JFP **17** (6): 731–776, 2007.
doi:10.1017/S0956796807006314 First published online 15 March 2007 Printed in the United Kingdom

Monadic augment and generalised short cut fusion

NEIL GHANI
University of Nottingham, Nottingham, NG7 2RD, UK
(*e-mail:* nxg@cs.nott.ac.uk)

PATRICIA JOHANN*
Rutgers University, Camden, NJ 08102, USA
(*e-mail:* pjohann@crab.rutgers.edu)

Abstract

Monads are commonplace programming devices that are used to uniformly structure computations; in particular, they are often used to mimic the effects of impure features such as state, error handling, 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. Ghani, Uustalu, and Vene have recently shown that every inductive type has an associated `build` combinator and an associated short cut fusion law. They have also used the notion of a *parameterised monad* to describe those monads that give rise to inductive types, and have shown that the standard `augment` combinators and `cata`/`augment` fusion rules for algebraic data types can be generalised to fixed points of all parameterised monads. We revisit these `augment` combinators and generalised short cut fusion rules for such types but consider them from a functional programming perspective, rather than a categorical one. In addition to making the category-theoretic ideas of Ghani, Uustalu, and Vene more easily accessible to a wider audience of functional programmers, we demonstrate their practical applicability by developing nontrivial application programs and performing modest benchmarking on them. We also show how the `cata`/`augment` rules can serve as the basis for deriving additional generic fusion laws, thus opening the way for an *algebra of fusion*. Finally, we offer deep theoretical insights, arguing that the `augment` combinators are monadic in nature, and thus that the `cata`/`build` and `cata`/`augment` rules are arguably the best generally applicable fusion rules obtainable.

1 Introduction

As originally conceived by Moggi (1991), *monads* form a useful computational abstraction which can be used to perform such diverse tasks as structuring computations, modeling the effects of impure features such as state, error handling, and I/O in pure languages, and safely separating purely functional code from impure code – all in a modular, uniform, and principled manner. Over a decade ago, Wadler (1992) led the call to turn Moggi's theory of monads into a practical programming methodology.

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

Monads are now firmly established as part of Haskell (Peyton Jones, 2003), supported by specific language features and used in a wide range of applications. The essential idea behind monads is the type-safe separation of *computations*, which might have effects, from *values*, which don't. An alternative view of a monad is as a data type equipped with a well-behaved *generalised substitution* operation, although these two views are easily reconciled by *regarding the effect of a data structure as storing data*. Because monads abstract computational structure – and, in particular, abstract the mechanism for composing computations – 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, it frequently happens that a component in a monadic program will 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 is suitable for this purpose, and a number of fusion transformations appropriate to the non-monadic functional setting have been developed in recent years (Chitil, 1999; Gill *et al*., 1993; Hu *at al*., 1996; Johann, 2002; Jürgensen, 2005; Sheard and Fegaras, 1993; Svenningsson, 2002; Takano and Meijer, 1995; Voigtländer, 2002). Perhaps the best known of these is *short cut fusion* (Gill *et al*., 1993), a local transformation based on two combinators – namely, `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 which manipulate them.

Unfortunately, there are common list producers, such as the append function, that `build` cannot express in a manner which is both efficient and 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 (Gill, 1996). This rule has subsequently been generalised to give `cata/augment` rules[1] which fuse producers and consumers of arbitrary non-list algebraic data types (Johann, 2002). Fusion rules which are dual to the `cata/build` rules in a precise category-theoretic sense (Svenningsson, 2002; Takano and Meijer, 1995), and which eliminate list-manipulating operations other than data constructors (Voigtländer, 2002), have also been developed.

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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

1.1 This paper

This paper describes a further generalisation of short cut fusion to laws which eliminate certain intermediate monadic structures (Ghani *et al*., 2004; Ghani *et al*., 2005). In order to write consumers of expressions of type `m t` in terms of `cata`s, attention is restricted to types `m t` which are inductive types in a uniform manner. A monad `m` with the property that `m t` is an inductive data type for every type `t` is called an *inductive monad*.

In Ghani *et al*. (2004, 2005) it was shown that `build` combinators and `cata`/`build` fusion rules can be defined for *all* inductive types. It is therefore natural to ask whether `augment` combinators and `cata`/`augment` rules can similarly be generically defined. As we observe in Section 4.2, there are inductive types which do not support `augment` combinators, but a large class of inductive monads do. In Ghani *et al*. (2004, 2005), these monads are described using the notion of a *parameterised monad* (Uustalu, 2003), and the observation that the least fixed point of every parameterised monad is an inductive monad is used to define generic `augment` combinators and `cata`/`augment` rules for all such fixed points.

This paper revisits the `augment` combinators and `cata`/`augment` rules derived in Ghani *et al*. (2004, 2005) for fixed points of parameterised monads. But whereas those papers assume the reader has a rather extensive background in category theory, this paper is written from a functional programming perspective, and thus makes no such assumption. Like Ghani, Johann *et al*. (2005), on which it is based, this paper aims to render the category-theoretic ideas of Ghani *et al*. (2004, 2005) more easily accessible to a wider audience of functional programmers. But unlike these papers, it explores the significance of these rules from the point of view of a functional programmer in ways not done in previous papers. For example, it introduces the idea of an *algebra of fusion*, and shows how this idea can be used to derive further generic fusion rules from `cata`/`augment` rules. This paper also constitutes a significant expansion of Ghani, Johann *et al*. (2005) with a substantial addition of expository material to aid the reader. This expansion consists primarily of deeper and more careful explanations of the kind afforded by the journal format, implementations of new nontrivial examples of a more sophisticated nature, and indicative benchmarking demonstrating roughly a 10% gain in program efficiency.

More specifically, this paper illustrates the generic `augment` combinators and `cata`/`augment` rules from Ghani *et al*. (2004, 2005) with expression languages, rose trees, interactive input/output computations, 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 were previously known, the techniques developed in Ghani *et al*. (2004, 2005) yield more expressive `augment` combinators. On the other hand, the examples involving rose trees and interactive input/output computations show that there exist well-known and widely used monads for which neither `augment` combinators nor `cata`/`augment` fusion rules were previously known, but for which both can be derived using these techniques. In addition, since the `bind` operations for monads which are least fixed points of parameterised monads can be written in terms of their `augment` combinators (see

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Section 4.3), the `cata`/`augment` fusion rules from Ghani *et al.* (2004, 2005) can be applied whenever an application of `bind` is followed by a `cata`. In fact, as we show in Section 5, these rules can be used to optimise sequences of `bind`s of the form `(...((m >>= k1) >>= k2)...>>= kn)` whenever they occur. This is expected to be often, since `bind` is the fundamental operation in monadic computation.

In this paper we show, via modest benchmarking, that the results of Ghani *et al.* (2004, 2005) are of practical interest, since `cata`/`augment` rules have the potential to improve the efficiency of modularly constructed programs using a variety of different monads. (See Section 5.1). But these results are of theoretical importance as well: they clearly establish the monadic nature of `augment` combinators by showing that these combinators are interdefinable with the monadic `bind` operations. The facts that `cata`/`build` rules can be defined for *all* functors, and that `cata`/`augment` rules can be defined for *all* least fixed points of parameterised monads, suggest that the results described in this paper are close to the best achievable. We expect, therefore, that these ideas 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, this paper will enable experts in each of these communities to gain greater understanding of, and facility with, the ideas and motivations of the other.

We stress that *the results described in this paper apply to all monads which are fixed points of parameterised monads*. That is, we need not restrict attention to a particular syntactic class of monadic data types in order to obtain our results. We also stress that *the results described in this paper apply only to those monads which are fixed points of parameterised monads*: such fixed points carry a monadic interface which makes it possible to define `augment` combinators, as well as an inductive structure which makes it possible to define `cata` combinators for them. All monads induced by parameterised monads in this way are definable in pure Haskell. Although they are often most naturally viewed as (pure) data structures supporting generalised substitution operators, some, such as the error monad in Example 4 and the interactive I/O monad in Example 5, are more naturally regarded as modeling computations. It is unclear at present whether or not similar results can be developed for monads, such as the state monad or the impure Haskell I/O monad, which do not arise as inductive types.

The rest of this paper is organised as follows:

- In Section 2 we recall the monadic approach to functional programming and give examples of data types which are instances of Haskell's monad class. We also discuss the structuring possibilities afforded by monadic code. This discussion is new relative to Ghani, Johann *et al.* (2005).
- In Section 3 we show how `cata` and `build` combinators, and a `cata`/`build` fusion rule, are derived in Ghani *et al.* (2004, 2005) for the least fixed point of *any* functor. We also introduce the new idea of an *algebra of fusion*, arguing that the generic `cata`/`build` rules make it possible to define additional generic

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

fusion rules which are applicable to any data type. We illustrate this idea by deriving entirely new generic `cata`/`map` and `map`/`build` fusion rules for fixed points of arbitrary bifunctors.

