Citation for published version


DOI

Link to record in KAR

http://kar.kent.ac.uk/21702/

Document Version

UNSPECIFIED

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
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 \( \lambda \)-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 \( \lambda \)-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]:

\[
\text{any} \ p \ xs = \text{or} \ (\text{map} \ p \ xs)
\]

The function \( \text{map} \) applies \( p \) to all elements of \( xs \) yielding a list of boolean values. The function \( \text{or} \) 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 \( \text{or} \) and becomes garbage, which can be reclaimed. Thus the function \( \text{any} \) runs in constant space. Furthermore, when \( \text{or} \) 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 \( \text{any} \) is more efficient:

\[
\begin{align*}
\text{any} \ p \ [] &= \text{False} \\
\text{any} \ p \ (x:xs) &= p \ x \| \text{any} \ p \ xs
\end{align*}
\]

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

\[
\text{foldr} \ e_1 \ e_2 \ (\text{build} \ g) \leadsto g \ e_1 \ e_2\]

where \( \text{build} \) is a new function with a second-order type:

\[
\text{build} :: \forall \alpha . (\forall \gamma . (\alpha \rightarrow \gamma \rightarrow \gamma) \rightarrow \gamma) \rightarrow [\alpha]
\]

\[
\text{build} \ g = g \ (\cdot) \ []
\]

Short cut deforestation removes an intermediate list that is produced by the expression \( \text{build} \ g \) and consumed by the function \( \text{foldr} \). 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 \( \text{build} \) and \( \text{foldr} \). For example the definition of \( \text{map} \) is:

\[
\text{map} \ f \ xs = \text{build} \ (\lambda \ c \ n \rightarrow \text{foldr} \ (c \ . \ f) \ n \ xs)
\]
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

\[
\text{build } (\forall \ c \ n \rightarrow \text{foldr } c \ n \ e)
\]

and push the foldr into the e by term rewriting, hoping that it cancels with builds 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 \(\lambda\)-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 \(\lambda\)-calculus is undecidable [Wel94] seems to pose a problem, because using build is generally considered as good, modular programming style, whereas using foldr is similar to the intermediate language Core used in- side 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 e and the empty list constructor \([\ ]\) by a constant n:

\[
\text{foldr } c \ n \ [x_1, \ldots, x_k] = x_1 \cdot c'((x_2 \cdot c'(\ldots (x_k \cdot c' n)\ldots)))
\]

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:

\[
\text{foldr } e @ (c, n) \ e \quad \leadsto \quad e[e_{c}]/(c) [e_{0}/[\ ]]
\]

Unfortunately this rule is wrong:

\[
\text{foldr } (\text{map } p (1, 2)) \neq \text{map } p (1 || (2 || \text{False}))
\]

In this example the constructors in the definition of map, which is not part of the expression, needed to be replaced. Otherwise the transformation is abandoned. This method is rather expensive and poses substantial problems for an implementation.

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 e and the empty list constructor \([\ ]\) by a constant n:

\[
\text{foldr } c \ n \ [x_1, \ldots, x_k] = x_1 \cdot c'((x_2 \cdot c'(\ldots (x_k \cdot c' n)\ldots)))
\]

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:

\[
\text{foldr } e @ (c, n) \ e \quad \leadsto \quad e[e_{c}]/(c) [e_{0}/[\ ]]
\]

Unfortunately this rule is wrong:

\[
\text{foldr } (\text{map } p (1, 2)) \neq \text{map } p (1 || (2 || \text{False}))
\]

In this example the constructors in the definition of map, which is not part of the expression, needed to be replaced. Otherwise the transformation is abandoned. This method is rather expensive and poses substantial problems for an implementation.

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 e and the empty list constructor \([\ ]\) by a constant n:

\[
\text{foldr } c \ n \ [x_1, \ldots, x_k] = x_1 \cdot c'((x_2 \cdot c'(\ldots (x_k \cdot c' n)\ldots)))
\]

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:

\[
\text{foldr } e @ (c, n) \ e \quad \leadsto \quad e[e_{c}]/(c) [e_{0}/[\ ]]
\]

Unfortunately this rule is wrong:

\[
\text{foldr } (\text{map } p (1, 2)) \neq \text{map } p (1 || (2 || \text{False}))
\]

In this example the constructors in the definition of map, which is not part of the expression, needed to be replaced. Otherwise the transformation is abandoned. This method is rather expensive and poses substantial problems for an implementation.

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 e and the empty list constructor \([\ ]\) by a constant n:

\[
\text{foldr } c \ n \ [x_1, \ldots, x_k] = x_1 \cdot c'((x_2 \cdot c'(\ldots (x_k \cdot c' n)\ldots)))
\]

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:

\[
\text{foldr } e @ (c, n) \ e \quad \leadsto \quad e[e_{c}]/(c) [e_{0}/[\ ]]
\]

Unfortunately this rule is wrong:

\[
\text{foldr } (\text{map } p (1, 2)) \neq \text{map } p (1 || (2 || \text{False}))
\]

