Monadic Augment and Generalised Short Cut Fusion

Neil Ghani
University of Leicester
Leicester LE1 7RH, UK
ng13@mcs.le.ac.uk

Patricia Johann *
Rutgers University
Camden, NJ 08102, USA
pjohann@crab.rutgers.edu

Tarmo Uustalu †
Inst. of Cybernetics
EE-12618 Tallinn, Estonia
tarmo@cs.ioc.ee

Varmo Vene ‡
Univ. of Tartu
EE-50409 Tartu, Estonia
varmo@cs.ut.ee

Abstract

Monads are commonplace programming devices that are used to uniformly structure computations with effects such as state, exceptions, and I/O. This paper further develops the monadic programming paradigm by investigating the extent to which monadic computations can be optimised by using generalisations of short cut fusion to eliminate monadic structures whose sole purpose is to "glue together" monadic program components.

We make several contributions. First, we show that *every* inductive type has an associated `build` combinator and an associated short cut fusion rule. Second, we introduce the notion of an *inductive monad* to describe those monads that give rise to inductive types, and we give examples of such monads which are widely used in functional programming. Third, we generalise the standard `augment` combinators and `cata`/`augment` fusion rules for algebraic data types to types induced by inductive monads. This allows us to give the first `cata`/`augment` rules for some common data types, such as rose trees. Fourth, we demonstrate the practical applicability of our generalisations by providing Haskell implementations for all concepts and examples in the paper. Finally, we offer deep theoretical insights by showing that the `augment` combinators are monadic in nature, and thus that our `cata`/`build` and `cata`/`augment` rules are arguably the best generally applicable fusion rules obtainable.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Denotational semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Functional constructs, program and recursion schemes

General Terms design, languages, theory

Keywords short cut fusion, build, augment, monads, bind

* Supported in part by National Science Foundation grant CCF-0429072.

† Supported in part by Estonian Science Foundation grant 5567.

‡ Supported in part by Estonian Science Foundation grant 5567.

ICFP '05 September 26–28, 2005, Tallinn, Estonia.

1. Introduction

As originally conceived by Moggi, *monads* form a useful computational abstraction which models diverse effects such as stateful computations, exceptions, and I/O in a modular, uniform, and principled manner [13]. Wadler [24] led the call to turn Moggi's theory of effectful computation into a practical programming methodology, and showed how to use monads to structure such computations. Monads are now firmly established as part of Haskell [16], supported by specific language features and used in a wide range of applications. The essential idea behind monads is the type-safe separation of values from effectful computations that return those values.[1] Because monads abstract the nature of effectful computations, and in particular the mechanism for composing them, monadic programs are often more highly structured than non-monadic ones which perform the same computational tasks. Monadic programs thus boast the usual benefits of structured code, namely being easier to read, write, modify, and reason about than their non-monadic counterparts. However, compositionally constructed monadic programs also tend to be less efficient than monolithic ones. In particular, a component in such a program will often construct an intermediate monadic structure — i.e., an intermediate structure of type `m t` where `m` is a monad and `t` is a type — only to have it immediately consumed by the next component in the composition.

Given the widespread use of monadic computations, it is natural to try to apply automatable program transformation techniques to improve the efficiency of modularly constructed monadic programs. *Fusion* is one technique which has been used to improve modularly constructed functional programs, and a number of fusion transformations appropriate to the non-monadic setting have been developed in recent years [1, 6, 7, 8, 9, 19, 20, 21, 23]. Perhaps the best known of these is *short cut fusion* [6], a local transformation based upon two combinators — `build`, which produces lists in a uniform manner, and `foldr`, which uniformly consumes them — and a single, oriented replacement rule known as the `foldr`/`build` rule. (See Section 3.) The `foldr`/`build` rule replaces calls to `build` which are immediately followed by calls to `foldr` with equivalent computations that do not construct the intermediate lists introduced by `build` and consumed by `foldr`. Eliminating such lists via short cut fusion can significantly improve the efficiency of programs.

Unfortunately, there are common list producers — such as the append function — that `build` cannot express in a manner suitable for short cut fusion. This led Gill to introduce a list producer, called `augment`, which generalises `build`, together with an accompanying `foldr`/`augment` fusion rule for lists [5]. This rule has

[1] Monads, such as the expression monad in Example 1, which correspond to ordinary algebraic data types can be thought of as having an effect of storing data in a data structure.

subsequently been generalised to give `cata`/`augment` rules which fuse producers and consumers of arbitrary non-list algebraic data types [8].[2] Fusion rules which are dual to the `foldr`/`build` rule (in a precise category-theoretic sense) [20, 21], and rules which eliminate list-manipulating operations other than data constructors [23], have also been developed.

This paper further generalises short cut fusion to rules which eliminate intermediate monadic structures. In order to write consumers of expressions of type `m t` in terms of `cata`s we restrict attention to types `m t` which are inductive types in a uniform manner. We call a monad `m` with the property that `m t` is an inductive data type for every type `t` an *inductive monad*.

Our first observation is that `build` combinators and `cata`/`build` fusion rules can be defined for *all* inductive types. As we demonstrate, this opens the way for a generic theory of fusion. Next, we ask whether `augment` combinators and `cata`/`augment` rules can similarly be generically defined. We show that there are inductive types which do not support `augment` combinators (see Section 4.2), but that a large class of inductive monads do. To describe these monads, we introduce the notion of a *parameterised monad*, and use the observation that the least fixed point of every parameterised monad is an inductive monad [22] to define generic `augment` combinators and `cata`/`augment` rules for all such fixed points. We illustrate our results with expression languages, rose trees, interactive input/output monads, and hyperfunctions, all of which are commonly used monads arising as least fixed points of parameterised monads. When applied to types for which `augment` combinators are already known, our results yield more expressive `augment` combinators. On the other hand, the examples involving rose trees and interactive input/output monads show that there are well-known and widely used monads for which neither `augment` combinators nor `cata`/`augment` fusion rules were previously known, but for which we can derive both. Since, as we show in Section 4.3, the `bind` operations for monads which are least fixed points of parameterised monads can be written in terms of our `augment` combinators, our `cata`/`augment` fusion rules can be applied whenever an application of `bind` is followed by a `cata`. This is expected to be often, since the `bind` operation is the fundamental operation in monadic computation. We thus expect our `cata`/`augment` fusion rules to be widely applicable.

The results detailed in this paper are of practical interest since the `cata`/`augment` fusion rules we develop have the potential to improve the efficiency of modularly constructed programs using a variety of different monads. Our results are of theoretical importance as well: they clearly establish the monadic nature of our `augment` combinators by showing that they are interdefinable with the monadic `bind` operations. The fact that our results make it possible to define `cata`/`build` rules for all functors, as well as `cata`/`augment` rules for all least fixed points of parameterised monads, suggests that they are close to the best achievable. We expect, therefore, that our results will appeal to a variety of different audiences. Those who work with monads will be interested in parameterised monads and their applications, and those in the program transformation community will be interested in seeing their ideas for optimising computations successfully deployed in the monadic setting. We hope that, as with the best cross-fertilisations of ideas, ours will enable experts in each of these communities to gain greater understanding of, and facility with, the ideas and motivations of the other.

The concrete contributions of this paper are as follows:

- In Section 3 we derive a `build` combinator for the least fixed point of *any* functor, and show how this opens the way for an *algebra of fusion*.
- In Section 4 we define the notion of a *parameterised monad* and show that the least fixed point of *any* parameterised monad is a monad. We use this observation to generalise the standard `augment` combinators for algebraic data types to give `augment` combinators for *all* monads arising as least fixed points of parameterised monads. Finally, we argue that our `augment` combinators are inherently monadic in nature by showing that the `augment` combinator for each parameterised monad is interdefinable with the `bind` operation for the monad which is its least fixed point via the elegant equality

  ```
  augment g k = build g >>= k
  ```

  A more general development of `augment` combinators for a larger class of data types is hard to envisage.
- In Section 5 we generalise the standard `cata`/`augment` fusion rules for algebraic data types to give `cata`/`augment` rules for *all* monads arising as least fixed points of parameterised monads.
- We support this development with a variety of running examples and a Haskell implementation. The latter can be downloaded from `http://www.mcs.le.ac.uk/~ng13`.

