Type-Inference Based Deforestation of Functional Programs

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften
genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Olaf Chitil

aus Süchteln, jetzt Viersen

Berichter: Universitätsprofessor Dr. rer. nat. Klaus Indermark
Universitätsprofessor Dr. rer. nat. Michael Hanus

Tag der mündlichen Prüfung: 27. Oktober 2000

Abstract

In lazy functional programming modularity is often achieved by using intermediate data structures to combine separate parts of a program. Each intermediate data structure is produced by one part and consumed by another one. Deforestation optimises a functional program by transformation into a program which does not produce such intermediate data structures. In this thesis we present a new method for deforestation, which combines a known method, short cut deforestation, with a new analysis that is based on type inference.

Short cut deforestation eliminates an intermediate list by a single, local transformation. In return, short cut deforestation expects both producer and consumer of the intermediate list in a certain form. Whereas the required form of the consumer is generally considered desirable in a well-structured program anyway, the required form of the producer is only a crutch to enable deforestation. Hence only the list-producing functions of the standard libraries were defined in the desired form and short cut deforestation has been confined to compositions of these functions.

Here, we present an algorithm which transforms an arbitrary producer into the required form. Starting from the observation that short cut deforestation is based on a parametricity theorem of the second-order typed λ-calculus, we show how the construction of the required form can be reduced to a type inference problem. Typability for the second-order typed λ-calculus is undecidable, but we only need to solve a partial type inference problem. For this problem we develop an algorithm based on the well-known Hindley-Milner type inference algorithm.

The transformation of a producer often requires inlining of some function definitions. Type inference even indicates which function definitions need to be inlined. However, only limited inlining across module boundaries is practically feasible. Therefore, we extend the previously developed algorithm to split a function definition into a worker definition and a wrapper definition. We only need to inline the small wrapper definition, which transfers all information required for deforestation.

The flexibility of type inference allows us to remove intermediate lists which original short cut deforestation cannot remove, even with hand-crafted producers. In contrast to most previous work on deforestation, we give a detailed proof of completeness and semantic correctness of our transformation.

Acknowledgements

My supervisor Prof. Klaus Indermark introduced me to the foundations of functional programming and taught me to precisely formulate my statements. He employed me at his chair, left me much freedom in selecting a subject for my thesis and showed trust in me all the time. Dr. Markus Mohnen introduced me to the area of optimisation through program transformation in general and deforestation in particular. The enthusiasm of Christian Tüffers for his Diplomarbeit on deforestation raised my interest in the area.

Many people in and outside Aachen discussed the topic of my thesis with me. Of these Frank Huch has to be mentioned first. We talked so often about my deforestation method and he had non-enumerably many ideas for improvement. Prof. Michael Hanus was always helpful and agreed to be second examiner although my thesis has no logical elements. Prof. Simon Peyton Jones pointed out several shortcomings in the first versions of my deforestation method and answered all my questions about the internals of the Glasgow Haskell compiler. Matthias Höff, Dr. Armin Kühnemann and I compared several deforestation techniques, thus improving my understanding of these techniques. This was possible because Armin and Prof. Heiko Vogler had invited me to Dresden. Prof. Patricia Johann and Prof. Franklyn Turbak read versions of the thesis and made numerous suggestions for improvement.

All the members of the Lehrstuhl für Informatik II and the Lehr- und Forschungsgebiet für Informatik II at the RWTH Aachen made working there such a pleasure, especially my office mates Dr. George Botorog and Thomas Richert. Finally, working all these years on this thesis would not have been possible without the support of my friends and my family.

I thank all of you.

Contents

Chapter 1

Introduction

A primary feature of a good programming language for productive software development is that it supports writing of highly modular programs. A small program module which is weakly coupled with other modules can be both coded and debugged quickly. Furthermore, it can be reused in several different contexts. Even a complex task should be performed by a small module which uses other small modules.

1.1 Intermediate Data Structures for Modularity

In functional programming functions are the basic modules from which programs are built. A powerful form of modularity is based on the use of intermediate data structures. Two separately defined functions can be glued together by an intermediate data structure that is produced by one function and consumed by the other. For example, the function `any`, which tests whether any element of a list `xs` satisfies a given predicate `p`, may be defined as follows in HASKELL [PH+99]:

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

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

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

We survey some more examples. A function `anyTree` shall test whether any element of a tree satisfies a given predicate. We can define this function by composing `any` with a general purpose function `treeToList` which simply produces a list of all elements stored in a tree.

```
anyTree :: (a -> Bool) -> Tree a -> Bool

anyTree p xs = any p (treeToList xs)
```

A list is also a suitable intermediate data structure for defining the factorial function:

```
factorial :: Int -> Int

factorial n = prod [1..n]
```

The expression `[1..n]` produces the list of all integers from 1 to `n` and the function `prod` computes the product of the list elements. On a larger scale, intermediate data structures may be used to glue together the phases of a compiler. For example, the lexical analysis phase produces a list of symbols which is consumed by the parser.

In his seminal paper *"Why Functional Programming Matters"* [Hug89] John Hughes points out that lazy functional languages make modularity through intermediate data structures practicable (see also [ASS96]). Functional programming languages can be classified as either eager or lazy, according to their order of evaluation. Like in standard imperative languages, in functional languages with eager evaluation, such as SCHEME [KCR98] and ML [MTHM97], the arguments to a function are evaluated before the function is called. In contrast, in lazy languages, such as HASKELL and CLEAN [Pv98], arguments to a function are evaluated in a demand-driven fashion. The arguments are initially passed in unevaluated form and are evaluated only when (and if) the computation needs the result to continue.

Considering our definition of `any` we note that with eager evaluation the whole intermediate boolean list is produced before it is consumed by the function `or`.

```
   any (> 2) [1,2,3,4,5,6,7,8,9,10]
⇝
   or (map (> 2) [1,2,3,4,5,6,7,8,9,10])
⇝
   or [False, False, True, True, True, True, True, True, True, True]
⇝
   True
```

In languages with eager evaluation intermediate data structures often require large space.

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

```
   any (> 2) [1,2,3,4,5,6,7,8,9,10]
⇝
   or (map (> 2) [1,2,3,4,5,6,7,8,9,10])
⇝
   or (False : (map (> 2) [2,3,4,5,6,7,8,9,10]))
⇝
   or (map (> 2) [2,3,4,5,6,7,8,9,10])
⇝
   or (False : (map (> 2) [3,4,5,6,7,8,9,10]))
⇝
   or (map (> 2) [3,4,5,6,7,8,9,10])
⇝
   or (True : (map (> 2) [4,5,6,7,8,9,10]))
⇝
   True
```

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

1.2 Deforestation

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

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

It is the aim of **deforestation** algorithms to transform automatically a modular functional program which uses intermediate data structures as glue into another one which does not produce these intermediate data structures. We say that the producer and the consumer of a data structure are **fused**.

Besides removing the costs of an intermediate data structure, deforestation also brings together subexpressions of producer and consumer which previously were separated by the intermediate data structure. Thus new opportunities for further optimising transformations arise. In the deforested example program, the compiler might inline the definition of `(||)` and with a few simple transformations obtain the slightly more efficient definition:

```
any p []     = False
any p (x:xs) = if (p x) then True else (any p xs)
```

This definition directly uses the primitive `if-then-else` construct instead of calling the function `(||)`.

Deforestation may also be applied to languages with eager evaluation. For these languages deforestation makes modularity through intermediate data structures considerably more attractive. However, in eager languages deforestation usually changes the space complexity and may even turn a non-terminating program into a terminating one. As an example of the latter phenomenon note that eager evaluation of the expression

```
any (\x -> 1/x > 0.5) [1,0]
```

successfully terminates for a deforested definition of `any`, but not for the modular definition. In the latter case a runtime error is caused by a division by zero. Hence it seems vital to explicitly mark all compositions of functions that have to be deforested. In the end this marking means that the program is no longer written in an eager language but a language with a complex mix of eager and lazy evaluation.

The deforestation method which we present in this thesis only depends on the syntax and type system of the programming language. So it can be applied to both lazy and eager functional languages. However, we prove its correctness and discuss its effect on efficiency only for a lazy language.

Deforestation is related to the optimisation of nested queries in data base query languages. Some functional deforestation techniques have been transferred successfully to this area [Feg93].

1.3 Short Cut Deforestation

Although there is an extensive literature on various deforestation methods, their implementation in real compilers proved to be difficult. Therefore, a simple deforestation method, called **cheap** or **short cut deforestation**, was developed [GLP93, GP94, Gil96].

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

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

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

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

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

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

$$\texttt{foldr}\ M^{(:)}\, M^{[]}\ ((\backslash \mathtt{v}^{(:)}\ \mathtt{v}^{[]}\ \texttt{->} M')\ (:)\ [])\quad \rightsquigarrow \quad (\backslash \mathtt{v}^{(:)}\ \mathtt{v}^{[]}\ \texttt{->} M')\ M^{(:)}\, M^{[]}$$

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

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

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

Usually, the producer skeleton P has the form $\backslash \mathtt{v}^{(:)}\ \mathtt{v}^{[]}\ \texttt{->} M'$, but this is not required for the semantic correctness of the transformation. Strikingly, the polymorphic type of P already guarantees the correctness. Intuitively, P can only construct its result of type c from its two arguments, because only these have matching types. Formally, the transformation is an instance of a parametricity or free theorem [Wad89, Pit00].

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

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

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

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

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

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

1.4 Type Inference Identifies List Constructors

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

$$\begin{array}{l}\mathtt{map\ ::\ (a\ ->\ b)\ ->\ [a]\ ->\ [b]}\\ \mathtt{map\ f\ xs\ =\ build\ (\backslash v}^{\mathtt{(:)}}\ \mathtt{v}^{\mathtt{[]}}\ \mathtt{->\ foldr\ (v}^{\mathtt{(:)}}\ \mathtt{.\ f)\ v}^{\mathtt{[]}}\ \mathtt{xs)}\end{array}$$

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

In this thesis we present a new method for automatically transforming an arbitrary list-producing expression into the form that is required by the short cut fusion rule. The starting-point for our method is the observation that the correctness of the short cut fusion rule solely depends on the polymorphic type of the producer skeleton. So we reduce the problem of deriving the required form of the producer to a type inference problem.

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

Obviously, this method is prohibitively expensive. Fortunately, we can determine the list constructors that need to be replaced in one pass, if we use an algorithm which infers a most general, a principal typing. At the heart of our **list abstraction algorithm**, which transforms a producer into the required form, is a type inference algorithm, which is based on the well-known Hindley-Milner type inference algorithm.

The example producer `(map p [1,2])` already demonstrated that the constructors of a result list need not occur in the producer itself: The producer may call a function, such as `map`, which constructs the result. So to abstract the result list constructors from a producer, inlining of some function definitions is unavoidable. Nicely, type inference can even indicate which function definitions need to be inlined. Nonetheless inlining is a problem in practice, because extensive inlining across module boundaries would defeat the idea of separate compilation. Hence we develop a **worker/wrapper scheme**. We split every definition of a list-producing function into a definition of a worker and a definition of a wrapper. The definition of the wrapper is small and non-recursive. Because it contains all the constructors of the result list, only the wrapper definition needs to be inlined to enable deforestation. This split of a producer definition into a worker and a wrapper definition is performed by an algorithm which is based on our type-inference based list abstraction algorithm.

Altogether we obtain a new, type-inference based short cut deforestation algorithm which is more flexible than the original one. It can fuse `foldr` consumers with nearly arbitrary producers, especially user-defined producers. It can even remove some intermediate lists which the original short cut deforestation algorithm cannot remove, even with hand-crafted producers in `build` form.

We formally present our improved short cut deforestation for a small functional language with second-order types which we name F. On the one hand F is small enough for complete formal descriptions and proofs. On the other hand, most programs of a modern functional programming language such as Haskell can be translated into F. In fact, F is only a slightly simplified version of the intermediate language used inside the Glasgow Haskell compiler [PS98, GHC]. The Glasgow Haskell compiler is a highly optimising compiler for Haskell which itself is written in Haskell. It was designed for being easily extensible by further optimisation phases [Par94, Chi97] . Hence using F will also simplify a future implementation of our deforestation method. For a prototype implementation of our central list abstraction algorithm we reused data structures and some basic functions from the Glasgow Haskell compiler.

We already presented the idea of using type-inference for deforestation in [Chi99] and the worker/wrapper scheme in [Chi00].

1.5 Structure of the Thesis

In Chapter 2 we define the language F. We define its syntax, type system and semantics. We also define the notion of a correctness-preserving transformation rule and study the transformation rules on which our deforestation transformation is based. Finally, we present a cost model which serves as foundation for arguments about how a transformation modifies the performance of a program. In Chapter 3 we explain informally by examples how our new deforestation algorithm works, especially the type-inference based list abstraction method and the worker/wrapper scheme. Chapter 4 contains the central contributions of the thesis. We formally describe our list abstraction algorithm including the new type inference algorithm and prove its completeness and semantic correctness. In Chapter 5 we describe the complete deforestation algorithm. We both analyse and practically measure its effect on programs. Subsequently we discuss in Chapter 6 several ideas for further improving and extending our deforestation algorithm. In Chapter 7 we present the major methods which have been proposed for deforestation and relate them to our method. We conclude in Chapter 8 with some general thoughts about deforestation and the prospects of type inference as analysis for program transformations.

Chapter 2

The Functional Language F

For the definition of our deforestation method we introduce a small functional language which we name F. The language is essentially the second-order typed λ-calculus [Gir72, Rey74, Bar92], also named system F, augmented with recursion and constructor-based data types. We need second-order types to express the short cut fusion rule, recursion to make the language Turing-complete and realistic and constructor-based data structures, because we want to remove intermediate lists. Many HASKELL programs can be translated directly into F programs. F is a slightly simplified version of the intermediate language on which all optimising transformations in the Glasgow Haskell compiler are performed [PS98, GHC]. To keep the language simple we do not introduce any primitive types with primitive functions. In principle, finite data types such as characters (`Char`) and bounded numbers (`Int`) can be implemented as algebraic data types with many data constructors. Furthermore, F does not support higher-order types beyond second-order and HASKELL-like type synonyms.

2.1 Syntax

F is an extension of the Church-style second-order typed λ-calculus, that is, terms contain some type information [Bar92]. Thus the type of a term is unique and can easily be determined. The abstract syntax of **types** is defined in Figure 2.1 and the syntax of **terms** is defined in Figure 2.2. The definition of types is based on an infinite set of **type variables** and a (usually finite) set of **type constructors** (e.g., `Bool`, `Int`, `Tree`). Similarly, the definition of terms is based on an infinite set of **term variables** and a (usually finite) set of **data constructors** (e.g., `True`, `42`, `Leaf`).

Note that we often draw a line over a meta-variable to abbreviate a sequence of meta-variables. For example, we write $\overline{\tau}$ for $\tau_1 \ldots \tau_k$ and even $\{\overline{\tau}\}$ for $\{\tau_1, \ldots, \tau_k\}$. This informal but helpful abbreviation leaves implicit the index of the upper bound of the sequence.

Informally, the intended meaning of terms and types is as follows. $\lambda x : \tau.M$ is a function that maps the variable x of type τ to M, which usually uses x. Its type is a function type $\tau \to \rho$. $M\,N$ is the function M applied to the argument N. $\texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N$ defines variables x_i by terms M_i. These x_i can be used in N and also in the M_i; that is, mutually recursive definitions are possible. The elements of all data types are

$$
\begin{array}{rrcll}
\text{Type variables} & \alpha, \beta, \gamma, \delta & & & \\
\text{Type constructors} & T & & & \\
\text{Types} & \tau, \rho, \nu & ::= & T\,\overline{\tau} & \text{data type} \\
 & & \mid & \alpha & \text{variable} \\
 & & \mid & \tau_1 \rightarrow \tau_2 & \text{function type} \\
 & & \mid & \forall \alpha . \tau & \text{universal type}
\end{array}
$$

Figure 2.1: Types of F

$$
\begin{array}{rrcll}
\text{Term variables} & x, y, z & & & \\
\text{Data constructors} & c & & & \\
\text{Terms} & M, N, O, P & ::= & x & \text{variable} \\
 & & \mid & \lambda x : \tau . M & \text{term abstraction} \\
 & & \mid & M\,N & \text{term application} \\
 & & \mid & \texttt{let } \{x_i : \tau_i = M_i\}_{i=1}^{k} \texttt{ in } N & \text{mutual recursion} \\
 & & \mid & c & \text{data constructor} \\
 & & \mid & \texttt{case } M \texttt{ of } \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k} & \text{data destruction} \\
 & & \mid & \lambda \alpha . M & \text{type abstraction} \\
 & & \mid & M\,\tau & \text{type application}
\end{array}
$$

Figure 2.2: Terms of F

constructed by data constructors c. Because data types can be polymorphic, a data type $T\,\overline{\tau}$ consists of a type constructor T applied to some types $\overline{\tau}$. $\texttt{case}\ M\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k}$ tests for the top constructor of the value of M. It yields the value of the N_i corresponding to the constructor found. Furthermore, N_i may use the arguments $\overline{x}_i$ of the constructor. Church-style polymorphism means that polymorphism is not only visible in the types but also in the terms of the language. The polymorphic term $\lambda\alpha.M$ maps the type variable α to M. Its type is a universal type $\forall\alpha.\tau$. $M\,\tau$ is the instantiation of the polymorphic term M to the type τ.

In addition to the context free definition of the syntax we demand that the variables defined in a `let` construct are distinct, the variables within a pattern $c\,\overline{x}$ of a `case` construct are distinct and the constructors in a `case` construct are distinct.

By **expression** we refer to both types and terms. The meta-variable e is used to denote an expression. Similarly, we use the meta-variable a for a **variable** that may be either a type or a term variable.

In examples we use round brackets (and) to resolve ambiguities. On the other hand we often use Haskell-like indentation instead of curly brackets for writing the sets in `let` and `case` constructs. Here is an example for a term:

```
let not : Bool→Bool
        = λb:Bool. case b of
             True ↦ False
             False↦ True
    id : ∀α.α→α
       = λα. λx:α. x
```

```
in id Bool (not True)
```

To reduce the number of round brackets we use the following conventions:

$M\,e_1\,e_2$	abbreviates	$(M\,e_1)\,e_2$
$\lambda x:\tau.\,M\,e$	abbreviates	$\lambda x:\tau.\,(M\,e)$
$\lambda\alpha.\,M\,e$	abbreviates	$\lambda\alpha.\,(M\,e)$
`let` $\{x_i:\tau_i = M_i\}_{i=1}^{k}$ `in` $N\,e$	abbreviates	`let` $\{x_i:\tau_i = M_i\}_{i=1}^{k}$ `in` $(N\,e)$

As the preceding example demonstrates we use the same symbols α, β, γ and δ as type variables as we do for meta-variables ranging over type variables. For type constructors, term variables and data constructors we use the same syntax as Haskell, that is, with a few exceptions type constructors are words starting with a capital letter (e.g., `Bool`), term variables are words starting with a lower-case letter (e.g., `not`, `b`, `id`) and data constructors are words starting with a capital letter (e.g., `True`, `False`).

Central to this thesis is obviously the list type. We chose to use the same notation for the list type constructor and its data constructors as in Haskell, although this unfortunately leads to an ambiguity. The expression `[]` τ can be interpreted both as the type of lists with elements of type τ and as the application of the empty list constructor to the type τ. We usually write `[`τ`]` for the former but never use this abbreviation for the latter. Sometimes the context is needed to resolve the ambiguity. Besides the data constructor of the empty list `[]` there is also the data constructor `(:)` that puts a list element in front of a list. In `case` patterns we use the list constructor `(:)` as infix operator, that is, we write `y:ys` instead of `(:) y ys`. We can do this, because in `case` patterns even polymorphic constructors do not have type arguments. For example:

```
let head : ∀α.[α]→ α
         = λα.λxs:[α].case xs of { y:ys ↦ y }
in head Int ((:) Int 1 ([] Int))
```

A program in a functional language typically consists of some definitions together with a term to be evaluated modulo the given definitions. In F a program is just a term which usually uses the `let` construct for definitions. Nonetheless, we will sometimes consider a definition separately. For example, a definition of the function that maps a function on all elements of a list is:

```
map : ∀α.∀β.(α→β)→[α]→[β]
    = λα.λβ.λf:α → β.λxs:[α].case xs of
        []    ↦ [] β
        y:ys ↦ (:) β (f y) (map α β f ys)
```

As in Haskell we use `(,)` both as tuple type constructor and as tuple data constructor. However, we usually write `(`τ_1`,`τ_2`)` for the tuple type `(,)` $\tau_1\,\tau_2$ but never do so for the tuple data constructor:

```
fst  : ∀α.∀β.(α,β)→ α
    = λα.λβ.λp:(α,β).case p of { (x,y) ↦ x }

swap : ∀α.∀β.(α,β)→(β,α)
     = λα.λβ.λp:(α,β).case p of { (x,y) ↦ (,) β α y x }
```

Note that we did not define any names or symbols for the sets of type variables, type constructors, types, term variables, term constructors and terms. Instead, we fixed a set of meta-variables for elements of every set, for example M, N and O for terms. We use such conventions throughout the thesis, so that without futher comment it is always clear over which set any meta-variable ranges. To have an unbounded number of meta-variables we use indices, for example M_1 and N_3.

2.2 Substitutions and Renaming of Bound Variables

Type variables are placeholders for types and term variables are placeholders for terms. Many operations on types and terms are based on substituting some types or terms for some variables.

Our language F has binding constructs for both type variables and term variables. In types, universal quantification is a variable binding construct. In terms, term and type abstraction, definition in a `let` construct and destruction in a `case` construct are variable binding constructs. Hence we have to distinguish between bound and free variables of an expression.

The set of **bound variables**, bound(e), and the set of **free variables**, free(e), of an expression e are defined in Figure 2.3.

A type is **closed** if it has no free (type) variables and a term is **closed** if it has no free (term and type) variables. Otherwise an expression is called **open**. A closed term is an **F-program**.

2.2.1 Substitutions

A **substitution**, denoted by σ, is a mapping from variables to expressions such that a type variable is mapped to a type and a term variable is mapped to a term. Furthermore, we require that the **domain** of a substitution σ,

$$\mathrm{dom}(\sigma) := \{a \mid \sigma(a) \neq a\}$$

is finite. Hence we can write $[e_1/a_1, \ldots, e_k/a_k]$ or just $[\overline{e}/\overline{a}]$ for the substitution σ with $\mathrm{dom}(\sigma) \subseteq \{a_1, \ldots, a_k\}$ and $\sigma(a_1) = e_1, \ldots, \sigma(a_k) = e_k$.

We define the **bound and free variables of a substitution** and the set of **variables which are modified**, that is influenced, by a substitution:

$$\begin{aligned}
\mathrm{bound}(\sigma) &:= \bigcup\{\mathrm{bound}(\sigma(\gamma)) \mid \gamma \in \mathrm{dom}(\sigma)\} \\
\mathrm{free}(\sigma) &:= \bigcup\{\mathrm{free}(\sigma(\gamma)) \mid \gamma \in \mathrm{dom}(\sigma)\} \\
\mathrm{mod}(\sigma) &:= \mathrm{dom}(\sigma) \cup \mathrm{free}(\sigma)
\end{aligned}$$

It is well known that the definition of applying a substitution to an expression where the expression contains variable binders needs care to avoid confusion between free and bound variables [Bar81, FH88]. Bound variables shall not be replaced by a substitution and free variables that are introduced by a substitution shall not be bound by a binding construct of the expression. To ensure these properties we rename bound variables appropriately. Figure 2.4 shows our adaption of Curry's definition of **substitution application** [CF58]. Here $e\sigma_1\sigma_2$ abbreviates $(e\sigma_1)\sigma_2$.

In the definition for the first interesting case, $(\forall\alpha.\tau)\sigma$, we note the rather vague requirement that β shall be a "new" type variable. Such a "new" type variable may in fact

$$
\begin{aligned}
\mathrm{bound}(\alpha) &:= \emptyset\\
\mathrm{bound}(\forall\alpha.\tau) &:= \{\alpha\} \cup \mathrm{bound}(\tau)\\
\mathrm{bound}(\tau_1 \to \tau_2) &:= \mathrm{bound}(\tau_1) \cup \mathrm{bound}(\tau_2)\\
\mathrm{bound}(T\,\tau_1..\tau_k) &:= \mathrm{bound}(\tau_1) \cup \ldots \cup \mathrm{bound}(\tau_k)\\
\mathrm{bound}(x) &:= \emptyset\\
\mathrm{bound}(\lambda x : \tau.M) &:= \{x\} \cup \mathrm{bound}(\tau) \cup \mathrm{bound}(M)\\
\mathrm{bound}(M\,N) &:= \mathrm{bound}(M) \cup \mathrm{bound}(N)\\
\mathrm{bound}(\mathtt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \mathtt{in}\ N) &:= \bigcup_{i=1}^{k}(\mathrm{bound}(\tau_i) \cup \mathrm{bound}(M_i) \cup \{x_i\}) \cup \mathrm{bound}(N)\\
\mathrm{bound}(c) &:= \emptyset\\
\mathrm{bound}(\mathtt{case}\ M\ \mathtt{of}\ \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k}) &:= \mathrm{bound}(M) \cup \bigcup_{i=1}^{k}(\mathrm{bound}(N_i) \cup \{\overline{x}_i\})\\
\mathrm{bound}(\lambda\alpha.M) &:= \{\alpha\} \cup \mathrm{bound}(M)\\
\mathrm{bound}(M\,\tau) &:= \mathrm{bound}(M) \cup \mathrm{bound}(\tau)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{free}(\alpha) &:= \{\alpha\}\\
\mathrm{free}(\forall\alpha.\tau) &:= \mathrm{free}(\tau) \setminus \{\alpha\}\\
\mathrm{free}(\tau_1 \to \tau_2) &:= \mathrm{free}(\tau_1) \cup \mathrm{free}(\tau_2)\\
\mathrm{free}(T\,\tau_1..\tau_k) &:= \mathrm{free}(\tau_1) \cup \ldots \cup \mathrm{free}(\tau_k)\\
\mathrm{free}(x) &:= \{x\}\\
\mathrm{free}(\lambda x : \tau.M) &:= (\mathrm{free}(\tau) \cup \mathrm{free}(M)) \setminus \{x\}\\
\mathrm{free}(M\,N) &:= \mathrm{free}(M) \cup \mathrm{free}(N)\\
\mathrm{free}(\mathtt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \mathtt{in}\ N) &:= (\bigcup_{i=1}^{k}(\mathrm{free}(\tau_i) \cup \mathrm{free}(M_i)) \cup \mathrm{free}(N)) \setminus \{x_i\}_{i=1}^{k}\\
\mathrm{free}(c) &:= \emptyset\\
\mathrm{free}(\mathtt{case}\ M\ \mathtt{of}\ \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k}) &:= \mathrm{free}(M) \cup \bigcup_{i=1}^{k}(\mathrm{free}(N_i) \setminus \{\overline{x}_i\})\\
\mathrm{free}(\lambda\alpha.M) &:= \mathrm{free}(M) \setminus \{\alpha\}\\
\mathrm{free}(M\,\tau) &:= \mathrm{free}(M) \cup \mathrm{free}(\tau)
\end{aligned}
$$

Figure 2.3: Bound and free variables of an expression

be a type variable that already appears in some expression. By “new” we mean that we do not care which type variable is chosen, as long as it fulfills the given condition. Our algorithms and proofs only rely on this partial definition. Such use of “new” variables is common in the definition of type inference algorithms (see e.g. [DM82, LY98]). To have a complete definition we could fix a “new” variable for example by demanding that the

$$
\begin{aligned}
\alpha\sigma &:= \sigma(\alpha) \\
(\forall\alpha.\tau)\sigma &:= \forall\beta.(\tau[\beta/\alpha]\sigma) \\
&\quad \text{where } \beta \text{ new with } \beta \notin \mathrm{free}(\forall\alpha.\tau) \cup \mathrm{mod}(\sigma) \\[1em]
(T\,\tau_1..\tau_k)\sigma &:= T\,(\tau_1\sigma)..(\tau_k\sigma) \\[1em]
(\tau_1 \to \tau_2)\sigma &:= (\tau_1\sigma) \to (\tau_2\sigma) \\[1em]
x\sigma &:= \sigma(x) \\[1em]
(\lambda x:\tau.M)\sigma &:= \lambda y:(\tau\sigma).(M[y/x]\sigma) \\
&\quad \text{where } y \text{ new with } y \notin \mathrm{free}(\lambda x:\tau.M) \cup \mathrm{mod}(\sigma) \\[1em]
(M\,N)\sigma &:= (M\sigma)\,(N\sigma) \\[1em]
(\texttt{let }\{x_i:\tau_i = M_i\}_{i=1}^{k}\texttt{ in }N)\sigma &:= \texttt{let }\{y_i:\tau_i\sigma = M_i[\overline{y}/\overline{x}]\sigma\}_{i=1}^{k}\texttt{ in }(N[\overline{y}/\overline{x}]\sigma) \\
&\quad \text{where for all } i \in \{1..k\},\ y_i \text{ new and distinct with} \\
&\quad y_i \notin \mathrm{free}(\texttt{let }\{x_i:\tau_i = M_i\}_{i=1}^{k}\texttt{ in }N) \cup \mathrm{mod}(\sigma) \\[1em]
c\sigma &:= c \\[1em]
(\texttt{case }M\texttt{ of }\{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k})\sigma &:= \texttt{case }M\sigma\texttt{ of }\{c_i\,\overline{y}_i \mapsto N_i[\overline{y}_i/\overline{x}_i]\sigma\}_{i=1}^{k} \\
&\quad \text{where for all } i \in \{1..k\},\ \overline{y}_i \text{ are new and distinct with} \\
&\quad \overline{y}_i \cap (\mathrm{free}(N_i) \setminus \{\overline{x}_i\} \cup \mathrm{mod}(\sigma)) = \emptyset \\[1em]
(\lambda\alpha.M)\sigma &:= \lambda\beta.(M[\beta/\alpha]\sigma) \\
&\quad \text{where } \beta \text{ new with } \beta \notin \mathrm{free}(\lambda\alpha.M) \cup \mathrm{mod}(\sigma) \\[1em]
(M\,\tau)\sigma &:= (M\sigma)\,(\tau\sigma)
\end{aligned}
$$

Figure 2.4: Application of a substitution to a term

set of all variables is totally ordered and the new variable is the least variable for which the given condition holds. Alternatively, all functions that need “new” variables could pass around an infinite set of variables that have not yet been used. An implementation in an imperative programming language would probably use a global variable supply instead of passing around a set. We do not want to commit ourselves to any of the many different options for implementing the creation of “new” variables. Furthermore, most implementations would obfuscate the definition or algorithm of interest.

Finally, we define composition of two substitutions σ_1 and σ_2:

$$\sigma_1\sigma_2(a) := (\sigma_1(a))\sigma_2$$

So $a(\sigma_1\sigma_2) = (a\sigma_1)\sigma_2$.

$$\frac{e_1\ R\ e_2}{e_2\ R\ e_1}\ \text{SYMMETRY} \qquad \frac{e_1\ R\ e_2 \quad e_2\ R\ e_3}{e_1\ R\ e_3}\ \text{TRANSITIVITY}$$

$$\frac{}{\alpha\ R\ \alpha} \qquad \frac{\tau\ R\ \rho}{\forall\alpha.\tau\ R\ \forall\alpha.\rho} \qquad \frac{\tau_1\ R\ \rho_1 \quad \ldots \quad \tau_k\ R\ \rho_k}{T\,\tau_1 \ldots \tau_k\ R\ T\,\rho_1 \ldots \rho_k} \qquad \frac{\tau_1\ R\ \rho_1 \quad \tau_2\ R\ \rho_2}{\tau_1 \to \tau_2\ R\ \rho_1 \to \rho_2}$$

$$\frac{}{x\ R\ x} \qquad \frac{\tau\ R\ \rho \quad M\ R\ N}{\lambda x : \tau.M\ R\ \lambda x : \rho.N} \qquad \frac{M_1\ R\ N_1 \quad M_2\ R\ N_2}{M_1\,M_2\ R\ N_1\,N_2}$$

$$\frac{\forall i \in \{1..k\} \quad \tau_i\ R\ \rho_i \quad M_i\ R\ N_i \quad M\ R\ N}{\texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ M\ R\ \texttt{let}\ \{x_i : \rho_i = N_i\}_{i=1}^{k}\ \texttt{in}\ N}$$

$$\frac{}{c\ R\ c} \qquad \frac{M\ R\ N \quad \forall i \in \{1..k\} \quad M_i\ R\ N_i}{\texttt{case}\ M\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto M_i\}_{i=1}^{k}\ R\ \texttt{case}\ N\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k}}$$

$$\frac{M\ R\ N}{\lambda\alpha.M\ R\ \lambda\alpha.N} \qquad \frac{M\ R\ N \quad \tau\ R\ \rho}{M\,\tau\ R\ N\,\rho}$$

Figure 2.5: Defining properties of a congruence R on expressions

2.2.2 Renaming of Bound Variables

In our definition of substitution application we were able to rename bound variables, because the choice of name of a bound variable will prove to be irrelevant in all our uses of terms and types. In the literature it is common to identify expressions that differ only in bound variables already at the syntactical level [Bar81, Pit97], that is, a term actually represents an equivalence class of terms. However, we do not take this approach. First, this approach is very hard to make rigorous. Whereas the set of free variables of an expression is invariant under renaming, the set of bound variables is not. In inductive proofs expressions are taken apart and thus bound variables are turned into free variables and vice versa. Second, the algorithms, especially the type inference algorithm, that we present in this thesis have to work on concrete expressions. Meta-languages for comfortable inductive definitions on syntax with binding constructs are still subject of research [GP99, PG00].

In the semantics of terms, which we define in Section 2.4 and Section 2.6, terms that differ only in bound variables are equal. However, we do not define any semantics of types that would give a congruence on types with an analogous property. Nonetheless we want that for example any term of type $\forall\alpha.\texttt{[}\alpha\texttt{]} \to \alpha$ also has type $\forall\beta.\texttt{[}\beta\texttt{]} \to \beta$. Hence, for the type system, which we define in the next section, we have to make renaming of bound type variables explicit.

A relation R on expressions is a **congruence**, if all properties expressed as rules[1] in Figure 2.5 hold. We define a congruence $\equiv$, called **α-congruence**, on expressions which relates expressions that differ only in the names of bound type variables. Let $\equiv$ be the

[1] In this thesis we often use rules to state properties and to define relations. Appendix A explains their meaning.

least congruence on expressions which fulfills the following two rules:

$$\frac{\beta \notin \text{free}(\tau)}{\forall\alpha.\tau \equiv \forall\beta.\tau[\beta/\alpha]} \qquad \frac{\beta \notin \text{free}(M)}{\lambda\alpha.M \equiv \lambda\beta.M[\beta/\alpha]}$$

We note that the definition of $\equiv$ and of substitution application fit well together, because substitution on expressions determines well-defined substitution on $\equiv$-equivalence classes:

Lemma 2.1 (Substitution application and $\equiv$)

$$\frac{e \equiv e' \quad \tau \equiv \tau'}{e[\tau/\alpha] \equiv e'[\tau'/\alpha]} \qquad \frac{e_1 \equiv e_1' \quad e_2 \equiv e_2'}{e_1[e_2/x] \equiv e_1'[e_2'/x]} \qquad \square$$

Proof See [CF58], Section 3.E, Theorem 2(a). ■

When two universal types are equal up to renaming of bound type variables their bodies need not to be equal, but a similar property holds:

Lemma 2.2 (Equality up to renaming of universal types)

$$\frac{\forall\alpha.\tau \equiv \forall\beta.\rho \quad \gamma \notin \text{free}(\forall\alpha.\tau) \cup \text{free}(\forall\beta.\rho)}{\tau[\gamma/\alpha] \equiv \rho[\gamma/\beta]} \qquad \square$$

Proof By induction on the rules that define $\equiv$, using Lemma 2.1. ■

If the variables modified by a substitution and the bound variables of an expression are disjoint, then we can push the substitution freely into the expression.

Lemma 2.3 (Pushing a substitution over a variable binding)

$$\frac{\alpha \notin \text{mod}(\sigma)}{(\forall\alpha.\tau)\sigma \equiv \forall\alpha.(\tau\sigma)} \qquad \frac{\alpha \notin \text{mod}(\sigma)}{(\lambda\alpha.M)\sigma \equiv \lambda\alpha.(M\sigma)} \qquad \square$$

Proof Follows directly from the definitions of substitution application and of $\equiv$. ■

Finally, we can exchange substitutions under a certain condition.

Lemma 2.4 (Exchange of substitutions)

$$\frac{\alpha \notin \text{mod}(\sigma)}{e[\rho/\alpha]\sigma \equiv e\sigma[\rho\sigma/\alpha]} \qquad \square$$

Proof See [CF58], Section 3.E, Theorem 2(c). ■

2.3 Type System

F is a typed language, in the sense that we only consider a term to be well formed if it can be assigned a type, given an assignment of types to the free term variables occurring in the term. A **typing** is a judgement of the form

$$\Gamma \vdash M : \tau$$

meaning that M has type τ in the typing environment Γ. A typing environment Γ is a finite partial mapping from term variables to types. With dom(Γ) we denote the domain

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{var}$$

$$\frac{\Gamma + \{x : \tau_1\} \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . M : \tau_1 \to \tau_2}\ \textsc{term abs}$$

$$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\,N : \tau_2}\ \textsc{term app}$$

$$\frac{\forall i \in \{1..k\} \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash M_i : \tau_i \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash N : \tau}{\Gamma \vdash \texttt{let } \{x_i : \tau_i = M_i\}_{i=1}^{k} \texttt{ in } N : \tau}\ \textsc{let}$$

$$\frac{\Delta(c) = \tau}{\Gamma \vdash c : \tau}\ \textsc{cons}$$

$$\frac{\Gamma \vdash M : T\,\overline{\rho} \quad \forall i \in \{1..k\} \quad \vdash c_i : \forall\overline{\alpha}.\overline{\rho}_i \to T\,\overline{\alpha} \quad \Gamma + \{\overline{x}_i : \overline{\rho}_i[\overline{\rho}/\overline{\alpha}]\} \vdash N_i : \tau}{\Gamma \vdash \texttt{case } M \texttt{ of } \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k} : \tau}\ \textsc{case}$$

$$\frac{\Gamma \vdash M : \tau \quad \alpha \notin \mathrm{freeTy}(\Gamma)}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.\tau}\ \textsc{type abs}$$

$$\frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash M\,\rho : \tau[\rho/\alpha]}\ \textsc{type app}$$

$$\frac{\Gamma \vdash M : \tau \quad M \equiv M'}{\Gamma \vdash M' : \tau}\ \textsc{term rename} \qquad \frac{\Gamma \vdash M : \tau \quad \tau \equiv \tau'}{\Gamma \vdash M : \tau'}\ \textsc{type rename}$$

Figure 2.6: Type system

of definition of the mapping Γ. A typing environment is needed to determine the types of the free variables of M. We write it as a set of tuples $x : \tau$. The operator $+$ combines two typing environments by giving precedence to the second argument:

$$(\Gamma_1 + \Gamma_2)(x) := \begin{cases} \Gamma_2(x) & \text{, if } x \in \mathrm{dom}(\Gamma_2) \\ \Gamma_1(x) & \text{, otherwise} \end{cases}$$

The sets of bound and free type variables of a typing environment and application of a type substitution to a typing environment are given by

$$\begin{aligned} \mathrm{bound}(\Gamma) &:= \bigcup\{\mathrm{bound}(\Gamma(x)) \mid x \in \mathrm{dom}(\Gamma)\} \\ \mathrm{free}(\Gamma) &:= \bigcup\{\mathrm{free}(\Gamma(x)) \mid x \in \mathrm{dom}(\Gamma)\} \\ \Gamma\sigma &:= \{x : (\tau\sigma) \mid \Gamma(x) = \tau\} \end{aligned}$$

Sometimes we are interested only in free type variables or only in free term variables. We define

$$\begin{aligned} \mathrm{freeTy}(X) &:= \{a \mid a \in \mathrm{free}(X) \text{ and } a \text{ is a type variable}\} \\ \mathrm{freeTerm}(X) &:= \{a \mid a \in \mathrm{free}(X) \text{ and } a \text{ is a term variable}\} \end{aligned}$$

for all terms, substitutions and typing environments X.

The syntax of F does not contain definitions of data types such as

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

as HASKELL does. Instead, we assume the existence of a global **data typing environment** Δ that assigns closed types of the form $\forall\overline{\alpha}.\overline{\rho} \to T\,\overline{\alpha}$ to data constructors. Note that $\forall\overline{\alpha}.\overline{\rho} \to T\,\overline{\alpha}$ is an abbreviation for $\forall\alpha_1.\ldots\forall\alpha_l.\rho_1 \to \ldots \to \rho_k \to T\,\alpha_1..\alpha_l$, that is, $\forall\alpha_1.\ldots\forall\alpha_l.T\,\alpha_1..\alpha_l$ in the case $k = 0$. The data typing environment implicitly defines all data types. For example,

$$\{\texttt{Leaf}:\forall\alpha.\texttt{Tree}\ \alpha,\ \texttt{Node}:\forall\alpha.\alpha \to\texttt{Tree}\ \alpha \to\texttt{Tree}\ \alpha \to\texttt{Tree}\ \alpha\}$$

defines a binary tree data type that corresponds to the given HASKELL data type. It is not necessary but more convenient to have Δ separate from Γ. In the examples in this thesis we assume that the data typing environment is defined as follows:

$$\begin{aligned}\Delta \ := \ &\{\texttt{False}:\texttt{Bool},\ \texttt{True}:\texttt{Bool},\\ &\texttt{0}:\texttt{Int},\ \texttt{1}:\texttt{Int},\ldots,\\ &\texttt{(:)}:\forall\alpha.\alpha \to\texttt{[}\alpha\texttt{]}\to\texttt{[}\alpha\texttt{]},\ \texttt{[]}:\forall\alpha.\texttt{[}\alpha\texttt{]},\\ &\texttt{(,)}:\forall\alpha.\forall\beta.\alpha \to \beta \to\texttt{(}\alpha\texttt{,}\beta\texttt{)},\\ &\texttt{Leaf}:\forall\alpha.\texttt{Tree}\ \alpha,\ \texttt{Node}:\forall\alpha.\alpha \to\texttt{Tree}\ \alpha \to\texttt{Tree}\ \alpha \to\texttt{Tree}\ \alpha\}\end{aligned}$$

The **type system of** F, that is the set of valid typings, is defined by the typing rules given in Figure 2.6.

Note that the asymmetric definition of $+$ ensures correct term variable scoping in the TERM ABS, LET and CASE rule, so that for example $\emptyset \vdash \lambda\texttt{x:Bool}.\lambda\texttt{x:Int.x} : \texttt{Bool}\to\texttt{Int}\to\texttt{Int}$ is valid. The condition $\alpha \notin \text{freeTy}(\Gamma)$ in the TYPE ABS rule is necessary because of the dependency of terms on types. Without this condition we could derive that the term $(\lambda\alpha.\lambda\texttt{x:}\alpha.\lambda\alpha.\texttt{x})\ \texttt{Int 1 Bool}$ has type $\texttt{Bool}$. This typing is unsound, because the term evaluates to 1 as we will see in the next section. The condition assures that only the typing $\emptyset \vdash (\lambda\alpha.\lambda\texttt{x:}\alpha.\lambda\alpha.\texttt{x})\ \texttt{Int 1 Bool} : \texttt{Int}$ can be derived.

We need the TYPE RENAME rule for example to be able to apply a term of type $(\forall\alpha.\texttt{[}\alpha\texttt{]}\to\texttt{Int})\to\texttt{Bool}$ to a term of type $\forall\beta.\texttt{[}\beta\texttt{]}\to\texttt{Int}$. We need the TERM RENAME rule for example to derive a type for the term $\lambda\alpha.\texttt{let x:[}\alpha\texttt{] = []}\ \alpha\ \texttt{in}\ \lambda\alpha.\texttt{x}$ as the following derivation demonstrates:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Delta(\texttt{[]}) = \forall\alpha.\texttt{[}\alpha\texttt{]}}{\{x:\texttt{[}\alpha\texttt{]}\} \vdash \texttt{[]} : \forall\alpha.\texttt{[}\alpha\texttt{]}}\,\textsc{cons}}{\{x:\texttt{[}\alpha\texttt{]}\} \vdash \texttt{[]}\ \alpha : \texttt{[}\alpha\texttt{]}}\,\textsc{type app} \qquad \dfrac{\dfrac{\dfrac{\{x:\texttt{[}\alpha\texttt{]}\}(\texttt{x}) = \texttt{[}\alpha\texttt{]}}{\{x:\texttt{[}\alpha\texttt{]}\} \vdash \texttt{x} : \texttt{[}\alpha\texttt{]}}\,\textsc{var}}{\{x:\texttt{[}\alpha\texttt{]}\} \vdash \lambda\beta.\texttt{x} : \forall\beta.\texttt{[}\alpha\texttt{]}}\,\textsc{type abs}}{\{x:\texttt{[}\alpha\texttt{]}\} \vdash \lambda\alpha.\texttt{x} : \forall\beta.\texttt{[}\alpha\texttt{]}}\,\textsc{term rename}}{\emptyset \vdash \texttt{let x:[}\alpha\texttt{] = []}\ \alpha\ \texttt{in}\ \lambda\alpha.\texttt{x} : \forall\beta.\texttt{[}\alpha\texttt{]}}\,\textsc{let}}{\emptyset \vdash \lambda\alpha.\texttt{let x:[}\alpha\texttt{] = []}\ \alpha\ \texttt{in}\ \lambda\alpha.\texttt{x} : \forall\alpha.\forall\beta.\texttt{[}\alpha\texttt{]}}\,\textsc{type abs}$$

The TERM RENAME and the TYPE RENAME rule raise the question why there is no similar rule for renaming the bound type variables in the typing environment. The reason is, that such a rule is already derivable from the given type system. Defining renaming on typing environments in the obvious way by stating that $\Gamma \equiv \Gamma'$ if $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ and for all $x \in \text{dom}(\Gamma)$, $\Gamma(x) \equiv \Gamma'(x)$ holds, we obtain:

Lemma 2.5 (Environment renaming)

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \equiv \Gamma'}{\Gamma' \vdash M : \tau} \qquad \square$$

PROOF By induction on the typing rules. Whenever the typing environment is accessed by the VAR rule, the bound variables of the type can be renamed by a subsequent application of the TYPE RENAME rule. ■

We write $\vdash M : \tau$ for $\emptyset \vdash M : \tau$, if both M and τ are closed. Note that even if τ is closed, then M may nonetheless contain free type variables. We say that M **has a type** or that M is **well-typed**, if there exist Γ and τ such that $\Gamma \vdash M : \tau$ holds.

With the rules we can show that the terms and definitions given in Section 2.1 are well-typed, for example:

```
⊢ let head:∀α.[α]→ α = ... in head Int ((:) Int 1 ([] Int)) : Int

{map:∀α.∀β.(α→β)→[α]→[β]}
⊢ λα. λβ. λf:α → β. λxs:[α].case xs of
     []    ↦ [] β
     y:ys ↦ (:) β (f y) (map α β f ys)
 : ∀α.∀β.(α→β)→[α]→[β]
```

The standard properties of the second-order typed λ-calculus [Bar92] also hold for F:

Lemma 2.6 (Properties of the type system)

1. *Typing environment*

$$\frac{\Gamma' \supseteq \Gamma \quad \Gamma \vdash M : \tau}{\Gamma' \vdash M : \tau} \qquad \frac{\Gamma \vdash M : \tau}{\mathrm{freeTerm}(M) \subseteq \mathrm{dom}(\Gamma)}$$

$$\frac{\Gamma' + \Gamma \vdash M : \tau \quad \mathrm{freeTerm}(M) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash M : \tau}$$

2. *Typability of subterms: If M has a type, then every subterm of M has a type.*

3. *Substitution properties:*

$$\frac{\Gamma \vdash M : \tau}{\Gamma[\rho/\alpha] \vdash M[\rho/\alpha] : \tau[\rho/\alpha]} \qquad \frac{\Gamma + \{x : \rho\} \vdash M : \tau \quad \Gamma \vdash N : \rho}{\Gamma \vdash M[N/x] : \tau}$$

4. *Uniqueness of types:* $\dfrac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash M : \tau_2}{\tau_1 \equiv \tau_2}$ □

PROOF By induction on the typing rules. ■

Lemma 2.7 (Free type variables in a typing)

$$\frac{\Gamma \vdash M : \tau}{\mathrm{freeTy}(\tau) \subseteq \mathrm{freeTy}(\Gamma) \cup \mathrm{freeTy}(M)}$$ □

PROOF Suppose $\alpha \in \mathrm{freeTy}(\tau) \setminus (\mathrm{freeTy}(\Gamma) \cup \mathrm{freeTy}(M))$. Let β be a new type variable with $\beta \notin \mathrm{freeTy}(\tau) \cup \mathrm{freeTy}(\Gamma) \cup \mathrm{freeTy}(M)$. With Lemma 2.6 follows $\Gamma[\beta/\alpha] \vdash M[\beta/\alpha] : \tau[\beta/\alpha]$. Hence we have $\Gamma \vdash M : \tau[\beta/\alpha]$ with $\tau[\beta/\alpha] \not\equiv \tau$ which contradicts the uniqueness of types stated in Lemma 2.6. So no such α exists and the property holds. ■

$$\frac{}{\lambda x:\tau.M \Downarrow \lambda x:\tau.M}\ \textsc{term abs} \qquad \frac{M \Downarrow \lambda x:\tau.O \quad O[N/x] \Downarrow V}{M\,N \Downarrow V}\ \textsc{term abs app}$$

$$\frac{N\,[\texttt{let}\ \{x_i:\tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ M_1\,/\,x_1,\dots,\texttt{let}\ \{x_i:\tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ M_k\,/\,x_k] \Downarrow V}{\texttt{let}\ \{x_i:\tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N \Downarrow V}\ \textsc{let}$$

$$\frac{}{c \Downarrow c}\ \textsc{cons} \qquad \frac{M \Downarrow c\,\overline{e'}}{M\,e \Downarrow c\,\overline{e'}\,e}\ \textsc{cons app}$$

$$\frac{j \in \{1..k\} \quad M \Downarrow c_j\,\overline{\tau}\,\overline{O} \quad N_j[\overline{O}/\overline{x_j}] \Downarrow V}{\texttt{case}\ M\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k} \Downarrow V}\ \textsc{case}$$

$$\frac{}{\lambda\alpha.M \Downarrow \lambda\alpha.M}\ \textsc{type abs} \qquad \frac{M \Downarrow \lambda\alpha.N \quad N[\tau/\alpha] \Downarrow V}{M\,\tau \Downarrow V}\ \textsc{type abs app}$$

Figure 2.7: Evaluation of closed terms

2.4 Evaluation of Programs

We define the meaning of an F-program, that is, a closed term, by a *big step*, also called *natural* operational semantics [Gun92]. This operational semantics is much simpler than a denotational semantics could be. To denotationally describe the impredicative polymorphism of F rather complex domains would be required, for example partial equivalence relations of domains for an untyped λ-calculus [Gun92, Mit96]. In contrast, an operational semantics just relates terms. Furthermore, an operational semantics is more suitable for our purposes. Many program transformations are directly related to the operational semantics of a language, because they do at compile time what is otherwise done at runtime (*partial evaluation*).

We give a semantics to an F-program by defining to which value it evaluates. A **value** or **weak head normal form** is a closed term of the following form:

$$V ::= c\ \overline{\tau}\ \overline{M} \mid \lambda x:\tau.M \mid \lambda\alpha.M$$

The evaluation relation takes the form

$$M \Downarrow V$$

meaning that the closed term M evaluates to the value V. The relation is defined by the rules given in Figure 2.7. The rules formalise the intuitive semantics given in Section 2.1. The TERM ABS APP and the CONS APP rule show that F has a non-strict semantics[2]. An argument is passed unevaluated to the body of a λ-abstraction, respectively a constructor, in term application. The LET rule exposes the only disadvantage of an operational semantics. In a denotational semantics we could define the recursion simply by using a fixpoint operator of the meta-language. Here we have to unroll the recursive definitions.

[2] We formalise the sharing aspect of lazy evaluation in Section 2.9.

It may seem inconsistent that in the CASE rule the tested term is evaluated to a constructor that is applied to types and terms but in a pattern a constructor occurs only with term variables as arguments. However, a `case` destructs values not types, and hence only term variables are replaced during evaluation.

Note that there is no one-to-one relationship between syntactic constructs and evaluation rules. The TERM ABS APP rule can be used for a term application and the TYPE ABS APP rule can be used for a type application, but the CONS APP rule can be used for both kinds of applications. However, all three rules demand for the evaluation of $M\ e$ that M is evaluated. The value of M determines which APP rule has to be used. Hence the evaluation rules provide a simple, deterministic algorithm for evaluating a closed term.

Already the derivation tree for the evaluation of a simple term exceeds the width of the page:

$$\frac{\dfrac{}{\lambda\texttt{b:Bool. ...} \Downarrow \lambda\texttt{b:Bool. ...}} \qquad \dfrac{\dfrac{}{\texttt{True} \Downarrow \texttt{True}} \quad \dfrac{}{\texttt{False} \Downarrow \texttt{False}}}{\texttt{case True of } \{\texttt{True}{\mapsto}\texttt{False, False}{\mapsto}\texttt{True}\} \Downarrow \texttt{False}}}{(\lambda\texttt{b:Bool. case b of } \{\texttt{True}{\mapsto}\texttt{False, False}{\mapsto}\texttt{True}\})\ \texttt{True} \Downarrow \texttt{False}}$$

Hence, and to stress the sequential nature of evaluation, we use Launchbury's [Lau93] vertical notation for derivation trees of evaluation:

$$\text{Instead of} \quad \frac{\text{derivation}_1 \ldots \text{derivation}_k}{M \Downarrow V} \quad \text{we write} \quad \begin{array}{l} M \\ [\text{derivation}_1 \\ \quad\vdots \\ [\text{derivation}_k \\ V \end{array}$$

$$\text{and we abbreviate trivial derivation trees} \quad \frac{\text{derivation}_1 \ldots \text{derivation}_k}{V \Downarrow V} \quad \text{by} \quad \texttt{[}V.$$

With this notation our example looks as follows:

```
(λb:Bool. case b of {True↦False, False↦True}) True
[λb:Bool. case b of {True↦False, False↦True}
⎡case True of {True↦False, False↦True}
⎢[True
⎢[False
⎣False
False
```

Here is a longer evaluation derivation:

```
let head : ∀α.[α]→α
        = λα.λxs:[α].case xs of {y:ys↦y}
in head Int ((:) Int 1 ([] Int))
⌈(let head:... = ... in λα.λxs:[α].case xs of {y:ys↦y})
│ Int ((:) Int 1 ([] Int))
│⌈(let head:... = ... in λα.λxs:[α].case xs of {y:ys↦y})
││ Int
││⌈let head:... = ... in λα.λxs:[α].case xs of {y:ys↦y}
│││ [λα.λxs:[α].case xs of {y:ys↦y}
││⌊λα.λxs:[α].case xs of {y:ys↦y}
││ [λxs:[Int].case xs of {y:ys↦y}
│⌊λxs:[Int].case xs of {y:ys↦y}
│⌈case (:) Int 1 ([] Int) of {y:ys↦y}
││[(:) Int 1 ([] Int)
││[1
│⌊1
⌊1
1
```

Figure 2.8 shows an even longer example, although with many details elided. The result is a value, but the constructor arguments are not.

Lemma 2.8 (Properties of Evaluation) *Evaluation is deterministic and preserves typing, that is*

$$\frac{M \Downarrow V \quad M \Downarrow V'}{V = V'} \qquad\qquad \frac{M \Downarrow V \quad \vdash M : \tau}{\vdash V : \tau}$$

Proof By induction on the evaluation rules. We assume that the choice of new variables in substitutions is deterministic. ■

Note that the preservation of typing does not imply that well-typed programs cannot "go wrong" [Mil78]. In fact, the term

```
let head : ∀α.[α]→α
         = λα.λxs:[α].case xs of {y:ys↦y}
in head ([] Int)
```

does not evaluate to a value, because the definition of `head` has no pattern that matches the empty list. We say that a term M **converges**, written $M\Downarrow$, if there exists a value V with $M \Downarrow V$, and M **diverges**, written $M\Uparrow$, if there exists no such value. The other cause for divergence besides an "error" is that evaluation of a term might require an infinite derivation tree due to unbound recursion. Because safety of the type system is of no direct concern in this thesis, we neither define a notion of "error" to distinguish it from unbound recursion nor prove the absense of "errors" for well-typed terms. An incomplete pattern in a `case` construct is the only possible cause for an "error" in F and could syntactically be avoided.