In this example the constructors in the definition of map, which is not part of the expression, needed to be replaced. Otherwise the transformation is abandoned. This method is rather expensive and poses substantial problems for an implementation.
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 (λc 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 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 : τ. The operator + combines two type environments under the assumption that their domain is disjoint. We abbreviate Γ + {x : τ} by Γ + x : τ. The language does not have explicit definitions of algebraic data types like data T = c_1 T_1 | ... | c_k T_k. Such a definition is implicitly expressed by having the data constructors in the type environment: Γ(e_i) = T_{c_i} = ... = T_{n_i} = T = T_{γ_1} → T_{γ_2} → 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 [LY98], which is for the Hindley-Milner type system. Algorithm [LY98] 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) if σ e : τ. Furthermore, σ is a principal typing for (Γ, e, τ) if it is a typing for (Γ, e, τ) and for every typing σ' for (Γ, e, τ) there exists a substitution σ such that Γσ' = Γσ, eσ' = eσ and τσ' = τσ. Our type inference algorithm 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 σ ≤ σ', if there exists a substitution such that σ' = σσ. We say that σ 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 σ ≤ σ' there exists a substitution σ such that Γσ' = Γσ, eσ' = eσ and τσ' = τσ.

Theorem 3.1 Let Γ be an environment, e an expression and τ a type. If (Γσ, eσ, τσ) has a typing, then T(Γ, e, τ, σ)
proved in [LY98]. Similar to the soundness and completeness of \( \mathcal{M} \) yields a principal typing \( \hat{\sigma} \) with respect to \( \sigma \) for \( (\Gamma, e, \tau) \). Otherwise it fails. \[ \Box \]

**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 \( \sigma \) is a unifier for an equation \( \tau_1 \approx \tau_2 \) iff \( \tau_1 \sigma \approx \tau_2 \sigma \). Moreover, \( \hat{\sigma} \) is a most general unifier for \( \tau_1 \approx \tau_2 \) iff it is a unifier for \( \tau_1 \approx \tau_2 \) and for every unifier \( \sigma' \) for \( \tau_1 \approx \tau_2 \) we have \( \sigma \preceq \sigma' \).

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

We choose to formulate \( T \) 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.

\[
\begin{align*}
&\{g : \text{Int} \rightarrow \text{Int}. 1 : \text{Int}, 2 : \text{Int}, \}
&\quad \vdash \text{let map : } (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})
&\quad \rightarrow \text{Int} \rightarrow \text{Int}\}
&\quad = \lambda f : \text{Int} \rightarrow \text{Int}. \lambda xs : [\text{Int}].
&\quad \quad \text{case xs of }
&\quad \quad \quad [] \rightarrow []
&\quad \quad \quad y : \text{ys} \rightarrow (f y) \rightarrow (\text{map } f \text{ ys})
&\quad \text{in map } g \{1, 2\} : [\text{Int}]
\end{align*}
\]

Our algorithm replaces every list constructor \((\cdot)\), 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 \([\text{Int}]\) in the expression by a new type inference variable. Furthermore, we add \(c_i : \tau \rightarrow \delta_i \rightarrow \delta_i\), respectively \(n_i : \delta_i\), to the type environment, where \(\delta_i\) 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 \((\cdot)\) or the consumer argument \(e_i\), respectively \([],\) or \(e_1\). As remaining arguments we give \(T\) a type inference variable \(\gamma\) and the identity substitution. For our example, input and output of \(T\) look as follows

\[
\begin{align*}
&T(\{g : \text{Int} \rightarrow \text{Int}. 1 : \text{Int}, 2 : \text{Int},
&\quad (\cdot) : \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}], [] : [\text{Int}],
&\quad c_1 : \text{Int} \rightarrow \delta_1 \rightarrow \delta_1, c_2 : \text{Int} \rightarrow \delta_2 \rightarrow \delta_2,
&\quad c_3 : \text{Int} \rightarrow \delta_3 \rightarrow \delta_3, n_1 : \delta_4, n_2 : \delta_5\}
&\quad \vdash \text{let map : } (\text{Int} \rightarrow \text{Int}) \rightarrow \delta_6 \rightarrow \delta_7
&\quad = \lambda f : \text{Int} \rightarrow \text{Int}. \lambda \delta_8.
&\quad \quad \text{case xs of }
&\quad \quad \quad [] \rightarrow n_1,
&\quad \quad \quad y : \text{ys} \rightarrow c_1 y \rightarrow (\text{map } f \text{ ys})
&\quad \text{in map } g \{c_2 1 (c_3 2 n_2)\}
&\quad \quad \quad \gamma, \text{id}
&\quad \quad \quad \rightarrow \gamma_0 \approx [\gamma/\delta_1][\delta_2]/[\text{Int}] / [\text{Int}] / [\text{Int}]/\delta_8
&\quad \rightarrow \gamma_1 \approx [\gamma/\delta_0][\gamma/\delta_1]/[\text{Int}]/\delta_8
\end{align*}
\]

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

\[
\begin{align*}
&T(\{g : \text{Int} \rightarrow \text{Int}. 1 : \text{Int}, 2 : \text{Int},
&\quad (\cdot) : \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}], [] : [\text{Int}],
&\quad c_1 : \text{Int} \rightarrow \gamma \rightarrow \gamma, c_2 : \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}],
&\quad c_3 : \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}], n_1 : \gamma, n_2 : [\text{Int}]\}
&\quad \vdash \text{let map : } (\text{Int} \rightarrow \text{Int}) \rightarrow \gamma
&\quad = \lambda f : \text{Int} \rightarrow \text{Int}. \lambda \gamma : [\text{Int}].
\end{align*}
\]
case xs of 
  [] → n1,
  y:ys → c1 y (map f ys)

