \delta \to \tau, \sigma)$ creates new variables N_1 and returns σ_1 with $\sigma_1 \preceq_{N \cup N_1}^{(\Gamma, e_1, \delta \to \tau)} \sigma''$ and $\Gamma\sigma_0 \vdash e_1\sigma_0 : (\delta \to \tau)\sigma_0$.

According to Lemma A.6 $\sigma \preceq \sigma_1$. Hence Lemma A.4 gives $\sigma_1 \preceq_{N \cup N_1}^{(\Gamma, e_2)} \sigma''$. Again according to the induction hypothesis $\mathcal{T}(\Gamma, e_2, \delta, \sigma_1)$ creates new variables N_2 and returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_{N \cup N_1 \cup N_2}^{(\Gamma, e_2, \delta)} \sigma''$ and $\Gamma\tilde{\sigma} \vdash e_2\tilde{\sigma} : \delta\tilde{\sigma}$.

With Lemma A.4 follows that $\tilde{\sigma} \preceq_{N \cup N_1 \cup N_2}^{(\Gamma, e_1\, e_2, \tau)} \sigma''$. Hence $\mathcal{T}(\Gamma, e_1\, e_2, \tau, \sigma)$ creates new variables $\tilde{N} := N_1 \cup N_2 \cup \{\delta\}$ and returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_{N \cup \tilde{N}}^{(\Gamma, e_1\, e_2, \tau)} \sigma'$. Furthermore, the Substitution Lemma A.2 assures $\Gamma\tilde{\sigma} \vdash e_1\tilde{\sigma} : (\delta \to \tau)\tilde{\sigma}$. Therefore we can apply TERM APP to obtain $\Gamma\tilde{\sigma} \vdash (e_1\, e_2)\tilde{\sigma} : \tau\tilde{\sigma}$.

Case $\lambda\alpha.e$.

The typing $\Gamma\sigma' \vdash (\lambda\alpha.e)\sigma' : \tau\sigma'$ must be derived by rule TYPE ABS. Hence $\Gamma\sigma' \vdash e\sigma' : \hat{\tau}$ for $\hat{\tau}$ with $\tau\sigma' = \forall\alpha.\hat{\tau}$. So $\tau\sigma'' = (\forall\alpha.\delta)\sigma''$ for a new type variable δ and $\sigma'' := \sigma'[\hat{\tau}/\delta]$. From $\sigma \preceq_N^{(\Gamma, \lambda\alpha.e)} \sigma'$ follows $\sigma \preceq_N^{(\Gamma, e)} \sigma''$. So according to Lemma A.5 $\mathcal{U}(\tau \doteq \forall\alpha.\delta, \sigma)$ returns σ_0 with $\sigma_0 \preceq_N^{(\Gamma, e)} \sigma''$ and $\tau\sigma_0 = (\forall\alpha.\delta)\sigma_0$.

According to the induction hypothesis $\mathcal{T}(\Gamma, e, \delta, \sigma_0)$ creates new variables N_1 and returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_{N \cup N_1}^{(\Gamma, e, \delta)} \sigma''$ and $\Gamma\tilde{\sigma} \vdash e\tilde{\sigma} : \delta\tilde{\sigma}$.

According to Lemma A.6 $\sigma_0 \preceq \tilde{\sigma}$. Hence $\tau\tilde{\sigma} = (\forall\alpha.\delta)\tilde{\sigma}$ and it follows that $\tilde{\sigma} \preceq_{N \cup N_1}^{(\Gamma, \lambda\alpha.e, \tau)} \sigma''$. So $\mathcal{T}(\Gamma, \lambda\alpha.e, \tau, \sigma)$ creates new variables $\tilde{N} := N_1 \cup \{\delta\}$ and returns $\tilde{\sigma}$ with $\tilde{\sigma} \preceq_{N \cup \tilde{N}}^{(\Gamma, \lambda\alpha.e, \tau)} \sigma'$.

Because σ' is a typing for (Γ, e, τ) we know that $\alpha \notin \Gamma\sigma'$. Together with $\tilde{\sigma} \preceq_N \sigma'$ for some variables N not appearing in Γ, e or τ follows that $\alpha \notin \Gamma\tilde{\sigma}$. Therefore we can apply TYPE ABS to obtain $\Gamma\tilde{\sigma} \vdash (\lambda\alpha.e)\tilde{\sigma} : \tau\tilde{\sigma}$.

