Monadic Fold, Monadic Build, Monadic Short Cut Fusion

A research paper by Patricia Johann and Neil Ghani*

Abstract

Short cut fusion improves the efficiency of modularly constructed programs by eliminating intermediate data structures produced by one program component and immediately consumed by another. We define a combinator which expresses uniform production of data structures in monadic contexts, and is the natural counterpart to the well-known monadic `fold` which consumes them. Like the monadic `fold`, our new combinator quantifies over monadic algebras rather than standard ones. Together with the monadic `fold`, it gives rise to a new short cut fusion rule for eliminating intermediate data structures in monadic contexts. This new rule differs significantly from previous short cut fusion rules, all of which are based on combinators which quantify over standard, rather than monadic, algebras. We give examples illustrating the benefits of quantifying over monadic algebras, prove our new fusion rule correct, and show how it can improve programs. We also consider its coalgebraic dual.

1 THE PROBLEM

Consider the following variation on the well-known problem of fusing modular list-processing functions [5]. Suppose we want to sum the cubes of all the integers in a list. Suppose further that cubing any of the integers in the list can generate an out-of-range error, and that each of the partial sums of their cubes can also generate an out-of-range error. If out-of-range errors are tested for using `outOfRange :: Int -> Maybe Int`, where the datatype `data Maybe a = Nothing | Just a` is the standard error-handling monad given in the Haskell prelude, then it is natural to write this program as the composition of two functions: i) a function `mmapcube :: [Int] -> Maybe [Int]` which checks whether the cube of any integer in its input list generates an out-of-range error and, if so, propagates the error, and ii) a function `msum :: [Int] -> Maybe Int` which sums the elements of a list of integers, checking along the way whether each accumulated partial sum generates an out-of-range error. We'd have

```
msumOfCubes xs = mmapcube xs >>= msum
```

where `Nothing >>= k = Nothing` and `Just x >>= k = k x`, as usual.

If we want to optimize this program by eliminating the intermediate structure of type `Maybe [Int]` produced by `mmapcube` and consumed by `\x -> x >>= msum`, then we might look to the standard short cut — i.e., `fold`/`build` — fusion rule [5] for inspiration. Recalling that `>>=` is the analogue of composition for monadic computations, we first observe that the pure analogues `mapcube` and `sum` of the monadic functions `mmapcube` and `msum` above can be written in terms of the well-known uniform list-consuming and -producing functions `foldr` and `build` given in Figure 1. Letting `cube x = x * x * x` we have

```
mapcube :: [Int] -> [Int]
mapcube xs = build (\c n -> foldr (c . cube) n xs)
```

*University of Strathclyde, Glasgow, Scotland, {patricia,ng}@cis.strath.ac.uk.

```
foldr :: (b -> a -> a) -> a -> [b] -> a
foldr c n [] = n
foldr c n (x:xs) = c x (foldr c n xs)

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

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

FIGURE 1. The `foldr` and `build` combinators and `foldr`/`build` rule for lists.

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

The composition of `mapcube` and `sum` gives a non-error-checking version of `mmapcube` which be fused via the `foldr`/`build` rule, also given in Figure 1, as follows:

```
sum (mapcube xs) = foldr (+) 0
                      (build (\c n -> foldr (c . cube) n xs))
                 = (\c n -> foldr (c . cube) n xs) (+) 0
                 = foldr ((+) . cube) 0 xs
```

Note that the intermediate list produced by `mapcube` and immediately consumed by `sum` in `sum (mapcube xs)` is not constructed by the fused program.

In the monadic setting we wish to eliminate not just the intermediate list of cubes, but rather an entire intermediate structure of type `Maybe [Int]`. In general, in the monadic setting we seek to eliminate not just intermediate data structures, but intermediate data structures *within monadic contexts*. In the case of `msumOfCubes` we can write the monadic consumer `msum` in terms of the standard `fold` combinator for lists. Indeed we have

```
msum = foldr (\x p -> do {v <- p; z <- outOfRange x
                          outOfRange (z + v)}) (Just 0)
```

But we cannot similarly write the monadic producer `msumcube` in terms of the standard `build` combinator for lists. The difficulty is that `build` produces a list of type `[t]`, whereas `msumcube` produces a structure of type `Maybe [t]`. It is thus unclear how to write `msumOfCubes` in terms of `fold` and `build` for lists, or how to fuse `msumOfCubes` using standard short cut fusion. And although `fold` and `build` combinators and short cut fusion rules can be defined generically for all inductive types as in Figure 2, they are also unsuitable for fusing `msumOfCubes` because `Maybe [t]` is not an inductive type except in a trivial way.

The central question considered in this paper is whether or not there is a variant of short cut fusion which can fuse modular monadic programs like `msumOfCubes`. We answer in the affirmative by first giving a monadic `build` combinator to complement the monadic `fold` combinator from the literature [1, 10, 11] in the same way that the standard `build` combinator complements the standard `fold` combinator. We then give a monadic short cut rule for fusing monadic compositions (i.e.,

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

fold :: Functor f => (f a -> a) -> Mu f -> a
fold h (In k) = h (fmap (fold h) k)

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

fold k . build g = g k
```

FIGURE 2. The `fold` and `build` combinators and `fold`/`build` rule.

