Chapter 21

Short Cut Fusion of Recursive Programs with Computational Effects

Neil Ghani[1], Patricia Johann[2]
Category: Research

Abstract: Fusion is the process of improving the efficiency of modularly constructed programs by transforming them into monolithic equivalents. This paper defines a generalization of the standard `build` combinator which expresses uniform production of functorial contexts containing data of inductive types. It also proves correct a fusion rule which generalizes the `fold`/ `build` and `fold`/`buildp` rules from the literature, and eliminates intermediate data structures of inductive types without disturbing the contexts in which they are situated. An important special case arises when this context is monadic. When it is, a second rule for fusing combinations of producers and consumers via monad operations, rather than via composition, is also available. We give examples illustrating both rules, and consider their coalgebraic duals as well.

21.1 GENERALIZING SHORT CUT FUSION

21.1.1 Introducing Short Cut Fusion

Fusion is the process of improving the efficiency of modularly constructed programs by transforming them into monolithic equivalents. Short cut fusion [7] is concerned with eliminating list traversals from compositions of components that are "glued" together via intermediate lists. Short cut fusion uses a local transformation — known as the `foldr`/`build` rule — to fuse computations which can

[1]University of Nottingham, Nottingham, UK. `nxg@cs.nott.ac.uk`. Supported in part by EPSRC grant EP/C511964/2.

[2]Rutgers University, Camden, NJ, USA. `pjohann@crab.rutgers.edu`. Supported in part by NSF grant CCF-0700341.

```
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 21.1. The `fold` and `build` combinators and `fold`/`build` rule.

be written as compositions of applications of the uniform list-consuming function `foldr` and the uniform list-producing function `build` given by

```
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 (:) []
```

The function `foldr` is standard in the Haskell prelude. Intuitively, `foldr c n xs` produces a value by replacing all occurrences of `(:)` in `xs` by `c` and the occurrence of `[]` in `xs` by `n`. Thus, `sum xs = foldr (+) 0 xs` sums the (numeric) elements of the list `xs`. Uniform production of lists, on the other hand, is accomplished using the combinator `build`, which takes as input a type-independent template for constructing "abstract" lists and produces a corresponding "concrete" list. Thus, `build (\c n -> c 4 (c 7 n))` produces the list `[4,7]`. Uniform list transformers can be written in terms of both `foldr` and `build`. For example, the function `map` can be implemented as

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

The `foldr`/`build` rule capitalizes on the uniform production and consumption of lists to improve the performance of list-manipulating programs. It says

```
foldr c n (build g) = g c n                (21.1)
```

If `sqr x = x * x`, then this rule can be used, for example, to transform the modular function `sum . map sqr :: [Int] -> Int` which produces an intermediate list into an optimized form which does not:

```
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
```

```
buildp :: Functor f =>
   (forall a. (f a -> a) -> c -> (a,z)) -> c -> (Mu f, z)
buildp g = g In

fmap (fold k) . buildp g = g k
```

FIGURE 21.2. The `buildp` combinator and `fold`/`buildp` fusion rule.

21.1.2 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; as usual, it is the programmer's responsibility to ensure that this is the case. It is well-known that analogues of `foldr` exist for every inductive datatype. As shown in [5, 6], every inductive type also has an associated generalized `build` combinator; the extra type `c` in the type of `build` is motivated in those papers and to lesser extent in Section 21.3 below. These combinators can be implemented generically in Haskell as in Figure 21.1. 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 `fold`/`build` fusion 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 `c` to be the unit type as well. As usual, `fold` and `build` implement the isomorphisms between inductive types and their Church encodings.

21.1.3 Short Cut Fusion in Context

Short cut fusion handles compositions `g . f` in which the data structure produced by `f` is passed from `f` to `g`. But what if `f` produces not just a single data structure, but multiple such structures, embeds these data structures in a non-trivial context, and passes the result to `g` for consumption of these data structures "in context"? Is it possible to eliminate these intermediate data structures from `g . f` while keeping the context information, which `g` may need to compute its result, intact? Unfortunately, standard fusion techniques cannot achieve this: the intermediate data structures produced by `f` cannot be decoupled from the context in which they are situated. In [2], Fernandes, Pardo, and Saraiva introduce a technique for fusing compositions `g . f` in which `f` passes to `g` not only the intermediate data structure produced by `f`, but an additional datum as well. Although `g` requires this datum to compute its result, this datum is not used when processing the intermediate data structure, and so only the data structure itself needs to be eliminated from `g . f`. To do this, [2] uses a variant of the standard `fold`/`build` rule based on the combinator `buildp`, which captures the extra datum by returning a data structure

```
superbuild :: (Functor f, Functor h) =>
    (forall a. (f a -> a) -> c -> h a) -> c -> h (Mu f)
superbuild g = g In

fmap (fold k) . (superbuild g) = g k
```

FIGURE 21.3. The `superbuild` combinator and `fold`/`superbuild` fusion rule.

embedded in a pair context. The datatype-generic `buildp` combinator and its associated `fold`/`buildp` fusion rule are given in Figure 21.2. There, `fmap` is the map function