We discuss related work in Section 6 and conclude in Section 7. Throughout the paper we assume as little background of the reader as possible. In particular, no knowledge of category theory is assumed or required and, in order to make this paper accessible to as wide an audience as possible, the correctness of the fusion rules presented here is given in a separate paper [4]. On the other hand, this paper is addressed to the functional programming community and, aside from using the same combinators, is disjoint from [4].

2. Why monads?

Functional programming was recognised early on as providing a clean programming environment in which programs are easy to read, write, and prove correct. But the problem of performing effectful computations in a purely functional language without compromising the advantages of the functional paradigm proved difficult to solve. Moggi's very nice solution was to tag types with "flags" which indicate that effects are associated with values of those types. For example, if `t` is a type and `m` flags a particular computational effect, then `m t` is a new computational type whose inhabitants can be thought of as performing effectful computations described by `m` and (possibly) returning results of type `t`. For example, the type `Int` contains integer values, while the computational type `State Env Int` contains functions which transform the current state (given by an element of type `Env`) into an integer value and a new state. (See Example 3 below.)

In order to program with computational types we need two operations. The first, called `return`, lifts any value of the underlying type to the trivial computation which returns that value. The second, called `bind` and written `>>=`, composes two computations which have the same type of effect. A flag `m` together with its two operations forms a *monad*. Monads are represented in Haskell via the type class

```
class Monad m where
  return :: a -> m a
  >>=    :: m a -> (a -> m b) -> m b
```

From a semantic perspective, `return` and `bind` are expected to satisfy the three monad laws [13]. These can be thought of as requiring that the composition of effectful computations be associative and that values act as left and right units for it. Satisfaction of the

[2] As is standard in Haskell, we use `foldr` to denote the standard catamorphism for lists. Catamorphisms for other inductive data types are written as `cata`.

monad laws is, however, not enforced by the compiler. Instead, it is the programmer's responsibility to ensure that the `return` and `bind` operations for any instance of Haskell's monad class behave appropriately.

EXAMPLE 1. *The algebraic data type* `Expr a` *represents simple arithmetic expressions.*

```
data Ops = Add | Sub | Mul | Div

data Expr a = Var a | Lit Int
                    | Op Ops (Expr a) (Expr a)

instance Monad Expr where
  return            = Var
  Var x       >>= k = k x
  Lit i       >>= k = Lit i
  Op op e1 e2 >>= k = Op op (e1 >>= k) (e2 >>= k)
```

EXAMPLE 2. *The type* `Maybe a` *consists of values of type* `a` *and a distinguished error value.*

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return        = Just
  Nothing >>= k = Nothing
  Just x  >>= k = k x
```

EXAMPLE 3. *The type* `State s a` *represents computations that can change states of type* `s` *while computing results of type* `a`*.*

```
newtype State s a = State {runState :: s -> (a,s)}

instance Monad (State s) where
  return x      = State (\s -> (x,s))
  State g >>= k = State (\s -> let (y,t) = g s
                               in runState (k y) t)
```

We conclude this section by demonstrating how monads systematise, simplify, and highlight the structure of effectful programs by allowing us to structure them as though they were non-effectful. Suppose we want to write an evaluator `eval :: Expr Int -> Int` for (closed) expressions over the type `a`. In a non-monadic setting we might have the clause

```
eval (Op Div e1 e2) = (eval e1) `div` (eval e2)
```

together with similar clauses for expressions involving the other arithmetic operators. To better accommodate exceptions — arising, for example, from attempting to divide by 0 — we could instead use a monadic evaluator `eval' :: Expr Int -> Maybe Int` and write

```
eval' (Op Div e1 e2) = liftM2 div (eval' e1)
                                  (eval' e2)
```

Here, `liftM2` is a built-in Haskell function which lifts functions over types to functions over their corresponding monadic types. Note how the essential structure of the computation remains faithfully represented in the definition of `eval'` while all error handling is abstracted and hidden in the use of the monadic operation `liftM2`.

3. Short cut fusion

As already noted, modularly constructed programs tend to be less efficient than their non-modular counterparts. A major difficulty is that the direct implementation of compositional programs *literally* constructs, traverses, and discards intermediate data structures — although they play no essential role in a computation. Even in lazy languages like Haskell this is expensive, both slowing execution time and increasing heap requirements.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n xs = case xs of []   -> n
                          z:zs -> c z (foldr c n zs)

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

augment :: (forall b. (a -> b -> b) -> b -> b)
                                     -> [a] -> [a]
augment g xs = g (:) xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs

map :: (a -> b) -> [a] -> [b]
map f xs = build (\ c n -> foldr (c . f) n xs)
```

Figure 1. Combinators and functions for lists

3.1 Short cut fusion for algebraic data types

Fortunately, fusion rules often make it possible to avoid the creation and manipulation of intermediate data structures. The `foldr`/`build` rule [6], for example, capitalises on the uniform production of lists via `build` and the uniform consumption of lists via `foldr` to optimise list-manipulating programs. Intuitively, `foldr c n xs` produces a value by replacing all occurrences of `(:)` in `xs` by `c` and the single occurrence of `[]` in `xs` by `n`. For instance, `foldr (+) 0 xs` sums the (numeric) elements of the list `xs`. The function `build`, on the other hand, takes as input a function `g` providing a type-independent template for constructing "abstract" lists, and produces a corresponding "concrete" list. For example, `build (\c n -> c 3 (c 7 n))` produces the list `[3,7]`. The Haskell definitions of `foldr` and `build`, as well as those of other list-processing functions used in this paper, are given in Figure 1. The recursive combinator `foldr` is standard in the Haskell prelude.

The `foldr`/`build` rule serves as the basis for short cut fusion. It states that, for every closed type `t` and every closed function `g :: forall b. (t -> b -> b) -> b -> b`,

```
foldr c n (build g) = g c n                    (1)
```

Here, type instantiation is performed silently, as in Haskell. When this law, considered as a replacement rule oriented from left to right, is applied to a program, it yields a new program which avoids constructing the intermediate list produced by `build g` and immediately consumed by `foldr c n` in the original. Thus, if `sum` and `map` are defined as in Figure 1, and if `sqr x = x * x`, then

```
sumSqs :: [Int] -> Int
sumSqs xs = sum (map sqr xs)
          = foldr (+) 0
              (build (\c n -> foldr (c . sqr) n xs))
          = (\c n -> foldr (c . sqr) n xs) (+) 0
          = foldr ((+) . sqr) 0 xs
```

No intermediate lists are produced by this version of `sumSqs`.

Transformations such as the above can be generalised to other data structures. It is well-known that every algebraic data type `D` — whose definition appears in, e.g., [8] — has an associated `cata` combinator and an associated `build` combinator. Operationally, the `cata` combinator for an algebraic data type `D` takes as input appropriately typed replacement functions for each of `D`'s constructors

```
cata-E :: (a -> b) -> (Int -> b) ->
            (Ops -> b -> b -> b) -> Expr a - > b
cata-E v l o e = case e of
             Var x -> v x
             Lit i -> l i
             Op op e1 e2 -> o op (cata-E v l o e1)
                                 (cata-E v l o e2)

build-E :: (forall b. (a -> b) -> (Int -> b) ->
              (Ops -> b -> b -> b) -> b) -> Expr a
build-E g = g Var Lit Op

augment-E :: (forall b. (a -> b) -> (Int -> b) ->
               (Ops -> b -> b -> b) -> b) ->
                  (a -> Expr c) -> Expr c
augment-E g v = g v Lit Op
```

Figure 2. Combinators for expressions

and a data element d of D. It replaces all (fully applied) occurrences of D's constructors in d by corresponding applications of their replacement functions. The build combinator for an algebraic data type D takes as input a function g providing a type-independent template for constructing "abstract" data structures from values. It instantiates all (fully applied) occurrences of the abstract constructors which appear in g with corresponding applications of the "concrete" constructors of D. Versions of these combinators and related functions for the arithmetic expression data type of Example 1 appear in Figures 2 and 3. As we will see, the types of augment-E and subst are more general than those in [8] — a benefit arising from our monadic perspective.

