# Explicit Binds: Effortless Efficiency with and without Trees

Tarmo Uustalu

Institute of Cybernetics, Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia
tarmo@cs.ioc.ee

**Abstract.** We demonstrate a simple and robust program transformation technique that can improve asymptotic time complexity of data-manipulating programs (e.g., produce a linear-time list reversal function from the obvious quadratic one). In the version of the present paper, it applies to monadic inductive datatypes and can be stated in two flavors, through a datatype representation, with an explicit ("frozen") bind constructor and a special associated defining clause for the fold function, and in a functional form (generalized Church numerals), with a special definition of the bind function in terms of the build constructor. The technique explicates, systematizes, combines and scales a number of ideas known from the literature, achieving a new level of generality.

## 1 Introduction

This is a functional programming and program transformation paper, following up work on shortcut deforestation [8], the worker/wrapper transformation [20] and similar strands of program transformation research. Our objective is to find ways to improve the efficiency of datastructure-manipulating functions without having to give up their natural definitions. We want to be able to do this robustly and avoiding ad-hoc work.

This may sound impossible. But we present a simple technique whereby those functions on some datatype that make extensive use of some operation important for that datatype may become drastically more efficient thanks to nothing more than moving to a slightly non-standard implementation of the datatype. Moreover, the suitable non-standard implementation is found in a systematic way.

We begin with the motivating example of the datatype of lists. Here we are concerned with functions such as reverse or quicksort that repeatedly append lists. Append is the multiplication operation of the monoid of lists. Efficiency boosts are achieved by switching from "true" lists to lists with explicit or "frozen" appends (an implementation of lists where append is a constructor) and with Church lists (for those we carefully define append as a build).

We then proceed to other datatypes, which in this paper are all monadic inductive types, and use the same ideas. We either work with an augmented inductive datatype with explicit binds or with a Church encoding where we define

T. Schrijvers and P. Thiemann (Eds.): FLOPS 2012, LNCS 7294, pp. 317–331, 2012.

bind as a build.[1] Finally, we briefly comment on some different generalizations, such as functorial inductive datatypes with explicit map operations.

While building on a solid theoretical foundation (at least as far as mathematical correctness of optimizations considered is concerned), namely categorical semantics (initial algebras, monads etc.), this paper is not written as a semantics paper but for functional programmers. We present our theory development and examples on Haskell code, only alluding to the underlying theory. But actually, we have in mind a total programming language with inductive and coinductive types (based on sets and functions or a parametric model of polymorphism).

The paper is organized as follows. In Section 2, we discuss in quite some detail the motivating case of the datatype of lists and dynamic fusion of folds of appends, where we can turn the reverse function from quadratic-time to linear-time. In Section 3, we generalize from the list datatype to free monads. In Section 4, we sketch a further generalization to monadic inductive datatypes arising from parameterized monads, to then continue, in Section 5, with hints regarding other types of generalizations (to functorial inductive datatypes, to comonadic datatypes etc.) In Section 6, we review some related work, concluding and outlining some future research goals in Section 7.

## 2 Lists

We introduce the idea of explicit binds on the datatype of lists. This is the simplest interesting case and allows us to treat the famous optimization test case of reversing a list.

### 2.1 True Lists

Lists are defined as an inductive datatype with two constructors, nil and cons. The canonical way to consume a list, resulting from the inductive nature of the datatype, is to fold it.

A central function on lists, constituting the multiplication operation of the important monoid structure on list, is appending a list to a given one. Append is a simple fold, but we have a reason to pay it special attention.

```
data [e] = [] | e : [e]

foldL :: x -> (e -> x -> x) -> [e] -> x
foldL n c []       = n
foldL n c (e : es) = e `c` foldL n c es

(++) :: [e] -> [e] -> [e]
[]       ++ es' = es'
(e : es) ++ es' = e : (es ++ es')
```

[1] The terminology of 'explicit binds' derives from lambda calculi with explicit substitutions [1]. Explicit substitutions are object syntax; they are not meta-notation for actual substitution. Lambda-terms are a monad with substitution as the bind operation. For us, therefore, explicit substitutions are a special case of explicit binds.

```
{-
-- structured definition as a foldL
es ++ es' = foldL es' (:) es
-}
```

The unit of the monoid is given by singleton lists. The reverse function is a fold using both the singleton and append functions.

```
slgtL :: e -> [e]
slgtL e = e : []

reverseL :: [e] -> [e]
reverseL [] = []
reverseL (e : es) = reverseL es ++ slgtL e

{-
-- structured definition as a foldL
reverseL = foldL [] (\ e esR -> esR ++ slgtL e)
-}
```