```
fmap :: (a -> b) -> ([a],z) -> ([b],z)
fmap (as,z) = (map f as, z)
```

which witnesses the fact that the type constructor `h` given by `h x = (x,z)` is a functor. The context information produced by `buildp` and used by the consumer in the left-hand side of the `fold`/`buildp` fusion rule is reflected in the pair return types of `buildp` and its template argument, as well as in the mapping of `fold` across the pair in the associated `fold`/`buildp` rule. This rule eliminates intermediate data structures within the context of pairing with an additional datum.

But now suppose we want to write a function

```
gsplitWhen :: (b -> Bool) -> [b] -> [[b]]
```

which splits a list into sublists at every element that satisfies a given `p`. Note that the function `gsplitWhen` splits lists into arbitrary numbers of sublists, depending on the data they contain, and that the type `z` in the type of `buildp` cannot be instantiated to allow the return of a number of lists which has the potential to change on each program run. This means that `gsplitWhen` cannot be written in terms of `buildp`. Moreover, compositions of `gsplitWhen` with functions that consume each of the individual “inner” lists produced by `gsplitWhen` but require the information inherent in its “context list” to compute their results cannot be fused using the `fold`/`buildp` rule. But why try to structure programs only with contexts of the form `(-,z)`? That is, why not consider a generalization of the `buildp` combinator, and a generalization of the `fold`/`buildp` fusion rule which can be used to eliminate intermediate data structures, like those returned by `gsplitWhen`, which appear in contexts other than just pairs? That is precisely what this paper does. We call these generalizations `superbuild` and the `fold`/`superbuild` rule, respectively. Like `buildp` and the `fold`/`buildp` rule, our `superbuild` combinator and `fold`/`superbuild` fusion rule are available at every inductive datatype. Datatype-generic versions are given in Figure 21.3; note that the type of superbuild is actually generic in both `f` and `h`. The generalization of the pair context in the type of `buildp` is captured by the replacement in the type of `superbuild` of the type `(x,z)` by the type `h x` for a more general “context functor” `h`. This generalization is further reflected in the replacement of the `fmap` function for pairs in the `fold`/`buildp` rule by the `fmap` function for the more general context functor `h` in the `fold`/`superbuild` rule. The `fold` combinator in the `fold`/`superbuild` rule is the one for `Mu f`, as usual. These `fmap` and `fold` functions are guaranteed to be defined precisely because the type of `superbuild` requires both `f` and `h` to be functors. We argue in Section 21.3 that the `fold`/`superbuild` rule holds for a large class of functors `h`.

Taking `h x = x` gives the generalized `build` combinator and `fold`/`build` rule from Figure 21.1, while taking `h x = (x,z)` gives the `buildp` combinator and `fold`/ `buildp` rule from Figure 21.2. In general, the `fold`/`superbuild` rule can fuse compositions in which context information describable by non-pair functors is passed, along with intermediate data structures, from producer to consumer. Indeed, the `fold`/`superbuild` rule eliminates intermediate structures of type `Mu f` obtained by mapping a consumer expressed as a `fold` over the data of type `Mu f` stored in a context specified by a functor `h`. Thus, setting `c = [b]`, `h x = [x]`, and `f` to be the functor whose least fixed point is `[b]`, we can write

```
gsplitWhen p = superbuild go where
 go c n z = case z of
             []       -> []
             [w]      -> [c w n]
             (w : ws) -> let xs = go c n ws
                          in if p w then (c w n) : xs
                              else (c w (head xs)) : (tail xs)
```

If `lgh = foldr (\x -> (1+)) 0` then using the `fold`/`superbuild` rule to fuse the composition `evLghs = map lgh . gsplitWhen even` gives

```
evLghs' z = case z of
             []       -> []
             [w]      -> [1]
             (w : ws) -> let xs = evLghs' ws
                          in if even w then 1 : xs
                              else (head xs + 1) : (tail xs)
```

Note that `evLghs'` trades production and consumption of the list of intermediate lists returned by `gsplitWhen even` in `evLghs` for production of the corresponding list of values obtained by applying `lgh` to each such list.

21.1.4 Short Cut Fusion in Effectful Contexts

The ability to fuse intermediate data structures in context turns out to be the key to extending short cut fusion to the effectful setting. Although fusion in the presence of computational effects has been studied by other researchers (see, e.g., [11, 12, 14, 16]), short cut fusion in particular has not previously been formally explored in this context. To perform short cut fusion in an effectful context, the functional argument to `superbuild`, and thus `superbuild` itself, must have a monadic return type. Monads can be implemented in Haskell as type constructors supporting `>>=` and `return` operations as follows; these operations are expected to satisfy the semantic monad laws, but ensuring this for alleged instances of Haskell's `Monad` class is, as usual, the programmer's responsibility.

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

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

msuperbuild g c >>= fold k = g k c >>= id
```

FIGURE 21.4. The `msuperbuild` combinator and `fold`/`msuperbuild` fusion rule.