- In Section 4 we recall the notion of a *parameterised monad* and the fact that the least fixed point of *any* parameterised monad is a monad (Uustalu, 2003). We show how the latter fact is used in Ghani *et al.* (2004, 2005) to generalise the standard `augment` combinators for algebraic data types to `augment` combinators for *all* monads arising as least fixed points of parameterised monads. We note that the `augment` combinator for each parameterised monad is shown in Ghani *et al.* (2004, 2005) to be interdefinable with the `bind` operation for the monad which is its least fixed point via the elegant equation

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

  On the basis of this observation, we argue that `augment` combinators are inherently monadic in nature, and thus that the `cata`/`augment` rules are the best general fusion rules obtainable. This realises the import for functional programming of the theoretical work done in Ghani *et al.* (2004, 2005), which goes unremarked upon in those papers.
- In Section 5 we show how the standard `cata`/`augment` fusion rules from (Johann, 2002) for algebraic data types are generalised in Ghani *et al.* (2004, 2005) to give `cata`/`augment` rules for *all* monads arising as least fixed points of parameterised monads. We also detail how the `cata`/`augment` rules can be used to fuse programs of the form `(x >>= f) >>= g`. This discussion is greatly expanded over the passing mention in Ghani *et al.* (2005).
- We demonstrate in Section 5.1 the practical applicability of the `cata`/`augment` fusion rules with a variety of examples of programs over data structures arising as fixed points of parameterised monads, as well as with modest benchmarking. The extensive Examples 22 and 23 here replace the more toy-like Examples 20 and 21 of Ghani, Johann *et al.* (2005). Although they are still far from industrial-strength applications, these more significant examples demonstrate that `cata`/`augment` fusion scales up to programs other than small exercises. We provide Haskell implementations for all concepts and examples in the paper. These are used in our benchmarking, and can be downloaded from `www.cs.nott.ac.uk/~nxg`.

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. We emphasize that no knowledge of category theory is assumed or required and, in order to make this paper accessible to as wide an audience as possible, correctness proofs for all of the fusion rules presented here are given in a separate paper (Ghani *et al.*, 2005), which extends the categorical account of `cata`/`build` fusion given in Ghani *et al.* (2004). On the other hand, this paper is disjoint from Ghani *et al.* (2004, 2005), in that it is addressed to the functional programming community and, as did Wadler (1992) for monads, serves to illustrate the relevance to functional programming of the category-theoretic notion of a parameterised monad.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

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 (possibly) 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. That is, 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 `Err Int` introduced in Example 4 contains error messages, as well as integer values. Similarly, the computational type `IntIO i o a` introduced in Example 5 contains not just values of type `a`, but also functions which transform a token of type `i` into a new interactive input/output computation, and output configurations, each of which comprises an output token of type `o` together with a new interactive input/output computation.

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 following three monad laws (Moggi, 1991):

```
(return x) >>= k   = k x
x >>= return       = x
(x >>= k1) >>= k2  = x >>= (\y -> k1 y >>= k2)
```

These can be thought of as requiring that values act as left and right units for composition, and that the composition of effectful computations be associative. Satisfaction of the 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.

Note that, for any monad `m`, its `bind` operation `>>=` can be thought of as a "generalised substitution" operator which uses its second argument, a function of type `a -> m b`, to replace the data of type `a` in its first argument, a structure of type `m a`, and then applies a "flattening" operation of type `m (m b) -> m b` to the resulting structure of type `m (m b)` to get a new structure of type `m b`. This flattening operation is the multiplication operation in the standard categorical definition of a monad.

Here are some examples of monads and their associated computational types, given in their Haskell form.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Example 1

The built-in Haskell list data type has a natural interpretation as a monad of data structures with the following operations:

```
instance Monad [] where
  return x  = [x]
  xs >>= k = concatMap k xs
```

The function `concatMap`, which on input `k` and `xs` first maps the list-producing function `k` over the elements of `xs` and then concatenates the results, is standard in the Haskell prelude. This definition of `>>=` thus gives a generalised substitution operation for the list monad as described above. Similar remarks apply to each of the monads appearing in this paper.

Since the list monad can also be seen as a computational monad modeling non-determinism, it nicely demonstrates that monads describe data structures as well as effects. Interestingly, however, the list monad does not arise as a fixed point of any parameterised monad, and so the results of this paper do not apply directly to it. For this reason, the list monad is not discussed further in this paper except in Section 4.5 below, where we show how we can indirectly apply to lists the techniques developed in this paper.

Example 2

The data type `Expr a` represents simple arithmetic expressions. It has a natural interpretation as a monad of data structures whose `bind` operation performs substitution on those structures. But it can also be thought of as modeling computations which have the effect of constructing elements of type `Expr a`.

```
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 3

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

```
data Maybe a = Just a | Nothing

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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Example 4
The type `Err a` consists of values of type `a`, as well as string-valued error messages.

```
data Err a = OK a | Failed String


instance Monad Err where
  return         = OK
  Failed s >>= k = Failed s
  OK x     >>= k = k x
```

Example 5
An *interactive input/output computation* (Plotkin and Power, 2002) 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 configuration consisting of an output token of type `o` and a new interactive input/output computation. This is captured in the declaration

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

We will see in Section 4.4 that

```
instance Monad (IntIO i o) where
  return x         = Val x
  Val x >>= k      = k x
  Inp h >>= k      = Inp (\i -> h i >>= k)
  Outp (y,z) >>= k = Outp (y, z >>= k)
```

In fact, these last three equations verify that

```
intio >>= k = cataf k Inp Outp intio
```

where `cataf` is the instantiation for interactive input/output types of the `cata` combinator derived in Section 3.2.

We conclude this section by demonstrating that our results allow the use of monads to systematise, simplify, and highlight the structure of (possibly) effectful programs without forfeiting the possibility of automatic fusion optimisations. Suppose we want to perform a sequence of computations of the form `k2 (k1 x)`, where

```
x  :: m a
k1 :: m a -> m b
k2 :: m b -> m c
```

If we know that `x` is constructed using an `augment` combinator from Johann (2002) for the monad `m`, that `k1` consumes its input expression using the `cata` combinator for `m` and produces its result expression using the standard `augment` combinator for `m`, and that `k2` consumes its input expression using the `cata` combinator for `m`, then we can eliminate the intermediate result of type `m b` using known `cata`/`augment` fusion techniques. Being able to construct monadic structures using `augment` entails

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

that `k1` and `k2` perform computations that can be regarded as a kind of generalised substitution as discussed above. (See the introduction to Section 4 for additional details.)

If we instead use monads to structure the above computation, then the substitutions performed by `k1` and `k2` can be achieved using `bind`. Indeed, if

```
x   :: m a
k1' :: a -> m b
k2' :: b -> m c
```

then we can write `(x >>= k1') >>= k2'` to produce the same result as `k2 (k1 x)`. Here, each `ki'` corresponds to the substitution function `ki`. But whereas each `ki` takes a monadic structure as argument and thus specifies the action of a substitution on an *entire* such structure, the corresponding function `ki'` *need only specify the action of the substitution on data over whose type the monad is parameterised*. This is because traversal of monadic structures is encoded in `bind`, rather than in the substitution functions themselves, as is the case in the non-monadic setting.

The monadic code above is much easier to write: it is simpler to specify actions `k1'` and `k2'` than to write entire substitution functions `k1` and `k2`. Of course, we cannot escape defining the traversal mechanism for the monad `m` entirely, but in the monadic setting we do this exactly once – namely, in the definition of `bind` in the monad class instance for `m` – rather than incorporating the same traversal strategy into the definition of every substitution function individually.

One potential concern when using monads to structure programs is that standard program optimisation techniques might not be applicable. For example, the program `k2 (k1 x)` is subject to standard `cata`/`augment` optimisations, but such optimisations do not obviously apply to `(x >>= k1') >>= k2'`. However, the development of `augment` combinators and `cata`/`augment` rules for a more general class of monads than were handled previously, together with the observation that these `augment` combinators are interdefinable with the monadic `bind` operations, makes it possible to reap the benefits of monadic structuring while simultaneously enabling fusion. (See Section 5 for details.) One could, in fact, say that `augment` is a "fusion-optimised" version of `bind`.

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 will *literally* construct, traverse, and discard intermediate structures – even though 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.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

```
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)
```

Fig. 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 structures. The `foldr`/`build` rule (Gill *et al*., 1993), 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 is the basis for short cut fusion.

Theorem 1 (Gill *et al*., 1993)
For every closed type `t` and 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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

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

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

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

mapE :: (a -> b) -> Expr a -> Expr b
mapE env e = buildE (\v l o -> cataE (v . env) l o e)
```

Fig. 2. Combinators and functions for expressions.

```
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 types. It is well-known that every algebraic data type `D` has an associated `cata` combinator and an associated `build` combinator. An *algebraic data type* is, intuitively, a fixed point of a covariant functor which maps type variables to a type constructed using sum, product, arrow, `forall`, and other algebraic data types defined over those type variables – see Pitts (2000) for a formal definition. Algebraic data types can be parameterised over multiple types, and can be mutually recursive, but not all types definable using Haskell's `data` mechanism are algebraic. Non-algebraic data types include nested types and fixed points of mixed variance functors.

Operationally, the `cata` combinator for an algebraic data type `D` takes as input appropriately typed replacement functions for each of `D`'s constructors 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 2 appear in Figure 2.

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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

cata/build rule for D. For example, the following rule for the data type Expr t is a special case of rule (3) below.