In map g (c2 1 (c1 2 n2)) : γ

The type environment tells us that c1 and n1 construct the result of the expression whereas c2, c3, and n2 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 e1 e2 the variables c1 and n1 are replaced by e1 and e2. The other variables are turned back into list constructors. For example, fusion of our producer with the consumer foldr (+) 0 gives:

\[
\begin{align*}
\{g : \text{Int} \rightarrow \text{Int}, & \quad 1 : \text{Int}, \\
& \quad 2 : \text{Int}, \quad \text{c} : [\text{Int}], \\
& \quad (+) : \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}, \quad 0 : \text{Int}\} \\
\vdash \text{let map} : (\text{Int} \rightarrow \text{Int}) \rightarrow [\text{Int}] \rightarrow \text{Int}
\end{align*}
\]

\[
= \lambda f : \text{Int} \rightarrow \text{Int}. \quad \lambda xs : [\text{Int}].
\]

\[
\begin{align*}
\text{case xs of} \\
\quad [] & \rightarrow 0,
\end{align*}
\]

\[
\quad y : ys \rightarrow y + (\text{map } f y)
\]

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:

\[
\{\text{map} : (\text{Int} \rightarrow \text{Int}) \rightarrow \delta \rightarrow \gamma, \\
\text{g} : \text{Int} \rightarrow \text{Int}, \quad 1 : \text{Int}, \quad 2 : \text{Int}, \\
\text{n} : \delta, \quad \text{c} : \text{Int} \rightarrow \delta \rightarrow \delta, \quad \text{c} : \text{Int} \rightarrow \delta \rightarrow \delta \}
\]

\[
\vdash \text{map } g (\text{c}1 1 (\text{c}2 2 \text{n}2)) : \gamma
\]

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 c1 and n1 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 practice, the algorithm will already abandon list constructor abstraction when the inlined code exceeds a predefined size.
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 $\alpha$ instead of $\Box \alpha$ for the polymorphic list type and that $\Gamma((::)) = \forall \alpha. \alpha \rightarrow [\alpha] \rightarrow [\alpha]$ and $\Gamma([]) = \forall \alpha. [\alpha]$. The functions build and foldr are defined as follows

$$\text{build} : \forall \alpha. (\forall \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha]$$
$$\text{foldr} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$$

and the foldr/build rule takes the form:

$$\text{foldr} \quad \text{case \ xs of \ [\ ] \ \rightarrow n}$$
$$y : y s \rightarrow c \ y \ (\text{foldr} \ \alpha \ \beta \ c \ n \ y s)$$

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 $\sigma, \delta, \sigma_0, \sigma_1$, etc. denote substitutions that only replace type inference variables.

Note that in general $\mathcal{T}$ is neither sound nor complete. In the case of $\lambda \alpha. e$, $\mathcal{T}$ does 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 $\mathcal{T}(\{x : \delta\}, x \rho, \text{Int}, \text{id})$ 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 $\Gamma$ be an environment, $e$ an expression and $\tau$ and $\rho$ types. If there exists a substitution $\sigma'$ such that $\delta' = [\rho] \delta$ for all $\delta \in \text{freeTyVar}(\Gamma \sigma, \sigma \tau)$ such that $\sigma'$ is a typing for $(\Gamma \sigma, \sigma \tau)$, then $\mathcal{T}(\Gamma, e, \tau, \sigma)$ yields a principal typing $\sigma$ with respect to $\sigma$ for $(\Gamma, e, \tau)$. □

**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 $(::)$, $\Box$ and $[\text{Int}]$. Instead, if $[\tau]$ is the type of the intermediate list producer, we replace every $(::)$ $\tau$, respectively $\Box \tau$, by a new $e_i$, respectively $n_i$, and replace every type $[\tau]$ 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
\texttt{map : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta],}
prevent list constructor abstraction, because the type annotations contain polymorphic list types like \([\beta]\) instead of \([\tau]\).

Hence we replace all occurrences of a type application \(f \tau\), where \(f\) is an invariable with \(f : \forall \tau. \pi = \lambda x.e\), and \(\tau\) contains a polymorphic list type \([\alpha]\) with \(\alpha \in \pi\), by a new invariable variable \(f_\alpha\) that is defined through \(\forall \tau. \pi = \tau\). This type instantiation is performed before the list types \([\tau]\) 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:

\[
\begin{align*}
\{f_{\text{st}} : & \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \alpha, \text{ zs : } ([\text{Bool}], \text{ Char}], \\
\text{unzip : } & \forall \alpha. \forall \beta. ([\alpha], [\beta]) \rightarrow ([\alpha], [[\beta]]) \\
\end{align*}
\]

\[
\vdash \text{fst} [\text{Bool} \ [\text{Char}]] \ (\text{unzip} \ \text{Bool} \ \text{Char} \ \text{zs}) : [\text{Bool}]
\]

Our instantiation of polymorphism replaces all occurrences of \text{unzip} \ \text{Bool} \ \text{Char} by a monomorphic \text{unzip}':

\[
\begin{align*}
\{f_{\text{st}} : & \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \alpha, \text{ zs : } ([\text{Bool}], \text{ Char}], \\
\text{unzip' : } & \ ([\text{Bool}], \text{ Char}]) \rightarrow ([\text{Bool}], [[\text{Char}]]) \\
\vdash \text{fst} [\text{Bool} \ [\text{Char}]] \ (\text{unzip'} \ \text{zs}) : [\text{Bool}]
\end{align*}
\]

Then we replace all occurrences of \text{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.