If `m` is a monad, then instantiating the context functor `h` to `m` in `superbuild`'s type gives the `msuperbuild` combinator in Figure 21.4. The accompanying `fold`/`msuper build` rule is the natural “monadification” of the `fold`/`superbuild` rule; we give an example of its use in Section 21.2. This rule does not eliminate the monadic context described by `m`, but does eliminate intermediate data structures of type `Mu f` within that monadic context. Moreover, the rule does in general change the context containing the data structure, and so is more sophisticated than its non-monadic counterpart.

The remainder of this paper is structured as follows. In Section 21.2 we apply our new `fold`/`superbuild` and `fold`/`msuperbuild` rules to substantive examples. In Section 21.3 we show how the `superbuild` and `msuperbuild` combinators are derived from initial algebra semantics, and prove the correctness of their associated fusion rules. In Section 21.4 we give non-monadic and monadic `superdestroy`/`unfold` rules dual to our non-monadic and monadic `fold`/`super build` rules; our results for `superbuild` and `msuperbuild` are easily dualized to prove them correct. In Section 21.5 we discuss related work, and in Section 21.6 we conclude and offer directions for future research. A Haskell implementation of our results and an additional example highlighting the versatility of our rules are available at `http://www.cs.nott.ac.uk/`∼`nxg`.

21.2 EXAMPLES

In this section we give some more sophisticated examples showcasing the power of the `fold`/`superbuild` and `fold`/`msuperbuild` fusion rules. Our first example shows that the `fold`/`superbuild` rule can be used to eliminate intermediate data structures other than lists. Our second example shows that the `fold`/`msuperbuild` rule can eliminate data structures within the state monad.

Example 21.1. Consider the simple arithmetic expression datatype given by

```
data Oper = Add | Mul | Sub deriving (Eq, Show)

data Expr  = Lit Int | Op Oper Expr Expr deriving (Eq, Show)
```

The `fold` combinator for expressions, the instance of `superbuild` for expressions where `c` is `Expr` and `h x` is `[x]`, and the associated fusion rule are

```
foldExpr :: (Int -> a) -> (Oper -> a -> a -> a) -> Expr -> a
foldExpr l o e = case e of
                   Lit i -> l i
```

```
                    Op op e1 e2 -> o op (foldExpr l o e1)
                                        (foldExpr l o e2)

superbuildExpr :: (forall a. (Int -> a) ->
    (Oper -> a -> a -> a) -> Expr -> [a]) -> Expr -> [Expr]
superbuildExpr g = g Lit Op

map (foldExpr l o e) (superbuild g) = g l o e
```

If we define `opToHas Add = (+)`, `opToHas Mul = (*)`, and `opToHas Sub = (-)`, then we can implement an interpreter which traces the evaluation steps taken in computing the integer values represented by expressions as

```
trace :: Expr -> [Expr]
trace = superbuildExpr g

g :: (Int -> a) -> (Oper -> a -> a -> a) -> Expr -> [a]
g l o e =  case e of
             Lit i -> [l i]
             Op op e1 e2 -> let b1 = foldExpr l o e1
                                b2 = foldExpr l o e2
                                e′ = o op b1 b2
                                Lit k = last (g Lit Op e1)
                                Lit j = last (g Lit Op e2)
                                b1s = g l o e1
                                b2s = g l o e2
                             in  (e′ : (map (\x -> o op x b2)
                                     (tail b1s))
                                 ++ (map (o op (last b1s))
                                     (tail b2s))
                                 ++ [l (opToHas op k j)])
```

For example, the call

```
trace (Op Mul (Op Add (Lit 5) (Lit 6)) (Op Sub (Lit 7) (Lit 4)))
```

generates the trace

```
[ Op Mul (Op Add (Lit 5) (Lit 6)) (Op Sub (Lit 7) (Lit 4)),
  Op Mul (Lit 11) (Op Sub (Lit 7) (Lit 4)),
  Op Mul (Lit 11) (Lit 3), Lit 33 ]
```

Once an interpreter trace is built, it is possible to perform various analyses of it. For example, we can measure the computational effort required to compute the value represented by each expression arising in the evaluation of a given expression. For this we use `count`, which counts 0 units of effort to compute a literal, 2 to perform an addition, 3 to perform a subtraction, and 5 to perform a multiplication.

```
count Add x y = x + y + 2
count Sub x y = x + y + 3
count Mul x y = x + y + 5
```

We then have

```
costExprs :: Expr -> [Int]
costExprs expr = map (foldExpr (\x -> 0) count)
                     (superbuildExpr g expr)
```

For example, the call `costExprs (Op Mul (Op Add (Lit 5) (Lit 6)) (Op Sub (Lit 7) (Lit 4)))` generates the result `[10,8,5,0]`. Fusion using the `fold`/`superbuild` rule gives the equivalent function `costExprs’ g (\x -> 0) count`, in which the intermediate list of expressions is not constructed.