Compositions of data structure-consuming and -producing functions defined using the cata and build combinators for an algebraic data type D can be fused via a cata/build rule for D. For example, the rule for the data type Expr t states that, for every closed type t and every closed function g :: forall b. (t -> b) -> (Int -> b) -> (Ops -> b -> b -> b) -> b,

```
      cata-E v l o (build-E g) = g v l o          (2)
```

EXAMPLE 4. *Let* env :: a -> b *be a renaming environment and* e *be an expression. The function*

```
renameAccum :: (a -> b) -> Expr a -> [b]
```

which accumulates variables of renamings of expressions, can be defined modularly as

```
renameAccum env e = accum (map-E env e)
```

Using rule (2) and the definitions in Figure 3 we can derive the following optimised version of renameAccum*:*

```
  renameAccum env e
= cata-E (\x -> [x]) (\i -> []) (\op -> (++))
     (build-E (\v l o -> cata-E (v . env) l o e))
= (\v l o -> cata-E (v . env) l o e)
     (\x -> [x]) (\i -> []) (\op -> (++))
= cata-E ((\x -> [x]) . env) (\i -> [])
                 (\op -> (++)) e
```

Unlike the original expression accum (map-E env e)*, the optimised version of* renameAccum *does not construct the renamed expression but instead accumulates variables "on the fly" while renaming.*

3.2 Short cut fusion for functors

In this section we show that the least fixed point of every functor has an associated cata/build rule and provide clean Haskell

```
accum :: Expr a -> [a]
accum = cata-E (\x -> [x]) (\i -> []) (\op -> (++))

map-E :: (a -> b) -> Expr a -> Expr b
map-E env e = build-E
               (\v l o -> cata-E (v . env) l o e)

subst = (a -> Expr b) -> Expr a -> Expr b
subst env e = augment-E
               (\v l o -> cata-E v l o e) env
```

Figure 3. Functions for expressions

implementations of these rules. This opens the way for an *algebra of fusion*, which allows us to define generic fusion rules which are applicable to any data type, rather than only specific rules for specific data types. Haskell's Functor class, which represents type constructors supporting map functions, is given by

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The function fmap is expected to satisfy two semantic functor laws stating that fmap preserves identities and composition. Like the monad laws, they are enforced by the programmer rather than by the compiler.

Given an arbitrary functor f we can implement its least fixed point and cata and build combinators as follows:

```
newtype M f = Inn {unInn :: f (M f)}

cata-f :: Functor f => (f a -> a) -> M f -> a
cata-f h (Inn k) =  h (fmap (cata-f h) k)

build-f :: Functor f =>
              (forall b. (f b -> b) -> b) -> M f
build-f g = g Inn
```

The definition of the type M f represents in Haskell the standard categorical formulation of the initial algebra/least fixed point of f, while cata-f represents the unique mediating map from the initial algebra of f to any other f-algebra. For a categorical semantics of build and the other combinators introduced in this paper see [3]. By contrast with the various build combinators that have previously been defined for specific data types, the build combinators defined above are entirely generic. Moreover, all previously known definitions of build for specific types are instances of these. We call a type of the form M f for an instance f of the Functor class an *inductive data type*, and we call an element of an inductive data type an *inductive data structure*. By definition, every algebraic data type is an inductive data type.

EXAMPLE 5. *The algebraic data type* Expr a *in Example 1 is* M (E a) *for the functor* E a *defined by*

```
data E a b = Var a | Lit Int | Op Ops b b
```

EXAMPLE 6. *An* interactive input/output computation *[18] is either i) a value of type* a*, ii) an input action, which for* every *input token of type* i *results in a new interactive input/output computation, or iii) an output of an output token of type* o *and a new interactive input/output computation. The algebraic data type*

```
data IntIO i o a = Val a | Inp (i -> IntIO i o a)
                         | Outp (o, IntIO i o a)
```

of such computations is the least fixed point of the functor K i o a *where*

```
data K i o a b = V a | I (i -> b) | O (o,b)
```

We can derive a `build` *combinator for* `K i o a` *by instantiating our generic definition of* `build`.[3] *Writing* `f` *for* `K i o a` *gives*

```
cata-f :: (a -> b) -> ((i -> b) -> b) ->
            ((o,b) -> b) -> IntIO i o a -> b
cata-f v p q k = case k of
               Val x      -> v x
               Inp h      -> p (cata-f v p q . h)
               Outp (y,z) -> q (y, cata-f v p q z)

build-f :: (forall b. (a -> b) -> ((i -> b) -> b)
             -> ((o,b) -> b) -> b) -> IntIO i o a
build-f g = g Val Inp Outp
```

Pleasingly, our generic `cata` and `build` combinators for any functor `f` can be used to eliminate inductive data structures of type `M f` from computations. For *every* functor `f`, and for every closed function `g` of closed type `forall b. (f b -> b) -> b`, we can generalise rules (1) and (2) to the following `cata`/`build` rule for `f`:

```
cata-f h (build-f g) = g h                    (3)
```

In Section 3.1 we saw how the `foldr`/`build` rule can be used to eliminate from `sumSqs` the intermediate list produced by `map` and consumed by `sum`. In Example 4, we saw how the `cata`/`build` rule for expressions can be used to eliminate from `renameAccum` the intermediate expression produced by `map-E` and consumed by `accum`. Since modularly constructed programs often use `cata`s to consume data structures produced by `map`s, it is convenient to derive a generic `cata`/`map` fusion rule that can be instantiated at different types, rather than having to invent a new such rule for each data type. We now show that our `build` combinators make this possible.

A *bifunctor* is a functor in two variables. In Haskell, we have

```
class BiFunctor f where
  bmap :: (a -> b) -> (c -> d) -> f a c -> f b d
```

If `f` is a bifunctor then, for every type `a`, `f a` is a functor, and the type `M (f a)` is sensible. If we define the type constructor `Mu f` by `Mu f a = M (f a)` then, by inlining the definition of `M` in that of `Mu f`, we see that `Mu f` is a functor and its `cata` and `build` combinators can be represented in Haskell as

```
newtype Mu f a = In {unIn :: f a (Mu f a)}

cata-f :: BiFunctor f => (f a c -> c) -> Mu f a -> c
cata-f h (In k) = h (bmap id (cata-f h) k)

build-f :: (forall c. (f a c -> c) -> c) -> Mu f a
build-f g = g In
```

Here, we have written `cata-f` and `build-f` rather than `cata-(f a)` and `build-(f a)`, respectively. Suppressing reference to the type `a` is reasonable because the definitions of the `build` and `cata` combinators for `f a` are uniform in `a`. The function

```
fmap :: (a -> b) -> h a -> h b
```

for a functor `h` can be defined in terms of `cata-f` provided `h a` is uniformly a least fixed point. This is certainly the case when `h` is of the form `Mu f` for some bifunctor `f`, and we have

```
instance BiFunctor f => Functor (Mu f) where
  fmap f xs = build-f
                (\k -> cata-f (k . bmap f id) xs)
```

[3] Here, and at several places below, we must appropriately unbundle type isomorphisms to obtain the desired instantiation. So rather than `cata-f` for `f = K i o a` having type `(K i o a b -> b) -> IntIO i o a -> b`, we take it to have the type given above. Unbundling is done without comment henceforth.

EXAMPLE 7. *If* `f` *is the bifunctor* `E` *from Example 5 then the above instance declaration gives the function* `map-E` *from Figure 3. Using this definition of* `fmap` *we have, for every bifunctor* `f`*, the* `cata`/`map` *fusion rules*

```
  cata-f k (fmap f xs)
= cata-f k (build-f
             (\k -> cata-f (k . bmap f id) xs))
= cata-f (k . bmap f id) xs

fmap f (build g) = build-f
                     (\h -> g (h . bmap f id))
```

The first expression in the first rule above constructs an intermediate data structure via `fmap` *and then immediately consumes it with a call to* `cata-f`*. The optimised final expression avoids this. In the second fusion rule, the right-hand side expression is a call to* `build`*, making further fusions possible. Developing an* algebra of fusion *incorporating generic rules such as these is an exciting possibility.*

4. Augment

The instance of `build-E` used in `map-E` in Figure 3 can be thought of as constructing particularly simple substitution instances of expressions. It replaces data associated with the non-recursive constructor `Var` by new data, but not with arbitrary expressions. As demonstrated above, the process of mapping over an expression in this way and then accumulating variables in the resulting expression is well-suited for optimisation via the `cata`/`build` rule for expressions.