In the following we will sometimes use the polymorphic term $\bot$ which diverges for all type arguments:

```
⊥ : ∀α.α
  = λα.let {x : α = x} in x
```

```
let map:∀α.∀β.(α→β)→[α]→[β]
      = λα.λβ.λf:α→β.λxs:[α].case xs of{
          []   ↦[] β
          y:ys↦(:) β (f y) (map α β f ys)}
in map Int Int (λz:Int.z) ((:) Int 1 ([] Int))
⌈(let... in λα.λβ.λf:α→β.λxs:[α].case xs of
|   {[]↦[] β, y:ys↦(:) β (f y) (map α β f ys)} )
|  Int Int (λz:Int.z) ((:) Int 1 ([] Int))
|⌈(let... in λα.λβ.λf:α→β.λxs:[α].case xs of
||   {[]↦[] β, y:ys↦(:) β (f y) (map α β f ys)} )
||  Int Int (λz:Int.z)
||⌈(let... in λα.λβ.λf:α→β.λxs:[α].case xs of
|||   {[]↦[] β, y:ys↦(:) β (f y) (map α β f ys)} )
|||  Int Int
|||⌈(let... in λα.λβ.λf:α→β.λxs:[α].case xs of
||||   {[]↦[] β, y:ys↦(:) β (f y) (map α β f ys)} )
||||  Int
||||⌈let... in λα.λβ.λf:α→β.λxs:[α].case xs of
|||||    {[]↦[] β, y:ys↦(:) β (f y) (map α β f ys)}
|||||⌈λα.λβ.λf:α→β.λxs:[α].case xs of{
||||||    []   ↦[] β
|||||⌊    y:ys↦(:) β (f y) ((let... in λα....) α β f ys)}
||||| λα.λβ.λf:α→β.λxs:[α].case xs of{
|||||    []   ↦[] β
||||⌊    y:ys↦(:) β (f y) ((let... in λα...) α β f ys)}
|||| [...
|||| λβ.λf:Int→β.λxs:[Int].case xs of{
||||    []   ↦[] β
|||⌊    y:ys↦(:) β (f y) ((let... in λα...) Int β f ys)}
||| [...
||| λf:Int→Int.λxs:[Int].case xs of{
|||    []   ↦[] Int
||⌊    y:ys↦(:) Int (f y) ((let... in λα...) Int Int f ys)}
|| [...
|| λxs:[Int].case xs of{
||  []   ↦[] Int
|⌊  y:ys↦(:) Int ((λz:Int.z) y) ((let... in λα...) Int Int (λz:Int.z) ys)}
|⌈case ((:) Int 1 ([] Int)) of{
||  []   ↦[] Int
||  y:ys↦(:) Int ((λz:Int.z) y) ((let... in λα...) Int Int (λz:Int.z) ys)}
|| [((:) Int 1 ([] Int))
|| [...
|⌊(:) Int ((λz:Int.z) 1) ((let... in λα....) Int Int (λz:Int.z) ([] Int))}
⌊(:) Int ((λz:Int.z) 1) ((let... in λα....) Int Int (λz:Int.z) ([] Int))}
(:) Int ((λz:Int.z) 1) ((let... in λα...) Int Int (λz:Int.z) ([] Int))}
```

Figure 2.8: Example evaluation derivation

2.5 Program Transformations

A program transformation is a function that for a given program yields a new program. Using a semantic-preserving program transformation as a compiler optimisation is a well-established technique [ASU86], which has been put into practice especially in the area of functional programming languages [PS98]. Most program transformations can furthermore be broken down into simpler transformations that change only small local parts of a program. We call these simple transformations transformation rules.

We are interested in transformations of F-programs that preserve typing and semantics and improve a program. In this and the following sections we define and study the transformation rules from which our deforestation transformation is built.

An example for a transformation rule is the β-reduction rule

$$(\lambda x : \tau.M)\ N \overset{\beta}{\rightsquigarrow} M[N/x]$$

Basically, a transformation rule is a binary relation $\rightsquigarrow$ on terms. Applying a transformation rule to a program means replacing a subterm M_1 of the program by M_2 where $M_1 \rightsquigarrow M_2$.

We want to type a transformation rule. The need is not obvious for the β-reduction rule. If a program is well-typed, then all its subterms including those of the form $(\lambda x : \tau.M)\ N$ are well-typed (Lemma 2.6). If $(\lambda x : \tau.M)\ N$ is well-typed, then is $M[N/x]$ (cf. Lemma 2.8). Under these conditions we can show that replacing $(\lambda x : \tau.M)\ N$ by $M[N/x]$ gives a well-typed program. However, already applying the inverse rule, that is, β-expansion, causes a problem. How do we obtain the τ in $(\lambda x : \tau.M)\ N$ from the term $M[N/x]$? β-expansion does not preserve typing for an arbitrary τ. Finally, for the short cut fusion rule, which lies at the heart of short cut deforestation, the typing of its components is vital for its semantic correctness.

Because the term relations we are interested in usually relate terms with free variables, the term relations must also contain a typing environment.

Definition 2.9 A **(well-typed) term relation** is a relation R on typing environments, terms and types with

$$R \subseteq \{(\Gamma, M, N, \tau) \mid \Gamma \vdash M : \tau \text{ and } \Gamma \vdash N : \tau\}$$

We write $\Gamma \vdash M\,R\,N : \tau$ for $(\Gamma, M, N, \tau) \in R$ and $\Gamma \vdash M\,\not{R}\,N : \tau$ for $(\Gamma, M, N, \tau) \notin R$ As for typings, we write $\vdash M\,R\,N : \tau$ when M, N and τ contain no free term and type variables.

If R and R' are term relations, then

$$R \text{ is } \textbf{reflexive} \quad \text{iff} \quad \frac{}{\Gamma \vdash M\,R\,M : \tau}$$

$$R \text{ is } \textbf{symmetric} \quad \text{iff} \quad \frac{\Gamma \vdash M\,R\,N : \tau}{\Gamma \vdash N\,R\,M : \tau}$$

$$R \text{ is } \textbf{transitive} \quad \text{iff} \quad \frac{\Gamma \vdash M\,R\,N : \tau \qquad \Gamma \vdash N\,R\,O : \tau}{\Gamma \vdash M\,R\,O : \tau}$$

$$R \textbf{ implies } R' \quad \text{iff} \quad R \subseteq R' \text{ , that is, } \frac{\Gamma \vdash M\,R\,N : \tau}{\Gamma \vdash M\,R'\,N : \tau}$$

□

We include the type in the relation for convenience and because it stresses the connection with typings. According to Lemma 2.6 the type τ is determined uniquely up to renaming of bound variables by Γ and and one of M and N.

A transformation rule is nothing but a well-typed term relation. For example, the β-reduction rule is formally defined as follows:

$$\frac{\Gamma + \{x : \tau\} \vdash M : \rho \qquad \Gamma \vdash N : \tau}{\Gamma \vdash (\lambda x : \tau.M)\ N \overset{\beta}{\rightsquigarrow} M[N/x] : \rho}$$

We still have to define application of a transformation rule. In contrast to term rewriting systems [DJ90, Klo92], F has variable binding constructs. When a transformation rule is applied to a program, the free variables of the subterms that are to be exchanged are bound in the program, for example:

$$\lambda\texttt{y:Int.(}\lambda\texttt{x:Int.x) y} \quad \overset{\beta}{\rightsquigarrow} \quad \lambda\texttt{y:Int.y}$$

Hence we define the **contextual closure** R^{C} of a term relation R by the rules given in Figure 2.9. R^{C} contains R and closes it under all term constructs. Contextual closure involves capture of free variables. Because the rules are obtained mechanically from the type system rules in Figure 2.6, the contextual closure of a well-typed term relation is also a well-typed term relation. A transformation rule determines a transformation by its contextual closure. So formally:

$$\vdash \lambda\texttt{y:Int.(}\lambda\texttt{x:Int.x) y} \overset{\beta}{\rightsquigarrow}^{\mathrm{C}} \lambda\texttt{y:Int.y} : \texttt{Int}\rightarrow\texttt{Int}$$

In the literature it is more common to use the notion of a context to formalise exchanging of a subterm in a program. The idea is that a *context* is a term with an occurrence of a hole $[\cdot]$ at a position where the syntax of terms permits a term variable. $\mathbb{C}[M]$ is the term obtained from the context $\mathbb{C}$ by replacing the hole by the term M. In contrast to normal substitution this *context substitution* may involve capture of free variables. For example, $\mathbb{C} = \lambda x.[\cdot]$ is a context and $\mathbb{C}[x] = \lambda x.x$. Typing a context and assuring that substituting a term of the right type gives again a well-typed term is complicated. A typed version of Pitt's second-order representation of contexts [Pit94, San98] could be used, but context closure is a more tractable notion.

2.6 Contextual Equivalence

A program transformation should not change the program's observable result. Hence we define the notion of contextual (also called observational) equivalence for F [Pit97, Pit00]. Loosely speaking, two terms M and N are contextually equivalent if any occurrence of M in a complete program can be replaced by N without affecting the observable result of executing the program.

We have to specify what the observable results of an F-program are. We decide that the top constructor of a data value, to which a program of data type evaluates, is observable. Furthermore, for a program of function type it is observable if it converges. We will motivate this decision after the formal definition of contextual equivalence.

Definition 2.10 Let Γ be a typing environment, M and M' terms and τ a type with $\Gamma \vdash M : \tau$ and $\Gamma \vdash M' : \tau$. We write

$$\Gamma \vdash M \cong^{\mathrm{ctx}} M' : \tau$$

$$\frac{\Gamma \vdash M\,R\,M' \,:\, \tau}{\Gamma' + \Gamma \vdash M\,R^{\mathrm{C}}\,M' \,:\, \tau}\ \textsc{include}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x\,R^{\mathrm{C}}\,x \,:\, \tau}\ \textsc{var}$$

$$\frac{\Gamma + \{x : \tau_1\} \vdash M\,R^{\mathrm{C}}\,M' \,:\, \tau_2}{\Gamma \vdash \lambda x : \tau_1.M\ R^{\mathrm{C}}\ \lambda x : \tau_1.M' \,:\, \tau_1 \to \tau_2}\ \textsc{term abs}$$

$$\frac{\Gamma \vdash M\,R^{\mathrm{C}}\,M' \,:\, \tau_1 \to \tau_2 \quad \Gamma \vdash N\,R^{\mathrm{C}}\,N' \,:\, \tau_1}{\Gamma \vdash M\,N\ R^{\mathrm{C}}\ M'\,N' \,:\, \tau_2}\ \textsc{term app}$$

$$\frac{\forall i \in \{1..k\} \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash M_i\,R^{\mathrm{C}}\,M'_i \,:\, \tau_i \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash N\,R^{\mathrm{C}}\,N' \,:\, \tau}{\Gamma \vdash \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N\ R^{\mathrm{C}}\ \texttt{let}\ \{x_i : \tau_i = M'_i\}_{i=1}^{k}\ \texttt{in}\ N' \,:\, \tau}\ \textsc{let}$$

$$\frac{\Delta(c) = \tau}{\Gamma \vdash c\,R^{\mathrm{C}}\,c \,:\, \tau}\ \textsc{cons}$$

$$\frac{\begin{array}{c}\Gamma \vdash M\,R^{\mathrm{C}}\,M' \,:\, T\,\overline{\rho}\\ \forall i \in \{1..k\} \quad \Delta(c_i) = \forall\overline{\alpha}.\overline{\rho}_i \to T\,\overline{\alpha} \quad \Gamma + \{\overline{x}_i : \overline{\rho}_i[\overline{\rho}/\overline{\alpha}]\} \vdash N_i\,R^{\mathrm{C}}\,N'_i \,:\, \tau\end{array}}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k}\ R^{\mathrm{C}}\ \texttt{case}\ M'\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto N'_i\}_{i=1}^{k} \,:\, \tau}\ \textsc{case}$$

$$\frac{\Gamma \vdash M\,R^{\mathrm{C}}\,M' \,:\, \tau \quad \alpha \notin \mathrm{freeTy}(\Gamma)}{\Gamma \vdash \lambda\alpha.M\ R^{\mathrm{C}}\ \lambda\alpha.M' \,:\, \forall\alpha.\tau}\ \textsc{type abs}$$

$$\frac{\Gamma \vdash M\,R^{\mathrm{C}}\,M' \,:\, \forall\alpha.\tau}{\Gamma \vdash M\,\rho\ R^{\mathrm{C}}\ M'\,\rho \,:\, \tau[\rho/\alpha]}\ \textsc{type app}$$

Figure 2.9: Contextual closure

iff the singleton term relation $R = \{(\Gamma, M, M', \tau)\}$ satisfies

$$\frac{\vdash N\,R^{\mathrm{C}}\,N' \,:\, T\,\overline{\rho}}{N \Downarrow c\,\overline{\rho}\,\overline{O} \quad\Leftrightarrow\quad N' \Downarrow c\,\overline{\rho}\,\overline{O'}} \qquad \text{and} \qquad \frac{\vdash N\,R^{\mathrm{C}}\,N' \,:\, \rho_1 \to \rho_2}{N\Downarrow \quad\Leftrightarrow\quad N'\Downarrow}$$

We call the term relation $\cong^{\mathrm{ctx}}$ **contextual equivalence**. □

So two terms are contextually equivalent iff all term pairs in their contextual closure have the same observable behaviour.

Our choice of observable behaviour is determined by the observable behaviour of Haskell programs. In Haskell, a program, that is, the function `main`, must be of type `IO ()`, the input/output monad. However, to keep F simple, we did not add monadic input/output to it (see [Gor94b] for a formal descriptions of how we could do so). It is important that the output operations of Haskell can only output values of a data type, that is, constructors, not type abstractions $\lambda\alpha.M$ or term abstractions $\lambda x : \tau.M$. F-terms are only evaluated to weak head normal form. Hence we only compare the top constructors of data values, not their possibly unevaluated arguments. This comparison of top constructors suffices, because we quantify over all program pairs in the contextual closure. For example, to show that the two terms `(:) Int 1 ([] Int)` and `(:) Int 2 ([] Int)` are not contextually equivalent, we only have to find a pair of programs in the contextual closure of its singleton relation that show different observable behaviour:

$$\begin{array}{l}\texttt{case}\ \underline{\texttt{(:) Int 1 ([] Int)}}\ \texttt{of}\ \{\texttt{(:) x xs}\mapsto\texttt{x}\}\Downarrow 1\\ \texttt{case}\ \underline{\texttt{(:) Int 2 ([] Int)}}\ \texttt{of}\ \{\texttt{(:) x xs}\mapsto\texttt{x}\}\Downarrow 2\end{array}$$

The output of a Haskell program is also determined by its input. The necessary quantification over all input is also modelled by the quantification over all program pairs in the contextual closure. In fact, because of this quantification over all program pairs we do not even need to compare the constructors of data values:

Lemma 2.11 (Characterisation of contextual equivalence by convergence) *Let* Γ *be a typing environment,* M *and* M' *terms and* τ *a type with* $\Gamma \vdash M : \tau$ *and* $\Gamma \vdash M' : \tau$. *Then*

$$\Gamma \vdash M \cong^{ctx} M' : \tau$$

iff the singleton term relation $R = \{(\Gamma, M, M', \tau)\}$ *satisfies*

$$\frac{\vdash N\, R^C\, N' : \rho \qquad \rho \textit{ is function or data type}}{N\Downarrow \quad \Leftrightarrow \quad N'\Downarrow} \qquad \Box$$

Proof Obviously contextual equivalence implies the alternative definition. We only need to prove that also for data types a test of convergence suffices.

Let $\vdash N\, R^{\mathrm{C}}\, N' : T\,\overline{\tau}$ with $N \Downarrow c\,\overline{\tau}\,\overline{O}$ and $N' \Downarrow c'\,\overline{\tau}\,\overline{O'}$ and $c \neq c'$. Let

$$\begin{array}{l}P := \texttt{case}\ N\ \texttt{of}\ \{c\,\overline{x} \mapsto c\,\overline{\tau}\,\overline{x},\ c'\,\overline{x} \mapsto \bot\ (T\,\overline{\tau})\}\\ P' := \texttt{case}\ N'\ \texttt{of}\ \{c\,\overline{x} \mapsto c\,\overline{\tau}\,\overline{x},\ c'\,\overline{x} \mapsto \bot\ (T\,\overline{\tau})\}\end{array}$$

Then $\vdash P\, R^{\mathrm{C}}\, P' : T\,\overline{\tau}$, but $P\Downarrow$ and $P'\Uparrow$. So observation of constructors can be reduced to observation of convergence. ■

Our decision that convergence at function type is also observable is motivated by the Haskell function

```
seq :: a -> b -> b
```

This function evaluates its first argument to a value (weak head normal form) and then returns its second argument. The function enables the programmer to enforce strictness, for example to improve the efficiency of a program. Because a term `seq` M `True` yields `True` if and only if evaluation of M converges and M may be of functional type, we can indirectly observe convergence at functional types in Haskell. Hence we also observe convergence at functional types in our definition of contextual equivalence. As noted in Section 5 of [Pit97], this gives the same contextual equivalence relation as if we added `seq` to F and just observed constructors. The convergence test at functional type implies $\Gamma \vdash \bot\ (\tau_1 \to \tau_2) \not\cong^{\mathrm{ctx}} \lambda\texttt{x}\!:\!\tau_1.\,\bot\ \tau_2 : \tau_1 \to \tau_2$. Note, however, that the Haskell type system does not permit an argument of `seq` to be of a universally quantified type. So, convergence at universally quantified type is not observable in Haskell and hence we have $\Gamma \vdash \bot\ (\forall\alpha.\tau) \cong^{\mathrm{ctx}} \lambda\alpha.\,\bot\ \tau : \forall\alpha.\tau$.

Corollary 2.12 *Contextual equivalence is reflexive, symmetric and transitive.* □

Proof By induction on the contextual closure rules. ■

2.7 Kleene Equivalence

To establish that two terms are *not* contextually equivalent is usually quite easy. As demonstrated in the previous section, one just has to find a pair of programs of data or function type in the contextual closure that show different convergence behaviour. In contrast, establishing that a contextual equivalence *does* hold is much harder. The problem lies in the quantification over all pairs of programs in the definition of contextual equivalence. A straightforward induction on the contextual closure rules is not possible, because evaluation of a term usually involves evaluation of larger terms (cf. [Pit97]). Fortunately, a much more tractable characterisation of contextual equivalence has been found. Co-inductive techniques based on the notion of (bi)similarity have been transferred from concurrency theory to functional programming. Contextual equivalence can be characterised as a largest bisimulation. However, this theory is too extensive to be presented here. For an overview the reader is referred to [Pit97, Gor94a, Las98]. Here we only use some results of the theory.

Definition 2.13 **Kleene equivalence**, $\cong^{\mathrm{kl}}$, is a term relation on closed terms defined by the rule:

$$\frac{\forall V \quad M \Downarrow V \Leftrightarrow M' \Downarrow V \qquad \vdash M : \tau \qquad \vdash M' : \tau}{\vdash M \cong^{\mathrm{kl}} M' : \tau}$$ □

Kleene equivalence is only defined for closed terms, because we defined evaluation only for closed terms. However, we extend Kleene equivalence to open terms of open types by comparing all closed instances.

Definition 2.14 We extend Kleene equivalence as defined by the previous definition by closing it under the following rule:

$$\frac{\forall \overline{N} \quad \forall \overline{\rho} \qquad \vdash \overline{N} : \overline{\tau} \qquad \vdash M[\overline{N}/\overline{x}][\overline{\rho}/\overline{\alpha}] \cong^{\mathrm{kl}} M'[\overline{N}/\overline{x}][\overline{\rho}/\overline{\alpha}] : \nu}{\{\overline{x} : \overline{\tau}\} \vdash M \cong^{\mathrm{kl}} M' : \nu}$$ □

Lemma 2.15 *Kleene equivalence implies contextual equivalence, that is* $\cong^{kl} \subseteq \cong^{ctx}$. □

Proof See [Pit97], Proposition 3.9, or [Las98], Proposition 4.3.1. ■

Note that Kleene equivalence does not coincide with contextual equivalence. For example, the terms `(:) Int 1 ([] Int)` and `(:) Int ((`λ`x:Int.x) 1) ([] Int)` are contextually equivalent but obviously not Kleene equivalent.

Lemma 2.16 *The following equivalences hold:*

$$\frac{\Gamma + \{x : \rho\} \vdash M \ : \ \tau \qquad \Gamma \vdash N \ : \ \rho}{\Gamma \vdash (\lambda x.M)N \ \cong^{ctx} \ M[N/x] \ : \ \tau}\ \textit{term application}$$

$$\frac{\Gamma \vdash M \ : \ \tau}{\Gamma \vdash (\lambda\alpha.M)\,\rho \cong^{ctx} M[\rho/\alpha] \ : \ \tau[\rho/\alpha]}\ \textit{type application}$$

$$\frac{\Gamma + \{y : \tau_j\} \vdash \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N \ : \ \tau \qquad y \notin \{x_i\}_{i=1}^{k}}{\begin{array}{ll}\Gamma \vdash & \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N[x_j/y] \\ \cong^{ctx} & \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N[\texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ M_j \,/\, y] \\ : & \tau\end{array}}\ \textit{inlining}[3]$$

$$\frac{\Gamma \vdash \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N \ : \ \tau \qquad \{x_i\}_{i=1}^{k} \cap \mathrm{free}(N) = \emptyset}{\begin{array}{ll}\Gamma \vdash & \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N \\ \cong^{ctx} & N \\ : & \tau\end{array}}\ \textit{\texttt{let} elimination}$$

□

Proof For every term relation given above the closed instances of related terms evaluate to the same value. Then contextual equivalence follows with Lemma 2.15. ■

We call both the use of the term application and of the type application relation from left to right ***β*-reduction**. By **inlining** we often refer to a combination of the actual inlining relation with subsequent β-reduction. After inlining of all occurrences of a variable let elimination is sensible.

2.8 The Short Cut Fusion Rule

For the definition of the short cut fusion rule types are vital.

Definition 2.17 The **short cut fusion rule** $\overset{\text{fusion}}{\rightsquigarrow}$ is defined as follows:

$$\frac{\begin{array}{c}\Gamma \vdash M^{(:)} \ : \ \tau_1 \to \tau_2 \to \tau_2 \qquad \Gamma \vdash M^{[]} \ : \ \tau_2 \\ \Gamma \vdash P \ : \ \forall\gamma.(\tau_1 \to \gamma \to \gamma) \to \gamma \to \gamma\end{array}}{\Gamma \vdash \texttt{foldr}\ \tau_1\ \tau_2\ M^{(:)}\ M^{[]}\ \texttt{(}P\ \texttt{[}\tau_1\texttt{]}\ \texttt{((:)}\,\tau_1\texttt{)}\ \texttt{([]}\,\tau_1\texttt{))} \overset{\text{fusion}}{\rightsquigarrow} P\ \tau_2\ M^{(:)}\ M^{[]}}$$

The idea of the short cut fusion rule is that because of its polymorphic type the producer skeleton P constructs its result only with its two arguments (and these two arguments are not used for any other purpose). It is vital that we demand in the definition that P has type $\forall\gamma.(\tau_1 \to \gamma \to \gamma) \to \gamma \to \gamma$. This typing would not follow if we just demanded that left and right side of the rule are well-formed terms of the same type. Dropping the condition on the type of P would for example admit for $\tau_2 = [\tau_1]$ the producer skeleton

$$\begin{array}{ll}\texttt{P} \ : & \forall\gamma.\texttt{(}\tau_1 \to \texttt{[}\tau_1\texttt{]} \to \texttt{[}\tau_1\texttt{])} \to \texttt{[}\tau_1\texttt{]} \to \texttt{[}\tau_1\texttt{]} \\ \ \ \texttt{=} & \lambda\gamma.\,\lambda\texttt{c:}\tau_1 \to \texttt{[}\tau_1\texttt{]} \to \texttt{[}\tau_1\texttt{]}.\,\lambda\texttt{n:[}\tau_1\texttt{].}\ \texttt{[]}\ \tau_1\end{array}$$

[3] Usually "inlining" only refers to the substitution of right hand sides of definitions for variables, not complete recursive definitions. Here we use the term in a more liberal sense which is sometimes referred to as "specialisation".

which obviously does not use its arguments to construct its result.

As it stands the short cut fusion rule is neither well-typed nor does it imply contextual equivalence. This is because `foldr` is an arbitrary free variable in the rule whereas for well-typedness and our informal correctness argument we assumed it to be defined as follows:

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

With this definition contextual equivalence holds:

$$\frac{\begin{array}{c}\Gamma \vdash M^{(:)} : \tau_1 \to \tau_2 \to \tau_2 \qquad \Gamma \vdash M^{[]} : \tau_2 \\ \Gamma \vdash P : \forall\gamma.(\tau_1 \to \gamma \to \gamma) \to \gamma \to \gamma\end{array}}{\begin{array}{ll}\Gamma \vdash & \texttt{(let foldr ... in foldr)}\ \tau_1\ \tau_2\ M^{(:)}\ M^{[]}\ (P\ \texttt{[}\tau_1\texttt{]}\ \texttt{((:)}\,\tau_1\texttt{)}\ \texttt{([]}\,\tau_1\texttt{))} \\ \cong^{\text{ctx}} & P\ \tau_2\ M^{(:)}\ M^{[]} \\ : & \tau_2\end{array}} \quad (*)$$

We can apply the short cut fusion rule nonetheless, because in Haskell `foldr` is predefined in all programs (cf. Lemma 2.16 about inlining). We do however have to take care that the variable `foldr` is not a locally defined variable that happens to have the same name and hides “our” `foldr`.

We do not give a proof here that (∗) holds but just refer to relevant work. The two sides of (∗) are not Kleene equivalent. Nor is the mentioned bisimulation characterisation useful for proving contextual equivalence of the two sides. However, as stated already in the introduction, the contextual equivalence of the two sides is a direct consequence of the so called free theorem of the polymorphic type of the producer skeleton.

Reynolds was the first to show that the second-order typed λ-calculus is **parametric**, which intuitively means that the semantics of different instances of a polymorphic term are closely related [Rey83, MR92]. Later Wadler noticed that parametricity implies that for every type there exists a semantic equivalence theorem that holds for every term of this type [Wad89]. For a monomorphic type this theorem is trivial, but not so for a polymorphic type. For these theorems Wadler coined the term **theorems for free**.

Reynold's proof is based on a simple set theoretical model of the second-order λ-calculus that was later proved not to exist [Rey84]. Wadler uses a complex denotational model. Pitts gives a detailed proof of parametricity for contextual equivalence [Pit00]. Furthermore, he uses a second-order typed λ-calculus with recursion and a list data type, which is very similar to F. The only technical problem in applying his proofs to F is that F permits contravariant data types (see Chapter 6). Pitt's theory is only applicable to languages with covariant data types and he only gives hints on how this restriction could be overcome. However, just defining all notions necessary to state parametricity would require a chapter of its own. The central theme of this thesis is not the fusion rule but how a producer can be transformed so that the short cut fusion rule can be applied. The short cut fusion rule itself is already known from [Gil96, GLP93], where also an informal proof of its correctness based on Wadler's free theorems is given.

2.9 A Cost Model

The purpose of deforestation is to reduce the costs of a program. There are various kinds of costs. A simple kind of cost is the program size. The size of a term is a good approximation of the final program code size. The effect of a transformation rule on the size of a term is easy to determine. Small program code size is seldom an important aim in practice — in fact, most program transformations trade program size for reduction of other costs — but it should not be neglected. Repeated application of transformation rules may easily lead to an exponential growth of the program, so that even the compiler itself can no longer handle the program. Deforestation methods based on unfold/fold transformations are known to suffer from the large scale inlining they perform (cf. Section 7.3).

Here we are mainly interested in the time and space costs of a running program. Obviously, a complete analysis of the effect of a transformation on these costs is impossible. As a first step the relationship with all other transformations that are performed by a compiler would need to be analysed. However, even hardware details such as the size of the processor cache and the word size of the processor influences time and space costs. Despite the impossiblity of a complete cost analysis we need some description of the most relevant properties of program execution as a guide for the design of a transformation. For this purpose our operational evaluation semantics is not sufficient. It defines the observable property of lazy evaluation, namely non-strictness, but it does not describe one important implementation aspect of laziness: sharing. Laziness does not only require that a function argument is only evaluated if it is needed for obtaining the result, but furthermore, that such a function argument is evaluated at most once. The following example demonstrates that our evaluation semantics duplicates unevaluated arguments and thus also duplicates their evaluation:

```
(λx:Bool. case x of {True↦x}) ((λy:Bool.y) True)
 [λx:Bool. case x of {True↦x}
 ⎡case ((λy:Bool.y) True) of {True↦((λy:Bool.y) True)}
 ⎢⎡(λy:Bool.y) True ⎫
 ⎢⎢[λy:Bool.y       ⎬ first evaluation
 ⎢⎢[True            ⎪
 ⎢⎣True             ⎭
 ⎢⎡(λy:Bool.y) True ⎫
 ⎢⎢[λy:Bool.y       ⎬ second evaluation
 ⎢⎢[True            ⎪
 ⎢⎣True             ⎭
 ⎣True
True
```

Abstract machines that describe the implementation of lazy functional languages also describe sharing. However, there exists a large number of different abstract machines for lazy evaluation, for example the G-machine [Joh84], the STG-machine [Pey92], the TIM [FW87] and TIGRE [KL89]. More importantly, abstract machines are already too detailed for reasoning about them.

2.9.1 A Semantics for Lazy Evaluation

John Launchbury defined a natural operational semantics for lazy evaluation [Lau93] that lies midway between our evaluation semantics and an operational semantics of an abstract machine. This semantics has been adopted by a number of researchers as the formal definition of lazy evaluation [TWM95, SP97, WP99, MS99, GS99]. Similar semantics had been independently developed for PCF [SI96] and Clean [BS98]. Furthermore, Sestoft formally derived a simple abstract machine from this lazy semantics [Ses97]. Hence we adopt this semantics of lazy evaluation here as well.

$$
\begin{array}{lrcl}
\text{G-terms} & A, B & ::= & x \mid \lambda x.A \mid A\,x \mid \texttt{let}\ \{x_i = A_i\}_{i=1}^{k}\ \texttt{in}\ B \mid \\
 & & & c \mid \texttt{case}\ A\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto B_i\}_{i=1}^{k} \\
\text{G-values} & Z & ::= & \lambda x.A \mid c\,\overline{x}
\end{array}
$$

Figure 2.10: Syntax of untyped terms

$$
\begin{aligned}
x^* &:= x \\
(\lambda x : \tau.M)^* &:= \lambda x.(M^*) \\
(M\,N)^* &:= \begin{cases} M^*\,x & \text{, if } N = x \\ \texttt{let}\ x = N^*\,\texttt{in}\,(M^*)\,x & \text{, otherwise,} \\ \quad \text{where } x \text{ does not occur free in } N \text{ or } M & \end{cases} \\
(\texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N)^* &:= \texttt{let}\ \{x_i = M_i^*\}_{i=1}^{k}\ \texttt{in}\ N^* \\
c^* &:= c \\
(\texttt{case}\ M\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto N_i\}_{i=1}^{k})^* &:= \texttt{case}\ M^*\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto N_i^*\}_{i=1}^{k} \\
(\lambda\alpha.M)^* &:= M^* \\
(M\,\tau)^* &:= M^*
\end{aligned}
$$

Figure 2.11: Translation from F-terms to G-terms

$$
\frac{}{\Phi : \lambda x.A \Downarrow \Phi : \lambda x.A}\ \text{ABS}
\qquad
\frac{\Phi : A \Downarrow \Psi : \lambda y.B \qquad \Psi : B[x/y] \Downarrow \Omega : Z}{\Phi : A\,x \Downarrow \Omega : Z}\ \text{ABS APP}
$$

$$
\frac{\Phi : \Phi(x) \Downarrow \Psi : Z}{\Phi : x \Downarrow \Psi[x \mapsto Z] : Z}\ \text{VAR}
\qquad
\frac{\exists\overline{y} \qquad \Phi + \{y_i \mapsto A_i[\overline{y}/\overline{x}]\}_{i=1}^{k} : B[\overline{y}/\overline{x}] \Downarrow \Psi : Z}{\Phi : \texttt{let}\ \{x_i = A_i\}_{i=1}^{k}\ \texttt{in}\ B \Downarrow \Psi : Z}\ \text{LET}
$$

$$
\frac{}{\Phi : c \Downarrow \Phi : c}\ \text{CONS}
\qquad
\frac{\Phi : A \Downarrow \Psi : c\,\overline{y}}{\Phi : A\,x \Downarrow \Psi : c\,\overline{y}\,x}\ \text{CONS APP}
$$

$$
\frac{j \in \{1..k\} \qquad \Phi : A \Downarrow \Psi : c_j\,\overline{y} \qquad \Psi : B_j[\overline{y}/\overline{x_j}] \Downarrow \Omega : Z}{\Phi : \texttt{case}\ A\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto B_i\}_{i=1}^{k} \Downarrow \Omega : Z}\ \text{CASE}
$$

Figure 2.12: Lazy Evaluation

We define lazy evaluation for a language G, whose syntax is given in Figure 2.10. The language is untyped, because types are not needed for evaluation. Besides types, the only difference in syntax between G and F is that in G the argument in an application has to be a variable. This restriction considerably simplifies the lazy evaluation semantics, because the semantics only has to assure that `let`-defined variables are shared. The syntax assures that the argument in every application is `let`-defined. G-values directly correspond to F-values, except that no type application exists and that all arguments of a data constructor have to be variables. Figure 2.11 gives a translation from F-terms into G-terms that drops all type information and introduces `let` constructs where necessary.

To express sharing the semantics for lazy evaluation operates on graphs instead of terms. For graphs a machine-oriented representation is chosen. A graph $\Phi : A$ consists of a heap Φ and a G-term A. The term component represents the top of the graph and the heap contains possibly shared parts. A heap, denoted by Φ, Ψ or Ω, is a partial mapping from variables to G-terms. We write it as a set of tuples $x \mapsto A$. By domain of a heap we mean the domain of the mapping. The operator $+$ combines two heaps under the assumption that their domains are disjoint. We only consider closed graphs, that is, graphs where the free variables of the term and of all terms in the heap are a subset of the domain of the heap. So a closed graph represents a closed term with shared subterms. For example,

$$\{\texttt{z}\mapsto(\lambda\texttt{y:Bool.y})\ \texttt{t},\ \texttt{t}\mapsto\texttt{True}\} : \texttt{case z of } \{\texttt{True}\mapsto\texttt{z}\}$$

may be graphically visualised as

$$\texttt{case z of } \{\texttt{True}\mapsto\texttt{z}\}$$

$$(\lambda\texttt{y:Bool.y})\ \texttt{t}$$

$$\texttt{True}$$

and represents the term

$$\texttt{case }(\lambda\texttt{y:Bool.y})\texttt{ True of } \{\texttt{True}\mapsto(\lambda\texttt{y:Bool.y})\texttt{ True}\}$$

In Figure 2.12 the semantics of lazy evaluation is given. Similar to the definition of evaluation in Section 2.4 we define a **lazy evaluation relation**

$$\Phi : A \Downarrow \Psi : Z$$

meaning that the graph $\Phi : A$ evaluates to the graph $\Psi : Z$. We write $\Phi : A\Downarrow$, if there exists a graph $\Psi : Z$ with $\Phi : A \Downarrow \Psi : Z$. The rules ABS, CONS and CONS APP directly correspond to their F-counterpart. The rules ABS APP and CASE are slightly simpler, because we only substitute variables for variables. Thus the duplication of subterms that happens during the evaluation of F-terms is avoided. The central parts of lazy evaluation are the LET and the VAR rule. The LET rule just adds new variable bindings to the heap. Thus the definition bodies of the `let` construct are shared. Recursive definitions create cyclic graphs. Finally, the VAR rule assures that a term in the heap is evaluated at most once. To evaluate a variable x, its corresponding term in the heap is evaluated. Because we only consider closed graphs, the variable x must be in the domain of the heap. The resulting graph is also the result of evaluating x. However, by $\Psi[x \mapsto Z]$ we mean that the heap is updated to map x to the value Z. Hence another evaluation of x will directly yield Z without any further evaluation.

As an example consider the term that demonstrated the lack of sharing in the evaluation semantics of F-terms.

$$\begin{aligned}
&((\lambda\texttt{x:Bool. case x of } \{\texttt{True}\mapsto\texttt{x}\})\ ((\lambda\texttt{y:Bool.y})\texttt{ True}))^{*} \\
= \ &\texttt{let z = (let t=True in }(\lambda\texttt{y:Bool.y})\texttt{ t)} \\
&\texttt{in }(\lambda\texttt{x:Bool. case x of } \{\texttt{True}\mapsto\texttt{x}\})\texttt{ z}
\end{aligned}$$

For lazy evaluation we use the same vertical notation as for evaluation.

```
{} : let z = (let t = True in (λy:Bool.y) t)
       in (λx:Bool. case x of {True↦x}) z
┌{z↦let t = True in (λy:Bool.y) t} : (λx:Bool. case x of {True↦x}) z
│ [  "  : λx:Bool. case x of {True↦x}
│┌  "  : case z of {True↦z}
││┌  "  : z                                   ⎫
│││┌  "  : let t = True in (λy:Bool.y) t      ⎪
││││┌{..., t↦True} : (λy:Bool.y) t           ⎪
│││││ [  "  : λy:Bool.y                      ⎪
│││││┌  "  : t                               ⎪
││││││ [  "  : True                          ⎬ evaluation
│││││└  "  : True                            ⎪
││││└  "  : True                             ⎪
│││└  "  : True                              ⎪
││└{z↦True, ...} : True                      ⎭
││┌  "  : z       ⎫
│││ [  "  : True  ⎬ no repeated evaluation, only look up
││└  "  : True    ⎭
│└  "  : True
└  "  : True
{z↦True, t↦True} : True
```

Note that the computation is longer than for the unshared evaluation semantics, because the lazy semantics describes evaluation at a higher level of detail.

The two semantics agree in the following sense. The translation * from F-terms into G-terms is **adequate**, that is, evaluation of an F-term and lazy evaluation of its translation have the same convergence behaviour:

$$\frac{\vdash M : \tau \qquad \tau \text{ not a universal type}}{M\Downarrow \;\Leftrightarrow\; \{\} : M^*\Downarrow\!\!\Downarrow}$$

Because of the quantification over all F-terms this implies especially

$$\frac{\vdash M : T\overline{\tau}}{M \Downarrow c\overline{\tau}\,\overline{N} \;\Rightarrow\; \exists \Psi, \overline{x} \quad \{\} : M^*\Downarrow\!\!\Downarrow \Psi : c\overline{x}} \quad \text{and} \quad \frac{\vdash M : T\overline{\tau}}{\{\} : M^*\Downarrow\!\!\Downarrow \Psi : c\overline{x} \;\Rightarrow\; \exists \overline{N} \quad M \Downarrow c\overline{\tau}\,\overline{N}}$$

It would be too much here to extend the translation * to a binary relation between F-terms and graphs and prove that the relation is a bisimulation. The adequacy result indirectly follows from [Abr90], where the agreement between a natural evaluation semantics and a denotational semantics is proved, and [Lau93], where the agreement between this denotational semantics and his lazy evaluation semantics is proved.

Our lazy evaluation semantics differs from Launchbury's in two respects. First, Launchbury requires all bound variable to be distinct and renames variables in the VAR rule. We do not require distinct bound variables and rename variables in the LET rule, as suggested in [Ses97]. The introduction of new variables in the LET rule is closer to an actual implementation. Second, in Launchbury's VAR rule the reference to x is omitted in the heap during the evaluation of $\Phi(x)$. This omission models the so-called "black-holing" in implementations of lazy languages [Pey92]. Mainly it enables the garbage collector to recognise more parts of the heap as garbage. We left this detail out for the sake of simplicity.

2.9.2 Improvement

The lazy evaluation semantics is well suited to determine and compare costs of evaluation. The number of high-level steps to evaluate a program P, that is, the size of the derivation tree of $\{\} : P^* \Downarrow \Phi : Z$, is an abstract measure for the runtime of P. Because the heap is modelled by the semantics, the number of heap cells that are allocated during evaluation can be determined. Note that the size of the heap is not a measure of the space costs, because parts of the heap may be garbage, that is, they are no longer reachable from the term component of the graph. The lazy evaluation semantics can be extended to identify garbage and even perform garbage collection ([Lau93], Section 6.2).

We say that a transformation rule $\Gamma \vdash M \rightsquigarrow M' : \tau$ is **improving**, if for all programs P and P' in its contextual closure P' has lower evaluation costs than P. Thus improvement is described similarly to contextual equivalence, but it is more complex, because the lazy evaluation semantics is more complex. This notion of improvement has been formally defined for runtime in [MS99] and for space in [GS99]. Unfortunately, the theories developed there are still too weak to prove improvement for many of the transformation rules used in our deforestation algorithm, especially inlining and the fusion rule. Nonetheless, the lazy evaluation semantics and improvement serve as background for intuitive arguments about the effect of a transformation. Furthermore, they are a good basis to show that certain transformation rules could worsen performance. In general inlining may increase evaluation costs by duplicating evaluation. However, inlining improves runtime performance for example if the inlined definition body is a value or the variable whose definition body is inlined is used at most once [PM99].

Chapter 3

Type-Inference Based Deforestation

In this chapter we explain how our new deforestation algorithm works. We mostly use examples to illustrate the working of the algorithm and discuss certain design decisions. The central part of the deforestation algorithm, the list abstraction algorithm, will be presented fully and formally and proved to be complete and correct in the following chapter. The remaining part of the deforestation algorithm is quite simple and is a combination of several well-known transformations, such as inlining, whose correctness we have proved in the previous chapter.

3.1 List Abstraction through Type Inference

We deforest a program with the short cut fusion rule

$$\frac{\begin{array}{c}\Gamma \vdash M^{(:)} : \tau_1 \to \tau_2 \to \tau_2 \qquad \Gamma \vdash M^{[]} : \tau_2 \\ \Gamma \vdash P : \forall\gamma.(\tau_1 \to \gamma \to \gamma) \to \gamma \to \gamma\end{array}}{\Gamma \vdash \mathtt{foldr}\ \tau_1\ \tau_2\ M^{(:)}\ M^{[]}\ (P\ [\tau_1]\ ((:)\,\tau_1)\ ([]\,\tau_1)) \overset{\text{fusion}}{\rightsquigarrow} P\,\tau_2\,M^{(:)}\ M^{[]}}$$

that we discussed in Section 2.8. Our deforestation algorithm searches for terms of the form $\mathtt{foldr}\ \tau_1\ \tau_2\ M^{(:)}\ M^{[]}\ M^{\mathtt{P}}$ and then tries to transform the producer $M^{\mathtt{P}}$ into the form $P\ [\tau_1]\ ((:)\,\tau_1)\ ([]\,\tau_1)$ with $\Gamma \vdash P : \forall\gamma.(\tau_1 \to \gamma \to \gamma) \to \gamma \to \gamma$ to subsequently apply the fusion rule. To obtain the polymorphic producer skeleton P the transformation abstracts from the producer $M^{\mathtt{P}}$ the list type $[\tau_1]$ and the constructors $(:)\,\tau_1$ and $[]\,\tau_1$ that construct the result list. We call this transformation **list abstraction**.

3.1.1 Basic List Abstraction

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

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

For the moment we only consider monomorphic list constructors

$$\Delta = \{\texttt{(:):Int}\to\texttt{[Int]}\to\texttt{[Int]},\ \texttt{[]:[Int]},\dots\}$$

and a monomorphic version of `map` for type `Int`. Furthermore we assume that the definition of `mapInt` is part of the producer. We will lift these restrictions step by step in the following sections. We start list abstraction with the typing of the producer:

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

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

```
{inc:Int→Int, $v_1^{(:)}$:Int→ $\gamma_1 \to \gamma_1$, $v_2^{(:)}$:Int→ $\gamma_2 \to \gamma_2$,
 $v_3^{(:)}$:Int→ $\gamma_3 \to \gamma_3$, $v_1^{[]}$:$\gamma_4$, $v_2^{[]}$:$\gamma_5$}
⊢ let mapInt : (Int→Int)→ $\gamma_6 \to \gamma_7$
             = λf:Int→Int.λxs:$\gamma_8$.case xs of
                 []   ↦ $v_1^{[]}$
                 y:ys ↦ $v_1^{(:)}$ (f y) (mapInt f ys)
  in mapInt inc ($v_2^{(:)}$ 1 ($v_3^{(:)}$ 2 $v_2^{[]}$))
```

This typing environment and term with type variables do not form a valid typing for any type. They are the input to a type inference algorithm that we will present in Section 4.3.5. The type inference algorithm replaces some of the new type variables $\gamma_1, \dots, \gamma_8$ and determines a type to obtain again a valid typing. More precisely, the type inference algorithm determines a principal typing, that is, the most general instance of the input that gives a valid typing. Note that type inference cannot fail, because the typing we start with is valid. In the worst case the type inference algorithm yields the typing of the original producer. We just try to find a more general typing. For our example the type inference algorithm yields the valid typing:

```
{inc:Int→Int, $v_1^{(:)}$:Int→ $\gamma \to \gamma$, $v_2^{(:)}$:Int→[Int]→[Int],
 $v_3^{(:)}$:Int→[Int]→[Int], $v_1^{[]}$:$\gamma$, $v_2^{[]}$:[Int]}
⊢ let  mapInt : (Int→Int)→ [Int] → $\gamma$
              = λf:Int→Int.λxs:[Int].case xs of
                  []   ↦ $v_1^{[]}$
                  y:ys ↦ $v_1^{(:)}$ (f y) (mapInt f ys)
  in mapInt inc ($v_2^{(:)}$ 1 ($v_3^{(:)}$ 2 $v_2^{[]}$))
: $\gamma$
```

The type of the term is a type variable γ and nothing other than the $\mathtt{v}_i^{(:)}$'s and the $\mathtt{v}_i^{[]}$'s has γ in its type. Hence list abstraction is possible: The typing environment tells us that $\mathtt{v}_1^{(:)}$ and $\mathtt{v}_1^{[]}$ construct values of type γ, so they construct the result of the producer. In contrast $\mathtt{v}_2^{(:)}$, $\mathtt{v}_3^{(:)}$, and $\mathtt{v}_2^{[]}$ have the types of normal list constructors. Hence they construct lists that are internal to the producer. So we reinstantiate $\mathtt{v}_2^{(:)}$, $\mathtt{v}_3^{(:)}$, and $\mathtt{v}_2^{[]}$ to normal list constructors and abstract the type variable γ and the variables $\mathtt{v}_1^{(:)}$ and $\mathtt{v}_1^{[]}$ to obtain the producer skeleton of the required type:

```
{inc:Int→Int}
⊢ $\lambda\gamma.\ \lambda\mathtt{v}_1^{(:)}$:Int→ $\gamma \to \gamma$. $\lambda\mathtt{v}_1^{[]}$:$\gamma$.
    let mapInt : (Int→Int)→ [Int] → $\gamma$
               = $\lambda$f:Int→Int.$\lambda$xs:[Int].case xs of
                   []    ↦ $\mathtt{v}_1^{[]}$
                   y:ys ↦ $\mathtt{v}_1^{(:)}$ (f y) (mapInt f ys)
    in mapInt inc ((:) 1 ((:) 2 []))
: $\forall\gamma.$(Int→ $\gamma \to \gamma) \to \gamma \to \gamma$
```

The complete producer can be written as

```
($\lambda\gamma.\ \lambda\mathtt{v}_1^{(:)}$:Int→ $\gamma \to \gamma$. $\lambda\mathtt{v}_1^{[]}$:$\gamma$. let ... in ...) [Int] (:) []
```

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

3.1.2 Polymorphic List Constructors

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

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

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

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

```
{inc:Int→Int}
⊢ $\lambda\gamma.\ \lambda\mathtt{v}_1^{(:)}$:Int→ $\gamma \to \gamma$. $\lambda\mathtt{v}_1^{[]}$:$\gamma$.
    let mapInt : (Int→Int)→[Int]→ $\gamma$
               = $\lambda$f:Int→Int.$\lambda$xs:[Int].case xs of
                   []    ↦ $\mathtt{v}_1^{[]}$
                   y:ys ↦ $\mathtt{v}_1^{(:)}$ (f y) (mapInt f ys)
    in mapInt inc ((:) Int 1 ((:) Int 2 ([] Int)))
: $\forall\gamma.$(Int→ $\gamma \to \gamma) \to \gamma \to \gamma$
```

and the complete producer looks as follows:

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

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

3.2 Inlining of Definitions

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

3.2.1 Determination of Variables that Need to be Inlined

Rather nicely, instead of having to use some heuristics for inlining, we can use the typing environment of a principal typing to determine exactly those variables whose definitions need to be inlined to enable list abstraction. Note that it is important not just to inline the right hand side of a recursive definition but the whole recursive definition.

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

$$\begin{array}{l}\{\mathtt{inc}{:}\mathtt{Int}\!\to\!\mathtt{Int},\ \mathtt{mapInt}{:}\mathtt{(Int\!\to\!Int)\!\to\![Int]\!\to\![Int]}\}\\ \vdash \mathtt{mapInt\ inc\ ((:)\ Int\ 1\ ((:)\ Int\ 2\ ([]\ Int)))}\\ :\ \mathtt{[Int]}\end{array}$$

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

$$\begin{array}{l}\{\mathtt{inc}{:}\mathtt{Int}\!\to\!\mathtt{Int},\ \mathtt{mapInt}{:}\mathtt{(Int\!\to\!Int)}\!\to\gamma_1\to\gamma_2,\\ \ \mathtt{v}_1^{(:)}{:}\mathtt{Int}\!\to\gamma_3\to\gamma_3,\ \mathtt{v}_2^{(:)}{:}\mathtt{Int}\!\to\gamma_4\to\gamma_4,\ \mathtt{v}_1^{[]}{:}\gamma_5\}\\ \vdash \mathtt{mapInt\ inc\ (v}_1^{(:)}\ \mathtt{1\ (v}_2^{(:)}\ \mathtt{2\ v}_1^{[]}\mathtt{))}\end{array}$$

The type inference algorithm gives us the principal typing

$$\begin{array}{l}\{\mathtt{inc}{:}\mathtt{Int}\!\to\!\mathtt{Int},\ \mathtt{mapInt}{:}\mathtt{(Int\!\to\!Int)}\!\to\gamma_1\to\gamma_2,\\ \ \mathtt{v}_1^{(:)}{:}\mathtt{Int}\!\to\gamma_1\to\gamma_1,\ \mathtt{v}_2^{(:)}{:}\mathtt{Int}\!\to\gamma_1\to\gamma_1,\ \mathtt{v}_1^{[]}{:}\gamma_1\}\\ \vdash \mathtt{mapInt\ inc\ (v}_1^{(:)}\ \mathtt{1\ (v}_2^{(:)}\ \mathtt{2\ v}_1^{[]}\mathtt{))}\\ :\ \gamma_2\end{array}$$

The type of the term is a type variable γ_2. However, we cannot abstract γ_2, because it appears in the typing environment in the type of a variable other than the $\mathtt{v}_i^{(:)}$'s and the $\mathtt{v}_i^{[]}$'s. The type variable γ_2 also occurs in the type of `mapInt`. This occurrence of γ_2 signifies that the definition of `mapInt` needs to be inlined.

We do not even have to repeat the whole substitution and type inference process for the extended producer, that is, the producer with the definition of `mapInt` inlined. Instead we can continue processing the right hand side of `mapInt` inside the extended producer. We just have to unify the resulting type of the right hand side with $\mathtt{(Int\!\to\!Int)}\!\to\gamma_1\to\gamma_2$. Thus we can inline definitions and process them until either the type of the extended producer becomes [Int] and list abstraction fails, or it is a type variable that appears in no type of a variable of the typing environment, except those of the $\mathtt{v}_i^{(:)}$ and $\mathtt{v}_i^{[]}$ of course. Subsequently, all processed (potentially mutually recursive) definitions are put into a single let binding. For our example the output coincides modulo variable names with the output from the previous section. Note that only the definition of mapInt is inlined, not the definition of inc.

To ensure that code explosion never happens, an implementation of inlining will probably abandon list constructor abstraction when the inlined code exceeds a predefined size.

3.2.2 Instantiation of Polymorphism

There is one limitation still present in our example. The producer should use the polymorphic function map instead of mapInt.

$$\begin{array}{l}\{\mathtt{inc}{:}\mathtt{Int}{\rightarrow}\mathtt{Int},\ \mathtt{map}{:}\forall\alpha.\forall\beta.(\alpha{\rightarrow}\beta){\rightarrow}[\alpha]{\rightarrow}[\beta]\}\\ \vdash \mathtt{map\ Int\ Int\ inc\ ((:)\ Int\ 1\ ((:)\ Int\ 2\ ([]\ Int)))}\\ :\ \mathtt{[Int]}\end{array}$$

On this input our current list abstraction algorithm fails. In the type of map no type [Int] occurs that could be replaced by a new type variable and thus type inference will assign the producer the type [Int] instead of a type variable. Replacing the types $[\alpha]$ and $[\beta]$ by type variables and abstracting these type variables later is not semantically sound, because α and β are bound locally.

The solution is to instantiate polymorphic list manipulating functions that are used in the producer as far as possible before applying the list abstraction algorithm as described before. So in the producer we replace map Int Int by a new variable $\mathtt{map}_{\mathtt{Int\ Int}}$. We obtain a definition for $\mathtt{map}_{\mathtt{Int\ Int}}$ by dropping the abstraction from α and β in the definition of map and replacing both α and β by Int. Thus we reach the same situation as in the previous subsection and can abstract the produced list successfully.

In the general case we 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 F yields a program where only definition bodies in let constructs are type-abstracted and term variables of polymorphic type occur only in type applications. Programs in the intermediate language of the Glasgow Haskell compiler are nearly completely in this form, because the Haskell type systems is based on the Hindley-Milner type system.

More examples and information about about how type-inference based inlining can be implemented efficiently are given in [Chi99]. We do not go into these details here, because in the next section we will present a better alternative to inlining.

3.3 The Worker/Wrapper Scheme

It is neat that the algorithm determines exactly the functions that need to be inlined, but nonetheless inlining causes problems in practice. Extensive inlining across module boundaries would defeat the idea of separate compilation. Furthermore, we noted in Section 2.9.2 that inlining may also decrease performance and in practice “inlining is a black art, full of delicate compromises that work together to give good performance without unnecessary code bloat" [PM99]. It is best implemented as a separate optimisation pass. Consequently, we would like to use our list abstraction algorithm without it having to perform inlining itself.