`binds`) of functions written in terms of monadic `fold` and `build`, and show how it solves the `msumOfCubes` problem outlined above. Finally, we prove the correctness of our monadic short cut fusion rule. Our monadic combinators and short cut fusion rule are given in Figure 3 and explained in Section 2.2. As we shall see later, their conceptual basis lies in the observation that, in a monadic setting, it is fruitful to consider not just standard algebras — i.e., pure functions `f a -> a` where `f` is the functor underlying the datatype of the inductive structure to be eliminated "in context" — but rather *monadic algebras*, i.e., functions `f a -> m a` where `m` is the monad in question. Just as the monadic `fold` combinator consumes monadic, rather than standard, algebras, so the monadic `build` combinator introduced herein quantifies over monadic, rather than standard, algebras. Our monadic short cut fusion rule thus eliminates not just intermediate data structures, like `[t]`, but also intermediate structures situated in monadic contexts, like `Maybe [t]`.

The remainder of this paper is structured as follows. In Section 2 we discuss background and related work; in particular we consider the relative merits of structuring code using standard `fold` and monadic `fold`. In Section 3 we apply our monadic fusion rule to a larger application, namely an interpreter for nondeterministic expressions. In Section 4 we prove our main result, the correctness of our monadic fusion rule. In Section 5 we consider its coalgebraic dual. Finally, in Section 6 we set out some directions for future work and conclude.

2 BACKGROUND AND RELATED WORK

2.1 Short Cut Fusion for Inductive Types

Inductive datatypes are fixed points of functors. Functors can be implemented in Haskell as type constructors supporting `fmap` functions as follows:

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

The function `fmap` is expected to satisfy the two semantic functor laws stating that `fmap` preserves identities and composition. It is well-known that analogues of `foldr` exist for every inductive datatype. As shown in [3, 4], every inductive type also has an associated generalized `build` combinator and `fold`/`build` rule; these

```
mfold :: (Functor f, Monad m, Dist f m) =>
              (f a -> m a) -> Mu f -> m a
mfold h = fold (\xs -> delta xs >>= h)

mbuild :: (Functor f, Monad m) => (forall a.
    (f a -> m a) -> c -> m a) -> c -> m (Mu f)
mbuild g = g (return . In)

mbuild g c >>= mfold h = g h c
```

FIGURE 3. The `mfold` and `mbuild` combinators and `mfold`/`mbuild` rule.

can be implemented generically in Haskell as in Figure 2. There, `Mu f` represents the least fixed point of the functor `f`, and `In` represents the structure map for `f`, i.e., the "bundled" constructors for the datatype `Mu f`. The "extra" type `c` in the type of `build` is motivated in citegjuv05,guv03 and to lesser extent in Section 4 below. The `fold`/`build` rule for inductive types can be used to eliminate data structures of type `Mu f` from computations. The `foldr` and `build` combinators for lists can be recovered by taking `f` to be the functor whose fixed point is `[b]`; the `foldr`/`build` rule can be recovered by taking, in addition, `c` to be the unit type.

2.2 Short Cut Fusion in the Presence of Monads

Monads model a variety of computational effects. They are implemented in Haskell as type constructors supporting `>>=` and `return` operations as follows:

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

These operations are expected to satisfy the semantic monad laws [9].

The monadic `fold` combinator `mfold` given in Figure 3 is well-known [1, 10, 11]. It is defined in terms of the standard `fold` and a distributivity law `delta :: f (m a) -> m (f a)` which describes how the results of monadic computations embodied by `m` on components of a data structure described by a functor `f` are propagated to the overall data structure; distributivity laws can be implemented in Haskell using the class `Dist f m` of types, each of which supports a distributivity law for `f` and `m`. But, until now, a monadic `build` which complements the monadic `fold` in the same way that the standard `build` complements the standard `fold` has not been given. Thus, although monadic fusion has been studied by other researchers [8, 10, 11, 12], all techniques developed thus far involve the standard `fold` rather than its monadic counterpart. In particular, a monadic short cut fusion rule – i.e., a short cut rule for programs expressed in terms of monadic `fold` and monadic `build` — has until now not been studied.

To define a monadic `build` combinator we first observe that the functional argument to such a `build`, and thus the monadic `build` combinator itself, must have a monadic return type. Moreover, just as we expect the functions the monadic `fold`

```
mfoldl :: Monad m => (Int -> a -> m a) ->
                          m a -> [Int] -> m a
mfoldl c n = foldr (\ i y -> do {v <- y; c i v}) n

mbuildl :: (forall a. (Int -> a -> m a) -> m a ->
                           c -> m a) -> c -> m [Int]
mbuildl g = g (\ x y -> return (x:y)) (return [])

mbuildl g k >>= mfoldl c n = g c n k
```

FIGURE 4. The `mfoldl` and `mbuildl` combinators and `mfoldl`/`mbuildl` rule.

combinator uses to consume a data structure to have monadic return types, we also expect the monadic `build` combinator to quantify over all uses of functions with monadic return types to consume such structures. This leads us to define the monadic `build` combinator `mbuild` given in Figure 3. Then, to optimize modular programs constructed from these combinators we can also introduce the monadic short cut fusion rule for `mfold` and `mbuild` given there. Unlike the monadic fusion rule of [11], which eliminates the data structure but leaves the monadic context in which it is situated intact, the rule in Figure 3 eliminates both the entire intermediate data structure and its entire monadic context. The correctness of this rule is the main result of this paper. We prove it in Section 4 below.