Although it is possible to use `build-E` to construct more general substitution instances of expressions which replace data with arbitrary expressions — and, indeed, to use `build-f` to construct general substitution instances of structures of any inductive data type `M f` — the `build` representations of these more robust substitution instances are inefficient. The problem is that extra consumptions must be introduced to process the subexpressions introduced by the substitution. Unfortunately, subsequent removal of such consumptions via fusion cannot be guaranteed [5].

Suppose, for example, that we want to write a substitution function for expressions of type `Expr a` in terms of `build-E` and `cata-E`. It is tempting to write

```
badSub :: (a -> Expr a) -> Expr a -> Expr a
badSub env e = build-E (\v l o -> cata-E env l o e)
```

but the expression on the right hand side is ill-typed: `env` has type `a -> Expr a`, while `build-E` requires `cata-E`'s replacement for `Var` to be of the more general type `a -> b` for some type variable `b`. The difficulty here is that the constructors in the expressions introduced by `env` are part of the result of `badSub`, but they are not properly abstracted by `build-E`. More generally, the argument `g` to `build-E` must abstract *all* of the concrete constructors that appear in the data structure it produces, not just the top-level ones contributed by `g` itself. To achieve this, extra consumptions using `cata-E` are required:

```
goodSub env e = build-E
  (\v l o -> cata-E ((cata-E v l o) . env) l o e)
```

In the literature, eliminating such extra consumptions has been addressed by the introduction of more general `augment` combinators. The `augment` combinator for lists was introduced in [5] and appears in Figure 1. Analogues for arbitrary algebraic data types are given in [8]; the `augment` combinator given in [8] for the `Expr` data type, for example, is

```
aug-E :: (forall b. (a -> b) -> (Int -> b) ->
          (Op -> b -> b -> b) -> b) ->
             (a -> Expr a) -> Expr a
aug-E g v = g v Lit Op
```

Note that the type of `aug-E` is more restrictive than that of the `augment` combinator `augment-E` developed in this paper, which appears in Figure 2. Using `aug-E` we can express `subst` as

```
subst env e = aug-E (\v l o -> cata-E v l o e) env
```

The `aug-E` combinator offers more than a nice means of expressing substitution, however. When expression-producing functions are written in terms of `aug-E` and are composed with expression-consuming functions written in terms of `cata-E`, a `cata`/`augment` rule generalising the `cata`/`build` rule for expressions can eliminate the intermediate data structure produced by `aug-E`. This fusion rule asserts that, for every closed type `t` and every closed function `g :: forall b. (t -> b) -> (Int -> b) -> (Ops -> b -> b -> b) -> b`,

$$\begin{array}{ll} & \texttt{cata-E v l o (aug-E g f)} \\ = & \texttt{g (cata-E v l o . f) l o} \end{array} \qquad (4)$$

EXAMPLE 8. *First inlining the* `aug-E` *form of* `subst` *above and the* `cata-E` *form of* `accum` *from Figure 3, and then applying the above rule, eliminates the intermediate expression in*

```
substAccum :: (a -> Expr b) -> Expr a -> [b]
substAccum env e = accum (subst env e)
```

to give

```
substAccum env e = cata-E (accum . env) (\i -> [])
                          (\op -> (++)) e
```

This example generalises Example 4 since renaming is a special case of substitution.

Note that `augment` combinators are derived only for algebraic data types in [8]. In Section 5 we generalise the combinators of [8] to give `augment` combinators, and analogues of the `cata`/`augment` rule (4), for non-algebraic inductive data types as well. The precise relationship between our combinators and those of [8] is discussed in Section 4.5 below, where we show how, for algebraic data types, the latter can be derived from the former.

4.1 Introducing monadic augment

We have seen that a `build` combinator can be defined for any functor. A natural question raised by the discussion in the previous section is thus: For how general a range of functors can `augment` combinators be defined?

The essence of `augment` is to extend `build` by allowing data structure-producing functions to take as input additional replacement functions. In [5], the `append` function is the motivating example, and the replacement function argument to the `augment` combinator for lists replaces the empty list occurring at the end of `append`'s first input list with `append`'s second input list. Similar combinators are defined for arbitrary algebraic types in [8]. There, each constructor of an algebraic data type is designated either recursive or non-recursive, and the `augment` combinator for each algebraic data type allows the replacement of data stored at the non-recursive constructors with arbitrary elements of that data type. (See Section 4.5.)

We take a different approach in this paper. We, too, start from the observations that i) each `augment` combinator extends the corresponding `build` combinator with a function which replaces data/values by structures/computations, and ii) the essence of monadic computation is precisely a well-behaved notion of such replacement. But we see these as evidence that the `augment` combinators are inherently monadic in nature. Moreover, as discussed at the end of Section 4.3, the `augment` combinators bear relationships to their corresponding `build` combinators similar to those that the `bind` operations bear to their corresponding `fmap`s. That is, both `build` and `fmap` support the replacement of data by data, while `augment` and `bind` allow the replacement of data by structures. Of course, `augment` and `bind` are defined for monads, while `build` and `fmap` are defined for functors.

This theoretical insight offers practical dividends. As we demonstrate below, it allows us to define more expressive `augment` combinators, and more general `cata`/`augment` rules, than those known before. It also allows us to define `augment` combinators and `cata`/`augment` rules for types for which these were not previously known to exist. We briefly illustrate our results before proceeding with the formal development of the monadic `augment` combinators and their associated fusion rules in the next section.

EXAMPLE 9. *The data type*

```
data Rose a = Node a [Rose a]
```

of rose trees has no non-recursive constructors. The associated `augment` *combinator of [8] therefore does not allow the replacement of data of type* `a` *with rose trees. But we will see in Section 4.3 that* `Rose` *is a monad, and thus that the* `augment` *combinator for* `Rose` *defined in this paper does allow such replacements. In fact, it allows replacements of data of type* `a` *with structures of type* `Rose b` *for any* `b`.

EXAMPLE 10. *The inductive data type*

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

has one non-recursive constructor storing data of type `b`. *The associated* `augment` *combinator of [8] thus supports replacement functions of type* `b -> Tree a b`. *But since* `Tree a` *is also a monad, the* `augment` *combinator defined in this paper supports replacement functions of the more general type* `b -> Tree a c`.

4.2 Parameterised monads

We have argued above that the essence of an `augment` combinator is to extend its corresponding `build` combinator with replacement functions mapping data/values to structures/computations. The types of the structures produced by the `augment` combinators must therefore be of the form `m a` for some monad `m`. But if we want to be able to consume with `cata`s the monadic structures produced by `augment` combinators then we must restrict our attention to those monads `m` for which `cata` combinators can be defined. This is possible provided `m` is an inductive monad.

One way to specify inductive monads uniformly is to focus on monads of the form `m a = Mu f a` for a bifunctor `f`. As we have seen, `Mu f` is a functor. But it is clear that `Mu f` is not, in general, a monad. Indeed, the data type `Tree a b` from Example 10 can be written as `Tree a b = Mu (T b) a` where `data T b a c = N c a c | L b`, but `Tree a b` is not a monad in `a`, i.e., does not admit a substitution function `Tree a b -> (a -> Tree c b) -> Tree c b`. Defining such a function would entail constructing new trees from old ones by replacing each internal node in a given tree by a new tree. Since there is no way to do this, we see that `Tree a b` is an example of a common inductive type which does not support an `augment` combinator. In light of this observation, it is quite satisfying to find weak and elegant conditions on `f` which guarantee that `Mu f` is indeed a monad.