To be able to abstract the result list from a producer without using inlining, all list constructors that construct the result list already have to be present in the producer. Therefore we use a so called worker/wrapper scheme as it is used inside the Glasgow Haskell compiler for transferring strictness information [PL91] and has been proposed by Gill for short cut deforestation (see Section 7.1.1). The idea is to automatically split every definition of a function that produces a list into a definition of a worker and a definition of a wrapper. The definition of the worker is obtained from the original definition by abstracting the result list type and its list constructors. The definition of the wrapper, which calls the worker, contains all the list constructors that construct the result list. For example, we split the definition of the function `mapInt`

```
mapInt : (Int→Int)→[Int]→[Int]
       = λf:Int→Int.
            foldr Int [Int] (λu:Int.λw:[Int].(:) Int (f u) w) ([] Int)
```

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

```
mapIntW : ∀γ.(Int→ γ → γ)→ γ → (Int→Int)→[Int]→ γ
        = λγ.λv^(:):Int→ γ → γ.λv^[]:γ.λf:Int→Int.
             foldr Int γ (λu:Int.λw:γ. v^(:) (f u) w) v^[]

mapInt : (Int→Int)→[Int]→[Int]
       = mapW [Int] ((:) Int) ([] Int)
```

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

```
map : ∀α.∀β.(α → β)→[α]→[β]
    = λα.λβ.λf:α → β.
         foldr α [β] (λu:α.λw:[β].(:) β (f u) w) ([] β)
```

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

```
mapW : ∀α.∀β.∀γ.(β → γ → γ)→ γ → (α → β)→[α]→ γ
     = λα.λβ.λγ.λv^(:):β → γ → γ.λv^[]:γ.λf:α → β.
          foldr α γ (λu:α.λw:γ. v^(:) (f u) w) v^[]

map : ∀α.∀β.(α → β)→[α]→[β]
    = λα.λβ.mapW α β [β] ((:) β) ([] β)
```

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

```
   or (map τ Bool p xs)
⇝ {inlining of or and map}
   foldr Bool Bool (||) False
     (mapW τ Bool [Bool] ((:)  Bool) ([] Bool) p xs)
⇝ {list abstraction of the producer}
   foldr Bool Bool (||) False
     ((λγ.λv^(:):β→γ→γ.λv^[]:γ.mapW τ Bool γ v^(:) v^[] p xs)
     [Bool] ((:) Bool) ([] Bool))
⇝ {fusion and subsequent β-reduction}
   mapW τ Bool Bool (||) False p xs
```

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

Note that in the definition of the worker `mapW` we insert the new λ-abstractions between the type abstractions and the term abstractions. We cannot insert the new term abstractions in front of the original type abstractions, because the list type `[`β`]`, from which we abstract, contains the type variable β which is bound in the type of the function. To insert the new abstractions in front of the original term abstractions has a minor advantage: The wrapper can be inlined and β-reduced even at call sites where it is only partially applied, because its definition partially applies the worker.

After we have discussed the list abstraction algorithm in detail in Chapter 4, we will discuss in Chapter 5 how a definition of a function that produces a list can be split fully automatically into a definition of a worker and a wrapper. In the remainder of this chapter we give examples of the expressive power of the worker/wrapper scheme and how it enables deforestation.

3.3.1 Functions that Produce Several Lists

A worker even can abstract from several lists. For example the definition of the function `unzip`, which produces two lists, can be split into the following worker and wrapper definitions:

```
unzipW : $\forall\alpha.\forall\beta.\forall\gamma.\forall\delta.(\alpha\to\gamma\to\gamma)\to\gamma\to(\beta\to\delta\to\delta)\to\delta\to$ [($\alpha$,$\beta$)] $\to$ ($\gamma$,$\delta$)
          = $\lambda\alpha.\,\lambda\beta.\,\lambda\gamma.\,\lambda\delta.\,\lambda v_1^{(:)}:\alpha\to\gamma\to\gamma.\,\lambda v_1^{[]}:\gamma.\,\lambda v_2^{(:)}:\beta\to\delta\to\delta.\,\lambda v_2^{[]}:\delta.$
              foldr ($\alpha$,$\beta$) ($\gamma$,$\delta$)
                ($\lambda$x:($\alpha$,$\beta$).$\lambda$y:($\gamma$,$\delta$).case x of
                  (u,w) $\mapsto$ case y of
                    (us,ws) $\mapsto$ (,) $\gamma$ $\delta$ ($v_1^{(:)}$ u us) ($v_2^{(:)}$ w ws) )
                ((,) $\gamma$ $\delta$ $v_1^{[]}$ $v_2^{[]}$)

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

The subsequent transformations demonstrate how the wrapper enables deforestation without requiring inlining of the larger worker:

```
   foldr $\tau_1$ $\tau_3$ $M^{(:)}$ $M^{[]}$  (fst [$\tau_1$] [$\tau_2$] (unzip $\tau_1$ $\tau_2$ zs))
$\rightsquigarrow$ {inlining of the wrapper unzip}
   foldr $\tau_1$ $\tau_3$ $M^{(:)}$ $M^{[]}$
     (fst [$\tau_1$] [$\tau_2$]
       (unzipW $\tau_1$ $\tau_2$ [$\tau_1$] [$\tau_2$] ((:) $\tau_1$) ([] $\tau_1$) ((:) $\tau_2$) ([] $\tau_2$) zs))
$\rightsquigarrow$ {list abstraction of the producer}
   foldr $\tau_1$ $\tau_3$ $M^{(:)}$ $M^{[]}$
     (($\lambda\gamma.\,\lambda v^{(:)}:\tau_1\to\gamma\to\gamma.\,\lambda v^{[]}:\gamma.$fst $\gamma$ [$\tau_2$]
       (unzipW $\tau_1$ $\tau_2$ $\gamma$ [$\tau_2$] $v^{(:)}$ $v^{[]}$ ((:) $\tau_2$) ([] $\tau_2$) zs))
     [$\tau_1$] ((:) $\tau_1$) ([] $\tau_1$))
$\rightsquigarrow$ {fusion and subsequent $\beta$-reduction}
   fst $\tau_3$ [$\tau_2$] (unzipW $\tau_1$ $\tau_2$ $\tau_3$ [$\tau_2$] $M^{(:)}$ $M^{[]}$ ((:) $\tau_2$) ([] $\tau_2$) zs)
```

Original short cut deforestation cannot handle producers of several lists and hence cannot fuse the producer `fst [Bool] [Char] (unzip Bool Char zs)` with any consumer (see Section 7.1.2).

We still cannot fuse if both result lists of `unzip` are consumed, but see Section 6.2 for an extension that can also handle that case.

3.3.2 List Concatenation

The list append function `(++)` is notorious for being difficult to fuse with (cf. Chapter 7), because the term `(++)` τ `xs ys` does not produce the whole result list itself. Only `xs` is copied but not `ys`. However, we can easily define a worker for `(++)` by abstracting not just the result list but simultaneously the type of the second argument:

```
appW : ∀α.∀γ.(α→γ→γ) → γ → [α] → γ → γ
     = λα. λγ. λv^(:):α→γ→γ. λv^[]:γ. λxs:[α]. λys:γ. foldr α γ v^(:) ys xs

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

Note that the variable v$^{[]}$, which abstracts the empty list, is not used in the body of the worker `appW`. Hence we do not need it and we can define worker and wrapper simply as:

```
appW : ∀α.∀γ.(α→γ→γ) → [α] → γ → γ
     = λα. λγ. λv^(:):α→γ→γ. λxs:[α]. λys:γ. foldr α γ v^(:) ys xs

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

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

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

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

```
concatW : ∀α.∀γ.(α→γ→γ) → γ → [[α]] → γ
        = λα. λγ. λv^(:):α→γ→γ. λv^[]:γ.foldr [α] γ (appW α γ v^(:)) v^[]

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

3.3.3 Recursively Defined Producers

In all previous examples recursion was hidden by `foldr`. Nonetheless recursively defined producers can be split into workers and wrappers just as easily. Consider the recursively defined function `tails` which returns the list of all final segments of xs, longest first:

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

It can be split into definitions of a worker `tailsW` and a wrapper `tails`:

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

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

Note that to abstract the list the recursive call in the definition of the worker must be to the worker itself, not to the wrapper. It is possible to avoid passing α, γ, `c` and `n` in the recursive call, as we will discuss in Section 5.1.2, but for the following examples that optimisation is undesirable.

3.4 Functions that Consume their Own Result

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

```
reverse : ∀α.[α] → [α]
        = λα. λxs:[α]. case xs of
             []    ↦ [] α
             y:ys  ↦ (++) α (reverse α ys) ((:) α y ([] α))
```

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

```
reverseW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
         = λα. λγ. λ$v^{(:)}$:α → γ → γ. λ$v^{[]}$:γ. λxs:[α]. case xs of
              []    ↦ $v^{[]}$
              y:ys  ↦ appW α γ $v^{(:)}$
                          (reverseW α [α] ((:) α) ([] α) ys) ($v^{(:)}$ y $v^{[]}$)

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

In this definition of `reverseW` the worker `appW` can be inlined and β-reduced, giving

```
reverseW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
         = λα. λγ. λ$v^{(:)}$:α → γ → γ. λ$v^{[]}$:γ. λxs:[α]. case xs of
              []    ↦ $v^{[]}$
              y:ys  ↦ $\underline{\texttt{foldr}}$ α γ $v^{(:)}$ ($v^{(:)}$ y $v^{[]}$)
                          (reverseW α [α] ((:) α) ([] α) ys)
```

So we have a `foldr` consumer of the producer `(reverseW` α `[`α`] ((:)` α`) ([]` α`) ys)`. We apply list abstraction to the producer to obtain

```
reverseW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
         = λα. λγ. λ$v^{(:)}$:α → γ → γ. λ$v^{[]}$:γ. λxs:[α]. case xs of
              []    ↦ $v^{[]}$
              y:ys  ↦ $\underline{\texttt{foldr}}$ α γ $v^{(:)}$ ($v^{(:)}$ y $v^{[]}$)
                          ((λδ. λ$v_2^{(:)}$ : α→δ→δ. λ$v_2^{[]}$ : δ. reverseW α δ $v_2^{(:)}$ $v_2^{[]}$ ys)
                            [α] ((:) α) ([] α))
```

Finally short cut fusion and subsequent β-reduction yields

```
$\texttt{reverseW} : \forall\alpha.\forall\gamma.(\alpha \to \gamma \to \gamma) \to \gamma \to \texttt{[}\alpha\texttt{]} \to \gamma$
           $= \lambda\alpha.\, \lambda\gamma.\, \lambda\texttt{v}^{\texttt{(:)}}\!:\!\alpha \to \gamma \to \gamma.\, \lambda\texttt{v}^{\texttt{[]}}\!:\!\gamma.\, \lambda\texttt{xs}\!:\!\texttt{[}\alpha\texttt{]}\,.\ \texttt{case xs of}$
             $\texttt{[]} \quad\ \mapsto \texttt{v}^{\texttt{[]}}$
             $\texttt{y:ys} \mapsto \texttt{reverseW}\ \alpha\ \gamma\ \texttt{v}^{\texttt{(:)}}\ (\texttt{v}^{\texttt{(:)}}\ \texttt{y}\ \texttt{v}^{\texttt{[]}})\ \texttt{ys}$
```

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

The type inference based method of inlining definitions which we discussed in Section 3.2 cannot achieve this transformation of the quadratic version into the linear version. To abstract the intermediate list, that algorithm would need to inline the definition of `reverse`. Then the intermediate list would be eliminated successfully, but the inlined definition of `reverse` would contain a new starting point for deforestation which would lead to new inlining of `reverse` ... The quadratic version produces at runtime an intermediate list between each recursive call. To remove all these intermediate lists through a finite amount of transformation a worker/wrapper scheme is required.

Gill noted this fact first in his thesis [Gil96], giving a similar example: a function which traverses a tree to collect all node entries in a list. A straightforward quadratic time definition can be split into a polymorphically recursive worker and a wrapper and then be deforested to obtain a linear time definition which uses an accumulating argument.

3.4.1 A larger example: `inits`

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

```
$\texttt{inits:}\ \forall\alpha.\texttt{[}\alpha\texttt{]} \to \texttt{[[}\alpha\texttt{]]}$
      $= \lambda\alpha.\ \lambda\texttt{xs:[}\alpha\texttt{]}\,.$
          $\texttt{case xs of}$
            $\texttt{[]} \quad\ \mapsto \texttt{(:) [}\alpha\texttt{] ([] }\alpha\texttt{) ([] [}\alpha\texttt{])}$
            $\texttt{y:ys} \mapsto \texttt{(:) [}\alpha\texttt{] ([] }\alpha\texttt{) (map [}\alpha\texttt{] [}\alpha\texttt{] ((:) }\alpha\texttt{ y)}$
                                      $\texttt{(inits }\alpha\texttt{ ys))}$
```

The definition is split into the following worker and wrapper definitions:

```
$\texttt{initsW:}\ \forall\alpha.\forall\gamma.\forall\delta.(\alpha{\to}\gamma{\to}\gamma) \to \gamma \to (\gamma{\to}\delta{\to}\delta) \to \delta \to \texttt{[}\alpha\texttt{]} \to \delta$
       $= \lambda\alpha.\, \lambda\gamma.\, \lambda\delta.\, \lambda\texttt{v}_1^{\texttt{(:)}}\!:\!\alpha{\to}\gamma{\to}\gamma\,.\ \lambda\texttt{v}_1^{\texttt{[]}}\!:\!\gamma\,.\ \lambda\texttt{v}_2^{\texttt{(:)}}\!:\!\gamma{\to}\delta{\to}\delta\,.\ \lambda\texttt{v}_2^{\texttt{[]}}\!:\!\delta\,.\ \lambda\texttt{xs:[}\alpha\texttt{]}\,.$
         $\texttt{case xs of}$
           $\texttt{[]} \quad\ \mapsto \texttt{v}_2^{\texttt{(:)}}\ \texttt{v}_1^{\texttt{[]}}\ \texttt{v}_2^{\texttt{[]}}$
           $\texttt{y:ys} \mapsto \texttt{v}_2^{\texttt{(:)}}\ \texttt{v}_1^{\texttt{[]}}\ \texttt{(mapW}\ \gamma\ \gamma\ \delta\ \texttt{v}_2^{\texttt{(:)}}\ \texttt{v}_2^{\texttt{[]}}\ \texttt{(v}_1^{\texttt{(:)}}\ \texttt{y)}$
                    $\texttt{(initsW}\ \alpha\ \gamma\ \texttt{[}\gamma\texttt{]}\ \texttt{v}_1^{\texttt{(:)}}\ \texttt{v}_1^{\texttt{[]}}\ \texttt{((:)}\,\gamma\texttt{)}\ \texttt{([]}\,\gamma\texttt{)}\ \texttt{ys))}$

$\texttt{inits:}\ \forall\alpha.\texttt{[}\alpha\texttt{]} \to \texttt{[[}\alpha\texttt{]]}$
      $= \lambda\alpha.\ \texttt{initsW}\ \alpha\ \texttt{[}\alpha\texttt{]}\ \texttt{[[}\alpha\texttt{]]}\ \texttt{((:)}\,\alpha\texttt{)}\ \texttt{([]}\,\alpha\texttt{)}\ \texttt{((:)}\,\texttt{[}\alpha\texttt{])}\ \texttt{([]}\,\texttt{[}\alpha\texttt{])}$
```

Note that both (nested) result lists are abstracted. In this definition of `initsW` the worker `mapW` can be inlined and β-reduced, giving

```
$\texttt{initsW:}\ \forall\alpha.\forall\gamma.\forall\delta.(\alpha{\to}\gamma{\to}\gamma) \to \gamma \to (\gamma{\to}\delta{\to}\delta) \to \delta \to \texttt{[}\alpha\texttt{]} \to \delta$
       $= \lambda\alpha.\, \lambda\gamma.\, \lambda\delta.\, \lambda\texttt{v}_1^{\texttt{(:)}}\!:\!\alpha{\to}\gamma{\to}\gamma\,.\ \lambda\texttt{v}_1^{\texttt{[]}}\!:\!\gamma\,.\ \lambda\texttt{v}_2^{\texttt{(:)}}\!:\!\gamma{\to}\delta{\to}\delta\,.\ \lambda\texttt{v}_2^{\texttt{[]}}\!:\!\delta\,.\ \lambda\texttt{xs:[}\alpha\texttt{]}\,.$
         $\texttt{case xs of}$
```

```
    []    ↦ $v_2^{(:)}$ $v_1^{[]}$ $v_2^{[]}$
    y:ys  ↦ $v_2^{(:)}$ $v_1^{[]}$ ($\underline{\texttt{foldr}}$ $\gamma$ $\delta$ ($\lambda$u:$\gamma$. $\lambda$w:$\delta$. $v_2^{(:)}$ ($v_1^{(:)}$ y u) w) $v_2^{[]}$
                (initsW $\alpha$ $\gamma$ [$\gamma$] $v_1^{(:)}$ $v_1^{[]}$ ((:) $\gamma$) ([] $\gamma$) ys))
```

So we have a `foldr` consumer of the producer (`initsW` ...). We apply list abstraction to the producer to obtain

```
initsW: $\forall\alpha.\forall\gamma.\forall\delta.(\alpha\to\gamma\to\gamma) \to \gamma \to (\gamma\to\delta\to\delta) \to \delta \to [\alpha] \to \delta$
     = $\lambda\alpha.\ \lambda\gamma.\ \lambda\delta.\ \lambda v_1^{(:)}:\alpha\to\gamma\to\gamma.\ \lambda v_1^{[]}:\gamma.\ \lambda v_2^{(:)}:\gamma\to\delta\to\delta.\ \lambda v_2^{[]}:\delta.\ \lambda$xs:[$\alpha$].
         case xs of
           []    ↦ $v_2^{(:)}$ $v_1^{[]}$ $v_2^{[]}$
           y:ys  ↦ $v_2^{(:)}$ $v_1^{[]}$ ($\underline{\texttt{foldr}}$ $\gamma$ $\delta$ ($\lambda$u:$\gamma$. $\lambda$w:$\delta$. $v_2^{(:)}$ ($v_1^{(:)}$ y u) w) $v_2^{[]}$
                       (($\lambda\delta'.\ \lambda v_3^{(:)} : \gamma\to\delta'\to\delta'.\ \lambda v_3^{[]} : \delta'$.
                          initsW $\alpha$ $\gamma$ $\delta'$ $v_1^{(:)}$ $v_1^{[]}$ $v_3^{(:)}$ $v_3^{[]}$ ys)
                        [$\gamma$] ((:) $\gamma$) ([] $\gamma$))
```

Finally short cut fusion and subsequent β-reduction yields

```
initsW: $\forall\alpha.\forall\gamma.\forall\delta.(\alpha\to\gamma\to\gamma) \to \gamma \to (\gamma\to\delta\to\delta) \to \delta \to [\alpha] \to \delta$
     = $\lambda\alpha.\ \lambda\gamma.\ \lambda\delta.\ \lambda v_1^{(:)}:\alpha\to\gamma\to\gamma.\ \lambda v_1^{[]}:\gamma.\ \lambda v_2^{(:)}:\gamma\to\delta\to\delta.\ \lambda v_2^{[]}:\delta.\ \lambda$xs:[$\alpha$].
         case xs of
           []    ↦ $v_2^{(:)}$ $v_1^{[]}$ $v_2^{[]}$
           y:ys  ↦ $v_2^{(:)}$ $v_1^{[]}$ (initsW $\alpha$ $\gamma$ $\delta$ $v_1^{(:)}$ $v_1^{[]}$
                              ($\lambda$u:$\gamma$. $\lambda$w:$\delta$. $v_2^{(:)}$ ($v_1^{(:)}$ y u) w) $v_2^{[]}$ ys)
```

The definition of the n-queens function which we consider in Section 5.3 is another example in the same spirit.

3.4.2 Deforestation Changes Complexity

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

A recursively defined function that consumes its own result with `foldr` will nearly always enable deforestation that changes the asymptotic time complexity of the definition. This power is, however, a double-edged sword. A small syntactic change of a program (cf. next section) may cause deforestation to be no longer applicable, and thus change the asymptotic complexity of the program. It may hence be argued that such far-reaching modifications should be left to the programmer.

3.4.3 Inaccessible Recursive Arguments

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

