Zip Fusion with Hyperfunctions

John Launchbury, Sava Krstić and Timothy E. Sauerwein

Oregon GraduateInstitute
{jl,krstic,sauer}@cse.ogi.edu

1 Introduction

Automatic removal of intermediate structures has been an exciting possibility for a long time, holding a promise of the best of two worlds: programming with explicit intermediate structures enables concise and modular solution to problems; and the removal of the structures provides efficient run-time implementations. One particularly effective technique is called the foldr-build rule [2, 1]. The rule exploits a convergence of three programming aspects—structured iteration, function abstraction, and parametricity—to achieve intermediate structure removal in a single transformation step.

One shortcoming of the technique is that, up to now, it has not been clear how to fuse `zip`. The purpose of this paper is to extend the *foldr-build* technique, showing how both branches of `zip` can be fused concurrently.

The paper is organized as follows. We review the *foldr-build* technique, then we introduce a new form of `foldr` that enables coroutining, and show how this provides a solution to the `zip` problem. Then we present alternative models for coroutining folds, discuss implementation, and demonstrate full fusion. Finally, we give axiomatization for an abstract type that unifies the various models and we initiate a study of the abstract type that we hope will lead to a proof of correctness of the new fusion algorithm.

2 Original Foldr-Build

The key goal of *foldr-build* is to achieve fusion in one step. It achieves deforestation without the need for knot-tying search or extra analysis that are present in many other techniques. While *foldr-build* applies to other data structures, it has been used most extensively with lists. We follow this trend, and for most of the rest of the paper will focus on lists alone.

The *foldr-build* idea has four key components:

- List producing functions are abstracted with respect to *cons* and *nil*;
- The list is reconstructed with a known function `build` defined by `build g = g (:) []`;
- Polymorphism is used to be sure that abstraction has been complete;
- List consumers are defined using `foldr`.

An example of a definition that exhibits these characteristics is

```
map f xs = build (\c n -> foldr (c . f) n xs).
```

It has often been observed that the effect of `foldr` is to replace the *conses* and *nil* of its list argument with the function areguments provided. The *foldr-build* theorem asserts that if a list producer has been properly abstracted with respect to its *conses* and *nil*, then the effect of `foldr` on the list can be achieved simply by function application. This is expressed by the following theorem.

Theorem 1. [2] *If for some type* `a` *a function* `g` *has the polymorphic type*

```
g :: forall b . (a->b->b) -> b -> b
```

then for all `k` *and* `z` *we have that*

```
foldr k z (build g) = g k z.
```

The proof follows pretty immediately from the parametricity theorem implied by the type of `g`.

To see the power of the theorem, consider the following additional definitions

```
sum xs = foldr (+) 0 xs
down m = build (\c n ->
           let loop x =
             if x==0 then n else c x (loop (x-1))
           in loop m)
```

as used in the expression `sum (map sqr (down z))`. Expanding the definitions and applying the *foldr-build* theorem proceeds as follows.

```
sum (map sqr (down z))
  = foldr (+) 0 (build (\c n ->
      foldr (c . sqr) n (down z)))
  = foldr ((+) . sqr) 0 (down z)
  = let loop x =
      if x==0 then 0 else sqr x + loop (x-1)
    in loop z
```

In just a couple of simple transformation steps, we have eliminated the intermediate data structures, and obtained a purely recursive definition of the computation. The cost is simply the somewhat arcane style of function definition required for list generators/consumers, but this can largely be obtained automatically from the more usual definitions [6].

Foldr-build works beautifully for a wide range of list processing functions but, unfortunately, `zip` presents us with one big fly in the ointment. By using a higher-order instance of `foldr` we can define `zip` as a fold on one of the input lists; the other list is passed as an inherited attribute as follows:

```
zip xs ys = build (\c n ->
  let c1 x g [] = n
      c1 x g (y:ys) = c (x,y) (g ys)
  in
      foldr c1 (\ys -> n) xs ys)
```

Using this technique, we could define two asymmetric versions of `zip`. Then, by using one or the other of these we can fuse a left branch or a right branch computation, but not both branches at the same time. It pretty much became accepted folklore that `zip` cannot be defined as a fold on both branches at the same time. Now we know that is not the case.

Before we address coding `zip` as a fold over both inputs concurrently, we will digress for a moment to explore further the relationship between `build` and `foldr`. Let us define `foldr'` to be the same as `foldr` except that the list argument comes first. We could then give it the following (non-Hindley-Milner) type:

```
foldr' :: [a] -> (forall b . (a->b->b) -> b -> b)
```

That is, `foldr'` maps from the list domain to the same type that arguments to `build` have. Furthermore, the foldr-build rule translates into `foldr' (build g) = g`. Given that `build (foldr' xs) = xs` (simply expand the definitions), we see that `foldr'` is the inverse to `build`. This inverse relationship is obscured with the order of arguments used by `foldr`.

Consequently, when we come to define new versions of `foldr` we will adopt the practice of placing the list argument first.