Example 21.2. Pardo [14] shows that graph traversal algorithms, such as depth-first traversal and breadth-first traversal, can be written as calls to a monadic `unfold` combinator. Here, we show that these algorithms can be written in terms of `msuper build`. The relationship between monadic and non-monadic `unfold` combinators, and between `superbuild` and `msuperbuild`, is discusssed in Section 21.5 below.

A graph traversal is represented as a function which takes as input a list of root vertices of a graph and returns a list containing the vertices met in order as the graph is traversed. We can represent the vertices of a graph by integers, and a graph by an adjacency list function for vertices as follows:

```
type V     = Int
type Graph = V -> [V]
```

In a graph traversal, each vertex is visited at most once. To avoid repeated visits to vertices we can use the state monad [13, 15] to maintain a list of vertices visited previously in the computation and thread this list through the traversal. We therefore define a datatype of *visit-dependent data*, each element of which is a function taking a list of vertices already visited as input and returning a datum depending on that list together with an updated list of visited vertices. We have

```
data State s a = State {runstate :: s -> (s,a)}

instance Monad (State s) where
  return a = State (\s -> (s,a))
  t >>= f  = State (\s -> let (s’,v) = runstate t s
                          in runstate (f v) s’)

type Vis a = State [V] a
```

Visit-dependent data support the following useful auxiliary functions:

```
data Unit = Unit

bot :: a
bot = bot

emp :: Vis a -> a
emp xs = snd (runstate xs [])
```

```
dft :: (V -> [V]) -> [V] -> [V]
dft g vs = emp (depthFirst g vs)

depthFirst :: (V -> [V]) -> [V] -> Vis [V]
depthFirst g = superbuild (df g)

df :: (V -> [V]) ->
        forall a. (V -> a -> a) -> a -> [V] -> Vis a
df g c n vs = case vs of
  []     -> return n
  (x:xs) -> mem x >>=
             (\b -> if b then df g c n xs
                     else sunion x >>=
                         (\z -> df g c n (g x ++ xs) >>=
                         (\ys -> return (c x ys))))
```

FIGURE 21.5. Depth-first graph traversal functions.

```
sunion :: V -> Vis Unit
sunion v = State (\vs -> (v:vs, bot))

mem :: V -> Vis Bool
mem v = State (\vs -> (vs, elem v vs))
```

With this machinery in place we can define depth-first traversal as in Figure 21.5. There, `dft` first allocates an empty list of visited vertices, then runs `depthFirst`, yielding a final list of visited vertices, and then de-allocates this visitation list and returns the list resulting from the traversal. At each iteration of the traversal, `df` explores the current list of roots in `vs` to find a vertex it has not reached before. This is accomplished by removing from the front of `vs` all vertices for which `mem x` is true until either an unvisited vertex or the end of `vs` is encountered. When an unvisited vertex `x` is encountered, `df` adds `x` to the list of vertices visited, recursively computes the depth-first traversals of the graphs rooted at `x`'s children, as well as those specified by the rest of the vertices in `vs`, and then returns the list of vertices obtained by adding `x` to the list of vertices recording the order in which the rest of the vertices are traversed. The code for breath-first search is identical, except that the function `bf` corresponding to `df` uses `xs ++ g x` rather than `g x ++ xs`. To traverse a particular graph we specify the desired traversal, the graph's adjacency list function, and its root vertices. For example, if the graph *G* is modeled by `g 0 = [2,1]`, `g 1 = []`, and `g x = [x+1]`, then `depthFirst g [0]` computes the depth-first search of *G* starting at root vertex `0`.

For example, to consume the result of a traversal with `filtergph odd` where

```
filtergph :: (V -> Bool) -> [V] -> Vis [V]
filtergph p = foldr (\v i -> if p v then return (v : emp i)
                              else return (emp i)) (return [])
```

we can write one of the following, depending on the desired traversal

```
dfFil g = emp (depthFirst g [0] >>= filtergph odd)

bfFil g = emp (breadthFirst g [0] >>= filtergph odd)
```

To perform the same computations without constructing the intermediate lists of visit-dependent vertices, we can use the `fold`/`msuperbuild` rule to get

```
dfFil′ g = emp ((df g) (\v i -> if odd v then return (v : emp i)
                                  else return (emp i))
                       (return []) [0] >>= id)

bfFil′ g = emp ((bf g) (\v i -> if odd v then return (v : emp i)
                                  else return (emp i))
                       (return []) [0] >>= id)
```

Note that the lists obtained by taking any non-empty initial segments of the results of `dfFil g` and `bfFil g` — and thus of `dfFil′ g` and `bfFil′ g` — reflect the distinction between the underlying depth-first and breadth-first traversals.