Case $e\, \rho$.

The typing $\Gamma\sigma' \vdash (e\, \rho)\sigma' : \tau\sigma'$ must be derived by TYPE APP. Hence $\Gamma\sigma' \vdash e\sigma' : \forall\alpha.\hat{\tau}$ for some α and $\hat{\tau}$ with $\hat{\tau}[\rho\sigma'/\alpha] = \tau\sigma'$. So $\Gamma\sigma'' \vdash e\sigma'' : \delta\sigma''$ for a new variable δ and $\sigma'' := \sigma'[\forall\alpha.\hat{\tau}/\delta]$. From $\sigma \preceq_N^{(\Gamma, e)} \sigma'$ follows $\sigma \preceq_N^{(\Gamma, e)} \sigma''$. According to the induction hypothesis $\mathcal{T}(\Gamma, e, \delta, \sigma)$ creates new variables N_0 and returns σ_0 with $\sigma_0 \preceq_{N \cup N_0}^{(\Gamma, e, \delta)} \sigma''$ and $\Gamma\sigma_0 \vdash e\sigma_0 : \delta\sigma_0$.

Hence there exists a substitution $\hat{\sigma}_0$ with $\sigma''{=}_{|N \cup N_0}\, \sigma_0\hat{\sigma}_0$ and for all $\delta' \in \text{freeTyVar}(\Gamma\sigma_0, e\sigma_0, \delta\sigma_0)$ the type $\delta'\hat{\sigma}_0$ is a non-$\forall$-type. Therefore $\delta\sigma_0 = \forall\alpha.\tau_0$ for a τ_0 with $\tau_0\hat{\sigma}_0 = \hat{\tau}$.

From the above follows:

$$\begin{aligned}\tau\sigma_0\hat{\sigma}_0 = \tau\sigma'' = \tau\sigma' = \hat{\tau}[\rho\sigma'/\alpha] \\ = \tau'\hat{\sigma}_0[\rho\sigma''/\alpha] = \tau'\hat{\sigma}_0[\rho\sigma_0\hat{\sigma}_0/\alpha]\end{aligned}$$

According to Lemma A.3 the last type is equal to $\tau'[\rho\sigma_0/\alpha]\hat{\sigma}$. So $\hat{\sigma}_0$ is a unifier of $\tau\sigma_0 \doteq \tau'[\rho\sigma_0/\alpha]$ and hence $\hat{\mathcal{U}}(\tau\sigma_0 \doteq \tau'[\rho\sigma_0/\alpha])$ returns σ_1 with $\sigma_1 \preceq \hat{\sigma}_0$ and $\tau\sigma_0\sigma_1 = (\tau_0[\rho\sigma_0/\alpha])\sigma_1$.

We have $\tilde{\sigma} = \sigma_0\sigma_1$. Together with $\sigma_0 \preceq_{N \cup N_0}^{(\Gamma, e, \delta)} \sigma''$ follows That $\mathcal{T}(\Gamma, e\, \rho, \tau, \sigma)$ creates new variables $\tilde{N} := N_0 \cup \{\delta\}$ and returns $\tilde{\sigma}$ with $\tilde{\sigma} = \sigma_0\sigma_1 \preceq_{N \cup N_0 \cup \{\delta\}}^{(\Gamma, e, \delta)} \sigma'$.

With Lemma A.3 follows $\tau\tilde{\sigma} = \tau_0\sigma_1[\rho\tilde{\sigma}/\alpha]$. From $\forall\alpha.\tau_0 = \delta\sigma_0$ follows $\delta\tilde{\sigma} = \forall\alpha.\tau_0\sigma_1$. Together with the Substitution Lemma A.2 follows $\Gamma\tilde{\sigma} \vdash e\tilde{\sigma} : \forall\alpha.\tau_0\sigma_1$. Therefore we can apply TYPE APP to obtain $\Gamma\tilde{\sigma} \vdash (e\, \rho)\tilde{\sigma} : \tau\tilde{\sigma}$.

The proofs for the remaining cases are similar to the given ones. ∎