Theorem 2 (Johann, 2002)
For every closed type t and closed function g :: forall b. (t -> b) -> (Int -> b) -> (Ops -> b -> b -> b) -> b,

```
cataE v l o (buildE g) = g v l o                    (2)
```

Example 6
Let env :: a -> b be a renaming environment and let e be an expression. The function

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

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

```
renameAccum env e = accum (mapE env e)
```

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

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

Unlike the original definition accum (mapE env e) of renameAccum env e, the optimised version 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 recall that the least fixed point of every functor has an associated cata/build rule and provide clean Haskell 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 the two semantic functor laws

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

stating that `fmap` preserves identities and composition. Like the monad laws, the functor laws 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 Muu f = Inn {unInn :: f (Muu f)}

cataf :: Functor f => (f a -> a) -> Muu f -> a
cataf h (Inn k) =  h (fmap (cataf h) k)

buildf :: Functor f => (forall b. (f b -> b) -> b) -> Muu f
buildf g = g Inn
```

The definition of the type `Muu f` represents in Haskell the standard categorical formulation of the initial algebra/least fixed point of `f`, while `cataf` 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 Ghani *et al.* (2004, 2005). Generic `cata` combinators go back to Malcolm (1990). But while generic `build` combinators are essentially given in Takano & Meijer (1995), attention is restricted there to "functors whose operation on functions are continuous." By contrast, the `build` combinators defined above are entirely generic over *all* instances of Haskell's functor class. Moreover, all previously known `build` combinators for specific types are instances of these. We call a type of the form `Muu 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 7
The algebraic data type `Expr a` in Example 2 is `Muu (E a)` for the functor `E a` defined by

```
data E a b = V a | L Int | O Ops b b

instance Functor (E a) where
  fmap k (V x)        = V x
  fmap k (L i)        = L i
  fmap k (O op e1 e2) = O op (k e1) (k e2)
```

The combinators `cataE` and `buildE` from Figure 2 can be obtained by first instantiating the above generic definitions of `cataf` and `buildf` for `f = E a`, and then using standard type isomorphisms to unbundle the type arguments to the functor (and guide the case analysis performed by `cataf`). Unbundling allows us to treat the single argument `h :: E a b -> b` to the instantiation of `cataf` as a curried triple of "constructor replacement functions" `v :: a -> b`, `l :: Int -> b`, and `o :: Ops -> b -> b -> b`, and to give these three functions, rather than the isomorphic "bundled" function `h`, as arguments to `cataf`. Unbundling is not in any sense necessary; its sole purpose is to allow the instantiation to take a form more

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

familiar to functional programmers. Similar remarks apply at several places below, and unbundling is performed without comment henceforth.

Instantiating `f` to `E a` we have

```
  cataf h (Inn e)
= h (fmap (cataf h) e)
= h (case e of V x -> V x
               L i -> L i
               O op e1 e2 -> O op (cataf h e1) (cataf h e2))
= case e of V x -> h (V x)
            L i -> h (L i)
            O op e1 e2 -> h (O op (cataf h e1) (cataf h e2))
= case e of V x -> v x
            L i -> l i
            O op e1 e2 -> o op (cataf v l o e1) (cataf v l o e2)
```

Apart from the names of the constructors, this is precisely the definition of `cataE` from Figure 2. We similarly have that

```
buildf g = g Inn
```

Since the type of `g` is `forall b. (E a b -> b) -> b`, the term argument to `g` can be considered a bundled triple of replacement functions `v :: a -> b`, `l :: Int -> b`, and `o :: Ops -> b -> b -> b`. In particular, since the bundled triple representation of `Inn` comprises `Var`, `Lit`, and `Op`, we have

```
buildf g = g Var Lit Op
```

precisely as in Figure 2.

Example 8

The data type `IntIO i o a` of interactive input/output computations in Example 5 is `Muu (K i o a)` for the functor `K i o a` defined by

```
data K i o a b = Vk a | Ik (i -> b) | Ok (o,b)

instance Functor (K i o a) where
  fmap k (Vk x)     = Vk x
  fmap k (Ik h)     = Ik (k . h)
  fmap k (Ok (y,z)) = Ok (o, k z)
```

We can obtain `cata` and `build` combinators for `K i o a` by instantiating the generic definitions of `cataf` and `buildf` for `f = K i o a`. Writing `f` for `K i o a` gives

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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

```
buildf :: (forall b. (a -> b) -> ((i -> b) -> b) ->
                          ((o,b) -> b) -> b) -> IntIO i o a
buildf 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 `Muu f` from computations. For *every* functor `f`, and 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`:

Theorem 3 (Ghani *et al.*, 2004, 2005)
For every functor `f` and every closed function `g` of closed type `forall b.(f b -> b) -> b`,

```
cataf h (buildf g) = g h                                    (3)
```

3.3 An algebra of fusion

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 6, we saw how the `cata`/`build` rule for expressions can be used to eliminate from `renameAccum` the intermediate expression produced by `mapE` 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 the `build` combinators make it possible to derive both `cata`/`map` and `map`/`build` rules, and thus illustrate the "algebra of fusion" idea introduced in the opening paragraph of the previous subsection.

Clearly, we cannot define `cata`/`map` and `map`/`build` rules for arbitrary functors. To see this, note that the key step for doing this is writing the function `fmap :: (a -> b) -> f a -> f b` for the functor `f` in question as `buildf` applied to a function which constructs its body using `cataf`, i.e., as an expression of the form `buildf (\k -> cataf h x)` for some function `h` involving `k`. Unfortunately, this is not possible in general, as even the types of `cataf` and `buildf`, which consume and produce structures of type `Muu f` rather than of `f` itself, suggest.

We can, however, define `cata`/`map` and `map`/`build` rules for functors which arise as fixed points of bifunctors. A *bifunctor* is a functor in two variables. In Haskell, we have

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

with `bmap` satisfying the semantic conditions

```
bmap id id = id
bmap (f . g) (h . k) = bmap f h . bmap g k
```

If `bf` is a bifunctor with mapping function `bmap` then, for every type `a`, `bf a` is a functor with `fmap = bmap id`, and the type `Muu (bf a)` is well-defined. If we

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

define the type constructor `Mu bf` by `Mu bf a = Muu (bf a)` then, by inlining the definition of `Muu` in that of `Mu bf`, we see that `Mu bf` is a functor and its `cata` and `build` combinators can be represented in Haskell as

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

cataMubf :: BiFunctor bf => (bf a c -> c) -> Mu bf a -> c
cataMubf h (In k) = h (bmap id (cataMubf h) k)

buildMubf :: (forall c. (bf a c -> c) -> c) -> Mu bf a
buildMubf g = g In
```

Here, we have written `cataMubf` and `buildMubf` rather than `cata(Mu bf a)` and `build(Mu bf a)`, respectively. Suppressing reference to the type `a` is reasonable because the definitions of the `build` and `cata` combinators for `Mu bf a` are uniform in `a`. We have the following analogue of rule (3) for bifunctors:

Theorem 4 (Ghani *et al.*, 2004, 2005)
For every bifunctor `bf`,

```
cataMubf h (buildMubf g) = g h
```
(4)

If `bf` is a bifunctor, then `Mu bf` is a functor. Indeed, we can define an `fmap` operation with type

```
fmap :: (a -> b) -> Mu bf a -> Mu bf b
```

by

```
instance BiFunctor bf => Functor (Mu bf) where
  fmap f x = buildMubf (\k -> cataMubf (k . bmap f id) x)
```

Note that this definition of `fmap` has the same form as the definitions of `map` and `mapE` in Figures 1 and 2. In fact, those definitions of `map` and `mapE` are instances of this definition of `fmap` for the standard list bifunctor `L a b = N | C a b` and the bifunctor `E` from Example 7, respectively.

Example 9
Using the above definition of `fmap` together with rule (4), we can derive, for every bifunctor `bf` with `f = Mu bf`, the `cata`/`map` and `map`/`build` fusion rules

```
cataMubf k (fmap f x) = cataMubf (k . bmap f id) x
```

and

```
fmap f (buildMubf g) = buildMubf (\k -> g (k . bmap f id))
```

The left-hand side expression in the first rule above constructs an intermediate data structure via `fmap` and then immediately consumes it with a call to `cataMubf`. The optimised final expression avoids this. In the second fusion rule, the right-hand side expression is a call to `buildMubf`, making further fusions possible. Developing an

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

"algebra of fusion" incorporating generic rules such as these is an exciting possibility, since it extends the standard notion of an algebra of programs based around generic `cata`s to include generic `build`s and generalised short cut fusion laws.

Finally, we can use short cut fusion as a potential proof technique. For example, using the above definitions, one can prove that `fmap` obeys the laws of the functor class. In particular, we can see that `fmap` preserves composition as follows:

```
  fmap f (fmap g x)
= fmap f (buildMubf (\h -> cataMubf (h . bmap g id) x))
= buildMubf (\k -> (\h -> cataMubf (h. bmap g id) x) (k. bmap f id))
= buildMubf (\k -> cataMubf (k . bmap f id . bmap g id) x)
= buildMubf (\k -> cataMubf (k . bmap (f . g) id) x)
= fmap (f . g) x
```

4 Augment

The instance of `buildE` used in `mapE` in Figure 2 can be thought of as constructing particularly simple substitution instances of expressions. It replaces data of type `a` associated with the non-recursive constructor `Var` by new data of type `b`, but not with arbitrary expressions of type `Expr b`. Thus, as demonstrated above, the process of mapping over an expression 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 `buildE` to construct more general substitution instances of expressions which replace data (e.g., of type `a` in the above example) with arbitrary expressions (e.g., of type `Expr b`) – and, more generally, to use `build` combinators to construct general substitution instances of structures of inductive data types – 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, as was originally shown in Gill (1996) and as we now illustrate.

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

```
badSub :: (a -> Expr a) -> Expr a -> Expr a
badSub env e = buildE (\v l o -> cataE env l o e)
```

but the expression on the right-hand side is ill-typed: `env` has type `a -> Expr a`, while `buildE` requires `cataE`'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 `buildE`. More generally, the argument `g` to `buildE` 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 `cataE` are required:

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