To demonstrate the monadic combinators and fusion rule, we use them to solve the `msumOfCubes` problem from the introduction. We can specialize the combinators and rule in Figure 3 to `f x = N | C Int x`, `m = Maybe`, and

```
delta N = Just N
delta (C i y) = do {v <- y; Just (C i v)}
```

to get the constructs in Figure 4. We have

```
msum = mfoldl (\x y -> outOfRange (x + y)) (return 0)

mmapcube = mbuildl g where
  g :: (Int -> a -> Maybe a) -> Maybe a -> [Int] -> Maybe a
  g c n []         = n
  g c n (v:vs) = do {k <- outOfRange (cube v); z <- g c n vs;
                     c k z}
```

Applying the monadic fusion rule from Figure 4 to `msumOfCubes` gives

```
msumOfCubes xs = mmapcube xs >>= msum
               = mbuildl g xs >>=
                  mfoldl (\x y -> outOfRange (x + y)) (return 0)
               = g (\x y -> outOfRange (x + y)) (return 0) xs
```

Thus

```
msumOfCubes []     = return 0
msumOfCubes (v:vs) = do {k <- outOfRange (cube v);
                         z <- msumOfCubes vs;
                         outOfRange (k + z)}
```

This fused version performs its range checks "on the fly" rather than prior to summing the cubes of the elements of its input list, and aborts the summing computation if any individual element of `xs` or any of its partial sums is out-of-range.

We now discuss the relative merits of structuring code with `fold` versus with `mfold`. This is critical, since the usefulness of our `mbuild` combinator and `mfold`/`mbuild` fusion rule depends on that of the `mfold` combinator. It turns out that we can write consumers like `msum` in terms of either `mfold` or `fold`. Indeed, every standard algebra `f a -> a` can be turned into a monadic one and, in the presence of distributivity, every monadic algebra can be turned into a standard one. So, in the presence of distributivity, `fold` and `mfold` are equally expressive. Which, then, should we prefer?

Both of these combinators recursively act on the subterms of a term to produce results which are of monadic type. The standard `fold` combinator then uses an algebra of type `f (m a) -> m a` to combine these results into an overall result for the original term. The carrier of the standard `fold`'s algebra is `m a`, which means that the programmer must specify how the monadic contexts generated by the recursive calls will be propagated to the original term. By contrast, the `mfold` combinator is used together with a distributivity law, whose role is precisely to achieve this propagation. The role of the monadic algebra of type `f a -> m a` is thus to describe how *pure* values can be combined and/or generate new effects.

Thus, there is a trade-off between structuring code with standard `fold`s and monadic `fold`s. Using the `mfold` combinator frees the programmer from having to manually propagate the monadic contexts produced by the recursive subcalls. The price paid is having to provide a distributivity law up front. Thus, in the presence of an appropriate distributivity law, programming can proceed as if the recursive calls produced pure, rather than monadic, computations. This seems an excellent trade: it is easier to modularize the problem and supply a monadic algebra than to supply a standard algebra with a monadic carrier. By contrast, using the standard `fold` essentially amounts to programming by hand the plumbing that `mfold` does automatically. Thus, were we to use `fold` rather than `mfold`, we would in effect be hardwiring the definition of `mfold` in terms of `fold` into every algebra supplied to `fold`. Abstracting common patterns of recursion into combinators — as `mfold` does — is widely recognised as key to writing clear, concise, and correct code, so we advocate using the monadic approach whenever possible.

3 APPLICATION: EVALUATING NONDETERMINISTIC EXPRESSIONS

Consider the following datatype of nondeterministic expressions:

```
data NExp = NVar Char | NNum Int | NAdd NExp NExp
          | NSet Char NExp | NOr NExp NExp
```

The first three clauses of this definition represent variables, integers, and sums. Expressions of the form `NSet c e` assign the value of `e` to the variable with name `c`, and those of the form `NOr e1 e2` represent the nondeterministic choice of `e1`

or `e2`. An evaluator for nondeterministic expressions must keep track of both the environment with respect to which the expression is evaluated and the nondeterminism introduced by the `NOr` constructor. Computations involving environments can be modeled by the state monad, while those involving nondeterminism can be modeled by the list monad. The canonical way to model computations involving both is to apply the state monad transformer to the list monad. This gives

```
newtype NState s a = NSt {runNState :: s -> [(a,s)]}

instance Monad (NState s) where
   return x    = NSt (\s -> [(x,s)])
   NSt f >>= g = NSt (\s ->
      concat [runNState (g v) s' | (v, s') <- f s])
```

The type of an evaluator `evalNExp` for nondeterministic expressions is `evalNExp :: NExp -> NState Env Int`, where `type Env = Char -> Int` defines a type of environments mapping variable names to integer values. We construct `evalNExp` as a composition of two functions. The first one, called `compile`, takes as input a nondeterministic expression and returns a list of expressions not containing the `NOr` constructor; we say that such expressions are *deterministic*. Rather than using `NExp` to represent both determinstic expressions and nondeterministic ones — thereby leaving the determinism constraint implicit at the meta-level — we introduce an object-level datatype to represent deterministic expressions. We have

```
data DExp = DVar Char | DNum Int
          | DAdd DExp DExp | DSet Char DExp
```