```
reverse: $\forall\alpha.[\alpha] \to [\alpha]$
      = $\lambda\alpha$. foldr $\alpha$ [$\alpha$]
          ($\lambda$y:$\alpha$.$\lambda$r:[$\alpha$]. (++) $\alpha$ r ((:) $\alpha$ y ([] $\alpha$))) ([] $\alpha$)
```

It is impossible to abstract the result list from this definition. No matter which list constructors and list types we abstract, the resulting term is never well-typed. Our list abstraction algorithm, which abstracts as many lists as possible, just returns the unchanged definition body. The cause of the problem is that the recursion argument `r` must be a list, because it is consumed by `(++)`.

To enable list abstraction we have to rewrite the definition, turning `r` into a function with a list type and its constructors as arguments:

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

From this definition our list abstraction algorithm can abstract the result list and thus split the definition into a worker and a wrapper. It is, however, unclear when and how such restructuring to lift a list type to a function type can be done in general. Our deforestation algorithm only tries to abstract list types and list constructors. It does not perform any further restructuring of a definition. Hence it cannot deforest the first `foldr` definition of `reverse`.

Chapter 4

List Abstraction And Type Inference

In the preceding chapter we informally introduced the idea of list abstraction through type inference. List abstraction prepares a list-producing term for fusion with a `foldr` consumer by the short cut fusion rule. Furthermore, we saw in the preceeding chapter how useful it is to split the definition of a list-producing function into a definition of a worker and a definition of a wrapper. We will see in Chapter 5 how easily we obtain these two definitions from the list abstracted form of the original definition body. The only necessary extension is that for the worker/wrapper split we have to abstract not only one but as many lists as possible from a term. Here we describe this abstraction in detail.

The list abstraction algorithm processes a producer in three phases. In the first phase all list types are replaced by type variables and all list data constructors are replaced by term variables, to which we will also refer as constructor variables. The result of the first phase is the input to the second phase, the type inference phase. Type inference replaces some of the type variables so that the typing is again derivable from the type inference rules, that is, the term is well-typed in the typing environment. Type inference cannot fail, because the original producer is well-typed. In the third phase some type variables are reinstantiated to list types, some constructor variables are reinstantiated to list data constructors and the remaining type and constructor variables are being abstracted.

4.1 The Instance Substitution

The central data structure of the list abstraction algorithm is a substitution η. It memorises for every new type variable the list type for which it was substituted in the first phase. For example, for a term $\lambda\texttt{x:[Int].\ case\ x\ of}\ \{\texttt{[]}\mapsto\texttt{x}\}$ the first phase will return the modified term $\lambda\texttt{x:}\gamma_1\texttt{.\ case\ x\ of}\ \{\texttt{[]}\mapsto\texttt{x}\}$ and the substitution $\eta = [\texttt{[Int]}/\gamma_1]$. We call this substitution η instance substitution. The importance of having the information that the instance substitution contains will become clear in the course of this chapter.

As just defined the instance substitution is not uniquely determined by a term. For example, for the term $\lambda\texttt{x:[[Int]].\ case\ x\ of}\ \{\texttt{[]}\mapsto\texttt{x}\}$ the list replacement phase will return the modified term $\lambda\texttt{x:}\gamma\texttt{.\ case\ x\ of}\ \{\texttt{[]}\mapsto\texttt{x}\}$. A fitting instance substitution would be $\eta_1 = [\texttt{[[Int]]}/\gamma]$ but also the substitution $\eta_2 = [\texttt{[}\gamma'\texttt{]}/\gamma, \texttt{[Int]}/\gamma']$. Note that applying η_2 to the modified term only yields $\lambda\texttt{x:[}\gamma'\texttt{].\ case\ x\ of}\ \{\texttt{[]}\mapsto\texttt{x}\}$. However, applying

η_2 to this term then yields again the input term λ`x:[[Int]].case x of {[]`$\mapsto$`x}`. For the list abstraction algorithm η_2 is preferable to η_1. The substitution η_2 describes list replacement in more detail and may lead to the abstraction of more lists. So the element types of the list types of an instance substitution should not contain any list types.

The previous example also demonstrates that for an instance substitution η the sets $\mathrm{dom}(\eta)$ and $\mathrm{free}(\eta)$ may overlap. However, η is always acyclic.

Definition 4.1 A substitution σ is **acyclic**, if there exists no set $\{\gamma_1, \ldots, \gamma_k\} \subseteq \mathrm{dom}(\sigma)$ with $\gamma_2 \in \mathrm{free}(\sigma(\gamma_1)), \ldots, \gamma_k \in \mathrm{free}(\sigma(\gamma_{k-1})), \gamma_1 \in \mathrm{free}(\sigma(\gamma_k))$. Especially, there exists no $\gamma \in \mathrm{dom}(\sigma)$ with $\gamma \in \mathrm{free}(\sigma(\gamma))$. □

Definition 4.2 A type substitution η is an **instance substitution**, if $\eta(\gamma) = \texttt{[}\tau_\gamma\texttt{]}$ for all $\gamma \in \mathrm{dom}(\eta)$ and some types τ_γ, and η is acyclic. □

In the remainder of this chapter the meta-variable η always denotes an instance substitution. As demonstrated by the last example we have to apply η several times, that is, we have to apply its compositional closure:

Definition 4.3 The **compositional closure of an acyclic substitution** σ is the substitution σ^∞ given by

$$\sigma^\infty(\gamma) := (\sigma(\gamma))\sigma^\infty$$

for all $\gamma \in \mathrm{dom}(\sigma^\infty) := \mathrm{dom}(\sigma)$. □

For example, $[\texttt{[}\gamma'\texttt{]}/\gamma, \texttt{[Int]}/\gamma']^\infty = [\texttt{[[Int]]}/\gamma, \texttt{[Int]}/\gamma']$. Because σ is acyclic, its compositional closure is well-defined. In fact, for every acyclic σ there exists a natural number k with $\sigma^\infty = \underbrace{\sigma\sigma\ldots\sigma}_{k}$.

The Typing Environment for Constructor Variables

Besides replacing list types by type variables and constructing an instance substitution the first phase also replaces every list constructor application `(:)` τ and `[]` τ for any type τ by a new term variable, called **constructor variable**. It proves to be technically convenient not to explicitly add these constructor variables to the typing environment. Instead it suffices to demand that every constructor variable is associated with a different type variable. To describe this association we index constructor variables by type variables: For any type variable γ the meta-variable $v_\gamma^{\texttt{(:)}}$ denotes a constructor variable that replaces an application of the constructor `(:)` and $v_\gamma^{\texttt{[]}}$ denotes a constructor variable that replaces an application of the constructor `[]`. The typing environment for the constructor variables is determined by an instance substitution η:

Definition 4.4 The **typing environment for constructor variables** determined by an instance substitution η, written $\mathrm{tyEnv}(\eta)$, is defined by

$$\mathrm{tyEnv}(\eta) := \{v_\gamma^{\texttt{(:)}}{:}\tau_\gamma \to \gamma \to \gamma, v_\gamma^{\texttt{[]}}{:}\gamma \mid \eta(\gamma) = \texttt{[}\tau_\gamma\texttt{]}, \gamma \in \mathrm{dom}(\eta)\}$$ □

For example,

$$\mathrm{tyEnv}([\texttt{[}\gamma'\texttt{]}/\gamma, \texttt{[Int]}/\gamma']) = \{v_\gamma^{\texttt{(:)}}{:}\gamma' \to \gamma \to \gamma,\ v_\gamma^{\texttt{[]}}{:}\gamma,\ v_{\gamma'}^{\texttt{(:)}}{:}\texttt{Int} \to \gamma' \to \gamma',\ v_{\gamma'}^{\texttt{[]}}{:}\gamma'\}$$

4.2 List Replacement Phase

In the first phase of list abstraction every list type is replaced by a new type variable, every list constructor application `(:)` τ is replaced by a new constructor variable $v^{\texttt{(:)}}_{\gamma}$, every list constructor application `[]` τ is replaced by a new constructor variable $v^{\texttt{[]}}_{\gamma}$ and an instance substitution, which records all these replacements, is constructed.

4.2.1 Locally Bound Type Variables

To be precise, not every list type and list constructor is replaced. Consider the input term $\lambda\alpha.\,\lambda\texttt{xs}\texttt{:}\texttt{[}\alpha\texttt{]}\,.\,\texttt{xs}$. It would be incorrect of the list replacement algorithm to replace the list type $\texttt{[}\alpha\texttt{]}$ by a type variable γ, that is, return an instance substitution $\eta = [\texttt{[}\alpha\texttt{]}/\gamma]$ and a term $\lambda\alpha.\,\lambda\texttt{xs}\texttt{:}\gamma.\,\texttt{xs}$. The problem is that we do not regain the original input when we apply η^{∞}. We have

$$(\lambda\alpha.\,\lambda\texttt{xs}\texttt{:}\gamma.\,\texttt{xs})\eta^{\infty} = \lambda\beta.\,\lambda\texttt{xs}\texttt{:}\texttt{[}\alpha\texttt{]}\,.\,\texttt{xs}$$

for a new type variable β, because α is free in $\eta^{\infty}(\gamma) = \texttt{[}\alpha\texttt{]}$ and hence the λ-bound type variable α is renamed. So the cause of this problem is that the list type $\texttt{[}\alpha\texttt{]}$ contains a type variable that is bound within the term. Such list types and similarly list constructors of such list types cannot be abstracted and hence are not replaced by the list replacement algorithm.

Note that list abstraction is not hindered by type variables that are bound outside the term that shall be transformed by list abstraction. So for the input term $\lambda\texttt{xs}\texttt{:}\texttt{[}\alpha\texttt{]}\,.\,\texttt{xs}$ the list replacement algorithm returns the instance substitution $\eta = [\texttt{[}\alpha\texttt{]}/\gamma]$ and the term $\lambda\texttt{xs}\texttt{:}\gamma.\,\texttt{xs}$.

4.2.2 The List Replacement Algorithm

The list replacement algorithm $\mathcal{R}$ is defined in Figure 4.1. Its method of working is straightforward. It recursively traverses the whole input term including its embedded types. It introduces a new type variable for every list type that it replaces. We assume that the set of these new type variables is disjoint from the set of all (free or bound) type variables of the complete term that is list abstracted. Similarly we assume that for every type variable γ the constructor variable $v^{\texttt{(:)}}_{\gamma}$ and the constructor variable $v^{\texttt{[]}}_{\gamma}$ is a new term variable and that the set of these constructor variables is disjoint from the set of all term variables that occur in the complete term that is list abstracted.

The purpose of the first argument of $\mathcal{R}$, a set of variables, is to collect all locally bound type variables during the recursive descent of the algorithm. So the list replacement algorithm starts with an empty set as first argument. Using this set the algorithm can easily check for each list type that it comes across if it is correct to replace it; similarly the set is used in the replacement of list constructors.

Note that the algorithm $\mathcal{R}$, when coming across a list type $\texttt{[}\tau\texttt{]}$, first recursively transforms τ which yields a type τ' and then replaces the list type $\texttt{[}\tau'\texttt{]}$, if possible. The same order is chosen in the case of list constructor applications. This order assures for example, that for a nested type `[[Int]]` the algorithm yields $([\texttt{[Int]}/\gamma_1, \texttt{[}\gamma_1\texttt{]}/\gamma_2], \gamma_2)$ instead of the more restrictive $([\texttt{[[Int]]}/\gamma_1], \gamma_1)$. Together with the fact that every list type and constructor is replaced by a new variable this assures that the replacement algorithm returns the most general instance substitution that is possible. In the three replacement cases the order in which the instance substitution is constructed is important, whereas in all other cases it is irrelevant, because the domains and sets of free type variables of the respective partial instance substitutions are disjoint.

$$\mathcal{R}(V, \texttt{(:)}\ \tau) := \begin{cases} (\eta[\texttt{[}\tau'\texttt{]}/\gamma], v_\gamma^{\texttt{(:)}}) & \text{, if } \mathrm{free}(\tau') \cap V = \emptyset \\ (\eta, \texttt{(:)}\ \tau') & \text{, otherwise} \end{cases}$$

$$\text{where } (\eta, \tau') := \mathcal{R}(V, \tau)$$
$$\gamma \text{ a new type variable}$$

$$\mathcal{R}(V, \texttt{[]}\ \tau) := \begin{cases} (\eta[\texttt{[}\tau'\texttt{]}/\gamma], v_\gamma^{\texttt{[]}}) & \text{, if } \mathrm{free}(\tau') \cap V = \emptyset \\ (\eta, \texttt{[]}\ \tau') & \text{, otherwise} \end{cases}$$

$$\text{where } (\eta, \tau') := \mathcal{R}(V, \tau)$$
$$\gamma \text{ a new type variable}$$

$$\mathcal{R}(V, x) := ([\,], x)$$

$$\mathcal{R}(V, \lambda x : \tau.M) := (\eta_\tau \eta_M, \lambda x : \tau'.M')$$
$$\text{where } (\eta_\tau, \tau') := \mathcal{R}(V, \tau)$$
$$(\eta_M, M') := \mathcal{R}(V, M)$$

$$\mathcal{R}(V, M_1\, M_2) := (\eta_1 \eta_2, M_1'\, M_2') \quad \text{where } (\eta_i, M_i') := \mathcal{R}(V, M_i)$$

$$\mathcal{R}(V, \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^k\ \texttt{in}\ N) := (\eta \eta_1 \ldots \eta_k \eta_1' \ldots \eta_k', \texttt{let}\ \{x_i : \tau_i' = M_i'\}_{i=1}^k\ \texttt{in}\ N')$$
$$\text{where } (\eta, N') := \mathcal{R}(V, N)$$
$$(\eta_i, \tau_i') := \mathcal{R}(V, \tau_i)$$
$$(\eta_i', M_i') := \mathcal{R}(V, M_i)$$

$$\mathcal{R}(V, c) := ([\,], c)$$

$$\mathcal{R}(V, \texttt{case}\ M\ \texttt{of}\ \{c_i\, \overline{x}_i \to N_i\}_{i=1}^k) := (\eta \eta_1 \ldots \eta_k, \texttt{case}\ M'\ \texttt{of}\ \{c_i\, \overline{x}_i \to N_i'\}_{i=1}^k)$$
$$\text{where } (\eta, M') := \mathcal{R}(V, M)$$
$$(\eta_i, N_i') := \mathcal{R}(V, N_i)$$

$$\mathcal{R}(V, \lambda\alpha.M) := (\eta, \lambda\alpha.M') \quad \text{where } (\eta, M') := \mathcal{R}(V \cup \{\alpha\}, M)$$

$$\mathcal{R}(V, M\, \tau) := (\eta_M \eta_\tau, M'\, \tau') \quad \text{,if } M \neq \texttt{(:)} \text{ and } M \neq \texttt{[]}$$
$$\text{where } (\eta_M, M') := \mathcal{R}(V, M)$$
$$(\eta_\tau, \tau') := \mathcal{R}(V, \tau)$$

$$\mathcal{R}(V, \texttt{[}\tau\texttt{]}) := \begin{cases} (\eta[\texttt{[}\tau'\texttt{]}/\gamma], \gamma) & \text{, if } \mathrm{free}(\tau') \cap V = \emptyset \\ (\eta, \texttt{[}\tau'\texttt{]}) & \text{, otherwise} \end{cases}$$

$$\text{where } (\eta, \tau') := \mathcal{R}(V, \tau)$$
$$\gamma \text{ a new type variable}$$

$$\mathcal{R}(V, T\, \tau_1 \ldots \tau_n) := (\eta_1 \ldots \eta_n, T\, \tau_1' \ldots \tau_n') \quad \text{,if } T \neq \texttt{[]}$$
$$\text{where } (\eta_i, \tau_i') := \mathcal{R}(V, \tau_i)$$

$$\mathcal{R}(V, \tau_1 \to \tau_2) := (\eta_1 \eta_2, \tau_1' \to \tau_2') \quad \text{where } (\eta_i, \tau_i') := \mathcal{R}(V, \tau_i)$$

$$\mathcal{R}(V, \forall\alpha.\tau) := (\eta, \forall\alpha.\tau') \quad \text{where } (\eta, \tau') := \mathcal{R}(V \cup \{\alpha\}, \tau)$$

$$\mathcal{R}(V, \alpha) := ([\,], \alpha)$$

Figure 4.1: List replacement algorithm

4.2.3 Properties of the List Replacement Algorithm

We have to prove for the complete list abstraction algorithm that it terminates, that its result is well-typed and that its result is contextually equivalent to its input. These proofs are simple when we can refer to respective proofs for all algorithms that form the list abstraction algorithm. Naturally every component algorithm also fulfils some unique property. For the list replacement algorithm we have to show that it really constructs an instance substitution.

Lemma 4.5

$$\frac{(N, \eta) = \mathcal{R}(V, M)}{\eta \textit{ is an instance substitution}} \qquad \square$$

Proof Obviously η substitutes lists for variables. It can be proved by induction on M that η is acyclic. The general idea in all induction steps is that the new η equals $\eta_1\eta_2$ where η_1 and η_2 are acyclic and $\mathrm{dom}(\eta_2) \cap (\mathrm{dom}(\eta_1) \cup \mathrm{free}(\eta_1)) = \emptyset$. Hence η is acyclic. ■

The following lemma states that the first argument of $\mathcal{R}$, the set of variables, fulfils its purpose.

Lemma 4.6 $\mathcal{R}(V, M)$ *does not replace types that contain variables from the set* V*:*

$$\frac{(\eta, N) = \mathcal{R}(V, M)}{\mathrm{free}(\eta) \cap V = \emptyset} \qquad \square$$

Proof Straightforward by structural induction on the term M. ■

Also the rather obvious property of termination has to be stated.

Lemma 4.7 (Termination) $\mathcal{R}(V, M)$ *terminates for every set of type variables* V *and every term* M*.* □

Proof Trivial, because $\mathcal{R}(V, M)$ is defined inductively over the term M. ■

Reinstantiation

It must be possible to use the instance substitution to regain the input of the list replacement algorithm $\mathcal{R}$. We split the reinstantiation property into two lemmas.

Lemma 4.8 (Reinstantiation of types)

$$\frac{(\eta, \rho') = \mathcal{R}(V, \rho)}{\rho'\eta^{\infty} = \rho} \qquad \square$$

Proof Structural induction on the type ρ analogously to the proof of Lemma 4.11. ■

In terms also the constructors have to be reinstantiated. The instance substitution η determines how they are reinstantiated:

Definition 4.9 The **constructor instantiation substitution** determined by a substitution η, written instCons(η), is defined by

$$\mathrm{instCons}(\eta) := [\texttt{(:)}\ \tau/v_\gamma^{\texttt{(:)}},\ \texttt{[]}\ \tau/v_\gamma^{\texttt{[]}} \mid \eta(\gamma) = \texttt{[}\tau\texttt{]}, \gamma \in \mathrm{dom}(\eta)] \qquad \square$$

Because terms also contain types we combine the two substitutions.

Definition 4.10 The **instantiation substitution** determined by a substitution η, written $\mathrm{inst}(\eta)$, is defined by

$$\mathrm{inst}(\eta) := \eta\,\mathrm{instCons}(\eta)$$

□

Lemma 4.11 (Reinstantiation of terms)

$$\frac{(\eta, N) = \mathcal{R}(V, M)}{N\,\mathrm{inst}(\eta^{\infty}) = M}$$

□

Proof Structural induction on the term M. We only prove three cases here. The proofs for the other cases are similar.

Case $\texttt{(:)}\ \tau$.

Let $(\eta, \tau') := \mathcal{R}(V, \tau)$ and γ be a new type variable.

Case $\mathrm{free}(\tau') \cap V = \emptyset$.

$$\begin{aligned}
& v_{\gamma}^{\texttt{(:)}}\,\mathrm{inst}((\eta[\texttt{[}\tau'\texttt{]}/\gamma])^{\infty}) \\
= \quad & \{\gamma \notin \mathrm{dom}(\eta) \cup \mathrm{freeTy}(\eta), \text{ because } \gamma \text{ is new}\} \\
& v_{\gamma}^{\texttt{(:)}}\,\mathrm{inst}(\eta^{\infty}[\texttt{[}\tau'\eta^{\infty}\texttt{]}/\gamma]) \\
= \quad & \{\eta^{\infty}[\texttt{[}\tau'\eta^{\infty}\texttt{]}/\gamma](\gamma) = \texttt{[}\tau'\eta^{\infty}\texttt{]}\} \\
& \texttt{(:)}\ (\tau'\eta^{\infty}) \\
= \quad & \{\text{Lemma 4.8}\} \\
& \texttt{(:)}\ \tau
\end{aligned}$$

Otherwise

$$\begin{aligned}
& (\texttt{(:)}\ \tau')\,\mathrm{inst}(\eta^{\infty}) \\
= \quad & \texttt{(:)}\ (\tau'\,\mathrm{inst}(\eta^{\infty})) \\
= \quad & \{\text{Lemma 4.8}\} \\
& \texttt{(:)}\ \tau
\end{aligned}$$

Case $M_1\,M_2$.

Let $(\eta_1, M_1') := \mathcal{R}(V, M_1)$ and $(\eta_2, M_2') := \mathcal{R}(V, M_2)$.

$$\begin{aligned}
& (M_1'\,M_2')\,\mathrm{inst}(\eta^{\infty}) \\
= \quad & (M_1'\,\mathrm{inst}(\eta^{\infty}))\,(M_2'\,\mathrm{inst}(\eta^{\infty})) \\
= \quad & (M_1'\,\mathrm{inst}((\eta_1\eta_2)^{\infty}))\,(M_2'\,\mathrm{inst}((\eta_1\eta_2)^{\infty})) \\
= \quad & \{\mathrm{dom}(\eta_1) \text{ and } \mathrm{dom}(\eta_2) \text{ are new type variables}\} \\
& (M_1'\,\mathrm{inst}(\eta_1^{\infty}\eta_2^{\infty}))\,(M_2'\,\mathrm{inst}(\eta_1^{\infty}\eta_2^{\infty})) \\
= \quad & \{\mathrm{dom}(\eta_1^{\infty}) \text{ and } \mathrm{dom}(\eta_2^{\infty}) \text{ are new type variables}\} \\
& (M_1'\,\mathrm{inst}(\eta_1^{\infty}))\,(M_2'\,\mathrm{inst}(\eta_2^{\infty})) \\
= \quad & \{\text{induction hypothesis}\} \\
& M_1\,M_2
\end{aligned}$$

Case $\lambda\alpha.\, M$.

Let $(\eta, M') := \mathcal{R}(V \cup \{\alpha\}, M)$.

$$\begin{array}{cl}
 & (\lambda\alpha.M')\,\text{inst}(\eta^\infty) \\
= & \quad \{\alpha \notin \text{free}(\eta^\infty) = \text{free}(\text{inst}(\eta^\infty)) \text{ according to Lemma 4.6}\} \\
 & \lambda\alpha.(M'\,\text{inst}(\eta^\infty)) \\
= & \quad \{\text{induction hypothesis}\} \\
 & \lambda\alpha.M
\end{array}$$

■

Well-typed Instance

The result of the replacement algorithm is generally not well-typed, that is, for $(\eta, N) := \mathcal{R}(V, M)$ generally exists *no* type ρ so that

$$\Gamma + \text{tyEnv}(\eta) \vdash N : \rho$$

is valid. However, only instantiating the type variables, hence leaving the constructor variables unchanged, suffices to obtain a valid typing. This property is important, because it assures that the input to the type inference algorithm is typeable and type inference must hence succeed.

Lemma 4.12 (Well-typedness)

$$\frac{(\eta, N) = \mathcal{R}(V, M) \quad \Gamma \vdash M : \rho}{\Gamma + \text{tyEnv}(\eta)\eta^\infty \vdash N\eta^\infty : \rho}$$

□

Proof Structural induction on the term M. We only prove three cases here. The proofs for the other cases are similar.

Case $\mathtt{(:)}\ \tau$.

Let $(\eta, \tau') := \mathcal{R}(V, \tau)$ and γ be a new type variable.

Case $\text{free}(\tau') \cap V = \emptyset$.

According to the typing rules $\Gamma \vdash \mathtt{(:)}\,\tau : \tau \rightarrow \mathtt{[}\tau\mathtt{]} \rightarrow \mathtt{[}\tau\mathtt{]}$. Hence we know for the premise that $\rho = \tau \rightarrow \mathtt{[}\tau\mathtt{]} \rightarrow \mathtt{[}\tau\mathtt{]}$.

$$\begin{array}{cl}
= & \begin{array}{l} \text{tyEnv}(\eta[\mathtt{[}\tau'\mathtt{]}/\gamma])(\eta[\mathtt{[}\tau'\mathtt{]}/\gamma])^\infty(v_\gamma^{\mathtt{(:)}}) \\ (\tau' \rightarrow \gamma \rightarrow \gamma)(\eta[\mathtt{[}\tau'\mathtt{]}/\gamma])^\infty \end{array} \\
= & \quad \{\gamma \notin \text{dom}(\eta) \cup \text{freeTy}(\eta), \text{ because } \gamma \text{ is new}\} \\
 & (\tau' \rightarrow \gamma \rightarrow \gamma)\eta^\infty[\mathtt{[}\tau'\eta^\infty\mathtt{]}/\gamma] \\
= & \quad \{\gamma \notin \text{dom}(\eta) \cup \text{freeTy}(\eta)\} \\
 & \tau'\eta^\infty \rightarrow \mathtt{[}\tau'\eta^\infty\mathtt{]} \rightarrow \mathtt{[}\tau'\eta^\infty\mathtt{]} \\
= & \quad \{\text{Lemma 4.8}\} \\
= & \begin{array}{l} \tau \rightarrow \mathtt{[}\tau\mathtt{]} \rightarrow \mathtt{[}\tau\mathtt{]} \\ \rho \end{array}
\end{array}$$

Otherwise

According to Lemma 4.8 we have $(\mathtt{(:)}\ \tau')\eta^\infty = \mathtt{(:)}\ \tau$. So the conclusion follows with the extensibility of typing environments of Lemma 2.6 from the premise $\Gamma \vdash \mathtt{(:)}\ \tau : \rho$.

Case $M_1\, M_2$.

Let $(\eta_1, M_1') := \mathcal{R}(V, M_1)$ and $(\eta_2, M_2') := \mathcal{R}(V, M_2)$.

From the premise $\Gamma \vdash M_1\, M_2 : \rho$ and the typing rules it follows that $\Gamma \vdash M_1 : \rho' \rightarrow \rho$ and $\Gamma \vdash M_2 : \rho'$ for some type ρ'.

According to the induction hypothesis

$$\begin{aligned}\Gamma + \text{tyEnv}(\eta_1)\eta_1^\infty &\vdash M_1'\eta_1^\infty : \rho' \rightarrow \rho\\ \Gamma + \text{tyEnv}(\eta_2)\eta_2^\infty &\vdash M_2'\eta_2^\infty : \rho'\end{aligned}$$

are valid. Because $\text{dom}(\eta_1) \cap (\text{dom}(\eta_2) \cup \text{freeTy}(\eta_2)) = \emptyset$ and $\text{dom}(\eta_2) \cap (\text{dom}(\eta_1) \cup \text{freeTy}(\eta_1)) = \emptyset$,

$$\text{tyEnv}(\eta_1)\eta_1^\infty + \text{tyEnv}(\eta_2)\eta_2^\infty = \text{tyEnv}(\eta_1\eta_2)(\eta_1\eta_2)^\infty$$

holds. According to Lemma 2.6 the typing environments of the preceding typings can be extended, and with the typing rule TERM APP it follows

$$\Gamma + \text{tyEnv}(\eta_1\eta_2)(\eta_1\eta_2)^\infty \vdash (M_1'\eta_1^\infty)\,(M_2'\eta_2^\infty) : \rho$$

Again because $\text{dom}(\eta_1) \cap (\text{dom}(\eta_2) \cup \text{freeTy}(\eta_2)) = \emptyset$ and $\text{dom}(\eta_2) \cap (\text{dom}(\eta_1) \cup \text{freeTy}(\eta_1)) = \emptyset$,

$$(M_1'\eta_1^\infty)\,(M_2'\eta_2^\infty) = (M_1'\, M_2')(\eta_1\eta_2)^\infty$$

holds. Hence the conclusion of the lemma holds.

Case $\lambda\alpha.\, M$.

Let $(\eta, M') := \mathcal{R}(V \cup \{\alpha\}, M)$.

From the premise $\Gamma \vdash \lambda\alpha.\, M : \rho$ and the typing rules it follows that $\Gamma \vdash M : \rho'$ with $\rho = \forall\alpha.\rho'$ and $\alpha \notin \text{freeTy}(\Gamma)$.

According to the induction hypothesis

$$\Gamma + \text{tyEnv}(\eta)\eta^\infty \vdash M'\eta^\infty : \rho'$$

holds. From Lemma 4.6 it follows that $\alpha \notin \text{freeTy}(\eta) = \text{freeTy}(\text{tyEnv}(\eta)\eta^\infty)$. Therefore we can construct with the typing rule TYPE ABS the valid typing

$$\Gamma + \text{tyEnv}(\eta)\eta^\infty \vdash \lambda\alpha.\,(M'\eta^\infty) : \forall\alpha.\rho'$$

Again because $\alpha \notin \text{freeTy}(\eta)$, we have $\lambda\alpha.\,(M'\eta^\infty) = (\lambda\alpha.\, M)\eta^\infty$. Hence the conclusion of the lemma holds. ■

4.3 Type Inference Phase

The second phase of list substitution consists of applying a type inference algorithm. At first, the fact that typeability for the second-order typed λ-calculus is undecidable [Wel94] seems to pose a problem. However, for our purposes a *partial* type inference algorithm that only instantiates some type variables and makes use of the existing type annotations of the program suffices.

The idea is that the algorithm gets a term together with a typing environment for its free term variables as input and then replaces some of the type variables in the term and the typing environment to obtain a valid typing. That is, for an input $\Gamma \vdash M$ it yields a type substitution σ and a type τ such that $\Gamma\sigma \vdash M\sigma : \tau\sigma$. The substitution σ holds the information required for list abstraction whereas the type τ will only be needed in the recursive definition of the type inference algorithm itself.

Definition 4.13 A tuple of a type substitution and a type (σ, τ) is a **typing for a typing environment and a term** $\Gamma \vdash M$, if $\Gamma\sigma \vdash M\sigma : \tau\sigma$. □

Although the type inference algorithm yields a tuple (σ, τ) and all subsequent algorithms will work on the substitution component σ, we usually give $\Gamma\sigma \vdash M\sigma : \tau\sigma$ as result of type inference, because it is easier to read.

For example, for the typing environment and term

$$\emptyset \vdash \lambda\texttt{x:Int}.\lambda\texttt{xs:[Int]}.\,\texttt{(:) Int x xs}$$

the list replacement algorithm yields

$$([\texttt{[Int]}/\gamma_1,\, \texttt{[Int]}/\gamma_2],\ \lambda\texttt{x:Int}.\lambda\texttt{xs:}\gamma_1.\, v^{\texttt{(:)}}_{\gamma_2}\ \texttt{x xs})$$

The types of the constructor variable is determined by the typing environment

$$\text{tyEnv}([\texttt{[Int]}/\gamma_1,\, \texttt{[Int]}/\gamma_2]) = \{v^{\texttt{(:)}}_{\gamma_1}\texttt{:Int}\to\gamma_1\to\gamma_1,\ v^{\texttt{[]}}_{\gamma_1}\texttt{:}\gamma_1,\ v^{\texttt{(:)}}_{\gamma_2}\texttt{:Int}\to\gamma_2\to\gamma_2,\ v^{\texttt{[]}}_{\gamma_2}\texttt{:}\gamma_2\}$$

The fact that the typing environment contains superfluous constructor variables which do not occur in the term $\lambda\texttt{x:Int}.\lambda\texttt{xs:}\gamma_1.\, v^{\texttt{(:)}}_{\gamma_2}\ \texttt{x xs}$, is unimportant.

Naturally, the type inference algorithm takes the instance substitution as input instead of the explicit typing environment for constructor variables, but for the moment we ignore this as we already did in the previous chapter. So the input of the type inference algorithm is:

$$\emptyset + \text{tyEnv}([\texttt{[Int]}/\gamma_1,\, \texttt{[}\gamma_1\texttt{]}/\gamma_2]) \vdash \lambda\texttt{x:Int}.\lambda\texttt{xs:}\gamma_1.\, v^{\texttt{(:)}}_{\gamma_2}\ \texttt{x xs}$$

The type inference algorithm yields the valid typing

$$([\gamma_2/\gamma_1], \texttt{Int}\to\gamma_1\to\gamma_2)$$

So

$$\begin{array}{l}(\emptyset + \text{tyEnv}([\texttt{[Int]}/\gamma_1,\, \texttt{[Int]}/\gamma_2]))\ \ [\gamma_2/\gamma_1]\\ \vdash\ (\lambda\texttt{x:Int}.\lambda\texttt{xs:}\gamma_1.\, v^{\texttt{(:)}}_{\gamma_2}\ \texttt{x xs})\ \ \ [\gamma_2/\gamma_1]\\ :\ \ (\texttt{Int}\to\gamma_1\to\gamma_2)\ \ [\gamma_2/\gamma_1]\end{array}$$

that is,

$$\begin{array}{l}\{v^{\texttt{(:)}}_{\gamma_1}\texttt{:Int}\to\gamma_2\to\gamma_2,\ \ v^{\texttt{[]}}_{\gamma_1}\texttt{:}\gamma_2,\ \ \ v^{\texttt{(:)}}_{\gamma_2}\texttt{:Int}\to\gamma_2\to\gamma_2,\ \ \ v^{\texttt{[]}}_{\gamma_2}\texttt{:}\gamma_2\}\\ \vdash\ \lambda\texttt{x:Int}.\lambda\texttt{xs:}\gamma_2.\, v^{\texttt{(:)}}_{\gamma_2}\ \texttt{x xs}\\ :\ \ \texttt{Int}\to\gamma_2\to\gamma_2\end{array}$$

According to the type substitution property of Lemma 2.6 we can apply any type substitution σ' to a typing $\Gamma\sigma \vdash M\sigma : \tau\sigma$ and obtain a (possibly different) typing $\Gamma\sigma\sigma' \vdash M\sigma\sigma' : \tau\sigma\sigma'$. So for the preceding example $([\texttt{[Int]}/\gamma_1,\, \texttt{[Int]}/\gamma_2], \texttt{Int}\to\texttt{[Int]}\to\texttt{[Int]})$ is also a valid typing.

Definition 4.14 A substitution σ is **more general** than a substitution $\hat{\sigma}$, written $\sigma \preceq_\alpha \hat{\sigma}$, if there exists a substitution σ' with $\sigma\sigma' \equiv \hat{\sigma}$. Alternatively we say that $\hat{\sigma}$ **extends** σ. □

A typing $(\tilde{\sigma}, \tilde{\tau})$ for a typing environment and a term $\Gamma \vdash M$ is a **principal typing**[1] for $\Gamma \vdash M$, if for all typings $(\hat{\sigma}, \hat{\tau})$ of $\Gamma \vdash M$, $\tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and there exists a type substitution σ with $\tilde{\tau}\tilde{\sigma}\sigma \equiv \hat{\tau}\hat{\sigma}$. For reasons that we discuss in Section 4.3.3 the type inference phase will not always compute a principal typing but something closely related.

We distinguish those type variables that are introduced by the list replacement algorithm $\mathcal{R}$, to which we refer as **type inference variables**, from all other type variables. In the whole type inference phase we replace only type inference variables. All other type variables, which are bound somewhere in the program by type abstraction or universal quantification, are treated like constants. To avoid extension of the syntax of F we do not formally introduce type inference variables as a new category of variables. Instead, the list substitution η that is constructed by the list replacement algorithm $\mathcal{R}$ comes in handy for the first time. Its domain, $\text{dom}(\eta)$, contains exactly the type variables introduced by $\mathcal{R}$, that is, $\text{dom}(\eta)$ is the set of type inference variables.

4.3.1 Idempotent Substitutions

Consider the environment and term

$$\emptyset \vdash \lambda\texttt{x:}\gamma_1\texttt{.}\lambda\texttt{y:}\gamma_2\texttt{. case x of }\{\texttt{[]}\mapsto\texttt{y}\}$$

$([\texttt{[Int]}/\gamma_1, \gamma_1/\gamma_2], \texttt{[Int]} \rightarrow \gamma_1 \rightarrow \gamma_1)$ is a typing for it. It is however pointless that γ_2 is replaced by γ_1. In general the substitution of a typing has to replace type variables by other type variables, but if a type variable is replaced, then it should disappear completely. In other words, it should be idempotent:

Definition 4.15 A substitution σ is idempotent, if $\sigma\sigma = \sigma$. □

All parts of the type inference algorithm and also of the third phase yield only idempotent substitutions provided the substitutions they get as input are idempotent. This invariant simplifies both algorithms and proofs. In the following we give several properties of idempotent substitutions that we will use later.

Lemma 4.16 (Characterisation of idempotent substitutions)

$$\sigma \textit{ idempotent} \Longleftrightarrow \text{dom}(\sigma) \cap \text{free}(\sigma) = \emptyset$$ □

Proof Trivial. ■

Lemma 4.17 (Extension of idempotent substitution)

$$\frac{\sigma \preceq_\alpha \rho \quad \sigma \textit{ idempotent}}{\sigma\rho \equiv \rho}$$ □

Proof Simple, see [Ede85]. ■

Lemma 4.18 (Preservation of idempotency)

$$\frac{\sigma \textit{ idempotent} \quad \gamma \notin \text{dom}(\sigma) \quad \gamma \notin \text{free}(\tau) \quad \text{dom}(\sigma) \cap \text{free}(\tau) = \emptyset}{\sigma[\tau/\gamma] \textit{ idempotent}}$$ □

[1] [Jim96] distinguishes between a principal type property and a principal typing property. Given a typing environment and a term its principal type is a type that "represents" all valid types for this typing environment and term. Given only a term, a principal typing is a typing environment and a type that "represent" all valid typings for this term. Type inference algorithms for the Hindley-Milner type system determine principal types [Mil78, DM82]. The Hindley-Milner type system does not have principal typings in any sensible way. Our definition of principal typing differs from both definitions of Jim. Here a typing environment and a term with *partial* type information are given.

Proof We show that the domain and the set of free variables of $\sigma[\tau/\gamma]$ are disjoint.

$$\begin{array}{ll}
 & \mathrm{dom}(\sigma[\tau/\gamma]) \cap \mathrm{free}(\sigma[\tau/\gamma]) \\
\subseteq & \{\text{equal only if } \gamma \neq \tau\} \\
 & (\mathrm{dom}(\sigma) \cup \{\gamma\}) \cap \mathrm{free}(\sigma[\tau/\gamma]) \\
= & \{\text{premise: } \gamma \notin \mathrm{dom}(\sigma)\} \\
 & (\mathrm{dom}(\sigma) \cup \{\gamma\}) \cap ((\mathrm{free}(\sigma) \setminus \{\gamma\}) \cup \mathrm{free}(\tau)) \\
= & \\
 & (\mathrm{dom}(\sigma) \cap (\mathrm{free}(\sigma) \setminus \{\gamma\})) \cup (\{\gamma\} \cap (\mathrm{free}(\sigma) \setminus \{\gamma\})) \cup (\mathrm{dom}(\sigma) \cap \mathrm{free}(\tau)) \\
 & \cup(\{\gamma\} \cap \mathrm{free}(\tau)) \\
= & \{\text{premise: } \gamma \notin \mathrm{free}(\tau)\} \\
 & (\mathrm{dom}(\sigma) \cap (\mathrm{free}(\sigma) \setminus \{\gamma\})) \cup (\{\gamma\} \cap (\mathrm{free}(\sigma) \setminus \{\gamma\})) \cup (\mathrm{dom}(\sigma) \cap \mathrm{free}(\tau)) \\
= & \{\text{premise: } \mathrm{dom}(\sigma) \cup \mathrm{free}(\tau) = \emptyset\} \\
 & (\mathrm{dom}(\sigma) \cap (\mathrm{free}(\sigma) \setminus \{\gamma\})) \cup (\{\gamma\} \cap (\mathrm{free}(\sigma) \setminus \{\gamma\})) \\
= & \\
 & (\mathrm{dom}(\sigma) \cap (\mathrm{free}(\sigma) \setminus \{\gamma\})) \\
= & \{\text{premise: } \sigma \text{ idempotent}\} \\
 & \emptyset
\end{array}$$

■

Corollary 4.19 (Preservation of idempotency)

$$\frac{\sigma \textit{ idempotent} \quad \gamma \notin \mathrm{dom}(\sigma) \quad \gamma \notin \mathrm{free}(\tau\sigma)}{\sigma[\tau\sigma/\gamma] \textit{ idempotent}}$$

□

Proof We know that σ is idempotent, $\gamma \notin \mathrm{dom}(\sigma)$, $\gamma \notin \mathrm{free}(\tau\sigma)$ and

$$\begin{array}{ll}
 & \mathrm{dom}(\sigma) \cap \mathrm{free}(\tau\sigma) \\
\subseteq & \\
 & \mathrm{dom}(\sigma) \cap \mathrm{free}(\sigma) \\
= & \{\sigma \text{ idempotent}\} \\
 & \emptyset
\end{array}$$

Hence the conclusion follows with Lemma 4.18. ■

Lemma 4.20 (Extension of substitution)

$$\frac{\sigma\hat{\sigma} \equiv \hat{\sigma} \quad \gamma\hat{\sigma} \equiv \tau\hat{\sigma}}{\sigma[\tau/\gamma]\hat{\sigma} \equiv \hat{\sigma}}$$

□

Proof We show that $[\tau/\gamma]\hat{\sigma} \equiv \hat{\sigma}$ by case analysis:

Case γ.
$\gamma[\tau/\gamma]\hat{\sigma} = \tau\hat{\sigma} \equiv \gamma\hat{\sigma}$.

Case $\alpha \neq \gamma$.
$\alpha[\tau/\gamma]\hat{\sigma} = \alpha\hat{\sigma}$.

So $\sigma[\tau/\gamma]\hat{\sigma} \equiv \sigma\hat{\sigma} \equiv \hat{\sigma}$. ■

Lemma 4.21 (Equality of idempotent substitutions)

$$\frac{\sigma \textit{ idempotent} \quad \sigma \preceq_\alpha \sigma' \quad \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')}{\sigma \equiv \sigma'}$$

□

Proof Let $\gamma \in \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$.

$$\begin{aligned}
& \gamma\sigma \\
\equiv \quad & \{\ \mathrm{free}(\gamma\sigma) \cap \mathrm{dom}(\sigma') = \mathrm{free}(\gamma\sigma) \cap \mathrm{dom}(\sigma) \subseteq \mathrm{free}(\sigma) \cap \mathrm{dom}(\sigma) = \emptyset\} \\
& \gamma\sigma\sigma' \\
\equiv \quad & \{\sigma\sigma' \equiv \sigma'\} \\
& \gamma\sigma'
\end{aligned}$$

■

Lemma 4.22 (Inclusion of modified variables)

$$\frac{\sigma\sigma' \equiv \sigma'}{\mathrm{mod}(\sigma) \subseteq \mathrm{mod}(\sigma')}$$

□

Proof

$$\begin{aligned}
& \mathrm{mod}(\sigma) \\
= \quad & \\
& \mathrm{dom}(\sigma) \cup \mathrm{free}(\sigma) \\
\subseteq \quad & \{\sigma \preceq_\alpha \sigma'\} \\
& \mathrm{dom}(\sigma') \cup \mathrm{free}(\sigma) \\
= \quad & \\
& \mathrm{dom}(\sigma') \cup (\mathrm{free}(\sigma) \setminus \mathrm{dom}(\sigma')) \\
\subseteq \quad & \\
& \mathrm{dom}(\sigma') \cup \mathrm{free}(\sigma\sigma') \\
= \quad & \{\sigma\sigma' \equiv \sigma'\} \\
& \mathrm{dom}(\sigma') \cup \mathrm{free}(\sigma') \\
= \quad & \\
& \mathrm{mod}(\sigma')
\end{aligned}$$

■

4.3.2 The Unification Algorithm $\mathcal{U}$

At the heart of the type inference algorithm lies a unification algorithm, which combines typings of subterms.

Definition 4.23 A type substitution σ is a **unifier** of an equation $\tau_1 \doteq \tau_2$, that is, of the two types τ_1 and τ_2, if $\tau_1\sigma \equiv \tau_2\sigma$. Moreover, $\tilde{\sigma}$ is a **most general unifier** for $\tau_1 \doteq \tau_2$, if it is a unifier for $\tau_1 \doteq \tau_2$ and for every unifier $\hat{\sigma}$ for $\tau_1 \doteq \tau_2$ we have $\tilde{\sigma} \preceq_\alpha \hat{\sigma}$. □

Many algorithms for computing a most general unifier are known [Rob65, MM82, Mit96]. Unfortunately, we need to define our own unification algorithm, because we distinguish between type inference variables and "normal" type variables and F contains type variable binders, that is, type abstraction in terms and universal quantification in types. Nonetheless, we do not have to invent a completely different algorithm or enter the realms of higher-order unification, which is undecidable [Gol81, BS94]. According to the definition of $\mathcal{R}$, the set of type inference variables $\mathrm{dom}(\eta)$ is disjoint from the set of all bound type variables. Hence abstracted and universally quantified variables are not to be replaced by the unification algorithm and universal quantification itself need not be handled differently from other syntactic constructs such as function or data type.

Because the input of the type inference algorithm is typeable, all inputs of the unification algorithm must be unifiable by a substitution which only replaces type inference variables. Based on this knowledge we can drop several tests in our unification algorithm,

Structural calls:

$$
\begin{aligned}
\mathcal{U}_V(T\,\tau_1..\tau_k \doteq T\,\rho_1..\rho_k, \sigma) &:= \mathcal{U}_V(\tau_1 \doteq \rho_1, \mathcal{U}_V(\tau_2 \doteq \rho_2, ..\mathcal{U}_V(\tau_k \doteq \rho_k, \sigma)..)) \\
& (= \sigma, \text{ if } k = 0) \\
\mathcal{U}_V(\tau_1 \to \tau_2 \doteq \rho_1 \to \rho_2, \sigma) &:= \mathcal{U}_V(\tau_1 \doteq \rho_1, \mathcal{U}_V(\tau_2 \doteq \rho_2, \sigma)) \\
\mathcal{U}_V(\forall\alpha.\tau \doteq \forall\beta.\rho, \sigma) &:= \mathcal{U}_V(\tau \doteq \rho, \sigma) \\
\mathcal{U}_V(\alpha \doteq \beta, \sigma) &:= \sigma \quad, \text{if } \alpha \notin V \text{ and } \beta \notin V
\end{aligned}
$$

Substitution calls:

$$
\begin{aligned}
\mathcal{U}_V(\gamma \doteq \tau, \sigma) &:= \mathcal{U}_V(\sigma(\gamma) \doteq \tau, \sigma) \quad, \text{if } \gamma \in \text{dom}(\sigma) \\
\mathcal{U}_V(\tau \doteq \gamma, \sigma) &:= \mathcal{U}_V(\tau \doteq \sigma(\gamma), \sigma) \quad, \text{if } \gamma \in \text{dom}(\sigma)
\end{aligned}
$$

Unification calls:

$$
\begin{aligned}
\mathcal{U}_V(\gamma \doteq \tau, \sigma) &:= \sigma[\tau\sigma/\gamma] \quad, \text{if } \gamma \in V \setminus \text{dom}(\sigma) \\
\mathcal{U}_V(\tau \doteq \gamma, \sigma) &:= \sigma[\tau\sigma/\gamma] \quad, \text{if } \gamma \in V \setminus \text{dom}(\sigma)
\end{aligned}
$$

Figure 4.2: Definition of the unification algorithm $\mathcal{U}$

$$
\begin{aligned}
& \mathcal{U}_{\{\gamma\}}(\forall\alpha.\texttt{(}\alpha\texttt{,[Int])}{\to}\texttt{[Int]} \doteq \forall\beta.\texttt{(}\beta\texttt{,}\gamma\texttt{)}{\to}\gamma, [\,]) \\
= \; & \mathcal{U}_{\{\gamma\}}(\texttt{(}\alpha\texttt{,[Int])}{\to}\texttt{[Int]} \doteq \texttt{(}\beta\texttt{,}\gamma\texttt{)}{\to}\gamma, [\,]) \\
= \; & \mathcal{U}_{\{\gamma\}}(\texttt{(}\alpha\texttt{,[Int])} \doteq \texttt{(}\beta\texttt{,}\gamma\texttt{)}, \mathcal{U}_{\{\gamma\}}(\texttt{[Int]} \doteq \gamma, [\,])) \\
= \; & \mathcal{U}_{\{\gamma\}}(\texttt{(}\alpha\texttt{,[Int])} \doteq \texttt{(}\beta\texttt{,}\gamma\texttt{)}, [\texttt{[Int]}/\gamma]) \\
= \; & \mathcal{U}_{\{\gamma\}}(\alpha \doteq \beta, \mathcal{U}_{\{\gamma\}}(\texttt{[Int]} \doteq \gamma, [\texttt{[Int]}/\gamma]) \\
= \; & \mathcal{U}_{\{\gamma\}}(\alpha \doteq \beta, \mathcal{U}_{\{\gamma\}}(\texttt{[Int]} \doteq \texttt{[Int]}, [\texttt{[Int]}/\gamma]) \\
= \; & \mathcal{U}_{\{\gamma\}}(\alpha \doteq \beta, \mathcal{U}_{\{\gamma\}}(\texttt{Int} \doteq \texttt{Int}, [\texttt{[Int]}/\gamma]) \\
= \; & \mathcal{U}_{\{\gamma\}}(\alpha \doteq \beta, [\texttt{[Int]}/\gamma]) \\
= \; & [\texttt{[Int]}/\gamma]
\end{aligned}
$$

Figure 4.3: An example computation of $\mathcal{U}$

so that it is simpler than standard unification algorithms. Note that the following arguments assume that the input is unifiable by a substitution which only replaces type inference variables. Our algorithm yields wrong results for other inputs; it may not even terminate then.

The unification algorithm $\mathcal{U}$ is given in Figure 4.2. Its first argument V is the set of type inference variables, that is, the type inference algorithm will call $\mathcal{U}$ with $V = \text{dom}(\eta)$. Besides two types τ and ρ, the algorithm also gets an idempotent substitution σ as input. The algorithm computes a unifier $\tilde{\sigma}$ of $\tau\sigma \doteq \rho\sigma$ which is not necessarily a most general unifier but which is more general than all unifiers $\hat{\sigma}$ of $\tau\sigma \doteq \rho\sigma$ that extend σ. $\mathcal{U}_V(\tau \doteq \rho, [\,])$ computes the most general unifier of $\tau \doteq \rho$.

Basically the unification algorithm works by stepwise extension of the original input substitution and threading of this substitution through the whole computation. If, for example, both types are function types, then it unifies the two result types and the two argument types. The substitution obtained from the first unification call is passed as input to the second unification call. So instead of applying this substitution to the two type arguments of the second unification call, as is done by standard unification algorithms, we pass this information separately. We chose to formulate $\mathcal{U}$ with this additional input substitution, because it makes the definition of the unification and of the type inference algorithm more readable. Furthermore, it already suggests an efficient implementation method (cf. Section 4.5). For data types and universal types it works similarly. In general the unification algorithm recursively traverses both type arguments in parallel. The interesting cases are those where one of the types is a type variable.

If both types are type variables that are not in V, that is, are not type inference variables, we do not have to extend the substitution, because according to our assumption the input of $\mathcal{U}$ is unifiable. However, if this is in a recursive call of $\mathcal{U}$, then the two type variables need not to be equal. The reason is that in the unification of two universal types the algorithm completely ignores the bound type variables. Sure, according to our assumption the two universal types must be unifiable, but to make them syntactically equal it usually would be necessary to rename one of the bound variables. This renaming would also change the type variable in the body of the type. The unification algorithm does not perform this renaming and hence does not require two type variables which are not type inference variables to be equal for successful unification. To be precise, we assume the input of $\mathcal{U}$ to be unifiable modulo renaming of some variables. However, the details of how this works are given in the proof of Lemma 4.24. The central point is that $\mathcal{U}$ can handle unification up to renaming of bound type variables without needing to perform any renaming.

If one of the two type arguments of $\mathcal{U}$ is a type variable in V, that is, a type inference variable, then we have to distinguish two cases. If the variable is in the domain of the current substitution, then we apply the substitution at this point and continue unification. So we see that by passing the substitution as a separate argument we only defer the substitution. As the proof of Lemma 4.24 will show, infinite repetition of this situation, which would lead to non-termination, cannot happen , because the substitution is idempotent. If one of the two type arguments of $\mathcal{U}$ is a type inference variable that is not in the domain of the current substitution, then the unification algorithm extends the current substitution by substituting the other type argument for the type variable. Note that the occur check of standard unification algorithms is not necessary here, because we assume that the input of $\mathcal{U}$ is unifiable.

The algorithm $\mathcal{U}$ is non-deterministic in that some arguments match more than one case of the definition. For example, for $\gamma_1 \doteq \gamma_2$ with $\gamma_1, \gamma_2 \in V \setminus \mathrm{dom}(\sigma)$, γ_1 may be replaced by γ_2 or γ_2 by γ_1. However, it follows from Lemma 4.24 that in any case the result is a unifier with the desired properties.

Note that the definition of $\mathcal{U}$ covers all possible arguments, because we assume that the input is unifiable. We cannot have a mismatch of arguments, that is, for example one type is a data type and the other one a function type. Also we assume for the input of $\mathcal{U}$ that $\mathrm{dom}(\sigma) \subseteq V$ and the algorithm preserves this invariant.

The example in Figure 4.3 illustrates how $\mathcal{U}$ works.

Finally, the following lemma states exactly under which conditions $\mathcal{U}$ computes a unifier and in which respect it is most general. Note that the result of $\mathcal{U}$ is computed independently of the unifier $\hat{\sigma}$. So the result is more general than any unifier $\hat{\sigma}$ which fulfils the given premises. For any sets V_1 and V_2 we abbreviate the proposition $V_1 \cap V_2 = \emptyset$

by $V_1 \mathrel{\#} V_2$.

Lemma 4.24 (Most general unifier)

$$\frac{\begin{array}{c}\tau\hat{\sigma} \equiv \rho\hat{\sigma} \quad \sigma \textit{ idempotent} \quad \sigma \preceq_\alpha \hat{\sigma} \quad \mathrm{dom}(\hat{\sigma}) \subseteq V \\ V \mathrel{\#} \mathrm{bound}(\tau) \quad V \mathrel{\#} \mathrm{bound}(\rho) \quad V \mathrel{\#} \mathrm{bound}(\sigma)\end{array}}{\exists\tilde{\sigma} \quad \tilde{\sigma} = \mathcal{U}_V(\tau \doteq \rho, \sigma) \quad \tau\tilde{\sigma} \equiv \rho\tilde{\sigma} \quad \tilde{\sigma} \textit{ idempotent} \quad \sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma} \quad V \mathrel{\#} \mathrm{bound}(\tilde{\sigma})} \quad \square$$

Proof We have to strengthen the property for an inductive proof. We prove the following:

$$\frac{\begin{array}{c}\varsigma_{\mathrm{l}} \text{ variable renaming} \quad \varsigma_{\mathrm{r}} \text{ variable renaming} \\ \mathrm{free}(\varsigma_{\mathrm{l}}) \mathrel{\#} \mathrm{free}(\hat{\sigma}) \quad \mathrm{free}(\varsigma_{\mathrm{r}}) \mathrel{\#} \mathrm{free}(\hat{\sigma}) \quad V \mathrel{\#} \mathrm{mod}(\varsigma_{\mathrm{l}}) \quad V \mathrel{\#} \mathrm{mod}(\varsigma_{\mathrm{r}}) \\ \tau\varsigma_{\mathrm{l}}\hat{\sigma} \equiv \rho\varsigma_{\mathrm{r}}\hat{\sigma} \quad \sigma \text{ idempotent} \quad \sigma \preceq_\alpha \hat{\sigma} \quad \mathrm{dom}(\hat{\sigma}) \subseteq V \\ V \mathrel{\#} \mathrm{bound}(\tau) \quad V \mathrel{\#} \mathrm{bound}(\rho) \quad V \mathrel{\#} \mathrm{bound}(\sigma)\end{array}}{\exists\tilde{\sigma} \quad \tilde{\sigma} = \mathcal{U}_V(\tau \doteq \rho, \sigma) \quad \tau\varsigma_{\mathrm{l}}\tilde{\sigma} \equiv \rho\varsigma_{\mathrm{r}}\tilde{\sigma} \quad \tilde{\sigma} \text{ idempotent} \quad \sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma} \quad V \mathrel{\#} \mathrm{bound}(\tilde{\sigma})}$$

where a **variable renaming** is a substitution which maps variables only to variables.

For a fixed set V let the degree of $\tau \doteq \rho$ and σ be the triple (m, n, o) where $m = |V \setminus \mathrm{dom}(\sigma)|$, that is, the number of variables in V which are not replaced by σ, and n is the size of $\tau\sigma$ and $\rho\sigma$, that is, the number of occurrences of type constructors, →, ∀ and variables in it, and o is given by:

$$o := \begin{cases} 2 & \text{, if } \tau \in \mathrm{dom}(\sigma) \text{ and } \rho \in \mathrm{dom}(\sigma) \\ 1 & \text{, if either } \tau \in \mathrm{dom}(\sigma) \text{ or } \rho \in \mathrm{dom}(\sigma) \\ 0 & \text{, otherwise} \end{cases}$$

We say that (m, n, o) is smaller than (m', n', o') if either $m < m'$, or $m = m'$ and $n < n'$, or $m = m'$ and $n = n'$ and $o < o'$ (cf. [Mit96], Proof of Lemma 11.2.4).

We prove by induction on the degree of $\tau \doteq \rho$ and σ:

Case $\mathcal{U}_V(T\,\tau_1..\tau_k \doteq T\,\rho_1..\rho_k, \sigma) = \mathcal{U}_V(\tau_1 \doteq \rho_1, \mathcal{U}_V(\tau_2 \doteq \rho_2, ..\mathcal{U}_V(\tau_k \doteq \rho_k, \sigma)..))$.
Similar to the next case.

Case $\mathcal{U}_V(\tau_1 \to \tau_2 \doteq \rho_1 \to \rho_2, \sigma) = \mathcal{U}_V(\tau_1 \doteq \rho_1, \mathcal{U}_V(\tau_2, \rho_2, \sigma))$.
From $(\tau_1 \to \tau_2)\varsigma_{\mathrm{l}}\hat{\sigma} \equiv (\rho_1 \to \rho_2)\varsigma_{\mathrm{r}}\hat{\sigma}$ it follows $\tau_2\varsigma_{\mathrm{l}}\hat{\sigma} \equiv \rho_2\varsigma_{\mathrm{r}}\hat{\sigma}$. The degree of $\tau_2 \doteq \rho_2$ and σ is obviously smaller than the degree of $\tau_1 \to \tau_2 \doteq \rho_1 \to \rho_2$ and σ.

Hence by induction hypothesis there exists an idempotent $\sigma' = \mathcal{U}_V(\tau_2 \doteq \rho_2, \sigma)$ with $\tau_2\varsigma_{\mathrm{l}}\sigma' =_\varsigma \rho_2\varsigma_{\mathrm{r}}\sigma'$ and $\sigma \preceq_\alpha \sigma' \preceq_\alpha \hat{\sigma}$ and $V \mathrel{\#} \mathrm{bound}(\sigma')$.

From $(\tau_1 \to \tau_2)\varsigma_{\mathrm{l}}\hat{\sigma} \equiv (\rho_1 \to \rho_2)\varsigma_{\mathrm{r}}\hat{\sigma}$ also it follows $\tau_1\varsigma_{\mathrm{l}}\hat{\sigma} \equiv \rho_1\varsigma_{\mathrm{r}}\hat{\sigma}$. If $\tau_1\varsigma_{\mathrm{l}}\sigma' \equiv \tau_1\varsigma_{\mathrm{l}}\sigma$ and $\rho_1\varsigma_{\mathrm{r}}\sigma' \equiv \rho_1\varsigma_{\mathrm{r}}\sigma$, then the degree of $\tau_1 \doteq \rho_1$ and σ' is smaller than the degree of $\tau_1 \to \tau_2 \doteq \rho_1 \to \rho_2$ and σ. Otherwise $\sigma \not\equiv \sigma'$ and hence according to Lemma 4.21 $\mathrm{dom}(\sigma) \subset \mathrm{dom}(\sigma')$. So in this case the first component of the degree is reduced.

Hence by induction hypothesis there exists an idempotent $\tilde{\sigma} = \mathcal{U}_V(\tau_1 \doteq \rho_1, \sigma')$ with $\tau_1\varsigma_{\mathrm{l}}\tilde{\sigma} \equiv \rho_1\varsigma_{\mathrm{l}}\tilde{\sigma}$ and $\sigma' \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and $V \mathrel{\#} \mathrm{bound}(\tilde{\sigma})$.

So we have $\sigma \preceq_\alpha \sigma' \preceq_\alpha \tilde{\sigma}$ and

$$\begin{aligned}
& (\tau_1 \to \tau_2)\varsigma_{\mathrm{l}}\tilde{\sigma} \\
= \; & \tau_1\varsigma_{\mathrm{l}}\tilde{\sigma} \to \tau_2\varsigma_{\mathrm{l}}\tilde{\sigma} \\
\equiv \; & \quad \{\sigma' \text{ idempotent and } \sigma' \preceq_\alpha \tilde{\sigma}\text{; Lemma 4.17}\} \\
& \tau_1\varsigma_{\mathrm{l}}\tilde{\sigma} \to \tau_2\varsigma_{\mathrm{l}}\sigma'\tilde{\sigma} \\
\equiv \; & \quad \{\tau_1\varsigma_{\mathrm{l}}\tilde{\sigma} \equiv \rho_1\varsigma_{\mathrm{r}}\tilde{\sigma} \text{ and } \tau_2\varsigma_{\mathrm{l}}\sigma' \equiv \rho_2\varsigma_{\mathrm{r}}\sigma'\} \\
& \rho_1\varsigma_{\mathrm{r}}\tilde{\sigma} \to \rho_2\varsigma_{\mathrm{r}}\sigma'\tilde{\sigma} \\
\equiv \; & \quad \{\sigma' \text{ idempotent and } \sigma' \preceq_\alpha \tilde{\sigma}\text{; Lemma 4.17}\} \\
& \rho_1\varsigma_{\mathrm{r}}\tilde{\sigma} \to \rho_2\varsigma_{\mathrm{r}}\tilde{\sigma} \\
= \; & (\rho_1 \to \rho_2)\varsigma_{\mathrm{r}}\tilde{\sigma}
\end{aligned}$$

Case $\mathcal{U}_V(\forall\alpha.\tau \doteq \forall\beta.\rho, \sigma) = \mathcal{U}_V(\tau \doteq \rho, \sigma)$.

Let γ be a type variable with $\gamma \notin V \cup \mathrm{mod}(\hat{\sigma}) \cup \mathrm{mod}(\varsigma_{\mathrm{l}}) \cup \mathrm{mod}(\varsigma_{\mathrm{r}}) \cup \mathrm{free}(\tau) \cup \mathrm{free}(\rho)$. We have

$$\begin{aligned}
& \forall\gamma.(\tau[\gamma/\alpha]\varsigma_{\mathrm{l}}\hat{\sigma}) \\
= \; & \quad \{\gamma \notin \mathrm{mod}(\hat{\sigma}) \cup \mathrm{mod}(\varsigma_{\mathrm{l}}) \text{ and Lemma 2.3}\} \\
& (\forall\gamma.\tau[\gamma/\alpha])\varsigma_{\mathrm{l}}\hat{\sigma} \\
\equiv \; & \quad \{\gamma \notin \mathrm{free}(\tau)\text{, Definition of } \equiv\} \\
& (\forall\alpha.\tau)\varsigma_{\mathrm{l}}\hat{\sigma} \\
\equiv \; & \quad \{\text{premise}\} \\
& (\forall\beta.\rho)\varsigma_{\mathrm{r}}\hat{\sigma} \\
\equiv \; & \quad \{\gamma \notin \mathrm{free}(\rho)\text{, Definition of } \equiv\} \\
& (\forall\gamma.\rho[\gamma/\beta])\varsigma_{\mathrm{r}}\hat{\sigma} \\
= \; & \quad \{\gamma \notin \mathrm{mod}(\hat{\sigma}) \cup \mathrm{mod}(\varsigma_{\mathrm{r}}) \text{ and Lemma 2.3}\} \\
& \forall\gamma.(\rho[\gamma/\beta]\varsigma_{\mathrm{r}}\hat{\sigma})
\end{aligned}$$

So $\tau[\gamma/\alpha]\varsigma_{\mathrm{l}}\hat{\sigma} \equiv \rho[\gamma/\beta]\varsigma_{\mathrm{r}}\hat{\sigma}$.

With Lemma 4.22 it follows that $\mathrm{mod}(\sigma) \subseteq \mathrm{mod}(\hat{\sigma})$ and hence $(\forall\alpha.\tau)\sigma = \forall\gamma.(\tau[\gamma/\alpha]\sigma)$ and $(\forall\beta.\rho)\sigma = \forall\gamma.(\rho[\gamma/\beta]\sigma)$. Because $\alpha, \beta, \gamma \notin \mathrm{dom}(\sigma) \subseteq V$, $\tau[\gamma/\alpha]\sigma$ and $\tau\sigma$ have the same size and also $\rho[\gamma/\beta]\sigma$ and $\rho\sigma$. Therefore the size of $\tau\sigma$ and $\rho\sigma$ is smaller than the size of $(\forall\alpha.\tau)\sigma$ and $(\forall\beta.\rho)\sigma$. So the degree of $\tau \doteq \rho$ and σ is smaller than the degree of $\forall\alpha.\tau \doteq \forall\beta.\rho$ and σ.

Hence by induction hypothesis there exists an idempotent $\tilde{\sigma} = \mathcal{U}_V(\tau \doteq \rho, \sigma)$ with $\tau[\gamma/\alpha]\varsigma_{\mathrm{l}}\tilde{\sigma} \equiv \rho[\gamma/\beta]\varsigma_{\mathrm{r}}\tilde{\sigma}$ and $\sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and $V \,\#\, \mathrm{bound}(\tilde{\sigma})$.

With Lemma 4.22 it follows that $\mathrm{mod}(\tilde{\sigma}) \subseteq \mathrm{mod}(\hat{\sigma})$. So

$$\begin{aligned}
& (\forall\alpha.\tau)\varsigma_{\mathrm{l}}\tilde{\sigma} \\
\equiv\quad & \{\gamma \notin \mathrm{free}(\tau), \text{ Definition of } \equiv\} \\
& (\forall\gamma.\tau[\gamma/\alpha])\varsigma_{\mathrm{l}}\tilde{\sigma} \\
=\quad & \{\gamma \notin \mathrm{mod}(\varsigma_{\mathrm{l}}) \cup \mathrm{mod}(\tilde{\sigma}) \text{ and Lemma 2.3}\} \\
& \forall\gamma.(\tau[\gamma/\alpha]\varsigma_{\mathrm{l}}\tilde{\sigma}) \\
\equiv\quad & \{\tau[\gamma/\alpha]\varsigma_{\mathrm{l}}\tilde{\sigma} \equiv \rho[\gamma/\beta]\varsigma_{\mathrm{r}}\tilde{\sigma}\} \\
& \forall\gamma.(\rho[\gamma/\beta]\varsigma_{\mathrm{r}}\tilde{\sigma}) \\
=\quad & \{\gamma \notin \mathrm{mod}(\varsigma_{\mathrm{r}}) \cup \mathrm{mod}(\tilde{\sigma}) \text{ and Lemma 2.3}\} \\
& (\forall\gamma.\rho[\gamma/\beta])\varsigma_{\mathrm{r}}\tilde{\sigma} \\
\equiv\quad & \{\gamma \notin \mathrm{free}(\rho), \text{ Definition of } \equiv\} \\
& (\forall\beta.\rho)\varsigma_{\mathrm{r}}\tilde{\sigma}
\end{aligned}$$

Case $\mathcal{U}_V(\alpha \doteq \beta, \sigma) = \sigma$ where $\alpha \notin V$ and $\beta \notin V$.

With $V \mathbin{\#} \mathrm{mod}(\varsigma_{\mathrm{l}})$ and $V \mathbin{\#} \mathrm{mod}(\varsigma_{\mathrm{r}})$ it follows $\alpha\varsigma_{\mathrm{l}} \notin V$ and $\beta\varsigma_{\mathrm{r}} \notin V$. Note that we have $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\hat{\sigma}) \subseteq V$. So

$$\begin{aligned}
& \alpha\varsigma_{\mathrm{l}}\sigma \\
=\quad & \{\alpha\varsigma_{\mathrm{l}} \notin \mathrm{dom}(\sigma)\} \\
& \alpha\varsigma_{\mathrm{l}} \\
=\quad & \{\alpha\varsigma_{\mathrm{l}} \notin \mathrm{dom}(\hat{\sigma})\} \\
& \alpha\varsigma_{\mathrm{l}}\hat{\sigma} \\
\equiv\quad & \{\text{premise: } \alpha\varsigma_{\mathrm{l}}\hat{\sigma} \equiv \beta\varsigma_{\mathrm{r}}\hat{\sigma}\} \\
& \beta\varsigma_{\mathrm{r}}\hat{\sigma} \\
=\quad & \{\beta\varsigma_{\mathrm{r}} \notin \mathrm{dom}(\hat{\sigma})\} \\
& \beta\varsigma_{\mathrm{r}} \\
=\quad & \{\beta\varsigma_{\mathrm{r}} \notin \mathrm{dom}(\sigma)\} \\
& \beta\varsigma_{\mathrm{r}}\sigma
\end{aligned}$$

Hence all conclusions hold for $\tilde{\sigma} = \sigma$.

Case $\mathcal{U}_V(\gamma \doteq \tau, \sigma) = \mathcal{U}_V(\sigma(\gamma) \doteq \tau, \sigma)$ where $\gamma \in \mathrm{dom}(\sigma)$.

$$\begin{aligned}
& \sigma(\gamma)\hat{\sigma} \\[-1ex]
=\quad & \\[-1ex]
& \gamma\sigma\hat{\sigma} \\
\equiv\quad & \{\sigma \text{ idempotent and } \sigma \preceq_{\alpha} \hat{\sigma}\text{; Lemma 4.17}\} \\
& \gamma\hat{\sigma} \\
=\quad & \{\gamma \notin \mathrm{dom}(\varsigma_{\mathrm{l}}), \text{ because } \gamma \in \mathrm{dom}(\sigma) \subseteq V \text{ but } V \mathbin{\#} \mathrm{mod}(\varsigma_{\mathrm{l}})\} \\
& \gamma\varsigma_{\mathrm{l}}\hat{\sigma} \\
\equiv\quad & \{\text{premise}\} \\
& \tau\varsigma_{\mathrm{r}}\hat{\sigma}
\end{aligned}$$

We have $V \mathbin{\#} \mathrm{bound}(\sigma(\gamma))$, because $V \mathbin{\#} \mathrm{bound}(\sigma)$. The substitution σ is unchanged and the size of $\gamma\sigma$ and $\sigma(\gamma)\sigma = \gamma\sigma$ are equal, but the third component of the degree is smaller in the recursive call.

Hence by induction hypothesis there exists an idempotent $\tilde{\sigma} = \mathcal{U}_V(\sigma(\gamma) \doteq \tau, \sigma)$ with $\sigma(\gamma)\tilde{\sigma} \equiv \tau\varsigma_{\mathrm{r}}\tilde{\sigma}$ and $\sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and $V \# \mathrm{bound}(\tilde{\sigma})$.

Finally, it follows

$$\begin{array}{ll}
 & \gamma\varsigma_{\mathrm{l}}\tilde{\sigma} \\
= & \{\gamma \notin \mathrm{dom}(\varsigma_{\mathrm{l}}), \text{ because } \gamma \in \mathrm{dom}(\sigma) \subseteq V \text{ but } V \# \mathrm{mod}(\varsigma_{\mathrm{l}})\} \\
 & \gamma\tilde{\sigma} \\
\equiv & \{\sigma \text{ idempotent and } \sigma \preceq_\alpha \tilde{\sigma}; \text{ Lemma 4.17}\} \\
 & \gamma\sigma\tilde{\sigma} \\
\equiv & \\
 & \tau\varsigma_{\mathrm{r}}\tilde{\sigma}
\end{array}$$

Case $\mathcal{U}_V(\tau \doteq \gamma, \sigma) = \mathcal{U}_V(\tau \doteq \sigma(\gamma), \sigma)$ where $\gamma \in \mathrm{dom}(\sigma)$.
Similar to the previous case.

Case $\mathcal{U}_V(\gamma \doteq \tau, \sigma) = \sigma[\tau\sigma/\gamma]$ where $\gamma \in V \setminus \mathrm{dom}(\sigma)$.
Because $\tilde{\sigma} = \sigma[\tau\sigma/\gamma]$, $\sigma \preceq_\alpha \tilde{\sigma}$ holds, we have

$$\begin{array}{ll}
 & \gamma\hat{\sigma} \\
= & \{\gamma \notin \mathrm{dom}(\varsigma_{\mathrm{l}}), \text{ because } \gamma \in \mathrm{dom}(\sigma) \subseteq V \text{ but } V \# \mathrm{mod}(\varsigma_{\mathrm{l}})\} \\
 & \gamma\varsigma_{\mathrm{l}}\hat{\sigma} \\
\equiv & \{\text{premise}\} \\
 & \tau\varsigma_{\mathrm{r}}\hat{\sigma}
\end{array}$$

Suppose $\tau\varsigma_{\mathrm{r}} \neq \tau$. Then there exists $\alpha \in \mathrm{free}(\tau) \cap \mathrm{dom}(\varsigma_{\mathrm{r}})$. With $\mathrm{dom}(\hat{\sigma}) \subseteq V$ and $V \# \mathrm{mod}(\varsigma_{\mathrm{r}})$ it follows $\alpha\varsigma_{\mathrm{r}} \notin \mathrm{dom}(\hat{\sigma})$ and hence $\alpha\varsigma_{\mathrm{r}} \in \mathrm{free}(\tau\varsigma_{\mathrm{r}}\hat{\sigma})$. However, with $\gamma\hat{\sigma} \equiv \tau\varsigma_{\mathrm{r}}\hat{\sigma}$ it follows $\alpha\varsigma_{\mathrm{r}} \in \mathrm{free}(\hat{\sigma})$. This contradicts $\mathrm{free}(\varsigma_{\mathrm{r}}) \# \mathrm{free}(\hat{\sigma})$. So $\tau\varsigma_{\mathrm{r}} = \tau$ and $\gamma\hat{\sigma} \equiv \tau\hat{\sigma}$.

Case $\gamma \notin \mathrm{free}(\tau\sigma)$.
According to Corollary 4.19, $\tilde{\sigma}$ is idempotent. Also $V \# \mathrm{bound}(\tilde{\sigma})$, because $V \# \mathrm{bound}(\sigma)$ and $V \# \mathrm{bound}(\tau)$. Finally

$$\begin{array}{ll}
 & \gamma\varsigma_{\mathrm{l}}\tilde{\sigma} \\
= & \{\gamma \notin \mathrm{dom}(\varsigma_{\mathrm{l}}), \text{ because } \gamma \in \mathrm{dom}(\sigma) \subseteq V \text{ but } V \# \mathrm{mod}(\varsigma_{\mathrm{l}})\} \\
 & \gamma\tilde{\sigma} \\
= & \\
 & \gamma\sigma[\tau\sigma/\gamma] \\
= & \{\gamma \notin \mathrm{dom}(\sigma)\} \\
 & \gamma[\tau\sigma/\gamma] \\
= & \\
 & \tau\sigma \\
= & \{\gamma \notin \mathrm{free}(\tau\sigma)\} \\
 & \tau\sigma[\tau\sigma/\gamma] \\
= & \\
 & \tau\tilde{\sigma} \\
= & \{\tau = \tau\varsigma_{\mathrm{r}}\} \\
 & \tau\varsigma_{\mathrm{r}}\tilde{\sigma}
\end{array}$$

Case $\gamma \in \text{free}(\tau\sigma)$.

We have $\gamma\hat{\sigma} \equiv \tau\hat{\sigma} \equiv \tau\sigma\hat{\sigma}$. Considering the size of the two sides of the equation we see that together with $\gamma \in \text{free}(\tau\sigma)$ it follows that $\gamma = \tau\sigma$.

Hence $\tilde{\sigma} = \sigma[\tau\sigma/\gamma] = \sigma$ is obviously idempotent and we have $\gamma\tilde{\sigma} = \gamma\sigma[\tau\sigma/\gamma] = \tau\sigma = \tau\tilde{\sigma}$.

Case $\mathcal{U}_V(\gamma \doteq \tau, \sigma) = \sigma[\tau\sigma/\gamma]$ where $\gamma \in V \setminus \text{dom}(\sigma)$.

Similar to the previous case.

Otherwise

None of the equations that define $\mathcal{U}$ applies. This cannot occur because of the premise $\tau\varsigma_{\text{l}}\hat{\sigma} =_{\varsigma} \rho\varsigma_{\text{r}}\hat{\sigma}$. ■

4.3.3 Consistency of Types of Constructor Variables

In Chapter 3 we illustrated at the hand of several examples how type inference enables list abstraction. There is however a problem that we have not mentioned yet. Consider list abstracting the well-typed term

$$\{\texttt{xs}{:}\texttt{[[Int]]}\} \vdash \texttt{(:) [Int] ([] Int) xs} : \texttt{[[Int]]}$$

The first phase of list replacement gives (ignoring for the moment that it yields an instance substitution which indirectly defines the types of the constructor variables):

$$\{\text{xs}{:}\texttt{[[Int]]}, v^{\texttt{(:)}}_{\gamma_2}{:}\gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_2, v^{\texttt{[]}}_{\gamma_3}{:}\gamma_3\} \vdash v^{\texttt{(:)}}_{\gamma_2}\ \ v^{\texttt{[]}}_{\gamma_3}\ \texttt{xs}$$

Subsequently, type inference yields the principal typing

$$\{\text{xs}{:}\texttt{[[Int]]}, v^{\texttt{(:)}}_{\gamma_2}{:}\gamma_1 \rightarrow \texttt{[[Int]]} \rightarrow \texttt{[[Int]]}, v^{\texttt{[]}}_{\gamma_3}{:}\gamma_1\} \vdash v^{\texttt{(:)}}_{\gamma_2}\ \ v^{\texttt{[]}}_{\gamma_3}\ \texttt{xs} : \texttt{[[Int]]}$$

This typing is too general for our purposes. In the third phase we should reinstantiate $v^{\texttt{(:)}}_{\gamma_2}$ to `(:) [Int]`, because γ_2 was replaced by `[[Int]]`. We cannot do so, because the type of the first argument of $v^{\texttt{(:)}}_{\gamma_2}$ is γ_1 instead of the expected `[Int]`.

This too general result of the type inference algorithm is caused by the too general input to the type inference algorithm. In the first phase we replace a constructor application $\texttt{(:)}\ \tau$ of type $\tau \rightarrow \texttt{[}\tau\texttt{]} \rightarrow \texttt{[}\tau\texttt{]}$ by a variable $v^{\texttt{(:)}}_{\gamma}$ of type $\tau \rightarrow \gamma \rightarrow \gamma$. This type does not relate the types τ and γ in any way. Hence, if τ also contains some type variable γ', the type inference algorithm may instantiate γ' independently of how it instantiates γ.

This problem of too general types of constructor variables manifests itself also in another situation. Consider the well-typed term

$$\vdash \texttt{(:) [Int] ([] Int) ((:) [Int] ([] Int) ([] [Int]))} : \texttt{[[Int]]}$$

The first phase of list replacement gives

$$\begin{array}{l} \{v^{\texttt{(:)}}_{\gamma_2}{:}\gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_2\,,\ \ v^{\texttt{(:)}}_{\gamma_4}{:}\gamma_3 \rightarrow \gamma_4 \rightarrow \gamma_4\,,\ \ v^{\texttt{[]}}_{\gamma_5}{:}\gamma_5\,,\ \ v^{\texttt{[]}}_{\gamma_6}{:}\gamma_6\,,\ \ v^{\texttt{[]}}_{\gamma_7}{:}\gamma_7\} \\ \vdash\ v^{\texttt{(:)}}_{\gamma_2}\ \ v^{\texttt{[]}}_{\gamma_5}\ \ (v^{\texttt{(:)}}_{\gamma_4}\ \ v^{\texttt{[]}}_{\gamma_6}\ \ v^{\texttt{[]}}_{\gamma_7}) \end{array}$$

Subsequently, type inference yields the principal typing

$$\begin{array}{l} \{v^{\texttt{(:)}}_{\gamma_2}{:}\underline{\gamma_1} \rightarrow \gamma_2 \rightarrow \gamma_2\,,\ \ v^{\texttt{(:)}}_{\gamma_4}{:}\underline{\gamma_3} \rightarrow \gamma_2 \rightarrow \gamma_2\,,\ \ v^{\texttt{[]}}_{\gamma_5}{:}\gamma_1\,,\ \ v^{\texttt{[]}}_{\gamma_6}{:}\gamma_3\,,\ \ v^{\texttt{[]}}_{\gamma_7}{:}\gamma_2\} \\ \vdash\ v^{\texttt{(:)}}_{\gamma_2}\ \ v^{\texttt{[]}}_{\gamma_5}\ \ (v^{\texttt{(:)}}_{\gamma_4}\ \ v^{\texttt{[]}}_{\gamma_6}\ \ v^{\texttt{[]}}_{\gamma_7}) \\ :\ \ \ \gamma_2 \end{array}$$

This typing is too general, because both $v^{\texttt{(:)}}_{\gamma_2}$ and $v^{\texttt{(:)}}_{\gamma_4}$ construct values of type γ_2, but $v^{\texttt{(:)}}_{\gamma_2}$ requires an element of type γ_1 as first argument whereas $v^{\texttt{(:)}}_{\gamma_4}$ requires an element of type γ_2. Hence we are not able to merge $v^{\texttt{(:)}}_{\gamma_2}$ and $v^{\texttt{(:)}}_{\gamma_4}$ into a single constructor variable in the third phase of list abstraction. We would like to obtain the typing

$$\begin{array}{l}
\{v^{\texttt{(:)}}_{\gamma_2}{:}\gamma_1\!\rightarrow\!\gamma_2\!\rightarrow\!\gamma_2\texttt{,}\ v^{\texttt{(:)}}_{\gamma_4}{:}\gamma_1\!\rightarrow\!\gamma_2\!\rightarrow\!\gamma_2\texttt{,}\ v^{\texttt{[]}}_{\gamma_5}{:}\gamma_1\texttt{,}\ v^{\texttt{[]}}_{\gamma_6}{:}\gamma_1\texttt{,}\ v^{\texttt{[]}}_{\gamma_7}{:}\gamma_2\} \\
\vdash\ v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_5}\ \texttt{(}v^{\texttt{(:)}}_{\gamma_4}\ v^{\texttt{[]}}_{\gamma_6}\ v^{\texttt{[]}}_{\gamma_7}\texttt{)} \\
:\ \ \gamma_2
\end{array}$$

In [Chi99] we did not have the described problem, because we abstracted only a single list from a producer. The list replacement algorithm needed only to replace all list types $\texttt{[}\tau\texttt{]}$ and all constructor applications $\texttt{(:)}\ \tau$ and $\texttt{[]}\ \tau$ for a single fixed type τ. Because only nested lists expose the missing constraints in the types of constructor variables, the problem did not occur. However, to perform a worker/wrapper split we have to be able to abstract from several lists. Hence the list replacement algorithm replaces all list types $\texttt{[}\tau\texttt{]}$ and all constructor applications $\texttt{(:)}\ \tau$ and $\texttt{[]}\ \tau$ for (nearly) all types τ. Then nested lists lead to the problem.

From the two examples we learn that for a typing to be usable in the third phase, its typing environment should be consistent in the following sense.

Definition 4.25 A typing environment Γ is **consistent**, if for all constructor variables $v^{\texttt{(:)}}_1$ and $v^{\texttt{(:)}}_2$

$$\frac{\Gamma(v^{\texttt{(:)}}_1) = \tau\rightarrow\tau'\rightarrow\tau' \quad \tau' \text{ not a variable}}{\texttt{[}\tau\texttt{]} = \tau'} \quad \text{and}$$

$$\frac{\Gamma(v^{\texttt{(:)}}_1) = \tau_1\rightarrow\gamma\rightarrow\gamma \quad \Gamma(v^{\texttt{(:)}}_2) = \tau_2\rightarrow\gamma\rightarrow\gamma}{\tau_1 = \tau_2}$$

□

Our type inference algorithm does not yield a typing environment but a substitution σ (and a type). The types of the constructor variables are then determined by

$$\begin{array}{cl}
 & \mathrm{tyEnv}(\eta)\sigma \\
= & \{v^{\texttt{(:)}}_\gamma{:}\tau_\gamma\rightarrow\gamma\rightarrow\gamma,\ v^{\texttt{[]}}_\gamma{:}\gamma \mid \eta(\gamma) = \texttt{[}\tau_\gamma\texttt{]}, \gamma\in\mathrm{dom}(\eta)\}\sigma \\
= & \{v^{\texttt{(:)}}_\gamma{:}\tau_\gamma\sigma\rightarrow\gamma\sigma\rightarrow\gamma\sigma,\ v^{\texttt{[]}}_\gamma{:}\gamma\sigma \mid \eta(\gamma) = \texttt{[}\tau_\gamma\texttt{]}, \gamma\in\mathrm{dom}(\eta)\}
\end{array}$$

where η is the instance substitution from the list replacement phase. Hence we reformulate consistency as a property of substitutions:

Definition 4.26 A substitution σ is **consistent with** an instance substitution η, if the following three properties hold:

$$\frac{\sigma(\gamma) \text{ not a variable}}{\sigma(\gamma) = \eta(\gamma)\sigma} \tag{1}$$

$$\frac{\sigma(\gamma) = \gamma'}{\eta(\gamma)\sigma = \eta(\gamma')\sigma} \tag{2}$$

$$\sigma \preceq \eta^\infty \tag{3}$$

A typing (σ, τ) is **consistent with** an instance substitution η, if σ is consistent with η.

□

We added property (3) to the definition of consistency, because it simplifies some proofs. We know that the input of the type inference algorithm is typeable with η^∞ and want to find a more general typing. Hence we are only interested in typings with $\sigma \preceq \eta^\infty$ anyway.

Corollary 4.27

$$\frac{\sigma \textit{ idempotent} \quad \sigma \textit{ consistent with } \eta}{\sigma\eta^\infty = \eta^\infty} \qquad \square$$

Proof Follows with Lemma 4.17 from property (3) of Definition 4.26. ■

Lemma 4.28 *For any instance substitution η its compositional closure η^∞ is consistent with η.* □

Proof Straightforward. ■

As desired, the consistency of the substitution implies the consistency of the typing environment:

Lemma 4.29 (Consistency)

$$\frac{\sigma \textit{ consistent with } \eta}{\mathrm{tyEnv}(\eta)\sigma \textit{ consistent}} \qquad \square$$

Proof Follows directly from the definitions of tyEnv(η) and consistency. Property (1) of consistency of σ with η implies the first property of the consistency of tyEnv(η)σ and property (2) of consistency of σ with η implies the second property of the consistency of tyEnv(η)σ. ■

To illustrate the relationship between consistency of substitutions with an instance substitution and the consistency of a typing environment we consider again the typing

$$\{\texttt{xs:[[Int]]}\} \vdash \texttt{(:) [Int] ([] Int) xs : [[Int]]}$$

The list replacement phase returns the term $v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_3}$ xs and the instance substitution

$$\eta = [\texttt{[Int]}/\gamma_1, \texttt{[}\gamma_1\texttt{]}/\gamma_2, \texttt{[Int]}/\gamma_3]$$

This instance substitution η determines the following typing environment for the constructor variables:

$$\begin{array}{ll} & \mathrm{tyEnv}(\eta) \\ = & \{v^{\texttt{(:)}}_{\gamma_1}\texttt{:Int}\to\gamma_1\to\gamma_1, v^{\texttt{(:)}}_{\gamma_2}\texttt{:}\gamma_1\to\gamma_2\to\gamma_2, v^{\texttt{(:)}}_{\gamma_3}\texttt{:Int}\to\gamma_3\to\gamma_3, v^{\texttt{[]}}_{\gamma_1}\texttt{:}\gamma_1, v^{\texttt{[]}}_{\gamma_2}\texttt{:}\gamma_2, v^{\texttt{[]}}_{\gamma_3}\texttt{:}\gamma_3\}\end{array}$$

Finally, the principal typing of

$$\{\texttt{xs:[[Int]]}\} + \mathrm{tyEnv}(\eta) \vdash v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_3}\ \texttt{xs}$$

that is consistent with η is

$$([\texttt{[Int]}/\gamma_1, \texttt{[[Int]]}/\gamma_2, \texttt{[Int]}/\gamma_3],\ \texttt{[[Int]]})$$

Note that the substitution must replace γ_1 by `[Int]` to fulfil property (1) of consistency. Applying the substitution to tyEnv(η) yields

$$\{v^{(\mathtt{:})}_{\gamma_1}{:}\mathtt{Int}\to\mathtt{[Int]}\to\mathtt{[Int]},\, v^{(\mathtt{:})}_{\gamma_2}{:}\mathtt{[Int]}\to\mathtt{[[Int]]}\to\mathtt{[[Int]]},\\ v^{(\mathtt{:})}_{\gamma_3}{:}\mathtt{Int}\to\mathtt{[Int]}\to\mathtt{[Int]},\, v^{\mathtt{[]}}_{\gamma_1}{:}\mathtt{[Int]},\, v^{\mathtt{[]}}_{\gamma_2}{:}\mathtt{[[Int]]},\, v^{\mathtt{[]}}_{\gamma_3}{:}\mathtt{[Int]}\}$$

which is obviously a consistent typing environment.

To obtain a consistent typing for the third phase of list abstraction we post-process the possibly inconsistent result of type inference with an algorithm that we will describe in Section 4.3.6. However, we will already take advantage of the fact that we are only interested in consistent typings in our type inference algorithm, as we will explain in Section 4.3.5. We shortly discuss alternative approaches to ensuring consistency in Section 4.6.

4.3.4 Towards a Type Inference Algorithm

We want to define our type inference algorithm inductively over the input term. Unfortunately the type system of F as defined in Figure 2.6 is non-deterministic in that a typing can be proved by several derivation trees. For example, the following two derivation trees prove the same typing:

$$\dfrac{\dfrac{\dfrac{\{\mathtt{x}{:}\forall\alpha.\alpha\}(\mathtt{x}) = \forall\alpha.\alpha}{\{\mathtt{x}{:}\forall\alpha.\alpha\} \vdash \mathtt{x} : \forall\alpha.\alpha}\,\textsc{var}}{\vdash \lambda\mathtt{x}{:}\forall\alpha.\alpha.\mathtt{x} : (\forall\alpha.\alpha)\to(\forall\alpha.\alpha)}\,\textsc{term abs}}{\vdash \lambda\mathtt{x}{:}\forall\alpha.\alpha.\mathtt{x} : (\forall\alpha.\alpha)\to(\forall\beta.\beta)}\,\textsc{ty. ren.} \qquad \dfrac{\dfrac{\dfrac{\{\mathtt{x}{:}\forall\alpha.\alpha\}(\mathtt{x}) = \forall\alpha.\alpha}{\{\mathtt{x}{:}\forall\alpha.\alpha\} \vdash \mathtt{x} : \forall\alpha.\alpha}\,\textsc{var}}{\{\mathtt{x}{:}\forall\alpha.\alpha\} \vdash \mathtt{x} : \forall\beta.\beta}\,\textsc{type rename}}{\vdash \lambda\mathtt{x}{:}\forall\alpha.\alpha.\mathtt{x} : (\forall\alpha.\alpha)\to(\forall\beta.\beta)}\,\textsc{t. abs}$$

The cause of this non-determinism are the TYPE RENAME and TERM RENAME rule, which are applicable to terms of any form. However, these two rules do not change the form of terms either. Hence, although, for example, the last rule used in the derivation of a typing $\Gamma \vdash \lambda x : \tau_1.M : \tau$ may not be the TERM ABS rule, we can argue that there exists a type τ_2 such that $\Gamma + \{x : \tau_1\} \vdash M : \tau_2$ is valid and $\tau \equiv \tau_1 \to \tau_2$.

For every term form we can find such a generation property, which is basically the inverse of the appropriate typing rule except that types are only compared modulo renaming.

The only problematic case is type abstraction. A valid typing $\Gamma \vdash \lambda\alpha.M : \tau$ does *not* imply that there exists a type $\hat{\tau}$ such that $\Gamma \vdash M : \hat{\tau}$ is valid and $\tau \equiv \forall\alpha.\hat{\tau}$. For example, there is a derivation tree

$$\dfrac{\dfrac{\dfrac{\{\mathtt{x}{:}\alpha\}(\mathtt{x}) = \alpha}{\{\mathtt{x}{:}\alpha\} \vdash \mathtt{x} : \alpha}\,\textsc{var}}{\{\mathtt{x}{:}\alpha\} \vdash \lambda\beta.\mathtt{x} : \forall\beta.\alpha}\,\textsc{type abs}}{\{\mathtt{x}{:}\alpha\} \vdash \lambda\alpha.\mathtt{x} : \forall\beta.\alpha}\,\textsc{term rename}$$

and $\{\mathtt{x}{:}\alpha\} \vdash \mathtt{x} : \alpha$ is valid, but $\forall\beta.\alpha \not\equiv \forall\alpha.\alpha$. Because of the TERM RENAME rule the type variable in the abstraction ($\lambda\alpha$) need not coincide with the type variable in the universal quantification ($\forall\beta$). It is, however, the case that a valid typing $\Gamma \vdash \lambda\alpha.M : \tau$ implies that there exists a type $\hat{\tau}$ and a type variable β such that $\Gamma \vdash M[\beta/\alpha] : \hat{\tau}$ is valid and $\tau \equiv \forall\beta.\hat{\tau}$. The existence of such a type variable β is still not sufficient for the definition of the type inference algorithm, because the algorithm must know for which type variable β it shall infer the typing of Γ and $M[\beta/\alpha]$. Fortunately it is easy to prove that *any* type variable β that is not free in the typing environment Γ or the term $\lambda\alpha.M$ does the job.

$$\frac{\Gamma \vdash x \,:\, \tau}{\Gamma(x) \equiv \tau}\;\text{G-VAR}$$

$$\frac{\Gamma \vdash \lambda x : \tau_1.M \,:\, \tau}{\exists \tau_2 \quad \tau \equiv \tau_1 \to \tau_2 \quad \Gamma + \{x : \tau_1\} \vdash M \,:\, \tau_2}\;\text{G-TERM ABS}$$

$$\frac{\Gamma \vdash M_1\, M_2 \,:\, \tau}{\exists \tau_2 \quad \Gamma \vdash M_1 \,:\, \tau_2 \to \tau \quad \Gamma \vdash M_2 \,:\, \tau_2}\;\text{G-TERM APP}$$

$$\frac{\Gamma \vdash \texttt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^{k}\ \texttt{in}\ N \,:\, \tau}{\forall i \in \{1..k\} \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash M_i \,:\, \tau_i \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash N \,:\, \tau}\;\text{G-LET}$$

$$\frac{\Gamma \vdash c \,:\, \tau}{\Delta(c) \equiv \tau}\;\text{G-CONS}$$

$$\frac{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{c_i\, \overline{x}_i \mapsto N_i\}_{i=1}^{k} \,:\, \tau}{\Gamma \vdash M \,:\, T\,\overline{\rho} \quad \forall i \in \{1..k\} \quad \vdash c_i \,:\, \forall \overline{\alpha}.\overline{\rho}_i \to T\,\overline{\alpha} \quad \Gamma + \{\overline{x}_i : \overline{\rho}_i[\overline{\rho}/\overline{\alpha}]\} \vdash N_i \,:\, \tau}\;\text{G-CASE}$$

$$\frac{\Gamma \vdash \lambda\alpha.M \,:\, \tau \quad \beta \notin \mathrm{freeTy}(\Gamma) \cup \mathrm{freeTy}(\lambda\alpha.M)}{\exists \hat{\tau} \quad \tau \equiv \forall\beta.\hat{\tau} \quad \Gamma \vdash M[\beta/\alpha] \,:\, \hat{\tau}}\;\text{G-TYPE ABS}$$

$$\frac{\Gamma \vdash M\,\rho \,:\, \tau}{\exists \alpha, \hat{\tau} \quad \tau \equiv \hat{\tau}[\rho/\alpha] \quad \Gamma \vdash M \,:\, \forall\alpha.\hat{\tau}}\;\text{G-TYPE APP}$$

Figure 4.4: Generation of typings

Lemma 4.30 (Generation of typings) *The properties in Figure 4.4, which state how terms of a certain form get typed, hold.* □

PROOF We give proofs for some of the properties.

Case G-TERM ABS.
From

$$\Gamma \vdash \lambda x : \tau_1.M \,:\, \tau$$

follows with the type inference rules the existence of τ_1', M' and τ' such that $\tau_1' \equiv \tau_1$ and $M' \equiv M$ and

$$\Gamma \vdash \lambda x : \tau_1'.M' \,:\, \tau'$$

is derivable by the TERM ABS rule. Hence exists τ_2 with $\tau' = \tau_1' \to \tau_2$ and

$$\Gamma + x : \tau_1' \vdash M' \,:\, \tau_2$$

With the ENV RENAME rule of Lemma 2.5 and the TERM RENAME rule it follows

$$\Gamma + x : \tau_1 \vdash M \,:\, \tau_2$$

Finally $\tau \equiv \tau' = \tau_1' \to \tau_2 \equiv \tau_1 \to \tau_2$.

Case G-TYPE ABS.
From

$$\Gamma \vdash \lambda\alpha.M \ : \ \tau$$

follows with the type inference rules the existence of γ, M' and τ' such that $\gamma \notin \mathrm{free}(\lambda\alpha.M)$, $\lambda\alpha.M \equiv \lambda\gamma.M'[\gamma/\alpha]$ and

$$\Gamma \vdash \lambda\gamma.M'[\gamma/\alpha] \ : \ \tau'$$

is derivable by the TYPE ABS rule. Hence exists $\hat{\tau}'$ with $\tau' = \forall\gamma.\hat{\tau}'$ and

$$\Gamma \vdash M'[\gamma/\alpha] \ : \ \hat{\tau}'$$

and $\gamma \notin \mathrm{free}(\Gamma)$.

With the substitution property of Lemma 2.6 we obtain

$$\Gamma[\beta/\gamma] \vdash M'[\gamma/\alpha][\beta/\gamma] \ : \ \hat{\tau}'[\beta/\gamma]$$

Using the TERM RENAME rule we obtain

$$\Gamma[\beta/\gamma] \vdash M[\gamma/\alpha][\beta/\gamma] \ : \ \hat{\tau}'[\beta/\gamma]$$

We let $\hat{\tau} = \hat{\tau}'[\beta/\gamma]$ and with $\gamma \notin \mathrm{free}(\Gamma)$ and $\gamma \notin \mathrm{free}(\lambda\alpha.M)$ it follows

$$\Gamma \vdash M[\beta/\alpha] \ : \ \hat{\tau}$$

From $\Gamma \vdash \lambda\alpha.M \ : \ \forall\gamma.\hat{\tau}'$, $\beta \notin \mathrm{free}(\Gamma)$ and $\beta \notin \mathrm{free}(\lambda\alpha.M)$ follows according to Lemma 2.7, $\beta \notin \mathrm{free}(\forall\gamma.\hat{\tau}')$.

Hence $\tau \equiv \tau' = \forall\gamma.\hat{\tau}' \equiv \forall\beta.\hat{\tau}'[\beta/\gamma] = \forall\beta.\hat{\tau}$.

Case G-TYPE APP.
From

$$\Gamma \vdash M\,\rho \ : \ \tau$$

follows with the type inference rules the existence of M', ρ' and τ' such that $M' \equiv M$ and $\rho' \equiv \rho$ and

$$\Gamma \vdash M'\,\rho' \ : \ \tau'$$

is derivable by the TYPE APP rule. Hence exist α and $\hat{\tau}$ with $\tau' = \hat{\tau}[\rho'/\alpha]$ and

$$\Gamma \vdash M' \ : \ \forall\alpha.\hat{\tau}$$

With the TERM RENAME rule it follows

$$\Gamma \vdash M \ : \ \forall\alpha.\hat{\tau}$$

and finally $\tau' = \hat{\tau}[\rho'/\alpha] \equiv \hat{\tau}[\rho/\alpha]$. ■

4.3.5 The Type Inference Algorithm $\mathcal{T}$

The starting point for the development of our type inference algorithm $\mathcal{T}$ was Milner's type inference algorithm $\mathcal{W}$ for the Hindley-Milner type system [Mil78, DM82]. We formulated the generation lemma mainly to find out how to handle type abstraction and type application, which do not occur in the Hindley-Milner type system. In fact, algorithm $\mathcal{W}$ can also be considered as based on the appropriate generation lemma for the Hindley-Milner type system.

Note that in contrast to $\mathcal{W}$ the terms that are input to our type inference algorithm contain type annotations. These types and the types in the typing environment contain type inference variables which are to be replaced to obtain a valid typing. Furthermore, the type inference algorithm will only be applied to input which is typeable by a substitution which only substitutes list types. Finally, we are only interested in typings which are consistent. All these properties together permit us to define a type inference algorithm that is simpler than $\mathcal{W}$ in many aspects. The most obvious simplification is that our type inference algorithm does not need to create any new type inference variables. The type inference variables that occur in the input are sufficient.

Our type inference algorithm $\mathcal{T}$ is given in Figure 4.5. Its first argument is the instance substitution η. It is needed for the typing of constructor variables, its domain is needed for unification, and type inference of the `case` construct is simplified by using η. Besides a typing environment Γ and a term M, the algorithm also gets an idempotent substitution σ as input. Its purpose is the same as in the unification algorithm. The type inference algorithm works by stepwise extension of σ to obtain a typing for $\Gamma\sigma$ and $M\sigma$. The use of this separate substitution makes the algorithm more readable and is close to our implementation. The algorithm returns a valid typing, that is, a substitution $\tilde{\sigma}$ and a type $\tilde{\tau}$ such that $(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash M\tilde{\sigma} : \tilde{\tau}\tilde{\sigma}$.

The type inference algorithm is defined inductively over the form of the input term and recursively traverses the input term. The result of the algorithm for constructor variables, term variables and data constructors is directly determined by the typing environment for constructor variables $\mathrm{tyEnv}(\eta)$, the typing environment Γ and the data typing environment Δ, respectively.