3 Coroutining Folds

The key step towards including `zip` within the scope of *foldr-build* comes by giving `fold` an extra argument that behaves as a coroutining continuation:

```
fold []     c n = \k -> n
fold (x:xs) c n = \k -> c x (k (fold xs c n))
```

We will explicitly avoid any premature commitment to types at this stage: the implication of types will come later. As it is, the definition above would be rejected by *Haskell*, but let us proceed nonetheless. The intuition behind the definition is that the cdfold function receives an interleaving continuation `k`, and applies the "*cons* function" `c` both to `x` and to the result of applying `k` to the recursive call of `fold`. Note that the continuation `k` is *not* provided as an argument in the recursive call of `fold`. Instead, `k` takes the recursive call of `fold` as its own interleaving continuation, and would be expected to call it with a new continuation.

As an example, consider the case where the interleaving continuation is another instance of `fold` itself.

```
fold [x1,x2,x3] c n (fold [y1,y2] d m)
  = c x1 (fold [y1,y2] d m (fold [x2,x3] c n))
  = ...
  = c x1 (d y1 (c x2 (d y2 (c x3
            (fold [] d m (fold [] c n))))))
  = c x1 (d y1 (c x2 (d y2 (c x3 m))))
```

The `fold`s over the `x` and `y` elements each invoke the other in turn, and thereby produce the interleaving effect.

There are two useful operators on interleaving computations, defined as follows.

```
self k = k self
(f # g) k = f (g # k)
```

The operator `self` acts as a trivial continuation, `#` as a composition operation. So for example, the expression `fold xs c n self` is equal to `foldr c n xs`. At each level of the recursion, `self` simply hands control back to `fold`. The role played by `#` is complementary. If we need to interleave three or more computations, we do so using `#`. The combined computation `f # g` when given a continuation `k` invokes `f` with continuation `g # k`. When that continuation is invoked (assuming it ever is) then `g # k` will be applied to some follow-on from `f`, `f'` say. Then `(g # k) f'` will invoke `g` with continuation `k # f'`, and so on. A simple exercise which demonstrates this behavior is to reduce the expression

```
(fold [x1,x2,x3] c n #
   fold [y1,y2] d m) fold [z1,z2] e p
```

The result is the interleaving of the computations over the three lists.

Theorem 2. [4] *The operation* `#` *is associative, and* `self` *is a left and right identity for* `#`.

Using this result, the interleaving of folds over two lists above can be written as

```
(fold [x1,x2,x3] c n  #  fold [y1,y2] d ) self.
```

It should be apparent, now that we have a version of `fold` that can interleave computations on multiple lists, that `zip` is now definable in terms of independent `fold`s on its two branches, thus:

```
zip xs ys =
  (fold xs c []  #  fold ys c Nothing) self
    where
      c x Nothing        = []
      c x (Just (y,xys)) = (x,y) : xys
      d y xys = Just (y,xys)
```

Note that while `c` is strict—the presence or absence of a `y` element is needed to know whether to produce a new output element—the `d` function is not, thus `zip` has its usual lazy behavior.

4 Typing

The definition of `fold` above would be rejected by *Haskell*'s typechecker on the grounds of unification requiring an infinite type. The type equation that needs to be solved is

```
H a b = H b a -> b,
```

which when expanded gives the infinite type

```
H a b = (((... -> a) -> b) -> a) -> b.
```

Interestingly, all occurrences of `b` are in positive (covariant) positions and all occurrences of `a` are in negative (contravariant) positions. In effect, `a` acts as an argument type, and `b` acts as a result. That is, the type `H a b` is a kind of function from `a` to `b`. Later on, we demonstrate that `H a b` can act like a layered stack of functions from `a` to `b`. We use the term *hyperfunction* to express this.

To code hyperfunctions in *Haskell* we introduce `H` as a *newtype*, and define appropriate access functions.

```
newtype H a b = Cont (H b a -> b)

invoke :: H a b -> H b a -> b
invoke (Cont f) k = f k

base :: a -> H b a
base p = Cont (\k -> p)

(<<) :: (a -> b) -> H a b -> H a b
f << q = Cont (\k -> f (invoke k q))

lift :: (a->b) -> H a b
lift f = f << lift f

self :: H a a
self = lift id

(#) :: H b c -> H a b -> H a c
f # g = Cont (\k -> invoke f (g # k))

run :: H a a -> a
run f = invoke f self

fold :: [a] -> (a -> b -> c) -> c -> H b c
fold [] c n     = base n
fold (x:xs) c n = c x << fold xs c n
```

The use of the constructor in the definition of `H` and the associated access functions obscure some of the definitions a little. The `(<<)` operator acts rather like a *cons* operator, taking a function element `f` and adding to the stack of functions `q`. Without types we could define it by `(f<<q) k = f (k q)`. The `lift` operator takes a normal function `f` and turns it into a hyperfunction by acting as `f` whenever it is invoked. If we were to expand its definition (and again present it untyped) we get `lift f k = f (k (lift f))`. Interestingly, the `self` operator is simply an instance of `lift`. The type of `#` makes it clear that it is acting as a composition operator. In fact, hyperfunctions form a category over the same objects as the base functions use, and `lift` is a functor from the base category into the hyperfunction category (Theorem 2). We shall return to this point later.

The (re-)definition of `fold` makes it clear that it is an instance of the usual `foldr` as follows. In fact, the following two equations hold

```
fold xs c n = foldr (\x z -> c x << z) (base n) xs
foldr c n xs = run (fold xs c n)
```

showing that `fold` and `foldr` are equivalent in the sense that one can be defined in terms of the other.