Thus `compile` has type `NExp -> NState Env DExp`. The second function used to construct `evalNExp` is an evaluator `evalDExp :: DExp -> NState Env Int` for deterministic expressions. In essence, the modular approach reduces the problem of constructing an evaulator for nondeterministic expressions to the problem of constructing one for deterministic expressions. We have

```
evalNExp e = compile e >>= evalDExp
```

We can write `evalDExp` using the instance of `mfold` for `DExp`. We have

```
evalDExp = mfoldDExp fetch return (\i j -> return (i+j)) update
```

where

```
fetch :: Char -> NState Env Int
fetch c = NSt (\env -> [(env c, env)])
```

looks up the value of a variable in the current environment and

```
update :: Char -> Int -> NState Env Int
update c i = NSt (\e -> [(i, \c' -> if c == c' then i else e c')])
```

makes a new environment which is obtained from the current one by updating the binding for the variable `c` to `i`. In addition,

```
mfoldDExp ::  Monad m => (Char -> m a) -> (Int -> m a) ->
               (a -> a -> m a) -> (Char -> a -> m a) ->
                 DExp -> m a
mfoldDExp v n a s = foldDExp v n
                     (\e1 e2 -> do {x1 <- e1; x2 <- e2; a x1 x2})
                      (\i e -> do {x <- e; s i x})
```

is defined in terms of the instance of the standard `fold` combinator for `DExp`:

```
foldDExp :: (Char -> a) -> (Int -> a) -> (a -> a -> a) ->
              (Char -> a -> a) -> DExp -> a
foldDExp v n a s (DVar x)     = v x
foldDExp v n a s (DNum i)     = n i
foldDExp v n a s (DAdd e1 e2) = a (foldDExp v n a s e1)
                                  (foldDExp v n a s e2)
foldDExp v n a s (DSet x e)   = s x (foldDExp v n a s e)
```

This definition of `mfoldDExp` in terms of `foldDExp` is the result of instantiating the definition of `mfold` in terms of `fold` for the datatype `DExp`. Of course, we could define `evalDExp` in terms of `foldDExp`. This would require manually extracting the results of evaluating subterms from the monad before combining them to produce the result for an entire term, rather than hiding the extraction within the `mfoldDExp` combinator. For example, using `fold` would require a function `add :: NState Env Int -> NState Env Int -> NState Env Int` to interpret the `DAdd` constructor, whereas using `mfold` requires only an interpreting function for `DAdd` with type `Int -> Int -> NState Env Int`. This is a specific instance of the general phenomenon described at the end of Section 2.

Finally, the function `compile` is written in terms of `mbuildDExp` as follows:

```
compile = mbuildDExp g where
  g v n a s (NVar x) = v x
  g v n a s (NNum i) = n i
  g v n a s (NAdd e1 e2) = do {x1 <- g v n a s e1;
                               x2 <- g v n a s e2;
                               a x1 x2}
  g v n a s (NSet x e) = do {z <- g v n a s e; s x z}
  g v n a s (NOr e1 e2) = njoin (g v n a s e1) (g v n a s e2)

njoin :: NState s a -> NState s a -> NState s a
njoin (NSt f) (NSt g) = NSt (\s -> f s ++ g s)
```

Here, `mbuildDExp` is the instantiation of `mbuild` for `DExp`:

```
mbuildDExp :: Monad m =>
  (forall a. (Char -> m a) -> (Int -> m a) -> (a -> a -> m a)
         -> (Char -> a -> m a) -> c -> m a) -> c -> m DExp
mbuildDExp g = g (return . DVar) (return . DNum)
                 (return . DAdd) (return . DSet)
```

We can optimize the modular function `evalNExp` using the instantiation of the monadic fusion rule for the monad `NState` and the functor whose fixed point is `DExp`. The instantiation is

```
mbuildDExp g x >>= mfoldDExp v n a s = g v n a s x
```

Fusing with this rule gives

```
   evalNExp e
=  compile e >>= evalDExp
=  mbuildDExp g e >>= mfoldDExp fetch return
                        (\i j -> return (i+j)) update
= g fetch return (\i j -> return (i+j)) update e
= case e of
    NVar x     -> fetch x
    NNum i     -> return i
    NAdd e1 e2 -> do {x1 <- evalNExp e1; x2 <- evalNExp e2;
                      return (x1 + x2)}
    NSet x e   -> do {z <- evalNExp e; update x z}
    NOr e1 e2  -> njoin (evalNExp e1) (evalNExp e2)
```

This fused version of `evalNExp` does not construct the intermediate structure of type `NState Env DExp` produced by `mbuildDExp` and consumed by `mfoldDExp`.

4 CORRECTNESS

4.1 Categorical Preliminaries

Let $\mathscr{C}$ be a category and F be an endofunctor on $\mathscr{C}$. An *F-algebra* is a morphism $h : FA \to A$ in $\mathscr{C}$. The object A is called the *carrier* of the F-algebra. The F-algebras for a functor F are objects of a category called the *category of F-algebras* and denoted F-$\mathscr{A}lg$. A morphism from $h : FA \to A$ to $g : FB \to B$ in F-$\mathscr{A}lg$ is a morphism $f : A \to B$ such that $g \circ Ff = f \circ h$. We call such a morphism an *F-algebra morphism*. If the category of F-algebras has an initial object then Lambek's Lemma ensures that this *initial F-algebra* is an isomorphism, and thus that its carrier is a fixed point of F. Initiality ensures that the carrier of the initial F-algebra is actually a *least* fixed point of F. If it exists, the least fixed point for F is unique up to isomorphism. Henceforth we write μF for the least fixed point for F and $in : F(\mu F) \to \mu F$ for the initial F-algebra.