To determine the typing of a term abstraction $\lambda x : \tau_1.M$, the algorithm recursively determines the typing of the abstraction body M. Note that the typing environment has to be extended by a type for the abstracted term variable, which is directly obtained from the type annotation of the term abstraction $x : \tau$.

To determine the typing of a term application $M_1\, M_2$, the algorithm recursively determines the typings of its subterms M_1 and M_2. Because the input of the algorithm is typeable by a substitution that only substitutes list types, the typing of M_1 must contain a function type $\tau_2' \to \tau_1$. Its argument type τ_2' and the inferred type of the argument M_2 finally have to be unified, because they have to agree.

The type system of F has explicit type abstraction and hence no type generalisation step (type closure) at the `let` construct, as the Hindley-Milner type system has. So our type inference algorithm is closer to Hindley's original type inference algorithm [Hin69] than to the Hindley-Milner type inference algorithm. For the `let` construct the definition of $\mathcal{T}$ is simply a combination of its definition for term application and term abstraction.

The definition of $\mathcal{T}$ for the `case` construct is best explained at the hand of an example. Consider the typing environment and term $\emptyset \vdash \lambda$`x:[`α`].case x of {y:ys` $\mapsto$ `y}`. For this input $\mathcal{R}$ yields (`[[`α`]/`τ_1`]`, λ`x:`τ_1`.case x of {y:ys` $\mapsto$ `y}`). The type inference algorithm determines that the variable `x`, which is scrutinised in the `case` construct, has type τ_1. Furthermore, the pattern `y:ys` conveys the information that the type of `x` must

$$\mathcal{T}_\eta(\Gamma, v_\gamma^{\mathtt{(:)}}, \sigma) := (\sigma, \tau_\gamma \to \gamma \to \gamma) \text{ where } \mathtt{[}\tau_\gamma\mathtt{]} = \eta(\gamma)$$

$$\mathcal{T}_\eta(\Gamma, v_\gamma^{\mathtt{[]}}, \sigma) := (\sigma, \gamma)$$

$$\mathcal{T}_\eta(\Gamma, x, \sigma) := (\sigma, \Gamma(x)) \quad \text{, if } x \text{ not a constructor variable}$$

$$\begin{array}{rl}
\mathcal{T}_\eta(\Gamma, \lambda x : \tau_1.M, \sigma) := & \text{let } (\tilde{\sigma}, \tau_2) := \mathcal{T}_\eta(\Gamma + \{x : \tau_1\}, M, \sigma) \\
& \text{in } \ (\tilde{\sigma}, \tau_1 \to \tau_2)
\end{array}$$

$$\begin{array}{rl}
\mathcal{T}_\eta(\Gamma, M_1\, M_2, \sigma) := & \text{let } (\sigma_1, \tau_2' \to \tau_1) := \mathcal{T}_\eta(\Gamma, M_1, \sigma) \\
& \phantom{\text{let }} (\sigma_2, \tau_2) := \mathcal{T}_\eta(\Gamma, M_2, \sigma_1) \\
& \text{in } \ (\mathcal{U}_{\text{dom}(\eta)}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1)
\end{array}$$

$$\begin{array}{rl}
\mathcal{T}_\eta(\Gamma, \mathtt{let}\ \{x_i : \tau_i = M_i\}_{i=1}^k\ \mathtt{in}\ N, \sigma) := & \text{let } (\sigma_0, \tau) := \mathcal{T}_\eta(\Gamma + \{x_i : \tau_i\}_{i=1}^k, N, \sigma) \\
& \phantom{\text{let }} (\sigma_1, \tau_1') := \mathcal{T}_\eta(\Gamma + \{x_i : \tau_i\}_{i=1}^k, M_1, \sigma_0) \\
& \phantom{\text{let }} \quad \vdots \qquad\qquad \vdots \\
& \phantom{\text{let }} (\sigma_k, \tau_k') := \mathcal{T}_\eta(\Gamma + \{x_i : \tau_i\}_{i=1}^k, M_k, \sigma_{k-1}) \\
& \text{in } \ (\mathcal{U}_{\text{dom}(\eta)}(\tau_1 \doteq \tau_1', ..\mathcal{U}_{\text{dom}(\eta)}(\tau_k \doteq \tau_k', \sigma_k)..), \tau)
\end{array}$$

$$\mathcal{T}_\eta(\Gamma, c, \sigma) := (\sigma, \Delta(c))$$

$$\begin{array}{rl}
\mathcal{T}_\eta(\Gamma, \mathtt{case}\ M\ \mathtt{of}\ \{c_i\, \overline{x}_i \mapsto N_i\}_{i=1}^k, \sigma) := & \text{let } \forall\alpha.\overline{\rho}_i \to C\,\overline{\alpha} := \Delta(c_i) \qquad \forall i = 1..k \\
& \phantom{\text{let }} (\sigma_0', \tau') := \mathcal{T}_\eta(\Gamma, M, \sigma) \\
& \phantom{\text{let }} (\sigma_0, C\,\overline{\tau}) := \begin{cases} (\sigma_0'[\gamma\eta\sigma_0'/\gamma], \gamma\eta\sigma_0') \\ \quad \text{, if } \tau' = \gamma \in \text{dom}(\eta) \\ (\sigma_0', \tau') \quad \text{, otherwise} \end{cases} \\
& \phantom{\text{let }} (\sigma_1, \tau_1) := \mathcal{T}_\eta(\Gamma + \{\overline{x}_1 : \overline{\rho}_1[\overline{\tau}/\overline{\alpha}]\}, N_1, \sigma_0) \\
& \phantom{\text{let }} \quad \vdots \qquad\qquad \vdots \\
& \phantom{\text{let }} (\sigma_k, \tau_k) := \mathcal{T}_\eta(\Gamma + \{\overline{x}_k : \overline{\rho}_k[\overline{\tau}/\overline{\alpha}]\}, N_k, \sigma_{k-1}) \\
& \text{in } \ (\mathcal{U}_{\text{dom}(\eta)}(\tau_1 \doteq \tau_2, ..\mathcal{U}_{\text{dom}(\eta)}(\tau_{k-1} \doteq \tau_k, \sigma_k)..), \tau_k)
\end{array}$$

$$\begin{array}{rl}
\mathcal{T}_\eta(\Gamma, \lambda\alpha.M, \sigma) := & \text{let } \beta \text{ new with} \\
& \phantom{\text{let }} \beta \notin \text{free}(\Gamma) \cup \text{free}(\lambda\alpha.M) \cup \text{mod}(\eta) \\
& \phantom{\text{let }} (\tilde{\sigma}, \tau) := \mathcal{T}_\eta(\Gamma, M[\beta/\alpha], \sigma) \\
& \text{in } \ (\tilde{\sigma}, \forall\beta.\tau)
\end{array}$$

$$\begin{array}{rl}
\mathcal{T}_\eta(\Gamma, M\,\rho, \sigma) := & \text{let } (\tilde{\sigma}, \forall\alpha.\tau) := \mathcal{T}_\eta(\Gamma, M, \sigma) \\
& \text{in } \ (\tilde{\sigma}, \text{subs}_{\text{dom}(\eta)}(\tau, [\rho/\alpha]))
\end{array}$$

Figure 4.5: Type inference algorithm

be a list type. However, how shall the algorithm know the type of the list elements? We could define $\mathcal{T}$ so that it introduces a new type variable τ_2 for the list elements. Then it would infer the principal typing $([\mathtt{[}\gamma_2\mathtt{]}/\gamma_1], \mathtt{[}\gamma_2\mathtt{]} \to \gamma_2)$ However, we are not interested in the fact that the term is polymorphic in the list argument type. We only want to know about polymorphism in list types to abstract them. Formally, we are only interested in typings whose substitutions are consistent with the instance substitution $[\texttt{[}\alpha\texttt{]}/\tau_1]$. So, because γ_1 has to be replaced by a list because of the pattern `y:ys`, we can replace it by $\tau_1[\texttt{[}\alpha\texttt{]}/\tau_1] = \texttt{[}\alpha\texttt{]}$ and obtain the principal consistent typing $([\texttt{[}\alpha\texttt{]}/\gamma_1], \texttt{[}\alpha\texttt{]} \rightarrow \alpha)$. In general, we use the instance substitution η when we have to replace a type inference variable during type inference of a `case` construct, because thus we avoid the introduction of a new type inference variable and because we are only interested in consistent typings anyway.

The definition of $\mathcal{T}$ for type abstraction follows from the Generation Lemma 4.30. Note that the new variable β is not a type inference variable, that is, β is never replaced by unification.

The definition of $\mathcal{T}$ for a type application $M\,\rho$ also follows from the Generation Lemma 4.30. Because the input of the algorithm is typeable by a substitution that only substitutes list types, the inferred typing of M must contain a universal type $\forall\alpha.\tau$. To obtain the result type, α has to be replaced by ρ, but note that this substitution is not part of the substitution of the typing. The type variable α is not a type inference variable. To use our unification algorithm $\mathcal{U}$ we have to preserve the invariant that the set of type inference variables $\mathrm{dom}(\eta)$ is disjoint from the set of all bound type variables occurring in the processed typing environment and term. Hence we use a slightly modified version of substitution that never introduces a type inference variable as new variable:

Definition 4.31 (Application of a substitution excluding some new variables)

$$\begin{aligned}
\mathrm{subs}_V(\alpha,\sigma) &:= \sigma(\alpha)\\
\mathrm{subs}_V(\forall\alpha.\tau,\sigma) &:= \forall\beta.\,\mathrm{subs}_V(\tau,[\beta/\alpha]\sigma)\\
&\qquad \text{where } \beta \text{ new with } \beta \notin V \cup \mathrm{free}(\forall\alpha.\tau) \cup \mathrm{mod}(\sigma)\\
\mathrm{subs}_V(T\,\tau_1..\tau_k,\sigma) &:= T\,(\mathrm{subs}_V(\tau_1,\sigma))..(\mathrm{subs}_V(\tau_k,\sigma))\\
\mathrm{subs}_V(\tau_1 \rightarrow \tau_2,\sigma) &:= (\mathrm{subs}_V(\tau_1,\sigma)) \rightarrow (\mathrm{subs}_V(\tau_2,\sigma))
\end{aligned}$$

□

Lemma 4.32 (Properties of subs)

$$\mathrm{subs}_V(\tau,\sigma) \equiv \tau\sigma \qquad\qquad \frac{V \,\#\, \mathrm{bound}(\tau) \quad V \,\#\, \mathrm{bound}(\sigma)}{V \,\#\, \mathrm{bound}(\mathrm{subs}_V(\tau,\sigma))}$$

□

Proof Both by induction on τ. ■

To illustrate how the type inference algorithm works, Figure 4.6 shows intermediate steps of the computation of

$$\mathcal{T}_\eta(\{\texttt{zs:[}\beta\texttt{]}\}, \texttt{(}\lambda\alpha.\lambda\texttt{x:}\alpha\texttt{.x)}\ \gamma\ \texttt{zs}, [\,]) = ([\texttt{[}\beta\texttt{]}/\gamma], \gamma)$$

with $\eta = [\texttt{[}\beta\texttt{]}/\gamma]$.

Finally, the following lemma states exactly under which conditions $\mathcal{T}$ computes a typing and in which respect it is principal (most general). Note that the result of $\mathcal{T}$ is computed independently of the typing $(\hat{\sigma},\hat{\tau})$. So the result is more general than any typing $(\hat{\sigma},\hat{\tau})$ that fulfils the given premises.

$$
\begin{array}{ll}
 & \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{(}\lambda\alpha\mathtt{.}\lambda\mathtt{x}{:}\alpha\mathtt{.x)}\ \gamma\ \mathtt{zs}, [\,]) \\
= & \text{let } (\sigma_1, \tau_2' \to \tau_1) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{(}\lambda\alpha\mathtt{.}\lambda\mathtt{x}{:}\alpha\mathtt{.x)}\ \gamma, [\,]) \\
 & \quad\ (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, \sigma_1) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1) \\
= & \text{let } (\sigma_1, \tau_2' \to \tau_1) := \text{let } (\tilde{\sigma}, \forall\alpha.\tau) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \lambda\alpha\mathtt{.}\lambda\mathtt{x}{:}\alpha\mathtt{.x}, [\,]) \\
 & \qquad\qquad\qquad\qquad\quad \text{in } \ (\tilde{\sigma}, \text{subs}_{\{\gamma\}}(\tau, [\gamma/\alpha])) \\
 & \quad\ (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, \sigma_1) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1) \\
= & \text{let } (\sigma_1, \tau_2' \to \tau_1) := \text{let } (\tilde{\sigma}, \forall\alpha.\tau) := \text{let } (\tilde{\sigma}, \tau) := \text{let } (\tilde{\sigma}, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}, \mathtt{x}{:}\delta\}, \mathtt{x}, [\,]) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } \ (\tilde{\sigma}, \delta \to \tau_2) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } \ (\tilde{\sigma}, \forall\delta.\tau) \\
 & \qquad\qquad\qquad\qquad\quad \text{in } \ (\tilde{\sigma}, \text{subs}_{\{\gamma\}}(\tau, [\gamma/\alpha])) \\
 & \quad\ (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, \sigma_1) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1) \\
= & \text{let } (\sigma_1, \tau_2' \to \tau_1) := \text{let } (\tilde{\sigma}, \forall\alpha.\tau) := \text{let } (\tilde{\sigma}, \tau) := ([\,], \delta \to \delta) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } \ (\tilde{\sigma}, \forall\delta.\tau) \\
 & \qquad\qquad\qquad\qquad\quad \text{in } \ (\tilde{\sigma}, \text{subs}_{\{\gamma\}}(\tau, [\gamma/\alpha])) \\
 & \quad\ (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, \sigma_1) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1) \\
= & \text{let } (\sigma_1, \tau_2' \to \tau_1) := \text{let } (\tilde{\sigma}, \forall\alpha.\tau) := ([\,], \forall\delta.\delta \to \delta) \\
 & \qquad\qquad\qquad\qquad\quad \text{in } \ (\tilde{\sigma}, \text{subs}_{\{\gamma\}}(\tau, [\gamma/\alpha])) \\
 & \quad\ (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, \sigma_1) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1) \\
= & \text{let } (\sigma_1, \tau_2' \to \tau_1) := ([\,], \text{subs}_{\{\gamma\}}(\delta \to \delta, [\gamma/\delta])) \\
 & \quad\ (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, \sigma_1) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1) \\
= & \text{let } (\sigma_1, \tau_2' \to \tau_1) := ([\,], \gamma \to \gamma) \\
 & \quad\ (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, \sigma_1) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\tau_2' \doteq \tau_2, \sigma_2), \tau_1) \\
= & \text{let } (\sigma_2, \tau_2) := \mathcal{T}_\eta(\{\mathtt{zs}{:}\mathtt{[}\beta\mathtt{]}\}, \mathtt{zs}, [\,]) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\gamma \doteq \tau_2, \sigma_2), \gamma) \\
= & \text{let } (\sigma_2, \tau_2) := ([\,], \mathtt{[}\beta\mathtt{]}) \\
 & \text{in } \ (\mathcal{U}_{\{\gamma\}}(\gamma \doteq \tau_2, \sigma_2), \gamma) \\
= & (\mathcal{U}_{\{\gamma\}}(\gamma \doteq \mathtt{[}\beta\mathtt{]}, [\,]), \gamma) \\
= & ([\mathtt{[}\beta\mathtt{]}/\gamma], \gamma)
\end{array}
$$

Figure 4.6: An example computation of $\mathcal{T}$

Lemma 4.33 (Principal Typing)

$$\frac{\begin{array}{c}(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash M\hat{\sigma} : \hat{\tau} \quad \sigma \preceq_\alpha \hat{\sigma} \quad \sigma \textit{ idempotent} \\ \hat{\sigma} \textit{ consistent with } \eta \quad \mathrm{dom}(\hat{\sigma}) \subseteq V \quad V = \mathrm{dom}(\eta) \quad V \mathbin{\#} \mathrm{bound}(\eta) \\ V \mathbin{\#} \mathrm{bound}(\Delta) \quad V \mathbin{\#} \mathrm{bound}(\Gamma) \quad V \mathbin{\#} \mathrm{bound}(M) \quad V \mathbin{\#} \mathrm{bound}(\sigma)\end{array}}{\exists(\tilde{\sigma}, \tilde{\tau}) \quad \begin{array}{l}(\tilde{\sigma}, \tilde{\tau}) = \mathcal{T}_\eta(\Gamma, M, \sigma) \quad (\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash M\tilde{\sigma} : \tilde{\tau}\tilde{\sigma} \quad \tilde{\tau}\hat{\sigma} \equiv \hat{\tau} \\ \sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma} \quad \tilde{\sigma} \textit{ idempotent} \quad V \mathbin{\#} \mathrm{bound}(\tilde{\tau}) \quad V \mathbin{\#} \mathrm{bound}(\tilde{\sigma})\end{array}}$$

□

Proof We prove by structural induction on the term M.

Case $v_\gamma^{(:)}$.
From

$$(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash v_\gamma^{(:)}\hat{\sigma} : \hat{\tau}$$

it follows with property G-VAR that $((\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma})(v_\gamma^{(:)}) \equiv \hat{\tau}$.

So $\hat{\tau} \equiv (\tau_\gamma \to \gamma \to \gamma)\hat{\sigma}$ where $[\tau_\gamma] = \eta(\gamma)$. Then the VAR rule assures that

$$(\Gamma + \mathrm{tyEnv}(\eta))\sigma \vdash v_\gamma^{(:)}\sigma : (\tau_\gamma \to \gamma \to \gamma)\sigma$$

Finally $V \mathbin{\#} \mathrm{bound}(\eta)$ implies $V \mathbin{\#} \mathrm{bound}(\tau_\gamma \to \gamma \to \gamma)$. Hence all conclusions hold for $(\sigma, \tau_\gamma \to \gamma \to \gamma)$.

Case $v_\gamma^{[]}$.
Similar to the previous case.

Case x.
From

$$(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash x\hat{\sigma} : \hat{\tau}$$

it follows with property G-VAR that $((\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma})(x) \equiv \hat{\tau}$, that is, $\Gamma\hat{\sigma}(x) \equiv \hat{\tau}$. So $x \in \mathrm{dom}(\Gamma)$ and hence the VAR rule assures that

$$(\Gamma + \mathrm{tyEnv}(\eta))\sigma \vdash x\sigma : (\Gamma(x))\sigma$$

Finally $V \mathbin{\#} \mathrm{bound}(\Gamma)$ implies $V \mathbin{\#} \mathrm{bound}(\Gamma(x))$. Hence all conclusions hold for $(\sigma, \Gamma(x))$.

Case $\lambda x : \tau_1.M$.
From

$$(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash (\lambda x : \tau_1.M)\hat{\sigma} : \hat{\tau}$$

it follows

$$(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash \lambda x : \tau_1\hat{\sigma}.M\hat{\sigma} : \hat{\tau}$$

and hence with property G-TERM ABS the existence of $\hat{\tau}_2$ with $\hat{\tau} \equiv \tau_1\hat{\sigma} \to \hat{\tau}_2$ and

$$(\Gamma + \{x{:}\tau_1\} + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash M\hat{\sigma} : \hat{\tau}_2$$

$V \mathbin{\#} \mathrm{bound}(\lambda x : \tau_1.M)$ implies $V \mathbin{\#} \mathrm{bound}(M)$ and $V \mathbin{\#} \mathrm{bound}(\tau_1)$. The latter together with $V \mathbin{\#} \mathrm{bound}(\Gamma)$ implies $V \mathbin{\#} \mathrm{bound}(\Gamma + \{x{:}\tau_1\})$.

Hence by induction hypothesis there exists $(\tilde{\sigma}, \tau_2) = \mathcal{T}_\eta(\Gamma + \{x{:}\tau_1\}, M, \sigma)$ with

$$(\Gamma + \{x{:}\tau_1\} + \text{tyEnv}(\eta))\tilde{\sigma} \vdash M\tilde{\sigma} : \tau_2\tilde{\sigma}$$

and $\sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and $\tilde{\sigma}$ is idempotent and $V \# \text{bound}(\tau_2)$ and $V \# \text{bound}(\tilde{\sigma})$. With the TERM ABS rule it follows

$$(\Gamma + \text{tyEnv}(\eta))\tilde{\sigma} \vdash \lambda x : (\tau_1\tilde{\sigma}).(M\tilde{\sigma}) : \tau_1\tilde{\sigma} \rightarrow \tau_2\tilde{\sigma}$$

that is,

$$(\Gamma + \text{tyEnv}(\eta))\tilde{\sigma} \vdash (\lambda x : \tau_1.M)\tilde{\sigma} : (\tau_1 \rightarrow \tau_2)\tilde{\sigma}$$

Furthermore, $(\tau_1 \rightarrow \tau_2)\hat{\sigma} = \tau_1\hat{\sigma} \rightarrow \tau_2\hat{\sigma} \equiv \tau_1\hat{\sigma} \rightarrow \hat{\tau}_2 \equiv \hat{\tau}\hat{\sigma}$. Finally, $V \# \text{bound}(\tau_1)$ and $V \# \text{bound}(\tau_2)$ imply $V \# \text{bound}(\tau_1 \rightarrow \tau_2)$.

Hence all conclusions hold for $(\tilde{\sigma}, \tau_1 \rightarrow \tau_2)$.

Case $M_1\, M_2$.
From

$$(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash (M_1\, M_2)\hat{\sigma} : \hat{\tau}$$

it follows

$$(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash (M_1\hat{\sigma})\,(M_2\hat{\sigma}) : \hat{\tau}$$

and hence with property G-TERM APP the existence of $\hat{\tau}_2$ with

$$\begin{aligned}&(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash M_1\hat{\sigma} : \hat{\tau}_2 \rightarrow \hat{\tau} \quad \text{and}\\ &(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash M_2\hat{\sigma} : \hat{\tau}_2\end{aligned}$$

From $V \# \text{bound}(M_1\, M_2)$ it follows $V \# \text{bound}(M_1)$ and $V \# \text{bound}(M_2)$.

Hence by induction hypothesis there exists $(\sigma_1, \tilde{\tau}_1) = \mathcal{T}_\eta(\Gamma, M_1, \sigma)$ with

$$(\Gamma + \text{tyEnv}(\eta))\sigma_1 \vdash M_1\sigma_1 : \tilde{\tau}_1\sigma_1$$

and $\tilde{\tau}_1\hat{\sigma} \equiv \hat{\tau}_2 \rightarrow \hat{\tau}$ and $\sigma \preceq_\alpha \sigma_1 \preceq_\alpha \hat{\sigma}$ and σ_1 idempotent and $V \# \text{bound}(\tilde{\tau}_1)$ and $V \# \text{bound}(\sigma_1)$.

Suppose $\tilde{\tau}_1$ is a variable. Because $\hat{\sigma} \preceq_\alpha \eta^\infty$ and $\tilde{\tau}_1\hat{\sigma} \equiv \hat{\tau}_2 \rightarrow \hat{\tau}$, $\eta^\infty(\tilde{\tau}_1)$ must be a function type. However, this contradicts with the fact that η substitutes list types. So $\tilde{\tau}_1$ is not a variable and there exist τ_2' and τ_1 with $\tau_2' \rightarrow \tau_1 = \tilde{\tau}_1$.

By induction hypothesis there exists $(\sigma_2, \tau_2) = \mathcal{T}_\eta(\Gamma, M_2, \sigma_1)$ with

$$(\Gamma + \text{tyEnv}(\eta))\sigma_2 \vdash M_2\sigma_2 : \tau_2\sigma_2$$

and $\tau_2\hat{\sigma} \equiv \hat{\tau}_2$ and $\sigma_1 \preceq_\alpha \sigma_2 \preceq_\alpha \hat{\sigma}$ and σ_2 idempotent and $V \# \text{bound}(\tau_2)$ and $V \# \text{bound}(\sigma_2)$.

From $\tau_2'\hat{\sigma} \rightarrow \tau_1\hat{\sigma} = (\tau_2' \rightarrow \tau_1)\hat{\sigma} = \tilde{\tau}_1\hat{\sigma} \equiv \hat{\tau}_2 \rightarrow \hat{\tau} \equiv \tau_2\hat{\sigma} \rightarrow \hat{\tau}$ it follows $\tau_2'\hat{\sigma} \equiv \tau_2\hat{\sigma}$ and $\tau_1\hat{\sigma} \equiv \tau$. Also $V \# \text{bound}(\tilde{\tau}_1)$ together with $\tilde{\tau}_1 = \tau_2' \rightarrow \tau_1$ imply $V \# \text{bound}(\tau_2')$.

Hence according to Lemma 4.24 there exists a most general unifier, that is, there exists an idempotent $\tilde{\sigma} = \mathcal{U}_{\text{dom}\,\eta}(\tau_2' \doteq \tau_2, \sigma_2)$ with $\tau_2'\tilde{\sigma} = \tau_2\tilde{\sigma}$ and $\sigma_2 \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and $V \# \text{bound}(\tilde{\sigma})$.

With $\sigma_1 \preceq_\alpha \sigma_2 \preceq_\alpha \tilde{\sigma}$ and Lemma 2.6 it follows

$$(\Gamma + \text{tyEnv}(\eta))\tilde{\sigma} \vdash M_1\tilde{\sigma} : (\tau_2' \rightarrow \tau_1)\tilde{\sigma} \quad \text{and}$$
$$(\Gamma + \text{tyEnv}(\eta))\tilde{\sigma} \vdash M_2\tilde{\sigma} : \tau_2\tilde{\sigma}$$

Then the TERM APP rule and $\tau_2'\tilde{\sigma} = \tau_2\tilde{\sigma}$ give us

$$(\Gamma + \text{tyEnv}(\eta))\tilde{\sigma} \vdash (M_1\tilde{\sigma})\,(M_2\tilde{\sigma}) : \tau_1\tilde{\sigma}$$

that is,

$$(\Gamma + \text{tyEnv}(\eta))\tilde{\sigma} \vdash (M_1\, M_2)\tilde{\sigma} : \tau_1\tilde{\sigma}$$

Hence all conclusions hold for $(\tilde{\sigma}, \tau_1)$.

Case `let` $\{x_i : \tau_i = M_i\}_{i=1}^k$ `in` N.
Similar to the case $\lambda x : \tau_1.M$.

Case c.
Similar to the case x. Note that $V \# \text{bound}(\Delta(c))$, because $V \# \text{bound}(\Delta)$.

Case `case` M `of` $\{c_i\, \overline{x}_i \mapsto N_i\}_{i=1}^k$.
Similar to the case $\lambda x : \tau_1.M$. To justify the use of η in the determination of the type of the scrutinised term we need that $\hat{\sigma}$ is consistent with η.

Case $\lambda\alpha.M$.
From $\hat{\sigma} \preceq_\alpha \eta^\infty$ it follows $\text{mod}(\hat{\sigma}) \subseteq \text{mod}(\eta^\infty) = \text{mod}(\eta)$. Together with $\beta \notin \text{free}(\Gamma) \cup \text{mod}(\eta)$ it follows $\beta \notin \text{free}((\Gamma + \text{tyEnv}(\eta))\hat{\sigma}$.

From

$$(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash (\lambda\alpha.M)\hat{\sigma} : \hat{\tau}$$

it follows with $\beta \notin \text{free}(\lambda\alpha.M)$ and $\beta \notin \text{mod}(\hat{\sigma}) \subseteq \text{mod}(\eta)$

$$(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash \lambda\beta.M[\beta/\alpha]\hat{\sigma} : \hat{\tau}$$

Hence property G-TYPE ABS assures the existence of $\hat{\tau}'$ with $\hat{\tau} \equiv \forall\beta.\hat{\tau}'$ and

$$(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash M[\beta/\alpha]\hat{\sigma}[\beta/\beta] : \hat{\tau}'$$

that is,

$$(\Gamma + \text{tyEnv}(\eta))\hat{\sigma} \vdash M[\beta/\alpha]\hat{\sigma} : \hat{\tau}'$$

From $V \# \text{bound}(\lambda\alpha.M)$ it follows $V \# \text{bound}(M[\beta/\alpha])$.

Hence by induction hypothesis there exists $(\tilde{\sigma}, \tau) = \mathcal{T}_\eta(\Gamma, M[\beta/\alpha], \sigma)$ with

$$(\Gamma + \text{tyEnv}(\eta))\tilde{\sigma} \vdash M[\beta/\alpha]\tilde{\sigma} : \tau\tilde{\sigma}$$

and $\tau\hat{\sigma} \equiv \hat{\tau}'$ and $\sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and $\tilde{\sigma}$ is idempotent and $V \# \text{bound}(\tau)$ and $V \# \text{bound}(\tilde{\sigma})$.

From $\tilde{\sigma} \preceq_\alpha \hat{\sigma} \preceq_\alpha \eta^\infty$ it follows $\text{mod}(\tilde{\sigma}) \subseteq \text{mod}(\eta^\infty) = \text{mod}(\eta)$. Together with $\beta \notin \text{free}(\Gamma) \cup \text{mod}(\eta)$ it follows $\beta \notin \text{free}((\Gamma + \text{tyEnv}(\eta))\tilde{\sigma}$.

Hence the TYPE ABS rule gives us

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash \lambda\beta.M[\beta/\alpha]\tilde{\sigma} \,:\, \forall\beta.\tau\tilde{\sigma}$$

Using the RENAME rule with

$$\begin{array}{ll} & \lambda\beta.M[\beta/\alpha]\tilde{\sigma} \\ \equiv & \{\text{Lemma 2.3; } \beta \notin \mathrm{mod}(\tilde{\sigma}) \subseteq \mathrm{mod}(\eta)\} \\ & (\lambda\beta.M[\beta/\alpha])\tilde{\sigma} \\ \equiv & \{\text{Definition of } \equiv\text{; } \beta \notin \mathrm{free}(\lambda\alpha.M)\} \\ & (\lambda\alpha.M)\tilde{\sigma} \end{array}$$

and

$$\begin{array}{ll} & \forall\beta.\tau\tilde{\sigma} \\ \equiv & \{\text{Lemma 2.3; } \beta \notin \mathrm{mod}(\tilde{\sigma}) \subseteq \mathrm{mod}(\eta)\} \\ & (\forall\beta.\tau)\tilde{\sigma} \end{array}$$

we obtain

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash (\lambda\alpha.M)\tilde{\sigma} \,:\, (\forall\beta.\tau)\tilde{\sigma}$$

Therefore

$$\begin{array}{ll} & (\forall\beta.\tau)\hat{\sigma} \\ \equiv & \{\text{Lemma 2.3; } \beta \notin \mathrm{mod}(\hat{\sigma}) \subseteq \mathrm{mod}(\eta)\} \\ & \forall\beta.\tau\hat{\sigma} \\ \equiv & \\ & \forall\beta.\hat{\tau}' \\ \equiv & \{\text{Definition of } \hat{\tau}'\} \\ & \hat{\tau} \end{array}$$

Finally, $\beta \notin V$ and $V \# \mathrm{bound}(\tau)$ gives us $V \# \mathrm{bound}(\forall\beta.\tau)$.

Case $M\,\rho$.
From

$$(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash (M\,\rho)\hat{\sigma} \,:\, \hat{\tau}$$

it follows

$$(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash (M\hat{\sigma})\,(\rho\hat{\sigma}) \,:\, \hat{\tau}$$

and hence with property G-TYPE APP the existence of γ and $\hat{\tau}'$ with $\hat{\tau} \equiv \hat{\tau}'[\rho\hat{\sigma}/\gamma]$ and

$$(\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \vdash M\hat{\sigma} \,:\, \forall\gamma.\hat{\tau}'$$

Hence by induction hypothesis there exists $(\tilde{\sigma}, \tau') = \mathcal{T}_\eta(\Gamma, M, \sigma)$ with

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash M\tilde{\sigma} \,:\, \tau'\tilde{\sigma}$$

and $\tau'\hat{\sigma} \equiv \forall\gamma.\hat{\tau}'$ and $\sigma \preceq_\alpha \tilde{\sigma} \preceq_\alpha \hat{\sigma}$ and $\tilde{\sigma}$ is idempotent and $V \# \mathrm{bound}(\tilde{\sigma})$ and $V \# \mathrm{bound}(\tau')$.

Suppose τ' is a variable. Because $\hat{\sigma} \preceq_\alpha \eta^\infty$ and $\tau'\hat{\sigma} \equiv \forall\gamma.\hat{\tau}'$, $\eta^\infty(\tau')$ must be a universal type. However, this contradicts with the fact that η substitutes list types. So τ' is not a variable and there exist α and τ with $\tau' = \forall\alpha.\tau$.

So

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash M\tilde{\sigma} \,:\, (\forall\alpha.\tau)\tilde{\sigma}$$

Let β be new with $\beta \notin \mathrm{free}(\forall\alpha.\tau) \cup \mathrm{mod}(\hat{\sigma}) \cup \mathrm{free}(\hat{\tau}')$.

With

$$\begin{array}{ll} & (\forall\alpha.\tau)\tilde{\sigma} \\ \equiv & \{\text{Definition of } \equiv;\ \beta \notin \mathrm{free}(\forall\alpha.\tau)\} \\ & (\forall\beta.\tau[\beta/\alpha])\tilde{\sigma} \\ \equiv & \{\text{Lemma 2.3};\ \beta \notin \mathrm{mod}(\tilde{\sigma}) \subseteq \mathrm{mod}(\hat{\sigma})\} \\ & \forall\beta.\tau[\beta/\alpha]\tilde{\sigma} \end{array}$$

and the TYPE RENAME rule we obtain

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash M\tilde{\sigma} \,:\, \forall\beta.\tau[\beta/\alpha]\tilde{\sigma}$$

Then the TYPE APP rule gives us

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash (M\tilde{\sigma})\,(\rho\tilde{\sigma}) \,:\, \tau[\beta/\alpha]\tilde{\sigma}[\rho\tilde{\sigma}/\beta]$$

Together with

$$\begin{array}{ll} & \tau[\beta/\alpha]\tilde{\sigma}[\rho\tilde{\sigma}/\beta] \\ \equiv & \{\text{Lemma 2.4};\ \beta \notin \mathrm{mod}(\tilde{\sigma}) \subseteq \mathrm{mod}(\hat{\sigma})\} \\ & \tau[\beta/\alpha][\rho/\beta]\tilde{\sigma} \\ \equiv & \{\beta = \alpha \text{ or } \beta \notin \mathrm{free}(\tau), \text{ because } \beta \notin \mathrm{free}(\forall\alpha.\tau)\} \\ & \tau[\rho/\alpha]\tilde{\sigma} \\ \equiv & \{\text{Lemma 4.32}\} \\ & \mathrm{subs}_{\mathrm{dom}(\eta)}(\tau, [\rho/\alpha]) \end{array}$$

and the TYPE RENAME rule we obtain

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash (M\tilde{\sigma})\,(\rho\tilde{\sigma}) \,:\, \mathrm{subs}_{\mathrm{dom}(\eta)}(\tau, [\rho/\alpha])\tilde{\sigma}$$

that is,

$$(\Gamma + \mathrm{tyEnv}(\eta))\tilde{\sigma} \vdash (M\,\rho)\tilde{\sigma} \,:\, \mathrm{subs}_{\mathrm{dom}(\eta)}(\tau, [\rho/\alpha])\tilde{\sigma}$$

$$\mathcal{C}_\eta(\gamma,\sigma) := \begin{cases} \mathcal{U}_{\mathrm{dom}(\eta)}(\gamma'\eta \doteq \gamma\eta, \sigma) & \text{, if exists } \gamma' \text{ with } \gamma\sigma = \gamma' \\ \mathcal{U}_{\mathrm{dom}(\eta)}(\gamma \doteq \gamma\eta, \sigma) & \text{, otherwise} \end{cases}$$

$$\begin{aligned} \mathcal{C}_\eta(\emptyset,\sigma) &:= \sigma \\ \mathcal{C}_\eta(V \dot\cup \{\gamma\},\sigma) &:= \mathcal{C}_\eta(V, \mathcal{C}_\eta(\gamma,\sigma)) \end{aligned}$$

$$\begin{aligned} \mathcal{C}_\eta(\sigma) := \text{let } & \sigma_0 := \sigma \\ & \sigma_{i+1} := \mathcal{C}_\eta(\mathrm{dom}(\eta), \sigma_i) \text{ for all natural numbers } i \\ & j \text{ the least natural number with } \sigma_j = \sigma_{j+1} \\ \text{in } & \sigma_j \end{aligned}$$

Figure 4.7: Consistency closure algorithm $\mathcal{C}$

Furthermore,

$$\begin{aligned} & \mathrm{subs}_{\mathrm{dom}(\eta)}(\tau, [\rho/\alpha])\hat\sigma \\ \equiv\ & \quad \{\text{Lemma 4.32}\} \\ & (\tau[\rho/\alpha])\hat\sigma \\ \equiv\ & \quad \{\beta = \alpha \text{ or } \beta \notin \mathrm{free}(\tau), \text{ because } \beta \notin \mathrm{free}(\forall\alpha.\tau)\} \\ & \tau[\beta/\alpha][\rho/\beta]\hat\sigma \\ \equiv\ & \quad \{\text{Lemma 2.4; } \beta \notin \mathrm{mod}(\hat\sigma)\} \\ & \tau[\beta/\alpha]\hat\sigma[\rho\hat\sigma/\beta] \\ \equiv\ & \quad \left\{\begin{array}{l} \text{Lemma 2.2; for any } \delta \notin \mathrm{free}(\forall\alpha.\tau) \\ \forall\delta.\tau[\delta/\alpha]\hat\sigma \equiv (\forall\alpha.\tau)\hat\sigma \equiv \forall\gamma.\hat\tau' \end{array}\right\} \\ & \hat\tau'[\beta/\gamma][\rho\hat\sigma/\beta] \\ \equiv\ & \quad \{\beta \notin \mathrm{free}(\hat\tau')\} \\ & \hat\tau'[\rho\hat\sigma/\gamma] \\ \equiv\ & \hat\tau \end{aligned}$$

Finally, $V \# \mathrm{bound}(\tau')$ implies $V \# \mathrm{bound}(\tau)$. Also $V \# \mathrm{bound}(\rho)$. So Lemma 4.32 assures $V \# \mathrm{bound}(\mathrm{subs}_V(\tau, [\rho/\alpha]))$. ■

4.3.6 The Consistency Closure Algorithm $\mathcal{C}$

As we discussed in Section 4.3.3, we have to post-process the result of type inference to ensure that we only obtain typings that are consistent with the instance substitution η. The algorithm for ensuring consistency, the consistency closure algorithm $\mathcal{C}$, is defined in Figure 4.7.

$\mathcal{C}_\eta(\gamma,\sigma)$ computes an extension of σ which is consistent with η with respect to the variable γ, assuming that $\sigma \preceq_\alpha \eta^\infty$ holds. We obtain the definition of $\mathcal{C}_\eta(\gamma,\sigma)$ directly from the properties of a consistent substitution as given in Definition 4.26. In practice we may define $\mathcal{C}_\eta(\gamma,\sigma) := \sigma$ in case of $\gamma\sigma = \gamma$ to avoid unnecessary computation.

$\mathcal{C}_\eta(V,\sigma)$ extends σ by ensuring consistency iteratively with respect to every variable $\gamma \in V$. Note that $\mathcal{C}_\eta(\gamma,\sigma)$ is consistent with respect to the variable γ, but may not be consistent with respect to a variable γ' for which σ is consistent. That is, consistency closure for one variable may destroy consistency for another. For example $[\gamma_4/\gamma_1]$ is consistent with the instance substitution

$$\eta = [\texttt{[}\gamma_2\texttt{]}/\gamma_1, \texttt{[}\gamma_3\texttt{]}/\gamma_2, \texttt{[Int]}/\gamma_3, \texttt{[}\gamma_5\texttt{]}/\gamma_4, \texttt{[}\gamma_6\texttt{]}/\gamma_5, \texttt{[Int]}/\gamma_6]$$

for all type inference variables but γ_1. However, $[\gamma_4/\gamma_1, \gamma_5/\gamma_2] = \mathcal{C}_\eta(\gamma_1, [\gamma_4/\gamma_1])$ is consistent with respect to γ_1 but not with respect to γ_2. Hence we define the consistency closure of a substitution σ, $\mathcal{C}_\eta(\sigma)$, by iteration until the substitution is consistent with respect to all type inference variables. Alternatively we could obtain a total order on type inference variables from the acyclic instance substitution η, such that if a substitution is closed under consistency with respect to the type inference variables in the determined order, no iteration is necessary. However, in practice the improvement in performance is probably small, so that it is not worth the additional complexity.

The algorithm $\mathcal{C}$ is nondeterministic, because the variables in $\mathrm{dom}(\eta)$ can be processed in any order. However, from Lemma 4.36 it follows that the result is unique up to variable renaming.

The consistency closure algorithm obviously yields a consistent substitution.

Lemma 4.34 (Consistency)

$$\frac{\sigma = \mathcal{C}_\eta(\sigma) \quad \sigma \textit{ idempotent} \quad \sigma \preceq_\alpha \eta^\infty \\ \mathrm{dom}(\eta) \mathbin{\#} \mathrm{bound}(\eta) \quad \mathrm{dom}(\eta) \mathbin{\#} \mathrm{bound}(\sigma)}{\sigma \textit{ consistent with } \eta} \qquad \square$$

Proof According to the definition of $\mathcal{C}$ we have $\sigma = \mathcal{C}_\eta(\mathrm{dom}(\eta),\sigma)$ and furthermore $\sigma = \mathcal{C}_\eta(\gamma,\sigma)$ for all $\gamma \in \mathrm{dom}(\eta)$. The other premises assure that the premises of the unification Lemma 4.24 hold.

Case $\gamma\sigma = \gamma'$.
According to the definition of $\mathcal{C}$ we have $\sigma = \mathcal{U}_{\mathrm{dom}(\eta)}(\gamma'\eta \doteq \gamma\eta, \sigma)$. With Lemma 4.24 it follows $\gamma'\eta\sigma \equiv \gamma\eta\sigma$.

Case $\gamma\sigma$ not a variable.
According to the definition of $\mathcal{C}$ we have $\sigma = \mathcal{U}_{\mathrm{dom}(\eta)}(\gamma \doteq \gamma\eta, \sigma)$. With Lemma 4.24 it follows $\gamma\tilde{\sigma} \equiv \gamma\eta\tilde{\sigma}$.

For $\gamma \notin \mathrm{dom}(\eta)$ the consistency constraints trivially hold. ■

Because the definition of the consistency closure algorithm is based on the algorithm $\mathcal{U}$, which determines a most general unifier, $\mathcal{C}$ yields a most general consistent extension.

Lemma 4.35 (Most general consistent extension)

$$\frac{\tilde{\sigma} = \mathcal{C}_\eta(\gamma,\sigma) \quad \gamma \in \mathrm{dom}(\eta) \\ \sigma \preceq_\alpha \hat{\sigma} \quad \hat{\sigma} \textit{ consistent with } \eta \quad \sigma \textit{ idempotent} \\ \mathrm{dom}(\eta) \mathbin{\#} \mathrm{bound}(\eta) \quad \mathrm{dom}(\eta) \mathbin{\#} \mathrm{bound}(\sigma)}{\tilde{\sigma} \preceq_\alpha \hat{\sigma}} \qquad \square$$

Proof Case $\exists\gamma'$ with $\gamma\sigma = \gamma'$.
With the definition of $\mathcal{C}$ it follows $\tilde{\sigma} = \mathcal{U}_{\mathrm{dom}(\eta)}(\gamma'\eta \doteq \gamma\eta, \tilde{\sigma})$. The premises of the unification Lemma 4.24 are fulfilled. We have $\gamma\hat{\sigma} = \gamma\sigma\hat{\sigma} = \gamma'\hat{\sigma}$.

Case $\exists\gamma''$ with $\gamma'\hat\sigma = \gamma''$.

The consistency of $\hat\sigma$ with η implies $\gamma'\eta\hat\sigma = \gamma''\eta\hat\sigma = \gamma\eta\hat\sigma$. With Lemma 4.24 follows the existence of $\tilde\sigma$ with $\tilde\sigma \preceq_\alpha \hat\sigma$.

Case $\gamma'\hat\sigma$ not a variable.

Then $\gamma\hat\sigma$ is not a variable either. The consistency of $\hat\sigma$ with η implies $\gamma\hat\sigma = \gamma\eta\hat\sigma$ and $\gamma'\hat\sigma = \gamma'\eta\hat\sigma$. Together these equations give $\gamma\eta\hat\sigma = \gamma'\eta\hat\sigma$. With Lemma 4.24 follows the existence of $\tilde\sigma$ with $\tilde\sigma \preceq_\alpha \hat\sigma$.

Case $\gamma\sigma$ not a variable.

With the definition of $\mathcal{C}$ it follows $\tilde\sigma = \mathcal{U}_{\mathrm{dom}(\eta)}(\gamma \doteq \gamma\eta, \tilde\sigma)$. The premises of the unification Lemma 4.24 are fulfilled. Because $\gamma\sigma\hat\sigma = \gamma\hat\sigma$ is not a variable and $\hat\sigma$ is consistent with η, we have $\gamma\hat\sigma = \gamma\eta\hat\sigma$. Then Lemma 4.24 assures the existence of $\tilde\sigma$ with $\tilde\sigma \preceq_\alpha \hat\sigma$. ■

Lemma 4.36 (Most general consistent extension)

$$\frac{\begin{array}{c}\sigma \preceq_\alpha \hat\sigma \quad \hat\sigma \textit{ consistent with } \eta \quad \sigma \textit{ idempotent}\\ \mathrm{dom}(\eta)\,\#\,\mathrm{bound}(\eta) \quad \mathrm{dom}(\eta)\,\#\,\mathrm{bound}(\sigma)\end{array}}{\exists\tilde\sigma \quad \begin{array}{c}\tilde\sigma = \mathcal{C}_\eta(\sigma) \quad \tilde\sigma \textit{ idempotent} \quad \tilde\sigma \textit{ consistent with } \eta\\ \sigma \preceq_\alpha \tilde\sigma \preceq_\alpha \hat\sigma \quad \mathrm{dom}(\eta)\,\#\,\mathrm{bound}(\tilde\sigma)\end{array}}$$ □

Proof Lemma 4.24 assures that all calls of the unification algorithm terminate. Furthermore, the lemma assures that $\sigma_0 \preceq_\alpha \sigma_1 \preceq_\alpha \sigma_2 \preceq_\alpha \ldots$, especially $\mathrm{dom}(\sigma_0) \subseteq \mathrm{dom}(\sigma_1) \subseteq \mathrm{dom}(\sigma_2) \subseteq \ldots$. Also $\mathrm{dom}(\sigma_i) \subseteq \mathrm{dom}(\eta)$ holds for all natural numbers i and we know that $\mathrm{dom}(\eta)$ is finite. Hence there exists a natural number j with $\mathrm{dom}(\sigma_j) = \mathrm{dom}(\sigma_{j+1})$ and with Lemma 4.21 it follows $\sigma_j = \sigma_{j+1}$. So $\mathcal{C}$ terminates.

Lemma 4.35 assures $\tilde\sigma \preceq_\alpha \hat\sigma$, Lemma 4.34 assures that $\tilde\sigma$ is consistent with η, and the remaining conclusions follow again from Lemma 4.24. ■

4.3.7 Principal Consistent Typing

Finally, we combine the type inference algorithm and the consistency closure algorithm:

$$\mathcal{CT}_\eta(\Gamma, M) := \text{let } (\sigma,\tau) := \mathcal{T}_\eta(\Gamma, M, [\,]) \text{ in } (\mathcal{C}_\eta(\sigma), \tau)$$

The algorithm $\mathcal{CT}$ computes a principal consistent typing:

Lemma 4.37 (Principal consistent typing)

$$\frac{\begin{array}{c}(\Gamma + \mathrm{tyEnv}(\eta))\hat\sigma \vdash M\hat\sigma : \hat\tau \quad \hat\sigma \textit{ consistent with } \eta \quad \mathrm{dom}(\hat\sigma) \subseteq V \quad V = \mathrm{dom}(\eta)\\ V\,\#\,\mathrm{bound}(\eta) \quad V\,\#\,\mathrm{bound}(\Delta) \quad V\,\#\,\mathrm{bound}(\Gamma) \quad V\,\#\,\mathrm{bound}(M)\end{array}}{\exists(\tilde\sigma,\tilde\tau) \quad \begin{array}{c}(\tilde\sigma,\tilde\tau) = \mathcal{CT}_\eta(\Gamma, M) \quad (\Gamma + \mathrm{tyEnv}(\eta))\tilde\sigma \vdash M\tilde\sigma : \tilde\tau\tilde\sigma \quad \tilde\tau\hat\sigma \equiv \hat\tau\\ \tilde\sigma \textit{ consistent with } \eta \quad \tilde\sigma \preceq_\alpha \hat\sigma \quad \tilde\sigma \textit{ idempotent}\end{array}}$$ □

Proof Follows from Lemma 4.33 and Lemma 4.36. ■

We revisit the two examples from Section 4.3.3. Consider the well-typed term

{xs:[[Int]]} ⊢ (:) [Int] ([] Int) xs : [[Int]]

The list replacement algorithm yields

$$\mathcal{R}(\emptyset, \texttt{(:) [Int] ([] Int) xs}) = (\underbrace{[\texttt{[Int]}/\gamma_1,\ \texttt{[}\gamma_1\texttt{]}/\gamma_2,\ \texttt{[Int]}/\gamma_3]}_{\eta}, v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_3}\ \texttt{xs})$$

and then we obtain the consistent principal typing

$$\begin{aligned}
& \mathcal{CT}_\eta(\{\text{xs}{:}\,\texttt{[[Int]]}\}, v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_3}\ \texttt{xs}) \\
=\ & \text{let } (\sigma,\tau) := \mathcal{T}_\eta(\{\text{xs}{:}\,\texttt{[[Int]]}\}, v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_3}\ \texttt{xs}, [\,]) \text{ in } (\mathcal{C}_\eta(\sigma),\tau) \\
=\ & (\mathcal{C}_\eta([\gamma_1/\gamma_3,\ \texttt{[[Int]]}/\gamma_2], \gamma_2) \\
=\ & ([\texttt{[Int]}/\gamma_3,\ \texttt{[Int]}/\gamma_1,\ \texttt{[[Int]]}/\gamma_2], \gamma_2)
\end{aligned}$$

because

$$\begin{aligned}
& \mathcal{C}_\eta(\{\gamma_1,\gamma_2\}, [\gamma_1/\gamma_3,\ \texttt{[[Int]]}/\gamma_2]) \\
=\ & \mathcal{C}_\eta(\{\gamma_2\}, \mathcal{C}_\eta(\gamma_1, [\gamma_1/\gamma_3,\ \texttt{[[Int]]}/\gamma_2]) \\
=\ & \mathcal{C}_\eta(\{\gamma_2\}, [\gamma_1/\gamma_3,\ \texttt{[[Int]]}/\gamma_2]) \\
=\ & \mathcal{U}_{\{\gamma_1,\gamma_2\}}(\gamma_2 \doteq \texttt{[}\gamma_1\texttt{]}, [\gamma_1/\gamma_3,\ \texttt{[[Int]]}/\gamma_2]) \\
=\ & [\texttt{[Int]}/\gamma_3,\ \texttt{[Int]}/\gamma_1,\ \texttt{[[Int]]}/\gamma_2]
\end{aligned}$$

Similarly for the well-typed term

$$\vdash \texttt{(:) [Int] ([] Int) ((:) [Int] ([] Int) ([] [Int])) : [[Int]]}$$

the list replacement algorithm yields

$$\begin{aligned}
& \mathcal{R}(\emptyset, \texttt{(:) [Int] ([] Int) ((:) [Int] ([] Int) ([] [Int]))}) \\
=\ & (\overbrace{[\texttt{[Int]}/\gamma_1,\ \texttt{[}\gamma_1\texttt{]}/\gamma_2,\ \texttt{[Int]}/\gamma_3,\ \texttt{[}\gamma_3\texttt{]}/\gamma_4,\ \texttt{[Int]}/\gamma_5,\ \texttt{[Int]}/\gamma_6,\ \texttt{[}\gamma_8\texttt{]}/\gamma_7,\ \texttt{[Int]}/\gamma_8]}^{\eta} \\
& , v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_5}\ (v^{\texttt{(:)}}_{\gamma_4}\ v^{\texttt{[]}}_{\gamma_6}\ v^{\texttt{[]}}_{\gamma_7}))
\end{aligned}$$

and then we obtain the consistent principal typing

$$\begin{aligned}
& \mathcal{CT}_\eta(\emptyset, v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_5}\ (v^{\texttt{(:)}}_{\gamma_4}\ v^{\texttt{[]}}_{\gamma_6}\ v^{\texttt{[]}}_{\gamma_7})) \\
=\ & \text{let } (\sigma,\tau) := \mathcal{T}_\eta(\emptyset, v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_5}\ (v^{\texttt{(:)}}_{\gamma_4}\ v^{\texttt{[]}}_{\gamma_6}\ v^{\texttt{[]}}_{\gamma_7}), [\,]) \text{ in } (\mathcal{C}_\eta(\sigma),\tau) \\
=\ & (\mathcal{C}_\eta([\gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_3/\gamma_6, \gamma_2/\gamma_7]), \gamma_2) \\
=\ & ([\gamma_1/\gamma_3, \gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_1/\gamma_6, \gamma_2/\gamma_7, \gamma_1/\gamma_8], \gamma_2)
\end{aligned}$$

because

$$\begin{aligned}
& \mathcal{C}_\eta(\{\gamma_1,\ldots,\gamma_8\}, [\gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_3/\gamma_6, \gamma_2/\gamma_7]) \\
=\ & \mathcal{C}_\eta(\{\gamma_1,\ldots,\gamma_6\}, \mathcal{C}_\eta(\gamma_7, [\gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_3/\gamma_6, \gamma_2/\gamma_7])) \\
=\ & \mathcal{C}_\eta(\{\gamma_1,\ldots,\gamma_6\}, \mathcal{U}_{\text{dom}(\eta)}(\texttt{[}\gamma_1\texttt{]} \doteq \texttt{[}\gamma_8\texttt{]}, [\gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_3/\gamma_6, \gamma_2/\gamma_7])) \\
=\ & \mathcal{C}_\eta(\gamma_4, [\gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_3/\gamma_6, \gamma_2/\gamma_7, \underline{\gamma_1/\gamma_8}]) \\
=\ & \mathcal{U}_{\text{dom}(\eta)}(\texttt{[}\gamma_1\texttt{]} \doteq \texttt{[}\gamma_3\texttt{]}, [\gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_3/\gamma_6, \gamma_2/\gamma_7, \gamma_1/\gamma_8]) \\
=\ & [\underline{\gamma_1/\gamma_3}, \gamma_2/\gamma_4, \gamma_1/\gamma_5, \underline{\gamma_1/\gamma_6}, \gamma_2/\gamma_7, \gamma_1/\gamma_8]
\end{aligned}$$

The following two properties are rather straightforward but they are important for the correctness of the complete list abstraction algorithm.

Lemma 4.38 (Instantiability)

$$\frac{\begin{array}{c}(\tilde{\sigma},\tilde{\tau}) = \mathcal{CT}_\eta(\Gamma, M)\\ (\Gamma + \mathrm{tyEnv}(\eta))\hat{\sigma} \ \vdash\ M\hat{\sigma}\ :\ \hat{\tau} \quad \hat{\sigma}\ \textit{consistent with}\ \eta \quad \mathrm{dom}(\hat{\sigma}) \subseteq V \quad V = \mathrm{dom}(\eta)\\ V \mathbin{\#} \mathrm{bound}(\eta) \quad V \mathbin{\#} \mathrm{bound}(\Delta) \quad V \mathbin{\#} \mathrm{bound}(\Gamma) \quad V \mathbin{\#} \mathrm{bound}(M)\end{array}}{M\tilde{\sigma}\,\mathrm{inst}(\eta^\infty) \equiv M\,\mathrm{inst}(\eta^\infty)} \qquad \square$$

Proof

$$\begin{array}{ll} & M\tilde{\sigma}\,\mathrm{inst}(\eta^\infty)\\ = & \{\text{Definition of inst}(\eta^\infty)\}\\ & M\tilde{\sigma}\eta^\infty\,\mathrm{instCons}\,\eta^\infty\\ \equiv & \{\tilde{\sigma}\text{ is idempotent and }\tilde{\sigma} \preceq_\alpha \eta^\infty\text{ according to Lemma 4.37}\}\\ & M\eta^\infty\,\mathrm{instCons}\,\eta^\infty\\ = & \{\text{Definition of inst}(\eta^\infty)\}\\ & M\,\mathrm{inst}(\eta^\infty)\end{array}$$

■

Lemma 4.39 (Typing)

$$\frac{\begin{array}{c}(\tilde{\sigma},\tilde{\tau}) = \mathcal{CT}_\eta(\Gamma, M) \quad \Gamma + \mathrm{tyEnv}(\eta)\eta^\infty \ \vdash\ M\eta^\infty\ :\ \hat{\tau}\\ V = \mathrm{dom}(\eta) \quad V \mathbin{\#} \mathrm{free}(\Gamma)\\ V \mathbin{\#} \mathrm{bound}(\eta) \quad V \mathbin{\#} \mathrm{bound}(\Delta) \quad V \mathbin{\#} \mathrm{bound}(\Gamma) \quad V \mathbin{\#} \mathrm{bound}(M)\end{array}}{\Gamma + \mathrm{tyEnv}(\eta)\tilde{\sigma} \ \vdash\ M\tilde{\sigma}\ :\ \tau\tilde{\sigma} \quad \tilde{\tau}\eta^\infty \equiv \hat{\tau}} \qquad \square$$

Proof Follows from Lemma 4.37, because η^∞ is consistent with η. ■

4.4 Instantiation of Variables and Abstraction Phase

With the principal consistent typing we can build the list abstracted term. The typing determines which constructor variables have to be reinstantiated to list constructors and which are to be abstracted. However, first we may have to instantiate some type inference variables.

4.4.1 Superfluous Type Inference Variables

Consider the well-typed term

$$\emptyset \vdash \texttt{case ([] Int, [] Int) of (x,y)} \mapsto \texttt{x : [Int]}$$

List replacement gives

$$\{v^{[]}_{\gamma_1}{:}\gamma_1, v^{[]}_{\gamma_2}{:}\gamma_2\} \vdash \texttt{case (}v^{[]}_{\gamma_1}\texttt{, }v^{[]}_{\gamma_2}\texttt{) of (x,y)} \mapsto \texttt{x}$$

and the type inference algorithm $\mathcal{CT}$ does not substitute any type inference variable so that

$$\{v^{[]}_{\gamma_1}{:}\gamma_1, v^{[]}_{\gamma_2}{:}\gamma_2\} \vdash \texttt{case (}v^{[]}_{\gamma_1}\texttt{, }v^{[]}_{\gamma_2}\texttt{) of (x,y)} \mapsto \texttt{x : }\gamma_1$$

is the principal consistent typing. Then list abstraction as discussed in Chapter 3 would construct

$$\lambda\gamma_1.\,\lambda\gamma_2.\,\lambda v^{\texttt{[]}}_{\gamma_1} : \gamma_1.\,\lambda v^{\texttt{[]}}_{\gamma_2}{:}\gamma_2.\,\texttt{case (}v^{\texttt{[]}}_{\gamma_1}\texttt{, }v^{\texttt{[]}}_{\gamma_2}\texttt{) of (x,y)} \mapsto \texttt{x}$$

However, this term is unnecessarily complicated and large. The type inference variable γ_2 does not occur in the type γ_1 of $\texttt{case (}v^{\texttt{[]}}_{\gamma_1}\texttt{, }v^{\texttt{[]}}_{\gamma_2}\texttt{) of (x,y)} \mapsto \texttt{x}$. Abstracting γ_2 would not improve deforestation but only increase the size of a program. The type inference variable γ_2 is superfluous. Hence we instantiate it to the type `[Int]` and have list abstraction return the following term:

$$\lambda\gamma_1.\,\lambda v^{\texttt{[]}}_{\gamma_1} : \gamma_1.\,\texttt{case (}v^{\texttt{[]}}_{\gamma_1}\texttt{, [] Int) of (x,y)} \mapsto \texttt{x}$$

The situation is similar to the Hindley-Milner type system where the type generalisation step at `let` bindings, the only source of universal types, only generalises over type variables that occur in the type [Mil78, DM82].

Unfortunately a type inference variable $\gamma \in \mathrm{dom}(\eta)$ is not necessarily superfluous with respect to a typing (σ, τ) if it does not occur in the type $\tau\sigma$.

Consider the the well-typed term

$$\vdash \texttt{(:) [Int] ([] Int) ((:) [Int] ([] Int) ([] [Int])) : [[Int]]}$$

As we saw in Section 4.3.7 the list replacement and the subsequent type inference phase give the principal consistent typing

$$\begin{array}{l}\{v^{\texttt{(:)}}_{\gamma_2}{:}\gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_2, v^{\texttt{(:)}}_{\gamma_4}{:}\gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_2, v^{\texttt{[]}}_{\gamma_5}{:}\gamma_1, v^{\texttt{[]}}_{\gamma_6}{:}\gamma_1, v^{\texttt{[]}}_{\gamma_7}{:}\gamma_2\} \\ \vdash\ v^{\texttt{(:)}}_{\gamma_2}\ \ v^{\texttt{[]}}_{\gamma_5}\ \ (v^{\texttt{(:)}}_{\gamma_4}\ \ v^{\texttt{[]}}_{\gamma_6}\ \ v^{\texttt{[]}}_{\gamma_7}) \\ :\ \ \gamma_2\end{array}$$

The type inference variable γ_1 does not occur in the inferred type γ_2. However, γ_1 replaces the element type of the list type that γ_2 replaces. For the construction of most general workers we want to abstract nested lists, as we did in Chapter 3. When we abstract from the type inference variable γ_2 we also abstract from its constructor variable $v^{\texttt{(:)}}_{\gamma_2}$ in whose type γ_1 appears.

The set of superfluous type inference variables of an instance substitution with respect to a typing (σ, τ) is the largest subset of $\mathrm{dom}(\eta) \setminus \mathrm{free}(\tau\sigma)$ such that the set is disjoint with $\mathrm{free}(\eta(\gamma'))$ for all type inference variables $\gamma' \in \mathrm{dom}(\eta)$ that are not in the set. Note that the last sentence implicitly defines a monotonic function of which the set of superfluous type inference variables is the largest fixpoint.

However, only for constructing a worker we are interested in abstracting as many lists as possible. If we perform list abstraction of a producer to apply the short cut fusion rule, then we have to abstract only the produced list, to fit the form of the fusion rule. So in that case all type inference variables except for $\tau\sigma$, which hopefully is a type inference variable, are superfluous with respect to a typing (σ, τ).

We instantiate a superfluous type inference variable $\gamma \in \mathrm{dom}(\eta) \setminus \mathrm{dom}(\sigma)$ by extending the substitution by $[\gamma\eta\sigma/\gamma]$. That is, instantiating the typing (σ, τ) for a superfluous variable γ yields $(\sigma[\gamma\eta\sigma/\gamma], \tau)$. Note that here again we need the instance substitution η.

By instantiating superfluous type inference variables we obviously loose the principality property of the typing. However, all other properties of the substitution are preserved as the subsequent lemmas prove.

Lemma 4.40

$$\frac{\sigma \textit{ is idempotent} \quad \gamma \notin \mathrm{dom}(\sigma) \quad \sigma \preceq_{\alpha} \eta^{\infty}}{\gamma \notin \mathrm{free}(\gamma\eta\sigma)}$$

□

Proof Because $\sigma \preceq_\alpha \eta^\infty$ and σ is idempotent, we have $\sigma\eta^\infty \equiv \eta^\infty$. Hence $\gamma\eta^\infty \equiv \gamma\eta\eta^\infty \equiv (\gamma\eta\sigma)\eta^\infty$. Because η is a list substitution, $\gamma\eta\sigma$ is not a variable. So, just considering the size of the types it is clear that the last equation can only hold if $\gamma \notin \text{free}(\gamma\eta\sigma)$. ■

Lemma 4.41 (Preservation of idempotency)

$$\frac{\sigma \textit{ is idempotent} \quad \sigma \textit{ consistent with } \eta \quad \gamma \notin \text{dom}(\sigma)}{\sigma[\gamma\eta\sigma/\gamma] \textit{ idempotent}}$$ □

Proof From Lemma 4.40 and Corollary 4.19. ■

Lemma 4.42 (Preservation of consistency)

$$\frac{\sigma \textit{ consistent with } \eta \quad \gamma \notin \text{dom}(\sigma)}{\sigma[\gamma\eta\sigma/\gamma] \textit{ consistent with } \eta}$$ □

Proof Case $\gamma'\sigma[\gamma\eta\sigma/\gamma] = \gamma''$.

Because η substitutes lists we have $\gamma'\sigma = \gamma''$. With the consistency of σ it follows $\gamma'\eta\sigma \equiv \gamma''\eta\sigma$ and hence $\gamma'\eta\sigma[\gamma\eta\sigma/\gamma] \equiv \gamma''\eta\sigma[\gamma\eta\sigma/\gamma]$.

Case $\gamma'\sigma[\gamma\eta\sigma/\gamma]$ not a variable.

Case $\gamma'\sigma$ not a variable.

With the consistency of σ it follows $\gamma'\sigma \equiv \gamma'\eta\sigma$ and hence $\gamma'\sigma[\gamma\eta\sigma/\gamma] \equiv \gamma'\eta\sigma[\gamma\eta\sigma/\gamma]$.

Case $\gamma'\sigma = \gamma$.

$$\begin{aligned}
& \gamma'\sigma[\gamma\eta\sigma/\gamma] \\
= \;& \gamma[\gamma\eta\sigma/\gamma] \\
= \;& \gamma\eta\sigma \\
= \;& \quad \{\gamma \notin \text{free}(\gamma\eta\sigma) \text{ according to Lemma 4.40}\} \\
& \gamma\eta\sigma[\gamma\eta\sigma/\gamma] \\
\equiv \;& \quad \{\sigma \text{ consistent with } \eta \text{ and } \gamma'\sigma = \gamma\} \\
& \gamma'\eta\sigma[\gamma\eta\sigma/\gamma]
\end{aligned}$$

■

Lemma 4.43 (Preservation of instantiability)

$$\frac{\sigma \textit{ idempotent} \quad \sigma \textit{ consistent with } \eta}{\sigma[\gamma\eta\sigma/\gamma]\eta^\infty = \eta^\infty}$$ □

Proof Because σ is idempotent and $\sigma \preceq_\alpha \eta^\infty$ we have $\sigma\eta^\infty = \eta^\infty$ and furthermore $\gamma\eta^\infty = \gamma\eta\eta^\infty = \gamma\eta(\sigma\eta^\infty) = (\gamma\eta\sigma)\eta^\infty$. Hence the conclusion follows with Lemma 4.20. ■

With $\text{inst}(\eta^\infty) = \eta^\infty \, \text{instCons}(\eta^\infty)$ it immediately follows

$$\frac{\sigma \text{ idempotent} \quad \sigma \text{ consistent with } \eta}{M\sigma[\gamma\eta\sigma/\gamma]\,\text{inst}(\eta^\infty) = M\sigma\,\text{inst}(\eta^\infty)}$$

Lemma 4.44 (Preservation of typing)

$$\frac{\Gamma + \text{tyEnv}(\eta)\sigma \vdash M\sigma : \tau\sigma \quad \gamma \notin \text{free}(\Gamma)}{\Gamma + \text{tyEnv}(\eta)\sigma[\gamma\eta\sigma/\gamma] \vdash M\sigma[\gamma\eta\sigma/\gamma] : \tau\sigma[\gamma\eta\sigma/\gamma]}$$ □

Proof Follows from the substitution property of Lemma 2.6. ■

4.4.2 Instantiation of Constructor Variables

The list replacement algorithm replaces all list constructors by constructor variables. Now some constructor variables have to be reinstantiated to the original list constructors.

Let (σ, τ) be the typing computed by the type inference phase and η be the instance substitution computed by the list replacement algorithm. The types of the constructor variables are given by the typing environment $\mathrm{tyEnv}(\eta)\sigma$. If $\mathrm{tyEnv}(\eta)\sigma(v_\gamma^{\texttt{(:)}}) = \tau \rightarrow \texttt{[}\tau\texttt{]} \rightarrow \texttt{[}\tau\texttt{]}$, then $v_\gamma^{\texttt{(:)}}$ is reinstantiated to $\texttt{(:)}\ \tau$, and if $\mathrm{tyEnv}(\eta)\sigma(v_\gamma^{\texttt{[]}}) = \texttt{[}\tau\texttt{]}$, then $v_\gamma^{\texttt{[]}}$ is reinstantiated to $\texttt{[]}\ \tau$. Using

$$\mathrm{tyEnv}(\eta)\sigma = \{v_\gamma^{\texttt{(:)}}{:}\tau_\gamma\sigma \rightarrow \gamma\sigma \rightarrow \gamma\sigma,\ v_\gamma^{\texttt{[]}}{:}\gamma\sigma \mid \eta(\gamma) = \texttt{[}\tau_\gamma\texttt{]}, \gamma \in \mathrm{dom}(\eta)\}$$

and the fact that the typing (σ, τ) is consistent, we can reformulate instantiation as follows: If $\sigma(\gamma) = \texttt{[}\tau\texttt{]}$, then $v_\gamma^{\texttt{(:)}}$ is reinstantiated to $\texttt{(:)}\ \tau$ and $v_\gamma^{\texttt{[]}}$ is reinstantiated to $\texttt{[]}\ \tau$. So reinstantiation of constructor variables is performed by the substitution $\mathrm{instCons}(\sigma)$.

For the example term

$$\texttt{case (}v_{\gamma_1}^{\texttt{[]}}\texttt{, }v_{\gamma_2}^{\texttt{[]}}\texttt{) of (x,y)} \mapsto \texttt{x}$$

from the preceeding section we have — after instantiation of superfluous type inference variables — the typing $([\texttt{[Int]}/\gamma_2], \gamma_1)$. So instantiation of constructor variables gives:

$$\begin{aligned} & (\texttt{case (}v_{\gamma_1}^{\texttt{[]}}\texttt{, }v_{\gamma_2}^{\texttt{[]}}\texttt{) of (x,y)} \mapsto \texttt{x})\,\mathrm{instCons}([\texttt{[Int]}/\gamma_2]) \\ = & (\texttt{case (}v_{\gamma_1}^{\texttt{[]}}\texttt{, }v_{\gamma_2}^{\texttt{[]}}\texttt{) of (x,y)} \mapsto \texttt{x})[\texttt{(:) Int}/v_{\gamma_2}^{\texttt{(:)}}, \texttt{[] Int}/v_{\gamma_2}^{\texttt{[]}}] \\ = & (\texttt{case (}v_{\gamma_1}^{\texttt{[]}}\texttt{, [] Int) of (x,y)} \mapsto \texttt{x}) \end{aligned}$$

The following two lemmas state properties that are essential for the correctness of the complete list abstraction algorithm.

Lemma 4.45 (Preservation of instantiability)

$$\frac{\sigma \textit{ idempotent} \quad \sigma \textit{ consistent with } \eta}{\mathrm{instCons}(\sigma)\,\mathrm{inst}(\eta^\infty) = \mathrm{inst}(\eta^\infty)} \qquad \square$$

Proof First, both sides have the same domain

$$\mathrm{dom}(\mathrm{instCons}(\sigma)\,\mathrm{inst}(\eta^\infty)) = \mathrm{dom}(\mathrm{instCons}(\sigma)) \cup \mathrm{dom}(\mathrm{inst}(\eta^\infty)) = \mathrm{dom}(\mathrm{inst}(\eta^\infty))$$

because $\sigma \preceq \eta^\infty$. Second, both sides agree on type variables $\gamma \in \mathrm{dom}(\mathrm{inst}(\eta^\infty))$:

$$\gamma\,\mathrm{instCons}(\sigma)\,\mathrm{inst}(\eta^\infty) = \gamma\eta^\infty = \gamma\,\mathrm{inst}(\eta^\infty)$$

Finally, we show that both sides agree on term variables $v_\gamma^c \in \mathrm{dom}(\mathrm{inst}(\eta^\infty))$.

Case $\sigma(\gamma) = \texttt{[}\tau\texttt{]}$.

Because σ is consistent with η, we have $\texttt{[}\tau\texttt{]}\eta^\infty = \gamma\sigma\eta^\infty = \gamma\eta^\infty =: \texttt{[}\tau'\texttt{]}$. So:

$$v_\gamma^c\,\mathrm{instCons}(\sigma)\,\mathrm{inst}(\eta^\infty) = c\,\tau\,\mathrm{inst}(\eta^\infty) = c\,(\tau\eta^\infty) = c\,\tau' = v_\gamma^c\,\mathrm{inst}(\eta^\infty)$$

Case $\sigma(\gamma) \neq \texttt{[}\tau\texttt{]}$.

Hence $v_\gamma^c \notin \mathrm{dom}(\mathrm{instCons}(\sigma))$. Then $v_\gamma^c\,\mathrm{instCons}(\sigma)\,\mathrm{inst}(\eta^\infty) = v_\gamma^c\,\mathrm{inst}(\eta^\infty)$. ■

Lemma 4.46 (Preservation of typing) *Instantiation of superfluous constructor variables preserves typing.*

$$\frac{\sigma \textit{ consistent with } \eta \quad \Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash M : \tau}{\Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash M\,\mathrm{instCons}(\sigma) : \tau} \qquad \square$$

Proof We have to verify that for every replaced variable v^c_γ its replacement $c\,\tau$ has the same type. Then the substitution preserves typing according to Lemma 2.6.

A variable v^c_γ is replaced by $c\,\tau$, if $\sigma(\gamma) = \texttt{[}\tau\texttt{]}$ for some τ.

Case $c = \texttt{[]}$.

We have $\mathrm{tyEnv}(\eta)\sigma(v^{\texttt{[]}}_\gamma) = \gamma\sigma$ and $\Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash \texttt{[]}\,\tau : \texttt{[}\tau\texttt{]}$. According to assumption $\gamma\sigma = \texttt{[}\tau\texttt{]}$.

Case $c = \texttt{(:)}$.

We have $\mathrm{tyEnv}(\eta)\sigma(v^{\texttt{(:)}}_\gamma) = (\tau' \to \gamma \to \gamma)\sigma = (\tau'\sigma) \to \texttt{[}\tau\texttt{]} \to \texttt{[}\tau\texttt{]}$ where $\texttt{[}\tau'\texttt{]} := \eta(\gamma)$. On the other hand $\Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash \texttt{(:)}\,\tau : \tau \to \texttt{[}\tau\texttt{]} \to \texttt{[}\tau\texttt{]}$. Because σ is consistent with η, we have $\texttt{[}\tau\texttt{]} = \sigma(\gamma) = \eta(\gamma)\sigma = \texttt{[}\tau'\texttt{]}\sigma$. Thus $\tau = \tau'\sigma$ and both types agree. ■

4.4.3 Merge of Constructor Variables

The list replacement algorithm replaces every occurrence of a list constructor by a different constructor variable. Now we have to identify all constructor variables which construct values of the same type and merge them, that is, replace them by the same constructor variable.

Consider for example the term

$$v^{\texttt{(:)}}_{\gamma_2}\; v^{\texttt{[]}}_{\gamma_5}\; (v^{\texttt{(:)}}_{\gamma_4}\; v^{\texttt{[]}}_{\gamma_6}\; v^{\texttt{[]}}_{\gamma_7})$$

and the typing

$$([\gamma_1/\gamma_3, \gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_1/\gamma_6, \gamma_2/\gamma_7, \gamma_1/\gamma_8], \gamma_2)$$

According to the typing $v^{\texttt{(:)}}_{\gamma_2}$, $v^{\texttt{(:)}}_{\gamma_4}$ and $v^{\texttt{[]}}_{\gamma_7}$ construct values of the same type γ_2. Hence we replace $v^{\texttt{(:)}}_{\gamma_4}$ by $v^{\texttt{(:)}}_{\gamma_2}$ and for simplicity also $v^{\texttt{[]}}_{\gamma_7}$ by $v^{\texttt{[]}}_{\gamma_2}$. Similarly we see from the typing that $v^{\texttt{[]}}_{\gamma_5}$ and $v^{\texttt{[]}}_{\gamma_6}$ construct values of type γ_1. Hence we replace both constructor variables by $v^{\texttt{[]}}_{\gamma_1}$.

In general, when we have a typing (σ, τ) we merge the constructor variables of a term by applying the substitution merge(σ):

Definition 4.47

$$\mathrm{merge}(\sigma) := [v^{\texttt{(:)}}_{\gamma'}/v^{\texttt{(:)}}_{\gamma}, v^{\texttt{[]}}_{\gamma'}/v^{\texttt{[]}}_{\gamma} \mid \sigma(\gamma) = \gamma'] \qquad \square$$

As discussed we obtain for the example:

$$\begin{aligned} & v^{\texttt{(:)}}_{\gamma_2}\; v^{\texttt{[]}}_{\gamma_5}\; (v^{\texttt{(:)}}_{\gamma_4}\; v^{\texttt{[]}}_{\gamma_6}\; v^{\texttt{[]}}_{\gamma_7})\,\mathrm{merge}([\gamma_1/\gamma_3, \gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_1/\gamma_6, \gamma_2/\gamma_7, \gamma_1/\gamma_8]) \\ = \; & v^{\texttt{(:)}}_{\gamma_2}\; v^{\texttt{[]}}_{\gamma_5}\; (v^{\texttt{(:)}}_{\gamma_4}\; v^{\texttt{[]}}_{\gamma_6}\; v^{\texttt{[]}}_{\gamma_7})[v^{\texttt{(:)}}_{\gamma_1}/v^{\texttt{(:)}}_{\gamma_3}, v^{\texttt{[]}}_{\gamma_1}/v^{\texttt{[]}}_{\gamma_3}, \dots, v^{\texttt{(:)}}_{\gamma_1}/v^{\texttt{(:)}}_{\gamma_8}, v^{\texttt{[]}}_{\gamma_1}/v^{\texttt{[]}}_{\gamma_8}] \\ = \; & v^{\texttt{(:)}}_{\gamma_2}\; v^{\texttt{[]}}_{\gamma_1}\; (v^{\texttt{(:)}}_{\gamma_2}\; v^{\texttt{[]}}_{\gamma_1}\; v^{\texttt{[]}}_{\gamma_2}) \end{aligned}$$

Finally, we have to prove again two properties for the correctness of the complete list abstraction algorithm.

Lemma 4.48 (Preservation of instantiability)

$$\frac{\sigma \textit{ idempotent} \quad \sigma \textit{ consistent with } \eta}{\mathrm{merge}(\sigma)\,\mathrm{inst}(\eta^\infty) = \mathrm{inst}(\eta^\infty)}$$

Proof First, both sides have the same domain

$$\mathrm{dom}(\mathrm{merge}(\sigma)\,\mathrm{inst}(\eta^\infty)) = \mathrm{dom}(\mathrm{merge}(\sigma)) \cup \mathrm{dom}(\mathrm{inst}(\eta^\infty)) = \mathrm{dom}(\mathrm{inst}(\eta^\infty))$$

because $\sigma \preceq \eta^\infty$. Second, because $\mathrm{dom}(\mathrm{merge}(\sigma))$ contains no type variables, both sides agree on type variables. Finally, both sides agree on term variables v^c_γ of their common domain:

Case $\sigma(\gamma) = \gamma'$.

With

$$\begin{array}{cl} & \eta^\infty(\gamma)\\ = & \\ & \gamma\eta^\infty \\ = & \{\sigma \text{ idempotent and } \sigma \preceq_\alpha \eta^\infty\}\\ & \gamma\sigma\eta^\infty\\ = & \{\text{assumption: } \sigma(\gamma) = \gamma'\}\\ & \gamma'\eta^\infty\\ = & \\ & \eta^\infty(\gamma')\end{array}$$

it follows

$$v^c_\gamma\,\mathrm{merge}(\sigma)\,\mathrm{inst}(\eta^\infty) = v^c_{\gamma'}\,\mathrm{inst}(\eta^\infty) = c\,(\eta^\infty(\gamma')) = c\,(\eta^\infty(\gamma)) = v^c_\gamma\,\mathrm{inst}(\eta^\infty)$$

Case $\sigma(\gamma) \neq \gamma'$.

Hence $v^c_\gamma \notin \mathrm{dom}(\mathrm{merge}(\sigma))$. Then $v^c_\gamma\,\mathrm{merge}(\sigma)\,\mathrm{inst}(\eta^\infty) = v^c_\gamma\,\mathrm{inst}(\eta^\infty)$. ■

Lemma 4.49 (Preservation of typing)

$$\frac{\sigma \textit{ idempotent} \quad \sigma \textit{ consistent with } \eta \quad \Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash M : \tau}{\Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash M\,\mathrm{merge}(\sigma) : \tau}$$

Proof We have to verify that for every replaced variable v^c_γ its replacement $v^c_{\gamma'}$ has the same type. Then the substitution preserves typing according to Lemma 2.6.

A variable v^c_γ is replaced by $v^c_{\gamma'}$, if $\sigma(\gamma) = \gamma'$ for some γ'. Because σ is idempotent, we have $\gamma'\sigma = \gamma\sigma\sigma = \gamma\sigma$.

Case $c = \mathtt{[]}$.

We have $\mathrm{tyEnv}(\eta)\sigma(v^{\mathtt{[]}}_\gamma) = \gamma\sigma$ and $\Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash v^{\mathtt{[]}}_{\gamma'} : \gamma'\sigma$.

Case $c = \mathtt{(:)}$.

We have $\mathrm{tyEnv}(\eta)\sigma(v^{\mathtt{(:)}}_\gamma) = (\tau \to \gamma \to \gamma)\sigma = (\tau\sigma) \to (\gamma\sigma) \to (\gamma\sigma)$ where $\mathtt{[}\tau\mathtt{]} := \eta(\gamma)$. On the other hand $\Gamma + \mathrm{tyEnv}(\eta)\sigma \vdash v^{\mathtt{(:)}}_{\gamma'} : (\tau' \to \gamma' \to \gamma')\sigma$ where $\mathtt{[}\tau'\mathtt{]} := \eta(\gamma')$. Because σ is consistent with η, we have $\mathtt{[}\tau\mathtt{]}\sigma = \eta(\gamma)\sigma = \eta(\gamma')\sigma = \mathtt{[}\tau'\mathtt{]}\sigma$. Thus $\tau\sigma = \tau'\sigma$ and both types agree. ■

$$
\begin{array}{lll}
\mathcal{A}(\Gamma, M) := \texttt{let} & (\tilde{M}, \eta) := \mathcal{R}(\emptyset, M) \\
 & (\tilde{\sigma}, \tau) := \mathcal{CT}_{\eta}(\Gamma, \tilde{M}) \\
 & \sigma := \tilde{\sigma}[\hat{\gamma}_1 \eta \tilde{\sigma} / \hat{\gamma}_1] \ldots \\
 & \{\gamma_1, \ldots, \gamma_k\} := \mathrm{dom}(\eta) \setminus \mathrm{dom}(\sigma) \\
 & \sigma' := \sigma\, \mathrm{instCons}(\sigma)\, \mathrm{merge}(\sigma) \\
 & F := \mathrm{freeTerm}(\tilde{M}\sigma') \\
 & \{v^{\texttt{(:)}}_{\gamma'_1}, \ldots, v^{\texttt{(:)}}_{\gamma'_m}\} := \{v^{\texttt{(:)}}_{\gamma} \mid \gamma \in \mathrm{dom}(\eta)\} \cap F \\
 & \{v^{\texttt{[]}}_{\gamma''_1}, \ldots, v^{\texttt{[]}}_{\gamma''_n}\} := \{v^{\texttt{[]}}_{\gamma} \mid \gamma \in \mathrm{dom}(\eta)\} \cap F \\
 & \texttt{[}\tau_i\texttt{]} := \gamma'_i \eta \sigma \quad \forall 1 \leq i \leq m \\
 & \texttt{[}\tau'_i\texttt{]} := \eta^{\infty}(\gamma'_i) \quad \forall 1 \leq i \leq m \\
 & \texttt{[}\tau''_i\texttt{]} := \eta^{\infty}(\gamma''_i) \quad \forall 1 \leq i \leq n \\
\qquad\qquad \texttt{in} & \texttt{(}\lambda\gamma_1. \ldots \lambda\gamma_k. \\
 & \quad \lambda v^{\texttt{(:)}}_{\gamma'_1} : \tau_1 \to \gamma'_1 \to \gamma'_1. \ldots \lambda v^{\texttt{(:)}}_{\gamma'_m} : \tau_m \to \gamma'_m \to \gamma'_m. \\
 & \quad \lambda v^{\texttt{[]}}_{\gamma''_1} : \gamma''_1. \ldots \lambda v^{\texttt{[]}}_{\gamma''_n} : \gamma''_n. \tilde{M}\sigma'\texttt{)} \\
 & \quad \eta^{\infty}(\gamma_1) \ldots \eta^{\infty}(\gamma_k)\ \texttt{((:)}\ \tau'_1\texttt{)} \ldots \texttt{((:)}\ \tau'_m\texttt{)}\ \texttt{([]}\ \tau''_1\texttt{)} \ldots \texttt{([]}\ \tau''_n\texttt{)}
\end{array}
$$

Figure 4.8: List abstraction algorithm

4.4.4 Abstraction

Finally, the list abstracted form has to be constructed. We do not formulate this as a separate algorithm, but define the complete list abstraction algorithm $\mathcal{A}$ in Figure 4.8.

First, the list replacement algorithm replaces all list constructors by constructor variables and list types by type inference variables. Subsequently, the type inference algorithm determines a principal consistent typing $(\tilde{\sigma}, \tau)$. Note that the type component τ is not used in the algorithm. It is only useful for proving the correctness of $\mathcal{A}$. Next, superfluous type inference variables $\hat{\gamma}_1, \ldots$ are instantiated as discussed in Section 4.4.1, giving us a substitution σ. We do not specify which type inference variables are superfluous, because that depends on whether we perform list abstraction for the short cut fusion rule or for a worker/wrapper split. The list abstraction algorithm has to abstract those type inference variables $\gamma_1, \ldots, \gamma_k$ that are not replaced by σ. We extend σ to instantiate some constructor variables by instCons(σ) and merge the others by merge(σ). We do not abstract all type constructor variables for the types $\gamma_1, \ldots, \gamma_k$ but only those that occur in the term $\tilde{M}\sigma$. Note that $\{\gamma'_1, \ldots, \gamma'_m\} \subseteq \{\gamma_1, \ldots, \gamma_k\}$ and $\{\gamma''_1, \ldots, \gamma''_n\} \subseteq \{\gamma_1, \ldots, \gamma_k\}$. Finally, the instance substitution η is used to provide all type arguments. Stretching our overline notation for abbreviating sequences we can say that the result is of the form

$$(\overline{\lambda\gamma.}\ \overline{\lambda\mathtt{v}^{\texttt{(:)}} : \tau{\to}\gamma{\to}\gamma.}\ \overline{\lambda\mathtt{v}^{\texttt{[]}} : \gamma.}\ M)\ \overline{\texttt{[}\tau_\gamma\texttt{]}}\ \overline{\texttt{((:)}\ \tau_\gamma\texttt{)}}\ \overline{\texttt{([]}\ \tau_\gamma\texttt{)}}$$

For example, for the well-typed term

$$\vdash \texttt{(:) [Int] ([] Int) ((:) [Int] ([] Int) ([] [Int]))} : \texttt{[[Int]]}$$

we have seen in the previous sections how the instance substitution

$$\texttt{[[Int]}/\gamma_1, \texttt{[}\gamma_1\texttt{]}/\gamma_2, \texttt{[Int]}/\gamma_3, \texttt{[}\gamma_3\texttt{]}/\gamma_4, \texttt{[Int]}/\gamma_5, \texttt{[Int]}/\gamma_6, \texttt{[}\gamma_8\texttt{]}/\gamma_7, \texttt{[Int]}/\gamma_8]$$

the principal consistent typing

$$([\gamma_1/\gamma_3, \gamma_2/\gamma_4, \gamma_1/\gamma_5, \gamma_1/\gamma_6, \gamma_2/\gamma_7, \gamma_1/\gamma_8], \gamma_2)$$

and the term

$$v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_1}\ (v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_1}\ v^{\texttt{[]}}_{\gamma_2})$$

are constructed. So

$$\begin{array}{l} \mathcal{A}(\emptyset, \texttt{(:) [Int] ([] Int) ((:) [Int] ([] Int) ([] [Int]))}) \\ = \\ (\lambda\gamma_1.\, \lambda\gamma_2.\, \lambda v^{\texttt{(:)}}_{\gamma_2} : \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_2.\, \lambda v^{\texttt{[]}}_{\gamma_1} : \gamma_1.\, \lambda v^{\texttt{[]}}_{\gamma_2}.\, v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_1}\ (v^{\texttt{(:)}}_{\gamma_2}\ v^{\texttt{[]}}_{\gamma_1}\ v^{\texttt{[]}}_{\gamma_2})) \\ \texttt{[Int] [[Int]] ((:) [Int]) ([] Int) ([] [Int])} \end{array}$$

4.5 Properties and Implementation

The correctness of the list abstraction algorithm $\mathcal{A}$ can easily be proved from the correctness of the various component algorithms:

Lemma 4.50 (Termination)

$$\frac{\Gamma \vdash M : \tau}{\mathcal{A}(\Gamma, M)\ \textit{terminates}} \qquad \square$$

Proof Follows from Lemma 4.7 and Lemma 4.37. ■

Lemma 4.51 (Well-typedness)

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathcal{A}(\Gamma, M) : \tau} \qquad \square$$

Proof Follows from Lemmas 4.12, 4.37, 4.44, 4.46 and 4.49 and the type abs and term abs rule. ■

Lemma 4.52 (Correctness)

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash M \cong^{ctx} \mathcal{A}(\Gamma, M) : \tau} \qquad \square$$

Proof From Lemmas 4.11, 4.38, 4.43, 4.45 and 4.48 it follows that $\mathcal{A}(\Gamma, M)$ can be β-reduced in several steps to M. Together with Lemma 4.51, Lemma 2.16 implies that both terms are contextually equivalent. ■

The list abstraction algorithm is complete in the sense that it can abstract a list from a term if this is possible at all. The list replacement algorithm replaces every occurrence of a list type and a list constructor by a different variable. The type inference algorithm determines the principal consistent typing of this term (Lemma 4.37). Thus it determines which variables have to be list types and list constructors and hence cannot be abstracted and which variables can be free and hence be abstracted. Naturally it depends on the choice of type variables that are instantiated in the third phase, which of the lists that can be abstracted are finally abstracted. It depends on the purpose of list abstraction, that is, if we use it for preparing a producer for fusion or for the worker/wrapper split, which lists it is desirable to abstract.

The list abstraction algorithm is also efficient. It only consists of several traversals of the (modified) input term. The unification algorithm can work in linear time, if types are represented by directed acyclic graphs [PW78]. Then the type inference algorithm and hence the whole list abstraction algorithm also works in linear time. Note that it does not affect our type inference algorithm that Hindley-Milner type inference requires exponential time in the worst case (Section 11.3.5 of [Mit96]). The Hindley-Milner type inference algorithm is only exponential in the nesting depth of `let` definitions, because of its type generalisation step at `let` definitions, but otherwise it is linear. Our algorithm does not have this type generalisation step.

We implemented a prototype of the list abstraction algorithm in HASKELL. The translation of the formal description of the algorithm into HASKELL was straightforward and the prototype was successfully applied to several examples. To enable a later integration the implementation uses mostly the same data structures as are used in the Glasgow Haskell compiler. Instead of passing explicit substitutions we implemented type inference variables as updateable references (`IORef`s, cf. [LP94]) just as the type inference pass of the Glasgow Haskell compiler does. This choice leads to the mentioned efficient implementation of unification. Note that the list abstraction algorithm does not need the inferred substitution itself but only the result of applying it to the term and instance substitution which are computed by list replacement.

4.6 Alternative List Abstraction Algorithms

In [Chi99] we presented a different type inference algorithm which is based on the algorithm $\mathcal{M}$ [LY98] for the Hindley-Milner system instead of the algorithm $\mathcal{W}$. $\mathcal{M}$ takes not only a typing environment and a term as argument but also a type and returns only a substitution. The idea is that it takes a typing that contains type variables and returns the most general substitution that yields a valid typing. When abstracting a list producer a type inference algorithm based on $\mathcal{M}$ can notice early if abstraction from the list type is impossible. It can then abort the list abstraction algorithm and thus save time. However, when abstracting from several lists, as is necessary for the worker/wrapper split, this trick is no longer applicable. We prefer the algorithm $\mathcal{T}$ presented here also, because it has the nice property of not requiring any new type inference variables, in contrast to the algorithm presented in [Chi99].

The main problem we had to solve in this chapter was to assure that at the beginning of the third phase the list abstraction algorithm has a consistent typing of constructor variables. We chose to post-process the possibly inconsistent result of type inference with the algorithm $\mathcal{C}$. However, we can also assure consistency differently.

On the one hand we could modify the type inference algorithm, strictly speaking the unification algorithm, such that it always yields the most general consistent typing. Whenever the unification algorithm would bind a variable, it would subsequently perform a unification with the list substitution η to ensure consistency. However, this algorithm would be less modular and more difficult to prove correct. It is even doubtful that an improvement in performance would be gained.

On the other hand we could give the type inference algorithm a different input, in which the types of the constructor variables express the required relation between a list type and its element type. A list type $[\tau]$ is related to its element type τ simply by the fact that the list type $[\tau]$ = `[]` τ consists of the list type constructor `[]` being applied to the element type τ. So to preserve this relationship in the input to the type inference algorithm we can use **type constructor variables** that only replace list type constructors `[]`, not whole list types.

To outline the idea we reconsider the example from the beginning of the chapter:

$\vdash$ `(:) [Int] ([] Int) ((:) [Int] ([] Int) ([] [Int])) : [[Int]]`

List replacement gives

$$\begin{array}{l}
\{\texttt{v}_1^{\texttt{(:)}}{:}\forall\alpha.\,\alpha\rightarrow\pi_1\,\alpha\rightarrow\pi_1\,\alpha\texttt{,}\;\;\texttt{v}_2^{\texttt{(:)}}{:}\forall\alpha.\,\alpha\rightarrow\pi_2\,\alpha\rightarrow\pi_2\,\alpha\texttt{,}\\
\;\texttt{v}_1^{\texttt{[]}}{:}\forall\alpha.\,\pi_3\,\alpha\texttt{,}\;\;\texttt{v}_2^{\texttt{[]}}{:}\forall\alpha.\,\pi_4\,\alpha\texttt{,}\;\;\texttt{v}_3^{\texttt{[]}}{:}\forall\alpha.\,\pi_5\,\alpha\}\\
\vdash\;\texttt{v}_1^{\texttt{(:)}}\;\texttt{(}\pi_6\;\texttt{Int)}\;\texttt{(v}_1^{\texttt{[]}}\;\texttt{Int)}\;\texttt{(v}_2^{\texttt{(:)}}\;\texttt{(}\pi_7\;\texttt{Int)}\;\texttt{(v}_2^{\texttt{[]}}\;\texttt{Int)}\;\texttt{(v}_3^{\texttt{[]}}\;\texttt{(}\pi_8\;\texttt{Int))}
\end{array}$$

Here the π_i are type constructor variables. Note that we replace list constructors `(:)` and `[]` by constructor variables, not applications of list constructors, that is, `(:)` τ and `[]` τ. Hence the constructor variables have polymorphic types, just as the list constructors.

Subsequently, a type inference algorithm that instantiates type constructor variables instead of type variables yields the valid typing

$$\begin{array}{l}
\{\texttt{v}_1^{\texttt{(:)}}{:}\forall\alpha.\,\alpha\rightarrow\pi_1\,\alpha\rightarrow\pi_1\,\alpha\texttt{,}\;\;\texttt{v}_2^{\texttt{(:)}}{:}\forall\alpha.\,\alpha\rightarrow\pi_1\,\alpha\rightarrow\pi_1\,\alpha\texttt{,}\\
\;\texttt{v}_1^{\texttt{[]}}{:}\forall\alpha.\,\pi_2\,\alpha\texttt{,}\;\;\texttt{v}_2^{\texttt{[]}}{:}\forall\alpha.\,\pi_2\,\alpha\texttt{,}\;\;\texttt{v}_3^{\texttt{[]}}{:}\forall\alpha.\,\pi_1\,\alpha\}\\
\vdash\;\texttt{v}_1^{\texttt{(:)}}\;\texttt{(}\pi_2\;\texttt{Int)}\;\texttt{(v}_1^{\texttt{[]}}\;\texttt{Int)}\;\texttt{(v}_2^{\texttt{(:)}}\;\texttt{(}\pi_2\;\texttt{Int)}\;\texttt{(v}_2^{\texttt{[]}}\;\texttt{Int)}\;\texttt{(v}_3^{\texttt{[]}}\;\texttt{(}\pi_1\;\texttt{Int)))}\\
:\;\;\pi_1\;\texttt{(}\pi_2\;\texttt{Int)}
\end{array}$$

So the typing correctly states that two lists can be abstracted. In the third phase constructor variables of the same type are merged. (If there were constructor variables of type $\forall\alpha.\,\alpha\rightarrow\texttt{[}\alpha\texttt{]}\rightarrow\texttt{[}\alpha\texttt{]}$, respectively $\forall\alpha.\,\texttt{[}\alpha\texttt{]}$, they would be reinstantiated to the list constructors `(:)`, respectively `[]`).

$$\begin{array}{l}
\{\texttt{v}_1^{\texttt{(:)}}{:}\forall\alpha.\,\alpha\rightarrow\pi_1\,\alpha\rightarrow\pi_1\,\alpha\texttt{,}\;\;\texttt{v}_1^{\texttt{[]}}{:}\forall\alpha.\,\pi_2\,\alpha\texttt{,}\;\;\texttt{v}_3^{\texttt{[]}}{:}\forall\alpha.\,\pi_1\,\alpha\}\\
\vdash\;\texttt{v}_1^{\texttt{(:)}}\;\texttt{(}\pi_2\;\texttt{Int)}\;\texttt{(v}_1^{\texttt{[]}}\;\texttt{Int)}\;\texttt{(v}_1^{\texttt{(:)}}\;\texttt{(}\pi_2\;\texttt{Int)}\;\texttt{(v}_2^{\texttt{[]}}\;\texttt{Int)}\;\texttt{(v}_3^{\texttt{[]}}\;\texttt{(}\pi_1\;\texttt{Int)))}\\
:\;\;\pi_1\;\texttt{(}\pi_2\;\texttt{Int)}
\end{array}$$

Afterwards, we still need a fourth phase which transforms type constructor variables into type variables. This phase also replaces applications of constructor variables by constructor variables.

$$\begin{array}{l}
\{\texttt{v}_1^{\texttt{(:)}}{:}\gamma_2\rightarrow\gamma_1\rightarrow\gamma_1\texttt{,}\;\;\texttt{v}_1^{\texttt{[]}}{:}\gamma_2\texttt{,}\;\;\texttt{v}_3^{\texttt{[]}}{:}\gamma_1\}\\
\vdash\;\texttt{v}_1^{\texttt{(:)}}\;\texttt{v}_1^{\texttt{[]}}\;\texttt{(v}_1^{\texttt{(:)}}\;\texttt{v}_1^{\texttt{[]}}\;\texttt{v}_3^{\texttt{[]}}\texttt{)}\\
:\;\;\gamma_1
\end{array}$$

Finally, we transform this typing into list abstracted form:

$$\begin{array}{l}
\texttt{(}\lambda\gamma_1.\,\lambda\gamma_2.\,\lambda\texttt{v}_1^{\texttt{(:)}}:\gamma_2\rightarrow\gamma_1\rightarrow\gamma_1.\,\lambda\texttt{v}_1^{\texttt{[]}}:\gamma_2.\,\lambda\texttt{v}_3^{\texttt{[]}}:\gamma_1.\;\;\texttt{v}_1^{\texttt{(:)}}\;\texttt{v}_1^{\texttt{[]}}\;\texttt{(v}_1^{\texttt{(:)}}\;\texttt{v}_1^{\texttt{[]}}\;\texttt{v}_3^{\texttt{[]}}\texttt{))}\\
\quad\texttt{[[Int]] [Int] ((:) [Int]) ([] Int) ([] [Int])}
\end{array}$$

Note that for this final step we need to know which list variable γ abstracts which list type $\texttt{[}\tau\texttt{]}$. So in the fourth phase some list substitution like η^∞ must be constructed.

This list abstraction algorithm based on type constructor variables looks more elegant than the one we presented in the previous sections of this chapter. It only requires a simple type inference algorithm without any post-processing to ensure consistency. This algorithm might be more suitable for some extensions discussed in Chapter 6. However, the algorithm is also more complex. Especially, it requires the notion of type constructor variable and a fourth phase which transforms type constructor variables into type variables.

Chapter 5

The Deforestation Algorithm

After having illustrated our deforestation method at the hand of examples in Chapter 3 and having studied type-inference based list abstraction in detail in Chapter 4 we here finally present our deforestation algorithm. We discuss both its correctness and its effect on the performance of programs. To quantify the improvement that can be gained by our deforestation method we finally measure its effect on the well-known n-queens program.

5.1 The Worker/Wrapper Split Algorithm

For the worker/wrapper scheme each definition of a list-producing function has to be split into a worker and a wrapper definition. This is straightforward for non-recursive definitions but unfortunately not so for recursive ones.

5.1.1 Non-Recursive Definitions

For a non-recursive definition nearly the whole worker/wrapper split is performed by the list abstraction algorithm.

Suppose we have the non-recursive definition of a list-producing function f

$$\begin{aligned} f \ &:\ \forall\overline{\alpha}.\tau \\ &=\ \lambda\overline{\alpha}.\,M \end{aligned}$$

where M is a term abstraction. Remember that we insert the new λ-abstractions between the type abstractions and the term abstractions (Section 3.3). We apply the list abstraction algorithm to the term M and obtain

$$f \ = \ \lambda\overline{\alpha}.\ (\lambda\overline{\gamma}.\,\lambda\overline{\mathtt{v}_\gamma^{(:)} : \tau\rightarrow\gamma\rightarrow\gamma}.\,\lambda\overline{\mathtt{v}_\gamma^{[]} : \gamma}.\,M') \quad \overline{[\tau_\gamma]}\ \ \overline{((:)\ \tau_\gamma)}\ \ \overline{([]\ \tau_\gamma)}$$

Then we use type application expansion

$$f \ = \ \lambda\overline{\alpha}.\ (\lambda\overline{\alpha}.\,\lambda\overline{\gamma}.\,\lambda\overline{\mathtt{v}_\gamma^{(:)} : \tau\rightarrow\gamma\rightarrow\gamma}.\,\lambda\overline{\mathtt{v}_\gamma^{[]} : \gamma}.\,M') \quad \overline{\alpha}\ \ \overline{[\tau_\gamma]}\ \ \overline{((:)\ \tau_\gamma)}\ \ \overline{([]\ \tau_\gamma)}$$

and inverse inlining and `let` elimination (cf. Section 2.7) to obtain the worker and wrapper definition:

$$
\begin{aligned}
fW\ &:\ \forall\overline{\alpha}.\forall\overline{\gamma}.\overline{\tau_\gamma\to\gamma\to\gamma}\to\overline{\gamma}\to\tau' \\
&=\ \lambda\overline{\alpha}.\ \lambda\overline{\gamma}.\ \overline{\lambda\texttt{v}^{\texttt{(:)}}_\gamma : \tau_\gamma\to\gamma\to\gamma}.\ \overline{\lambda\texttt{v}^{\texttt{[]}}_\gamma : \gamma}.\ M'
\end{aligned}
$$

$$
\begin{aligned}
f\ &:\ \forall\overline{\alpha}.\tau \\
&=\ \lambda\overline{\alpha}.\ fW\ \ \overline{\alpha}\ \ \overline{\texttt{[}\tau_\gamma\texttt{]}}\ \ \overline{\texttt{((:)}\ \tau_\gamma\texttt{)}}\ \ \overline{\texttt{([]}\ \tau_\gamma\texttt{)}}
\end{aligned}
$$

Here τ' is the type of M' in its typing environment. Because of the type annotations in F this type can easily be obtained from M' with the type rules. Alternatively, we could extend the list abstraction algorithm to return τ'.

If the list abstraction algorithm cannot abstract any list, then no worker/wrapper split takes place.

From the correctness of the list abstraction algorithm, Lemma 4.52, and from the correctness of the small transformations that we performed, Lemma 2.16, follows the semantic correctness of the worker/wrapper split.

For example, consider the well-known non-recursive definition of `map`:

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

With the list abstraction algorithm we obtain the following definition:

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

Then we use type application expansion

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

and inverse inlining and `let` elimination to obtain the worker and wrapper definitions which we already know:

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

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

Similarly the worker/wrapper split for the definitions of `(++)`, `concat` and `unzip` of Section 3.3 are performed.

5.1.2 Recursive Definitions

Consider the recursive definition of the function `enumFrom`, which produces an infinite list of increasing integers.

$$
\begin{aligned}
&\texttt{enumFrom : Int}\to\texttt{Int} \\
&\qquad\quad = \lambda\texttt{l:Int.\,(:) Int l (enumFrom (l+1))}
\end{aligned}
$$

We can rewrite it as