\[
T([\text{fst} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \alpha, \text{ zs : } [\text{Bool}], \text{ Char}], \\
\text{unzip' : } \ ([\text{Bool}], \text{ Char}]) \rightarrow ([\text{Bool}], [[\text{Char}]]) \\
, \text{fst} \delta_2 \ [\text{Char}] \ (\text{unzip'} \ \text{zs}) , \gamma, \ \text{id}) = [\gamma/\delta_1][\gamma/\delta_2]
\]

The \(\gamma\) in the inferred type of \text{unzip}' indicates that its definition must be inlined. The original right hand side of the definition of \text{unzip} is:

\[
\begin{align*}
\{\text{unzip} : & \forall \alpha. \forall \beta. ([\alpha], [\beta]) \rightarrow ([\alpha], [[\beta]]) \\
\text{foldl} : & \forall \alpha. \forall \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \\
(\_ : & \forall \alpha. \alpha \rightarrow [\alpha], [] : \forall \alpha. [\alpha], \\
\_ : & \forall \alpha. \beta \rightarrow (\alpha, \beta) \\
\vdash \lambda \alpha. \lambda \beta. \text{foldl} (\alpha, \beta) ([\alpha], [\beta]) \ (\lambda y : (\alpha, \beta), \lambda u : ([\alpha], [\beta])). \ \text{case y of } \{ \\
(v, w) \rightarrow & \text{ case u of } \{ \\
(vs, ws) \rightarrow & (\_, [\alpha] [\beta] ((\_, [\alpha] [\beta]) ((\,, : (\alpha, w) vs))) ((\,, : (\beta, w) vs))) \\
(\_, [\alpha] [\beta]) (\_, [\alpha] [\beta]) : & \forall \alpha. \forall \beta. ([\alpha], [\beta]) \rightarrow ([\alpha], [\beta])
\}
\end{align*}
\]

For processing \text{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.

\[
T([\text{unzip'} : \ ([\text{Bool}], \text{ Char}]) \rightarrow (\delta_1, \text{ [Char]}), \\
\text{foldl} : \forall \alpha. \forall \beta. (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \\
(\_ : & \forall \alpha. \alpha \rightarrow [\alpha], [] : \forall \alpha. [\alpha], \\
\_ : & \forall \alpha. \beta \rightarrow (\alpha, \beta) \\
\vdash \lambda \alpha. \lambda \beta. \text{foldl} (\alpha, \beta) (\delta_1, \text{ [Char]}). \ (\lambda y : (\alpha, \text{ [Char]}), \lambda u : (\delta_1, \text{ [Char]}). \ \text{case y of } \{ \\
(v, w) \rightarrow & \text{ case u of } \{ \\
(vs, ws) \rightarrow & (\_, \delta_1 [\text{Char}] (c_1 : v vs))
\}
\]

\[
\begin{align*}
((\_, [\alpha] [\beta]) ((\,, : (\alpha, w) vs))) ((\,, : (\beta, w) vs))) \\
((\_, [\alpha] [\beta]) (\_, [\alpha] [\beta]) : & \forall \alpha. \forall \beta. ([\alpha], [\beta]) \rightarrow ([\alpha], [\beta])
\}
\end{align*}
\]

\[
\text{There is no } \gamma \text{ in the type of any further variable of the type environment. Hence list constructor abstraction has terminated successfully.}
\]

We can now express the producer in \texttt{build form} as follows:

\[
\begin{align*}
\{\text{build} : & \forall \alpha. (\forall \beta. (\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow \beta \\
\text{fst : } & \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \alpha, \text{ zs : } [\text{Bool}], \text{ Char}], \\
\text{foldl : } & \forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \beta \\
(\_ : & \forall \alpha. \alpha \rightarrow [\alpha], [] : \forall \alpha. [\alpha], \\
(\_ : & \forall \alpha. \beta \rightarrow (\alpha, \beta) \\
\vdash \lambda \alpha. \lambda \beta. \text{foldl} (\alpha, \beta) ([\alpha], [\beta]). \ (\lambda y : (\alpha, \text{ [Char]}), \lambda u : ([\alpha], [\beta]). \ \text{case y of } \{ \\
(v, w) \rightarrow & \text{ case u of } \{ \\
(vs, ws) \rightarrow & (\_, \beta [\text{Char}] (c_1 : v vs))
\}
\end{align*}
\]

\[
\text{Let } \text{unzip}' : \ ([\text{Bool}], \text{ Char}) \rightarrow (\beta, \text{ [Char]}). \ (\lambda y : \text{ [Char]}). \ (\lambda u : ([\alpha], [\beta]). \ \text{case y of } \{ \\
(v, w) \rightarrow & \text{ case u of } \{ \\
(vs, ws) \rightarrow & (\_, \beta [\text{Char}] (c_1 : v vs))
\}
\]

\[
\begin{align*}
\text{in } \text{fst} \beta \ [\text{Char}] \ (\text{unzip'} \ \text{zs}) : & [\text{Bool}]
\end{align*}
\]

Note that the producer \texttt{fst [Bool] [Char] (unzip Bool Char zs)} cannot be fused by standard short cut deforestation, because neither \texttt{fst} nor \texttt{unzip} can individually be expressed in \texttt{build form}. Note also that the definition of \texttt{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 \texttt{foldr \ r_1 \ r_2 \ e \ (\_ \ e \ \_ \ e)}. Furthermore, we collect all definitions from \texttt{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 \rightarrow in their type.