5 Fold-Build

As in the original *foldr-build* work, we now define a build function which ensures that the list generator function is suitably abstracted.

```
build ::
  (forall (b,c).(a->b->c) -> c -> H b c) -> [a]
build g = run (g (:) [])
```

Initially we hoped that, as in the standard *foldr-build* case described by Theorem 1, the parametric nature of the type of `g` would imply the fusion law

```
fold (build g) c n = g c n.
```

Unfortunately, the law is not true in such generality. Its failures, however, are of the nature that seems extremely unlikely to have adverse effects in applications. This leads us to conjecture the essential correctness of the law: *If a deforestation algorithm uses hyperfunctions and the* fold-build *law in a controlled way, then it will transform any program into an equivalent one.*

We will discuss the conjecture some more in later sections. Now, assuming it is safe to work with the *fold-build* law, we can start putting it into practice. As a first example, consider fusing the program

```
sum (zipW (*) (map sqr xs) (map inc ys))
```

where `zipW` is a `zipWith`-like function whose definition is similar to the definition of `zip` we saw earlier:

```
zipW f xs ys = build (zipW' f xs ys)
zipW' f xs ys c n =
  fold xs c1 n  #  fold ys c2 Nothing
    where
      c1 x Nothing        = n
      c1 x (Just (y,xys)) = c (f x y) xys
      c2 y xys = Just (y,xys)
```

The other list processing functions are defined using `build` and `fold` more or less as usual:

```
map f xs = build (\c n -> fold xs (c . f) n)
sum xs = foldr (+) 0 xs
```

The fusion proceeds through beta reduction and application of *fold-build*:

```
sum (zipW (*) (map sqr xs) (map inc ys))
= run
    (fold (zipW (*) (map sqr xs) (map inc ys)) (+) 0)
= run
    (fold (map sqr xs) c 0
     # fold (map inc ys) d Nothing)
      where
      c x Nothing        = 0
      c x (Just (y,xys)) = (x * y) + xys
      d y xys = Just (y,xys)
= run
    (fold xs (c . sqr) 0 # fold ys (d . inc) Nothing)
      where
        c x Nothing        = 0
        c x (Just (y,xys)) = (x * y) + xys
        d y xys = Just (y,xys)
= run
    (fold xs c 0 # fold ys d Nothing)
      where
        c x Nothing        = 0
        c x (Just (y,xys)) = (sqr x * inc y) + xys
        d y xys = Just (y,xys)
```

The intermediate lists produced by the two uses of `map` have both been fused away, even though they occurred in separate branches of the `zip`.

Stepping back to gain a wider perspective is probably useful at this point. Simple folds have been well studied. These are uses of fold at ground types, that is, that build non-function structures. In this style of use, the fold constructs synthesized attributes only, and `foldl` and `foldr` more or less act as duals to each other. Once we allow folds to produce functions as their results, we gain significant extra power. The function arguments allow us to model inherited attributes, and `foldl` is seen clearly as an instance of `foldr` (and not the other way around).

```
foldl c n xs = foldr (\x g z -> g (c x z)) id xs n
```

When we go further and allow folds to produce hyperfunctions, we allow coroutining which permits distinct fold computations to be interleaved. Thus we have to depart from the usual language of attribute grammars with inherited and synthesized attributes, and now talk of attributes that flow across the tree structure, between the nodes of different subtrees that are at the same level. It is in this setting that a (nearly) symmetric definition of `zip` becomes possible.

6 The Stream Model

The elements of `H` we have been using behave like a stream of functions: some work is performed at first, and then the remainder is given to the continuation. When the continuation reinvokes the remainder a little more work is done, and again the rest is given to its continuation. In other words, work is performed piece by piece with interruptions allowing for interleaved computation to proceed.

This intuition leads us to represent hyperfunctions explicitly as streams. We use the name `L` for this model.

```
data L a b = Cons (a->b) (L a b)

(#) :: L b c -> L a b -> L a c
(Cons f fs) # (Cons g gs) = Cons (f . g) (fs # gs)

lift :: (a->b) -> L a b
lift f = Cons f (lift f)

run :: L a a -> a
run (Cons f fs) = f (run fs)

(<<) :: (a->b) -> L a b -> L a b
f << p = Cons f p

base :: a -> L b a
base x = lift (const x)

invoke :: L a b -> L b a -> b
invoke fs gs = run (fs # gs)
```

The operation `fold` is defined in terms of `base` and `<<` as in Section 4.

One interesting aspect of this model is that `run` is more naturally primitive than `invoke`, whereas in the previous model the opposite was the case. Furthermore, the identity and associativity laws between `#` and `self` (still defined by `self = lift id`) become very easy to prove just by list induction and properties of composition. In contrast, the corresponding theorems about the `H` model turned out to be rather challenging, to say the least [4].

The stream of functions acts like a fix-point waiting to happen. The stream can be manipulated in two different ways: either the functions are interspersed with another stream of functions by means of `#`, or all the functions are composed together by `run`. In this way, `run` ties the recursive knot, and removes opportunities for further coroutining.