To define these conditions we introduce the notion of a *parameterised monad* [22]. Parameterised monads are represented in Haskell via the following type class:

```
class PMonad f where
  preturn :: a -> f a c
  (>>!)   :: f a c -> (a -> f b c) -> f b c
  pmap    :: (c -> d) -> f a c -> f a d
```

The operations `preturn`, `>>=`, and `pmap` are expected to satisfy the following five parameterised monad laws:

```
>>! preturn = id
(>>! g) . preturn = g
>>! ((>>! g) . j) = (>>! j) . (>>! j)
pmap g . preturn = preturn
pmap g . (>>! j) = (>>! (pmap g . j)) . pmap g
```

Thus a parameterised monad is just a type-indexed family of monads. That is, for each type `c`, the map `f' c` sending a type `a` to `f a c` is the monad whose `return` operation is given by `preturn`, and whose `bind` operation is given by `>>!`. Note how the first three parameterised monad laws ensure this. Moreover, the fact that `f' c` is a monad *uniformly* in `c` is expressed by requiring the operation `pmap` to be such that every map `g :: c -> d` lifts to a map `pmap g` between the monads `f' c` and `f' d`. This is ensured by the last two parameterised monad laws. Intuitively, we think of `>>!` as replacing, according to its second argument, the non-recursive data of type `a` in structures of type `f a c`, and of `pmap` as modifying, according to its first argument, the recursively defined substructures of structures of type `f a c` to give corresponding structures of type `f a d`. As for the monad and functor laws, the compiler does not check that the operations of a parameterised monad satisfy the required semantic conditions. Note that a parameterised monad is a special form of bifunctor with `pmap`, `>>!`, and `preturn` implementing the required `bmap` operation:

```
instance PMonad m => BiFunctor m where
  bmap f g xs = (pmap g xs) >>! (preturn . f)
```

There are many parameterised monads commonly occurring in functional programming. To illustrate, we first show that the expression language `Expr a` is generated by a parameterised monad. We then give three different mechanisms for constructing parameterised monads and, for each such mechanism, give a widely used example of a parameterised monad constructed using that mechanism.

EXAMPLE 11. *We can derive expression monads from parameterised monads as follows. If*

```
data E a b = Var a | Lit Int | Op Ops b b
```

as in Example 5, then `E` *is a parameterised monad with operations given as follows, and* `Expr a = Mu E a`.

```
instance PMonad E where
  preturn              = Var
  Var x >>! h          = h x
  Lit i >>! h          = Lit i
  Op op e1 e2 >>! h    = Op op e1 e2
  pmap g (Var x)       = Var x
  pmap g (Lit i)       = Lit i
  pmap g (Op op e1 e2) = Op op (g e1) (g e2)
```

EXAMPLE 12. *If* `h` *is* any *functor, then the following defines a parameterised monad:*

```
data SumFunc h a b = Val a | Con (h b)

instance Functor h => PMonad (SumFunc h) where
  preturn         = Val
  Val x >>! h     = h x
  Con y >>! h     = Con y
  pmap g (Val x) = Val x
  pmap g (Con y) = Con (fmap g y)
```

The name `SumFunc` *reflects the fact that* `SumFunc h a` *is the sum of the functor* `h` *and the constantly* `a`*-valued functor. The data type* `Expr a` *from Example 1 is (essentially, i.e., ignoring terms induced by the "extra" lifting implicit in the data declaration for* `h b`*)* `Mu (SumFunc h) a` *for*

```
data h b = Lit Int | Op Ops b b
```

The data type `IntIO i o a` *of interactive input/output computations from Example 6 is (essentially)* `Mu (SumFunc h) a` *for* `h = k i o` *and* `data k i o b = I (i -> b) | O (o,b)`.

A parameterised monad of the form `SumFunc h` constructs monads with a tree-like structure in which data is stored at the leaves. We can instead consider monads with a tree-like structure in which data is stored at the nodes, i.e., in the recursive constructors. These are induced by parameterised monads of the form `ProdFunc h a b = Node a (h b)`. Because the `>>!` operation of a parameterised monad must replace (internal) tree nodes with other trees, the branching structure of such trees must form a monoid. We therefore restrict attention to "structure functors" `h` such that, for each type `t`, the type `h t` forms a monoid. This restriction is captured in the following Haskell type class definition:

```
class Functor h => FunctorPlus h where
  zero :: h a
  plus :: h a -> h a -> h a
```

The programmer is expected to verify that the operations `zero` and `plus` form a monoid on `h a`.

EXAMPLE 13. *If* `h` *is an instance of the* `FunctorPlus` *class, then the following defines a parameterised monad:*

```
newtype ProdFunc h a b = Node a (h b)

instance FunctorPlus h => PMonad (ProdFunc h) where
  preturn x          = Node x zero
  Node x t >>! k     = let Node y s = k x
                       in Node y (plus t s)
  pmap g (Node x t) = Node x (fmap g t)
```

A commonly occurring data type which is the least fixed point of a parameterised monad of the form `ProdFunc h` *is the data type of rose trees from Example 9. Indeed, the data type* `Rose` *is* `Mu (ProdFunc [])` *where* `[]` *is the list functor and*

```
instance FunctorPlus [] where
   zero = []
   plus = (++)
```

Our final example of a general mechanism for generating parameterised monads concerns a generalisation of hyperfunctions [10]. Here, we start with a contravariant "structure functor", i.e., with a functor in the class

```
class ContraFunctor f where
   cfmap :: (a -> b) -> f b -> f a
```

EXAMPLE 14. *If* `h` *is a contravariant functor, then the following defines a parameterised monad:*

```
newtype H h a b = H {unH :: h b -> a}

instance ContraFunctor h => PMonad (H h) where
   preturn x    = H (\f -> x)
```

```
H h >>! k    = H (\f -> unH (k (h f)) f)
pmap g (H h) = H (\f -> h (cfmap g f))
```

An example of a data type which arises as the least fixed point of a parameterised monad of the form `H h` *is the data type of hyperfunctions with argument type* `e` *and result type* `a`*:*

```
newtype Hyp e a = Hyp {unHyp :: (Hyp e a -> e) -> a}
```

Indeed, `Hyp e` *is* `Mu (H h)` *for the contravariant functor* `h b = b -> e`*. This example shows that the data types induced by parameterised monads go well beyond those induced by polynomial functors, and include exotic and sophisticated examples which arise in functional programming.*

We now turn our attention to showing that every parameterised monad has an `augment` combinator and an associated `cata`/`augment` fusion rule. This will allow us to show that every least fixed point of a parameterised monad is a monad by writing the required `bind` operation for the least fixed point in terms of the `augment` combinator for the parameterised monad whose least fixed point it is. That this can be done is very important and we will return to it in the next section. We will also show there that we can write the `augment` combinators in terms of their corresponding `bind`s, and thus that the `augment` combinators really are gmonadic in nature.

4.3 Augment for parameterised monads

The central contribution of this paper is the definition, for each parameterised monad `f`, of an `augment` combinator and `cata`/`augment` fusion rule for the monad `Mu f`. Our definition is entirely generic, and extends the definition of the `augment` combinators from [8] to accommodate non-algebraic inductive data types.

If `f` is a parameterised monad then we can define an `augment` combinator for it by

```
augment-f :: PMonad f => (forall c.
      (f a c -> c) -> c) -> (a -> Mu f b) -> Mu f b
augment-f g k = g (In . ( >>! (unIn . k)))
```

Here, `>>! (unIn . k)` is the application of the infix operator `>>!` to its second argument. We can now see clearly that the definition of `augment` is the same as that of `build`, except that it allows an extra input of type `a -> Mu f b` which is used to replace data of type `a` in the structure generated by `g` with structures of type `Mu f b`. Note that `a -> Mu f b` is the type of a Kleisli arrow for what we will see is the *monad* `Mu f`. It is the `augment` combinators' ability to consume Kleisli arrows — mirroring the `bind` operations' ability to do so — that precisely locates `augment` as a monadic concept. Indeed, as we now show, the `bind` operation for `Mu f` can be written in terms of the `augment` combinator for `f`.

We have already observed that if `f` is a bifunctor then `Mu f` is a functor. But if `f` satisfies the stronger criteria on bifunctors necessary to ensure that it is a parameterised monad, then `Mu f` is actually an inductive monad. The relationship between a parameterised monad `f` and the induced monad `Mu f` is captured in the Haskell instance declaration

```
instance PMonad f => Monad (Mu f) where
  return x = In (preturn x)
  x >>= k = augment-f g k where g h = cata-f h x
```

Although not stated explicitly, this instance declaration entails that if `f` satisfies the semantic laws for a parameterised monad, then `Mu f` is guaranteed to satisfy the semantic laws for monads. Moreover, while `Mu f` may support more than one choice of monadic `return` and `bind` operations, this declaration uniquely determines a choice of monadic operations for `Mu f` which respect the structure of the underlying parameterised monad `f`. By analogy with the situation for inductive data types, we call a type of the form `Mu f a` which is induced by a parameterised monad in this way a *parameterised monadic data type*. Further, we call an element of a parameterised monadic data type a *parameterised monadic data structure*.