```
enumFrom : Int→Int
         = let go : Int→Int
                  = λl:Int.(:) Int l (go (l+1))
           in go
```

The new definition of `enumFrom` is not recursive and we can hence apply the worker/wrapper split algorithm that we described in the previous section.

With the list abstraction algorithm we obtain the definition

```
enumFrom : Int→Int
         = (λγ.λv^{(:)}:Int→γ→γ.let go : Int→Int
                                      = λl:Int.v^{(:)} l (go (l+1))
                                 in go )
           [Int] ((:) Int)
```

and subsequently we apply inverse inlining and `let` elimination to obtain the worker and wrapper definition

```
enumFromW : ∀γ.(Int→γ→γ)→Int→γ
          = λγ.λv^{(:)}:Int→γ→γ.let go : Int→Int
                                     = λl:Int.v^{(:)} l (go (l+1))
                                in go

enumFrom : Int→Int
         = enumFromW [Int] ((:) Int)
```

The definition of the worker `enumFromW` is unusual. From the discussion in Chapter 3 we would expect the definition

```
enumFromW : ∀γ.(Int→γ→γ)→Int→γ
          = λγ.λv^{(:)}:Int→γ→γ.λl:Int.v^{(:)} l (enumFromW v^{(:)} (l+1))
```

However, the definition that we derived is preferable. Passing the additional argument $\mathtt{v}^{(:)}$ in all recursive calls is costly. Actually, this is less so because of the additional term application reduction in every recursive call but mostly, because the size of a heap cell for an unevaluated term is proportional to the number of its free variables. So a heap cell for $\mathtt{enumFromW}\ \mathtt{v}^{(:)}\ \mathtt{(l+1)}$ requires more space than a heap cell for `go (l+1)`. In our cost model (Section 2.9) we did not consider the size of heap cells. In Section 7.1 of [San95] this effect on costs is discussed together with the presentation of the **static argument transformation**. An argument in a recursive function definition is **static**, if it is passed unchanged in all recursive calls. The static argument transformation transforms the latter definition of `enumFromW` into the former.

In general, let

$$
\begin{aligned}
f\ &:\ \forall\overline{\alpha}.\tau \\
&= \lambda\overline{\alpha}.M
\end{aligned}
$$

be a recursive definition such that f occurs in M only in the form $f\ \overline{\alpha}$. Then we apply the static argument transformation for the static type arguments $\overline{\alpha}$, that is, we rewrite the definition as

$$
\begin{aligned}
f\ &:\ \forall\overline{\alpha}.\tau \\
&= \lambda\overline{\alpha}.\mathtt{let}\ go\ :\ \tau\ =\ M'\ \mathtt{in}\ go
\end{aligned}
$$

where M' is obtained from M by replacing all occurrences of $f\ \overline{\alpha}$ by the term variable *go*. The new definition of f is not recursive and we can hence apply the worker/wrapper split algorithm of the previous section.

We split the definition of `enumFrom` by this algorithm. As a second, more complex example we split the recursive definition of the function `tails` from Section 3.3.3.

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

First, we apply the static argument transformation to obtain

```
tails : ∀α.[α] → [[α]]
      = λα.let go : [α]→[[α]]
                  = λxs:[α].case xs of
                      []    ↦ (:) [α] ([] α) ([] [α])
                      y:ys  ↦ (:) [α] xs (go ys)
           in go
```

and, second, we split this non-recursive definition into a worker and a wrapper definition:

```
tailsW : ∀α.∀γ.([α]→γ→γ)→ γ →[α]→ γ
       = λα. λγ. λv^{(:)} : [α] →γ→γ. λv^{[]} :γ.
           let go : [α]→ γ
                  = λxs:[α] . case xs of
                      []    ↦ v^{(:)} ([] α) v^{[]}
                      y:ys  ↦ v^{(:)} xs (go ys)
           in go

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

In [Chi00] we proposed a slightly different method for splitting recursive definitions such as the ones considered here. This method, which was inspired by how type inference algorithms for the Hindley-Milner type system handle the `let` construct, however, requires an extension of the list abstraction algorithm.