The behavior of `fold` in this model is instructive, as seen in this example:

```
fold [x1,x2,x3] c n
 = Cons (c x1) (Cons (c x2)
     (Cons (c x3) (Cons (Const n) ...)))
```

where the dots indicate an infinite stream of `Const n`. Thus `fold` turns a list of elements into an infinite stream of partial applications of the `c` function to the elements of the list. At this point we might pause and ask whether we have actually gained anything. After all, we have simply converted a list into a stream. Even worse, the much vaunted definition of `zip` turns out to be defined in terms of `#`, which is defined just like `zip` in the first place! However, the stream is merely intended to act as a temporary structure which helps the compiler perform its optimizations. As with `H`, the `L` model can be used for *fold-build* fusion, and the stream structures are optimized away. Any that exist after the fusion phase ought to be removable by inlining the definition of `run`. In other words, the stream structure simply should not exist at run-time—its purpose is compile-time only.

To demonstrate this really is feasible, we introduce a variation of this model in the next section, and describe our implementation experience using it.

7 Fusing with Recursive Generators

One strength of the original *foldr-build* is that it could fuse with recursive generators for lists, often ending up with computations that had no occurrence of lists whatsoever—the initial review in Section 2 above contained one such example. In one sense it was easy to achieve this. By restricting fusion to act only ever along a single branch of zip-like functions (i.e. functions consuming multiple input lists), we always ended up with a single ultimate origin for the computation. All *foldr-build* had to do was to place the subsequent processing of list elements into the appropriate places of this (arbitrarily recursive) computation and we were done.

In contrast, multi-branch fusion may have many sources each acting as a partial origin of the computation. To achieve fusion, we would need to combine multiple (abitrarily recursive) generators. This is very hard in general. One way to make the problem tractable is to focus on recursive generators that are state machines, also known as tail calls or anamorphisms. This leads us to yet another model where we represent the stream of functions of the L model as a state machine, hiding the type of the state by using an existential type.

```
data A a b =
  forall u . Hide (u -> Either b (a -> b,u)) u

(#) :: A b c -> A a b -> A a c
Hide g x  #  Hide g' x' =
  Hide
    (\(z,z') -> case g z of
         Left n     -> Left n
         Right(f,y) -> case g' z' of
           Left m        -> Left (f m)
           Right(f',y')  -> Right (f . f', (y,y')))
    (x,x')

lift :: (a->b) -> A a b
lift f = Hide (\u -> Right (f, u)) (error "Null")

run :: A a a -> a
run (Hide f v) = loop v
  where
    loop x = case f x of
               Left n     -> n
               Right(h,y) -> h (loop y)

fold :: [a] -> (a -> b -> c) -> c -> A b c
fold xs c n =
  Hide (\ys -> case ys of
                 []     -> Left n
                 (w:ws) -> Right (c w,ws))
       xs
```

The form of the type declaration is a little misleading and needs to be understood correctly. It gives `Hide` the type

```
Hide :: forall u .
  (u -> Either b (a -> b,u)) -> u -> A a b
```

while introducing `Hide` as the sole constructor of `A a b`.

Interestingly, once again the choice of the model has affected which primitives are natural to define. This time the `(<<)` operator does not look the most natural and `fold` is defined directly.

Let's put these definitions to work. We define a couple of typical generators.

```
down z = build (down' z)

down' :: Int -> (Int -> b -> c) -> c -> A b c
down' w c n = Hide (\z -> if z<=0 then Left n
                          else Right (c z,z-1))
                   w

upto i j = build (upto' i j)

upto' :: Int -> Int -> (Int -> b -> c) -> c -> A b c
upto' a b c n =
  Hide (\(i,j) -> if i>j then Left n
                  else Right (c i,(i+1,j)))
       (a,b)
```

and use them in fusing the expression `sum (zipW (*) (upto 2 10) (down 6))`. The various steps are given in Figure 1, but we note that the method is sufficiently powerful to remove all the intermediate lists in this example, leaving the recursive pattern

```
loop ((2,10),6)
  where
  loop ((i,j),z) = if i>j then 0 else
                   if z<=0 then 0 else
                   (i*z) + loop ((i+1,j),z-1)
```

Implementation

An early version of these ideas was presented to the IFIP working group on functional programming (WG2.8, 1999) immediately after Simon Peyton Jones had presented the new rule-based transformation engine built into the Glasgow Haskell Compiler GHC [7]. After being challenged to demonstrate the workability of both sets of ideas by using the rule system to implement this new version of *fold-build* (and being given only one evening in which to do so!), thanks entirely to Simon's skill and the power and flexibility of the GHC Rule Engine, this was accomplished.

We began by using a slightly different model than the `A` model above, relying on constant functions to achieve the effect of ending the streams after a finite time:

```
data A' a b =
  forall u . Hide (u -> (a -> b,u)) u
```

The corresponding definitions of the various operators for the `A'` model are simpler than for `A` above. We added all the new definitions of the operators, including `fold` and `build`, we defined functions such as `map`, `zip`, and `upto` in terms of them, and we added the new *fold-build* rule. The result was only partially successful. It turned out that GHC was unable to spot that it could safely perform some useful simplifying transformations, and so the fusion process was halted prematurely.

In order to fix this, the `A` model was used. By using explicit constructors, the compiler could tell easily that case expressions would cancel out with constructors, and the fusion process ran to completion.