This is the natural primary definition of reverse from the mathematical point of view. But operationally, it is unsatisfactory, as it runs in quadratic time. A list is reversed by appending singletons of its elements to longer and longer prefixes of the reversed list. Each such append action involves a traversal of the current prefix. Roughly, for the reversal of a 3-element list, we get the following behavior:

```
reverseL [0,1,2]

==> reverseL [1,2] ++ [0]
==> (reverseL [2] ++ [1]) ++ [0]
==> ((reverseL [] ++ [2]) ++ [1]) ++ [0]

==> (([] ++ [2]) ++ [1]) ++ [0]         -- TRAVERSE []

==> ([2] ++ [1]) ++ [0]                 -- TRAVERSE [2]
==> (2 : ([] ++ [1])) ++ [0]

==> [2,1] ++ [0]                        -- TRAVERSE [2,1]
==> 2 : ([1] ++ [0])
==> 2 : 1 : ([] ++ [0])

==> [2,1,0]
```

The most common way to overcome this embarrassment and obtain the desired linear-time behavior is to reject the above definition and adopt a different one, resorting to an accumulator. But this takes ad-hoc work and means giving up the natural definition in favor of a completely different one, whereby the mathematical equivalence of the two is by no means immediate.

In the solutions of this paper, we repair the time-complexity of the reverse function without changing its definition. Instead, we change the underlying implementation of the list datatype.

### 2.2 Towards other Implementations of Lists

To be able to work systematically with multiple implementations of lists, we introduce a type class of list implementations. In essence, we say that are willing to use as the type of lists any type supporting nil, cons, fold and append. Singleton lists and list reversal are defined via this interface generically.

```
class ListType e es | es -> e where
  nil :: es
  cons :: e -> es -> es
  fold :: c -> (e -> c -> c) -> es -> c
  app :: es -> es -> es

sglt :: ListType e es => e -> es
sglt e = e `cons` nil

reverse :: ListType e es => es -> es
reverse = fold nil (\ e esR -> esR `app` sglt e)
```

Note that we could not have defined fold generically in terms of nil and cons, as structural recursion with the help of pattern-matching is not available: nil and cons are not constructors any more, just functions; in fact, the list type is not necessarily inductive. Note also that we could have defined append generically as a fold, but refrained from doing so. Our reasons and why this is fine will become clear shortly.

Of course, the "true" lists provide an implementation of lists, the primary such. Any other implementation interprets into the true lists and, conversely, the true lists are representable in any other implementation.

```
instance ListType e [e] where
  nil = []
  cons = (:)
  fold = foldL
  app = (++)

fromL :: ListType e es => [e] -> es
fromL = foldL nil cons

toL ::  ListType e es => es -> [e]
toL = fold [] (:)
```

We allow an implementation of lists to have multiple representations for the same list. Hence, while it is imperative that `toL (fromL es) == es`, we do not insist on `fromL (toL es) == es`. Instead, we define `es ~ es'` to mean `toL es == toL es'`. Cons, fold and append must respect this equivalence. Further, the natural defining equations of fold and append need not hold strictly on the level of representations, but they must hold up to `~`. (Mathematically speaking, the *quotient* of the implementation by `~` must be isomorphic to the true lists.)

We are now ready to introduce our two non-standard implementations of lists.

### 2.3 Lists with Explicit ("Frozen") Appends

The underlying idea of our first non-standard implementation of lists is to treat *folds of appends* specifically by making append an additional constructor of lists and crafting the corresponding additional clause for the append pattern in the definition of fold with care.

We introduce an inductive datatype with three constructors: in addition to the customary nil and cons constructors, there is an "explicit" (or "frozen") append constructor.

```
data ListX e = Nil | e :< ListX e | ListX e :++ ListX e
```

While nil and cons are obviously to be implemented as the corresponding constructors, regarding fold (of lists) and append, we have options. First, we can define the append function to be the fold for the true append, but we can also just use the append constructor. And, second, we have a choice in the definition of fold where we have to make a clause for the append pattern. We can confine ourselves to exploiting that the append constructor is a representation of the append function and inlining the definition of append as a fold. But we can also take the opportunity and *fold-fuse* in addition. We go for the latter choices (annotated as "smart" rather than "naive" in the code below).

```
instance ListType e (ListX e) where
   nil = Nil
   cons = (:<)

   fold n c Nil          = n
   fold n c (e :< es)    = e `c` fold n c es

-- fold n c (es :++ es') = fold n c (es `app` es')                    -- NAIVE
--                       = fold n c (fold es' cons es)
   fold n c (es :++ es') = fold (fold n c es') c es                   -- SMART

-- es `app` es' = fold es' cons es                                    -- NAIVE
   app = (:++)                                                        -- SMART
```