5.1.3 Polymorphic Recursion

The algorithm which we just described is only applicable to definitions which are not polymorphically recursive. A definition is **polymorphically recursive** if its type arguments are not static, that is, a recursive call uses type arguments which differ from the abstracted type variables. Polymorphically recursive definitions are rare in HASKELL. Unfortunately, the algorithm which we just described also cannot achieve the worker/wrapper split of the definition of `reverse` which we described in Section 3.4. The definition of `reverse` is not polymorphically recursive, but the definition of its worker `reverseW` (see the definition before deforestation inside the worker in Section 3.4). Obviously, functions that consume their own result need such polymorphically recursive worker definitions.

Typeability in the Hindley-Milner type system with polymorphic recursion is only semi-decidable [Hen93, KTU93], that is, there are algorithms which do infer the most general type of a term within the Hindley-Milner type system with polymorphic recursion if it is typeable. However, if the term is not typeable, then these algorithms may diverge. Fortunately, the input of the worker/wrapper split algorithm is typeable, we only try to find a more general type than we have.

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

```
reverseW$_0$ : $\forall\alpha.\forall\gamma_1.\forall\gamma_2.(\alpha\to\gamma_1\to\gamma_1)\to\gamma_1\to(\alpha\to\gamma_2\to\gamma_2)\to\gamma_2\to(\gamma_1\to\gamma_2)$
```

We define a well-typed wrapper for this hypothetical worker:

```
reverse$_0$ = $\lambda\alpha$.reverseW$_0$ $\alpha$ [$\alpha$] [$\alpha$] ((:) $\alpha$) ([] $\alpha$) ((:) $\alpha$) ([] $\alpha$)
```

Then we replace in the original definition body of `reverse` (without $\lambda\alpha$.) all occurrences of `reverse` by this wrapper:

```
$\lambda$xs:[$\alpha$]. case xs of
  []   $\mapsto$ [] $\alpha$
  y:ys $\mapsto$ appW $\alpha$ [$\alpha$] ((:) $\alpha$)
            (reverseW$_0$ $\alpha$ [$\alpha$] [$\alpha$] ((:) $\alpha$) ([] $\alpha$) ((:) $\alpha$) ([] $\alpha$) ys)
            ((:) $\alpha$ y ([] $\alpha$))
```

So we have a term with a typing environment for all its free variables; to `reverseW` it assigns the most general worker type, which we just constructed. For this term list abstraction yields:

```
($\lambda\gamma$. $\lambda$v$^{(:)}$ : $\alpha\to\gamma\to\gamma$. $\lambda$v$^{[]}$ : $\gamma$.
  $\lambda$xs:[$\alpha$]. case xs of
    []   $\mapsto$ v$^{[]}$
    y:ys $\mapsto$ appW $\alpha$ $\gamma$ v$^{(:)}$
              (reverseW$_0$ $\alpha$ [$\alpha$] [$\alpha$] ((:) $\alpha$) ([] $\alpha$) ((:) $\alpha$) ([] $\alpha$) ys)
              (v$^{(:)}$ y v$^{[]}$) )
 [$\alpha$] ((:) $\alpha$) ([] $\alpha$)
```

Now we can construct the following first approximation of a worker definition:

```
reverseW$_1$ : $\forall\alpha.\forall\gamma.(\alpha\to\gamma\to\gamma)\to\gamma\to$ [$\alpha$] $\to\gamma$
           = $\lambda\alpha$. $\lambda\gamma$. $\lambda$v$^{(:)}$:$\alpha\to\gamma\to\gamma$. $\lambda$v$^{[]}$:$\gamma$. $\lambda$xs:[$\alpha$]. case xs of
               []   $\mapsto$ v$^{[]}$
               y:ys$\mapsto$ appW $\alpha$ $\gamma$ v$^{(:)}$
                       (reverseW$_0$ $\alpha$ [$\alpha$] [$\alpha$] ((:) $\alpha$) ([] $\alpha$) ((:) $\alpha$) ([] $\alpha$) ys)
                       (v$^{(:)}$ y v$^{[]}$)
```

Subsequently we repeat these steps, this time under the assumption that reverseW$_1$ has the type $\forall\alpha.\forall\gamma.(\alpha\to\gamma\to\gamma)\to\gamma\to$ [α] $\to\gamma$. So the well-typed wrapper for the worker reverseW$_1$ is

```
reverse$_1$ = $\lambda\alpha$.reverseW$_1$ $\alpha$ [$\alpha$] ((:) $\alpha$) ([] $\alpha$)
```

the input to the list abstraction algorithm is

```
$\lambda$xs:[$\alpha$]. case xs of
  []   $\mapsto$ [] $\alpha$
  y:ys $\mapsto$ appW $\alpha$ [$\alpha$] ((:) $\alpha$)
            (reverseW$_1$ $\alpha$ [$\alpha$] ((:) $\alpha$) ([] $\alpha$) ys)
            ((:) $\alpha$ y ([] $\alpha$))
```

and its result

```
$(\lambda\gamma.\,\lambda$v$^{(:)}$ : $\alpha\to\gamma\to\gamma.\,\lambda$v$^{[]}$ : $\gamma$.
  $\lambda$xs:[$\alpha$]. case xs of
    []     $\mapsto$ v$^{[]}$
    y:ys  $\mapsto$ appW $\alpha$ $\gamma$ v$^{(:)}$
               (reverseW$_1$ $\alpha$ [$\alpha$] ((:) $\alpha$) ([] $\alpha$) ys)
               (v$^{(:)}$ y v$^{[]}$) )
 [$\alpha$] ((:) $\alpha$) ([] $\alpha$)
```

From this we can construct the second approximation of a worker definition:

```
reverseW$_2$  : $\forall\alpha.\forall\gamma.(\alpha\to\gamma\to\gamma)\to\gamma\to[\alpha]\to\gamma$
            = $\lambda\alpha.\,\lambda\gamma.\,\lambda$v$^{(:)}$:$\alpha\to\gamma\to\gamma.\,\lambda$v$^{[]}$:$\gamma.\,\lambda$xs:[$\alpha$]. case xs of
               []    $\mapsto$ v$^{[]}$
               y:ys$\mapsto$ appW $\alpha$ $\gamma$ v$^{(:)}$
                        (reverseW$_1$ $\alpha$ [$\alpha$] ((:) $\alpha$) ([] $\alpha$) ys)
                        (v$^{(:)}$ y v$^{[]}$)
```

The whole process iterates until the derived worker type is stable, that is input and output type are identical. For our example the second iteration already yields the same worker type as the first. So the definitions of `reverseW`$_2$ and `reverse`$_1$ are the definitions of the worker and wrapper of the definition of `reverse`.

In general, the worker/wrapper split stops latest after $n+1$ iterations, where n is the number of list types in the type of the original function. This follows analogously to the syntactic completeness in [Myc84]. Whereas intuitively the semantic correctness of the result of this extended worker/wrapper split algorithm is clear, a formal proof is hard. We believe that correctness can be proved with the techniques presented in [San96b, San97].

As we discussed in Section 3.4.2, it is debatable if deforestation of recursively defined functions that consume their own results is desirable. Depending on our decision we may either use the worker/wrapper split algorithm described in this section or the one described in the previous section.

If we use the algorithm described in this section, then it is useful to subsequently apply the static argument transformation. As we saw in the preceding section, the static argument transformation improves the definitions of workers which are not polymorphically recursive, such as `enumFromW` and `tailsW`.

5.1.4 Traversal Order

The worker/wrapper split algorithm splits each `let`-defined block of mutually recursive definitions separately. In the example of `concat` in Section 3.3.2 the split was only possible after the wrapper of `(++)` had been inlined. Hence the split algorithm must traverse the program in top-down order and inline wrappers in the remaining program directly after they were derived.

Additionally, definitions can be nested, that is, the definition body of a `let` construct can contain another `let` construct. Here the inner definitions have to be split first. Their wrappers can then be inlined in the body of the outer definition and thus enable the abstraction of more lists from the outer definition.

5.2 The Complete Deforestation Algorithm

First, our worker/wrapper split algorithm splits all definitions of list-producing functions of a program into worker and wrapper definitions. The wrapper definitions are non-recursive and small enough to be inlined unconditionally everywhere, also across module boundaries. They transfer the information needed for list abstraction in the split algorithm and the actual deforestation algorithm.

The actual deforestation algorithm searches for occurrences of full applications of `foldr`, that is `foldr` τ_1 τ_2 $M^{(:)}$ $M^{[]}$ $M^{\mathtt{P}}$, abstracts the result list from the producer $M^{\mathtt{P}}$ and then, if successful, directly applies the short cut fusion rule.

For example, suppose it finds the following potentially fusible term:

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

List abstraction of the producer is successful and gives:

```
...
 foldr Bool Bool (&&) True
   ((λγ.λv^(:):Bool→ γ → γ.λv^[]:γ.
     (fst γ [Char] (unzipW Bool Char γ [Char]
       v^(:) v^[] ((:) Char) ([] Char) zs)) )
     [Bool] ((:) Bool) ([] Bool) )
...
```

Then the short cut fusion rule is applied:

```
...
  (λγ.λv^(:):Bool→ γ → γ.λv^[]:γ.
    (fst γ [Char] (unzipW Bool Char γ [Char]
      v^(:) v^[] ((:) Char) ([] Char) zs)) )
    Bool (&&) True
...
```

Further optimisations may be obtained by a subsequent standard simplification pass which performs β-reduction.

```
...
  fst Bool [Char] (unzipW Bool Char Bool [Char]
    (&&) True ((:) Char) ([] Char) zs)
...
```

In principle, the list abstraction algorithm could directly replace the right list constructors by $M^{(:)}$ and $M^{[]}$, thereby avoiding completely the explicit construction of the λ-abstraction and subsequent β-reduction. However, the modular separation of the two tasks is desirable, because the worker/wrapper split algorithm uses the same list abstraction algorithm. An optimising compiler will be able to perform β-reduction anyway.

Similarly, we separate the worker/wrapper split algorithm from the actual deforestation algorithm. Deforestation brings together subterms of producer and consumer which previously were separated by the intermediate list. This may lead to new deforestation opportunities and hence it may be useful to repeat the actual deforestation algorithm several times, whereas the worker/wrapper split is only performed once.

5.2.1 Avoiding Inefficient Workers

As Gill already noticed [Gil96], there is a substantial performance difference between calling a function as originally defined (for example `map` τ' τ) and calling a worker with list constructors as arguments (for example `mapW` τ' τ `[`τ`]` `((:)`τ`)` `([]`τ`)`). The latter call is considerably more expensive.

In our worker/wrapper split algorithm we demanded that in a definition

$$\begin{aligned} f &: \forall\overline{\alpha}.\tau \\ &= \lambda\overline{\alpha}.M \end{aligned}$$

which we want to split into a worker and a wrapper definition the term M is a term application. Thus it is a value and not a term that may be evaluated once to a value that is then shared between all uses of the function f.[1] The definition body of the wrapper

$$\begin{aligned} f &: \forall\overline{\alpha}.\tau \\ &= \lambda\overline{\alpha}.\ fW\ \ \overline{\alpha}\ \ \overline{\texttt{[}\tau_\gamma\texttt{]}}\ \ \overline{\texttt{((:)}\ \tau_\gamma\texttt{)}}\ \ \overline{\texttt{([]}\ \tau_\gamma\texttt{)}} \end{aligned}$$

is not a value but an application that evaluates to the value M. So we know that evaluation of f requires only a few steps. These evaluation steps are duplicated when the wrapper f is inlined. Furthermore, workers have additional arguments for the abstracted list constructors. We already discussed in Section 5.1.2 how the static argument transformation can move these arguments out of the recursion. Only to polymorphically recursive workers, that is, workers of functions that recursively consume their own result, this transformation is not applicable. Finally, because the list constructors are abstracted, they can only be accessed in the worker by the VAR rule, whereas they are directly present in the original unsplit definition.

So a worker/wrapper split decreases performance. The effect can only be made good by deforestation. Hence a worker should never be called when it is not used for deforestation, that is, when it has only list types and list constructors as arguments.

A simple solution is to replace after deforestation all calls to workers that still have only list types and list constructors as arguments by calls to more efficiently defined functions. This could be the original, unsplit definition of the function. Better may be a version of the worker that is specialised to the list types and list constructors. Thus this definition can profit from any optimisations, especially deforestation, that were performed inside the worker definition. Note that we only derive one specialised definition for every worker.

Alternatively, we could change the deforestation algorithm so that it only replaces the call to a list producer by a wrapper, if this is necessary to enable deforestation. Basically, we could use our type-inference based method of Section 3.2 for detecting functions that need to be inlined to inline wrappers only where necessary.

The worker/wrapper scheme increases code size through the introduction of wrapper and worker definitions. However, this increase is naturally bounded in contrast to the code increase that is caused by our original list abstraction algorithm with inlining. Note that the definitions of workers that are not needed for deforestation can be removed by standard dead code elimination after worker specialisation has been performed.

5.2.2 Consumers in `foldr` form

The HASKELL prelude, the library of standard functions that are available in any HASKELL program [PH+99], contains many functions that consume lists. Several of these are de-

[1] Note that in discussions of the effect on program performance we have to ignore type abstractions and applications, because they disappear during the translation into the language G of our cost model.

fined in terms of `foldr`. [GLP93, GP94, Gil96] also give definitions in terms of `foldr` for many others, which we do not list again here. However, besides `foldr` the prelude also contains a few other higher-order functions that consume lists, namely `foldl`, `foldr1` and `foldl1`. The function `foldl` uses an accumulating parameter to combine all elements of a list.

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

Many list-consuming functions are naturally defined in term of `foldl`, for example the linear version of `reverse`:

```
reverse : ∀α.[α]→[α]
        = λα.foldl α [α] (λys:[α]. λy:α. (:) α y ys) ([] α)
```

As shown in [Gil96], `foldl` can be expressed in terms of `foldr`:

```
foldl : ∀α.∀β.(β→α→β)→ β →[α]→ β
      = λα. λβ. λc:β→α→β. λn:β. λxs:[α].
          foldr α (β→β) (λy:α. λfys:β→β. λn:β. fys (f n y))
            (λn:β. n) xs n
```

For every element of the list `xs` this definition allocates a term of functional type $(\beta \to \beta)$ on the heap. All these intermediate functions are composed so that the term `foldr` ... `xs` produces a function that is finally applied to `n`. So the definition of `foldl` in terms of `foldr` is less efficient then the direct definition given before. Therefore the definition in terms of `foldr` should not be chosen as standard definition of `foldl` in the Haskell prelude. Instead, the deforestation algorithm searches for applications of `foldl` just as it searches for applications of `foldr`. If list abstraction of the list argument is successful, then the definition of `foldl` in terms of `foldr` is used to fuse consumer and producer. Because this definition introduces intermediate functions but fusion removes the intermediate list, the whole transformation keeps execution costs constant. It is nonetheless useful to perform this transformation, because subterms of producer and consumer that previously were separated by the intermediate list are brought together and thus new opportunities for other optimising transformations arise.

The higher-order functions `foldl1` and `foldr1` are only defined for non-empty argument lists. Besides the definitions in the Haskell report [PH$^+$99], there also exist the following equivalent definitions. Note that `fold` is not a standard function.

```
foldl1 : ∀α.(α→α→α)→[α]→ α
       = λα. λc:α→α→α. λxs:[α]. case xs of
           y:ys↦ foldl α c y ys

foldr1 : ∀α.(α→α→α)→[α]→ α
       = λα. λc:α→α→α. λxs:[α]. case xs of
           y:ys↦ fold α c y ys

fold : ∀α.(α→α→α)→ α →[α]→ α
     = λα. λc:α→α→α. λn:α. λxs:[α]. case xs of
         []   ↦ n
         y:ys ↦ c n (fold α c y ys)
```

We do not define foldl1 and foldr1 directly in terms of foldr. However, their definitions use foldl, respectively fold, which can be defined in terms of foldr. The definitions of foldl1 and foldr1 have to be removed by other transformations than fusion.

```
fold : ∀α.(α→α→α)→ α →[α]→ α
     = λα. λc:α→α→α. λn:α. λxs:[α].
         foldr α (α→α) (λy:α. λfys:α→α. λn:α. f n (fys y))
           (λn:α. n) xs n
```

So besides foldr the deforestation algorithm knows foldl and fold. If their list arguments can be list abstracted, then their definitions in terms of foldr are used for fusion.

Contextual equality of the original definitions and the definitions in terms of foldr can be proved by giving appropriate bisimulations, which we mentioned in Section 2.7.

5.2.3 Improvement by Short Cut Deforestation

To illustrate how short cut deforestation improves performance we compare the evaluation steps of a term with intermediate list and its deforested counterpart. We consider a simplified version of the definition body of the function any, which we discussed in the introduction. The following F-program combines the elements of a constant boolean list with the logical or operator.

```
let 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)
    (||) : Bool→Bool→Bool
         = λx:Bool.λy:Bool. case x of
             True ↦ True
             False↦ y
in foldr Bool Bool (||) False ((:) Bool False ((:) Bool True ([] Bool)))
```

Our short cut deforestation method transforms this F-program into the following F-program:

```
let (||) : Bool→Bool→Bool
         = λx:Bool.λy:Bool. case x of
             True ↦ True
             False↦ y
in (||) False ((||) True False)
```

We translate both programs by * into a corresponding G-program (cf. Section 2.9.1). To avoid some trivial evaluation steps we simplify the let structure of the G-terms. Figure 5.1 shows the evaluation of the G-term of the original F-program and Figure 5.2 shows the evaluation of the G-term of the deforested F-program. We note that the evaluation of the original program causes the allocation of an intermediate list on the heap. The evaluation of the deforested program takes fewer steps, mainly because no such intermediate list is allocated.

Naturally, further optimisations are desirable. For example, the allocation of the boolean value of the result, which occurs in both cases, could be avoided by a transformation that uses the fact that (||) is strict. However, in the case of the original program

```
{} : let true = True
         false = False
         foldr = λc.λn.λxs. case xs of
                   {[] ↦ n, y:ys ↦ let z = foldr c n ys in c y z}
         (||) = λx.λy. case x of {True ↦ True, False ↦ y}
     in let us = (let vs = (let zs = [] in (:) true zs) in (:) false vs)
        in foldr (||) false us
 {true↦True, false↦False, foldr↦..., (||)↦...}
  : let us =... in foldr (||) false us
  {..., us↦let vs = ... in (:) false vs} : foldr (||) false us
     "  : foldr (||) false
            ⋮
     "  : λxs.case xs of {[]↦false,y:ys↦let z = ... in (||) y z}
     "  : case us of {[]↦false,y:ys↦let z = ... in (||) y z}
      "  : us
             ⋮
    {..., us↦(:) false vs, vs↦let zs = [] in (:) true zs} : (:) false vs
      "  : let z = foldr (||) false vs in (||) false z
     { ... , z↦foldr (||) false vs} : (||) false z
        "  : (||) false
               ⋮
        "  : λy.case false of {True↦True, False↦y}
        "  : case false of {True↦True, False↦z}
         "  : false
                ⋮
         "  : False
         "  : z
          "  : foldr (||) false vs

                 ⋮  reduces to (||) true z' which reduces to True
          {true↦True, false↦False, foldr↦..., (||)↦...,
           us↦(:) false vs, vs↦(:) true zs, zs↦[],
           z↦foldr (||) false vs, z'↦foldr (||) false zs} : True
         {..., z↦True} : True
        "  : True
       "  : True
      "  : True
     "  : True
    "  : True
   "  : True
{true↦True, false↦False, foldr↦..., (||)↦...,
 us↦(:) false vs, vs↦(:) true zs, zs↦[],
 z↦True, z'↦foldr (||) false zs} : True
```

Figure 5.1: Evaluation derivation of program with intermediate list

```
{} : let true = True
         false = False
         (||) = λx.λy. case x of {True ↦ True, False ↦ y}
     in let z = or true false in or false z
┌{true↦True, false↦False, (||)↦λx.λy.case x of {True↦True, False↦y}}
│ : let z = or true false in or false z
│┌{..., z↦or true false} : or false z
││┌  "  : or false
│││        ⋮
││└  "  : λy. case false of {True↦True, False↦y}
││┌  "  : case false of {True↦True, False↦z}
│││┌  "  : false
││││[  "  : False
│││└  "  : False
│││┌  "  : z
││││┌  "  : or true false
│││││┌  "  : or true
││││││        ⋮
│││││└  "  : λy. case true of {True↦True, False↦y}
│││││┌  "  : case true of {True↦True, False↦false}
││││││┌  "  : true
│││││││[  "  : True
││││││└  "  : True
││││││[  "  : True
│││││└  "  : True
││││└  "  : True
│││└{...,z↦True} : True
││└  "  : True
│└  "  : True
└  "  : True
{true↦True, false↦False, (||)↦λx.λy.case x of {True↦True, False↦y},
 z↦True} : True
```

Figure 5.2: Evaluation derivation of the deforested program

such a transformation would be much more difficult, because the unwanted heap cells are allocated by the definition of foldr. The strictness transformation implemented in the Glasgow Haskell compiler [PL91] is only able to improve the deforested program.

5.2.4 No Duplication of Evaluation — Linearity is Irrelevant

Unfold/fold deforestation methods, which we will discuss in Section 7.3, have to take care to not to duplicate evaluation of some terms and thus increase costs. Consider the following consumer. It is defined in Haskell, because we only argue informally in this section.

```
mkTree :: [a] -> Tree a
mkTree []     = Leaf
mkTree (x:xs) = Node x (mkTree xs) (mkTree xs)
```

For example, mkTree [4,2] yields Node 4 (Node 2 Leaf Leaf) (Node 2 Leaf Leaf). The consumer traverses parts of its argument several times to produce its complete result. Our lazy evaluation semantics shows that in mkTree P the producer P, which may be expensive to evaluate, is shared and hence nonetheless only evaluated once.

Naïve deforestation of mkTree P leads to a term in which the remaining part of the producer is evaluated as often as the list is traversed in the original term. So deforestation may lead to an exponential increase of costs. Hence unfold/fold deforestation algorithms usually require their input to be linear, that is, every variable occurs at most once in a definition body.

For short cut deforestation we need mkTree to be defined in terms of foldr. As intermediate step we define

```
mkTree []     = Leaf
mkTree (x:xs) = let r = mkTree xs in Node x r r
```

and then obtain

```
mkTree = foldr (\x r -> Node x r r) Leaf
```

If the list abstraction algorithm successfully processes the producer P, then the short cut fusion rule removes the intermediate list. There is no duplication of evaluation, because a foldr consumer only traverses its list argument once. This property of foldr ensures that we do not have to worry about linearity and possible duplication of evaluation. So the important point is that we have the consumer in foldr form.

To stress this point, we consider the following definition with a nested recursive call:

```
f :: [a] -> [a] -> [a]
f [] zs     = zs
f (x:xs) zs = f xs (x : f xs zs)
```

For example, f [1,2,3] [] yields [3,2,3,1,3,2,3]. We first transform this definition into

```
f []     = \zs -> zs
f (x:xs) = \zs -> let r = f xs in r (x : r zs)
```

to finally obtain

```
f = foldr (\x r -> \zs -> r (x : r zs)) (\zs -> zs)
```

In both examples the definition that we gave as intermediate step differs from the original definition just in that respect that the list argument is traversed only once. We performed a kind of common subexpression elimination to obtain this form. Although common subexpression elimination has the beneficial effect of avoiding repeated computation, it may unfortunately unboundedly increase space usage and thus indirectly also garbage collection time [Chi98]. This problem does not concern us, because our deforestation method expects consumers to be already in `foldr` form. However, deforestation methods that automatically transform a general recursive definition into `foldr` form, such as warm fusion (Section 7.1.3) and hylomorphism fusion (Section 7.2), have to take care that they do not increase space and runtime costs by common subexpression elimination.

5.3 Measuring an Example: Queens

To quantify the improvement that can be gained by our deforestation method we measure its effect on the well-known n-queens program, using the Glasgow Haskell compiler [GHC].

A solution to the n-queens problem is a placement of n queens on an $n \times n$ chessboard so that no two queens hold each other in check. The HASKELL program in Figure 5.3 computes all solutions to the ten queens problem. It was adapted from [BW87], Section 6.5.1, and was already used in [GLP93, Gil96] to measure the effect of the original short cut deforestation algorithm. Ten instead of the traditional eight queens were chosen to increase the runtime and thus obtain reliable timings. Because printing all solutions to the ten queens problem would mean that I/O takes up most of the time, evaluation is simply forced by adding up all positions.

The original queens program uses list comprehensions. These are a powerful form of syntactic sugar. There are well-known techniques for translating list comprehensions into recursive functions such that only one list cell is constructed for each element of the result [Aug87, Wad87]. However, as discussed in [GLP93, Gil96], a good deforestation method makes these sophisticated translation techniques for list comprehensions unnecessary. In the HASKELL report [PH$^+$99] the semantics of list comprehensions is given by a straightforward translation. This translation introduces intermediate lists that will be removed by deforestation. Applying the translation to the original program of Figure 5.3 gives the program in Figure 5.4.

The desugared queens program directly or indirectly uses many standard list functions. The definitions of all standard list functions that are relevant for deforestation are listed in Appendix B. Our deforestation algorithm splits the definitions of all list-producing functions into worker and wrapper definitions and performs inlining of consumers and deforestation. Figure 5.5 gives the result of applying our deforestation algorithm to the desugared queens program. This deforested program contains only those transformed standard list functions which are used by `queens` and `safe` and which were not inlined. Appendix B additionally lists the split and deforested versions of all other standard list functions. Note that for example the definitions of `concatMapW` and `enumFromToW` were deforested.

First, deforestation removes all the intermediate lists that were introduced by the desugaring of list comprehensions. In the definition of `queensW` we note additionally, that the list `[1..10]` is no longer constructed. Furthermore, the original definition of `queens` recursively consumes its own result. All intermediate lists arising from this recursive consumption are removed, similarly to the example `inits` which we discussed in Section 3.4.1. The worker `queensW` abstracts both nested result lists. However, only the abstraction of the outer list is used for deforestation; `queensW` is always called with `(:)` and `[]` as first

```
main = (print . sum . concat . queens) 10

queens :: Int -> [[Int]]
queens 0 = [[]]
queens m = [p ++ [n] | p <- queens (m-1), n <- [1..10], safe p n]

safe :: [Int] -> Int -> Bool
safe p n = and [(j /= n) && (i+j /= m+n) && (i-j /= m-n)
               |(i,j) <- zip [1..] p]
  where
  m = length p + 1
```

Figure 5.3: Original queens program

```
main = (print . sum . concat . queens) 10

queens :: Int -> [[Int]]
queens 0 = [[]]
queens m = concatMap
             (\p -> concatMap
               (\n -> if safe p n then [p ++ [n]] else [])
               (enumFromTo 1 10))
             (queens (m-1))

safe :: [Int] -> Int -> Bool
safe p n = and
  (concatMap (\(i,j) -> [(j /= n) && (i+j /= m+n) && (i-j /= m-n)])
    (zip (enumFrom 1) p))
  where
  m = length p + 1
```

Figure 5.4: Queens with list comprehension desugared according to Haskell report

arguments. It seems desirable to find a method for removing this unnecessary abstraction. In this example it hardly matters, because `queensW` is only called 10 times, but in general unnecessary abstraction may decrease performance considerably. Note that the list of queen positions `p` is not removed by deforestation. It cannot be removed, because it is consumed twice, by `safe` and `appW`. If this were not the case, then the fact that `p` is λ-bound would still make list abstraction and thus fusion impossible. Similarly, `p` is consumed twice in the definition of `safe`, by `zipW` and `length`. The function `zip`, respectively `zipW`, consumes two lists. It is well known for not being suitable for short cut deforestation, see the discussion in [Gil96]. Hence it is not fused with the producer `enumFrom 1`. So in the definition of `safe` only the list consumed by `and` is removed by deforestation, besides the list introduced by desugaring of the list comprehension. Finally, some intermediate lists in the definition of `main` are removed.

In general we note that the deforested program is considerably less readable than the

```
main = print
  ((queensW (:) [] (appW (\y fys n -> fys $! (n+y))) id 10) 0)

queensW :: (Int -> c -> c) -> c -> (c -> d -> d) -> d -> Int -> d
queensW c1 n1 c2 n2 0 = c2 n1 n2
queensW c1 n1 c2 n2 m =
  queensW (:) []
    (\p ws -> enumFromToW (\l zs -> if safe p l
                                    then c2 (appW c1 p (c1 l n1)) zs
                                    else zs ) ws 1 10)
    n2 (m-1)

queens :: Int -> [[Int]]
queens = queensW (:) [] (:) []

safe :: [Int] -> Int -> Bool
safe p n =
  zipW (\(i,j) zs -> ((j /= n) && (i+j /= m+n) && (i-j /= m-n)) && zs)
    True (enumFrom 1) p
  where
  m = length p + 1

appW :: (a -> c -> c) -> [a] -> c -> c
appW c xs ys = foldr c ys xs

zipW :: ((a,b) -> c -> c) -> c -> [a] -> [b] -> c
zipW c n xs ys = go xs ys
  where
  go (x:xs) (y:ys) = c (x,y) (go xs ys)
  go _      _      = n

enumFromW :: (Int -> c -> c) -> Int -> c
enumFromW c l = go l
  where
  go l = l ‘c‘ (go $! (l+1))

enumFrom :: Int -> [Int]
enumFrom = enumFromW (:)

enumFromToW :: (Int -> c -> c) -> c -> Int -> Int -> c
enumFromToW c n l m =
  enumFromW (\x fxs -> if (<= m) x then x ‘c‘ fxs else n) l
```

Figure 5.5: Queens after type-inference based short cut deforestation

```
main = (print . sum . concat . queens) 10

queens :: Int -> [[Int]]
queens 0 = [[]]
queens m =
  foldr (\p ws ->
      foldr (\n zs -> if safe p n then (:) (p ++ [n]) zs else zs) ws
        (enumFromTo 1 10))
    []
    (queens (m-1))

safe :: [Int] -> Int -> Bool
safe p n = and
  (foldr (\(i,j) zs -> (:) ((j /= n) && (i+j /= m+n) && (i-j /= m-n)) zs)
    [] (zip (enumFrom 1) p))
  where
  m = length p + 1
```

Figure 5.6: Queens with list comprehension desugared according to Ghc

original program. Especially the working of `queensW` is far from obvious.

To obtain the program of Figure 5.5 from the program of Figure 5.4 we have to apply η-expansion three times, that is, replace a term M by `\`x `->` $M\ x$ for a variable x which does not occur in M. η-expansion is applied to the three anonymous functions `\p ->` ..., `\n ->` ... and `\(i,j) ->` ... which are introduced by the desugaring of list comprehensions. To avoid having to use η-expansion later, the desugaring translation itself should perform it. In fact, the Glasgow Haskell compiler uses a slightly different translation scheme for list comprehensions which translates the program of Figure 5.3 into the program given in Figure 5.6. Deforestation of this program does not require η-expansion and gives the program of Figure 5.5 as result as well.

We measured[2] the performance of the four versions of the queens program on a Linux PC with a 500 MHz Pentium III and 256 MB Ram with version 4.06 of the Glasgow Haskell compiler and maximal optimisations (`-O2`). This version of the Glasgow Haskell compiler has already a form of short cut deforestation built-in. Many standard list functions are defined in terms of `foldr` and `build` and the compiler applies the `foldr`/`build` fusion rule (see Section 7.1). To measure the performance of our programs without this built-in deforestation, we hide all the standard list functions[3], which are defined in terms of `foldr` and `build`, and give our own definitions instead, as listed in Appendix B. Thus the special function `build` does not occur in our programs. Hence the compiler cannot apply the `foldr`/`build` fusion rule. When compiling the original queens program with list comprehensions it is not possible to avoid the built-in deforestation transformation completely, because the compiler internally translates list comprehensions into `foldr`s and `build`s. So for this program we do not hide the definitions of the standard list functions

[2] by using the option `+RTS -s -RTS` that any program compiled by the Glasgow Haskell compiler understands

[3] by using `import Prelude hiding (sum, concat, map, ...)`

program version	megabytes allocated in the heap	runtime in seconds
list comprehensions desugared (Haskell report), Fig. 5.4	205.78	1.71
list comprehensions desugared (Ghc), Fig. 5.6	98.48	1.01
original with built-in short cut deforestation, Fig. 5.3	33.41	0.57
deforested by type-inference based deforestation, Fig. 5.5	22,43	0.51

Figure 5.7: Effect of deforestation on heap allocation and runtime

at all but measure the full effect of the built-in short cut deforestation of the Glasgow Haskell compiler.

We measured both the runtime and the number of bytes allocated on the heap. Note that this is the total number of bytes allocated during the program run. The runtime system automatically chooses a heap size of 1 or 2 megabytes and at any given point of time only a few kilobytes of these are live, that is not garbage. Figure 5.7 gives the results of our measurements. These show clearly the costs of intermediate lists and the improvement that deforestation can achieve. The program of Figure 5.6, where list comprehensions were desugared according to the method used by the Glasgow Haskell compiler, uses fewer intermediate lists than the program of Figure 5.4 and hence it is faster and allocates fewer bytes on the heap. Built-in short cut deforestation and our deforestation method give similar results, because the program only uses standard list functions, for which the Glasgow Haskell compiler has definitions in terms of `foldr` and `build`. The Glasgow Haskell compiler only cannot deforest user defined list producers. Our deforestation method still gives better results for the queens program. This is probably due to the worker/wrapper split of the definition of `queens` and the removal of the intermediate lists between recursive calls, a transformation that is not performed by the Glasgow Haskell compiler (cf. Section 7.1.4).

Figure 5.8 gives the results of some further measurements which we made to evaluate our worker/wrapper scheme. To obtain the deforested program of Figure 5.5 we applied the static argument transformation to the definitions of the workers `splitW` and `enumFromW` (cf. Section 5.1.2). Unfortunately, version 4.06 of the Glasgow Haskell compiler does not perform this transformation. To see the effect of this transformation we

program version	megabytes allocated in the heap	runtime in seconds
deforested without static argument transformation	68.53	0.79
only worker/wrapper split (Haskell report)	210.76	1.77
desugared with non-inlinable producers (Haskell report)	190.42	1.79
deforested with non-inlinable workers (Haskell report)	72.42	0.92

Figure 5.8: Heap allocation and runtime for different workers and wrappers

measured the performance of a deforested program which differs only from the program of Figure 5.5 in that the following definitions with static arguments are used:

```
zipW :: ((a,b) -> c -> c) -> c -> [a] -> [b] -> c
zipW c n (x:xs) (y:ys) = c (x,y) (zipW c n xs ys)
zipW c n _      _      = n

enumFromW :: (Int -> c -> c) -> Int -> c
enumFromW c l = l ‘c‘ (enumFromW c $! (l+1))
```

This version uses substantially more heap and also more time, thus substantiating our arguments in favour of the static argument transformation in Section 5.1.2.

We stated in Section 5.2.1 that worker definitions are less efficient than their corresponding original function definitions. So we measured the performance of the program that is obtained from applying only the worker/wrapper split algorithm but not the actual deforestation algorithm to the desugared program of Figure 5.4. This program is given in Appendix B. The program runs only slightly slower with slightly more heap allocation. So this example suggests that the inefficiency introduced by the worker/wrapper split might not be problematic.

Finally we investigate the effect of inlining. Because all function definitions are in a single module, the compiler can inline any definition. The idea of the worker/wrapper scheme is to enable deforestation across module boundaries without large-scale inlining; only the small wrappers need to be inlined. So we forbid the compiler to inline[4] the workers `concatW`, `mapW`, `concatMapW`, `appW`, `zipW`, `enumFromW`, `enumFromToW` and `takeWhileW`. Our example is artificial in that respect that the worker definitions are so small that the Glasgow Haskell compiler would normally inline them even across module boundaries. For a fair comparison we also measure the performance of a version of the desugared program of Figure 5.4, where the corresponding list-producing functions are non inlinable. Comparing the results of the desugared program with and without inlining we see that inlining increases space usage but decreases runtime slightly. For the deforested versions inlining makes a considerable difference. The version without inlining requires considerably more space and time than the version with inlining. Nonetheless the version without inlining still performs much better than the non-deforested version. So it is sensible to use workers and wrappers to enable deforestation where otherwise no deforestation would be possible. However, a non-inlined worker does not bring together subterms of producer and consumer to enable the application of further optimising transformations. The measurements show that this second effect of deforestation can be substantial.

[4] by using the NOINLINE pragma of the Glasgow Haskell compiler

Chapter 6

Extensions

In this chapter we present several ideas for improving and extending our deforestation algorithm. First, we discuss a worker/wrapper scheme for definitions of list-*consuming* functions. This scheme enables deforestation without requiring inlining of possibly large producer definitions. Subsequently, we generalise the short cut fusion rule so that it no longer requires that the list argument of a `foldr` contains the constructors which construct the list. Afterwards, we make several suggestions for improving the efficiency, that is, the runtime of our deforestation algorithm. Finally, we demonstrate that our type-inference based deforestation method is not limited to lists but can easily be extended to other recursive data types.

We are less formal in this chapter and hence use HASKELL in all examples for better readability.

6.1 A Worker/Wrapper Scheme for Consumers

To enable deforestation without requiring large-scale inlining we developed a scheme that splits a definition of a list producer into a definition of a worker and a definition of a wrapper. The wrapper contains all the list constructors that are needed for list abstraction. Dually, the consumer must be a `foldr` and hence sufficient inlining must be performed in the consumer to expose the `foldr`. If the consumer is a function with a large definition, then this is a problem. So a worker/wrapper scheme for list consumers is desirable, a scheme which splits the definition of a consumer into a small wrapper definition which contains the `foldr` and a large worker definition, which resembles the original definition.

We have two different worker/wrapper schemes for consumers. The first is the more obvious one, whereas the second fits better into our deforestation method. In both schemes the worker definitions are less efficient than the original definitions. Hence it has to be ensured that they are only used when they enable fusion, analogously to the worker/wrapper scheme for producers.

6.1.1 Scheme A

First, the arguments of the `foldr` of the consumer may be so large that it is desirable to separate them from the consumer. In that case we may define new functions for the arguments, so that the `foldr` of the consumer only has these small function calls as arguments. For a simple example consider the definition

```
map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\x fxs -> f x : fxs) [] xs
```

Suppose the term `(\x fxs -> f x :  fxs)` is too large for inlining. Then we define a new function `mapF` and redefine `map` as follows:

```
mapF :: (a -> b) -> a -> [b] -> [b]
mapF f = \x fxs -> f x : fxs

map :: (a -> b) -> [a] -> [b]
map f xs = foldr (mapF f) [] xs
```

Second, there may be a large context around the `foldr` in the definition of the consumer. Then we can define a new function to abstract this context. For example, consider the definition

```
foo :: Bool -> (a -> b) -> [a] -> [b]
foo b f xs = if b then foldr (mapF f) [] xs else []
```

Suppose the context of `foldr` is too large for inlining. Then we split this definition as follows:

```
cont :: Bool -> [b] -> [b]
cont b y = if b then y else []

foo :: Bool -> (a -> b) -> [a] -> [b]
foo b f xs = cont b (foldr (mapF f) [] xs)
```

The two transformations do not require any type-based analysis but can be performed directly on the syntactic structure.

6.1.2 Scheme B

The basic idea of an alternative worker/wrapper scheme for consumers is to transform a function that consumes a list with `foldr` so that it consumes a polymorphic list producer instead. We illustrate this again at the hand of the definition of the function `map`.

```
map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\x fxs -> f x : fxs) [] xs
```

It is split into the definition of a worker

```
mapW :: (a -> b) -> (forall c. (a -> c -> c) -> c -> c) -> [b]
mapW f g = g (\x fxs -> f x : fxs) []
```

and the definition of a wrapper

```
map :: (a -> b) -> [a] -> [b]
map f xs = mapW f (\$v^{(:)}$ $v^{[]}$ -> foldr $v^{(:)}$ $v^{[]}$ xs)
```

We can easily fuse this consumer wrapper with a list abstracted producer:

map (+1) (enumFrom 1)

$\leadsto$ {inlining and β-reduction of wrappers}

mapW (+1) (\v$^{(:)}$ v$^{[]}$ -> foldr v$^{(:)}$ v$^{[]}$ (enumFromW (:) 1))

$\leadsto$ {list abstraction and short cut fusion}

mapW (+1) (\v$^{(:)}$ v$^{[]}$ -> enumFromW v$^{(:)}$ 1))

It is still a problem that inlining and β-reduction must replace the variable `xs` in the definition of the wrapper `map` by the producer `enumFromW (:) 1`. A standard inliner will refuse to do so, because `xs` is within the scope of a λ-abstraction (\v$^{(:)}$ v$^{[]}$ -> ...) and hence in general inlining the producer could duplicate computation (cf. Section 2.9.2). The inliner would need to know that the λ-abstraction is only used once.

The worker/wrapper split could be performed by a modified version of our type-inference based list abstraction algorithm. Instead of abstracting list constructors it would abstract occurrences of `foldr`. This is also advantageous, because for functions that are both consumers and producers the worker/wrapper scheme for consumers and producers have to be combined.

6.2 Deforestation Beyond Arguments of `foldr`

Our deforestation algorithm searches for terms of the form `foldr` $M^{(:)}$ $M^{[]}$ $M^{\mathtt{P}}$, applies list abstraction to $M^{\mathtt{P}}$ and, if successful, applies the short cut fusion rule. List abstraction cannot succeed, if $M^{\mathtt{P}}$ is a variable or the program only contains the partial application `foldr` $M^{(:)}$ $M^{[]}$. This restriction is sometimes unfortunate, as the following examples demonstrate.

Our deforestation algorithm successfully deforests the term `or (fst unzip xs)`. However, the compiler may inline the small definition of `fst` and simplify the term to obtain `case unzip xs of (vs,ws) -> or vs`. Here the argument of `or` is just a variable and hence `or` cannot be fused with its argument.

A monadic function may produce a list that is consumed by `foldr`, but the monadic producer can never be the argument of a `foldr`. Consider the term

```
getLine >>= putStr
```

which reads a list of characters from standard input and subsequently writes it to standard output. The functions `putStr` and `getLine` can be defined as follows:

```
putStr :: [Char] -> IO ()
putStr = foldr (\c r -> putChar c >> r) (return ())

getLine :: IO [Char]
getLine  =  do c <- getChar
               if c == '\n' then return [] else
                 do s <- getLine
                    return (c:s)
```

The definition of the producer `getLine` can be split into a worker and a wrapper:

getLineW :: (Char -> c -> c) -> c -> IO c

getLineW v$^{(:)}$ v$^{[]}$ = do c <- getChar

```
               if c == '\n' then return v^[] else
                 do s <- getLine
                   return (v^(:) c s)

getLine :: IO [Char]
getLine = getLineW (:) []
```

Nonetheless the term `getLine >>= putStr` cannot be deforested, because `getLine` is not an argument of `putStr`.

The solution is to abstract the intermediate list and `foldr` with its first two arguments from a term which we want to deforest. We use the following, generalised short cut fusion rule[1]:

$$\frac{M\ \texttt{::}\ \ (A\ \texttt{->}\ c\ \texttt{->}\ c)\ \texttt{->}\ c\ \texttt{->}\ (c\ \texttt{->}\ B)\ \texttt{->}\ C \qquad c \notin \mathrm{free}(C)}{M\ \texttt{(:) [] (foldr}\ M^{(:)}\ M^{[]}\texttt{)} \quad \leadsto \quad M\ M^{(:)}\ M^{[]}\ \texttt{id}}$$

Here A, B and C are types and c a type variable, so that the term M is polymorphic. The fusion rule is the parametricity theorem of the polymorphic type.

With this generalised fusion rule we can deforest our example terms as follows:

```
    case unzip xs of (vs,ws) -> or vs
 ⇝   {inlining}
    case unzipW (:) [] (:) [] xs of (vs,ws) -> foldr (||) False vs
 ⇝   {type-inference based abstraction}
    (\v^(:) v^[] f -> case unzipW v^(:) v^[] (:) [] xs of (vs,ws) -> f vs)
      (:) [] (foldr (||) False)
 ⇝   {generalised fusion rule and β-reduction}
    case unzipW (||) False (:) [] xs of (vs,ws) -> vs
```

Similarly:

```
    getLine >>= putStr
 ⇝   {inlining}
    getLineW (:) [] >>= foldr (\c r -> putChar c >> r) (return ())
 ⇝   {type-inference based abstraction}
    (\v^(:) v^[] f -> getLineW v^(:) v^[] >>= f)
      (:) [] (foldr (\c r -> putChar c >> r) (return ()))
 ⇝   {generalised fusion rule and β-reduction}
    (getLineW (\c r -> putChar c >> r) (return ()) >>= id)
```

In the latter example `getLineW` constructs an I/O action instead of a list of characters.

A limitation for the generalised short cut fusion rule is that we can only abstract `foldr` $M^{(:)}$ $M^{[]}$ from a term M, if no free variable of $M^{(:)}$ and $M^{[]}$ is bound in M.

6.3 More Efficient Algorithms

First, the efficiency of the type inference algorithm can be improved. In fact, in [Chi99] we used a different type inference algorithm based on the algorithm $\mathcal{M}$ [LY98] for the

[1] Here, where we use Haskell instead of F, we informally ignore free variables and their types.

Hindley-Milner type system. This type inference algorithm permits the optimisation that type inference notices early if the desired polymorphic type cannot be inferred and then may abort the whole type inference process. Unfortunately, this optimisation cannot be used for the worker/wrapper split algorithm. We did not want to have two different type inference algorithms, one for abstracting from a single list and that may fail early, and one for all other occasions. It may be possible to integrate the two algorithms to have the best of both.

More importantly, the worker/wrapper split algorithm is not as efficient as it could be. The list abstraction algorithm traverses a whole definition body once. Even if we ignore polymorphic recursion, if n `let` definitions are nested, then the body of the inner definition is traversed n times, leading to quadratic complexity. However, as stated in Section 4.3.5, the list abstraction algorithm uses a modified version of the Hindley-Milner type inference algorithm. The abstraction of list types corresponds to the generalisation step of the Hindley-Milner algorithm. The list abstraction algorithm just additionally abstracts list constructors and inserts both type and term abstractions into the program. The Hindley-Milner algorithm recursively traverses a program only once. So it should be possible to integrate explicit type and term abstraction at `let` definitions into this type inference algorithm to obtain a single pass worker/wrapper split algorithm. Note that this further step towards the Hindley-Milner algorithm would not lead to the same worst case exponential time complexity. The inferred type annotations can only be smaller than the type annotations of the original list-producing term. Hence the resulting algorithm would only require linear time. To deal with polymorphic recursion as well, the type inference algorithm of Emms and Leiß, which integrates semiunification into the Hindley-Milner algorithm, may provide a good basis [EL99].

6.4 Other Data Types than Lists

Our type-inference based deforestation method is not specific to lists but can be used for nearly arbitrary recursive data types. For fusion we basically only need a kind of `foldr` for the new data type, a so called **catamorphism**. In Section 7.2 we give a formal definition of a catamorphism. The intuitive idea is that it uniformly replaces every data constructor of the data type by a given term. Consider for example the type of arbitrarily branching trees:

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

Its catamorphism `foldRT` is defined as follows:

$$\begin{array}{l}\texttt{foldRT :: (a -> [b] -> b) -> RoseTree a -> b}\\ \texttt{foldRT v}^{\texttt{Node}}\texttt{ (Node x ts) = v}^{\texttt{Node}}\texttt{ x (map (foldRT v}^{\texttt{Node}}\texttt{) ts)}\end{array}$$

The programmer could explicitly use this catamorphism and inform the compiler about it through a specific directive. Alternatively, an algorithm like warm fusion (cf. Section 7.1.3) could transform a generally recursive definition into catamorphism form automatically.

The short cut fusion rule for the data type `RoseTree a` is:

$$\frac{M^{\mathsf{P}}\texttt{:: (}A\texttt{ -> [}c\texttt{] -> }c\texttt{) -> }c}{\texttt{foldRT }M^{\texttt{Node}}\texttt{ (}M^{\mathsf{P}}\texttt{ Node)} \quad\rightsquigarrow\quad M^{\mathsf{P}}\ M^{\texttt{Node}}}$$

The rule is an instance of the parametricity theorem of the type $\texttt{(}A\texttt{ -> [}c\texttt{] -> }c\texttt{) -> }c$. As an aside note that $\forall\gamma.\texttt{(}A \rightarrow \texttt{[}\gamma\texttt{]} \rightarrow \gamma\texttt{)} \rightarrow \gamma$ is the canonical encoding of the data type `RoseTree` A in the second-order λ-calculus, just as $\forall\gamma.\texttt{(}A \rightarrow \gamma \rightarrow \gamma\texttt{)} \rightarrow \gamma \rightarrow \gamma$ is the canonical encoding of the data type `[`A`]` and the function `build` A, which is used in the original short cut deforestation method, is the isomorphism from the former type to the latter (cf. [Pit00]).

It is straightforward to transform our list abstraction algorithm into a rose tree abstraction algorithm. Similarly our worker/wrapper scheme can be adapted to rose trees. In fact, a single worker can abstract from data constructors and types for several different data types simultaneously. We can even handle mutually recursive data types. The catamorphisms of a set of mutually recursive data types are mutually recursive.

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, FS96]. Nonetheless these limitations are mild, because nearly all types which appear in practice are regular and covariant. Note however that the validity of parametricity theorems has only been proved for languages with only regular, covariant algebraic data types [Pit00].

Chapter 7

Other Deforestation Methods

An enormous number of deforestation methods have been developed. Here we give an overview, concentrating on the methods that have received most attention. We relate them to our deforestation method and note that some of them could also be improved by a type-inference based analysis. We start with short cut deforestation.

7.1 Short Cut Deforestation

The fundamental idea of short cut deforestation is that the uniform replacement of the constructors `(:)` and `[]` by a `foldr` consumer can already be performed at compile time. Our own deforestation method performs short cut deforestation. However, whereas we transform the producer into the required list abstracted form by a type-inference based algorithm, the original short cut deforestation method [GLP93, GP94, Gil96] requires that the producer is already in list abstracted from. A special function `build` is used both to easily recognise the producer and to enforce the type requirement of the short cut fusion rule:

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

With `build` the short cut fusion rule can be written as follows:

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

The compiler writer defines all list-manipulating functions in the standard libraries in terms of `foldr` and `build`.

