Concatenate, Reverse and Map Vanish For Free

Janis Voigtländer[*]
Department of Computer Science
Dresden University of Technology
01062 Dresden, Germany
voigt@tcs.inf.tu-dresden.de

Abstract

We introduce a new transformation method to eliminate intermediate data structures occurring in functional programs due to repeated list concatenations and other data manipulations (additionally exemplified with list reversal and mapping of functions over lists).

The general idea is to uniformly abstract from data constructors and manipulating operations by means of rank-2 polymorphic combinators that exploit algebraic properties of these operations to provide an optimized implementation. The correctness of transformations is proved by using the *free theorems* derivable from parametric polymorphic types.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types, polymorphism*; D.3.4 [**Programming Languages**]: Processors—*optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms

Languages, Algorithms, Design

Keywords

Combinators, correctness proofs, denotational semantics, list abstraction, parametricity, program transformation, rank-2 types, shortcut deforestation, the concatenate vanishes, theorems for free

[*]Research supported by the "Deutsche Forschungsgemeinschaft" under grant KU 1290/2-1 and by the "Gesellschaft von Freunden und Förderern der TU Dresden" with a travel grant.

1 Introduction

Consider the following definition in the functional programming language Haskell, implementing the partition of a list l according to some predicate p:

```
part :: ∀α . (α → Bool) → [α] → [α]
part p l = let f    []     z = z
               f (x : xs) z = if p x then x : (f xs z)
                                      else  f xs (z ++ [x])
           in  f l []
```

While serving as a good specification of *part*, this definition has an inefficient runtime behavior due to repeated concatenate operations $(+\!\!+)$ on intermediate lists in the second argument position of f. Since $+\!\!+$ requires runtime linear in the length of its left argument, this overhead can be considerable. Similar efficiency problems due to a modular programming style can also be caused by other data manipulating operations often used in specifications.

In order to optimize function definitions that make extensive use of $+\!\!+$, Hughes [12] proposed an alternative list representation supporting efficient concatenations. Therefor, a list xs is represented as the function $(\lambda ys \rightarrow xs +\!\!+ ys)$. It is left to the programmer to decide where conversion between the two representations should be performed, hence the transformation is not very systematic.

Wadler [27] presented an algorithmic transformation that introduces *accumulating parameters* and can achieve many — but not all — of the effects of Hughes' approach. For example, it is not applicable to the above *part*-function.

The contribution of this paper is an optimization technique for eliminating concatenate operations with the following characteristics:

- It **is applicable to more programs** than Wadler's method (cf. Sections 6 and 7.2).
- It **extends to a general methodology** for eliminating also other data manipulating operations than list concatenation (cf. Sections 4 and 5).
- It produces function definitions that closely resemble the original specifications, and thus **does not hamper readability and maintainability** of transformed programs.
- It has a sound semantic basis as it **facilitates concise correctness proofs** of transformations by using Wadler's *free theorems* [28].
- It **requires no compiler modification**, only that function definitions are written in a special form (similarly to using *build* for *shortcut deforestation* [10]).

- This special form moreover **can be obtained automatically** by using Chitil's approach of *list abstraction through type inference* [6, 7].

For example, our new method replaces the above *part*-function by the following:

```
part⋆ :: ∀α . (α → Bool) → [α] → [α]
part⋆ p l = vanish++
  (λn c a → let f   []    z = z
                f (x : xs) z = if p x then x ‘c‘ (f xs z)
                                      else  f xs (z ‘a‘ (x ‘c‘ n))
            in  f l n)
```

This definition takes advantage of a rank-2 polymorphic function $vanish_{++}$ that is provided in Figure 1. All the programmer needs to know in order to benefit from our concatenate elimination method is a certain law about $vanish_{++}$ that will be proved in Section 3.1. We will show in Section 4 how to systematically develop such reusable *vanish*-combinators also for other data manipulations than list concatenation.

```
vanish++ :: ∀α . (∀β . β → (α → β → β)
                     → (β → β → β) → β) → [α]
vanish++ g = g id (λx h ys → x : (h ys)) (∘) []
```

Figure 1. Definition of $vanish_{++}$**.**

Note that the above definition of $part^{\star}$ closely resembles the original specification of *part*, if we read n, ‘c‘ and ‘a‘ as [], : and ++, respectively. But while the original function *part* has a quadratic worst-case time complexity (disregarding the time for executing p) in the length of its input list l, the obtained $part^{\star}$ is only of linear time complexity, as exemplified in the following measurements[1]:

$n =$	3000	5000	7000	9000	11000
part even [1..*n*]	0.4	1.1	2.2	3.5	5.6
$part^{\star}$ *even* [1..*n*]	0.004	0.006	0.009	0.012	0.015

The following prerequisites are necessary to apply our method:

1. We need a type system that supports *rank-2 polymorphism* [17], which is the case, e.g., for most current Haskell implementations, but also for MetaML [25].
2. We have to believe in the validity of free theorems for the functional language under consideration. Although the correctness of these free theorems relies on relationally parametric models [23] — which are not known to exist for modern functional languages — Pitts' recent proof [22] of the existence of such models for lambda calculi with higher-order polymorphic functions and fixpoint recursion justifies this assumption. For example, Johann [14] used Pitts' result to justify shortcut deforestation — the correctness of which also depends on parametricity — for languages like Haskell.

Although we use Haskell throughout this paper, our methodology is also applicable to other lazy or strict functional languages meeting the above two prerequisites.

The remainder of this paper is organized as follows. Section 2 considers the used functional language and its semantics. Section 3 presents our technique for eliminating concatenate operations and discusses its application on examples. Section 4 uses the elimination of list reversal to illustrate the general methodology. Section 5 presents the result of applying this methodology to eliminate ++, *reverse* and *map*. Section 6 shows how to eliminate concatenate operations also from producers of nested lists. Section 7 compares our method with related work and Section 8 concludes.

[1]The runtimes (in seconds) shown here and in the following tables were measured on a SPARC workstation using the profiling capabilities of the optimizing Glasgow Haskell Compiler.