Automagically, the generically defined reverse function becomes linear-time. This happens because, on its own, the reverse function no longer truly appends lists. It only forms appropriate frozen appends. These melt away when we apply some fold to the result of the reversal, e.g., when converting from the implementation to real lists with `toL`. We get:

```
reverse (fromL [0,1,2] :: ListX Int)
==> (((Nil :++ (2 :< Nil)) :++ (1 :< Nil)) :++ (0 :< Nil))

toL (reverse (fromL [0,1,2] :: ListX Int))
==> [2,1,0]
```

In more detail, the last answer is computed roughly like this:

```
toL (((nil :++ sglt 2) :++ sglt 1) :++ sglt 0)
==> fold [] (:)
      (((nil :++ sglt 2) :++ sglt 1) :++ sglt 0)
==> fold (fold [] (:) (sglt 0)) (:)
      ((nil :++ sglt 2) :++ sglt 1)
==> fold (fold (fold [] (:) (sglt 0)) (:) (sglt 1)) (:)
      (nil :++ sglt 2)
==> fold (fold (fold (fold [] (:) (sglt 0)) (:) (sglt 1)) (:) (sglt 2)) (:)
      nil

==> fold (fold (fold [] (:) (sglt 0)) (:) (sglt 1)) (:) (sglt 2)
==> 2 : fold (fold [] (:) (sglt 0)) (:) (sglt 1)
==> 2 : 1 : fold [] (:) (sglt 0)
==> 2 : 1 : 0 : []
```

We have defined fold by general recursion. Of course, we should ask whether we can define it also with honest structural recursion. The answer is positive.

The fold function for the inductive type of lists-with-explicit-appends is this:

```
foldLX :: x -> (e -> x -> x) -> (x -> x -> x) -> ListX e -> x
foldLX n c ap Nil          = n
foldLX n c ap (e :< es)    = e `c` foldLX n c ap es
foldLX n c ap (es :++ es') = foldLX n c ap es `ap` foldLX n c ap es'
```

The fold of lists can be defined as a special case at a higher type:

```
instance ListType e (ListX e) where
   ...
   -- structured definition as a foldLX
   fold n c es = foldLX id (\ e i -> c e . i) (\ i i' -> i . i') es n
   ...
```

It is not immediate, but this definition is equivalent to the "smart" definition both mathematically and operationally, i.e., reverse remains linear-time.

We finish this discussion of lists-with-explicit-appends by remarking that the smart append-clause of the definition of fold can also be split into cases, leading to a "small-step" version of our special treatment of folds of appends:

```
instance ListType e (ListX e) where
   ...
   -- small-step definition
   fold n c (Nil           :++ es') = fold n c es'
   fold n c ((e :< es0)    :++ es') = e `c` fold n c (es0 :++ es')
   fold n c ((es0 :++ es1) :++ es') = fold n c (es0 :++ (es1 :++ es'))
   ...
```

### 2.4 Church Lists

Our second non-standard implementation of lists is based on the so-called impredicative encoding of lists (or the Church encoding, cf. Church numerals as an implementation of natural numbers), which is also the basis of shortcut deforestation. A list is represented as a *build* by providing a polymorphic function encompassing all folds of this particular list. Folding then becomes instantiation of this polymorphic function for a particular return type and application to given arguments. Nil and cons are specific builds describing how nil and cons-lists are folded (based on the customary definition of fold).

Regarding append, we have options again. We can view append as a fold (i.e., instantiation and application), based on the definition of append as a fold. But a better idea is to see append as a build describing how it is folded. This allows us to invoke the fold-fused description of folds of appends and renders reverse linear-time again.

```
data ListCh e = Build (forall x. x -> (e -> x -> x) -> x)

instance ListType e (ListCh e) where
   nil              = Build (\ n c -> n)
-- e `cons` es      = Build (\ n c -> e `c` fold n c es)
   e `cons` Build g = Build (\ n c -> e `c` g n c)

   fold n c (Build g) = g n c

-- es      `app` es'      = fold es' cons es                                  -- NAIVE
-- Build g `app` es'      = g es' cons

-- es      `app` es'      = Build (\ n c -> fold n c (fold es' cons es))      -- STILL NAIVE
-- es      `app` es'      = Build (\ n c -> fold (fold n c es') c es)         -- SMART
   Build g `app` Build g' = Build (\ n c -> g (g' n c) c)
```

### 2.5 Comparison

We have seen two implementations of lists that detect folds of appends (as they emerge in computations) and treat them specifically, whereby the standard definition of reverse becomes linear-time.