```
goodSub :: (a -> Expr a) -> Expr a -> Expr a
goodSub env e = buildE (\v l o -> cataE ((cataE v l o) . env) l o e)
```

The function goodSub is well-suited for composition with a consuming cata. But if the expression produced by goodSub is not immediately consumed by a cata, so that no fusion is possible, then as noted above this definition of goodSub is inefficient.

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 Gill (1996) and appears in Figure 1. Analogues for arbitrary algebraic data types are given in Johann (2002); the augment combinator given there for the data type Expr a, for example, is

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

Note that the type of augE is more restrictive than that of the augment combinator augmentE developed in this paper; see Example 17 below. Using augE we can express substitution for expressions as

```
sub :: (a -> Expr a) -> Expr a -> Expr a
sub env e = augE (\v l o -> cataE v l o e) env
```

The augE combinator offers more than a nice means of expressing substitution, however. When expression-producing functions are written in terms of augE and are composed with expression-consuming functions written in terms of cataE, a cata/augment fusion rule generalising the cata/build rule for expressions can eliminate the intermediate data structure produced by augE. This rule states that

Theorem 5 (Johann, 2002)
For every closed type t and every closed function g :: forall b. (t -> b) -> (Int -> b) -> (Ops -> b -> b -> b) -> b,

```
cataE v l o (augE g f)  =  g (cataE v l o . f) l o        (5)
```

Example 10
Inlining the augE form of sub above and the cataE form of accum from Figure 2, and then applying the above rule, eliminates the intermediate expression in

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

to give

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

This example generalises Example 6, since renaming is a special case of substitution.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Note that `augment` combinators are derived only for algebraic data types in Johann (2002). In Section 5 the combinators of Johann (2002) are generalised to give `augment` combinators, and analogues of the `cata`/`augment` rule (5), for non-algebraic inductive data types as well. The precise relationship between these combinators and those of Johann (2002) is discussed in Section 4.5 below, where we show how 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 Gill (1996), 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 the first input list to `(++)` with the second input list. Similar combinators are defined for arbitrary algebraic types in Johann (2002). 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.)

A different approach is taken in Ghani *et al.* (2004, 2005). Those papers start from Johann's observation that each `augment` combinator extends the corresponding `build` combinator with a function which replaces data/values by structures/computations. However, they go further and note that the essence of monadic computation is precisely a well-behaved notion of such replacement. We see these observations 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 `fmaps`. 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 illustrate below, it allows in Ghani *et al.* (2004, 2005) the definition of `augment` combinators and `cata`/`augment` rules for data types for which these combinators and rules were not previously known to exist. It also allows the definition of more expressive `augment` combinators, and more general `cata`/`augment` rules, than those known before. Example 11 gives an example of a data type for which the former is possible, and Example 12 gives an example of a data type for which the latter is possible.

Example 11
The data type

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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

of rose trees has no non-recursive constructors. The associated `augment` combinator of Johann (2002) therefore does not allow the replacement of data of type `a` with rose trees, and so is trivial. But we will see in Section 4.3 that `Rose` is a monad, and thus that the nontrivial `augment` combinator for `Rose` defined in this paper does allow such replacements. In fact, it allows the replacement of data of type `a` with structures of type `Rose b` for any given `b`.

Example 12
The 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 Johann (2002) therefore allows 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 `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 bf a` for a bifunctor `bf`. As we have seen, `Mu bf` is a functor, but it is clear that `Mu bf` is not, in general, a monad. For instance, the data type `Tree a b` from Example 12 can be written as

```
Tree a b = Mu (T b) a
```

where

```
data T b a c = N c a c | L b
```

i.e., as `Mu (T b) a` for the bifunctor `T b`. Yet `Tree a b`, i.e., `Mu (T b) a`, is not a monad in `a`, since it 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 `T b` is an example of a bifunctor whose fixed point is not a monad. On the other hand, `Tree a b` can also be realised as `Mu (T' a) b` for the bifunctor `T' a` given by

```
data T' a b c = N' c a c | L' b
```

Since this fixed point is indeed monadic in `b`, we see that the same data type can be represented as the fixed point of more than one bifunctor. Moreover, one bifunctor for a given type may have a fixed point which is monadic, while another may not. In

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

light of the existence of bifunctors, such as `T b`, whose fixed points are not monadic, it is quite satisfying to find elegant conditions on `bf` which guarantee that `Mu bf` is indeed a monad.

Such conditions can be defined using the notion of a *parameterised monad* (Uustalu, 2003). Parameterised monads are represented in Haskell via the following type class:

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

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

```
>>! preturn = id
(>>! g) . preturn = g
>>! ((>>! g) . j) = (>>! g) . (>>! 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 `pm' c` sending a type `a` to `pm 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 `pm' 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 `pm' c` and `pm' 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 `pm a c`, and of `pmap` as modifying, according to its first argument, the recursively defined substructures of structures of type `pm a c` to give corresponding structures of type `pm 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.

Giving the arguments to `pm` in the order specified in the class declaration for parameterised monads ensures that `Mu pm` is a monad. Changing the order of the arguments to `pm` is, of course, possible, but this would make our definition of a parameterised monad differ from that in Uustalu (2003), and would entail the added complication that `Mu` be redefined to compute the fixed point over the *first* variable of a bifunctor (rather than the second). The latter would make computing with `Mu` counterintuitive, as well as notationally more cumbersome.

Note that a parameterised monad is a special form of bifunctor with `pmap`, `>>!`, and `preturn` implementing the required `bmap` operation:

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

It is not difficult to check that the semantic restrictions associated with the `BiFunctor` class are satisfied.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

There are many parameterised monads commonly occurring in functional programming. To illustrate, we first show that the expression language `Expr a` from Example 2 is generated by a parameterised monad. We then give three different mechanisms for constructing parameterised monads and, for each, give a widely used example of a parameterised monad constructed using that mechanism.

Example 13
We can derive expression monads from parameterised monads as follows. If

```
data E a b = V a | L Int | O Ops b b
```

as in Example 7, then `E` is a parameterised monad with operations given as follows, and `Expr a = Mu E a`.

```
instance PMonad E where
  preturn             = V
  V x >>! h           = h x
  L i >>! h           = L i
  O op e1 e2 >>! h    = O op e1 e2
  pmap g (V x)        = V x
  pmap g (L i)        = L i
  pmap g (O op e1 e2) = O op (g e1) (g e2)
```

These definitions of `preturn`, `>>!`, and `pmap` are easily seen to satisfy the semantic restrictions associated with the `PMonad` class. Similar comments apply for the class instances of `PMonad` appearing below.

Example 14
If `f` is *any* functor, then the following defines a parameterised monad:

```
data SumFunc f a b = ValS a | Con (f b)


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

The name `SumFunc` reflects the fact that `SumFunc f a` is the sum of the functor `f` and the constantly `a`-valued functor. The data type `Expr a` from Example 2 is essentially (i.e., ignoring constructors induced by the "extra" lifting implicit in the data declaration for `F b`) `Mu (SumFunc F) a` for

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

Similarly, the data type `IntIO i o a` of interactive input/output computations from Example 5 is (essentially) `Mu (SumFunc f) a` for `f = K' i o` where

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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

A parameterised monad of the form `SumFunc f` 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 f a b = Node a (f 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" `f` such that, for each type `t`, the type `f t` forms a monoid. This restriction is captured in the following Haskell type class definition:

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

The programmer is expected to verify that the operations `zero` and `plus` form a monoid on `f a`, i.e., that they satisfy the laws

```
plus x zero       = x
plus zero x       = x
plus (plus x n) k = plus x (plus n k)
```

Example 15
The following defines a parameterised monad:

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

instance FunctorPlus f => PMonad (ProdFunc f) 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)
```

Both `plus s t` and `plus t s` are possible in the definition of `>>!` above, and these two choices give rise to different parameterised monads.

A commonly occurring data type which is the least fixed point of a parameterised monad of the form `ProdFunc f` is the data type of rose trees from Example 11. Indeed, the data type `Rose` is `Mu (ProdFunc [])` where `[]` is the standard list functor and

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

The use of `plus t s` in the definition of `>>!` entails that new trees are put to the right of old trees. If `>>!` were instead defined in using `plus s t`, then new trees would be put to the left of old ones.