8 The Type of Build

We now turn to address the correctness of the new *fold-build* law. The law as stated in Section 5 can fail for more than

```
sum (zipW (*) (upto 2 10) (down 6))
 = run (fold (zipW (*) (upto 2 10) (down 6)) (+) 0)
 = run (zipW' (*) (upto 2 10) (down 6) (+) 0)
 = run (fold (upto 2 10) c 0  #  fold (down 6) d Nothing)
   where
     c x Nothing        = 0
     c x (Just (y,xys)) = (x * y) + xys
     d y xys = Just (y,xys)
 = run (upto' 2 10 c 0  #  down' 6 d Nothing)
   where
     c x Nothing        = 0
     c x (Just (y,xys)) = (x * y) + xys
     d y xys = Just (y,xys)
 = run (Hide (\(i,j) -> if i>j then stop 0 else (c i,(i+1,j)))
             (2,10)
        #
        Hide (\z -> if z<=0 then stop Nothing else (d z,z-1))
             6)
   where
     c x Nothing        = 0
     c x (Just (y,xys)) = (x * y) + xys
     d y xys = Just (y,xys)
 = run (Hide (\((i,j),z) -> case if i>j then Left 0 else Right (c i,(i+1,j)) of
                               Left n      -> Left n
                               Right (f,y) ->
                            case if z<=0 then Left Nothing else Right (d z,z-1) of
                               Left m        -> Left (f m)
                               Right (f',y') -> (f . f', (y,y')))
               ((2,10),6))
   where
     c x Nothing        = 0
     c x (Just (y,xys)) = (x * y) + xys
     d y xys = Just (y,xys)
= run (Hide (\((i,j),z) -> if i>j then Left 0 else
                           let (f,y) = (c i,(i+1,j)) in
                           if z<=0 then Left (f Nothing) else
                           let (f',y') = (d z,z-1) in
                           (f . f', (y,y')))
             ((2,10),6))
  where
    c x Nothing        = 0
    c x (Just (y,xys)) = (x * y) + xys
    d y xys = Just (y,xys)
 = run (Hide (\((i,j),z) -> if i>j then Left 0 else
                            if z<=0 then Left 0 else
                            (\w -> (i*z)+w, ((i+1,j),z-1)))
       ((2,10),6))
 = loop ((2,10),6)
   where
     loop ((i,j),z) = if i>j then 0 else
                      if z<=0 then 0 else
                      (i*z) + loop ((i+1,j),z-1)
```

Figure 1: An example of fusion.

one reason. In this section we fix a typing problem. Other points are discussed in the next section.

A counterexample to the law is provided by the function

```
bad_g c n = (\x -> c 5 bottom) << bottom.
```

Indeed, `fold (build bad_g) c n = c 5 << bottom`, and this is different from `bad_g c n`.

Unfortunately this is not just a technical problem, but is directly observable at the level of program fusion. Consider the expression

```
zip (build one) (build bad_g)
one c n = c 1 << base n
```

Without fusion, the expression evaluates to `[(1,5)]`. On the other hand, after fusion, we get the expression

```
let c x Nothing        = []
    c x (Just (y,xys)) = (x,y) : xys
    d y xys = Just (y,xys)
in
  run (one c [] # bad d Nothing)
```

where the local functions `c` and `d` come from the definition of `zip`. This expression evaluates to `(1,5):`$\perp$ which is less defined than the expression we started with. The problem is that `bad_g` fails to return control back to the first list, and so it is not able to complete the list.

The undefined value `bottom` is at the heart of the problem. Since it exists in every *Haskell* type, the function `bad_g`, knowing about it, is at liberty to manufacture the unfortunate value `c a bottom`. A richer type system can prevent the above from happening. For example, one can use the type system of Launchbury and Paterson [5] that distinguishes types with `bottom` (*pointed* types) from general types that might not contain `bottom`. Noticing that the construction of the recursively defined type `H b c` requires only that `c` be pointed can be used to give a more constrained type to the source domain of the function `build`:

```
build :: (forall b . Pointed c .
            (a->b->c) -> c -> H b c) -> [a]
```

The extended type system uses a separate quantifier `Pointed` for pointed types. Now, `bad_g` is ill-typed.

With this intervention in the type system, we can characterize the elements for which the *fold-build* law is true. Let `G(a)` denote the domain

```
forall b . Pointed c . (a->b->c) -> c -> H b c
```

and let `S` be the subset of `G(a)` defined coinductively by the condition that if `g` is in `S`, then there exists `f` and there exists `w` in `S` such that

```
g c n = f c n << w c n
```

The elements of `G(a)` belonging to `S` will be called *linear.*

Theorem 3. *An element* `g` *of* `G(a)` *satisfies the identity*

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

if and only if it is linear.

An informal analysis of the fusion algorithm suggests that the algorithm can never access other than linear elements of `G(a)`. Combined with Theorem 3, this observation corrobates the conjecture we made in Section 5. We leave Theorem 3 without proof here.

9 The Abstract Type of Hyperfunctions

Now that we have three models that support programming with coroutining continuations, it is natural to ask about the core functionality provided by all of them. So we embark on a study of an abstract type. We give the axiomatics satisfied by all three models. We establish some connections between the models and obtain some precise results about "linear" models which seem to be the most relevant in this context. The upshot of this research is our thesis that all models of the abstract type are potentially usable in deforestation algorithms, and that the implementors should feel free to search among various models and choose those that do the best job.