Common to both of them is that, at their core, they rely on fold-fusion, replacing `fold n c (es ‘app‘ es’)` by `fold (fold n c es’) c es`. Unusually, however, this rewrite rule is not employed for transforming programs statically. Rather, it is exploited to twist the naive versions of these implementations in such a way that we can (if we wish) think of this rewrite rule as being applied dynamically whenever a fold of an append crops up in a computation.

The mechanisms employed by the two implementations to detect and control folds of appends are quite different (and, in an informal sense, dual to each other). In the explicit appends approach, appends are made constructors, which, on their own, do nothing. Folds of appends are handled by the carefully crafted append-clause in the definition of fold. In the Church lists approach, folding amounts to just instantiation and application and does not nothing smart. How appends are folded is controlled by a clever definition of append as a build.

### 2.6 From Fold to Primitive Recursion

Let us consider the following variation of reverse, which, given a list, reverses its maximal prefix whose all elements satisfy a given predicate, keeping the remainder unreversed.

```
reverseWhileL :: (e -> Bool) -> [e] -> [e]
reverseWhileL f [] = []
reverseWhileL f (e : es) = if f e then reverseWhileL f es ++ sgltL e else e : es
```

reverseWhile is not a fold, but it is a primitive-recursive function (in the cons clause we depend not only on the result of the recursive call on the tail of the given list, but also on the tail itself).

```
primrecL :: x -> (e -> x -> [e] -> x) -> [e] -> x
primrecL n c []       = n
primrecL n c (e : es) = c e (primrecL n c es) es

reverseWhileL :: (e -> Bool) -> [e] -> [e]
reverseWhileL f = primrecL [] (\ e esR es -> if f e then esR ++ sgltL e else e : es)
```

We can define reverseWhile generically, assuming an implementation of lists that supports primitive recursion.

```
class ListType e es | es -> e where
  ...
  primrec :: x -> (e -> x -> es -> x) -> es -> x

reverseWhile :: ListType e es => (e -> Bool) -> es -> es
reverseWhile f = primrec nil (\ e esR es -> if f e then esR ‘app‘ sglt e else e  ‘cons‘ es)

instance ListType e [e] where
  ...
  primrec = primrecL
```

Just as reverse, reverseWhile on true lists runs in quadratic time. We can speed it up to linear-time by switching to lists-with-explicit-appends.

```
instance ListType e (ListX e) where
   ...
   primrec n c Nil          = n
   primrec n c (e :< es)    = c e (primrec n c es) es
   primrec n c (es :++ es') = primrec (primrec n c es') c' es
                                 where c' e x es = c e x (es :++ es')
```

It is, of course, also possible define primitive recursion generically as a projection from a fold computing a pair (of the value of the function of interest and a copy of the argument).

```
primrec :: ListType e es => x -> (e -> x -> es -> x) -> es -> x
primrec n c = fst . fold (n, nil) (\ e (x, es) -> (c e x es, e `cons` es))
```

But this approach has the drawback that the tail function, naturally defined as follows, becomes linear-time rather than constant-time. Moreover, no constant-time definition of tail in terms of fold is possible.[2]

```
tail :: ListType e es => es -> es
tail = primrec nil (\ _ _ es -> es)
```

## 3 Leaf Trees (Free Monads)

Lists understood, we can proceed to a whole class of datatypes for which optimizations similar to those considered above are possible.

Lists and the append function are a special case of (wellfounded) *leaf-labelled trees* (with a fixed branching factor) and *grafting*. The official name for these datatypes is *free monads*. Grafting is the bind operation of such monads, while the unit (return) is given by leaves. Lists with explicit appends and Church lists generalize to leaf-labelled trees with explicit (or "frozen") grafts and Church representations of leaf-labelled tree datatypes. We get effortless efficiency for functions manipulating leaf-labelled trees.

### 3.1 Leaf Trees

Given a functor specifying a branching factor, the leaf-labelled trees with this branching factor are given by the following inductive datatype, with the following induced fold operation:

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

foldT :: Functor f => (a -> x) -> (f x -> x) -> Tree f a -> x
foldT lf nd (Leaf a ) = lf a
foldT lf nd (Node ts) = nd (fmap (foldT lf nd) ts)
```

[2] Notice that printing a list can never be faster than linear in its size, but printing a constant many first elements, e.g., just the head, can be constant-time. This is what we keep in mind when speaking about constant-time list-valued functions on lists.

The important operation of grafting trees on a tree is defined as follows and is a fold:

```
graftT :: Functor f => Tree f a -> (a -> Tree f b) -> Tree f b
Leaf a  `graftT` k = k a
Node ts `graftT` k = Node (fmap (`graftT` k) ts)

{-
-- structured definition
t `graftT` k = foldT k Node t
-}
```

The leaf tree datatype is a monad with the leaf constructor as the unit and graft as bind (in fact, it is the free monad on our given functor). Since we want to work with multiple implementations of leaf trees, we define a type class of leaf tree implementations. True leaf trees are of course the primary instance.