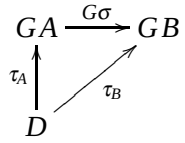
Within the paradigm of initial algebra semantics, every datatype is the carrier μF of the initial algebra of a suitable endofunctor F on a suitable category $\mathscr{C}$. The unique F-algebra morphism from *in* to any other F-algebra $h : FA \to A$ is given by the interpretation *fold* of the `fold` combinator for the interpretation μF of the datatype `Mu F`. The *fold* combinator for μF thus makes the following commute:

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(fold\,h)} & FA \\ {\scriptstyle in}\downarrow & & \downarrow{\scriptstyle h} \\ \mu F & \xrightarrow{fold\,h} & A \end{array}$$

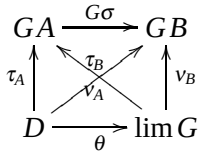
From this diagram, we see that *fold* has type $(FA \to A) \to \mu F \to A$ and that *fold h* satisfies $fold\ h\ (in\ t) = h\ (F\ (fold\ h)\ t)$. The uniqueness of the mediating map

between *in* and h ensures that, for every F-algebra h, the map $\mathit{fold}\,h$ is defined uniquely.

As shown in [4], the carrier of the initial algebra of an endofunctor F on $\mathscr{C}$ can be seen not only as the carrier of the initial F-algebra, but also as the limit of the forgetful functor $U_F : F\text{-}\mathscr{A}lg \to \mathscr{C}$ mapping each F-algebra $h : FA \to A$ to A and each morphism between F-algebras to itself. If $G : \mathscr{C} \to \mathscr{D}$ is a functor, then a *cone* $\tau : D \to G$ to the base G with vertex D is an object D of $\mathscr{D}$ and a family of morphisms $\tau_C : D \to GC$, one for every object C of $\mathscr{C}$, such that for every arrow $\sigma : A \to B$ in $\mathscr{C}$, $\tau_B = G\sigma \circ \tau_A$ holds.

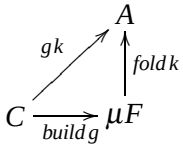


We usually refer to a cone simply by its family of morphisms, rather than the pair comprising the vertex together with the family of morphisms. A *limit* for $G : \mathscr{C} \to \mathscr{D}$ is an object $\lim G$ of $\mathscr{D}$ and a limiting cone $\nu : \lim G \to G$, i.e., a cone $\nu : \lim G \to G$ with the property that if $\tau : D \to G$ is any cone, then there is a unique morphism $\theta : D \to \lim G$ such that $\tau_A = \nu_A \circ \theta$ for all $A \in \mathscr{C}$.



The characterization of μF as $\lim U_F$ provides a principled derivation of the interpretation *build* of the `build` combinator for μF which complements the derivation of its *fold* combinator from standard initial algebra semantics. It also guarantees the correctness of the standard `fold`/`build` rule. Indeed, the universal property that the carrier μF of the initial F-algebra enjoys as $\lim U_F$ ensures:

- The projection from the limit μF to the carrier of each F-algebra defines the *fold* operator with type $(FA \to A) \to \mu F \to A$.
- Given a cone $\theta : C \to U_F$, the mediating morphism from it to the limiting cone $\nu : \lim U_F \to U_F$ defines a map from C to $\lim U_F$, i.e., from C to μF. Since a cone to the base U_F with vertex C has type $\forall x.(Fx \to x) \to C \to x$, this mediating morphism defines the *build* operator with type $(\forall x.\,(Fx \to x) \to C \to x) \to C \to \mu F$.
- The correctness of the `fold`/`build` rule then follows from the fact that $\mathit{fold}\,k$ after $\mathit{build}\,g$ is a projection after a mediating morphism, and thus is equal to the cone g applied to the specific algebra k. We have



4.2 A Categorical Interpretation of the Monadic Fusion Rule

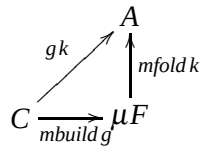
The key to proving the correctness of our monadic fusion rule is to interpret a suitable variant of the preceding diagram in a suitable category. Let $\mathscr{C}$ be a category. A *monad on* $\mathscr{C}$ is a functor $M : \mathscr{C} \to \mathscr{C}$ together with two operations $bind : MA \to (A \to MB) \to MB$ (normally written infix as $\gg=$) and $return : A \to MA$ satisfying the *monad laws* [9]. If M is a monad on $\mathscr{C}$, then the *Kleisli category of* M is the category $\mathscr{C}_M$ whose objects are the objects of $\mathscr{C}$, and whose morphisms from A to B are the morphisms from A to MB in $\mathscr{C}$. The identity morphism in $\mathscr{C}_M$ is *return*, and the composition of morphisms $h : A \to B$ and $k : B \to C$ in $\mathscr{C}_M$ is given by $k \bullet h = k^* \circ h$, where f^* is defined to be $\lambda x.(x \gg= f)$ for any morphism f in $\mathscr{C}$, and the computation on the right-hand side is performed in $\mathscr{C}$.[1]