We now consider the relationship between `augment`, `build`, and `bind`. We have seen above that the `bind` operation for the least fixed point of a parameterised monad can be defined in terms of the associated `augment` combinator. It is also known that the `build` combinators for specific data types can be defined as specialisations of the `augment` combinators for those types, e.g., `build g = augment g []`. Our generic definitions allow us to show that this holds in general. We have, for every parameterised monad `f`:

```
build-f g >>= k = augment-f g k            (5)
```

Setting `k = return` and using the monad laws, we see that `build-f` is definable from `augment-f`. Together with the observation that

```
fmap k = >>= (return . k)
```

the equality (5) shows that the implementation of `build` in terms of `augment` is similar to that of `fmap` in terms of `bind`. But (5) also shows how `augment` combinators can be defined in terms of `bind` operations. The equality (5) is very elegant indeed! In addition, it provides support for our assertion that the `augment` combinators are monadic by demonstrating that they are interdefinable with, and hence are essentially optimisable forms of, the `bind` operations for their associated monads.

4.4 Examples

Examples of the monads and `augment` combinators derived from the parameterised monads `E`, `SumFunc (k i o)`, `ProdFunc []`, and `H h` for `h b = b -> e` from Examples 11 through 14 appear below. In the interest of completeness we give the correspondence between the generic combinators derived from the definition based on parameterised monads and the specific combinators given earlier for the expression language in Example 1. The monadic interpretation of our `augment` combinators makes it possible to generalise those of [8], which allow replacement only of data stored in the non-recursive constructors of data types, to allow replacement of data stored in recursive constructors of data types as well. (See Example 17.) It also makes it possible to go well beyond algebraic data types, as is illustrated in Example 18.

EXAMPLE 15. *If* `E` *is the parameterised monad from Example 11, then the data type induced by* `E` *is the expression monad* `Expr a` *from Example 1, whose* `return` *and* `bind` *operations are defined below. Instantiating the generic derivations of the* `cata`, `build`, *and* `augment` *combinators for* `E` *and then simplifying the results gives the* `cata`, `build`, *and* `augment` *combinators in Figure 2.*

```
return x = In (preturn x) = In (Var x) = Var x
e >>= k  = augment-E g k
              where g h l o = cata-E h l o e
         = g k Lit Op
              where g h l o = cata-E h l o e
         = cata-E k Lit Op e
         = case e of
             Var x       -> k x
             Lit i       -> Lit i
             Op op e1 e2 -> Op op
                              (cata-E k Lit Op e1)
                              (cata-E k Lit Op e2)
```

EXAMPLE 16. *If* `f = SumFunc (k i o)` *is the parameterised monad from Example 12, then the data type induced by* `f` *is (es-*

sentially) that of interactive input/output computations from Example 6. Instantiating the generic derivations of the `cata`*,* `build`*, and* `augment` *combinators for the parameterised monad* `f` *yields the definitions for* `cata-f` *and* `build-f` *from Example 6 and*

```
augment-f :: (forall b. (a -> b) -> ((i -> b) -> b)
                        -> ((o,b) -> b) -> b)
             -> (a -> IntIO i o c) -> IntIO i o c
augment-f g k = g k Inp Outp
```

Using the above definitions, we can also instantiate the generic derivation of the monad operations for `IntIO i o` *from the operations for the underlying parameterised monad* `f`*. This gives*

```
return x    = Val x
intio >>= k = cata-f k Inp Outp intio
```

EXAMPLE 17. *If* `f = ProdFunc []` *is the parameterised monad from Example 13, then the data type induced by* `f` *is that of rose trees from Example 9. Instantiating the generic derivations of the* `cata`*,* `build`*, and* `augment` *combinators for the parameterised monad* `f` *gives*

```
cata-f :: (a -> [b] -> b) -> Rose a -> b
cata-f n (Node x tas) = n x (map (cata-f n) tas)

build-f :: (forall b. (a -> [b] -> b) -> b)
                                    -> Rose a
build-f g = g Node

augment-f :: (forall b. (a -> [b] -> b) -> b)
               -> (a -> Rose c) -> Rose c
augment-f g k = g (\x t -> let Node y s = k x
                           in Node y (t ++ s))
```

The definitions of `cata-f` *and* `build-f` *coincide with those in [15]. Using the above definitions, we can also instantiate the generic derivation of the monad operations for* `Rose a` *from the operations for the underlying parameterised monad* `f`*. This gives*

```
return x = Node x []
t >>= k  = cata-f (\x ts -> let Node y s = k x
                            in Node y (ts ++ s)) t
```

EXAMPLE 18. *If* `f = H h` *with* `h b = b -> e` *is the parameterised monad from Example 14, then the data type induced by* `f` *is the monad of hyperfunctions given there. Instantiating the generic derivations of the* `cata`*,* `build`*, and* `augment` *combinators for the parameterised monad* `f` *gives*

```
cata-f :: ((b -> e) -> a) -> b) -> Hyp e a -> a
cata-f h (Hyp k) = h (\g -> k (g . cata-f h))

build-f :: (forall b. (((b -> e) -> a) -> b) -> b)
             -> Hyp e a
build-f g = g Hyp

augment-f :: (forall b. (((b -> e) -> a) -> b) -> b)
               -> (a -> Hyp e c) -> Hyp e c
augment-f g k = g (\u -> Hyp (\f -> unHyp
                                (k (u f)) f))
```

Using the above definitions, we can also instantiate the generic derivation of the monad operations for `Hyp e a` *from the operations for the underlying parameterised monad* `f`*. This gives*

```
return x      = Hyp (\k -> x)
(Hyp h) >>= k = Hyp (\f -> unHyp
                       (k (h (f . (>>= k)))) f)
```

4.5 Representing algebraic augment

In addition to providing new `augment` combinators for rose trees, as well as `augment` combinators for other types which were not previously known to have them, our results also generalise the `augment` combinators of [8]. At first glance this does not appear to be the case, however, since the `augment` combinators from [8] are derived for all algebraic data types, while the ones in this paper are derived for types of the form `Mu f a` where `f` is a parameterised monad. Surely, one thinks, there are more algebraic types than inductive monads arising as least fixed points of parameterised monads. Put differently, it seems that one can distinguish between recursive and non-recursive constructors, as Johann does, more often than one can distinguish between values and computations, as we do.

The key to resolving this apparent conundrum is the observation that, for each algebraic data type, we can form a parameterised monad by bundling all the non-recursive constructors of the algebraic type together and treating them as values. The `augment` combinator derived from this parameterised monad will allow replacement of all of these values, thereby achieving the expressiveness of Johann's `augment` combinators for the original algebraic data type. Lack of space prevents a full treatment of this observation, but we illustrate with two examples, namely Gill's `augment` combinator for lists and Johann's `augment` combinator for expressions.

The list monad is not of the form `Mu L` for any parameterised monad `L`. However, if we define

```
data L a e b = Var e | Cons a b
```

then, for each type `a`, the type `L a` is a parameterised monad. The data type `Lt a e = Mu (L a) e` can be thought of as representing lists of elements of type `a` that end with elements of type `e`, rather than with the empty list. We therefore have that `[a] = Lt a ()`, where `()` is the one element type. The `augment` combinator for this parameterised monad can take as input a replacement function of type `() -> Lt a ()`, i.e., can take as input another list of type `a`. This gives precisely the functionality of Gill's `augment` combinator for lists. Note the key step of generalising the non-recursive constructor `[]` of lists to variables.

Johann's `augment` combinator for expressions allows the replacement of both variables *and* literals with other expressions. By contrast, our `augment` combinator for the expression data type allows only the replacement of variables with other expressions. However, the same approach we used to derive the standard `augment` combinator for lists works here as well. If we define the parameterised monad

```
data Ex a b = Op op b b | Var a
```

then the type `Expr a` is `Mu Ex (Plus a)` where

```
data Plus a = Left a | Right Int
```

Here, any occurrences of the constructor `Left` can be thought of as the true variables of `Expr a`, while any occurrences of the constructor `Right` can be thought of as its literals.