```
class Functor f => TreeType f t | t -> f where
   leaf :: a -> t a
   node :: f (t a) -> t a
   foldtree :: (a -> x) -> (f x -> x) -> t a -> x
   graft :: t a -> (a -> t b) -> t b

instance Functor f => TreeType f (Tree f) where
   leaf = Leaf
   node = Node
   foldtree = foldT
   graft = graftT
```

Lists are a special case of leaf trees with trivial leaf-labels.

```
instance Functor ((,) e) where
   fmap f (e, x) = (e, f x)

-- this requires -fallow-undecidable-instances
instance TreeType ((,) e) t => ListType e (t ()) where
   nil = leaf ()
   cons e es = node (e, es)
   fold n c es = foldtr (\ () -> n) (\ (e, x) -> c e x) es
   app es es' = graft es (\ () -> es')
```

### 3.2 Leaf Trees with Explicit Grafts

To obtain a special treatment (dynamic fusion) of folds of grafts, we can introduce an inductive datatype of trees with explicit grafts as follows:

```
data TreeX f b = LeafX b | NodeX (f (TreeX f b))
                         | forall a . TreeX f a `GraftX` (a -> TreeX f b)

instance Functor f => TreeType f (TreeX f) where
   leaf = LeafX
   node = NodeX

   foldtree lf nd (LeafX b )     = lf b
   foldtree lf nd (NodeX ts)     = nd (fmap (foldtree lf nd) ts)

-- foldtree lf nd (t `GraftX` k) = foldtree lf nd (t `graft` k)                    -- NAIVE
--                               = foldtree lf nd (foldtree k node t)

   foldtree lf nd (t `GraftX` k) = foldtree (foldtree lf nd . k ) nd t             -- SMART

-- t `graft` k = foldtree k node t                                                 -- NAIVE
   graft = GraftX                                                                  -- SMART
```

Notice that we exploit fold-fusion and replace `foldtree lf nd (t ‘graft‘ k)` by `foldtree (foldtree lf nd . k ) nd t`.

The definition of fold above is general-recursive. A structured definition as a fold for trees with explicit grafts (instantiated at a higher type) is also possible.

```
foldTX :: (Functor f, Functor g) => (forall b . b -> g b)
                -> (forall b . f (g b) -> g b)
                -> (forall a b . g a -> (a -> g b) -> g b)
                                          -> TreeX f b -> g b
foldTX lf nd bd (LeafX b )     = lf b
foldTX lf nd bd (NodeX ts)     = nd (fmap (foldTX lf nd bd) ts)
foldTX lf nd bd (t ‘GraftX‘ k) = bd (foldTX lf nd bd t) (foldTX lf nd bd . k)

data CPS x b = CPS { uncps :: (b -> x) -> x }

instance Functor (CPS c) where
   fmap f (CPS c) = CPS (\ k -> c (k . f))

instance Monad (CPS c) where
   return b = CPS (\ k -> k b)
   CPS c >>= kl = CPS (\ k -> c (\ b -> uncps (kl b) k))

instance Functor f => TreeType f (TreeX f) where
   ...
   {-
   -- structured definition
   foldtree lf nd t = uncps (foldTX return
                                    (\ ts -> CPS (\ k -> nd (fmap (\ (CPS c) -> c k) ts)))
                                    (>>=)
                                    t
                             ) lf
   -}
   ...
```

### 3.3 Church Leaf Trees

A good alternative to leaf trees with explicit grafts is provided by Church leaf trees. Here we obtain an efficient treatment of folds of grafts by giving up the natural definition of graft as a fold in favor of an equivalent build.