21.3 CORRECTNESS

21.3.1 Categorical Preliminaries

Let C be a category and F be an endofunctor on C. An *F-algebra* is a morphism $h : FA \to A$ in C. The object A is called the *carrier* of the F-algebra. The F-algebras for a given functor F are the objects of a category called the *category of F-algebras* and denoted F-$\mathcal{A}lg$. In the category of F-algebras, a morphism from $h : F\,A \to A$ to $g : F\,B \to B$ is a morphism $f : A \to B$ such that the following diagram commutes:

$$\begin{tikzcd} FA \arrow[r, "Ff"] \arrow[d, "h"'] & FB \arrow[d, "g"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

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 C. The unique F-algebra morphism from in to any other F-algebra $h : F\,A \to A$ is given by the interpretation *fold* of the `fold` combinator for the interpretation μF of the datatype `Mu F`. The *fold* operator for μF thus makes the following commute:

$$\begin{tikzcd} F(\mu F) \arrow[r, "F(fold\,h)"] \arrow[d, "in"'] & FA \arrow[d, "h"] \\ \mu F \arrow[r, "fold\,h"] & A \end{tikzcd}$$

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 ensures that, for every F-algebra h, the map *fold h* is defined uniquely.

As shown in [6], the carrier of the initial algebra of an endofunctor F on $\mathcal{C}$ can be seen not only as the carrier of the initial F-algebra, but also as the limit of the forgetful functor $U_F : F\text{-}\mathcal{A}lg \to \mathcal{C}$ mapping each F-algebra $h : F\,A \to A$ to its carrier A. If $G : \mathcal{C} \to \mathcal{D}$ is a functor, then a *cone* $\tau : D \to G$ to the base G with vertex D comprises an object D of $\mathcal{D}$ and a family of morphisms $\tau_C : D \to GC$, one for every object C of $\mathcal{C}$, such that for every arrow $\sigma : A \to B$ in $\mathcal{C}$, $\tau_B = G\sigma \circ \tau_A$ holds.

$$\begin{tikzcd}
GA \arrow[r, "G\sigma"] & GB \\
D \arrow[u, "\tau_A"] \arrow[ur, "\tau_B"'] &
\end{tikzcd}$$

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 $G : \mathcal{C} \to \mathcal{D}$ is an object $\lim G$ of $\mathcal{D}$ 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 $\tau_C = \nu_C \circ \theta$ for all $C \in \mathcal{C}$.

$$\begin{tikzcd}
GA \arrow[r, "F\sigma"] & GB \\
D \arrow[u, "\tau_A"] \arrow[ur, "\tau_B"] \arrow[r, "\theta"'] & \lim G \arrow[u, "\nu_B"'] \arrow[ul, "\nu_A"]
\end{tikzcd}$$

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* operator from standard initial algebra semantics given above. It also guarantees the correctness of the standard `fold`/`build` rules. 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 $\lim U_F$. Since a cone to 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` fusion rule then follows from the fact that *fold* after *build* is a projection after a mediating morphism, and thus is equal to the cone applied to the specific algebra. Diagrammatically, we have

$$\begin{tikzcd}
 & A \\
C \arrow[ur, "g\,k"] \arrow[r, "build\,g"'] & \mu F \arrow[u, "fold\,k"']
\end{tikzcd}$$

21.3.2 Correctness of the `fold`/`superbuild` Rule

To prove correctness of our `fold`/`superbuild` rule we are actually interested in the following variation of the preceding diagram:

$$\begin{tikzcd}
 & H\,A \\
C \arrow[ur, "g\,k"] \arrow[r, "superbuild\,g"'] & H\,(\mu F) \arrow[u, "H(fold\,k)"']
\end{tikzcd}$$

Here, *superbuild* is the interpretation in $\mathcal{C}$ of `superbuild`. If the functor $H : \mathcal{D} \to \mathcal{E}$ preserves limits — i.e., if, for every functor $G : \mathcal{C} \to \mathcal{D}$ and every limiting cone $\nu : \lim G \to G$, the cone $H\nu : H(\lim G) \to H \circ G$ is also a limit, henceforth denoted $\lim\,(H \circ G)$ — then this is the diagram for the universal property of $\lim\,(H \circ U_F)$. This leads us to ask which functors H preserve limits. It is well-known that right adjoints preserve limits, but this is a more restrictive class of functors than we would ideally like. On the other hand, H needn't preserve *all* limits, just $\lim U_F$.

A *connected category* is a non-empty category whose underlying graph is connected. A *connected limit* is a limit of a functor whose domain is a connected category. The limit $\lim U_F : F\text{-}\mathcal{A}lg \to \mathcal{C}$ is a connected limit since the category of F-algebras is connected (there is a morphism from the initial F-algebra $in : F\,(\mu F) \to \mu F$ to any other F-algebra), so knowing that the functor H interpreting the type constructor `h` in the type of `superbuild` preserves connected limits is sufficient to ensure correctness of the `fold`/`superbuild` rule. It is well-known that strictly positive functors preserve connected limits [3, 8]; in particular, all polynomial functors preserve them. More generally, all functors created by containers preserve connected limits [8]. The class of containers includes functors, such as those whose least fixed points are nested types, which are not strictly positive; the above proof thus covers many situations that are interesting in practice. To prove correctness of the `fold`/`superbuild` rule for functors H which do not preserve connected limits, it should be possible to give a formal argument based on logical relations [1]. However, a proof based upon logical relations would not cover examples such such as nested types which preserve connected limits but are not definable in the underlying type theory of the logical relation.