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

7.1.1 Gill's Worker/Wrapper Scheme

In the last chapter of his thesis [Gil96] Gill also proposes a worker/wrapper scheme. It enables short cut fusion with user-defined producers without requiring large scale inlining.

The prerequisite is that these producers are defined in terms of standard list functions or producers which themselves are directly or indirectly defined only in terms of standard list functions. Our own worker/wrapper scheme is based on Gill's scheme. Whereas we move all data constructors which construct a result list to the wrapper, so that they are visible at the deforestation site, Gill has to make `build` visible at the deforestation site. Note that for the `foldr`/`build` fusion rule it is only necessary that the producer is in `build` form, the argument of `build` is of no interest but is just rearranged by the transformation.

So, for example, the definition of `map` in `build` form

```
map f xs = build (\v$^{(:)}$ v$^{[]}$ -> foldr (v$^{(:)}$ . f) v$^{[]}$ xs)
```

is split up as follows:

```
mapW :: (a -> b) -> [a] -> (b -> c -> c) -> c -> c
mapW f xs v$^{(:)}$ v$^{[]}$ = foldr (v$^{(:)}$ . f) v$^{[]}$ xs

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

When `build` is inlined in these definitions, the similarity to our worker/wrapper scheme becomes obvious. Gill does not suggest to apply the worker/wrapper split to the definitions of the standard list functions such as `map`, which are always inlined, but to user-defined list-producing functions. Gill also demonstrates how the worker/wrapper scheme enables removal of lists that exist between recursive calls of a single function (cf. Section 3.4)

7.1.2 Limitations of `build`

Besides the restriction of the original short cut deforestation method to producers which are defined in terms of standard list-producing functions the use of `build` has also some other disadvantages.

In Section 3.3.2 we showed how our method can split the definition of the list append function `(++)` into a worker and a wrapper and fuse a consumers with a producer that uses `(++)`. However, because a term M_1 `++` M_2 does not produce the whole resulting list itself — only the list produced by M_1 is copied but not the one produced by M_2 — `(++)` cannot be expressed in terms of `build`. Modifying the definition of `(++)` to copy as well the list which is produced by M_2, as done in [GLP93] and also [LS95, TM95], runs contrary to our aim of eliminating data structures. Hence Gill defines a further second-order typed function `augment` to solve this problem [Gil96]:

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

Similar to the `foldr`/`augment` fusion rule there exists a `foldr`/`augment` fusion rule:

```
foldr $M^{(:)}$ $M^{[]}$ (augment $P$ $M$)   $\rightsquigarrow$   $P$ $M^{(:)}$ (foldr $M^{(:)}$ $M^{[]}$ $M$)
```

The function `(++)` can be defined in terms of `augment`

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = augment (\v$^{(:)}$ v$^{[]}$ -> foldr v$^{(:)}$ v$^{[]}$ xs) ys
```

and hence be fused with a `foldr` consumer. The function `build` can be defined in terms of `augment`

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

and the `foldr/build` fusion rule is just an instance of the `foldr/augment` fusion rule. Probably all realistic list producers can be defined in terms of `augment`.

The function `build` (and similarly `augment`) can only wrap a producer that returns a single list. Hence, for example, the function `unzip` cannot be expressed in terms of `build`. Therefore, the original short cut deforestation algorithm cannot split its definition into a worker and a wrapper and fuse the producer `fst (unzip Bool Char zs)` with a consumer as we did in Section 3.3.1. Similarly, it is not possible to abstract nested result lists with `build` as we did in Section 3.4.1 for the function `inits` with our worker/wrapper scheme. Not even the definition (cf. Section 3.3.2)

$$\begin{aligned}\texttt{concat} &: \forall\alpha.\texttt{[[}\alpha\texttt{]]} \rightarrow \texttt{[}\alpha\texttt{]}\\ &= \lambda\alpha.\,\texttt{foldr [}\alpha\texttt{] [}\alpha\texttt{] (appW } \alpha \texttt{ [}\alpha\texttt{] ((:) } \alpha\texttt{)) ([] } \alpha\texttt{)}\end{aligned}$$

can directly be expressed in terms of `build`. It first needs to be transformed into

$$\begin{aligned}\texttt{concat} &: \forall\alpha.\texttt{[[}\alpha\texttt{]]} \rightarrow \texttt{[}\alpha\texttt{]}\\ &= \lambda\alpha.\lambda\texttt{xs}.\,\texttt{foldr [}\alpha\texttt{] [}\alpha\texttt{] (appW } \alpha \texttt{ [}\alpha\texttt{] ((:) } \alpha\texttt{)) ([] } \alpha\texttt{) xs}\end{aligned}$$

Finally, because of `build`, in Gill's scheme a wrapper cannot be inlined when it is only partially applied. In contrast we inlined in Section 3.3.2 the partially applied wrapper `(++)` in the definition of `concat` to derive its worker definition.

Altogether we see that list abstraction provides the means for simpler and more flexible deforestation. It is not possible to usefully combine `build` and type-inference based list abstraction. For example, from the producer `build (mapW f xs)` no list constructors can be abstracted, because they are hidden by `build`. We would need to inline `build` to proceed with list abstraction.

7.1.3 Warm Fusion

Warm fusion [LS95] transforms a generally recursive definition into `build/foldr` form, that is, it automatically derives a definition in terms of both `build` and `foldr`, if the function is both a list producer and a list consumer. Its basic idea for transforming an arbitrary list producer M into `build` form is to rewrite it as

$$\texttt{build (\textbackslash c n -> foldr c n } M\texttt{)}$$

and push the `foldr` into the M, hoping that it cancels with `build`s in there. If it does, then the `build` form is used for fusion, otherwise the transformation is abandoned. Similarly, to transform the definition of a list-consuming function

$$\texttt{foo xs = } M[\texttt{xs}]$$

into `foldr` form, the transformation rewrites it as

$$\texttt{foo xs = } M[\texttt{foldr (:) [] xs}]$$

and then tries to fuse the function body M with the `foldr`. All this pushing and fusing of `foldr` is performed by term rewriting, where in the case of transforming a consumer into `foldr` form the rule set grows and shrinks dynamically. This later transformation is based on the fusion theorem of catamorphisms [MFP91]. The approach is not limited to lists, but works for a large class of data structures (cf. Section 7.2 and Section 6.4). However, warm fusion is rather expensive and poses substantial problems for an implementation [NP98].

7.1.4 Short Cut Fusion in the Glasgow Haskell compiler

The current version 4.06 of the Glasgow Haskell compiler [GHC] basically performs Gill's original short cut deforestation without any worker/wrapper scheme. The major problem with Gill's implementation of short cut deforestation was that the `foldr`/`build` form is good for enabling short cut fusion, but itself very inefficient[1]. If a `foldr` or a `build` is not used for fusion, then it must be inlined. By an elaborate set of rewriting rules and a two phase inlining system the current implementation ensures, that, if a `foldr`/`build` form of a function is not used for fusion, then it is replaced by a more efficient direct definition of the function.

7.2 Hylomorphism Fusion

Hylomorphisms enable the description of "regular" producers besides "regular" consumers such as `foldr` in a single form. In Section 6.4 we already demonstrated how the recursion pattern of `foldr` can be generalised to other data types. Unfortunately, fusion of the function `foldRT` with a producer of type `RoseTree a` requires a new fusion rule. Already the type of `foldRT` is quite different from the type of `foldr`. It is possible to give a single definition for `foldr`, `foldRT` and respective functions for other data types, if we use a uniform way to define recursive data types.

Category theory is required for a formal definition of hylomorphisms fusion. Here we completely avoid category theory. Instead we just illustrate the idea of hylomorphisms fusion at the hand of Haskell programs, similarly to [Jon95, MH95].

We only have a single recursive data type which constructs the fixpoint of a unary type constructor:

```
newtype Rec f = In (f (Rec f))
```

The constructor `In` converts a term of type `f (Rec f)` into a term of type `Rec f`. We define the inverse function `out` as follows:

```
out :: Rec f -> f (Rec f)
out (In x) = x
```

To define a recursive data type we first define a non-recursive data type with an additional argument:[2]

```
data NatF r = Zero | Succ r
data ListF a r = Nil | Cons a r
data RoseTreeF a r = Node a [r]
```

From a parameterised non-recursive data type we easily obtain the desired recursive data type:

```
type Nat = Rec NatF
type List a = Rec (List a)
type RoseTree a = Rec (RoseTree a)
```

[1] Personal communication with Simon Peyton Jones

[2] In the definition of `RoseTreeF` we use the standard Haskell list type `[r]` to keep the presentation simple. Naturally we could use the type `List r`, which we are about to define, instead.

The elements of the data types are slightly unusual. For example, `Nat` has the elements `In Zero`, `In (Succ (In Zero))`, `In (Succ (In (Succ (In Zero))))`,.... Similarly, `List Bool` has for example the elements `In Nil` and `In (Cons True (In Nil))`.

After we have defined recursive data types in a uniform way we now come to the definition of recursive functions over these data types. First we note that the non-recursive data types have at least one argument, the argument that is used to introduce recursion. So there is a natural definition of the function `fmap` for these data types:

```
instance Functor NatF where
  fmap f Zero     = Zero
  fmap f (Succ r) = Succ (f r)

instance Functor (ListF a) where
  fmap f Nil    = Nil
  fmap f (Cons x r) = Cons x (f r)

instance Functor (RoseTree a) where
  fmap f (Node x rs) = Node x (map f rs)
```

The function `foldr` uniformly replaces the list constructors `(:)` and `[]` by two given terms. Its generalisation, the **catamorphism**, replaces the constructor `In` by a function `phi` of the same type:

```
cata :: Functor f => (f a -> a) -> Rec f -> a
cata phi = phi . fmap (cata phi) . out
```

It works by removing the constructor `In` by `out`, applying itself recursively by `fmap` and finally applying `phi`. For example:

```
and :: List Bool -> Bool
and = cata (\fa -> case fa of
                     Nil -> True
                     Cons x r -> x && r)

map :: (a -> b) -> (List a -> List b)
map f = cata (\fa -> case fa of
                       Nil -> Nil
                       Cons x r -> Cons (f x) r)
```

To be precise, not the function `cata` is a catamorphism, but any function that can be defined in the form `cata` ϕ for some term ϕ of functional type. For catamorphisms a generalisation of the short cut fusion rule holds, the so-called acid rain rule:

$$\frac{M^{\mathsf{P}}\texttt{::}\ \texttt{(}F\ c\ \texttt{->}\ c\texttt{)}\ \texttt{->}\ B\ \texttt{->}\ c}{\texttt{(cata}\ \phi\texttt{)}\ \texttt{.}\ \texttt{(}M^{\mathsf{P}}\ \texttt{In)}\quad\leadsto\quad M^{\mathsf{P}}\ \phi}$$

The acid rain rule states that instead of first producing a value of type `Rec` F and subsequently replacing all data constructors `In` by ϕ, the producer skeleton M^{P} can directly construct the result with ϕ. Note that the producer skeleton M^{P} must be polymorphic in the type variable c. ϕ is a term of type F A `->` A, A and B are types and F is a type constructor for which an instance of the class `Functor` exists.

Because the constructor `In` and the function `out` are inverses of each other, the definition of a dual to a catamorphism, called **anamorphism**, suggests itself:

```
ana :: Functor f => (a -> f a) -> a -> Rec f
ana psi = In . fmap (ana psi) . psi
```

Whereas a catamorphism destructs a value of a recursive data type in a uniform way, an anamorphism constructs a value of a recursive data type in a uniform way. For example:

```
enumFrom :: Int -> List Int
enumFrom = ana (\n -> Cons n (n+1))

map :: (a -> b) -> (List a -> List b)
map f = ana (\la -> case la of
                      Nil -> Nil
                      Cons x r -> Cons (f x) r)
```

The function `ana` is a generalisation of the rarely used Haskell list function `unfoldr`:

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of
                Nothing    -> []
                Just (a,b) -> a : unfoldr f b
```

Also for catamorphisms a dual acid rain rule exists:

$$\frac{M^{\mathsf{C}}\texttt{::}\ (c\ \texttt{->}\ F\ c)\ \texttt{->}\ c\ \texttt{->}\ B}{(M^{\mathsf{C}}\ \texttt{out})\ \texttt{.}\ (\texttt{ana}\ \psi)\quad\rightsquigarrow\quad M^{\mathsf{C}}\ \psi}$$

Instead of consuming a value of type `Rec` F that is constructed uniformly with the function ψ `::` A `->` F A, the consumer skeleton M^{C} can directly consume the value of type A with ψ.

So for deforestation function definitions should be transformed into catamorphism or anamorphism form. However, many functions, such as `map`, can be defined as either catamorphisms or anamorphisms. Hence the **hylomorphism** is introduced, which is simply the composition of a catamorphism with an anamorphism:

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> (a -> b)
hylo phi psi = cata phi . ana psi
```

The composition of catamorphism and anamorphism can be fused by either of the two acid rain rules, giving the following direct definition of hylomorphisms:

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> (a -> b)
hylo phi psi = phi . fmap (hylo phi psi) . psi
```

From this definition it becomes obvious that catamorphisms and anamorphisms are just special cases of hylomorphisms:

```
cata phi = hylo phi out
ana  psi = hylo In  psi
```

The problem that a function such as `map` can be defined as hylomorphism in at least two different ways persists. The solution is the **hylomorphism in triplet form**. A polymorphic function that could be part of either the ϕ of the catamorphism or the ψ of the anamorphism is separated out as third argument:

```
hyloT :: (Functor f, Functor g) =>
  (g b -> b) -> (forall c. f c -> g c) -> (a -> f a) -> (a -> b)
hyloT phi eta psi = phi . eta . fmap (hylo phi psi) . psi
```

or equivalently

```
hyloT phi eta psi = phi . fmap (hylo phi psi) . eta . psi
```

Note that these two definitions are only equivalent because `eta` is polymorphic. The representation as hylomorphism in triplet form is more flexible, because it permits `phi` and `psi` to have types with different type constructors `g` and `f`. Naturally, the introduction of `eta` increases the number of possibilities for representing a function as hylomorphism. However, the idea is that "as much as possible" is shifted into this middle argument. So the canonical hylomorphism representation of `map` is the following:

```
map :: (a -> b) -> (List a -> List b)
map f = hylo In eta out
  where
  eta = \la -> case la of
                 Nil -> Nil
                 Cons x r -> Cons (f x) r
```

The acid rain rules can be formulated directly for hylomorphisms:

$$\frac{M^{\mathrm{P}}\texttt{::}\ (F\ c\ \texttt{->}\ c)\ \texttt{->}\ B\ \texttt{->}\ c}{\texttt{(hylo}\ \phi\ \eta\ \texttt{out)}\ \texttt{.}\ \texttt{(}M^{\mathrm{P}}\ \texttt{In)}\quad\leadsto\quad M^{\mathrm{P}}\ \texttt{(}\phi\ \texttt{.}\ \eta\texttt{)}}$$

$$\frac{M^{\mathrm{C}}\texttt{::}\ (c\ \texttt{->}\ F\ c)\ \texttt{->}\ c\ \texttt{->}\ B}{\texttt{(}M^{\mathrm{C}}\ \texttt{out)}\ \texttt{.}\ \texttt{(hylo In}\ \eta\ \psi\texttt{)}\quad\leadsto\quad M^{\mathrm{C}}\ \texttt{(}\eta\ \texttt{.}\ \psi\texttt{)}} \tag{*}$$

So the previously given definition for `map` is suitable for both fusion with a producer and a consumer.

Hylomorphism fusion methods assume that all functions that may be fused are in hylomorphism form. Hence the acid rain rules are specialised as follows:

$$\frac{\tau\ \texttt{::}\ (F\ c\ \texttt{->}\ c)\ \texttt{->}\ G\ c\ \texttt{->}\ c}{\texttt{(hylo}\ \phi\ \eta_1\ \texttt{out)}\ \texttt{.}\ \texttt{(hylo (}\tau\ \texttt{In)}\ \eta_2\ \psi\texttt{)}\quad\leadsto\quad\texttt{hylo (}\tau\ \texttt{(}\phi\ \texttt{.}\ \eta_1\texttt{))}\ \eta_2\ \psi}$$

$$\frac{\sigma\ \texttt{::}\ (c\ \texttt{->}\ F\ c)\ \texttt{->}\ c\ \texttt{->}\ G\ c}{\texttt{(hylo}\ \phi\ \eta_1\ \texttt{(}\sigma\ \texttt{out))}\ \texttt{.}\ \texttt{(hylo In}\ \eta_2\ \psi\texttt{)}\quad\leadsto\quad\texttt{hylo}\ \phi\ \eta_1\ \texttt{(}\sigma\ \texttt{(}\eta_2\ \texttt{.}\ \psi\texttt{))}}$$

Hylomorphisms were first introduced in [MFP91], where recursion schemes and algebraic laws that are well-known for lists are generalised to a large class of recursive data types. Subsequently [TM95] developed the hylomorphism fusion method. [Jür99, Jür00] studies the category theoretical foundations of hylomorphism fusion. A programmer cannot be expected to program in terms of hylomorphisms. Hence [HIT96] presents algorithms for transforming generally recursive definitions into hylomorphisms and restructuring them so that the acid rain rules can be applied. The authors implemented their algorithms in an experimental system called HYLO [OHIT97]. They extended hylomorphisms to handle functions such as `zip` which induct over multiple data structures [HIT97, IHT98]. The authors see hylomorphisms as a suitable representation for performing various other transformations [THT98], for example for automatic parallelisation

[HTC98]. The HYLO algorithms were implemented by Tüffers in the Glasgow Haskell compiler and found to be able to transform most definitions of real world programs into hylomorphisms [Tüf98]. Recently the HYLO algorithms were implemented in the pH compiler [Sch00], a compiler for a parallel variant of HASKELL [NAH+95].

Because hylomorphisms do not occur in normal programs and already data types are not defined in terms of `Rec`, hylomorphism fusion methods require complex program transformations. These transformations may degrade the performance of a program. So we note that the algorithm for deriving a hylomorphism implicitly performs some common subexpression elimination. For example, in Section 4.3 of [OHIT97] the definition of a function `foo` with two identical recursive calls `foo as` is transformed into a hylomorphism in which the computation of the recursive calls is shared. Common subexpression elimination avoids repeated computation but may unboundedly increase space usage and thus indirectly also garbage collection time [Chi98].

Our idea of using type inference to identify data constructors in a producer could also be applied to hylomorphism fusion. A type inference algorithm similar to ours could be used for deriving the polymorphic functions τ and σ that are needed for the acid rain rules. This algorithm would be more general than the current syntax based one and permit fusion of hylomorphisms with terms that are not in hylomorphism form, using the more general version ($*$) of the acid rain rules.

7.3 Unfold-Fold Transformations

A completely different family of deforestation methods is based on Burstall and Darlington's unfold/fold transformation system [BD77]. These methods are not concerned with any specific recursion schemes. Instead they transform a term by a number of rewriting steps, most of which resemble the usual evaluation rules of the operational semantics. Central steps are the unfolding of recursive definitions and the creation of new recursive definitions, called folding.

We illustrate the idea of these methods at the hand of an example. Let a program contain the following recursive definitions:

```
or :: [Bool] -> Bool
or = \us -> case us of
              []   -> True
              v:vs -> v || or vs

map :: (a -> b) -> [a] -> [b]
map = \f ws -> case ws of
                 []   -> []
                 y:ys -> f x : map f ys
```

Note that these definitions do not use any higher-order function such as `foldr` that encapsulates a recursion scheme. The term `or (map p xs)` constructs an intermediate list at runtime. The term is deforested by the steps given in Figure 7.1.

The most difficult step is obviously the folding step at the end. If unfolding and evaluation continued with the subterm `or (map p ys)`, then the deforestation transformation would never terminate. The transformation system has to recognise that `or (map p ys)` is, except for the argument `ys`, the same as the term it started with. Hence it can create a new recursive definition and thus terminate the deforestation process.

```
   or (map p xs)
⇝   {unfolding of definition of or}
   (\us -> case us of {[] -> True; v:vs -> v || or vs})
     (map p xs)
⇝   {evaluation}
   case (map p xs) of
     [] -> True
     v:vs -> v || or vs
⇝   {unfolding of definition of map}
   case (\f ws -> case ws of {[] -> []; y:ys -> f y : map f ys}) p xs of
     [] -> True
     v:vs -> v || or vs
⇝   {evaluation}
   case (case xs of {[] -> []; y:ys -> p y : map p ys}) of
     [] -> True
     v:vs -> v || or vs
⇝   {case of case transformation}
   case xs of
     [] -> case [] of {[] -> True; v:vs -> v || or vs}
     y:ys -> case (p y : map p ys) of {[] -> True; v:vs -> v || or vs}
⇝   {evaluation}
   case xs of
     [] -> True
     y:ys -> p y || or (map p ys)
⇝   {folding with a new definition g}
   g p xs
   where
   g = \p xs -> case xs of
                  [] -> True
                  y:ys -> p y || g p ys)
```

Figure 7.1: Steps of unfold/fold deforestation of `or (map p xs)`

Short cut deforestation and hylomorphism fusion are based on fusion of a data structure consumer with a data structure producer. In contrast, unfold/fold based deforestation methods always transform a whole term; consumers and producers are not identified.

Unfold/fold based deforestation was introduced by Wadler [Wad88, Wad90], who also coined the term deforestation, and is based on his earlier work on listlessness [Wad84, Wad85]. Although he suggests extensions, he defines the system only for a subset of a first-order functional language. Subsequently the system was extended first to full first-order languages [HJ92, CK95] and then to higher-order languages [MW92, Mar95, Chi92, Chi94, Ham96, SS98].

The transformation steps of unfold/fold deforestation perform call-by-name evaluation and hence may duplicate terms that have to be evaluated. We already discussed this issue in Section 5.2.4 to show that this is not a problem for short cut deforestation. The main problem of unfold/fold deforestation is to ensure that the transformation terminates on any input program. This has become the main topic of research on deforestation [FW88, Sør94, Sei96, SS97, SS98]. [Ham96] relates several deforestation techniques. [SGJ94] compares deforestation with other partial evaluation transformations. [San96a] gives the first proof of semantic correctness of unfold/fold based deforestation. It is also a good introduction to this deforestation method. It has not yet been formally proved that unfold/fold deforestation always improves the performance of a program, but because of the similarity of most transformation steps to well-known evaluation rules a proof seems feasible.

A prototype of Wadler's original system was implemented in [Dav87]. Later, Marlow extended the Glasgow Haskell compiler by higher-order deforestation. The implementation was, however, not generally useable, because it could not control inlining to avoid code explosion [Mar95]. The only implementation of unfold/fold deforestation in a real compiler that we are aware of is for the strict functional-logic languages Mercury [Tay98]. However, it does not perform any deforestation across module boundaries and its effect on real programs has not yet been investigated.

7.4 Attribute Grammar Composition

Attribute grammars extend context-free grammars by a set of attributes for every symbol and a set of rules for determining theses attributes for every grammar production. They are mostly used in the design and implementation of compilers [ASU86]. Because of their similarity to functional languages they are sometimes considered as a functional programming paradigm [Joh87]. Similar to functional languages the question arises if two attribute grammars can be transformed into a single attribute grammar that computes the same function as the composition of the two original grammars. Such composition techniques for certain classes of attribute grammars are presented in [Gan83, Gie88]. [CDPR97] compares these composition techniques with various deforestation techniques for functional programs and [CDPR99] discusses how functional programs can be translated into attribute grammars, these be composed, and finally the result be retranslated into a functional program.

There exist various modifications and extensions of the attribute grammar formalism. **Attributed tree transducers** abstract from the language production aspect of a context-free grammar and just describe functions that map trees to trees instead. **Macro tree transducers** extend the limited recursion scheme of attribute grammars and **macro attributed tree transducers** [KV94] are an integration of attributed tree transducers and macro tree transducers. The set of functions definable by any of these formalisms is not closed under composition but with some additional restrictions various composition techniques have been developed [Rou70, Fül81, EV85, FV98]. [Küh98] transfers the composition of attributed tree transducers to functional programs. [Küh99] compares Wadler's unfold/fold deforestation technique with composition techniques for macro tree transducers and [Höf99] extends this comparison to nonlinear programs.

The various composition methods rely on the restricted recursion schemes of the formalisms. It remains to be seen if the methods can be generalised to arbitrary functional programs or suggest improvements to the standard methods.

7.5 Further Methods

The short cut fusion rule and its generalisations to other data types, the acid rain rules, are parametricity theorems of a polymorphic producer (or consumer in case of the anamorphism fusion rule). However, `foldr` and `cata` are polymorphic themselves. The catamorphism fusion rule is the parametricity theorem of catamorphisms [MFP91, MH95]:

$$\frac{h \text{ strict} \quad h \,\texttt{.}\, \phi \;\texttt{=}\; \phi' \,\texttt{.}\, \texttt{(fmap } h\texttt{)}}{h \,\texttt{.}\, \texttt{(cata } \phi\texttt{)} \quad \rightsquigarrow \quad \texttt{cata } \phi'}$$

The application of this rule to deforestation is studied in [SF93, FSZ94] and also warm fusion uses it.

[Feg96] describes an even more ambitious method of deriving a polymorphic skeleton from a recursive definition and using its parametricity theorem for fusion. The sketched algorithms are still far from an implementation. A type-inference based algorithm may be suitable for deriving the desired polymorphic recursion skeletons.

Gill lists several more deforestation methods, especially for lists, in his thesis [Gil96].

Generally deforestation methods can be divided into two classes. First, there are the unfold/fold based deforestation methods. Second, there are methods such as short cut deforestation which abandon the treatment of generally recursive definitions in favour of some “regular” forms 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. The composition methods of attribute grammars and related formalisms are on the borderline of the two deforestation classes. However, because they rely on the restricted recursion schemes of the formalisms they resemble more the “regular” recursion methods than the unfold/fold deforestation methods.

Chapter 8

Conclusions

The starting-point of this thesis was the observation that the short cut fusion rule for removing an intermediate list is a parametricity theorem. If we abstract the list constructors `(:)` and `[]` from a list producer so that the remaining producer skeleton has a given polymorphic type, then we can apply the short cut fusion rule, which can fuse any list consumer in `foldr` form with the producer. The producer skeleton must be polymorphic to ensure type-correctness of the transformation. Conversely, the polymorphic type of the producer skeleton already guarantees that we have abstracted exactly the constructors which construct the intermediate list and hence the transformation is semantically correct. So we use a type inference algorithm to determine the occurrences of list constructors which have to be abstracted to obtain the polymorphic producer skeleton. On the basis of the well-known Hindley-Milner type inference algorithm we developed a suitable type inference algorithm. It is at the heart of the list abstraction algorithm which yields the polymorphic producer skeleton and which we have presented in Chapter 4. The original short cut deforestation algorithm can only apply the short cut fusion rule when the producer is already given in terms of its polymorphic producer skeleton. The special second-order typed function `build` is used to mark a polymorphic producer skeleton. As we argued in the introduction, no programmer should be required to use `build`. Therefore the short cut deforestation algorithm currently implemented in the Glasgow Haskell compiler restricts deforestation to list-producing functions from the standard libraries, which are defined in terms of `build` by the compiler writer. Our new type-inference based algorithm for deriving the polymorphic producer skeleton renders hand made polymorphic producer skeletons unnecessary. Thus we can apply short cut fusion also to user defined list producers.

To enable deforestation across module boundaries without large-scale inlining we adapted Gill's worker/wrapper scheme to our `build`less setting. The benefit of not needing `build` showed up. Our worker/wrapper scheme is more flexible in that it can handle list-producing functions which cannot be defined in terms of `build`, as we discussed in Section 7.1.2. For example, we can handle producers of several lists and partial list producers such as `(++)`. We gave algorithms to automatically split a recursive function definition into a worker definition and a wrapper definition. These algorithms are based again on the list abstraction algorithm. Altogether we have defined a new short cut deforestation algorithm based on type inference.

From the principal typing property of the type inference algorithm we were able to deduce in Section 4.5 the completeness of our list abstraction algorithm. Whereas all purely syntax-directed deforestation methods raise the question of when and how often they are successful, we know that, if a list can be abstracted from a producer by abstracting its list constructors, then our list abstraction algorithm will abstract it. Hence our deforestation algorithm can fuse a `foldr` consumer with nearly any producer. The main reason why list constructors might not be abstractable from a producer is that the list constructors do not occur in the producer expression but in the definition of a function which is called by the producer. The worker/wrapper scheme ensures that these list constructors are moved to the producer and hence list abstraction becomes possible. In Section 3.4.3 we gave an example for a producer which cannot be list abstracted, because it recursively consumes its own result but is not recursively defined. This producer would require some substantial transformation, which we do not perform, before its result list constructors could be abstracted.

The major advantage of our deforestation method compared to other methods such as warm fusion, hylomorphism fusion and unfold/fold deforestation (Chapter 7) is that it is simple, nearly as simple as the more restrictive original short cut deforestation method. Our type inference algorithm is more simple than the well-known Hindley-Milner type inference algorithm and most other parts of our deforestation algorithm are straightforward. The simplicity has several consequences:

- We were able to give a complete formal proof of the semantic correctness of our list abstraction algorithm (Chapter 4) and most of the remaining parts of the deforestation algorithm.

- Its simplicity also makes it easier to ensure that the transformation always improves the performance of a program. We both discussed the effect of the transformation at the hand of a formal cost model and measured it. In contrast, the effect of warm fusion and hylomorphism fusion on program performance has not been studied. Already our simple worker/wrapper split has a small negative effect on program performance. We detected that the hylomorphism derivation algorithm uses common subexpression elimination, which may degrade performance. Hence it seems unlikely that complex transformations such as warm fusion and hylomorphism fusion do not sometimes degrade performance. A formal proof of improvement of our algorithm could give more insights but will still require substantial research into appropriate proof methods. Even then measurements are important, if only to validate the appropriateness of the cost model. Our measurements already suggest that the size of heap cells, which our cost model abstracts from, is relevant in practice.

- Our deforestation algorithm is also *transparent* [Mar95], that is, the programmer can easily determine from the source program where deforestation will be applicable. Our deforestation algorithm requires the consumer of an intermediate list to be defined in terms of `foldr`. It can handle nearly any producer. In our experience a programmer can usually see when a producer cannot be list abstracted without having to go through the details of type inference.

- Our deforestation algorithm is efficient. We noted in Section 4.5 that the list abstraction algorithm can be implemented to work in linear time. We noted in Section 6.3 that the worker/wrapper split algorithm has a higher complexity. However, on the one hand we do not believe that the worst case occurs in practice and on the other hand we suggested a path to a more efficient algorithm by integrating type inference and worker/wrapper split.

- Compared to other deforestation methods our deforestation algorithm is simple to implement. We derived the prototype implementation of the list abstraction algorithm in a straightforward way from the formal description. Because we concentrated on the theoretical foundations of the deforestation algorithm, especially its completeness and correctness, a full implementation was unfortunately outside the scope of the thesis.

The measurements of Section 5.3 suggest that the secondary effect of deforestation, that is, enabling further optimisations, is important. So workers should be inlined as far as possible. To ensure this the ability of our type-inference based method to clearly indicate which definitions need to be inlined, a control which has been much searched for in deforestation [Mar95, NP98], may be useful. In general, an optimising compiler of a high-level programming language has to undo the abstractions of a program. Hence it has to avoid unnecessary inlining but nonetheless transform a modular, concise program into efficient, longer code.

We presented our deforestation method for the second-order typed language F, because it is small and similar to the intermediate language of the Glasgow Haskell compiler. However, the full second-order type system is not necessary for our list abstraction algorithm. The parametricity theorems and hence the short cut fusion rule are valid for any subsystem. The list abstraction algorithm just requires the possibility to abstract type variables, not necessarily even explicitly in the term language. Because short cut deforestation is a local transformation, it is also desirable that all `let`-defined variables have an explicit type annotation. Then the typing environment of a term can be constructed without type inference of the whole program. So also a Hindley-Milner typed language can be used. The deforestation algorithm could be directly defined for Haskell programs. We used that in the queens example in Section 5.3. Only the class system and naturally the size of the language will complicate the algorithm.

In Chapter 6 we suggested several directions for future extensions of our deforestation method. So we noted that our list abstraction algorithm is not specific to lists but can be used for nearly arbitrary recursive data types. Our work started with the observation that the short cut fusion rule is the parametricity theorem of the polymorphic type of the producer skeleton. There are also many other parametricity theorems which might be exploited for type-inference based program transformations. Hylomorphism fusion uses two parametricity theorems. [Feg96] suggests to derive a polymorphic skeleton from a recursive definition and use its parametricity theorem for fusion. The transformation system in [TH98] is based on the parametricity theorem of the fixpoint combinator.

From a wider perspective we can say that we used type inference as an analysis which steers the deforestation transformation. In recent years an increasing number of type systems (including type and effect systems) have been developed for various kinds of program analysis [NNH99]. These systems use type systems to describe properties by syntactic objects, relate them to program expressions by structural rules and use inference algorithms based on unification or constraint solving for analysis. However, they are non-standard type systems, that is, their types do not coincide with the types of the programming language. Here we showed that standard polymorphic type systems can be applied to infer useful information about a program. We believe that there are many other possible applications of polymorphic type inference to program analysis. List abstraction can be seen as based on exchanging the standard representation of a list type `[`τ`]` by a representation as a function of type $\forall\gamma.(\tau \to \gamma \to \gamma) \to \gamma \to \gamma$. A wrapper converts the latter into the former. It would be worthwhile to enquire how close the relationship of our work to representation analysis is. In [Hal94] Hindley-Milner type inference is used to choose efficient representations of lists. In general, type inference determines some kind of data flow. Hence a strand of future research could be to examine the relationship of polymorphic type inference to data and control flow analysis.

Bibliography

[Abr90] Samson Abramsky. The lazy lambda-calculus. In *Research topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw Hill, second edition, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.

[Aug87] Lennart Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Department of Computer Science, Chalmers University, Sweden, November 1987.

[Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1981.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[BD77] Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BS94] Franz Baader and Jörg H. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 41–125. Oxford University Press, Oxford, UK, 1994.

[BS98] Erik Barendsen and Sjaak Smetsers. Strictness typing. In *Draft Proceedings of the 10th International Workshop on Implementation of Functional Languages*, pages 101–115, University College London, UK, 1998.

[BW87] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. International Series in Computer Science. Prentice-Hall, 1987.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

[CDPR97] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Attribute grammars and functional programming deforestation. In *Fourth International Static Analysis Symposium – Poster Session*, LNCS 1302, September 1997.

[CDPR99] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Declarative program transformation: a deforestation case-study. In *Principles and Practice of Declarative Programming, International Conference PPDP'99*, LNCS 1702. Springer, 1999.

[CF58] Haskell Brooks Curry and Robert Feys. *Combinatory Logic, Volume I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[Chi92] Wei-Ngan Chin. Safe fusion of functional expressions. In Jon L. White, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 11–20. ACM Press, June 1992.

[Chi94] Wei-Ngan Chin. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming*, 4(4):515–555, October 1994.

[Chi97] Olaf Chitil. Adding an optimisation pass to the Glasgow Haskell compiler. Available from `http://www.haskell.org/ghc`, November 1997.

[Chi98] Olaf Chitil. Common subexpressions are uncommon in lazy functional languages. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages 1997*, LNCS 1467, pages 53–71. Springer, 1998.

[Chi99] Olaf Chitil. Type inference builds a short cut to deforestation. *ACM SIGPLAN Notices*, 34(9):249–260, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[Chi00] Olaf Chitil. Type-inference based short cut deforestation (nearly) without inlining. In *Proceedings of the 11th International Workshop on Implementation of Functional Languages 1999*, LNCS. Springer, 2000. to appear.

[CK95] Wei-Ngan Chin and Siau-Cheng Khoo. Better consumers for deforestation. In *7th International Symposium on Programming Languages: Implementation, Logics and Programs*, LNCS 982, pages 223–240, 1995.

[Dav87] Ken Davies. Deforestation: Transformation of functional programs to eliminate intermediate trees. Master's thesis, Programming Research Group, Oxford University, September 1987.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, Amsterdam, 1990.

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

[Ede85] Elmar Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1(1):31–46, March 1985.

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

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Science*, 31:71–145, 1985.

[Feg93] Leonidas Fegaras. Efficient optimization of iterative queries. In *Fourth International Workshop on Database Programming Languages*, pages 200–225. Springer, Workshops on Computing, 1993.

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

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, 1996.

[FSZ94] Leonidas Fegaras, Tim Sheard, and Tong Zhou. Improving programs which recurse over multiple inductive structures. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 21–32, 1994.

[Fül81] Zoltán Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–279, 1981.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-directed semantics — Formal models based on tree transducers.* Monographs in Theoretical Computer Science, An EATCS Series. Springer, 1998.

[FW87] John Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, LNCS 274, pages 34–46. Springer, 1987.

[FW88] Alistair B. Ferguson and Philip Wadler. When will deforestation stop? In *Functional Programming, Glasgow 1988. Workshops in Computing*, pages 39–56, August 1988.

[Gan83] Harald Ganzinger. Increasing modularity and language–independency in automatically generated compilers. *Science of Computer Programming*, 3:223–278, 1983.

[GHC] The Glasgow Haskell compiler. `http://www.haskell.org/ghc/`.

[Gie88] Robert Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25:355–423, 1988.

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

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GLP93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM Press, 1993.

[Gol81] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, February 1981.

[Gor94a] Andrew Gordon. A tutorial on co-induction and functional programming. In *Glasgow functional programming workshop*, pages 78–95. Springer Workshops in Computing, 1994.

[Gor94b] Andrew D. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. CUP, September 1994.

[GP94] Andrew J. Gill and Simon L. Peyton Jones. Cheap deforestation in practice: An optimiser for Haskell. In Bjørn Pehrson and Imre Simon, editors, *Proceedings of the IFIP 13th World Computer Congress. Volume 1 : Technology and Foundations*, pages 581–586. Elsevier Science Publishers, 1994.

[GP99] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.

[GS99] Jörgen Gustavsson and David Sands. A foundation for space-safe transformations of call-by-need programs. In A. D. Gordon and A. M.Pitts, editors, *The Third International Workshop on Higher Order Operational Techniques in Semantics*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[Gun92] Carl A. Gunter. *Semantics of programming languages: structures and techniques.* Foundations of computing. MIT Press, 1992.

[Hal94] Cordelia V. Hall. Using Hindley Milner type inference to optimise list representation. In *Conference on Lisp and Functional programming*, 1994.

[Ham96] Geoff W. Hamilton. Higher order deforestation. In *8th International Symposium PLILP'96*, LNCS 1140, pages 213–227, 1996.

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structural Operational Semantics.* John Wiley and Sons, 1990.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60, 1969.

[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*, pages 73–82. ACM Press, 1996.

[HIT97] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An extension of the acid rain theorem. In T. Ida, A. Ohori, and M. Takeichi, editors, *Proceedings 2nd Fuji Intl. Workshop on Functional and Logic Programming*, pages 91–105, Singapore, 1997. World Scientific.

[HJ92] Geoff W. Hamilton and Simon B. Jones. Extending deforestation for first order functional programs. In R. Heldal, C. K. Holst, and P. L. Wadler, editors, *Functional Programming, Glasgow 1991: Proceedings of the 1991 Workshop*, pages 134–145. Springer, 1992.

[Höf99] Matthias Höff. Vergleich von Verfahren zur Elimination von Zwischenergebnissen bei funktionalen Programmen. Master's thesis, Dresden University of Technology, Department of Computer Science, September 1999.

[HTC98] Zhenjiang Hu, Masato Takeichi, and Wei-Ngan Chin. Parallelization in calculational forms. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–328, 1998.

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

[IHT98] Hideya Iwasaki, Zhenjiang Hu, and Masato Takeichi. Towards manipulation of mutually recursive definitions. In *Proceedings 3rd Fuji Intl. Workshop on Functional and Logic Programming*, 1998.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–53, 1996.

[Joh84] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69. ACM Press, 1984.

[Joh87] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, LNCS 274, pages 154–173. Springer, 1987.

[Jon95] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, LNCS 925, 1995.

[Jür99] Claus Jürgensen. Kategorientheoretisch fundierte Programmtransformationen für Haskell. Master's thesis, RWTH Aachen, 1999.

[Jür00] Claus Jürgensen. A formalization of hylomorphism based deforestation with an application to an extended typed λ-calculus. Technical Report TUD-FI00-13, Technische Universität Dresden, Fakultät Informatik, November 2000.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E.

Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[KL89] Philip J. Koopman, Jr. and Peter Lee. A fresh look at combinator graph reduction (or, having a TIGRE by the tail). In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 110–119, July 1989.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[KTU93] Assaf J. Kfoury, Jerzy Tiuryn, and Paweł Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.

[Küh98] Armin Kühnemann. Benefits of tree transducers for optimizing functional programs. In V. Arvind and R. Ramanujam, editors, *Foundations of Software Technology & Theoretical Computer Science – FST & TCS'98*, LNCS 1530, pages 146–157. Springer, December 1998.

[Küh99] Armin Kühnemann. Comparison of deforestation techniques for functional programs and for tree transducers. In A. Middeldorp and T. Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming – FLOPS'99*, LNCS 1722, pages 114–130. Springer, November 1999.

[KV94] Armin Kühnemann and Heiko Vogler. Synthesized and inherited functions — a new computational model for syntax–directed semantics. *Acta Informatica*, 31:431–477, 1994.

[Las98] Søren B. Lassen. *Relational Reasoning about Functions and Nondeterminism.* PhD thesis, Department of Computer Science, University of Aarhus, February 1998.

[Lau93] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154. ACM Press, 1993.

[LP94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. *ACM SIGPLAN Notices*, 29(6):24–35, June 1994.

[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*, pages 314–323. ACM Press, 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*, LNCS 523, pages 124–144. Springer, 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. ACM Press, 1995.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MR92] QingMing Ma and John C. Reynolds. Types, abstraction and parametric polymorphism, part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. A. Schmidt, editors, *Proceedings 7th Intl. Conf. on Math. Foundations of Programming Semantics, MFPS'91*, LNCS 598, pages 1–40. Springer, Berlin, 1992.

[MS99] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99, the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56. ACM Press, January 1999.

[MTHM97]Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[MW92] Simon Marlow and Philip Wadler. Deforestation for higher-order functions. In J. Launchbury and P.M. Sansom, editors, *1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 154–165. Springer, 1992.

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

[NAH$^+$95] Rishiyur S. Nikhil, Arvind, James E. Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Y. Zhou. pH language reference manual, version 1.0. Technical Report CSG-Memo-369, Massachussets Institute of Technology, Laboratory for Computer Science, January 1995.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

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

[OHIT97] Y. Onue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In R. Bird and L. Meertens, editors, *Proceedings IFIP TC 2 WG 2.1 Working Conf. on Algorithmic Languages and Calculi*, pages 76–106, London, 1997. Chapman & Hall.

[Par94] Will Partain. How to add an optimisation pass to the Glasgow Haskell compiler. Part of the Glasgow Haskell Compiler distribution, October 1994.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pey92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[PG00] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings*, LNCS. Springer, 2000. to appear.

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

[Pit94] Andrew M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Notes Series BRICS-NS-94-5, BRICS, Department of Computer Science, University of Aarhus, December 1994. vi+135 pp, draft version.

[Pit97] Andrew M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[Pit00] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in computer Science*, 10:1–39, 2000. A preliminary version appeared in *Proceedings, Second Workshop on Higher Order Operational Techniques in Semantics (HOOTS II)*, Electronic Notes in Theoretical Computer Science 10, 1998.

[PL91] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, LNCS 523, pages 636–666. Springer, June 1991.

[PM99] Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell compiler inliner. IDL '99, `http://www.binnetcorp.com/wshops/IDL99.html`, 1999.

[PS98] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998.

[Pv98] Rinus Plasmeijer and Marko van Eekelen. The Concurrent Clean language report. `http://www.cs.kun.nl/~clean`, 1998.

[PW78] Michael S. Paterson and Mark N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings, Colloque sur la Programmation*, LNCS 19, pages 408–425. Springer, 1974.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B.V.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Proceedings Intl. Symp. on the Semantics of Data Types*, LNCS 173, pages 145–156. 1984.

[Rob65] J. Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rou70] William C. Rounds. Mappings and grammars on trees. *Math. Syst. Theory*, 4:257–287, 1970.

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

[San96a] David Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1–2):193–233, October 1996.

[San96b] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[San97] David Sands. Improvement theory and its applications. In Andrew D. Gordon and Andrew M. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997.

[San98] David Sands. Computing with contexts: A simple approach. In A. D. Gordon, A. M. Pitts, and C. L. Talcott, editors, *Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B.V., 1998.

[Sch00] Jacob B. Schwartz. Eliminating intermediate lists in ph. Master's thesis, MIT Department of Electrical Engineering and Computer Science, May 2000.

[Sei96] Helmut Seidl. Integer constraints to stop deforestation. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, LNCS 1058, pages 326–340. Springer, 1996.

[Ses97] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Functional Programming and Computer Architecture*, pages 233–242, June 1993.

[SGJ94] Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In D. Sannella, editor, *European Symposium on Programming*, LNCS 788, pages 485–500. Springer, 1994.

[SI96] Jill Seaman and S. Purushothaman Iyer. An operational semantics of sharing in lazy evaluation. *Science of Computer Programming*, 27(3):289–322, November 1996.

[Sør94] Morten Heine Sørensen. A grammar-based data-flow analysis to stop deforestation. In S. Tison, editor, *Colloquium on Trees in Algebra and Programming*, LNCS 787, pages 335–351. Springer, 1994.

[SP97] Patrick M. Sansom and Simon L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997.

[SS97] Helmut Seidl and Morten H. Sørensen. Constraints to stop higher-order deforestation. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 400–413. ACM Press, 1997.

[SS98] Helmut Seidl and Morten H. Sørensen. Constraints to stop deforestation. *Science of Computer Programming*, 32(1–3):73–107, September 1998.

[Tay98] Simon Taylor. Optimization of Mercury programs. Department of Computer Science, University of Melbourne, 1998. Honours report.

[TH98] Mark Tullsen and Paul Hudak. An intermediate meta-language for program transformation. Technical Report YALEU/DCS/RR-1154, Yale University, June 1998.

[THT98] Akihiko Takano, Zhenjiang Hu, and Masato Takeichi. Program transformation in calculational form. *ACM Computing Surveys*, 30(3'), September 1998. Article 7.

[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. ACM Press, 1995.

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

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7'th International Conference on Functional Programming and Computer Architecture*, 1995.

[Wad84] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 45–52. ACM Press, August 1984.

[Wad85] Philip Wadler. Listlessness is better than laziness II: composing listless functions. In H. Ganziger and N. D. Jones, editors, *Proc. Workshop on Programs as Data Objects*, LNCS 217. Springer, 1985.

[Wad87] Philip Wadler. Compiling list comprehensions. Chapter 7 in [Pey87], 1987.

[Wad88] Philip Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, LNCS 300, pages 344–358. Springer, 1988.

[Wad89] Philip Wadler. Theorems for free! In *4th International Conference on Functional Programming Languages and Computer Architectures*, pages 347–359. ACM Press, 1989.

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

[Wel94] Joe 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. IEEE Computer Society Press, 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing series. MIT Press, February 1993.

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

Appendix A

Rules

In this thesis we often use rules to state propositions and to define relations.

A **rule** consists of a number of premise propositions A_i above a horizontal line and a single conclusion proposition B below the line:

$$\frac{A_1 \quad A_2 \quad \ldots \quad A_n}{B}$$

In principle it is just a more convenient form to write the proposition

$$\forall x_1, \ldots, x_k \quad A_1 \wedge A_2 \wedge \ldots \wedge A_n \Rightarrow B$$

where the universally quantified meta-variables $x_1, \ldots x_k$ are all the meta-variables that occur free in $A_1, \ldots, A_n$ and B. We use naming conventions for meta-variables to specify over which sets the variables are quantified. For example, l, m and n are natural numbers. The premise of a rule may be empty, thus expressing an axiom $\forall x_1, \ldots, x_k\ B$.

Additionally, rules are understood as syntactic objects that are used to syntactically construct proofs of properties. A proof or **derivation** is a tree of properties with leaves at the top and a root at the bottom, where each property is obtained from the ones immediately above it by some rule. For example the rules

$$\frac{}{n \leq n+1} \quad \text{and} \quad \frac{l \leq m \quad m \leq n}{l \leq n}$$

can be used to construct the derivation

$$\frac{\dfrac{}{1 \leq 2} \quad \dfrac{}{2 \leq 3}}{1 \leq 3}$$

We say that the root property $1 \leq 3$ **is derived** from the rules.

Furthermore, we use rules to define sets, especially relations. A set R is (inductively) defined by a set or **system of rules**. These rules define R by defining all valid **judgements** $x \in R$. A judgement is **valid**, if it can be derived from the rules of the system. For example, the two rules given before define the canonical order $\leq$ on natural numbers.

A rule system to define a set R contains R only in judgements $x \in R$, not for example in a proposition $x \notin R$. Thus it is assured that the set R is well-defined. R can be characterised as the least set for which all the rules of the system hold.

We can prove by **rule induction** that a property holds for all elements of a set that we defined through a rule system. Rule induction means that we show for all rules that if the property holds for all elements of R mentioned in the premise judgements then it also holds for the element mentioned in the conclusion judgement. This proof method can be understood as an induction on the depth of the derivation of a judgement. We loosely speak of "induction on the rules".

More detailed descriptions of the use of rules for type systems are given in [Car97] and for operational semantics in [Hen90, Win93].

Appendix B

Standard List Functions Used in the Program Queens

The following definitions of standard list functions were used to measure the performance of the queens program without deforestation.

```
sum :: [Int] -> Int
sum = foldl' (+) 0

concat :: [[a]] -> [a]
concat = foldr (++) []

map :: (a -> b) -> [a] -> [b]
map f = foldr ((:) . f) []

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs

and :: [Bool] -> Bool
and = foldr (&&) True

zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []

enumFrom :: Int -> [Int]
enumFrom l = l : (enumFrom $! (l+1))

enumFromTo :: Int -> Int -> [Int]
enumFromTo l m = takeWhile (<= m) (enumFrom l)
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p = foldr (\x fxs -> if p x then x:fxs else []) []

foldl'           :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []     = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

Next the result of applying our deforestation algorithm to the definitions is shown. The definitions of all list-producing functions are split into worker and wrapper definitions. In the definitions of `concapMap` and `enumFromTo` deforestation removes an intermediate list. The static argument transformation performed in the definitions of `zipW` and `enumFromW` is discussed in Section 5.1.2.

```
sum :: [Int] -> Int
sum = foldl' (+) 0

concatW :: (a -> c -> c) -> c -> [[a]] -> c
concatW c n = foldr (appW c) n

concat :: [[a]] -> [a]
concat = concatW (:) []

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

map :: (a -> b) -> [a] -> [b]
map = mapW (:) []

concatMapW :: (b -> c -> c) -> c -> (a -> [b]) -> [a] -> c
concatMapW c n f xs = mapW (appW c) n f xs

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap = concatMapW (:) []

appW :: (a -> c -> c) -> [a] -> c -> c
appW c xs ys = foldr c ys xs

(++) :: [a] -> [a] -> [a]
(++) = appW (:)

and :: [Bool] -> Bool
and = foldr (&&) True

zipW :: ((a,b) -> c -> c) -> c -> [a] -> [b] -> c
zipW c n xs ys = go xs ys
  where
  go (x:xs) (y:ys) = c (x,y) (go xs ys)
  go _      _      = n

zip :: [a] -> [b] -> [(a,b)]
zip = zipW (:) []

enumFromW :: (Int -> c -> c) -> Int -> c
enumFromW c l = go l
  where
  go l = l `c` (go $! (l+1))

enumFrom :: Int -> [Int]
enumFrom = enumFromW (:)

enumFromToW :: (Int -> c -> c) -> c -> Int -> Int -> c
enumFromToW c n l m =
  enumFromW (\x fxs -> if (<= m) x then x `c` fxs else n) l

enumFromTo :: Int -> Int -> [Int]
enumFromTo = enumFromToW (:) []

takeWhileW :: (a -> c -> c) -> c -> (a -> Bool) -> [a] -> c
takeWhileW c n p = foldr (\x fxs -> if p x then x `c` fxs else n) n

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile = takeWhileW (:) []

foldl'           :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []     = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

The following program is the result of applying only the worker/wrapper split algorithm to the program of Figure 5.4. So it represents an intermediate step between that program and the deforested program of Figure 5.5.

```
main = (print . sum . concatW (:) [] . queensW (:) [] (:) []) 10

queensW :: (Int -> c -> c) -> c -> (c -> d -> d) -> d -> Int -> d
queensW c1 n1 c2 n2 0 = c2 n1 n2
queensW c1 n1 c2 n2 m =
  concatMapW c2 n2
    (\p -> concatMapW (:) []
             (\l -> if safe p l then [appW c1 p (c1 l n1)] else [])
             (enumFromToW (:) [] 1 10))
    (queensW (:) [] (:) [] (m-1))

queens :: Int -> [[Int]]
queens = queensW (:) [] (:) []

safe :: [Int] -> Int -> Bool
safe p n = and
  (concatMapW (:) [] (\(i,j)-> [(j /= n) && (i+j /= m+n) && (i-j /= m-n)])
    (zipW (:) [] (enumFromW (:) 1) p))
  where
  m = length p + 1
```

```
sum :: [Int] -> Int
sum = foldl' (+) 0

concatW :: (a -> c -> c) -> c -> [[a]] -> c
concatW c n = foldr (appW c) n

concat :: [[a]] -> [a]
concat = concatW (:) []

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

map :: (a -> b) -> [a] -> [b]
map = mapW (:) []

concatMapW :: (b -> c -> c) -> c -> (a -> [b]) -> [a] -> c
concatMapW c n f = concatW c n . mapW (:) [] f

concatMap :: (a -> [b]) -> [a] -> [b]
concatMap = concatMapW (:) []

appW :: (a -> c -> c) -> [a] -> c -> c
appW c xs ys = foldr c ys xs

(++) :: [a] -> [a] -> [a]
(++) = appW (:)

and :: [Bool] -> Bool
and = foldr (&&) True

zipW :: ((a,b) -> c -> c) -> c -> [a] -> [b] -> c
zipW c n xs ys = go xs ys
  where
  go (x:xs) (y:ys) = c (x,y) (go xs ys)
  go _      _      = n

zip :: [a] -> [b] -> [(a,b)]
zip = zipW (:) []

enumFromW :: (Int -> c -> c) -> Int -> c
enumFromW c l = go l
  where
  go l = l `c` (go $! (l+1))

enumFrom :: Int -> [Int]
enumFrom = enumFromW (:)

enumFromToW :: (Int -> c -> c) -> c -> Int -> Int -> c
enumFromToW c n l m = takeWhileW c n (<= m) (enumFromW (:) l)
```

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo = enumFromToW (:) []

takeWhileW :: (a -> c -> c) -> c -> (a -> Bool) -> [a] -> c
takeWhileW c n p = foldr (\x fxs -> if p x then x ‘c‘ fxs else n) n

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile = takeWhileW (:) []

foldl'            :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []      = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

Lebenslauf

Name:	Olaf Chitil
geboren am:	14. Juni 1969
in:	Süchteln, jetzt Viersen

Aug. 1975	–	Nov. 1977	Grundschule Essen-Frintrop
Nov. 1977	–	Jul. 1979	Sollbrüggen-Grundschule, Krefeld-Bockum
Aug. 1979	–	Jun. 1988	Gymnasium am Stadtpark, Uerdingen; Abschluß Abitur
Jul. 1988	–	Sep. 1989	Wehrdienst
Okt. 1989	–	Jun. 1995	Studium der Informatik mit Nebenfach Physik an der RWTH Aachen; Abschluß Diplom
Okt. 1992	–	Jun. 1993	Studium Computer Science an der University of Kent at Canterbury, UK; Abschluß Diploma in Computer Science
Aug. 1995	–	Jun. 2000	Wissenschaftlicher Angestellter am Lehrstuhl für Informatik II der RWTH Aachen
seit Juni 2000			Research Associate am Department of Computer Science, University of York, UK