Our final example of a general mechanism for generating parameterised monads concerns a generalisation of hyperfunctions (Launchbury *et al.*, 2000). Here, we start with a contravariant "structure functor", i.e., with a functor in the class

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

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

Example 16
If `f` is a contravariant functor, then the following defines a parameterised monad:

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

instance ContraFunctor f => PMonad (H f) where
   preturn x     = H (\k -> x)
   H h >>! g     = H (\k -> unH (g (h k)) k)
   pmap g (H h) = H (\k -> h (cfmap g k))
```

An example of a data type which arises as the least fixed point of a parameterised monad of the form `H f` 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 f)` for the contravariant functor `f 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 see 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 `augment` combinators in terms of their corresponding `binds`, and thus that `augment` combinators really are monadic in nature.

4.3 Augment for parameterised monads

In this section we give the definitions from Ghani *et al.* (2004, 2005) of an `augment` combinator for each parameterised monad `bf`, and a `cata/augment` fusion rule for what we will see below is the monad `Mu bf`. These definitions are entirely generic, and extend the definitions of the `augment` combinators from Johann (2002) to non-algebraic inductive data types of the form `Mu bf t`.

Recall that every parameterised monad is a bifunctor. If `bf` is a parameterised monad then an `augment` combinator can be defined for it by

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

Here, `(>>! (unIn . k))` is the right-sectioning of `>>!` with `(unIn . k)`, i.e., is the application of the infix operator `>>!` to its second argument. The function

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

`(In . (>>! (unIn . k)))` thus returns `(In (x >>! (unIn . k))` on input `x`. Recalling that `>>!` acts as a generalised substitution operation (and ignoring the isomporhisms `In` and `unIn`), we see that `(In (x >>! (unIn . k))` essentially replaces, according to `k`, the non-recursive data of type `a` in the structure `x` with structures of type `Mu bf b`. The entire expression `g (In . (>>! (unIn . k)))` thus applies the template `g :: forall c. (bf a c -> c) -> c` for constructing "abstract" parameterised monadic structures to this specific replacement function. Moreover, we can now see that `augmentbf` generalises `buildMubf` to allow an extra input `k :: a -> Mu bf b` which is used to replace data of type `a` in the structure generated by `g` with structures of type `Mu bf b`. Of course, `buildMubf` leaves data of type `a` unchanged. Note that `a -> Mu bf b` is the type of a Kleisli arrow for what we will see below is the *monad* `Mu bf`. 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 bf` can be written in terms of the `augment` combinator for `bf`.

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

```
instance PMonad bf => Monad (Mu bf) where
  return x = In (preturn x)
  x >>= k = augmentbf g k where g h = cataMubf h x
```

Although not stated explicitly, this instance declaration entails that if `bf` satisfies the semantic laws for a parameterised monad, then `Mu bf` is guaranteed to satisfy the semantic laws for monads. Moreover, while `Mu bf` may support more than one choice for monadic `return` and `bind` operations, the above instance declaration uniquely determines a choice of monadic operations for `Mu bf` which respect the structure of the underlying parameterised monad `bf`. By analogy with the situation for inductive data types, we call a type of the form `Mu bf a` which is induced by a parameterised monad `bf` 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 []` for lists. The generic definitions given above allow us to show that this holds in general. Using (4), we have

Theorem 6 (Ghani *et al.*, 2004, 2005)
For every parameterised monad `bf`,

```
buildMubf g >>= k = augmentbf g k
```
(6)

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Setting `k = return` and using the monad laws, we see that `buildMubf` is definable from `augmentbf`, i.e., that

Corollary 1
For every parameterised monad `bf`,

```
buildMubf g = augmentbf g return
```

It is a well-known consequence of the monad laws that, for every monad `m`, the equation `fmap k = >>= (return . k)` holds. Equation (6) thus shows that the implementation of `buildMubf` in terms of `augmentbf` is similar to that of `fmap` in terms of `bind`.

As already noted, equation (6) shows how `augment` combinators for parameterised monads can be defined in terms of the `bind` operations for the monads which are their fixed points. But since the `bind` operations for monads which are fixed points of parameterised monads are defined in terms of the `augment` combinators for those parameterised monads, equation (6) actually establishes that `bind` operations and `augment` combinators are interdefinable. This observation 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. Equation (6) is very elegant indeed!

4.4 Examples

Examples of the monads and `augment` combinators derived from the parameterised monads `E`, `SumFunc (K' i o)`, `ProdFunc []`, and `H f` for `f b = b -> e` from Examples 13 through 16 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 for the expression language in Example 2. The monadic interpretation of our `augment` combinators makes it possible to generalise those of Johann (2002), 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 19). It also makes it possible to go well beyond algebraic data types, as is illustrated in Example 20.

Example 17

If `E` is the parameterised monad from Example 13, then the monad induced by `E` is the expression monad `Expr` from Example 2, whose `return` and `bind` operations are defined below. Instantiating the generic definitions of the `cataMubf` and `buildMubf` combinators for `E`, and then simplifying the results, gives the definitions in Figure 2, while instantiating the generic definition of `augmentbf` and simplifying the result gives

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

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

See Appendix A for details. Using the above definitions, we can also instantiate the generic derivations of the monad operations for `Expr a` from the operations for the underlying parameterised monad `E`. The results coincide with the definitions in Example 2:

```
return x = In (preturn x) = In (V x) = Var x
e >>= k  = augmentE g k where g h l o = cataE h l o e
         = g k Lit Op where g h l o = cataE h l o e
         = cataE k Lit Op e
```

Example 18

If `bf = SumFunc (K' i o)` is the parameterised monad from Example 14, then the monad induced by `bf` is (essentially) the monad `IntIO i o` of interactive input/output computations from Example 8. Instantiating the generic definitions of the `cataMubf`, `buildMubf`, and `augmentbf` combinators for this choice of parameterised monad `bf`, and then simplifying the results, yields the definitions for `cataf` and `buildf` from Example 8, as well as

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

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

```
return x    = Val x
intio >>= k = cataMubf k Inp Outp intio
```

Example 19

If `bf = ProdFunc []` is the parameterised monad from Example 15, then the monad induced by `bf` is that of rose trees from Example 11. Instantiating the generic definitions of the `cataMubf`, `buildMubf`, and `augmentbf` combinators for this choice of parameterised monad `bf`, and then simplifying the results, gives

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

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

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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

As noted above, the appearance of `t ++ s`, rather than `s ++ t`, in the definition of `augmentbf` is forced by the appearance of `plus t s` rather than `plus s t` in the definition of the operation `>>!` for the parameterised monad `bf` given in Example 15. The above `cata` and `build` combinators coincide with those in Peyton Jones *et al.* (2002) and in Johann (2002). But, unlike the `augment` combinator in Johann (2002), the `augment` combinator defined above allows replacement at "inner nodes" of rose trees, i.e., allows replacements of data of type `a` with structures of type `Rose b` for any `b`. The paper by Peyton Jones *et al.* (2001) does not contain an `augment` combinator for rose trees.

Using the above definitions, we can also instantiate the generic derivations of the monad operations for `Rose` from the operations for the underlying parameterised monad `bf`. This gives

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

Example 20

If `bf = H f` with `f b = b -> e` is the parameterised monad from Example 16, then the monad induced by `bf` is that of hyperfunctions given there. Instantiating the generic definitions of the `cataMubf`, `buildMubf`, and `augmentbf` combinators for this choice of parameterised monad `bf`, and then simplifying the results, gives

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

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

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

Using the above definitions, we can also instantiate the generic derivations of the monad operations for `Hyp e` from the operations for the underlying parameterised monad `bf`. 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 (nontrivial) `augment` combinators for rose trees and other types which were not previously known to have them, the results of Ghani *et al.* (2004, 2005) also generalise the `augment` combinators of Johann (2002). At first glance this does not appear to be the case, however, since the `augment` combinators from Johann (2002) are derived for all algebraic data types, while the ones in Ghani *et al.* (2004, 2005) are derived for types of the form `Mu bf a` where `bf` is a

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

parameterised monad. Surely, one thinks, there are more algebraic types than inductive monads arising as least fixed points of parameterised monads.

The key to resolving this apparent conundrum is the observation that, for each algebraic data type, we can form a parameterised monad by bundling together all the data associated with the non-recursive constructors of the algebraic type and treating the result as a value, i.e., as the (single) parameter of a parameterised monad. The `augment` combinator for the parameterised monad obtained in this way will allow replacement of these values, thereby achieving the expressiveness of Johann's `augment` combinators for the original algebraic data type. We do not provide a full treatment of this observation, but instead illustrate it 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 bf` for any parameterised monad `bf`. In particular, although the type `[a]` can be viewed as `Mu L a` for the standard list bifunctor

```
data L a b = N | C a b
```

`L` cannot be endowed with parameterised monadic structure. As a result, the construction captured in the instance declaration in Section 4.3 cannot be used to impose a monadic structure on `Mu L a`. We can, however, see the list monad as a particular instance of a fixed point of a parameterised monad. If we define

```
data L a e b = V e | C a b
```

then, for each type `a`, the type `L a` is a parameterised monad. The data type `L' 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] = L' a ()`, where `()` is the one element type. The `augment` combinator for this parameterised monad can take as input a replacement function of type `() -> L' 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, the `augment` combinator of Ghani *et al.* (2004, 2005) for the expression data type allows only the replacement of variables with other expressions. However, the same approach 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.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

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 Johann (2002), 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. In general, however, it is an open question whether or not it is possible, for an arbitrary given data type, to find a parameterised monad whose fixed point can be specialised to give precisely that type.

Although Johann's augment combinators can be derived from our monadic ones, the distinction between recursive and 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 the generic cata/augment fusion rule from Ghani *et al.* (2004, 2005), and note that it can be specialised for each parameterised monad. This rule generalises the cata/augment rules for lists and expressions discussed in Section 4, as well as the ones in Johann (2002).

Theorem 7 (Ghani *et al.*, 2005)
For every parameterised monad bf,