```
data TreeCh f a = Build (forall x. (a -> x) -> (f x -> x) -> x)

instance Functor f => TreeType f (TreeCh f) where
   leaf a  = Build (\ lf nd -> lf a)

-- node ts = Build (\ lf nd -> nd (fmap (foldtree lf nd) ts))
   node ts = Build (\ lf nd -> nd (fmap (\ (Build g) -> g lf nd) ts))

   foldtree lf nd (Build g) = g lf nd

-- t       ‘graft‘ k = foldtree k node t                                        -- NAIVE
-- Build g ‘graft‘ k = g k node

-- t       ‘graft‘ k = Build (\ lf nd -> foldtree lf nd (foldtree k node t))  -- STILL NAIVE
-- t       ‘graft‘ k = Build (\ lf nd -> foldtree (foldtree lf nd . k) nd t)  -- SMART
   Build g ‘graft‘ k = Build (\ lf nd -> g (\ a -> let Build h = k a in h lf nd) nd)
```

## 4 Monadic Inductive Datatypes from Parameterized Monads

From the free monad on a functor (for a functor $H$, the monad $T$ where $TA = \mu X.\, A + H\, X$), we can generalize to the monad inductively defined from a pa-

rameterized monad (viz., for a binary functor $R$ monadic in the first argument uniformly in the second, the monad $TA = \mu X.\, R\,(A, X)$), cf. the construction of [5,24].

The primary example here is that of rose trees (the case of $TA = \mu X.\, A \times \mathsf{List}\, X$; notice that $R\,(A, X) = A \times \mathsf{List}\, X$ is a monad in $A$ because of the monoid structure on $\mathsf{List}\, X$).

We can define:

```
class RoseType t where
   rose :: a -> [t a] -> t a
   foldrose :: (a -> [x] -> x) -> t a -> x
   bind :: t a -> (a -> t b) -> t b

-- true rose trees

data Rose a = Rose a [Rose a]

instance RoseType Rose where
   rose = Rose

   foldrose r (Rose a ts) = r a (map (foldrose r) ts)

   Rose a ts `bind` k = let Rose b ts' = k a in Rose b (ts' ++ map (`bind` k) ts)

-- rose trees with explicit grafts

data RoseX b = RoseX b [RoseX b] | forall a . RoseX a `BindX` (a -> RoseX b)

instance RoseType RoseX where
   rose = RoseX

   foldrose r (RoseX b ts)  = r b (map (foldrose r) ts)
   foldrose r (t `BindX` k) = foldrose (\ a cs ->
                                          foldrose (\ b cs' -> r b (cs' ++ cs)) (k a)) t
   t `bind` k = t `BindX` k
```

## 5 Other Generalizations

The technique of freezing an operation in a constructor and melting it when folding does not apply only to the bind operation of monadic inductive datatypes. It is applicable to any inductive datatype with an important operation that can be folded smartly. And, what is more, the dual idea applies also to a coinductive datatype that comes with an important operation specializing efficiently for unfolds. In this case, it makes sense to turn the operation into an additional destructor (a field selector) of the coinductive datatype. Here are some examples:

- functors and their fmap operations—covers any kind of labelled datastructures that support relabeling;
- free completely iterative monads—non-wellfounded leaf-labelled trees with grafting and (a specific flavor of) iteration;
- cofree recursive comonads—wellfounded node-labelled trees with upward accumulation and recursion;
- non-empty list type—a special case of the previous item; we get effortless efficiency for comonadically structured causal dataflow computation [25].

We will treat these examples elsewhere, but here is a preview.

Suppose that we want folds of maps of lists to work efficiently, i.e., we want dynamic fusion of `fold n c (map f es)` into `fold n (c . f) es` . In particular, we wish to define the function returning all prefixes of a given list in the natural way

```
prefixes :: [e] -> [[e]]
prefixes []       = [[]]
prefixes (e : es) = [] : map (e :) (prefixes es)
```

but also have this function behave efficiently.

The solution is to introduce an inductive datatype of lists with explicit maps

```
data ListX e = Nil | e :< ListX e | forall d. MapX (d -> e) (ListX d)
```

and to define fold and map suitably:

```
fold n c (e :< es)   = e `c` fold n c es
fold n c (MapX f es) = fold n (c . f) es