A *distributive law* for a monad $M : \mathscr{C} \to \mathscr{C}$ over a functor $F : \mathscr{C} \to \mathscr{C}$ is a natural transformation $\delta : FM \to MF$. Let $\mu_A = id_A^*$, and note that if $f : A \to MB$, then $f^* = \mu_B \circ Mf$. Also note that $Mf = (return \circ f)^*$. We use these well-known facts about monads later. Suppose δ is a distributive law for M over F satisfying

$$\begin{array}{lcl} \delta_A \circ F return & = & return_{FA} \\ \mu_{FA} \circ M\delta_A \circ \delta_{MA} & = & \delta_A \circ F\mu_A \end{array}$$

We can define a functor $\hat{F} : \mathscr{C}_M \to \mathscr{C}_M$ by $\hat{F}A = FA$ and $\hat{F}k = \delta \circ Fk$. The functor $\hat{F}$ is called the *monadic extension* of F by M. If $k : A \to B$ then $\hat{F}k : \hat{F}A \to \hat{F}B$.

An *M-monadic F-algebra*, or *MF-algebra* for short, is an $\hat{F}$-algebra or, equivalently, a morphism $h : FA \to MA$ in $\mathscr{C}$. A morphism between MF-algebras $k_1 : FA \to A$ and $k_2 : FB \to B$ is an $\hat{F}$-algebra homomorphism in $\mathscr{C}_M$. From the definition of composition in $\mathscr{C}_M$, we see that such a morphism is simply a map $f : A \to MB$ in $\mathscr{C}$ such that $f^* \circ k_1 = k_2^* \circ \delta \circ Ff$. The forgetful functor U_{MF} from the category of MF-algebras to $\mathscr{C}_M$ maps each MF-algebra $h : FA \to A$ in $\mathscr{C}_M$ to A and each morphism between MF-algebras to itself. We are thus interested in the interpretation, in $\mathscr{C}_M$, of the following variant of the diagram at the end of Section 4.1:



Here, *mfold* and *mbuild* are the interpretations in $\mathscr{C}_M$ of `mfold` and `mbuild`, respectively, M is the interpretation of `m` in the types of `mfold` and `mbuild`, and *bind* and *return* are the interpretations of the `>>=` and `return` operations for `m`, respectively. In other words, we are interested in showing that μF is the limit of U_{MF} in $\mathscr{C}_M$. Then, by the same reasoning as in Section 4.1, we would have that

- The projection from the limit μF to the carrier of each MF-algebra would define the *mfold* operator mapping each $\hat{F}$-algebra with carrier A to a map

[1] Implicit in the notation $\lambda x.x \gg= f$ is the assumption that $\mathscr{C}$ is cartesian closed. This is a reasonable assumption for programming language semantics; in particular, it is a consequence of parametricity.

from μF to A in $\mathscr{C}_M$, i.e., would define the *mfold* operator with type $(FA \to MA) \to \mu F \to MA$ in $\mathscr{C}$.

- Given a cone $\theta : C \to U_{MF}$, the mediating morphism from it to the limiting cone $\nu : \lim U_{MF} \to U_{MF}$ would define a map from C to $\lim U_{MF}$, i.e., from C to μF. Since such a cone maps each $\hat{F}$-algebra with carrier A to a map from C to A in $\mathscr{C}_M$, it would define the *mbuild* operator with type $(\forall x.\,(Fx \to Mx) \to C \to Mx) \to C \to M(\mu F)$ in $\mathscr{C}$.
- The correctness of the `mfold`/`mbuild` monadic fusion rule would then follow from the fact that *mfold k* after *mbuild g* is a projection after a mediating morphism in $\mathscr{C}_M$, and thus is equal to the cone g applied to the specific algebra k. We would therefore have precisely the previous diagram for $k : \hat{F}A \to A$ in $\mathscr{C}_M$, i.e., we'd have $mbuild\,g\,x \gg\!= mfold\,k = g\,k\,x$, as desired.

So, we need to show that $\mu F = \lim U_{MF}$ in $\mathscr{C}_M$, i.e., we need to show that i) *mfold* defines a cone to the base U_{MF} with vertex μF. We therefore show that $p \bullet mfold\,k_A = mfold\,k_B$ for all A and B and all morphisms p in $\mathscr{C}_M$ from $k_A : FA \to A$ to $k_B : FB \to B$; and that ii) if g is a cone to the base U_{MF} with vertex C, i.e., if $p \bullet g_{k_A} = g_{k_B}$ for all A and B and p as in i, then *mbuild g* is the unique morphism such that, for all A, $mfold\,k_A \bullet mbuild\,g = g\,k_A$. Translating these conditions from $\mathscr{C}_M$ into $\mathscr{C}$, we see that we need to show

i') $p^* \circ mfold\,k_A = mfold\,k_B$ for all A and B and all morphisms $p : A \to MB$ in $\mathscr{C}$ such that $p^* \circ k_A = k_B^* \circ \delta \circ F\,p$.

ii') If $p^* \circ g_{k_A} = g_{k_B}$ for all A, B and p as in i', then *mbuild g* is the unique morphism such that for all A, $(mfold\,k_A)^* \circ mbuild\,g = g\,k_A$.