The `augment` combinator for `Ex` can take as input replacement functions of type `Plus t -> Mu Ex (Plus u)`, which replace both the literals and true variables with expressions of type `Expr u`. This `augment` combinator is actually more general than the one in [8], which forces the type of the variables being replaced to be the same as that of the variables occurring in the replacement expressions. This extra generality, while appearing small, is actually very useful in practice, e.g., in implementing `map` functions using augment. Once again, the key step in the derivation here is the treatment of the non-recursive constructors as variables in the parameterised monad.

Although Johann's `augment` combinators can be derived from our monadic ones, the distinction between recursive/non-recursive constructors may be more intuitive for many programmers than the monadic distinction between values and computations. Of course, when `augment` combinators based on both distinctions are available, the programmer is free to choose between them. But a monadic `augment` may be available even if an algebraic one is not.

5. Generalised short cut fusion

We have seen that parameterised monads are particularly well-behaved, in the sense that their least fixed points are inductive monads which support `cata`, `build`, and `augment` combinators. In this section we give a generic `cata`/`augment` fusion rule which can be specialised for each parameterised monad. The rule we give generalises the `cata`/`augment` rules for lists and expressions discussed in Section 4, as well as the ones in [8].

The rule says that, for each parameterised monad `f`,

```
    cata-f h (augment-f g k)                              (6)
 =  g (h . (>>! (pmap (cata-f h)) . unIn . k))
```

The correctness, and indeed the derivation, of this rule is based on a categorical interpretation of the `augment` combinators which reduces correctness to parametricity; see [4] for details. As with the generic `cata`/`build` rule (3) from Section 3.2, the right-hand side of this rule is an application of the abstract template `g`, but now the extra replacement function `k` must be blended into the algebra `h`.

As we have seen in Section 4.3, the `bind` operation of the least fixed point of a parameterised monad `f` can be defined in terms of the associated `augment` combinator. The possibility of `cata`/`bind` fusion for `Mu f` is therefore hardwired into the very definition of parameterised monadic types. Moreover, since `bind` is the most fundamental of monadic operations, and since data structures uniformly constructed via `bind`s are often uniformly consumed by `cata`s, we expect to see many applications of `bind`s followed by `cata`s in monadic code. The intermediate data structures constructed by such `bind`s and consumed by such `cata`s are eligible for elimination via (6) and, because the `augment` representation of each `bind` is based on a `cata`, the fused optimisation of a `bind` followed by a `cata` will itself be a `cata`. This has the important consequence that not just a single `bind` followed by a `cata`, but in fact a *whole sequence* of `bind`s followed by a `cata`, can be optimised by a series of `cata`/`augment` fusions, each (except the first) enabled by the one that came before. These will ripple backward, allowing monadic code to intermingle and intermediate data structures to be eliminated from computations.

We now illustrate fusion using the generic rule (6). The examples below are natural generalisations of the optimisation of `sumSqs` in Section 3.1, which is typical of the applications found in the literature.

EXAMPLE 19. *To compute the list of free variables appearing in any expression, we can first substitute for each variable node in the expression a new variable node consisting of the singleton list containing the variable name, and then accumulate the contents of these lists by recursively appending them. We have*

```
free-vars   :: Expr a -> [a]
free-vars e = cata-E id (\i -> []) (\op -> (++))
                       (subst (\x -> Var [x]) e)
```

The instantiation of the generic `cata`/`augment` *rule for* `E` *is*

```
  cata-E v l o (augment-E g k)
= g (cata-E v l o . k) l o
```

where `cata-E` *and* `augment-E` *are as in Figure 2. Using this, together with the* `augment` *representation of* `subst` *from Figure 3, we can derive an equivalent version of* `free-vars` *in which the intermediate expression produced by* `subst` *has been eliminated from the modular computation:*

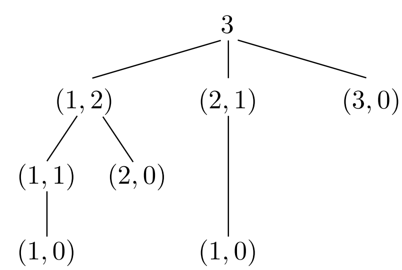
```
  free-vars e
= cata-E id (\i -> []) (\op -> (++)) (augment-E
       (\v l o -> cata-E v l o e) (\x -> Var [x]))
= (\v l o -> cata-E v l o e)
   (\x -> [x]) (\i -> []) (\op -> (++))
= cata-E (\x -> [x]) (\i -> []) (\op -> (++)) e
```

Note that whereas the intermediate expressions in Examples 4 and 8 are of type `Expr a`*, the one in* `free-vars` *has a type of the more general form* `Expr c`*, where* `c` *is taken to be* `[a]`*.*

EXAMPLE 20. *Consider again the monad of interactive input/output computations from Examples 12 and 16. The function* `down` *plays the game in which the user chooses an integer* `n` *and tries to incrementally decrease this number to* `0` *by inputting a number, recording that number as an output, decreasing* `n` *by the input, and playing the game from the result. Let* `f = SumFunc (k i o)` *as in Example 16. Then*

```
down   :: Int -> IntIO Int Int Int
down n = augment-f (\v in out -> let loop x =
                     if x <= 0 then v x else
                      in (\k -> out (k, loop (x-k)))
         in loop n) Val
```

We can represent such a game as a tree with nodes labelled by the last input and the remaining distance to go to zero. The exception is the root node, representing the start of the game, which does not have a preceding input. For example, ignoring the branches which fail by becoming negative, `down 3` *could be represented by*



The function `results` *takes as input a number* `n` *and an interactive input/output computation, and returns the list of values in the leaves of that computation. The user's inputs are assumed to be integers between* `1` *and* `n`*.*

```
results   :: Int -> IntIO Int o a -> [a]
results n = cata-f v in out where
        v x        = [x]
        in g       = concat [g x | x <- [1 .. n]]
        out (o, p) = p
```

The instantiation of the generic `cata`/`augment` *rule for* `f = SumFunc (k i o)` *is*

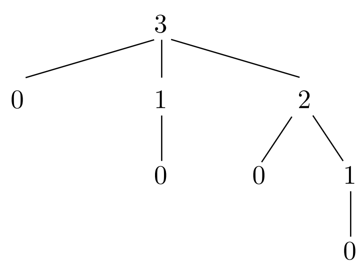
```
  cata-f v in out (augment-f g k)
= g (cata-f v in out . k) in out
```

We can optimise the function which returns the list of values in the leaves of the game tree rooted at `n`*. Since* `v x = [x]`*,* `in g = concat [g x | x <- [1 .. n]]`*, and* `out (o,p) = p`*, we have the following equivalent computation from which the intermediate tree of type* `IntIO Int Int Int` *has been eliminated:*

```
  results n (down n)
```

```
= cata-f v in out
   (augment-f (\v in out ->
     let loop x = if x <= 0 then v x else
                    in (\k -> out (k, loop (x-k)))
     in loop n) Val)
= (\v in out ->
    let loop x = if x <= 0 then v x else
                   in (\k -> out (k, loop (x-k)))
   in loop n) (cata-f v in out . Val) in out
= let loop x = if x <= 0
                then (cata-f v in out . Val) x
                 else in (\k -> out (k, loop (x-k)))
  in loop n
= let loop x = if x <= 0 then [x] else
                in (\k -> loop (x-k))
  in loop n
= let loop x = if x <= 0 then [x] else
                concat [loop (x-z) | z <- [1 .. n]]
  in loop n
```

EXAMPLE 21. *Consider again the monad of rose trees from Examples 13 and 17. The function* `down` *takes a non-negative integer* `n` *as input and produces a rose tree whose root is labelled* `n` *and in which each node has one child for each non-negative integer smaller than its label. For example,* `down 3` *produces*



Letting `f = ProdFunc []` *as in Example 17 we have*

```
return   :: a -> Rose a
return x = Node x []

down   :: Int -> Rose Int
down n = augment-f (\h ->
          let loop x = h x (map loop [0 .. x-1])
          in loop n) return
```