map f es = MapX f es
```

With true lists, the last prefix of a list (which is of course the list itself) is computed in quadratic time while the fine-tuned implementation computes it in linear time.

## 6 Related Work

The body of literature on optimizations of list and tree manipulating functions with traits in common with those in this paper is vast and it is difficult to give a complete picture.

Hughes [9] proposed representing lists as appends to them (difference lists in functional programming), with an application to reverse. Voigtländer [26] generalized this idea to leaf trees, choosing to represent them as corresponding graft functions (using the codensity monad for the leaf tree datatype). Kmett [14,15] entertained both this idea and some variations and showed how they interrelate.

Wadler [30] noticed that folds of appends of lists can be optimized (so that the appended list is never constructed). Wadler [28] also introduced deforestation as the general paradigm of program optimization by cutting out production and consumption of intermediate datastructures. Kühnemann and Maletti [17] generalized Wadler's point about folds of appends of lists to folds of grafts of leaf trees.

Shortcut deforestation is a specific program transformation method based on representing list functions in terms of folds and builds and rewriting them based on the fold/build rule, due to Gill et al. [8]. Takano and Meijer [23] generalized it to general inductive datatypes. Gill [6] invented the augment operation (combining append and build) and fold/augment rule for lists. Folds and builds are the native interface of Church encodings of datatypes, shown to correctly implement inductive types in parametric models, e.g., by Wadler [29], cf. also the

discussion of mathematical correctness of shortcut deforestation by Fegaras [2]. Pavlovic [19] and Ghani, Uustalu and Vene [4] justified shortcut deforestation and Church encodings with strong dinaturality. Ghani, Uustalu and Vene [4,5] generalized the augment combinator and fold/augment rule to free monads and monadic inductive types.

Johann [11,12] proved shortcut deforestation correct in an operational setting.

Kühnemann [16] and Jürgensen and Vogler [13] showed that syntactic composition of tree transducers is shortcut deforestation.

Svenningsson [22] and Voigtländer [27] have introduced several new variations of shortcut deforestation. Combinations of shortcut deforestation and monads, different from those in this paper, have been suggested by Manzino and Pardo [18] and by Ghani and Johann [3].

Gill, Hutton and Jaskelioff [7,10] have recently revisited Peyton Jones and Launchbury's worker/wrapper transformation [20], which is a systematic way of reducing computations of one type into computations in another type, generally with the purpose of improving efficiency. They have shown a general formulation of the transformation correct and proved that Hughes' [9] and Voigtländer's [26] methods form instances.

The work presented here is closest to those by Hughes [9] and Voigtländer [26] and their worker/wrapper recasts. Just as Hughes and Voigtländer, we wish to avoid rewriting function definitions and seek instead to change datatype implementations in the background. The difference is that we go a bit further in terms of datatypes covered and (what is more significant) consider different reimplementations. In particular, we find that explicit binds provide a particularly appealing reimplementations on the intuitive level, whereas Church encodings are nice in that they are entirely standard—all we do in these encodings is to opt for the operationally best behaved definition of bind.

## 7 Conclusion

We have shown a number of examples where the efficiency of functions on a datatype can be boosted without giving up the natural function definitions by switching to a different implementation of the datatype.

Our leading idea was to seek implementations where special care can be taken of operations that are central for the datatype (e.g., bind or map), but tend to contribute to inefficiency by building unnecessary intermediate datastructures. We saw that, with luck, such "positive discrimination" alone can give major gains.

Two lessons we learned while we did this research were that functionalization by itself is no silver bullet and datatypes are not intrinsically inefficient. In our endeavor, we were able to achieve efficiency gains both with datatype and functional reimplementations of the datatypes that we considered: efficiency with and without trees.

Quite obviously, we cannot claim we have invented anything very new. The ideas of this paper have appeared in various guises in many places in the literature. Reasonably, we expect we can claim that we have elucidated, organized

and generalized these ideas. The scope of application of the more exotic generalizations remains to be explored.

One project we would definitely like to undertake in the future is a systematic quantitative study of the efficiency improvements of the optimizations of this paper. While the mathematical correctness of these optimizations is out of question, for improvement proofs we do not even have a good mathematical framework currently and the literature contains few hints. We expect that we could benefit from the new exciting work of Seidel and Voigtländer [21] in this direction.