9.1 Axiomatics

Let us use the notation `K a b` for the abstract type of hyperfunctions. We begin by requiring that `K a b` be functorial in its two arguments (contravariant in the first, covariant in the second) and that the domains `K a b` can be regarded as the arrow sets of a category that contains our base category of domains as a subcategory. All this (and more) is achieved by specifying the primitive operators

```
primitive  (#) :: K b c -> K a b -> K a c
primitive  lift :: (a->b) -> K a b
primitive  run :: K a a -> a
```

which must satisfy the following conditions:

```
axiom(1)  (f # g) # h = f # (g # h)
axiom(2)  f # self = f = self # f
axiom(3)  lift (f . g) = (lift f) # (lift g)
axiom(4)  run (lift f) = fix f
```

where `self :: K a a` is defined by `self = lift id`. Thus, `lift` is a functor and viewing `lift f` as "`f` as a hyperfunction", we see that the operation `#` extends the composition and `run` extends the fixpoint operator. It is also a simple matter to check that the following definition of `mapK` makes `K` itself a functor.

```
mapK :: (a'->a) -> (b->b') -> K a b -> K a' b'
mapK r s f = (lift s) # f # (lift r)
```

We can now define

```
invoke :: K a b -> K b a -> b
invoke f g = run (f # g)
base :: b -> K a b
base k = lift (const k)
```

and it follows that `run f = invoke f self`, showing that `invoke` could replace `run` as a primitive.

The system we have built so far has a trivial model in which `K a b = a -> b`, the composition `#` is the ordinary function composition, `lift f = f`, `self = id` and `run = fix`. However, in order to bring continuations into play, we need to add the primitive operation

```
primitive  (<<) :: (a->b) -> K a b -> K a b
```

required for defining `fold`:

```
fold :: [a] -> (a -> b -> c) -> c -> K b c
fold [] c n     = base n
fold (x:xs) c n = c x << fold xs c n
```

The interaction of the new primitive with the others is described by the following axioms.

```
axiom(5) (f << p) # (g << q) = (f . g) << (p # q)
axiom(6) lift f = f << lift f
axiom(7) run ((f << p) # q) = f (run (q # p))
```

This finishes the definition of the abstract type of hyperfunctions. The trivial model is no longer possible.

We will use the term *hyperfunction model* for models of our abstract type. They are functors $\mathrm{K}\colon \mathcal{D}^{\mathrm{op}} \times \mathcal{D} \to \mathcal{D}$ (where $\mathcal{D}$ is the underlying category of domains) with the additional structure consisting of operations `#`, `run`, `lift` and `<<` satisfying Axioms 1–7.

All hyperfunction models have the property that distinct functions remain distinct when regarded as hyperfunctions. Any world of hyperfunctions thus contains a faithful copy of the world of ordinary functions:

Theorem 4. *The functor* `lift` *is faithful. (In other words, if* `lift f = lift g` *then* `f = g`*.)*

Proof. Define

```
project :: K a b -> (a -> b)
project q k = invoke q (base k)
```

It suffices to prove that `project` is a left-inverse of `lift`, i.e. that `project (lift f) = f`. Indeed,

```
project (lift f) x
  = invoke (lift f) (base x)
  = run (lift f # base x)
  = run ((f << lift f) # base x)
  = f (run (base x # lift f))
  = f (run (lift (const x) # lift f))
  = f (run ((const x << base x) # lift f))
  = f (const x (run (lift f # base x)))
  = f x
```

□

To study relationships between hyperfunction models, it is useful to view the models themselves as objects of a category. The morphisms are natural transformations `t :: K a b -> K' a b` preserving all the structure. More precisely, `t` must satisfy

```
axiom(M1)  t (f # g) = t f #' t g
axiom(M2)  t (lift f) = lift' f
axiom(M3)  run f = run' (t f)
axiom(M4)  t (f << q) = f <<' t q
```

We will use morphisms below in the discussion of linear models.

9.2 The Three Models

Theorem 5. `H`*,* `L`*, and* `A` *are hyperfunction models.*

We leave out the details of the proof. The proof is simplest for the `L` model. Checking the axioms there is straightforward. In fact, all the axioms except the seventh are also true in the "almost model" `L' a b = [a -> b]` of (finite or infinite) lists.

As already indicated, it is far from being obvious that `H` is a model. Most of the difficulty is contained in Theorem 2 above.

In the case of the `A` model, we need to give a definition of `<<`. Here it is:

```
(<<) :: (a -> b) -> A a b -> A a b
p << (Hide f v) =
  Hide (\x -> case x of
                Nothing -> Right(p, Just v)
                Just w  -> case f w of
                    Left n     -> Left n
                    Right(h,y) -> Right(h, Just y))
        Nothing
```

Equality of terms of type `A a b` is proved by bisimulation. Checking the axioms is an excruciating exercise.

9.3 Fold and build

To use fusion, we need to be able to write `foldr` in terms of `fold`. Here is the requisite law.