Proof of i'. We first note that, for all A, $mfold\,k_A = fold\,(k_A^* \circ \delta)$. So proving i' is equivalent to proving that, for all A, B, and p as specified there, $p^* \circ fold\,(k_A^* \circ \delta) = fold\,(k_B^* \circ \delta)$. We first observe that

$$\begin{array}{rcl}
\delta \circ F\,p^* & = & \delta \circ F(\mu \circ Mp) \\
& = & \mu \circ M\delta \circ \delta \circ FMp \quad \text{by functoriality of } F \text{ and axioms for } \delta \\
& = & \mu \circ M\delta \circ MF\,p \circ \delta \quad \text{by naturality of } \delta \\
& = & (\delta \circ F\,p)^* \circ \delta \quad \text{by functoriality of } M \text{ and characterization of } (-)^*
\end{array}$$

Thus

$$\begin{array}{rcl}
k_B^* \circ \delta \circ F\,p^* & = & k_B^* \circ (\delta \circ F\,p)^* \circ \delta \\
& = & (k_B^* \circ (\delta \circ F\,p))^* \circ \delta \text{ by the monad laws for } M \\
& = & (p^* \circ k_A)^* \circ \delta \text{ since } p \text{ is an } \hat{F}\text{-algebra homomorphism} \\
& = & p^* \circ k_A^* \circ \delta \text{ by the monad laws for } M
\end{array}$$

We therefore have that

$$\begin{array}{ccccc}
F(\mu F) & \xrightarrow{F(fold\,(k_A^* \circ \delta))} & FMA & \xrightarrow{F\,p^*} & FMB \\
\downarrow{\scriptstyle in} & & \downarrow{\scriptstyle k_A^* \circ \delta} & & \downarrow{\scriptstyle k_B^* \circ \delta} \\
\mu F & \xrightarrow{fold\,(k_A^* \circ \delta)} & MA & \xrightarrow{p^*} & MB
\end{array}$$

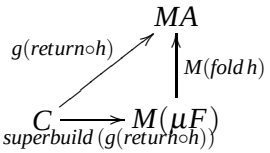
i.e., that each of the subsquares, and thus the outer square, commutes. By the uniqueness of *fold* we have that $p^* \circ fold\,(k_A \circ \delta) = fold\,(k_B \circ \delta)$, i.e., that $p \bullet mfold\,k_A = mfold\,k_B$ as desired.

Proof of ii'. Suppose g is such that for every morphism p from $k_A : FA \to A$ to $k_B : FB \to B$ in $\mathscr{C}_M$, we have $p^* \circ g_{k_A} = g_{k_B}$. Define the map *mbuild* by $mbuild\,g = g\,(return \circ in)$ or, equivalently, $mbuild\,g = superbuild\,(\lambda\alpha.g(return \circ \alpha))$. Here, the operator *superbuild* is the interpretation in $\mathscr{C}$ of the `superbuild` combinator of type `(forall a.  (f a -> a) -> c -> h a) -> c -> h (Mu f)`, with fusion rule `superbuild g >>= fold k = g k >>= id`, introduced in [2]. We first check that $mbuild\,g$ satisfies the property that, for all A, $mfold\,k_A \bullet mbuild\,g = g_{k_A}$ in $\mathscr{C}_M$, i.e., $(fold\,(k_A^* \circ \delta))^* \circ mbuild\,g = g_{k_A}$ in $\mathscr{C}$. For all c in C,

$$\begin{array}{rcl} & & (fold\,(k_A^* \circ \delta))^*(mbuild\,g\,c) \\ &=& mbuild\,g\,c \;\gg\!=\; fold\,(k_A^* \circ \delta) \\ &=& superbuild\,(\lambda\alpha.g(return \circ \alpha))\,c \;\gg\!=\; fold\,(k_A^* \circ \delta) \\ &=& (\lambda\alpha.g(return \circ \alpha))\,(k_A^* \circ \delta)\,c \;\gg\!=\; id \\ &=& g(return \circ k_A^* \circ \delta)\,c \;\gg\!=\; id \\ &=& g\,k_A\,c \end{array}$$

The final equivalence follows from the facts that g is a cone and that $id : MA \to A$ is a morphism in $\mathscr{C}_M$ from the MF-algebra $return \circ k_A^* \circ \delta : FMA \to MA$ in $\mathscr{C}_M$ to the MF-algebra $k_A : FA \to A$ in $\mathscr{C}_M$.

Next we verify that $mbuild\,g$ is the *unique* morphism such that, for all k_A, $mfold\,k_A \bullet mbuild\,g = g\,k_A$. Suppose $\phi : C \to \mu F$ is such that $mfold\,k_A \bullet \phi = g\,k_A$. We will show that $\phi = mbuild\,g$, i.e., that $\phi = superbuild(\lambda\alpha.g(return \circ \alpha))$ by showing that ϕ satisfies the property that $superbuild(\lambda\alpha.g(return \circ \alpha))$ is unique for and thus must actually *be* $superbuild(\lambda\alpha.g(return \circ \alpha))$. This property is known from [2] to be that of being a mediating morphism between the MU_F-cones $g(return \circ h)$ and $M(fold\,h)$, i.e., being such that