Suppose we find the following potentially fusible expression:

\[
\begin{align*}
\text{foldr Bool Bool (\&\& True) (fst [Bool] [Char] (unzip Bool Char zs))}
\end{align*}
\]

For every such expression we try to convert its producer into \texttt{build form}.

7
After this pairs in the second, actual deforestation pass.

Finally, a few \( \beta \)-reductions put the arguments \( e_1 \) and \( e_0 \) of the consumer in the former places of the list constructors.

\[
\text{let unzipped': } ([\text{Bool,Char}]) \rightarrow ([\text{Bool,Char}]) = \text{foldr}\ (\text{Bool,Char}) ([\text{Bool,Char}]). \lambda y : ([\text{Bool,Char}]). \lambda ys : ([\text{Bool,Char}]). \text{case } y \text{ of } \{(v, w) \rightarrow \emptyset\ (\text{unzip' } y) \text{ of } \{(v, w) \rightarrow (, ) \text{ Bool } [\text{Char}]) (v \& \& vs) ((:) \text{ Char } w ws)\}}\}
\]
\[
\[((:) \text{ Bool } [\text{Char}]) \text{ True } ([\text{Char}])\)
\]

The idea immediately suggests itself that we could directly replace the right list constructors by \( e_1 \) and \( e_0 \) 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 \( \beta \)-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

\[
\text{map } \text{sum} \ (\text{inits } [1..])
\]

The function inits returns the list of all prefixes of its argument, that is, \([1, 1, 1, 2, \ldots] \). 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 \([1, 1, 1, 2, \ldots]\) 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 \( (++\) \( \tau \) \( xs \) \( ys \)\) 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

\[
\text{augment: } \forall \alpha. (\forall \beta. (\alpha \rightarrow \beta) \rightarrow \beta) \rightarrow ([\alpha] \rightarrow [\alpha])
\]

\[
\lambda x : [\alpha]. \ g\ ([\alpha]) (\cdot (\cdot) \alpha) \ xs
\]

and the foldr/build rule

\[
\text{foldr } \tau_1 \tau_2 e_1 (e_0 (\text{augment } \tau_1 g (xs))) \rightarrow g \tau_2 e_1 (\text{foldr } \tau_1 \tau_2 e_1 (e_0 (xs))
\]

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 \( (++\) \( \tau \) \( xs \) \( ys \)\). Type inference indicates that the definition of \( (++\)\) needs to be inlined. After type inference of the definition, the type environment contains \( \gamma \) but \( ys \) may not be inlinable, because it is shared or \( \lambda \)-bound. Then this non-inlinable variable becomes the list argument of augment. So the augment form of our example is:

\[
\text{augment } \tau (\lambda \beta. \lambda x : \tau \rightarrow \beta \rightarrow \beta. \lambda n : \beta)
\]

\[
\text{let } (++\) \tau = \lambda y : \tau. \lambda x : \tau. \lambda a : [\alpha]. \ g (\lambda \beta. (\alpha \rightarrow \beta) \rightarrow \beta) n y
\]

The foldr/build rule is a special instance of the foldr/augment rule, we only have to replace \( \tau \) by \([\cdot] \) and \( \beta \)-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:

\[
\text{data RoseTree } a = \text{Node } a [\text{RoseTree } a]
\]

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

\[
\text{Node : } \forall \alpha. a \rightarrow [\text{RoseTree } a] \rightarrow (\text{RoseTree } a)
\]

\[
\text{buildRT } : \forall \alpha. (\forall \beta. (\alpha \rightarrow [\beta]) \rightarrow [\beta]) \rightarrow (\text{RoseTree } a)
\]

\[
\lambda x : [\alpha]. \ g (\lambda \beta. (\alpha \rightarrow [\beta]) \rightarrow [\beta]) (\cdot (\cdot) \alpha) (\text{Node } a)
\]

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

Furthermore, an algebraic data type comes with a cata-morphism which consumes its elements in a regular way (see e.g. [MH95] for how a catamorphism is defined in general):
The foldRT/buildRT rule is an instance of the parametricity theorem of the type of buildRT:

\[
\text{foldRT} : \forall \alpha \forall \beta . (\alpha \rightarrow [\beta] \rightarrow \beta) \rightarrow (\text{RoseTree}\, \alpha) \rightarrow \beta
\]

\[
= \lambda \alpha. \lambda \beta. \lambda \alpha. \alpha \rightarrow [\beta] \rightarrow \beta. \lambda \alpha. \text{RoseTree}\, \alpha.
\]

\[
\text{case } t \text{ of } \{ \text{Node } x \text{ ts} \rightarrow
\]

\[
\quad n \times (\text{map } (\text{RoseTree}\, \alpha) \beta)
\]

\[
= \text{foldRT } \alpha \beta n \text{ ts}
\]

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

\[
\text{foldRT } \tau_1 \tau_2 \text{ cnode } (\text{buildRT } \tau_1 g) \rightarrow g \tau_2 \text{ cnode}
\]

Obviously, we can derive a buildRT form from an expression of type RoseTree \(\tau\) 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 warml 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 \(\gamma\) 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

\[
\text{data Twist } a \ b = \text{Nil } | \text{Cons } a (\text{Twist } b \ a)
\]

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:

\[
\text{data Infinite } a = \text{I } (\text{Infinite } a \rightarrow a)
\]

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 [Pit87]. 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 [Ta98]. [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
Acknowledgements

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

References


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 $\sigma$, that is, that for any typing $\sigma'$ for $(\Gamma, e, \tau)$ there exists $\delta$ such that $\Gamma' = \Gamma \delta \sigma$, $e\sigma' = e\delta\sigma$ and $\tau' = \tau\delta\sigma$, is too weak for an induction. On the other hand, it is not true for the result $\sigma$ of $T$, that there exists $\delta$ such that $\sigma' = \delta\sigma$, because the call $T(\Gamma, e, \tau, \sigma)$ generally creates new variables which $\sigma'$ may substitute differently from $\delta$. However, we can prove that $\delta\sigma' = \delta\delta\sigma$ for all type inference variables $\delta$ except the newly created ones.