Lemma 1. `foldr c n xs = run (fold xs c n)`*, in any hyperfunction model.*

Proof. Using fixpoint induction on `xs`, it suffices to prove the result for $\perp$ and `[]` and also to show that `foldr c n xs = run (fold xs c n)` implies

```
foldr c n (x:xs) = run (fold (x:xs) c n)
```

When `xs` is $\perp$, both sides evaluate to $\perp$. For the case when `xs=[]` we need to check that `run (base n) = n`, which follows from the axioms as in the last five lines of the proof of Theorem 4. Finally, for the last part, we have

```
run (fold (x:xs) c n)
  = run (c x << fold cs c n)
  = c x (run (fold cs c n))
  = c x (foldr c n xs)
  = foldr c n (x:xs)
```

□

Every hyperfunction model has its own `build` function:

```
build :: (forall b . Pointed c .
             (a->b->c) -> c -> K b c) -> [a]
build g = run (g (:) [])
```

As before, `fold` can be regarded as a function going in the opposite direction from `build`. In fact, an immediate consequence of Lemma 1 is that `build` is a left inverse of `fold`:

Corollary 1. `build (fold xs) = xs`*, in any hyperfunction model.*

Thus, the two functions are full inverses of each other depending exactly on the truth of the fusion law:

```
fold (build g) c n = g c n.
```

Unfortunately, we cannot expect the law to be true in general. To see why, notice first that Lemma 1 remains true if `base` in the definition of `fold` is replaced by any function `base'` which has the property `run (base' n) = n`. If `base'` is a different function from `base`, it would follow that there are two distinct functions `fold` and `fold'` which are both right inverses for `build` and, consequently, that the *fold-build* law is not true. An example when this actually happens is provided by the model `A`, with

```
base' n = Hide (\u -> Left n) (error "Null"),
```

which is clearly not the same as

```
base n = Hide (\u -> Right (const n, u)
              (error "Null").
```

In fact, we have used this function `base'` in place of `base` in the definition of `fold`! One can check that, in the model `A`,

```
fold [] c n     = base' n
fold (x:xs) c n = c x << fold xs c n
```

Thus, it turns out that a single model can support more than one meaningful `fold` function. Indeed, it seems reasonable to accept in the definition of `fold` any function `base'` that behaves like a "constant" hyperfunction in the sense of satisfying the identity

```
 invoke (base' n) q = n.
```

In `H`, the only function satisfying this identity is the standard `base`, so there are no alternative `fold`s for `H`. Of course, this does not mean that the *fold-build* law holds in `H`. In fact, it seems appropriate to ask now if there is any model in which the law is true. We believe that the submodel `Q` of `H` defined below is such. However, more important than the quest for models satisfying the law is the task of proving our conjectured essential correctness—that the restricted use of the law is safe. We leave it to a future research to give a precise meaning of "restricted" and to prove the safety. All uses of the law that we make in this paper will fall into that class.

9.4 Linear models

The stream model is the simplest of our three; now we see that it is the simplest in general.

Theorem 6. *In the category of hyperfunction models,* `L` *is an initial object.*

Proof. Suppose `inK :: L a b -> K a b` defines a morphism from `L` to `K`. Axiom (M4) reads `inK (Cons f fs) = f << (inK fs)` and so it defines `inK`. Thus, `inK` is the only morphism from `L` to `K` if it actually is a morphism. We need to check the other three axioms for morphisms. They all turn out to be true; the proofs are straigtforward, using stream induction. The proof of Axiom (M1) relies in particular on Axiom 5. Similarly, Axioms 6 and 7 are crucial for the proof of morphism axioms (M2) and (M3) respectively. □

Let us call a hyperfunction model `K` *linear* if `inK` is a full functor (for every `a` and `b`, the function `inK :: L a b -> K a b` is onto). Thus, every hyperfunction model contains a *linear part*—the image of `inK`.

The model `A` is not linear since, for example, the functions `base' n` are not in the image of `inA`. The model `H` is not linear either, although it takes more effort to prove. (In fact, `H` is huge; it is just more difficult to reason about.)

We can also use `H` to see that `L` is not the unique linear type. For example, by a simple computation, `inH (Cons (const n) xs) = base n`, so all streams in `L a b` whose first member is `const n` are mapped by `inH` to the same linear hyperfunction in `H a b`. Thus, the linear part of `H` is a linear hyperfunction model non-isomorphic to `L`.

The just observed effect of constant functions in streams motivates the following definition. Let us say that two streams are *similar* if they are either equal or have a common finite prefix whose last member is a constant function. This is an equivalence relation. We define `Q a b` to be the domain whose elements are its equivalence classes.

Theorem 7. `Q` *is a hyperfunction model, with operations induced from* `L`*. Moreover,* `Q` *is the linear part of* `H`*.*

Proof. The first statement of the theorem can be proved directly in a straightforward manner. It also follows from the second statement, which is equivalent to saying that two streams have the same image under `inH` if and only if they are similar. We proceed to prove this statement and we claim that it follows from the following: In `H`, the equality `f << p = f' << q` occurs if and only if `f=f'` and either `f` is constant, or `p=q`. Omitting the proof of the claim, which is a simple inductive argument in both directions, we turn to the proof of the last equivalence. The "if" part follows from the already mentioned fact `const n << p = base n`.