The function `results` *returns the prefix list of data elements in a rose tree:*

```
results :: Rose a -> [a]
results = cata-f (\x ys -> x : concat ys)
```

The instantiation of the generic `cata`/`augment` *rule for* `f = ProdFunc []` *is*

```
  cata-f no (augment g k)
= g (\x t -> let Node y s = k x
             in no y (t ++ map (cata-f no) s))
```

Using this we can optimise the function which returns the prefix list of data elements in the rose tree produced by `down n`*. Letting*

```
no x ys = x : concat ys

g h = let loop y = h y (map loop [0 .. y-1])
      in loop n
```

we have the following equivalent computation from which the intermediate rose tree of integers produced by `down n` *has been eliminated:*

```
  results (down n)
= cata-f no (augment-f g return)
= g (\x t -> let Node y s = return x
             in no y (t ++ map (cata-f no) s))
= g (\x t -> no x) t)
= let loop y =
        (\x t -> no x t) y (map loop [0 .. y-1])
  in loop n
= let loop y = no y (map loop [0 .. y-1]) in loop n
= let loop y = y : concat (map loop [0 .. y-1])
  in loop n
```

EXAMPLE 22. *Rather than give another example in the same vein as previously, we add some variety by establishing the potential for the optimisation of programs which manipulate hyperfunctions by reimplementing the interface for hyperfunctions given in [10]. The original interface was based upon the following operations:*

```
run :: Hyp o o -> o
run (Hyp k) = k run

base :: o -> Hyp i o
base a = Hyp (\x -> a)

(<<) :: (i -> o) -> Hyp i o -> Hyp i o
f << fs = Hyp (\k -> f (k (fs)))
```

We can now reimplement this library using the combinators given in Example 18:

```
run = cata (\c -> c id)

base a = build (\h -> h (\x -> a))

f << fs = build (\h -> h (\k -> f (k (cata h fs))))
```

Correctness of the implementation of `run` *is proved as follows:*

```
run (Hyp k) = cata (\c -> c id) (Hyp k)
            = (\c -> c id) (\g ->
                   k (g . cata (\c -> c id)))
            = (\g -> k (g . cata (\c -> c id))) id
            = k (id . cata (\c -> c id))
            = k (cata (\c -> c id))
            = k run
```

Similar proofs exist for the other combinators. Code written using this interface can now potentially be optimised.

As a final observation, we note that, in the instance declaration for parameterised monadic data types, we could have written the `bind` operation of the monad `Mu f` as

```
x >>= k = cata-f (In . ( >>! (unIn . k))) x
```

rather than in terms of `augment-f`. There are, however, two reasons to not do this. First, this definition of `bind` is significantly less clear than the one involving `augment-f`, and it goes against the practice of abstracting away from programming details via high-level combinators. The second, bigger problem for the purpose of optimisation is that, if a `bind` is followed by a consuming `cata`, then it might not be possible to fuse the `cata` implementing the `bind` with this `cata` since not all compositions of `cata`s can be fused. To get around this difficulty we would be led to devise some kind of strategy for marking those compositions which can be so fused, which would be tantamount to inventing the `augment` combinators.

6. Related work

In addition to the literature on monads and program transformation cited above, there are some additional papers relating to the interaction of these subjects.

- Our work on generic `build` and `augment` combinators contributes to the fruitful line of research into generic recursion combinators. Research in this area has led, for example, to the generalisation of `fold` for lists to arbitrary mixed variance data types [2, 11].
- Like us, Pardo [14] sought to understand fusion in the context of monadic computation, but his goal was different from ours. Pardo investigated conditions under which an expression of type $M(\mu F)$, for M a monad and F a functor with least fixed point μF, can be fused with a function $\texttt{fold}\phi : \mu F \to X$ to produce an expression of type $M(X)$. The crucial difference with our work is that Pardo considered the monad M an ambient structure which was not to be eliminated by the fusion rule. Our goal, on the other hand, is to eliminate the construction of precisely such monadic structures.
- In a similar vein, [12] develops a variety of fusion laws in the monadic setting, including a short cut deforestation law for eliminating intermediate structures of the form M(List X). However, as with [14], the aim is not to eliminate the monad, but rather the list inside the monad.
- Jürgensen [9] defined a fusion combinator based on the uniqueness of the map from a free monad to any other monad. Thus, his technique is really a different form of fusion from ours and, in particular, isn't based upon writing consumers in terms of catamorphisms. Since catamorphisms appear in the literature far more frequently than monad morphisms, it is natural to want as well-developed a theory of catamorphism-based fusion as possible, irrespective of other possibilities such as Jürgensen's.
- Correctness proofs for the fusion rules presented in this paper rely on sophisticated categorical concepts — in particular, strong dinaturality, which, it has been suggested, is unsuitable for a general functional programming and programming transformation audience. Since our aim is to reach precisely such an audience, the correctness proofs of our fusion rules are given in a separate paper [4] which extends the categorical account of `cata`/`build` fusion given in [3].

7. Conclusion and future work

We have defined `build` combinators for all inductive types. In addition, we have demonstrated that `augment` is inherently an inductive and monadic construction, and defined `augment` combinators for inductive monads arising as least fixed points of parameterised monads. We believe it will be difficult to find a more general mechanism for defining inductive monads, and thus that these results are about as general as can be hoped for.

The categorical semantics of [4] reduces correctness of the fusion rules given here to the problem of constructing parametric models which respect the categorical semantics given there. An alternative approach to correctness is taken in [8], where the operational semantics-based parametric model of [17] is used to validate the fusion rules for algebraic data types introduced in that paper. Extending these techniques to tie the correctness of our monadic fusion rules into an operational semantics of the underlying functional language is ongoing work. Benchmarking the rules and developing a preprocessor for automatically converting monadically structured functions into `cata`/`augment` form are additional directions for future work.

Acknowledgments

We thank Graham Hutton and the anonymous reviewers for their comments.

References

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

[2] L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Principles of Programming Languages, Proceedings*, pages 284–194, 1996.

[3] N. Ghani, T. Uustalu, and V. Vene. Build, augment and destroy. Universally. In *Asian Symposium on Programming Languages, Proceedings*, pages 327–347. 2004.

[4] N. Ghani, T. Uustalu, and V. Vene. Generalizing the AUGMENT combinator. In *Trends in Functional Programming 5*, 2005. To appear.

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

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

[7] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving structural hylomorphisms from recursive definitions. In *International Conference on Functional Programming, Proceedings*, pages 73–82. 1996.

[8] P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15:273–300, 2002.

[9] C. Juergensen. Using monads to fuse recursive programs (extended abstract). `citeseer.ist.psu.edu/543861.html`.

[10] J. Launchbury, S. Krstic, and T. Sauerwein. Zip fusion with hyperfunctions. `citeseer.ist.psu.edu/launchbury00zip.html`.

[11] E. Meijer and G. Hutton. Bananas in space: Extending folds and unfolds to exponential types. In *Functional Programming and Computer Architecture, Proceedings*, pages 324–333, 1995.

[12] E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *Advanced Functional Programming, Proceedings*, pages 228–266, 1995.

[13] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[14] A. Pardo. Fusion of recursive programs with computational effects. *Theoretical Computer Science*, 260(1-2):165–207, 2001.

[15] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as an optimization technique in GHC. In *Haskell Workshop, Proceedings*, pages 203–233, 2001.

[16] S. L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[17] A. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:1–397, 2000.

[18] G. Plotkin and J. Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structure, Proceedings*, pages 342–356, 2002.

[19] T. Sheard and L. Fegaras. A fold for all seasons. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 233–242. 1993.

[20] J. Svenningsson. Shortcut fusion for accumulating parameters and zip-like functions. In *International Conference on Functional Programming, Proceedings*, pages 124–132, 2002.

[21] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 306–313, 1995.

[22] T. Uustalu. Generalizing substitution. *Theoretical Informatics and Applications*, 37(4):315–336, 2003.

[23] J. Voigtländer. Concatenate, reverse and map vanish for free. In *International Conference on Functional Programming, Proceedings*, pages 14–25, 2002.

[24] P. Wadler. The essence of functional programming. In *Principles of Programming Languages, Proceedings*, pages 1–14, 1992.