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

A non-$\forall$-type is a type that does not have a $\forall$ at the outermost level. We write $\sigma \preceq_{\mathcal{N}} \sigma'$ iff there exists $\delta$ with $\sigma \equiv_{\mathcal{N}} \delta\delta\sigma$ and for all $\delta \in \operatorname{freeTyVar}(M\delta)$ the type $\delta\delta\sigma$ is a non-$\forall$-type. The idea is that for extending $\sigma$ to obtain $\sigma'$ the variables in $M\delta$ need only be replaced by non-$\forall$-types.

Remember that $T$ treats type variables like constants, only type inference variables may be replaced. Hence $\sigma, \delta, \sigma_0, \ldots$ denote substitutions that only replace type inference variables.

Theorem A.1 (Soundness and Completeness of $T$)

Let $\sigma'$ be typing for $(\Gamma, e, \tau)$ with $\sigma \preceq_{\mathcal{N}} (\Gamma, e, \tau, \sigma)$ for some variables $N$ not appearing in $\Gamma, e$ or $\tau$. Then $T(\Gamma, e, \tau, \sigma)$ creates new variables $\tilde{N}$ and returns $\tilde{\sigma}$ with $\sigma \preceq \tilde{\sigma}$, $\tilde{\sigma} = \tilde{\tilde{\sigma}} \equiv_{\mathcal{N}} \sigma'$ and $\Gamma\tilde{\sigma} = e\tilde{\sigma} : \tau\tilde{\sigma}$. □

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 $\sigma$. □

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

Lemma A.3 Let $\tau_1, \tau_2, \alpha$ and $\sigma$ such that $\alpha \notin \operatorname{freeTyVar}(\tau_1\sigma)$. Then $\tau_1\sigma[\tau_2/\alpha] = (\tau_1[\tau_2/\alpha])\sigma$. □

Proof By induction on $\tau_1$, using that $\sigma$ replaces no type variable $\alpha$.

Lemma A.4 If $\sigma_1 \preceq_{\mathcal{M}} \sigma_2 \preceq_{\mathcal{N}} \sigma_3$, then $\sigma_2 \preceq_{\mathcal{M}} \sigma_3$. □

Proof Directly from the definition of $\preceq_{\mathcal{M}}$. □

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

1. If $\mathcal{U}(\tau_1 \triangleleft \tau_2, \sigma)$ returns $\tilde{\sigma}$, then $\sigma \preceq \tilde{\sigma}$.
2. If $\mathcal{U}(\tau_1 \triangleleft \tau_2, \sigma)$ returns $\tilde{\sigma}$, then $\tau_1\tilde{\sigma} = \tau_2\tilde{\sigma}$.
3. If $\tau_1\sigma' = \tau_2\sigma'$ and $\sigma \preceq_{\mathcal{N}} \sigma'$ then $\mathcal{U}(\tau_1 \triangleleft \tau_2, \sigma)$ returns $\tilde{\sigma}$ with $\sigma \preceq_{\mathcal{N}} \tilde{\sigma}$. □

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

Note that algorithm $T$ 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 $T$ by structural induction on the term argument, applying the induction hypothesis to the recursive calls of $T$.

Lemma A.6 ($T$ extends substitutions)

If $T(\Gamma, e, \tau, \sigma)$ 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 $\forall a. e$ soundness requires the existence of a typing. We could avoid this complication by adding a test to $T$ which corresponds to the condition $\alpha \notin \operatorname{freeTyVar}(\Gamma)$ in $\operatorname{typeAbs}$. However, we want to prove that for our purpose such a test is not necessary.

Lemma A.7 Let $\sigma'$ be typing for $(\Gamma, e, \tau)$ with $\sigma \preceq_{\mathcal{N}} (\Gamma, e, \tau, \sigma)$ for some variables $N$ not appearing in $\Gamma, e$ or $\tau$. Then $T(\Gamma, e, \tau, \sigma)$ creates new variables $\tilde{N}$ and returns $\tilde{\sigma}$ with $\tilde{\tilde{\sigma}} \equiv_{\mathcal{N}} \sigma'$ and $\Gamma\tilde{\sigma} = e\tilde{\sigma} : \tau\tilde{\sigma}$. □

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 $\operatorname{var}$. Hence $\Gamma\tilde{\sigma}'(x) = \tau\tilde{\tilde{\sigma}}$. Furthermore $\sigma \preceq_{\mathcal{N}} (\Gamma, x, \sigma)$. According to Lemma A.5 $\hat{\mathcal{U}}(\Gamma(x) \triangleleft \tau, \sigma)$ returns $\tilde{\sigma}$ with $\sigma \preceq_{\mathcal{N}} \sigma'$ and $\Gamma(x)\tilde{\sigma} = \tau\tilde{\tilde{\sigma}}$. Hence $\tilde{\tilde{\sigma}} \equiv_{\mathcal{N}} (\Gamma, e, \tau, \sigma)$. Furthermore with rule $\operatorname{var}$ follows $\Gamma\tilde{\sigma} = x\tilde{\sigma} : \tau\tilde{\tilde{\sigma}}$.