2 Functional Language

We use the pure and lazy functional language Haskell [2]. Definitions of Haskell functions that we use throughout the paper are given in Figure 2. Except for *swap* and *build*, these functions are part of the standard Haskell prelude, though the actual definitions may differ between Haskell implementations. Our performance measurements will use the Glasgow Haskell Compiler's prelude definitions, which might be more efficient but are semantically equivalent to the clearer definitions in Figure 2.

```
(++) :: ∀α . [α] → [α] → [α]
   []    ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)

reverse :: ∀α . [α] → [α]
reverse l = let f   []    ys = ys
                f (x : xs) ys = f xs (x : ys)
            in  f l []

map :: ∀α β . (α → β) → [α] → [β]
map f   []    = []
map f (x : xs) = (f x) : (map f xs)

id :: ∀α . α → α
id u = u

(∘) :: ∀α β γ . (β → γ) → (α → β) → α → γ
(f1 ∘ f2) u = f1 (f2 u)

fst :: ∀α β . (α,β) → α
fst (h1,h2) = h1

snd :: ∀α β . (α,β) → β
snd (h1,h2) = h2

swap :: ∀α β . (α,β) → (β,α)
swap (h1,h2) = (h2,h1)

foldr :: ∀α β . (α → β → β) → β → [α] → β
foldr c n   []    = n
foldr c n (x : xs) = x ‘c‘ (foldr c n xs)

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

Figure 2. Some Haskell functions.

2.1 Semantics

Unfortunately, there is not yet a formal semantics for the whole of Haskell, independent of any concrete implementation. Nevertheless, it is common practice to use a denotational style for reasoning about Haskell programs. We also follow this approach and very briefly recall the necessary notions, in particular the approximation $\sqsubseteq$ between values of the same type, interpreted as "less or equally defined as", and the value $\bot$ at every type, interpreted as "undefined". The reader who is mainly interested in the development and application of our transformation method rather than in the formal proofs might want to skip the rest of this section — as wells as the appendices — and only consider the laws in Figure 3, where all free variables are universally quantified over the appropriate types.

$$\bot\ u = \bot \tag{1}$$

$$\bot +\!\!+\, xs = \bot \tag{2}$$

$$(xs +\!\!+\, ys) +\!\!+\, zs = xs +\!\!+\, (ys +\!\!+\, zs) \tag{3}$$

$$xs +\!\!+\, [] = xs \tag{4}$$

$$fst\ (swap\ h) = snd\ h \tag{5}$$

$$snd\ (swap\ h) = fst\ h \tag{6}$$

$$reverse\ \bot = \bot \tag{7}$$

$$reverse\ (reverse\ xs) \sqsubseteq xs \tag{8}$$

$$reverse\ (x : xs) = (reverse\ xs) +\!\!+\, [x] \tag{9}$$

$$reverse\ (xs +\!\!+\, ys) \sqsubseteq (reverse\ ys) +\!\!+\, (reverse\ xs) \tag{10}$$

$$map\ f\ (xs +\!\!+\, ys) = (map\ f\ xs) +\!\!+\, (map\ f\ ys) \tag{11}$$

$$map\ f\ (map\ k\ xs) = map\ (f \circ k)\ xs \tag{12}$$

$$reverse\ (map\ k\ xs) = map\ k\ (reverse\ xs) \tag{13}$$

$$map\ f\ \bot = \bot \tag{14}$$

$$map\ id\ xs = xs \tag{15}$$

$$foldr\ (:)\ []\ xs = xs \tag{16}$$

$$foldr\ c\ n\ (build\ g) = g\ c\ n \tag{17}$$

Figure 3. Some simple laws.

In the spirit of denotational semantics [24], types are considered as sets equipped with the partial order $\sqsubseteq$, the least element $\bot$, and limits of all non-empty chains. A relation between such pointed complete partial orders is called *strict* if it contains the pair $(\bot,\bot)$. A relation is called *continuous* if the limits of two chains of pairwise related elements are again related. A strict and continuous relation is called *admissible*. A function is called *monotonic* if it preserves the approximation order. All functions definable in Haskell are monotonic and continuous.

Note that in the laws (8) and (10) from Figure 3 the approximation $\sqsubseteq$ may not be replaced by equality, because *xs* might be an infinite list in both cases, giving $\bot$ on the left-hand sides but not necessarily on the right-hand sides.

Further, note that law (17) — which is at the heart of shortcut deforestation for Haskell lists [10] — does not necessarily hold if *g* is defined using strict evaluation with the polymorphic primitive *seq* (see the counterexample in Appendix B). This is so, because *seq* — being available at every type — weakens the free theorems for Haskell. For the same reason, we will for now assume that *seq* is not used in the programs to be transformed. Since strict data types and the strict application function \$! are defined in terms of *seq*, they will also be excluded. Then, in Appendix B, we will consider how the proofs of our theorems fare in the presence of the strict evaluation primitive, and will see that — in contrast to shortcut deforestation — our optimizations are not hindered by allowing *seq* into the functional language.

3 More Concatenates Vanish

In this section we present our technique for eliminating calls to the list concatenation function. We first give a theorem with formal proof and then consider examples to illustrate how programs are optimized by applying this theorem.

3.1 A Theorem For Free

Consider the rank-2 polymorphic function $vanish_{+\!+}$ as defined in Figure 1. In the following we present a theorem that gives an alternative — but equivalent — semantics for $vanish_{+\!+}$. The proof of this result exploits the free theorem that comes with every polymorphic type [28], and is given in a similar style as the correctness proof for shortcut deforestation in [10]. Note that the relation chosen in the free theorem must be admissible (strict and continuous), because we consider programs in a functional language with fixpoint recursion and possible non-termination.

THEOREM 1. *For every fixed type* A *and function*

$$g :: \forall\beta\,.\,\beta \to (\mathsf{A} \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \beta$$

holds:

$$\boxed{g\ []\ (:)\ (+\!\!+) \;=\; vanish_{+\!+}\ g} \tag{18}$$

PROOF. The free theorem associated with *g*'s type is that for every choice of types B and B', values $n :: \mathsf{B}$, $n' :: \mathsf{B}'$, $c :: \mathsf{A} \to \mathsf{B} \to \mathsf{B}$, $c' :: \mathsf{A} \to \mathsf{B}' \to \mathsf{B}'$, $a :: \mathsf{B} \to \mathsf{B} \to \mathsf{B}$ and $a' :: \mathsf{B}' \to \mathsf{B}' \to \mathsf{B}'$, and an admissible relation $\mathcal{R} \subseteq \mathsf{B} \times \mathsf{B}'$, the following implication holds:

$$\begin{array}{ll} & (n,n') \in \mathcal{R} \\ \wedge & (\forall x :: \mathsf{A}, (l,l') \in \mathcal{R}\,.\,(c\ x\ l, c'\ x\ l') \in \mathcal{R}) \\ \wedge & (\forall (l_1,l_1') \in \mathcal{R}, (l_2,l_2') \in \mathcal{R}\,.\,(a\ l_1\ l_2, a'\ l_1'\ l_2') \in \mathcal{R}) \\ \Rightarrow & (g\ n\ c\ a, g\ n'\ c'\ a') \in \mathcal{R}. \end{array}$$

Here the polymorphic function *g* is silently instantiated at type B in the application to *n*, *c* and *a*, and at type B' in the application to n', c' and a'. Instantiating the above with $\mathsf{B} = [\mathsf{A}]$, $\mathsf{B}' = [\mathsf{A}] \to [\mathsf{A}]$, $n = []$, $n' = id$, $c = (:)$, $c' = (\lambda x\ h\ ys \to x : (h\ ys))$, $a = (+\!\!+)$ and $a' = (\circ)$, we obtain:

$$\begin{array}{ll} & ([], id) \in \mathcal{R} \\ \wedge & (\forall x :: \mathsf{A}, (l,l') \in \mathcal{R}\,.\,(x : l, (\lambda x\ h\ ys \to x : (h\ ys))\ x\ l') \in \mathcal{R}) \\ \wedge & (\forall (l_1,l_1') \in \mathcal{R}, (l_2,l_2') \in \mathcal{R}\,.\,(l_1 +\!\!+\, l_2, l_1' \circ l_2') \in \mathcal{R}) \\ \Rightarrow & (g\ []\ (:)\ (+\!\!+), g\ id\ (\lambda x\ h\ ys \to x : (h\ ys))\ (\circ)) \in \mathcal{R}. \end{array}$$

Consider the relation

$$\mathcal{R} = \{(p,q) \in \mathsf{B} \times \mathsf{B}' \mid \forall u :: [\mathsf{A}]\,.\,p +\!\!+\, u = q\ u\}.$$

From laws (1) and (2) follows the strictness of $\mathcal{R}$. Continuity of $\mathcal{R}$ follows from the facts that $(+\!\!+)$ is a continuous function and that function application is also continuous. Hence, $\mathcal{R}$ is admissible and thus can be used in the above implication. Then, the three conjuncts in the precondition of this implication read as follows:

(i). $\forall u :: [\mathsf{A}]\ .\ [] \mathbin{+\!\!+} u = id\ u$

(ii). for every $x :: \mathsf{A}$, $l :: [\mathsf{A}]$ and $l' :: [\mathsf{A}] \to [\mathsf{A}]$:

$$\begin{aligned} & (\forall u :: [\mathsf{A}]\ .\ l \mathbin{+\!\!+} u = l'\ u) \\ \Rightarrow\ & (\forall u :: [\mathsf{A}]\ .\ (x : l) \mathbin{+\!\!+} u = (\lambda x\ h\ ys \to x : (h\ ys))\ x\ l'\ u) \end{aligned}$$

(iii). for every $l_1, l_2 :: [\mathsf{A}]$ and $l'_1, l'_2 :: [\mathsf{A}] \to [\mathsf{A}]$:

$$\begin{aligned} & (\forall u_1 :: [\mathsf{A}]\ .\ l_1 \mathbin{+\!\!+} u_1 = l'_1\ u_1) \\ \wedge\ & (\forall u_2 :: [\mathsf{A}]\ .\ l_2 \mathbin{+\!\!+} u_2 = l'_2\ u_2) \\ \Rightarrow\ & (\forall u\ :: [\mathsf{A}]\ .\ (l_1 \mathbin{+\!\!+} l_2) \mathbin{+\!\!+} u = (l'_1 \circ l'_2)\ u). \end{aligned}$$

Conditions (i) and (ii) follow from the definitions of $(\mathbin{+\!\!+})$ and id and by beta-reduction; condition (iii) follows from law (3) and the definition of $(\circ)$. The instantiated free theorem thus implies

$$\begin{aligned} \forall u :: [\mathsf{A}]\ .\quad & (g\ []\ (:)\ (\mathbin{+\!\!+})) \mathbin{+\!\!+} u \\ & = g\ id\ (\lambda x\ h\ ys \to x : (h\ ys))\ (\circ)\ u\,, \end{aligned}$$

which for $u = []$ gives — using law (4) and the definition of $vanish_{\mathbin{+\!\!+}}$ — the following calculation:

$$\begin{aligned} & g\ []\ (:)\ (\mathbin{+\!\!+}) \\ = & (g\ []\ (:)\ (\mathbin{+\!\!+})) \mathbin{+\!\!+} [] \\ = & g\ id\ (\lambda x\ h\ ys \to x : (h\ ys))\ (\circ)\ [] \\ = & vanish_{\mathbin{+\!\!+}}\ g. \end{aligned}$$

□

Note how in the above proof the relation $\mathcal{R}$ captures the essence of Hughes' pair of functions *rep* and *abs* [12].

3.2 Applying the Theorem

Theorem 1 can be used to optimize list producers from which all list constructors and occurrences of $(\mathbin{+\!\!+})$ have been abstracted uniformly. This proper abstraction is guaranteed by the polymorphic argument type of $vanish_{\mathbin{+\!\!+}}$. Hence, if the Haskell type-checker accepts an expression $vanish_{\mathbin{+\!\!+}}\ g$, then this expression really has the same semantics as $g\ []\ (:)\ (\mathbin{+\!\!+})$, but might be dramatically more efficient due to an optimized implementation of concatenation. We give two examples in the remainder of this section.

Example 1. Consider the following definitions, implementing the recursive solution to the "Towers of Hanoi" problem:

```
data Pos = L | M | R
hanoi :: Int → [(Pos, Pos)]
hanoi t = let f   1   p q r = (p,q) : []
              f (s+1) p q r = (f s p r q) ++ ((p,q) : (f s r q p))
          in  f t L R M
```

Abstracting from the list constructors [] and (:), and from $(\mathbin{+\!\!+})$, the function *hanoi* can also be written as:

```
hanoi t = (λn c a → let f   1   p q r = (p,q) ‘c‘ n
                        f (s+1) p q r = (f s p r q)‘a‘
                                        ((p,q) ‘c‘ (f s r q p))
                    in  f t L R M) [] (:) (++)
```

Using law (18) from Theorem 1, this is equivalent to:

```
hanoi* :: Int → [(Pos, Pos)]
hanoi* t = vanish++
  (λn c a → let f   1   p q r = (p,q) ‘c‘ n
                f (s+1) p q r = (f s p r q)‘a‘
                                ((p,q) ‘c‘ (f s r q p))
            in  f t L R M)
```

The standard **compiler-built-in** simplifying techniques *inlining* and *beta-reduction* [21] are enough to automatically obtain the variant

```
hanoi* t =
  (let f   1   p q r = λys → (p,q) : ys
       f (s+1) p q r = λys → f s p r q ((p,q) : (f s r q p ys))
   in  f t L R M) []
```

that closely corresponds — by *let-floating* [20] — to the result of applying Wadler's introduction of accumulating parameters [27] to *hanoi* (as shown in Section 7.2) and hence enjoys the same runtime benefits. ◇

As a further example for our technique, we consider what we can do about a function that was termed *silly* in [27] and could not be optimized there.

Example 2. Consider the following definition:

```
flatten :: ∀α . [[α]] → [α]
flatten l = let silly x   []   = x
                silly x (y : ys) = silly (x ++ y) ys
            in  silly [] l
```

If we try to use $vanish_{\mathbin{+\!\!+}}$ by simply abstracting from the visible list constructors [] and $(\mathbin{+\!\!+})$, an "Inferred type is less polymorphic than expected"-error is produced by the type-checker. This guides us to additionally abstract from the list constructors of y in the second equation of *silly*, by using law (16) as follows:

```
flatten l = let silly x   []   = x
                silly x (y : ys) = silly (x ++ (foldr (:) [] y)) ys
            in  silly [] l
```

Note that the additional traversal of y with $foldr\ (:)\ []$ makes the definition less efficient. However, we can now use $vanish_{\mathbin{+\!\!+}}$ type-correctly and obtain:

```
flatten* :: ∀α . [[α]] → [α]
flatten* l = vanish++
  (λn c a → let silly x   []   = x
                silly x (y : ys) = silly (x ‘a‘ (foldr c n y)) ys
            in  silly n l)
```

Performance measurements show that this definition is not so silly after all. In fact, the function *flatten* — with a worst-case time complexity quadratic in the size of its argument — has been transformed to *flatten**, which has the same semantics but only requires linear time:

$n =$	1000	3000	5000	7000
$flatten\ [[i] \mid i \leftarrow [1..n]]$	0.18	1.6	4.5	9.0
$flatten^\star\ [[i] \mid i \leftarrow [1..n]]$	0.003	0.008	0.014	0.019

It would be interesting to make the guidance by the type-checker suggested above more explicit. ◇

So far, we have presented three examples (for *part*, *hanoi* and *flatten*) of applying our method **by hand**. The approach to list abstraction was to replace some occurrences of list constructors [], (:) and $(\mathbin{+\!\!+})$ by variables, and then to use the rank-2 polymorphic type-checker for detecting whether this replacement was sufficient to express the list producer with $vanish_{\mathbin{+\!\!+}}$. Chitil [6] reduced the problem of list abstraction for shortcut deforestation to a decidable partial type inference problem. Thus, he obtained a linear-time algorithm to derive *build*-forms of list producers by abstraction from [] and (:). By additionally treating $(\mathbin{+\!\!+})$ as just another list constructor, Chitil's idea can be extended to **automatically** derive the desired $vanish_{\mathbin{+\!\!+}}$-forms for our optimization technique. The decision for which functions to attempt an automatic abstraction, could come from the user via a pragma that would cause the compiler to check which data manipulating operations are used inside a given definition and whether a corresponding *vanish*-combinator exists.

Alternatively, $vanish_{++}$ and related combinators — e.g., from the next sections — may be consciously used when writing new programs. For example, a programmer might want to implement a list producing algorithm that is most naturally expressed using repeated concatenations or *map*s. Being aware of the danger of inefficiencies due to intermediate results, she can choose the appropriate *vanish*-function from a pre-defined module containing lots of such combinators for different data manipulations. Note that in order to do so, the user does not need to know the actual definitions of those *vanish*-functions, but only a specification of their semantics as provided by statements like in Theorems 1, 2, 3 and 4 of this paper.

4 A General Methodology

In this section we demonstrate that the method from the previous section is an instance of a more general new methodology for supplying data types with cheap versions of frequently used manipulating operations. This will be exemplified by adding to the standard list data type support for reversing lists without efficiency penalties due to repeated reversals. As an example where this is useful, consider the following naïve definition of a function for shuffling a list:

```
shuffle :: ∀α . [α] → [α]
shuffle    []    = []
shuffle (x : xs) = x : (reverse (shuffle xs))
```

Since every application of *reverse* takes time linear in the length of its argument, the runtime of *shuffle* is quadratic in the length of the input list. With the technique developed in the next three subsections we will obtain a version of *shuffle* needing only linear runtime.

4.1 Freezing and Abstraction

Firstly, we introduce a richer data type with an additional constructor for the additional operation that we want to support, and adapt *shuffle* to produce a value of that richer type:

```
data List α = Nil | Cons α (List α) | Rev (List α)
shuffle′ :: ∀α . [α] → List α
shuffle′    []    = Nil
shuffle′ (x : xs) = Cons x (Rev (shuffle′ xs))
```

This step of introducing data constructors for "external functions" — here: Rev for *reverse* — is inspired by that of "freezing" in Kühnemann *et al.* [15]. In order to expose these constructors for later fusion, we use the following $build_{\mathsf{List}}$-function:

```
build_List :: ∀α . (∀β . β → (α → β → β)
                        → (β → β) → β) → List α
build_List g = g Nil Cons Rev
```

to **uniformly abstract** the constructors of List α, giving:

```
shuffle″ :: ∀α . [α] → List α
shuffle″ l = build_List (λn c r → let f    []    = n
                                      f (x : xs) = c x (r (f xs))
                                  in  f l)
```

4.2 Efficient Conversion

Since we are interested in a value of the original type [α] and not of the enriched data type List α, we need a conversion function from the enriched to the standard list type. Using the following function:

```
fold_List :: ∀α . List α → (∀β . β → (α → β → β)
                              → (β → β) → β)
fold_List     Nil      n c r = n
fold_List (Cons x xs)  n c r = c x (fold_List xs n c r)
fold_List   (Rev xs)   n c r = r (fold_List xs n c r)
```

a trivial solution would be:

```
convert :: ∀α . List α → [α]
convert l = fold_List l [] (:) reverse
```

However, this is clearly not a good solution, because the composed function $convert \circ shuffle''$ will be less efficient than the original *shuffle*, as the traversal with $fold_{\mathsf{List}}$ only causes additional runtime overhead while still repeated applications of *reverse* occur.

Hence, we have to improve *convert* by **inventing** a new function $convert^{\star}$ that can perform at least the same conversions more efficiently. More precisely, we want to have the semantic property

$$convert \sqsubseteq convert^{\star}$$

and $convert^{\star}$ should have a better runtime efficiency than *convert*. The solution in the particular case under consideration here is to introduce two mutually recursive functions with accumulating parameters as follows[2]:

```
convert* :: ∀α . List α → [α]
convert* l = let h1     Nil     ys = ys
                 h1 (Cons x xs) ys = x : (h1 xs ys)
                 h1   (Rev xs)  ys = h2 xs ys
                 h2     Nil     ys = ys
                 h2 (Cons x xs) ys = h2 xs (x : ys)
                 h2   (Rev xs)  ys = h1 xs ys
             in  h1 l []
```

In order to be able to apply shortcut deforestation in the next subsection, we express this as a — higher-order and tupled — $fold_{\mathsf{List}}$:

```
convert* l = fst (fold_List l
                    (λys → (ys,ys))
                    (λx h ys → (x : (fst (h ys)),
                                snd (h (x : ys))))
                    (λh ys → swap (h ys))
                    [])
```

4.3 Fusion

Following the developments from the previous two subsections, the shuffling of a list *l* — of the standard list type — can be performed by computing $convert^{\star}\ (shuffle''\ l)$. Now, we calculate by inlining:

```
  convert* (shuffle″ l)
= fst (fold_List (shuffle″ l)
                 (λys → (ys,ys))
                 (λx h ys → (x : (fst (h ys)), snd (h (x : ys))))
                 (λh ys → swap (h ys))
                 [])
= fst (fold_List (build_List (λn c r → let f    []    = n
                                           f (x : xs) = c x (r (f xs))
                                       in  f l))
                 (λys → (ys,ys))
                 (λx h ys → (x : (fst (h ys)), snd (h (x : ys))))
                 (λh ys → swap (h ys))
                 []).
```

[2]An interesting alternative is to reuse an already existing *vanish*-combinator for expressing a new conversion, in this case:

```
convert* l = vanish_++
  (λn c a → let h1     Nil     = n
                h1 (Cons x xs) = x ‘c‘ (h1 xs)
                h1   (Rev xs)  = h2 xs
                h2     Nil     = n
                h2 (Cons x xs) = (h2 xs) ‘a‘ (x ‘c‘ n)
                h2   (Rev xs)  = h1 xs
            in  h1 l)
```

By shortcut deforestation [14] for the List type constructor, we know that for every type A and function g with

$$g :: (\forall\beta\,.\,\beta \to (\mathsf{A} \to \beta \to \beta) \to (\beta \to \beta) \to \beta)$$

the following **fusion law** holds:

$$fold_{\mathsf{List}}\ (build_{\mathsf{List}}\ g) \ = \ g.$$

Hence, the above calculation can be continued:

```
= fst ((λn c r → let f    []    = n
                     f (x : xs) = c x (r (f xs))
                 in  f l)
   (λys → (ys,ys))
   (λx h ys → (x : (fst (h ys)), snd (h (x : ys))))
   (λh ys → swap (h ys))
   []).
```

In the next subsection we will see that this calculated program shuffles the list l much more efficiently than the original *shuffle*-function did. The reason is that the original occurrences of list *reverse* have been replaced by an optimized implementation making use of the fact that two consecutive reversals cancel each other.

4.4 For General Use

Similarly to the development for *shuffle* in the last three subsections we could also proceed for other list producing functions that use *reverse*. In order to increase modularity, we introduce the $vanish_{rev}$-combinator in Figure 4 by generalizing the calculation result in the previous subsection.