For the "only if" part, assume `f << p = f' << q`. Using the computation

```
invoke (f << p) (const n)
  = f (invoke (const n) p) = f n
```

we can conclude `f n = f' n` for every `n`, so `f = f'`. Now our goal is to prove that the assumption `f << p = f << p'` implies that `f` is constant or that `p=p'`. Suppose neither is true. Then `invoke p q` $\neq$ `invoke p' q` for some `q` and `f n` $\neq$ `f n'` for some `n` and `n'`. Define `k` by

```
k = Cont(\r -> case invoke r q == invoke p q of
                 True  -> n
                 False -> n')
```

Then

```
invoke (f << p) k = f (invoke k p) = f n
invoke (f << p') k = f (invoke k p') = f n'
```

and we are done. □

As already mentioned, `Q` is a candidate for a model where (unrestricted) *fold-build* law holds. The prominence of `Q` comes from its being "optimal" among linear models. Precisely, in the subcategory (really a preorder) of linear hyperfunction models, `Q` is a terminal object. This is a direct consequence of the following result.

Theorem 8. *Let* `K` *be an arbitrary hyperfunction model and* `inK :: L a b -> K a b` *as before. If two streams have the same image under* `inK`*, then they are similar.*

Proof. Suppose `f` and `g` are non-similar streams in `L a b`. Then we can write

```
f = h_1 << ... << h_n << p << f'
g = h_1 << ... << h_n << q << g'
```

where `h_i` are non-constant, and `p` and `q` are non-equal. We can reduce this more general situation to the case of `L Bool Bool`, where all functions `h_i` are the identity. More precisely, there exist `r :: Bool -> a` and `s :: b -> Bool` such that

```
s # f # r = id << ... << id << p' << f''
s # g # r = id << ... << id << q' << g''
```

and `p'` and `q'` are non-equal. Let

```
t = id << id << ... << id << base v
```

with n+1 occurrences of id and with v being a fixed boolean value. We have

```
s # f # r # t = id << ... << id << p' << base v
s # g # r # t = id << ... << id << q' << base v
```

and so

```
run (s # f # r # t) = p' (run (base v)) = p' v
run (s # g # r # t) = p' (run (base v)) = q' v
```

By assumption, for some v these two values are non-equal and it follows that `run (inK s # inK f # inK r # inK t)` and `run (inK s # inK f # inK r # inK t)` are non-equal, so `inK f` and `inK g` are not equal. □

10 Conclusion

The original motivation for this paper was to broaden the power of the *fold-build* fusion technique to be able to handle multiple input lists. In doing so, we stumbled on two other insights we did not expect. First, we came to realize that the `fold` function is even more powerful than we had previously thought. In particular, it came as a palpable shock that `fold` was able to express interleaving computations. The view of `fold` as a generic expression simply of inherited and synthesized attributes over tree shaped structures had become quite deeply ingrained. Whether this understanding of the coroutining capability of `fold` will lead to new functions and techniques remains to be seen, but it cannot but help in broadening our perspectives.

Secondly, even though we have moved to use other models as well, we have found the original hyperfunctions fascinating in their own right. They have been devilishly tricky to reason about directly, but now we know that they form a category, have a weak product and seem to fit nicely into Hughes arrow class [3]. Again, whether they will turn out to be useful in other applications remains to be seen.

As to the main purpose of the paper: we have at last demonstrated that many occurrences of `zip` can be eliminated using the *fold-build* technique, leading to the fusion of multiple list generation routines. Our implementation is simple and it seems to work well, but as yet it is not clear how well in would work in large examples. The original *foldr-build* work was followed by a thesis which addressed the various techniques needed to make it truly practicable [1], and there is no reason to assume this more complex version would be any easier.

Establishing correctness of our method turns out to be an interesting problem in itself. Since the crucial *fold-build* law does not hold in general, it seems unlikely that the method will be proved correct by simple reasoning. On the other hand, it seems clear that the weaknesses of the law will not be exposed in the limited context of deforestation algorithms. Attempts to turn this intuitive understanding into proof are underway.

Unfortunately, it appears that the inclusion of `zip` into *fold-build* may have come at the cost of eliminating `reverse`. The original technique handled `reverse` beautifully [6], but as yet we have been unable to integrate it into the new framework. This leads to the prospect of the compiler having to choose—perhaps dynamically—between a `zip`-friendly and a `reverse`-friendly version of *fold-build*. Whether there is a single framework which encompasses both is unclear.

11 Acknowledgements

We are privileged to belong to the PacSoft research group at OGI; we received lots of helpful feedback and suggestions, in particular from Bruno Barbier and John Matthews. Participants at the WG2.8 working group meeting also provided very useful feedback,especially Richard Bird, Ralph Hinze and Erik Meijer. Finally, as noted in the text above, Simon Peyton Jones was the key driver in implementing these ideas within his rule-based transformation system for GHC.

References

[1] A. J. Gill. *Cheap Deforestations for Non-strict Functional Languages (Ph. D. Thesis).* University of Glasgow, 1996.

[2] A. J. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[3] J. Hughes. Generalising monads to arrows. *Science of Computer Programming.* to appear.

[4] S. Krstić and J. Launchbury. A category of hyperfunctions. Preprint, January 2000.

[5] J. Launchbury and R. Patterson. Parametricity and unboxing with unpointed types. In *Proceedings the Sixth European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 1996.

[6] J. Launchbury and T. Sheard. Warm fusion: deriving build-catas from recursive definitions. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, June 1995.

[7] S. L. Peyton Jones. Private communication.