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

The typing $\Gamma\sigma' \vdash (\lambda(x : \tau_1). e)\sigma' : \tau_1\sigma'$ must be derived by $\operatorname{termAbs}$. Hence exists $\tilde{\tau}$ with $\tau_1\sigma' = \tilde{\tau}\sigma' \rightarrow \tilde{\tau}$ and $\Gamma\sigma' + (x : \tau_1)\tilde{\tau}\sigma' = e\tilde{\tau}\sigma' : \tilde{\tau}$, that is, $\sigma'' = (\tau_1 \rightarrow \delta)\sigma''$ and $\Gamma + (x : \tau_1)\tilde{\tau}\sigma' = e\tilde{\tau}\sigma' : \delta\tilde{\tau}\sigma''$ for a new variable $\delta$ and $\sigma'' := \sigma'[\delta/\tilde{\tau}]$. From $\sigma \preceq_{\mathcal{N}} (\lambda(x : \tau_1). e)\sigma'$ follows $\sigma \preceq_{\mathcal{N}} (\lambda(x : \tau_1). \tilde{\tau}\sigma') \sigma''$. According to Lemma A.5 $\hat{\mathcal{U}}(\tau_1 \rightarrow \tau_2, \sigma)$ creates new variables $\tilde{N}$ and returns $\tilde{\tilde{\sigma}}$ with $\tilde{\tilde{\sigma}} \equiv_{\mathcal{N}} (\lambda(x : \tau_1). \tilde{\tau}\sigma') \sigma''$. According to Lemma A.5 $\hat{\mathcal{U}}(\tau_1 \rightarrow \tau_2, \sigma)$
\( \delta \vdash \tau, \sigma \) returns \( \sigma_0 \) with \( \sigma_0 \leq_N (\Gamma, \lambda(x: \tau_1), e) \sigma'' \) and \( (\tau_1 \rightarrow \delta) \sigma_0 = \tau_0 \sigma_0 \).

It follows that \( \sigma_0 \leq_N (\Gamma + x: \tau_1, e) \sigma'' \). According to the induction hypothesis \( T(\Gamma + x: \tau_1, e, \delta, \sigma_0) \) creates new variables \( N_1 \) and returns \( \hat{\sigma} \) with \( \hat{\sigma} \leq_N (\Gamma + x: \tau_1, e, \delta) \sigma'' \) and \( (\Gamma + x: \tau_1, e, \delta) \hat{\sigma} \vdash \tau_0 \hat{\sigma} \).

According to Lemma A.6 \( \sigma_0 \leq \hat{\sigma} \). Hence \( (\tau_1 \rightarrow \delta) \hat{\sigma} = \tau \hat{\sigma} \). It follows that \( \hat{\sigma} \leq_N (\Gamma + x: \tau_1, e, \delta) \sigma'' \) and \( (\Gamma + x: \tau_1, e, \delta) \hat{\sigma} \vdash \tau \hat{\sigma} \).

Case \( e_1, e_2 \).

The typing \( \Gamma \vdash (e_1, e_2) \sigma'' : \tau \sigma'' \) must be derived by rule TERM APP. Hence \( \hat{\tau}_2 \) with \( \Gamma \vdash \hat{e}_1 \sigma'' : \hat{\tau}_2 \rightarrow \tau \sigma'' \) and \( \Gamma \vdash \hat{e}_2 \sigma'' : \hat{\tau}_2 \), that is, \( \tau \sigma'' : (\delta \rightarrow \tau) \sigma'' \) and \( \Gamma \vdash \hat{e}_2 \sigma'' : \delta \sigma'' \) for a new variable \( \delta \) and \( \sigma'' := \sigma''[\hat{\tau}_2/\delta] \). From \( \sigma \leq_N (\Gamma, e_1, e_2) \sigma'' \) follows \( \sigma \leq_N (\Gamma, e_1, e_2) \sigma''' \). So according to the induction hypothesis \( T(\Gamma, e_1, e_2, \delta, \delta, \sigma) \) creates new variables \( N_2 \) and returns \( \hat{\delta} \) with \( \hat{\delta} \leq_N (\Gamma, e_1, e_2, \delta) \sigma'' \) and \( (\Gamma + e_1, e_2, \delta, \delta) \hat{\delta} \vdash \delta \hat{\sigma} \).

With Lemma A.4 follows that \( \hat{\delta} \leq_N (\Gamma, e_1, e_2, \delta) \sigma'' \). Hence \( T(\Gamma, e_1, e_2, \tau, \delta, \sigma) \) creates new variables \( N := N_1 \cup N_2 \cup \{ \delta \} \) and returns \( \hat{\sigma} \) with \( \hat{\sigma} \leq_N (\Gamma, e_1, e_2, \delta) \sigma'' \). Furthermore, the Substitution Lemma A.2 assures \( \Gamma \hat{\sigma} \vdash \hat{e}_1 \hat{\delta} : (\delta \rightarrow \tau) \hat{\sigma} \). Therefore we can apply TERM APP to obtain \( \Gamma \hat{\sigma} \vdash (e_1, e_2) \hat{\delta} : \tau \hat{\sigma} \).