```
vanish_rev :: ∀α . (∀β . β → (α → β → β)
                       → (β → β) → β) → [α]
vanish_rev g = fst (g (λys → (ys,ys))
                     (λx h ys → (x : (fst (h ys)),
                                     snd (h (x : ys))))
                     (λh ys → swap (h ys))
                     [])
```

Figure 4. Definition of ***vanish$_{rev}$*****.**

Then, we can replace *shuffle* from the beginning of this section by:

```
shuffle⋆ :: ∀α . [α] → [α]
shuffle⋆ l = vanish_rev (λn c r → let f    []    = n
                                      f (x : xs) = c x (r (f xs))
                                  in  f l)
```

Runtimes of *shuffle* and *shuffle** are compared in the following table:

$n =$	2000	4000	6000	8000	10000
shuffle [1..n]	0.33	1.3	2.8	5.0	8.0
shuffle* [1..n]	0.005	0.01	0.016	0.02	0.025

The development so far relied on the fact that indeed we have $convert \sqsubseteq convert^\star$ (and on the correctness of shortcut deforestation). Instead of proving this auxiliary fact, we directly give a result similar to Theorem 1 that can then be used to transform *shuffle* into *shuffle** without needing the calculations in the previous three subsections. Those calculations and the List data type only appeared for illustrating our general methodology (as summarized in the next subsection) of how to **obtain** optimizing *vanish*-combinators.

THEOREM 2. *For every fixed type* A *and function*

$$g :: \forall\beta\,.\,\beta \to (\mathsf{A} \to \beta \to \beta) \to (\beta \to \beta) \to \beta$$

holds:

$$\boxed{g\ []\ (:)\ reverse \sqsubseteq vanish_{rev}\ g \qquad (19)}$$

The theorem is a consequence of the free theorem for the type of g. Its proof using the laws (2)–(9) from Figure 3 is very similar to — but easier than — the proof of Theorem 3 in Appendix A. That is why we omit it here.

We only note that the approximation $\sqsubseteq$ emerges, because in lazy functional programming languages, where infinite data structures might be present, we have $reverse \circ reverse \sqsubseteq id$ — cf. law (8) — but not $reverse \circ reverse = id$. This is not harmful, because we can still safely optimize $(g\ []\ (:)\ reverse)$ to $(vanish_{rev}\ g)$, which is "at least as defined", whenever this replacement is type-correct.

For our example this implies $shuffle \sqsubseteq shuffle^\star$, and the replacement *shuffle** is not only more efficient, but also has — in certain contexts — a better termination behavior than the original specification of *shuffle*. For example, we have:

$$\begin{array}{c} shuffle\ [1..] = 1 : \bot \\ \sqsubseteq \\ shuffle^\star\ [1..] = 1 : 3 : 5 : 7 : 9 : 11 : \cdots \end{array}$$

4.5 The Methodology Summarized

We informally summarize our methodology of eliminating manipulating operations $f_1,\ldots,f_n$ for some algebraic data type D by the following steps:

1. Extend D to a new data type D' with additional constructors for the manipulating operations $f_1,\ldots,f_n$ and express D-producers as *build*s for this extended data type D', **freezing** $f_1,\ldots,f_n$ as constructors (compare Section 4.1).
2. Invent an **efficient conversion** function $convert^\star_{\mathsf{D}'\to\mathsf{D}}$ from D' to D and express it using a *fold* over D' (compare Section 4.2).
3. Introduce a *vanish*-function that composes D'-producers with $convert^\star_{\mathsf{D}'\to\mathsf{D}}$:
   $$vanish_{f_1,\ldots,f_n}\ g = convert^\star_{\mathsf{D}'\to\mathsf{D}}\ (build_{\mathsf{D}'}\ g)\,,$$
   and use **fusion** of $build_{\mathsf{D}'}$ vs. $fold_{\mathsf{D}'}$ to eliminate the intermediate result of type D' in this definition. Finally, use the **free theorem** of g's type to prove that $vanish_{f_1,\ldots,f_n}$ has the intended semantics — i.e., is suitable as replacement of $build_{\mathsf{D}'}$ for expressing D-producers that use $f_1,\ldots,f_n$ — thus justifying the choice of $convert^\star_{\mathsf{D}'\to\mathsf{D}}$ (compare Sections 4.3 and 4.4).

The key step that requires ingenuity here is of course the **invention** of an efficient conversion function, which naturally depends on the semantics of the involved operations $f_1,\ldots,f_n$. While it is unlikely that one can find a general strategy replacing this creative step, the chance of success is rather high if $f_1,\ldots,f_n$ are related by a rich algebraic theory of equations or approximations. In particular, one should consider such relationships for which one of the two sides of an equation (or the more defined side of an approximation) is more favorable with respect to efficiency. For example, the derivation of $vanish_{++}$ was guided by law (3), whereas the given implementation of the *convert**-function in Section 4.2 was motivated by law (8) in Figure 3. Law (12) plays a similar role for optimizing repeated applications of *map* in the next section.

5 The Concatenate, Reverse and Map Vanish

In this section we present the result of applying our methodology — as summarized in Section 4.5 — to the standard list data type, eliminating the manipulating operations $(+\!\!+)$, *reverse* and *map*[3]. We obtain the — rather scary — definition of $\mathit{vanish}_{+\!+,\mathit{rev},\mathit{map}}$ in Figure 5. However, once we have proved the following theorem, there is no need for the user of our combinator library [1] to be aware of this definition.

THEOREM 3. *For every fixed type* A *and function*

$$\begin{aligned} g :: \forall\beta\,.\,\beta &\to (\mathsf{A} \to \beta \to \beta) \to (\beta \to \beta \to \beta) \\ &\to (\beta \to \beta) \to ((\mathsf{A} \to \mathsf{A}) \to \beta \to \beta) \to \beta \end{aligned}$$

holds:

$$\boxed{g\ []\ (:)\ (+\!\!+)\ \mathit{reverse}\ \mathit{map} \sqsubseteq \mathit{vanish}_{+\!+,\mathit{rev},\mathit{map}}\ g \qquad (20)}$$

The theorem is proved in Appendix A as a direct consequence of the free theorem for g's type, using definitions from Figure 2 and the laws (2)–(15).

We show how the theorem is applied in an example that will also be used for comparison with a list elimination approach by shortcut deforestation for polymorphically recursive workers in Section 7.1.

Example 3. Consider the following specification of a function *inits* that returns the list of initial segments of its argument list — e.g., $\mathit{inits}\ [1..4] = [[],[1],[1,2],[1,2,3],[1,2,3,4]]$ — :

```
inits :: ∀α . [α] → [[α]]
inits []       = [[]]
inits (x : xs) = [] : (map (x :) (inits xs))
```

By abstracting from the list constructors of the **outer** result list — including *map* — and using Theorem 3, *inits* can be replaced by:

```
inits⋆ :: ∀α . [α] → [[α]]
inits⋆ l = vanish_{++,rev,map}
  (λn c a r m → let f []       = [] ‘c‘ n
                    f (x : xs) = [] ‘c‘ (m (x :) (f xs))
                in  f l)
```

Note that abstracting also from the constructors [] and (:) of the **inner** lists would not be type-correct[4].

Runtimes of *inits* and *inits** to compute the same nested lists are compared in the following table:

$n =$	1000	2000	3000	4000	5000
inits $[1..n]$	0.35	1.3	3.2	6.0	9.0
inits* $[1..n]$	0.08	0.3	0.7	1.3	2.0

These measurements show an improvement by a constant factor, because *inits** avoids the repeated mapping of functions of the form $(x\,:)$ over intermediate lists between recursive calls. Inlining the definition of $\mathit{vanish}_{+\!+,\mathit{rev},\mathit{map}}$ would show that instead the initial segments are accumulated in a functional list representation. Since the overall number of (:)-operations required for building the nested output list is still quadratic in the length of the input list, the asymptotic complexity is not changed here. ◇

[3]Only mapping of functions $f :: \mathsf{A} \to \mathsf{A}$ for some type A is considered, because otherwise $\mathit{vanish}_{+\!+,\mathit{rev},\mathit{map}}$ would require polymorphic recursion and a rank-3 type with quantification over type constructors.

[4]Unfortunately, the messages produced by current Haskell implementations in case of insufficient polymorphism are often not very helpful in diagnosing such errors. More advanced type explanation and correction techniques exist — see, e.g., [29, 8, 18] — but not in the presence of higher-ranked polymorphism.

6 The Concatenate Vanishes from Nested Lists

Already Hughes noted that not all instances of his concatenate optimization by representing lists as functions [12] can be achieved through introduction of accumulating parameters. As an example he considers an inefficient function *fields* for breaking up a list of characters into a list of words:

```
fields :: [Char] → [[Char]]
fields s = let f []                     = []
               f (x : xs) | x == ' '    = f xs
                          | otherwise   = g [x] xs
               g w (x : xs) | x /= ' '  = g (w ++ [x]) xs
               g w xs                   = w : (f xs)
           in  f s
```

Also our $\mathit{vanish}_{+\!+,\mathit{rev},\mathit{map}}$-combinator cannot eliminate the repeated concatenations that build up the single words in the result list. The reason is that these concatenations take place on the **inner** lists of *fields*' result, in contrast to the example *inits* from the last section, where the *map*s over the **outer** list where eliminated. Of course, we should not expect a successful optimization here, because $\mathit{vanish}_{+\!+}$ and $\mathit{vanish}_{+\!+,\mathit{rev},\mathit{map}}$ were developed to eliminate function calls to $(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha]$ from a list producer with result type $[\alpha]$ and not from one with result type $[[\alpha]]$.

The realization that we have to deal with this as a **new data type** different from $[\alpha]$ paves the way to success for optimizing also concatenations on the inner lists. Namely, we can apply our methodology as summarized in Section 4.5 to eliminate manipulating operations $(+\!\!+) :: [\alpha] \to [\alpha] \to [\alpha]$ **and** $(+\!\!+) :: [[\alpha]] \to [[\alpha]] \to [[\alpha]]$ from producers of nested lists of the algebraic data type $[[\alpha]]$.

Therefor, we simply apply the trick of replacing lists by functions twice, once on the inner lists and once on the outer list. The outcome is the combinator $\mathit{vanish}_{+\!+,+\!+}$ in Figure 6, with semantics as given by the theorem below.

```
vanish_{++,++} :: ∀α . (∀β γ . β → (α → β → β)
                              → (β → β → β)
                              → γ → (β → γ → γ)
                              → (γ → γ → γ) → γ) → [[α]]
vanish_{++,++} g = g id (λx h ys → x : (h ys)) (◦)
                     id (λh hs yss → (h []) : (hs yss)) (◦) []
```

Figure 6. Definition of $\mathit{vanish}_{+\!+,+\!+}$**.**

THEOREM 4. *For every fixed type* A *and function*

$$\begin{aligned} g :: \forall\beta\,\gamma\,.\,\beta &\to (\mathsf{A} \to \beta \to \beta) \to (\beta \to \beta \to \beta) \\ &\to \gamma \to (\beta \to \gamma \to \gamma) \to (\gamma \to \gamma \to \gamma) \to \gamma \end{aligned}$$

holds:

$$\boxed{g\ []\ (:)\ (+\!\!+)\ []\ (:)\ (+\!\!+) = \mathit{vanish}_{+\!+,+\!+}\ g \qquad (21)}$$

The proof of this theorem — using the free theorem of g's type and the laws (1)–(4) from Figure 3 — is only slightly more difficult than the proof of Theorem 1, but is omitted here due to space constraints.

```
vanish_{++,rev,map} :: ∀α . (∀β . β → (α → β → β) → (β → β → β) → (β → β) → ((α → α) → β → β) → β) → [α]
vanish_{++,rev,map} g = fst (g (λf ys → (ys,ys)) (λx h f ys → ((f x) : (fst (h f ys)), snd (h f ((f x) : ys))))
                               (λh_1 h_2 f ys → (fst (h_1 f (fst (h_2 f ys))), snd (h_2 f (snd (h_1 f ys)))))
                               (λh f ys → swap (h f ys)) (λk h f ys → h (f ∘ k) ys) id [])
```

Figure 5. Definition of $vanish_{++,rev,map}$**.**

In order to show how $vanish_{++,++}$ eliminates concatenate operations from producers of nested lists, we apply it to *fields*.

Example 4. Consider the function definition of *fields* from above. Abstracting from the list constructors [], (:) and (++) **type-correctly** and using law (21) from Theorem 4, this definition is equivalent to the much more efficient:

```
fields⋆ :: [Char] → [[Char]]
fields⋆ s = vanish_{++,++} (λn c a nn cc aa →
  let f     []             = nn
      f (x : xs) | x == ' '   = f xs
                 | otherwise  = g (x ‘c‘ n) xs
      g w (x : xs) | x /= ' ' = g (w ‘a‘ (x ‘c‘ n)) xs
      g w    xs               = w ‘cc‘ (f xs)
  in f s)
```

By inlining the definition of $vanish_{++,++}$ and performing some beta-reductions and let-floating, we obtain the variant

```
fields⋆ s =
  let f     []             = id
      f (x : xs) | x == ' '   = f xs
                 | otherwise  = g (λys → x : ys) xs
      g w (x : xs) | x /= ' ' = g (w ∘ (λys → x : ys)) xs
      g w    xs               = λyss → (w []) : (f xs yss)
  in f s []
```

which differs from Hughes' result only in that the functional representation is also used for production of the outer list. ◇

7 Comparison with Related Work

7.1 Shortcut Deforestation

Gill [9] gives an example of a recursive function consuming its own result via *foldr* and shows how a worker/wrapper scheme of shortcut deforestation can improve the asymptotic time complexity of this function by removing intermediate lists between recursive calls with law (17). In a similar way, Chitil [7] uses the representation of *map* as a *foldr* to transform the specification of *inits* from Example 3 into essentially the following:

```
inits′ :: ∀α . [α] → [[α]]
inits′ l =
  let f c n    []    = [] ‘c‘ n
      f c n (x : xs) = [] ‘c‘ (f (λys yss → (x : ys) ‘c‘ yss) n xs)
  in f (:) [] l
```

This is achieved by firstly splitting *inits* into a wrapper and a worker that uses polymorphic recursion, and then applying shortcut deforestation inside the worker.

While there is no obvious connection between *inits*⋆ and *inits*′, their runtimes are comparable:

$n =$	1000	2000	3000	4000	5000
inits⋆ [1..*n*]	0.08	0.3	0.7	1.3	2.0
inits′ [1..*n*]	0.07	0.3	0.8	1.2	2.0

However, *inits*⋆ leaves room for further improvement by using a more specialized $vanish_{map}$-combinator instead of the function $vanish_{++,rev,map}$. Note that such a switch from using one *vanish*-combinator to using a more general or a more specific one is usually easy, in this case by dropping *a* and *r* from the list of λ-abstracted variables in the definition of *inits*⋆.

On the other hand, Chitil's method does not work for functions like, e.g., *part* and *fields*. Also, he discourages the use of his own approach (to **automatically** obtain *inits*′ by type inference based deforestation) in the following words:

> "This power is, however, a double-edged sword. A small syntactic change of a program […] 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."

Our approach can provide the programmer with exactly this control. Moreover, it solves the problem in Section 3.4.3 of [7], because our method can optimize the *foldr*-form of the naïve list reverse:

```
rev :: ∀α . [α] → [α]
rev l = foldr (λx h → h ++ [x]) [] l
```

by transforming it into:

```
rev⋆ :: ∀α . [α] → [α]
rev⋆ l = vanish_{++} (λn c a → foldr (λx h → h ‘a‘ (x ‘c‘ n)) n l)
       = foldr (λx h ys → h (x : ys)) id l []
```

Thus, our method has eliminated the inefficient concatenations without destroying the possibility of using shortcut deforestation with *rev*⋆ as list consumer.

Of course, our technique does not supersede shortcut deforestation, because law (17) is also applicable to remove intermediate lists in a variety of cases that are not addressed by our approach of eliminating particular data manipulating operations. Rather, the two techniques complement each other quite well. Note however, that programs using the *vanish*-combinators benefit from our optimization method without requiring any additional support from the compiler, whereas optimization by shortcut deforestation crucially depends on a compiler that makes some nontrivial effort to apply law (17) — or an equivalent of it — at as many places as possible.

Svenningsson [26] proposes a new *destroy*/*unfoldr*-rule that gives rise to further possibilities of intermediate list removal and is in many respects a dual of law (17). Since our *vanish*-combinators vary the idea of shortcut deforestation — by abstracting not only over data constructors, but also over other data manipulating operations — it might be interesting to investigate whether a dual variation of the *destroy*/*unfoldr*-technique also exists.

7.2 Accumulating Parameters

Wadler [27] presented a transformation that eliminates concatenate operations by introducing accumulating parameters. For example, the function *hanoi* from Example 1 is improved to *hanoi*$'$ by generalizing the local function f to f^{++} with an additional list argument and applying a set of rewrite laws to the right-hand sides of a naïve definition of f^{++}, finally yielding:

$$\begin{array}{l} hanoi' :: \mathsf{Int} \to [(\mathsf{Pos},\mathsf{Pos})] \\ hanoi'\ t = \\ \quad \textbf{let}\ f^{++}\ \ 1\ \ \ \ p\ q\ r\ ys = (p,q) : ys \\ \quad\quad\ \ f^{++}\ (s+1)\ p\ q\ r\ ys = f^{++}\ s\ p\ r\ q\ ((p,q) : (f^{++}\ s\ r\ q\ p\ ys)) \\ \quad \textbf{in}\ \ f^{++}\ t\ \mathsf{L}\ \mathsf{R}\ \mathsf{M}\ [] \end{array}$$

The success of this method depends on the characterization of functions as *creative*. For example, an attempted application to the introductory example *part* produces the following definition:

$$\begin{array}{l} part' :: \forall\alpha\,.\,(\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\ part'\ p\ l = \textbf{let}\ f^{++}\ \ []\ \ \ \ z\ ys = z +\!\!+ ys \\ \qquad\qquad\ \ f^{++}\ (x:xs)\ z\ ys = \textbf{if}\ p\ x\ \textbf{then}\ x : (f^{++}\ xs\ z\ ys) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ f^{++}\ xs\ (z +\!\!+ [x])\ ys \\ \qquad\quad \textbf{in}\ \ f^{++}\ l\ []\ [] \end{array}$$

Note that here the concatenate does **not** vanish. The characterization of *part* as *plagiarizing* comes somewhat as a surprise, because *part* clearly **does** allocate each cons cell in its result and thus should be manageable — and is so with our new method.

On the other hand, it is easy to see that all functions fulfilling Wadler's syntactic test for creativity also allow the type-correct abstraction from list constructors $[]$, $(:)$ and $(+\!\!+)$, and can thus also be transformed by our technique. Hence, $vanish_{++}$ captures the introduction of an accumulating parameter as can be seen by comparing the above function *hanoi*$'$ with the final definition obtained for *hanoi** in Example 1.

Albert *et al.* [3] achieve the effects of Wadler's transformation for functional logic programs by temporarily introducing difference-lists that are then replaced by accumulating parameters. Their technique fails for the same examples as Wadler's, hence it would be interesting to study our method also in the more general framework of functional logic programming languages.

Kühnemann *et al.* [15] generalize list concatenation to *tree substitution functions* and eliminate such substitution functions from *top-down tree transducer modules* by introducing accumulating parameters. Their "integration step" could also be realized in the framework of our methodology, thus generalizing their accumulation technique to arbitrary functions that produce trees.

Bird [4], Hu *et al.* [11] use *calculational methods* to derive accumulative programs — in their setting higher-order *fold*s over algebraic data types — from first-order *fold*s, such as transforming *rev* from the previous subsection into *rev** (which none of the other accumulation methods discussed here achieves). Their techniques have a wider scope in deriving new algorithms rather than "just" eliminating inefficient function calls. On the other hand, they can only handle programs expressed with a fixed set of recursion operators, and must — of course — fail where introduction of accumulating parameters cannot remove the inefficiencies. Further approaches for deriving accumulative algorithms are compared by Boiten [5].

7.3 Abstract Data Types

Our steps of "freezing" and "efficient conversion" have similar goals as the transition from a *term representation* to a *context-passing representation* in Hughes' methodology for implementing domain-specific languages [13]. In fact, one might conceive, e.g., our developments in Sections 4.1–4.3 as the construction of yet another *novel representation of lists* in the sense of [12], but this time with support for efficient reversal instead of concatenation. Hence, the described methodology is another way to synthesize efficient implementations of abstract data types from the knowledge about algebraic properties of the involved operations. While Hughes uses this knowledge to guide the choice of the contexts from which to abstract, we employ it in the definition of *convert**-functions.

The rank-2 polymorphic *vanish*-combinators encode such efficient implementations by introducing a form of **anonymous** abstract data types. This local encapsulation limits cross-function optimization, but is the key to proving the correctness of transformations by free theorems. In particular, there are no analogues in Hughes' methodology to our Theorems 1–4 that embody each optimization in a single rule. It is unclear how other encodings of abstract data types could be employed semantically to do such proofs or even to formalize the essence of transformations in such a concise way. Also, the expression of optimizations by uniform abstraction from data constructors and manipulating functions paves the way to automation using Chitil's approach of data abstraction by type inference.

8 Conclusion

In this paper we developed a combinator library to optimize list producers involving the operations $+\!\!+$, *reverse* and *map*. The introduced methodology is also applicable to eliminate other list manipulations — e.g., the well-known *filter*-function [1] — and for nested lists and other algebraic data types.

We would like to further investigate the interplay between our technique and the elimination of intermediate results by shortcut deforestation. As seen in Section 7.1, a "good" consumer (one consuming its argument uniformly with *foldr*) remains "good" after applying our technique. Moreover, we can straightforwardly express the *vanish*-functions in *build*-form, for example:

$$vanish_{++}\ g = build\ (\lambda c\ n \to g\ id\ (\lambda x\ h\ ys \to x\,\text{`}c\text{`}\,(h\ ys))\ (\circ)\ n)$$

Then, every function expressed with $vanish_{++}$ is a "good" producer (one producing its result uniformly with *build*). For example, *part** becomes a "good" producer, while *part* was none. Furthermore, one can dualize Chitil's approach of "deforesting functions that consume their own result with *foldr*" to "deforesting functions that construct their own argument with *build*", which is then also applicable to the examples *part* and *fields*.

We have observed various examples where our method leads to dramatic efficiency improvements. However, a general statement about the relation between the runtimes of original and transformed programs is hard to make. Moran & Sands [19] argue that Wadler's accumulation technique will never degrade efficiency by more than a constant factor. We believe that the same is true for our more general technique of concatenate elimination and would like to investigate this claim formally. For computing the asymptotic time complexity of transformed programs that use $vanish_{++,rev,map}$, the runtime for performing the original $+\!\!+$-, *reverse*- and *map*-operations can be considered as constant rather than linear in the lengths of the lists they consume. In other applications of our methodology the efficiency of course depends on the quality of the chosen implementation for *convert**.

9 Acknowledgments

I want to thank Claus Jürgensen and Armin Kühnemann for comments and useful suggestions on drafts of this paper. Discussions with Olaf Chitil and Josef Svenningsson and remarks by the ICFP