```
    cataMubf h (augmentbf g k)                                   (7)
=   g (h . ((>>! (pmap (cataMubf 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 Ghani *et al.* (2004, 2005) for details. As with the generic cata/build rules (3) and (4) 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. The argument to g does this by first constructing the function (pmap (cataMubf h)) . unIn . k, and then constructing the right-sectioning of >>! with this function. The result can be understood intuitively as follows. If

```
h :: bf b d -> d
g :: forall c. (bf a c -> c) -> c
k :: a -> Mu bf b
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

so that `augmentbf g k` produces a structure of type `Mu bf b`, then the right-sectioning `(pmap (cataMubf h))` applies `cataMubf h` to the recursively defined substructures of type `Mu bf b` in the result, of type `bf b (Mu bf b)`, returned by `unIn . k`. The right-sectioning `(>>! (pmap (cataMubf h)) . unIn . k)` can thus be thought of as replacing, according to `k`, the nonrecursive data of type `a` in a structure of type `bf a d` with structures of type `Mu bf b`, and then converting the result into a corresponding structure of type `bf b d`. The composition of this function with `h` results in a function which produces from the structure of type `bf a d` the final result of type `d`. In summary, the definition of `augment` shows that, in the left-hand side of (7), the substitution function `k` is mixed with the constructor function `In` to create a structure of type `Mu bf b` and then a `cata` consumes this structure. On the other hand, the optimised right-hand side of (7) acts on the first argument of the parameterised monad `bf` with the `bind` and on the second argument with the `cata` (using `>>!` and `pmap`) to produce a function of type `bf a d -> d` which `g` can consume. The essence of the optimisation is thus the blending of the `bind` in with the `cata`.

As we have seen in Section 4.3, the `bind` operation of the least fixed point of a parameterised monad `bf` can be defined in terms of the associated `augment` combinator. The possibility of `cata`/`bind` fusion for `Mu bf` 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. For instance, patterns such as

```
eval (Add e1 e2) = do x <- eval e1
                      y <- eval e2
                      return (x + y)
```

whose right-hand sides desugar into sequences of `bind`s, are fairly common in monadic evaluators; of course, operations acting on more arguments will give rise to even longer chains of `bind`s. The intermediate data structures constructed by such `bind`s and consumed by such `cata`s are eligible for elimination via (7) 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. Indeed, if `x :: m a`, `k1' :: a -> m b`, and `k2' :: b -> m c`, then as discussed at the end of Section 2 the intermediate structure of type `m b` produced by `x >>! k1'` in `(x >>= k1') >>= k2'` can be eliminated using (7). We have

```
  (x >>= k1') >>= k2'
= augmentbf g2 k2' where g2 h = cataMubf h (x >>= k1')
= augmentbf g2 k2' where g2 h = cataMubf h (augmentbf g1 k1')
                                 where g1 l = cataMubf l x
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

```
= augmentbf g2 k2' where
     g2 h = g1 ((>>! (pmap (cataMubf h)) . unIn . k1')))
              where g1 l = cataMubf l x
= augmentbf g2 k2' where
     g2 h = cataMubf ((>>! (pmap (cataMubf h)) . unIn . k1'))) x
```

The first two steps of this derivation use the definition of `>>=` in terms of `augmentbf` from the instance declaration in Section 4.3, the third step uses (7), and the last step uses the definition of `g1` from the inner `where` clause of the penultimate expression. Fusion can also be performed on sequences `(...((x >>= k1) >>= k2)...>>= kn)` of more than two `binds`. The resulting monolithic code avoids the construction of intermediate structures and so is more efficient, even if less transparent, than its modular equivalent.

We now illustrate fusion for particular data types using the generic rule (7). 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 21

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

```
freeVars   :: Expr a -> [a]
freeVars e=cataE id (\i -> [])(\op -> (++))(subst (\x -> Var [x]) e)
```

The instantiation of the generic `cata/augment` rule for `E` simplifies to

```
cataE v l o (augmentE g k) = g (cataE v l o . k) l o
```

where `cataE` and `augmentE` are as in Example 17. Using this, together with the representation

```
subst :: (a -> Expr b) -> Expr a -> Expr b
subst env e = augmentE (\v l o -> cataE v l o e) env
```

of `subst` in terms of `augmentE`, we can derive an equivalent version of `freeVars` in which the intermediate expression produced by `subst` has been eliminated from the modular computation:

```
  freeVars e
= cataE id (\i -> []) (\op -> (++))
    (augmentE (\v l o -> cataE v l o e) (\x -> Var [x]))
= (\v l o -> cataE v l o e)
     (cataE id (\i -> []) (\op -> (++)) . (\x -> Var [x]))
     (\i -> [])
     (\op -> (++))
= (\v l o -> cataE v l o e) (\x -> [x]) (\i -> []) (\op -> (++))
= cataE (\x -> [x]) (\i -> []) (\op -> (++)) e
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Note that whereas the intermediate expressions in Examples 6 and 10 are of type `Expr b`, the one in `freeVars` has a type of the more general form `Expr c`, where `c` is `[a]`.

Example 22
Consider again the monad of interactive input/output computations from Examples 14 and 18. The instantiation of the generic `cata/augment` rule for interactive input/output computations simplifies to

```
  cataMubf v inn out (augmentbf g k)
= g (cataMubf v inn out . k) inn out
```

where `cataMubf` and `augmentbf` are as in Example 18.

We use an example based on the game of *hangman* to demonstrate how this fusion rule can be used to eliminate intermediate data structures from interactive input/output computations. In the game of hangman there is an unknown word which a player is trying to guess. At each turn, the player guesses a letter. If the letter occurs in the unknown word, then the player is told where all occurrences are. Otherwise, the player loses a life. The game is won if the player guesses all the letters in the word, and it is lost if the player loses eleven lives.

We make a simple model of the game of hangman. More refined models than ours exist, but our goal is to demonstrate fusion rather than make our model as accurate as possible. We model the state of the game as a triple consisting of the unknown word, the letters guessed so far, and the number of lives lost, and we use an interactive input/output computation to represent the possible evolution of a game of hangman. The inputs are characters, the outputs are the resulting states, and the values returned record the current state of the game. Finally, a constant represents the alphabet.

```
type GState = (String, String, Int)
type Game = IntIO Char GState GState

alphabet :: String
alphabet = "abcdefghijklmnopqrstuvwxyz"
```

The function `member :: Eq a => a -> [a] -> Bool` is the standard function which determines whether or not an element of an equality type is a member of a list of such elements. The function `guess` updates a state after a character has been guessed. The functions `won`, `lost`, and `over` determine whether or not the current state indicates that a game has been won or lost, or is over.

```
guess :: Char -> GState -> GState
guess c (w,g,n) = if member c g then (w,g,n+1)
                  else if not (member c w) then (w,c:g,n+1)
                    else (w,c:g,n)

won :: GState -> Bool
won (w,g,n) = and [member l g | l <- w]
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

```
lost :: GState -> Bool
lost (w,g,n) = n >= 11

over :: GState -> Bool
over s = lost s || won s
```

We now turn our attention to our central task, namely constructing a game tree for each game of hangman. Given an initial state of a game, we must construct the new state which arises from outputting that initial state, and then repeatedly inputting a character and outputting the resulting state until the game is over. We use an auxiliary function `turn` to represent one iteration of this loop.

```
turn :: GState -> (GState -> b) -> ((Char -> b) -> b)
                       -> ((GState, b) -> b) -> b
turn s = \v inn out -> out (s, inn (\c -> v (guess c s)))

mkGame :: GState -> Game
mkGame s = if over s then Val s else augmentbf (turn s) mkGame
```

Here, `turn s` is an abstraction of the process of outputting the state `s` and then inputting a character before finishing with the value of the resulting state. The function `mkGame s` repeats this process until the current state indicates that the game is over. Note the essential use of the `augmentbf` combinator with its non-trivial substitution to iterate the turns.

The idea behind `mkGame s` is that, once a game tree is built, it is possible to perform various analyses of it. We may be interested in the space of all possible results in such a tree, in how large the search space of possible plays is, or in how many wins there are in this search space, or in the list of traces showing how a game developed.

To calculate the results of a game tree, for example, we can use `cataMubf` as follows:

```
results :: Game -> [GState]
results = cataMubf v i o where v s     = [s]
                               i f     = concat [f c | c <- alphabet]
                               o (s,n) = n
```

The function `rGame :: GState -> [GState]` calculates the list of results in the game tree which arises from the initial game state to its conclusion. That is, `rGame = results . mkGame`. An optimised version of `rGame` which doesn't construct the intermediate structure of type `Game` can be obtained by `cata`/`augment` fusion for interactive input/output computations:

```
  rGame s
= cataMubf v i o
   (if over s then Val s else augmentbf (turn s) mkGame)
= if over s then cataMubf v i o (Val s)
   else cataMubf v i o (augmentbf (turn s) mkGame)
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

```
= if over s then [s] else turn s (results . mkGame) i o
= if over s then [s] else o (s, i (\c -> rGame (guess c s)))
= if over s then [s] else i (\c -> rGame (guess c s))
= if over s then [s] else concat [rGame (guess c s) | c <- alphabet]
```

Here, the first equation is obtained using the definitions of `results` and `mkGame`, the second by the distribution of `cataMubf` over the conditional, the third by applying `cata/augment` fusion, and the remaining equations are obtained by inlining the definitions of `turn`, `o`, and `i` appearing in the definition of `results`.

If we instead want to calculate the number of possible moves in the search space associated with a game, then we can define `pGame :: GState -> Int` by `pGame = length . results . mkGame`. The following optimised version can be obtained using reasoning similar to that for `rGame`:

```
pGame s = if over s then 1
          else sum [pGame (guess c s)) | c <- alphabet]
```

If we want to calculate the number of winning end states starting from a specific initial game state, then we can use the modular program `wGame :: GState -> Int` defined by

```
wGame = wins . mkGame
```

Here, `wins :: Game -> Int` is given by

```
wins = cataMubf v i o where v s     = if lost s then 0 else 1
                            i f     = sum [f c | c <- alphabet]
                            o (s,n) = n
```

The optimised form of `wGame` is the function `wGame'` defined by

```
wGame' s = if over s then (if lost s then 0 else 1)
             else sum [wGame' (guess c s)) | c <- alphabet]
```

Finally, if we want to calculate the list of traces through a game tree, then we can do this with the function

```
outputs :: Game -> [[GState]]
outputs = cataMubf v i o where v s = [[s]]
                               i f = concat [f c | c <- alphabet]
                               o (s,n) = map (s:) n
```

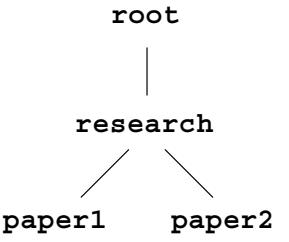
Defining `oGame :: GState -> [[GState]]` to be the function which calculates the list of traces from the evolution of a specific state as `oGame = outputs . mkGame`, then we obtain the following optimised version which does not require the construction of an intermediate game tree:

```
oGame s = if over s then [[s]]
          else map (s:) (concat [oGame (guess c s) | c <- alphabet])
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Over all, the construction of an intermediate game tree facilitates modular programming, since many consumers can naturally be defined as `cata`s. However, this structure can be eliminated to optimise the resulting functions using the `cata/augment` fusion rule for interactive input/output computations.