21.3.3 Correctness of the `fold`/`msuperbuild` Rule

To see that the `fold`/`msuperbuild` rule is correct, we consider the diagram

$$\begin{tikzcd}
 & M\,(MA) \arrow[r, "id^*"] & MA \\
C \arrow[ur, "g\,k"] \arrow[r, "msuperbuild\,g"'] & M\,(\mu F) \arrow[u, "M(fold\,k)"] \arrow[ur, "(fold\,k)^*"'] &
\end{tikzcd}$$

where M is the interpretation of `m` in the type of `msuperbuild`, *bind* and *return* are the interpretations of the `>>=` and `return` operations for `m`, respectively, and $f^*\,x = bind\,x\,f$. Correctness of the `fold`/`msuperbuild` rule is exactly commutativity of the diagram's outer parallelogram. The diagram's left-hand triangle commutes because it is an instance of the previous diagram, and standard properties of monads ensure that its right-hand side commutes as well. Then

$$\begin{array}{lll} g\,k\,c \gg\!= id & = & id^*\ (g\,k\,c) \\ & = & (id^* \circ g\,k)\,c \\ & = & ((fold\ k)^* \circ msuperbuild\ g)\,c \\ & = & (fold\ k)^*(msuperbuild\ g\ c) \\ & = & msuperbuild\ g\ c \gg\!= fold\ k \end{array}$$

It is worth noting here that many monads that arise in applications — including the exceptions monad, the state monad, and the list monad — preserve connected limits. The continuations monad, however, does not.

21.4 DUALITY

Our `fold`/`superbuild` and `fold`/`msuperbuild` rules dualize to the coinductive setting. Shortage of space prevents us from giving the corresponding constructs and results in detail here, so we simply present their implementation. We have

```
unfold :: Functor f => (a -> f a) -> a -> Mu f
unfold k x = In (fmap (unfold k) (k x))

superdestroy :: (Functor f, Functor h) =>
   (forall a. (a -> f a) -> h a -> c) -> h (Mu f) -> c
superdestroy g = g unIn

superdestroy g . fmap (unfold k) = g k
```

When `c` is `Mu f`, `superdestroy` returns an `h`-algebra which stores coalgebraic `f`-data. When `h` is a comonad, i.e., an instance of the `Comonad` class

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

we have

```
cmsuperdestroy :: (Functor f, Comonad cm) =>
   (forall a. (a -> f a) -> cm a -> c) -> cm (Mu f) -> c

cmsuperdestroy g = g unIn

cmsuperdestroy g (x =<< unfold k) = g k (x =<< id)
```

21.5 RELATED WORK

The work most closely related to ours is that of Pardo and his coauthors. Like this paper, [14] also investigates conditions under which the composition of a function producing an expression of type $M(\mu F)$ for M a monad and F a functor, and a function *fold k* of type $\mu F \to A$ can be fused to produce an expression of type MA. But there are several crucial differences with our work. First, Pardo uses `unfold` rather than `msuperbuild` to construct the intermediate expression. This gives his fusion rule some additional logical generality over ours, since `unfold` can construct elements of its associated functor `f`'s final coalgebra which are not in `f`'s initial algebra, whereas `msuperbuild` can construct only elements of `f`'s initial algebra. But when the initial and final algebras of each functor coincide, as in Haskell, this added logical generality yields no advantage in practice.

Secondly, Pardo's monadic hylofusion (and hylofusion in general) is only known to be correct in algebraically compact categories, i.e., categories in which the initial algebra and final coalgebra for each functor coincide. By contrast, our `fold`/`superbuild` rule is correct in any category supporting a parametric interpretation of `forall`, and this condition is independent of any compactness condition. The requirement that the interpreting category be algebraically compact is unfortunate since it generates strictness conditions that must be satisfied, and also requires the underlying monad to be strictness-preserving. This results in strictness condition propagation. By contrast, neither our `fold`/`superbuild` nor our `fold`/`msuperbuild` rules require the satisfaction of side conditions.

Thirdly, Pardo trades a composition of an `unfold` and a monadic `fold` for the computation of an equivalent fixed point. By contrast, our `fold`/`msuperbuild` rule trades a bind of a call to `msuperbuild` with a monadic `fold` for the bind of the application of the function argument to `msuperbuild` to the `fold`'s algebra with the identity function. Like all generalizations of the `fold`/`build` rule, our `fold`/`msuperbuild` rule requires "payment up front" in that the producer in a composition to be fused must be expressed in terms of `msuperbuild`. (This is not very different from the price paid by expressing consumers in terms of `unfold`). But our rule delivers a fused result which is simpler than that obtained using Pardo's technique. In particular, the functions obtained from our fusion rules involve only binds of applications involving data structure "templates", rather than fixed point calculations. Their computation is thus guaranteed to terminate.