**Acknowledgements.** I am grateful to Ralph Matthes, Edward Kmett, Varmo Vene, Graham Hutton and Mauro Jaskelioff for discussions and to my anonymous referees for comments.

This research was supported by the Estonian Science Foundation grants no. 6940 and 9475 and the ERDF funded Estonian Center of Excellence in Computer Science, EXCS. My trip to Kobe will be supported by the Tiger University Plus programme of the Estonian Information Technology Foundation.

## References

1. Abadi, M., Cardelli, L., Curien, P.-L., Levy, J.-J.: Explicit substitutions. J. of Funct. Program. 1(4), 375–416 (1991)
2. Fegaras, L.: Using the parametricity theorem for program fusion. Tech. report CSE-96-001. Oregon Grad. Inst. (1996)
3. Ghani, N., Johann, P.: Short cut fusion for effects. In: Achten, P., Koopman, P., Morazán, M. (eds.) Trends in Functional Programming, vol. 9, pp. 113–128. Intellect, Bristol (2009)
4. Ghani, N., Uustalu, T., Vene, V.: Build, Augment and Destroy, Universally. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 327–347. Springer, Heidelberg (2004)
5. Ghani, N., Uustalu, T., Vene, V.: Generalizing the augment combinator. In: Loidl, H.-W. (ed.) Trends in Functional Programming, vol. 5, pp. 65–78. Intellect, Bristol (2006)
6. Gill, A.: Cheap Deforestation for Non-strict Functional Languages. PhD thesis. University of Glasgow (1996)
7. Gill, A., Hutton, G.: The worker/wrapper transformation. J. of Funct. Program. 19(2), 227–251 (2009)
8. Gill, A., Launchbury, J., Peyton Jones, S.L.: A short cut to deforestation. In: Conf. Record of 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA 1993, Copenhagen, pp. 223–232. ACM Press, New York (1993)
9. Hughes, J.: A novel representation of lists and its application to the function 'reverse'. Inf. Process. Lett. 22(3), 141–144 (1986)
10. Hutton, G., Jaskelioff, M., Gill, A.: Factorising folds for faster functions. J. of Funct. Program. 20(3-4), 353–373 (2010)
11. Johann, P.: A generalization of short-cut fusion and its correctness proof. Higher-Order and Symb. Comput. 15(4), 273–300 (2002)
12. Johann, P.: Short-cut fusion is correct. J. of Funct. Program. 13(4), 797–814 (2003)

13. Jürgensen, C., Vogler, H.: Syntactic composition of top-down tree transducers is short cut fusion. Math. Struct. in Comput. Sci. 14(2), 215–282 (2004)
14. Kmett, E.: Kan extensions 1–3. Posts on the author's blog 'The Comonad.Reader' (2008), `http://comonad.com/reader/`
15. Kmett, E.: Free monads for less 1–3. Posts on the author's blog 'The Comonad.Reader' (2011), `http://comonad.com/reader/`
16. Kühnemann, A.: Comparison of Deforestation Techniques for Functional Programs and for Tree Transducers. In: Middeldorp, A., Sato, T. (eds.) FLOPS 1999. LNCS, vol. 1722, pp. 114–130. Springer, Heidelberg (1999)
17. Kühnemann, A., Maletti, A.: The Substitution Vanishes. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 173–188. Springer, Heidelberg (2006)
18. Manzino, C., Pardo, A.: Shortcut fusion of monadic programs. J. of Univ. Comput. Sci. 14(21), 3431–3446 (2008)
19. Pavlovic, D.: Logic of build fusion. Techn. report KES.U.00.9. Kestrel Inst. (2000)
20. Peyton Jones, S.L., Launchbury, J.: Unboxed Values as First Class Citizens in a Non-strict Functional Language. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 636–666. Springer, Heidelberg (1991)
21. Seidel, D., Voigtländer, J.: Improvements for free. In: Massink, M., Norman, G. (eds.) Proc. of 9th Wksh. on Quantitative Aspects of Programming Languages, QAPL 2011, Saarbrücken. Electron. Proc. in Theor. Comput. Sci., vol. 57, pp. 89–103. Elsevier, Amsterdam (2011)
22. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: Proc. of 7th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2002, Pittsburgh, PA, pp. 124–132. ACM Press, New York (2002)
23. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: Conf. Record of 7th ACM SIGPLAN/SIGARCH Conf. on Functional Programming Languages and Computer Architecture, FPCA 1995, La Jolla, pp. 306–316. ACM Press, New York (1995)
24. Uustalu, T.: Generalizing substitution. Theor. Inform. and Appl. 37(4), 315–336 (2003)
25. Uustalu, T., Vene, V.: The Essence of Dataflow Programming. In: Horváth, Z. (ed.) CEFP 2005. LNCS, vol. 4164, pp. 135–167. Springer, Heidelberg (2006)
26. Voigtländer, J.: Asymptotic Improvement of Computations over Free Monads. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 388–403. Springer, Heidelberg (2008)
27. Voigtländer, J.: Concatenate, reverse and map vanish for free. In: Proc. of 7th Int. Conf. on Functional Programming, ICFP 2002, Pittsburgh, PA, pp. 14–25. ACM Press, New York (2002)
28. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theor. Comput. Sci. 73(2), 231–248 (1990)
29. Wadler, P.: Recursive types for free! Unpublished note (1990)
30. Wadler, P.: The concatenate vanishes. Unpublished note (1987, updated 1989)
31. Wadler, P.: Theorems for free! In: Proc. of 4th Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA 1989, London, pp. 347–359. ACM Press, New York (1989)