Example 23
Consider again the monad of rose trees from Examples 15 and 19. Such a tree can be used to represent web pages with links to other web pages. Once again, our intention is not to create a sophisticated model but rather to demonstrate fusion at work. We represent a web page `w` with links to web pages `w1,...,wn` as the rose tree `Node w [w1,...,wn]`. An example of such a tree is



Letting `bf = ProdFunc []` as in Example 19, and taking the combinators and monad operations for rose trees to be as given there, we can define functions `initWeb` and `addWeb` to initialize a web page, and add web pages `ws = [w1,...,wn]` to an existing web page `w`, by

```
type Web = Rose String

initWeb :: Web
initWeb = buildMubf (\n -> n "home" [])

addWeb :: (String, [Web]) -> String -> Web
addWeb (w, ws) = \s -> if s == w then Node w ws else Node s []
```

respectively. We can use these functions to construct the web `web2` depicted above in successive steps by

```
web1 = initWeb >>= addWeb ("home", [Node "research" []])
web2 = web1 >>= addWeb ("research",
                        [Node "paper1" [], Node "paper2" []])
```

The function `pages` returns the prefix list of web pages stored in a rose tree:

```
pages :: Rose a -> [a]
pages = cataMubf (\w ws -> w : concat ws)
```

The instantiation of the generic `cata/augment` rule for rose trees is

```
   cataMubf h (augmentbf g k)
= g (\x t -> let Node y s = k x in h y (t ++ map (cataMubf h) s))
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Using this fusion rule, we can optimise the function which returns the prefix list of web pages in a web after the addition of new pages. Note that the choice of the parameterised monad structure on `bf = ProdFunc []` does not effect the definition of `pages`, but does effect how web pages are added to an existing web.

First note that all directories constructed from `initWeb` using `>>=` can be expressed in terms of `augmentbf`. Indeed,

```
initWeb = augmentbf (\n -> n "home" []) return

newWeb :: Web -> (String, [Web]) -> Web
newWeb oldWeb (page, links) = oldWeb >>= addWeb (page, links)
                            = augmentbf g (addWeb (page, links))
                                   where g f = cataMubf f oldWeb
```

Thus, to get the prefix list of page names in a web structure we can let

```
mkws = \w ws -> w : concat ws
```

and write

```
  results web
= cataMubf mkws (augmentbf g k)
= g (\x t -> let Node y s=k x in mkws y (t ++ map (cataMubf mkws)s))
```

for appropriate `g` and `k`. For example, we can prefix-collect the directory names in `web2` by taking

```
g1 f = cataMubf f initWeb
g2 f = cataMubf f web1

k1 = addWeb ("home", [Node "research" []])
k2 = addWeb ("research", [Node "paper1" [], Node "paper2" []])

foldsub :: (b -> [c] -> c) -> (a -> Rose b) -> a -> [c]-> c
foldsub h k x t = let Node y s = k x in h y (t ++ map (cataMubf h) s)
```

and computing

```
  cataMubf mkws (augmentbf g2 k2)
= g2 (foldsub mkws k2)
= cataMubf (foldsub mkws k2) web1
= cataMubf (foldsub mkws k2) (augmentbf g1 k1)
= g1 (foldsub (foldsub mkws k2) k1)
= cataMubf (foldsub (foldsub mkws k2) k1) initWeb
= cataMubf (foldsub (foldsub mkws k2) k1)
           (augmentbf (\n -> n "home" []) return)
= foldsub (foldsub (foldsub mkws k2) k1) return "home" []
= ["home", "research", "paper1", "paper2"]
```

The first, fourth, and seventh equalities are instances of fusion and the rest are obtained by inlining and standard simplifications.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Example 24

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 Launchnury *et al.* (2000). 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 20:

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

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

f << fs = buildMubf (\h -> h (\k -> f (k (cataMubf h fs))))
```

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

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

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

The `augment` combinator for hyperfunctions turns out to be exactly the bind operation >>= from Example 20 for the monad of hyperfunctions, and acts a kind of "diagonaliser". To see this, recall that inhabitants of type `Hyp i o` can be thought of as streams of functions of type `i -> o` (Launchbury *et al.*, 2000), and note that the stream type has a monadic structure which flattens a stream of streams to the stream whose *nth* element is the *nth* element of the *nth* stream.

As a final observation, we note that, in the instance declaration for parameterised monadic data types in Section 4.3, we could have written the `bind` operation of the monad `Mu bf` as

```
x >>= k = cataMubf (In . ( >>! (unIn . k))) x
```

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

rather than in terms of `augmentbf`. There are, however, two reasons to not do this. First, this definition of `bind` is significantly less clear than the one involving `augmentbf`, 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.

5.1 Measurements

We now demonstrate that the optimisations developed in this paper do indeed improve the efficiency of programs by tracking and comparing the number of cells, reductions, and garbage collections, as well as the execution times for several unfused example programs and their fused counterparts.

We first considered the modular function `wGame :: GState -> Int` and its fused version `wGame' :: GState -> Int`. The definitions of all functions appearing here are given in Example 22. We used an alphabet size of eight and a maximum of five lives. For each word given below, we ran the fused and non-fused functions on the game state consisting of the given word, no initial guesses and no lost lives. We used the February 2000 version of Hugs98, running on a Compusys Economist 865G workstation with 256M RAM and an Intel(R) Pentium(R) 4 CPU running at 2.40GHz, and obtained the following data on reductions and cells used:

Word	`wGame'` reductions(M)/cells(M)/collections	`wGame` reductions(M)/cells(M)/collections
a	0.6 / 0.8 / 3	0.7 / 0.9 / 3
ab	5.3 / 6.7 / 28	5.8 / 7.3 / 30
abc	38 / 46 / 197	41 / 50 / 213
abcd	267 / 325 / 1369	2871 / 350 / 1473

To obtain timings we used Glasgow Haskell Compiler (GHC) version 6.4.1 running on the same workstation. The flags `+s` and `-fallow-undecidable-instances` were set (the latter to handle the monad instance declaration in Section 4.3), and the optimsation level was set to `-O`. Under the same game configuration as above the results were:

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Word	wGame' (seconds)	wGame (seconds)
a	0.08	0.09
ab	0.78	0.85
abc	6.09	6.81
abcd	46	51

These timings are the average over five runs and were obtained using GHC's `getCPUTime` command.

Next, we considered the problem of calculating the number of nodes in a rose tree of integers constructed using `augment` (analogously to the way this combinator is used to construct web pages in Example 23). Specifically, we compared the number of reductions, number of cells, number of garbage collections, and execution times for the modular program

```
nodes :: Int -> Int
nodes = number . mkRose
```

where

```
number :: Rose a -> Int
number = cataRose (\w ws -> 1 + sum ws)

mkRose :: Int -> Rose Int
mkRose n = if n == 0 then Node n []
            else augmentRose
                    (layer n)
                    (\i -> if i >= n then Node n [] else mkRose i)

layer :: Int -> (Int -> [b] -> b) -> b
layer n l = l n [l x [] | x <- [1 .. n-1]]
```

The function call `mkRose n` constructs a tree whose root is labelled with `n`, and which has the property that every node labelled `x` in the tree has as its immediate descendents nodes labelled `1`, ... ,`x-1`.

We then computed the same statistics for its fused version

```
fnodes :: Int -> Int
fnodes n = 1 + sum [fnodes x | x <- [1 .. n-1]]
```

Running the same version of GHC on the same workstation with the same flags set, we obtained the following data:

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

Input size	`fnodes` reductions(M)/cells(M)/collections	`nodes` reductions(M)/cells(M)/collections
16	1.75 / 2.52 / 10	3.03 / 4.65 / 19
18	7.01 / 10.1 / 42	12.1 / 16.6 / 78
20	28.0 / 40.4 / 170	48.5 / 74.4 / 313
22	112 / 161 / 680	193 / 297 / 1255

We also obtained the following timings for these examples:

Input size	`fnodes` (seconds)	`nodes` (seconds)
16	0.05	0.12
18	0.16	0.53
20	0.64	2.08
22	2.54	8.54

These improvements are larger than expected as the result of just fusion alone. In fact, they are a product of the delicate interaction of fusion with inlining. More specifically, inlining can enable fusion rules to fire, and the application of fusion rules can create new opportunities for inlining which can be used to further simplify programs. Both of these effects come into play in deriving the fused program `fnodes`.