Finally, Pardo requires the existence of a distributivity law of the underlying monad over the underlying functor in order to construct the lifting of functors to the Kleisli category on which his monadic hylofusion rule depends. But distributivity laws for arbitrary functors, even those admitting fixed points, need not exist.

Meijer and Jeuring [12] also develop a variety of fusion laws in the monadic setting, including a short cut fusion law for eliminating intermediate structures of type $F\,A$ in a monadic context M. Many fusion methods, including those of both [12] and [14], eliminate data structures in the carriers of initial algebras for only restricted classes of functors. By contrast, our method can eliminate data structures of *any* inductive type, and can handle non-monadic contexts as well. In addition, Jürgensen [11] and Voigtländer [16] have each defined fusion combinators based on the uniqueness of the map from a free monad to any other monad. These techniques give very different forms of fusion from ours.

21.6 CONCLUSION AND DIRECTIONS FOR FUTURE WORK

In this paper we have defined a `superbuild` combinator which generalizes the standard `build` combinator and expresses uniform production of functorial contexts containing data of inductive types. We have also proved correct a `fold`/`super build` fusion rule which generalizes the `fold`/`build` and `fold`/`buildp` rules from the literature, and eliminates intermediate data structures of inductive types without disturbing the contexts in which they are situated. An important special case arises when this context is monadic. When it is, our `fold`/`msuperbuild` rule fuses combinations of producers and consumers via monad operations, rather than via composition. We have given examples illustrating both the `fold`/`superbuild` and `fold`/`msuperbuild` rules, and considered their coalgebraic duals as well.

The standard `fold` combinator can consume data structures in any context describable by a functor, but the algebra it uses cannot depend on the context in a non-trivial way. By contrast, context information can be used by algebras to partially determine how the `pfold` combinator given in [2] will consume the data structures, but unfortunately the contexts are limited to pairs. Interestingly, the `pfold`/`buildp` rule given there for context-dependent `fold`s derives from the `fold`/`buildp` rule from Figure 21.2 for standard `fold`s. As already noted, it is the `fold`/`buildp` rule that our `fold`/`superbuild` and `fold`/`msuperbuild` rules generalize. One direction for future work is to generalize these rules even further to accommodate *both* context-dependent algebras and non-pair contexts.

Another direction for future work is suggested by considering an even more monadic fusion rule based on `fold`- and `build`-like combinators which manipulate algebra-like functions of type `f a -> m a`. Such a rule would produce intermediate data structures using “templates” based on so-called monadic algebras and, in the presence of a distributivity rule `delta` for `m` over `f`, would consume data structures using them via a monadic `mafold` combinator. We’d have

```
mafold :: (Functor f, Monad m) => (f a -> m a) -> Mu f -> m a
mafold k = fold (\x -> fmap k (delta x) >>= id)

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

masuperbuild c >>= mafold k = g k c
```

Although a datatype-generic `masuperbuild` combinator is not defined in [12], several instances of the above fusion rule are given (albeit in monadic `do`-notation). Yet no correctness proofs for any of these specific instances — let alone any formulation of, or correctness proof for, a datatype-generic fusion rule — are given. We believe an independent proof of the `mafold`/`masuperbuild` rule similar to those in Section 21.3 is possible. Although it is not entirely clear how such a proof would go, a proof for monads which preserve connected limits will likely require independent verification that $\lim(M\,U_{F,M}) = M(\mu F)$ for the forgetful functor $U_{F,M}$ mapping each monadic algebra $h : M\,a \to F\,a$ to a, and a proof for monads which do not preserve connected limits will likely be based on logical relations.

At first glance, the facts that `mafold` is defined in terms of `fold` and that `masuper build g` can be expressed as `msuperbuild (\k -> g (return . k))` together suggest that the `mafold`/`masuperbuild` rule might be derivable from (distributivity and) the `fold/msuperbuild` rule. However, we believe the two rules to offer distinct fusion options in the presence of distributivity; it would 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 [9, 10] to give regular and monadic `superbuild`s, as well as associated fusion rules, for advanced datatypes, such as nested types, GADTs, and dependent types.

REFERENCES

[1] T. Altenkirch, P. Levy, and M. Hasagawa, 2008. Personal communication, and message 1192 from the TYPES mailing list archive.

[2] J. P. Fernandes, A. Pardo, and J. Saraiva. A shortcut fusion rule for circular program calculation. In *Proceedings, Haskell Workshop*, pages 95–106, 2007.

[3] N. Ghani, M. Abbott, and T. Altenkirch. Containers - constructing strictly positive types. *Theoretical Computer Science*, 341(1):3–27, 2005.

[4] N. Ghani and P. Johann. Monadic Augment and Generalised Short Cut Fusion. *Journal of Functional Programming*, 17(6):731–776, 2007.

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

[6] N. Ghani, T. Uustalu, and V. Vene. Build, augment and destroy. Universally. In *Proceedings, Asian Symposium on Programming Languages*, pages 327–347, 2003.