referees also provided interesting insights and inspiration for future research.

10 References

[1] `http://wwwtcs.inf.tu-dresden.de/~voigt/Vanish.lhs`.

[2] The Haskell 98 Report. `http://haskell.org/onlinereport`.

[3] E. Albert, C. Ferri, F. Steiner, and G. Vidal. Improving functional logic programs by difference-lists. In *Advances in Computing Science, Penang, Malaysia, Proceedings*, volume 1961 of *LNCS*, pages 237–254. Springer-Verlag, 2000.

[4] R. Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. on Prog. Lang. and Systems*, 6:487–504, 1984. Addendum *Ibid.*, 7:490–492, 1985.

[5] E. Boiten. The many disguises of accumulation. Technical Report 91-26, Dept. of Informatics, University of Nijmegen, 1991.

[6] O. Chitil. Type inference builds a short cut to deforestation. In *International Conference on Functional Programming, Paris, France, Proceedings*, pages 249–260. ACM Press, 1999.

[7] O. Chitil. *Type-Inference Based Deforestation of Functional Programs*. PhD thesis, RWTH Aachen, 2000.

[8] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *International Conference on Functional Programming, Florence, Italy, Proceedings*, pages 193–204. ACM Press, 2001.

[9] A. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1996.

[10] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, Proceedings*, pages 223–232. ACM Press, 1993.

[11] Z. Hu, H. Iwasaki, and M. Takeichi. Calculating accumulations. *New Generation Computing*, 17:153–173, 1999.

[12] J. Hughes. A novel representation of lists and its application to the function “reverse”. *Information Processing Letters*, 22:141–144, 1986.

[13] J. Hughes. The design of a pretty-printing library. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer-Verlag, 1995.

[14] P. Johann. Short cut fusion: Proved and improved. In *Semantics, Applications, and Implementation of Program Generation, Florence, Italy, Proceedings*, volume 2196 of *LNCS*, pages 47–71. Springer-Verlag, 2001.

[15] A. Kühnemann, R. Glück, and K. Kakehi. Relating accumulative and non-accumulative functional programs. In *Rewriting Techniques and Applications, Utrecht, The Netherlands, Proceedings*, volume 2051 of *LNCS*, pages 154–168. Springer-Verlag, 2001.

[16] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *European Symposium on Programming, Linköping, Sweden, Proceedings*, volume 1058 of *LNCS*, pages 204–218. Springer-Verlag, 1996.

[17] D. Leivant. Polymorphic type inference. In *Principles of Programming Languages, Austin, Texas, Proceedings*, pages 88–98. ACM Press, 1983.

[18] B. McAdam. *Repairing Type Errors in Functional Programs*. PhD thesis, University of Edinburgh, 2002.

[19] A. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Principles of Programming Languages, San Antonio, Texas, Proceedings*, pages 43–56. ACM Press, 1999.

[20] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: Moving bindings to give faster programs. In *International Conference on Functional Programming, Philadelphia, Pennsylvania, Proceedings*, pages 1–12. ACM Press, 1996.

[21] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Sci. of Comput. Prog.*, 32:3–47, 1998.

[22] A. Pitts. Parametric polymorphism and operational equivalence. *Math. Struct. Comput. Sci.*, 10:321–359, 2000.

[23] J. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Paris, France, Proceedings*, pages 513–523. Elsevier Science Publishers B.V., 1983.

[24] D. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[25] T. Sheard, Z. Benaissa, and M. Martel. Introduction to multistage programming using MetaML. `http://cse.ogi.edu/~sheard/papers/manual.ps`.

[26] J. Svenningsson. Shortcut fusion for accumulating parameters and zip-like functions. In *International Conference on Functional Programming, Pittsburgh, Pennsylvania, Proceedings*. ACM Press, 2002.

[27] P. Wadler. The concatenate vanishes. Note, University of Glasgow, 1987 (revised, 1989).

[28] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, London, England, Proceedings*, pages 347–359. ACM Press, 1989.

[29] J. Yang, G. Michaelson, P. Trinder, and J. Wells. Improved type error reporting. In *Implementation of Functional Languages, Aachen, Germany, Draft Proceedings*, pages 71–86, 2000.

A Proof of Theorem 3

PROOF. The free theorem associated with g's type is that for every choice of types B and B', values $n :: \mathsf{B}$, $n' :: \mathsf{B}'$, $c :: \mathsf{A} \to \mathsf{B} \to \mathsf{B}$, $c' :: \mathsf{A} \to \mathsf{B}' \to \mathsf{B}'$, $a :: \mathsf{B} \to \mathsf{B} \to \mathsf{B}$, $a' :: \mathsf{B}' \to \mathsf{B}' \to \mathsf{B}'$, $r :: \mathsf{B} \to \mathsf{B}$, $r' :: \mathsf{B}' \to \mathsf{B}'$, $m :: (\mathsf{A} \to \mathsf{A}) \to \mathsf{B} \to \mathsf{B}$ and $m' :: (\mathsf{A} \to \mathsf{A}) \to \mathsf{B}' \to \mathsf{B}'$, and an admissible relation $\mathcal{R} \subseteq \mathsf{B} \times \mathsf{B}'$, the following holds:

$$\begin{array}{ll}
 & (n,n') \in \mathcal{R} \\
\wedge & (\forall x :: \mathsf{A}, (l,l') \in \mathcal{R} \,.\, (c\,x\,l, c'\,x\,l') \in \mathcal{R}) \\
\wedge & (\forall (l_1,l_1') \in \mathcal{R}, (l_2,l_2') \in \mathcal{R} \,.\, (a\,l_1\,l_2, a'\,l_1'\,l_2') \in \mathcal{R}) \\
\wedge & (\forall (l,l') \in \mathcal{R} \,.\, (r\,l, r'\,l') \in \mathcal{R}) \\
\wedge & (\forall k :: \mathsf{A} \to \mathsf{A}, (l,l') \in \mathcal{R} \,.\, (m\,k\,l, m'\,k\,l') \in \mathcal{R}) \\
\Rightarrow & (g\,n\,c\,a\,r\,m, g\,n'\,c'\,a'\,r'\,m') \in \mathcal{R}.
\end{array}$$

Instantiating this with $\mathsf{B} = [\mathsf{A}]$, $n = []$, $c = (:)$, $a = (+\!\!+)$, $r = \mathit{reverse}$, $m = \mathit{map}$, $\mathsf{B}' = (\mathsf{A} \to \mathsf{A}) \to [\mathsf{A}] \to ([\mathsf{A}],[\mathsf{A}])$ and:

$$\begin{array}{lcl}
n' & = & (\lambda f\,ys \to (ys,ys)) \\
c' & = & (\lambda x\,h\,f\,ys \to (((f\,x) : (\mathit{fst}\,(h\,f\,ys)), \mathit{snd}\,(h\,f\,(((f\,x) : ys)))) \\
a' & = & (\lambda h_1\,h_2\,f\,ys \to (\mathit{fst}\,(h_1\,f\,(\mathit{fst}\,(h_2\,f\,ys))), \\
 & & \qquad\qquad \mathit{snd}\,(h_2\,f\,(\mathit{snd}\,(h_1\,f\,ys))))) \\
r' & = & (\lambda h\,f\,ys \to \mathit{swap}\,(h\,f\,ys)) \\
m' & = & (\lambda k\,h\,f\,ys \to h\,(f \circ k)\,ys),
\end{array}$$

we obtain (modulo beta-equality):

$$
\begin{array}{ll}
& ([], (\lambda f\, ys \rightarrow (ys, ys))) \in \mathcal{R} \\
\wedge & (\forall x :: \mathsf{A}, (l, l') \in \mathcal{R}\, . \\
& \quad (x : l, (\lambda f\, ys \rightarrow ((f\, x) : (fst\, (l'\, f\, ys)), \\
& \qquad\qquad snd\, (l'\, f\, ((f\, x) : ys))))) \in \mathcal{R}) \\
\wedge & (\forall (l_1, l'_1) \in \mathcal{R}, (l_2, l'_2) \in \mathcal{R}\, . \\
& \quad (l_1 \mathbin{+\!\!+} l_2, (\lambda f\, ys \rightarrow (fst\, (l'_1\, f\, (fst\, (l'_2\, f\, ys))), \\
& \qquad\qquad snd\, (l'_2\, f\, (snd\, (l'_1\, f\, ys)))))) \in \mathcal{R}) \\
\wedge & (\forall (l, l') \in \mathcal{R}\, .\, (reverse\, l, (\lambda f\, ys \rightarrow swap\, (l'\, f\, ys))) \in \mathcal{R}) \\
\wedge & (\forall k :: \mathsf{A} \rightarrow \mathsf{A}, (l, l') \in \mathcal{R}\, .\, (map\, k\, l, (\lambda f\, ys \rightarrow l'\, (f \circ k)\, ys)) \in \mathcal{R}) \\
\Rightarrow & (g\, []\, (:)\, (\mathbin{+\!\!+})\, reverse\, map, g\, n'\, c'\, a'\, r'\, m') \in \mathcal{R}.
\end{array}
$$

If we choose the admissible relation[5]

$$
\begin{array}{rll}
\mathcal{R} = \{(p, q) \mid \forall f :: \mathsf{A} \rightarrow \mathsf{A}, u :: [\mathsf{A}]\, . & & \\
& (map\, f\, p) \mathbin{+\!\!+} u & \sqsubseteq fst\, (q\, f\, u) \\
\wedge & (map\, f\, (reverse\, p)) \mathbin{+\!\!+} u & \sqsubseteq snd\, (q\, f\, u)\}\, ,
\end{array}
$$

then the conjuncts in the precondition of this implication read as follows (modulo some beta-reductions and the definitions of *fst* and *snd*):

(i). $\forall f :: \mathsf{A} \rightarrow \mathsf{A}, u :: [\mathsf{A}]\, . \quad (map\, f\, []) \mathbin{+\!\!+} u \sqsubseteq u \;\wedge\; (map\, f\, (reverse\, [])) \mathbin{+\!\!+} u \sqsubseteq u$

(ii). for every $x :: \mathsf{A}$, $l :: \mathsf{B}$ and $l' :: \mathsf{B}'$:

$$
\begin{array}{ll}
& (\forall f :: \mathsf{A} \rightarrow \mathsf{A}, v :: [\mathsf{A}]\, . \\
& \quad (map\, f\, l) \mathbin{+\!\!+} v \sqsubseteq fst\, (l'\, f\, v) \\
& \quad \wedge\; (map\, f\, (reverse\, l)) \mathbin{+\!\!+} v \sqsubseteq snd\, (l'\, f\, v)) \\
\Rightarrow & (\forall f :: \mathsf{A} \rightarrow \mathsf{A}, u :: [\mathsf{A}]\, . \\
& \quad (map\, f\, (x : l)) \mathbin{+\!\!+} u \sqsubseteq (f\, x) : (fst\, (l'\, f\, u)) \\
& \quad \wedge\; (map\, f\, (reverse\, (x : l))) \mathbin{+\!\!+} u \sqsubseteq snd\, (l'\, f\, ((f\, x) : u)))
\end{array}
$$

(iii). for every $l_1, l_2 :: \mathsf{B}$ and $l'_1, l'_2 :: \mathsf{B}'$:

$$
\begin{array}{ll}
& (\forall f :: \mathsf{A} \rightarrow \mathsf{A}, v :: [\mathsf{A}]\, . \\
& \quad (map\, f\, l_1) \mathbin{+\!\!+} v \sqsubseteq fst\, (l'_1\, f\, v) \\
& \quad \wedge\; (map\, f\, (reverse\, l_1)) \mathbin{+\!\!+} v \sqsubseteq snd\, (l'_1\, f\, v) \\