Although our benchmarking is neither formal nor extensive, it suggests that our fusion rules do improve programs. Indeed, whether measuring numbers of reductions performed, numbers of cells used, numbers of garbage collections, or time, an improvement of roughly 10% was achieved for each test case.

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.

- The 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 (Sheard and Fegaras, 1996; Meijer and Hutton, 1995). While this paper is concerned with `cata`, `build`, and `augment` combinators for fixed points of functors, structures arising as fixed points of mixed-variance functors – such as $\mu X.Nat + (X \to X)$ – do not lie within its scope.
- Pardo (2001) also 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

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

with least fixed point μF, can be fused with a function `fold` $\phi : \mu F \to X$ to produce an expression of type $M(X)$. The crucial difference with Ghani *et al.* (2004, 2005) is that Pardo considered the monad M an ambient structure which was not to be eliminated by the fusion rule. The goal of Ghani *et al.* (2004, 2005), on the other hand, is to eliminate the construction of precisely such monadic structures.

- In a similar vein, Meijer & Jeuring (1995) develops a variety of fusion laws in the monadic setting, including a short cut fusion law for eliminating intermediate structures of the form M(List X). However, as with Pardo (2001), the aim is not to eliminate the monad, but rather to eliminate the list inside the monad.
- Jürgensen (2005) 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 the one considered here and, in particular, is not 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, we refer the reader to Ghani *et al.* (2005) for correctness proofs for the fusion rules given here. That paper extends the categorical account of `cata`/`build` fusion given in Ghani *et al.*, (2004).

7 Conclusion and future work

In this paper we have recalled the techniques of Ghani *et al.* (2004, 2005) for defining `build` combinators for all inductive types, as well as for defining `augment` combinators for all inductive monads – i.e., for all monads `m` with the property that, for each type `t`, the type `m t` is an inductive type – arising as least fixed points of parameterised monads. We have further demonstrated that `augment` is inherently an inductive and monadic construct, and have seen that monads which arise as least fixed points of parameterised monads give rise to a number of canonical data types used in functional programming. We believe it will be difficult to find a mechanism for defining inductive monads which is more general than taking fixed points of parameterised monads, and therefore conclude that these results are about as general as can be hoped for.

The categorical semantics of Ghani *et al.* (2005) 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 Johann (2002), where the operational semantics-based parametric model of Pitts (2000) 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

in which we aim to show how to extend Pitts' techniques to construct parametric models which accommodate computation types.

We are also interested in developing the techniques required to generalise short cut fusion to other commonly occurring data types in functional programming. We are already writing a paper concerning short cut fusion for nested data types, and have contemplated the possibility of generalising short cut fusion from monads to arrows. The basic program is clear: first consider a general method for constructing instances of the `Arrow` class via fixed points, and then employ the techniques in Ghani *et al.* (2004, 2005) to derive, for each such fixed point, a `build` combinator and an associated fusion rule.

Finally, rigorous benchmarking our fusion rules is an additional direction for future work. This would be greatly facilitated by the development of a preprocessor for automatically converting monadically structured functions into `augment/cata` form, perhaps akin to that of Launchbury & Sheard (1995) for converting (some) recursive programs into `build/cata` forms. However, as is standard practice, we consider the development of such a preprocessor to be a line of research which is independent of the development of the fusion rules themselves.

Acknowledgments

We thank Tarmo Uustalu and Varmo Vene for allowing us to prepare this archival version of our joint ICFP'05 paper. We also thank Graham Hutton and the anonymous reviewers for their helpful comments, and Conor McBride for several useful discussions.

A Instantiating generic combinator definitions

In this appendix we show in detail the instantiation of the generic definitions of the `cataMubf`, `buildMubf`, and `augmentbf` combinators for the parameterised monad `E` from Examples 13 and 17. Instantiations for the other parameterised monads appearing in this paper are similar.

Instantiating the definition of `cataMubf` for `E` gives

```
cataE :: (E a b -> b) -> Expr a -> b
cataE h (In e) = h (bmap id (cataE h) e)
               = h (pmap ((cataE h) e) >>! (V . id))
               = h (pmap ((cataE h) e) >>! V)
               = h (pmap (cataE h) e)
               = h (case e of V x -> V x
                              L i -> L i
                              O op e1 e2 -> O op (cataE h e1)
                                                 (cataE h e2))
```

Now, since `In` and `unIn` give a type isomorphism between `Expr a` and `E a (Expr a)`, we have

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

```
In (V x)        = Var x
In (L i)        = Lit i
In (O op e1 e2) = Op op (In e1) (In e2)
```

and similarly for unIn. We can therefore write the above definition as

```
cataE :: (E a b -> b) -> Expr a -> b
cataE h e = h (case e of Var x -> V x
                         Lit i -> L i
                         Op op e1 e2 -> O op (cataE h e1)
                                             (cataE h e2))
```

Finally, unbundling `E a b` into types `a -> b`, `Int -> b`, and `Op -> b -> b -> b`, unbundling `h :: E a b -> b` into the functions `v :: a -> b`, `l :: Int -> b`, and `o :: Ops -> b -> b -> b`, and then distributing the application of `h` over the `case` expression, we can rewrite the definition for `cataE` one more time to get

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

which is precisely the definition of `cataE` from Figure 2.

Instantiating the definition of `buildMubf` for `E` gives

```
buildE :: (forall b. (E a b -> b) -> b) -> Expr a
buildE g = g In
```

Observing that the unbundled form of the type `In :: E a (Expr a) -> Expr a` comprises the three functions `Var :: a -> Expr a`, `Lit :: Int -> Expr a`, and `Op :: op -> Expr a -> Expr a -> Expr a`, we can rewrite this definition as

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

Instantiating the definition of `augmentbf` for `E` and unbundling the argument to `g` in the final step of the derivation similarly gives

```
augmentE :: (forall c. (E a c -> c) -> c) -> (a -> Expr b) -> Expr b
augmentE g k = g (In . (>>! unIn . k))
             = g (In . \e -> case e of V x -> unIn (k x)
                                       L i -> Lit i
                                       O op e1 e1 -> Op op e1 e2)
             = g (\e -> case e of V x -> k x
                                  L i -> Lit i
                                  O op e1 e1 -> Op op e1 e2)
             = g k Lit Op
```

These definitions of `buildE` and `augmentE` coincide precisely with those in Example 17.

https://doi.org/10.1017/S0956796807006314 Published online by Cambridge University Press

References

Chitil, O. 1999. Type inference builds a short cut to deforestation. In *International Conference on Functional Programming, Proceedings*, ACM Press, pp. 249–260.

Fegaras, L. and Sheard, T. 1996. Revisiting catamorphisms over datatypes with embedded functions. *Principles of Programming Languages, Proceedings*, ACM Press, pp. 284–194.

Ghani, N., Johann, P., Uustalu, T. and Vene, V. 2005. Monadic augment and generalised short cut fusion. In *International Conference on Functional Programming, Proceedings*, ACM Press, pp. 294–305.

Ghani, N., Uustalu, T. and Vene, V. 2004. Build, augment and destroy. Universally. In *Asian Symposium on Programming Languages, Proceedings*, LNCS 3302, Springer-Verlag, pp. 327–347.

Ghani, N., Uustalu, T. and Vene, V. 2005. Generalizing the AUGMENT combinator. In *Trends in Functional Programming 5*, Intellect, pp. 65–78.

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

Gill, A., Launchbury, J. and Peyton Jones, S. L. 1993. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, ACM Press, pp. 223–232.

Hu, Z., Iwasaki, H. and Takeichi, M. 1996. Deriving structural hylomorphisms from recursive definitions. In *International Conference on Functional Programming, Proceedings*, ACM Press, pp. 73–82.

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

Jürgensen, C. 2005. *Categorical Semantics and Composition of Tree Transducers*. PhD thesis, Technische Universität Dresden.

Launchbury, J., Krstic, S. and Sauerwein, T. 2000. Zip fusion with hyperfunctions. Available at `www.cse.ogi.edu/~krstic`.

Launchbury, J. and Sheard, T. 1995. Warm fusion: Deriving build-catas from recursive definitions. In *Functional Programming and Computer Architecture, Proceedings*, ACM Press, pp. 314–323.

Malcolm, G. 1990. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniveriseit Gröningen.

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

Meijer, E. and Jeuring, J. 1995. Merging monads and folds for functional programming. In *Advanced Functional Programming, Proceedings*, LNCS 925, Springer-Verlag, pp. 228–266.

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

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

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

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

Pitts, A. 2000. Parametric polymorphism and operational equivalence. *Mathematical Structures Computer Science* **10**, 1–397.

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

Sheard, T. and Fegaras, L. 1993. A fold for all seasons. In *Functional Programming Languages and Computer Architecture, Proceedings*, ACM Press, pp. 233–242.

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

Takano, A. and Meijer, E. 1995. Shortcut deforestation in calculational form. In *Functional Programming Languages and Computer Architecture, Proceedings*, ACM Press, pp. 306–313.

Uustalu, T. 2003. Generalizing substitution. *Theoretical Informatics & Applications* **37**(4), 315–336.

Voigtländer, J. 2002. Concatenate, reverse and map vanish for free. In *International Conference on Functional Programming, Proceedings*, ACM Press, pp. 14–25.

Wadler, P. 1992. The essence of functional programming. In *Principles of Programming Languages, Proceedings*, ACM Press, pp. 1–14.