for every $h : FA \to A$ under the assumption that M preserves $\lim U_F$ (which is inherited from the results in that paper). So, from the fact that $mfold\,k_A \bullet \phi = g\,k_A$ for all k_A, we must show that, for all $h : FA \to A$, $M(fold\,h) \circ \phi = g(return \circ h)$. Given such an h and any c in $\mathscr{C}$, we have that

$$\begin{array}{rcll} M(fold\,h)(\phi\,c) &=& \phi\,c \gg\!= return \circ (fold\,h) & \text{functoriality of } M \text{ via } \gg\!= \\ &=& \phi\,c \gg\!= (fold\,(M\,h \circ \delta)) & \\ &=& \phi\,c \gg\!= (fold\,((return \circ h)^* \circ \delta)) & \text{functoriality of } M \text{ via } \gg\!= \\ &=& ((fold\,((return \circ h)^* \circ \delta))^* \circ \phi)\,c & \\ &=& (mfold\,((return \circ h) \bullet \phi))c & \\ &=& g(return \circ h)c & \end{array}$$

The fact that $\mathit{return} \circ (\mathit{fold}\, h) = \mathit{fold}\, (M\, h \circ \delta)$, which is used in the third equivalence above, is a consequence of the distributivity laws and the naturality of *return*. Thus ϕ satisfies the property for which $\mathit{mbuild}\, g$ is unique and so must equal $\mathit{mbuild}\, g$. Both i' and ii' therefore hold, and so $\mu F = \lim U_{MF}$ in $\mathscr{C}_M$.

5 DUALITY

Shortage of space prevents us from giving the corresponding coalgebraic constructs and results in detail here, so we simply present their implementation. We have

```
class Comonad cm where
 coreturn :: cm a -> a
 (=>>)    :: cm b -> (cm b -> a) -> cm a

data Nu f = Out {unOut :: f (Nu f)}

unfold :: Functor f => (a -> f a) -> a -> Nu f
unfold k = Out . fmap (unfold k) . k

cunfold :: (Comonad c, Functor f, Dist c f) =>
                     (c a -> f a) -> c a -> Nu f
cunfold k = unfold (\xs -> delta (xs =>> k))


cdestroy :: (Comonad c, Functor f) =>
  (forall a. (c a -> f a) -> c a -> x) -> c (Nu f) -> x
cdestroy g = g (unOut . coreturn)

cunfold k c =>> cdestory g = g k c
```

6 CONCLUSION AND DIRECTIONS FOR FUTURE WORK

In this paper, we recalled the monadic `fold`, found in the literature, which consumes an inductive data structure and returns a value in a monadic context. This operator differs from the standard `fold` operator in that it takes as input a monadic algebra rather than a standard algebra. We have argued that, for both aesthetic and pratical reasons, the monadic `fold` operator is often preferable to the standard `fold` operator.

We therefore defined a monadic `build` combinator which is parameterized over functions which use monadic algebras to uniformly consume data structures, and is thus the natural counterpart to the monadic `fold` operator. We used this combinator to define a short cut fusion rule for eliminating from modular monadic programs intermediate "gluing" data structures in monadic contexts. We provided examples showing how the monadic `fold` and monadic `build` combinators can be used, and how the monadic short cut fusion rule can optimize modular programs written using them. We also established the correctness of the monadic short cut fusion rule. Finally, we sketched the coalgebraic duals of these operators and rule.

A number of possibilities for future work arise. At a theoretical level, we use distributivity to construct a functor on the Kleisli category of a monad from a functor on the underlying category. It would be interesting to be able to derive the correctness of the `mfold` and `mbuild` combinators directly. This would mean proving that the Kleisli category of a monad forms a parametric model whenever the underlying category does. In addition, both the monadic short cut fusion rule introduced here and the `fold`/`msuper build` rule from [2] eliminate intermediate data structures in monadic contexts. We believe these two rules to offer distinct fusion options in the presence of distributivity; it would therefore be interesting to see which is more useful for programs that arise in practice. A final direction for future work involves extending the results of [6, 7] to give monadic `build`s, as well as associated fusion rules, for advanced datatypes, such as nested types, GADTs, and dependent types. The latter seems particularly challenging, since impredicative quantification is not supported by pure dependent type theory.

REFERENCES

[1] M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, 1992.

[2] N. Ghani and P. Johann. Short cut fusion of recursive programs with computational effects. To appear, *Proceedings, TFP*, 2008.

[3] N. Ghani, P. Johann, T. Uustalu, and V. Vene. Monadic augment and generalised short cut fusion. In *Proceedings, ICFP*, pages 294–305, 2005.

[4] N. Ghani, T. Uustalu, and V. Vene. Build, augment and destroy. Universally. In *Proceedings, APLAS*, pages 327–347, 2003.

[5] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings, FPCA*, pages 223–232, 1993.

[6] P. Johann and N. Ghani. Initial algebra semantics is enough! In *Proceedings, TLCA*, pages 207–222, 2007.

[7] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Proceedings, POPL*, pages 297–308, 2008.

[8] C. Jürgensen. Using monads to fuse recursive programs (extended abstract), 2002.

[9] S. MacLane. *Categories for the Working Mathematician*. Springer, 1971.

[10] E. Meijer and J. Juering. Merging monads and folds for functional programming. In *Proceedings, AFP*, pages 228–266, 1995.

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

[12] J. Voigtländer. Asymptotic improvement of computations over free monads. In *Proceedings, MPC*, pages 388–403, 2008.