Case \( \lambda \alpha.e \).

The typing \( \Gamma \vdash (\lambda \alpha.e) \sigma'' : \tau \sigma'' \) must be derived by rule TYPE APP. Hence \( \Gamma \vdash \alpha : \hat{\tau} \) with \( \alpha \vdash \tau \sigma'' = \forall \alpha. \hat{\tau} \). So \( \tau \sigma'' = (\forall \alpha. \hat{\tau}) \sigma'' \) for a new type variable \( \delta \) and \( \sigma'' := \sigma''[\hat{\tau}/\delta] \). From \( \sigma \leq_N (\Gamma, \lambda \alpha.e) \sigma'' \) follows \( \sigma \leq_N (\Gamma, e) \sigma'' \). So according to Lemma A.5 \( U(\tau = \forall \alpha. \hat{\tau}) \) returns \( \sigma_0 \) with \( \sigma_0 \leq_N (\Gamma, e) \sigma'' \) and for all \( \tau_0 = (\forall \alpha. \hat{\tau}) \sigma_0 \).

According to the induction hypothesis \( T(\Gamma, e, \delta, \sigma_0) \) creates new variables \( N_1 \) and returns \( \hat{\sigma} \) with \( \hat{\sigma} \leq_N (\Gamma, e, \delta) \sigma'' \) and \( (\Gamma, e, \delta) \hat{\sigma} \vdash \delta \hat{\sigma} \).

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

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

Case \( e. \rho \).

The typing \( \Gamma \vdash (e. \rho) \sigma'' : \tau \sigma'' \) must be derived by rule TYPE APP. Hence \( \Gamma \vdash e \sigma'' : \forall \alpha. \hat{\tau} \) for some \( \alpha \) and

\( \tilde{\tau} \vdash \tilde{\tau}[\rho[\alpha]/\alpha] = \tau \sigma'' \). So \( \Gamma \sigma'' \vdash e \rho \sigma'' : \delta \sigma'' \) for a new variable \( \delta \) and \( \sigma'' := \sigma''[\forall \alpha. \tau/\delta] \). From \( \sigma \leq_N (\Gamma, e) \sigma'' \) follows \( \sigma \leq_N (\Gamma, e) \tilde{\sigma} \). According to the induction hypothesis \( T(\Gamma, e, \delta, \sigma) \) creates new variables \( N_0 \) and returns \( \sigma_0 \) with \( \sigma_0 \leq_N (\Gamma, e) \sigma'' \) and \( (\Gamma, e) \sigma_0 \vdash e \sigma_0 : \delta \sigma_0 \).

Hence there exists a substitution \( \tilde{\sigma}_0 \) with \( \sigma'' \leq \tilde{\sigma}_0 \leq_N (\forall \alpha. \sigma_0) \delta \sigma_0 \) and for all \( \delta \) in freeTyVar(\( \Gamma \sigma_0, e \sigma_0, \delta \sigma_0 \)) the type \( \delta \tilde{\sigma}_0 \) is a non-\( \forall \)-type. Therefore \( \forall \alpha. \tau_0 \sigma_0 \) for a \( \tau_0 \) with \( \tau_0 \sigma_0 = \tilde{\tau} \).

From the above follows:

\[ \tau \sigma_0 \tilde{\sigma}_0 = \tau \sigma'' = \tilde{\tau} \tilde{\sigma}_0[\rho[\alpha]/\alpha] = \tilde{\tau} \tilde{\sigma}_0[\rho \sigma_0 \tilde{\sigma}_0/\alpha] \]

According to Lemma A.3 the last type is equal to \( \tau'[\rho \sigma_0/\alpha]\tilde{\sigma}_0 \). So \( \tilde{\sigma}_0 \) is a unifier of \( \tau \sigma_0 \vdash \tau'[\rho \sigma_0/\alpha] \)

and hence \( U(\tau \sigma_0 = \tilde{\tau}'[\rho \sigma_0/\alpha]) \) returns \( \sigma_1 \) with \( \sigma_1 \leq \tilde{\sigma}_0 \) and \( \tau \sigma_0 \sigma_1 = (\tau_0[\rho \sigma_0/\alpha]) \sigma_1 \).

We have \( \sigma_0 = \sigma_0 \sigma_1 \). Together with \( \sigma_0 \leq (\forall \alpha. \sigma_0) \sigma'' \) follows \( T(\Gamma, e. \rho, \tau, \sigma) \) creates new variables \( N := N_0 \cup \{ \delta \} \) and returns \( \hat{\sigma} \) with \( \sigma = \sigma_0 \sigma_1 \leq (\forall \alpha. \sigma_0) \sigma' \).

With Lemma A.3 follows \( \tau \hat{\sigma} = \tau_0 \sigma_1[\rho \sigma_0/\alpha] \).

From \( \forall \alpha. \tau_0 = \delta \sigma_1 \), follows \( \delta \sigma_0 = \forall \alpha. \tau_0 \sigma_1 \). Together with the Substitution Lemma A.2 follows \( \Gamma \vdash e \sigma : \forall \alpha. \tau_0 \sigma_1 \).

Therefore we can apply TYPE APP to obtain \( \Gamma \hat{\sigma} \vdash (e. \rho) \hat{\sigma} : \tau \hat{\sigma} \).

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

12