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

[8] R. Hasegawa. Two applications of analytic functors. *Theoretical Computer Science*, 272(1-2):113–175, 2002.

[9] P. Johann and N. Ghani. Initial algebra semantics is enough! In *Proceedings, Typed Lambda Calculus and Applications*, pages 207–222, 2007.

[10] P. Johann and N. Ghani. Foundations for structured programming with GADTs. In *Proceedings, Principles of Programming Languages*, pages 297–308, 2008.

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

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

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

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

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

[16] J. Voigtländer. Asymptotic Improvement of Computations over Free Monads. In *Proceedings, Mathematics of Program Construction*, pages 388–403, 2008.

APPENDIX: A HANGMAN GAME

In this appendix we give an example based on the game of *hangman* to illustrate that the type of the data structures being eliminated by the `fold`/`superbuild` rule need not coincide with the datatype with which the combinators are associated, i.e., that `c` in the type of `superbuild` need not be instantiated to `Mu f`. A similar example appears in [4], but there it was the rose tree representing the game that was being eliminated, whereas here it is the inductive structure stored in the rose tree that is eliminated.

In the game of hangman there is an unknown word which a player is trying to guess, and a given number of lives in which it must be guessed. 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 of the letters in the word, and is lost if the player loses all of their allocated lives without guessing the word.

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 to make as accurate a model as possible. We model a game of hangman as a rose tree of game states.

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

instance Functor Rose where
  fmap :: (a -> b) -> Rose a -> Rose b
  fmap f (Node x xs) = Node (f x) (map (fmap f) xs)

type GState = [(Char, Bool)]

type Game = Rose GState
```

Each game state comprises a list of character-boolean pairs, with the characters representing a word over a predetermined alphabet — represented by a string constant such as `alphabet = "abcdefghijklmnopqrstuvwxyz"` — and the boolean value `True` associated with a character if and only if that character has been guessed. The occurrence of `map` on the right-hand side of the definition in the last line of the `Functor` instance declaration for `Rose` is that for lists, while the occurrence of `fmap` is the one being defined for rose trees. The function `guess` below updates a state after a character has been guessed, while the function `over` determines whether the current state indicates that the game is over.

```
guess :: Char -> (GState, Int) -> (GState, Int)
guess c ([],n) = ([], n-1)
guess c ((l,s):gs, n) = let (rs, k) = guess c (gs, n)
                         in if c == l then ((l,True):rs, k)
                            else ((l,s):rs, k)

over :: (GState, Int) -> Bool
over (s,n) = n == 0 || and (map snd s)
```

We now turn to our central task, namely constructing construct a game tree for each game of hangman. Given an initial state of a game and a number of lives in which to guess the specified word, we must construct the rose tree of new states which arises from returning that initial state, and then repeatedly processing a character from the alphabet and returning the resulting state until the game is over. To iterate the generation of game states we use the instance of `superbuild` for lists in which `h x` is `Rose x`. We have

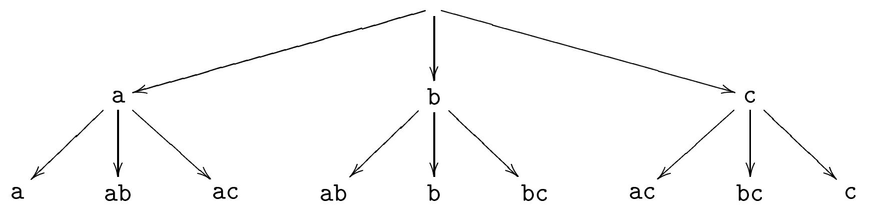
```
mkGame :: (GState, Int) -> Game
mkGame = superbuild g

g :: forall a. ((Char, Bool) -> a -> a) -> a ->
                               (GState, Int) -> Rose a
g c n s@(lbs,k) = if over s then Node (foldr c n lbs) []
                  else Node (foldr c n lbs)
                            [g c n (guess x s) | x <- alphabet]
```

For example, the call

```
mkGame ([('a', False), ('b', False), ('c', False)], 2)
```

generates the tree



Once a game tree has been built using `mkGame` we can perform various analyses of it. For example, we can compute the space of all possible results in a game tree with alphabet `"abc"` and two lives to live using a call to `mkGame` as above, or we can compute the list of letters which have been guessed at any point in such a game using

```
lettersGuessed :: Rose String
lettersGuessed = fmap letters (mkGame
                   ([('a', False),
                     ('b', False),
                     ('c', False)], 2))

letters :: GState -> String
letters = foldr (\(l,b) ls -> if b then l:ls else ls) []
```

Applying the `fold`/`superbuild` rule yields the following optimized version of `lettersGuessed` which doesn't construct an intermediate structure of type `Game`:

```
lettersGuessed' = g (\(l,b) ls -> if b then l:ls else ls) []
                    ([('a', False),
                      ('b', False),
                      ('c', False)], 2)
```