& \quad \wedge\; (map\, f\, l_2) \mathbin{+\!\!+} v \sqsubseteq fst\, (l'_2\, f\, v) \\
& \quad \wedge\; (map\, f\, (reverse\, l_2)) \mathbin{+\!\!+} v \sqsubseteq snd\, (l'_2\, f\, v)) \\
\Rightarrow & (\forall f :: \mathsf{A} \rightarrow \mathsf{A}, u :: [\mathsf{A}]\, . \\
& \qquad (map\, f\, (l_1 \mathbin{+\!\!+} l_2)) \mathbin{+\!\!+} u \\
& \quad \sqsubseteq fst\, (l'_1\, f\, (fst\, (l'_2\, f\, u))) \\
& \wedge \qquad (map\, f\, (reverse\, (l_1 \mathbin{+\!\!+} l_2))) \mathbin{+\!\!+} u \\
& \quad \sqsubseteq snd\, (l'_2\, f\, (snd\, (l'_1\, f\, u))))
\end{array}
$$

(iv). for every $l :: \mathsf{B}$ and $l' :: \mathsf{B}'$:

$$
\begin{array}{ll}
& (\forall f :: \mathsf{A} \rightarrow \mathsf{A}, v :: [\mathsf{A}]\, . \\
& \quad (map\, f\, l) \mathbin{+\!\!+} v \sqsubseteq fst\, (l'\, f\, v) \\
& \quad \wedge\; (map\, f\, (reverse\, l)) \mathbin{+\!\!+} v \sqsubseteq snd\, (l'\, f\, v)) \\
\Rightarrow & (\forall f :: \mathsf{A} \rightarrow \mathsf{A}, u :: [\mathsf{A}]\, . \\
& \qquad (map\, f\, (reverse\, l)) \mathbin{+\!\!+} u \\
& \quad \sqsubseteq fst\, (swap\, (l'\, f\, u)) \\
& \wedge \qquad (map\, f\, (reverse\, (reverse\, l))) \mathbin{+\!\!+} u \\
& \quad \sqsubseteq snd\, (swap\, (l'\, f\, u)))
\end{array}
$$

[5]Again, it is easy to see that $(\bot, \bot) \in \mathcal{R}$ by laws (7), (14) and (2). Continuity of $\mathcal{R}$ is shown by using the facts that the relation $\sqsubseteq$ is continuous and that the involved Haskell-functions are monotonic and continuous.

(v). for every $k :: \mathsf{A} \rightarrow \mathsf{A}$, $l :: \mathsf{B}$ and $l' :: \mathsf{B}'$:

$$
\begin{array}{ll}
& (\forall f' :: \mathsf{A} \rightarrow \mathsf{A}, v :: [\mathsf{A}]\, . \\
& \quad (map\, f'\, l) \mathbin{+\!\!+} v \sqsubseteq fst\, (l'\, f'\, v) \\
& \quad \wedge\; (map\, f'\, (reverse\, l)) \mathbin{+\!\!+} v \sqsubseteq snd\, (l'\, f'\, v)) \\
\Rightarrow & (\forall f :: \mathsf{A} \rightarrow \mathsf{A}, u :: [\mathsf{A}]\, . \\
& \quad (map\, f\, (map\, k\, l)) \mathbin{+\!\!+} u \sqsubseteq fst\, (l'\, (f \circ k)\, u) \\
& \quad \wedge\; (map\, f\, (reverse\, (map\, k\, l))) \mathbin{+\!\!+} u \sqsubseteq snd\, (l'\, (f \circ k)\, u)).
\end{array}
$$

Using reflexivity and transitivity of $\sqsubseteq$ and monotonicity of the involved Haskell functions, these five conditions can be established as follows:

(i). by the definitions of *map*, $(\mathbin{+\!\!+})$ and *reverse*

(ii). by the definitions of *map* and $(\mathbin{+\!\!+})$ and the laws (9), (11) and (3)

(iii). by the laws (11), (3) and (10)

(iv). by the laws (5), (8) and (6)

(v). by the laws (12) and (13).

As consequence of the instantiated free theorem we thus obtain:

$$(g\, []\, (:)\, (\mathbin{+\!\!+})\, reverse\, map, g\, n'\, c'\, a'\, r'\, m') \in \mathcal{R}.$$

Using our definition of $\mathcal{R}$, from this follows

$$
\begin{array}{ll}
\forall f :: \mathsf{A} \rightarrow \mathsf{A}, u :: [\mathsf{A}]\, . & (map\, f\, (g\, []\, (:)\, (\mathbin{+\!\!+})\, reverse\, map)) \mathbin{+\!\!+} u \\
& \sqsubseteq fst\, (g\, n'\, c'\, a'\, r'\, m'\, f\, u)\, ,
\end{array}
$$

which for $f = id$ and $u = []$ implies — using laws (4) and (15) and the definition of $vanish_{\mathbin{+\!\!+},rev,map}$ — the following calculation:

$$
\begin{array}{ll}
& g\, []\, (:)\, (\mathbin{+\!\!+})\, reverse\, map \\
= & (map\, id\, (g\, []\, (:)\, (\mathbin{+\!\!+})\, reverse\, map)) \mathbin{+\!\!+} [] \\
\sqsubseteq & fst\, (g\, n'\, c'\, a'\, r'\, m'\, id\, []) \\
= & vanish_{\mathbin{+\!\!+},rev,map}\, g.
\end{array}
$$

□

B Introducing Strict Evaluation

The equality stated in law (18) from Theorem 1 does not necessarily hold as such if g makes use of the strict evaluation primitive $seq :: \forall \alpha\, \beta\, .\, \alpha \rightarrow \beta \rightarrow \beta$, defined in Haskell 98 by the following equations:

$$
\begin{array}{l}
seq\, \bot\, b = \bot \\
seq\, a\, b = b,\ if\ a \neq \bot
\end{array}
$$

The reason is that for the relation $\mathcal{R}$ used in the proof we have $(\bot, \lambda ys \rightarrow \bot) \in \mathcal{R}$, but from $(p, q) \in \mathcal{R}$ does not necessarily follow $(seq\, \bot\, p, seq\, (\lambda ys \rightarrow \bot)\, q) \in \mathcal{R}$, because *seq* distinguishes between $\bot$ and $(\lambda ys \rightarrow \bot)$. Hence, *seq* does not map $\mathcal{R}$-related arguments to $\mathcal{R}$-related results (as it should if we want to exploit parametricity), and thus $\mathcal{R}$ is not suitable for usage in the free theorem of g's type if g may be defined using *seq*.

This problem can be fixed with an analogous strategy as used in Section 7 of [28] to enrich the functional language with the fixpoint primitive while preserving parametricity, namely by imposing further requirements on the relations used in free theorems. That is to say, we must restrict ourselves to relations that respect *seq*, in the sense that for such relations $\mathcal{A}$ and $\mathcal{B}$ it must follow from $(a, a') \in \mathcal{A}$ and $(b, b') \in \mathcal{B}$ that also $(seq\, a\, b, seq\, a'\, b') \in \mathcal{B}$. This can be guaranteed if every relation $\mathcal{X} \subseteq \mathsf{X} \times \mathsf{X}'$ used in the proof contains all pairs $(\bot, x')$ with $x' :: \mathsf{X}'$, but no pair $(x, \bot)$ with $x \neq \bot$.

Hence, we extend the relational interpretation of base types — and also of the fixed type A — from identity relations to the partial order $\sqsubseteq$ and enlarge $\mathcal{R}$ to

$$\mathcal{R}' = \{(p,q) \in \mathsf{B} \times \mathsf{B}' \mid \forall u :: [\mathsf{A}] \,.\, p +\!\!+ u \sqsubseteq q\, u\},$$

where admissibility of $\mathcal{R}'$ follows from admissibility of $\sqsubseteq$ and monotonicity and continuity of $(+\!\!+)$.

The instantiated free theorem then reads as follows:

$$\begin{array}{ll} & ([], id) \in \mathcal{R}' \\ \wedge & (\forall x, x' :: \mathsf{A}, (l, l') \in \mathcal{R}' \,. \\ & \quad x \sqsubseteq x' \Rightarrow (x : l, (\lambda x\, h\, ys \to x : (h\, ys))\, x'\, l') \in \mathcal{R}') \\ \wedge & (\forall (l_1, l'_1) \in \mathcal{R}', (l_2, l'_2) \in \mathcal{R}' \,.\, (l_1 +\!\!+ l_2, l'_1 \circ l'_2) \in \mathcal{R}') \\ \Rightarrow & (g\, []\, (:)\, (+\!\!+), g\, id\, (\lambda x\, h\, ys \to x : (h\, ys))\, (\circ)) \in \mathcal{R}'. \end{array}$$

With similar reasoning as in the proof of Theorem 1 — additionally using reflexivity and transitivity of $\sqsubseteq$ and monotonicity of $(:)$ and $(+\!\!+)$ — we can validate the three conjuncts in the precondition and obtain from the implied consequence the following replacement for law (18):

$$\boxed{g\, []\, (:)\, (+\!\!+) \sqsubseteq \mathit{vanish}_{+\!\!+}\, g \qquad (22)}$$

Likewise, the proof of Theorem 4 can be adapted to give in the presence of *seq*:

$$\boxed{g\, []\, (:)\, (+\!\!+)\, []\, (:)\, (+\!\!+) \sqsubseteq \mathit{vanish}_{+\!\!+,+\!\!+}\, g \qquad (23)}$$

Hence, our concatenate elimination is still applicable, it just becomes an approximation — being at least as defined, but potentially improving termination behavior — instead of a semantic equality. Note that this is already the case for Hughes' technique [12], because of:

$$\mathit{seq} \perp b = \perp \sqsubseteq b = \mathit{seq}\, (\lambda ys \to \perp +\!\!+ ys)\, b = \mathit{seq}\, (\mathit{rep} \perp)\, b.$$

The statements of Theorems 2 and 3 are not affected by the presence of *seq*, because their proofs already use relations $\mathcal{R}$ that fulfill the above restrictions and it is straightforward to adapt those proofs to use the partial order $\sqsubseteq$ instead of identity as relational interpretation for base types and for A.

Thus, our situation is much better than that of shortcut deforestation, which in the presence of unrestricted use of *seq* can transform terminating programs into non-terminating ones, for example:

$$\mathit{foldr} \perp []\, (\mathit{build}\ \mathit{seq}) = \mathit{foldr} \perp []\, (\mathit{seq}\, (:)\, []) = \mathit{foldr} \perp []\, [] = [],$$

but after application of law (17) to $\mathit{foldr} \perp []\, (\mathit{build}\ \mathit{seq})$:

$$\mathit{seq} \perp [] = \perp.$$

This flaw of shortcut deforestation may be remedied using more restricted qualified types for *build* or *seq*, similarly to the strategy of Launchbury & Paterson [16] for avoiding strictness side conditions on free theorems. This approach might also be applicable in our setting to control the use of *seq* and thus preserve laws (18) and (21) instead of (22) and (23).

While right in spirit, the discussion in this appendix does not cover all the subtleties that the presence of *seq* entails for proofs based on free theorems. For a correct treatment, consult the following papers:

P. Johann and J. Voigtländer. Free theorems in the presence of *seq*. In *Principles of Programming Languages, Venice, Italy, Proceedings*, volume 39(1) of SIGPLAN Notices, pages 99–110. ACM Press, 2004.

P. Johann and J. Voigtländer The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69:63–102, 2006.