Coroutining Folds with Hyperfunctions

J. Launchbury, S. Krstic, and T. E. Sauerwein

Fold functions are a general mechanism for computing over recursive data structures. First-order folds compute results bottom-up. With higher-order folds, computations that inherit attributes from above can also be expressed. In this paper, we explore folds over a form of recursive higher-order function, called hyperfunctions, and show that hyperfunctions allow fold computations to coroutine across data structures, as well as compute bottom up and top down. We use the compiler technique of foldr-build as an exemplar to show how hyperfunctions can be used.

1 Dedication

It is a privilege to submit a paper for the Festschrift symposium held to honor Dave Schmidt’s lifetime of contributions on the occasion of his 60th birthday. Many years ago, as a fresh PhD student, Dave’s excellent book on Denotational Semantics [Sch86] opened my eyes to rich possibilities of building functions over functions—recursively!—and our continued interactions over the years were always insightful. So it seemed appropriate to offer a paper whose foundations rely on the same mathematical models that Dave so ably expounded all those years ago, constructing recursive function spaces in a way that is not possible in traditional set-theoretic models. The paper is a revision of an earlier (unpublished) paper [LKS00] whose ideas deserved to see the light of day. – John Launchbury, 2013.

2 Introduction

Folds have been popular for a long time. At the one end of spectrum, they are presented in early classes introducing functional programming. At the other end, they form the foundation of Google’s famous world-scale map-reduce computational engine. In this paper, we play with the idea of folds. We use them to introduce and explore a category of coroutining functions, which we style hyperfunctions. We do so by tracing the story of a fascinating technique in code fusion, and show how hyperfunctions are able to open up apparently closed doors. Thus, while our story will be about fusion, our narrative purpose is actually to explore hyperfunctions. We will use the Haskell programming language as our setting.

Code fusion—automatic removal of intermediate computational structure—holds a promise of providing the best of two worlds: programming with the structures enables concise and modular solution to problems; and the removal of the structures provides efficient run-time implementations [Wad90]. One particularly enthralling technique is called the foldr-build rule [GLPJ93]. 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. The transformation was so effective that it was used for many years within the popular Glasgow Haskell compiler (GHC). However, a significant shortcoming of the technique is that it has not been clear how to extend it to fuse zip [LS95].

In this paper, we will introduce hyperfunctions, lift the foldr-build technique over hyperfunctions, and show how this enables both branches of zip can be fused concurrently. Thus, even though the foldr-build approach has been eclipsed in recent years by stream fusion techniques (which are able to handle zip without problem) [CLS07], it is a intriguing demonstration of the power of hyperfunctions that they are
able to overcome a previous shortcoming in what was a significant method for many years. Furthermore, given the prevalence of fold-like computational structures, folds over hyperfunctions may find other applications in due course.

3 Original Foldr-Build

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

The foldr-build idea has four key components:

1. List producing functions are abstracted with respect to cons and nil: for example, the list \([1,2,3]\) is represented by the function \(\lambda c \, n \rightarrow c \, 1 \, (c \, 2 \, (c \, 3 \, n))\).
2. Lists are reconstructed with a fixed function \(\text{build}\), defined by \(\text{build} \, g = g \, (:) \, []\), where the \(g\) argument is an abstracted list such as written in Step 1.
3. Parametric polymorphism is used to be sure that abstraction has been complete; this is expressed by requiring \(\text{build}\) to have type \(\text{build} :: (\forall b . (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b) \rightarrow [a]\), which is an example of rank-2 polymorphism; and
4. List consumers are defined using \(\text{foldr}\).

An example definition that follows these principles is:

\[
\text{map} \, f \, xs = \text{build} \, (\lambda c \, n \rightarrow \text{foldr} \, (c \cdot f) \, n \, xs)
\]

To understand this definition, consider mapping a function \(f\) down the list \([1,2,3]\). Being explicit about the list constructors, the result is \(\text{(}: \, (f \, 1) \, \text{(}: \, (f \, 2) \, \text{(}: \, (f \, 3) \, [])\)), or alternatively, \(\text{(}: \cdot f) \, 1 \, \text{(}: \cdot f) \, 2 \, \text{(}: \cdot f) \, 3 \, \text{[]}\)). The effect of \(\text{foldr}\) is to replace the conses and nils of its list argument with the function arguments provided. In this case, the original conses get replaced with \(\text{(}: \cdot f)\), or as we are abstracting over the conses and nils, with \((c \cdot f)\).

The foldr-build theorem below asserts that if a list producer has been properly abstracted with respect to its conses and nil, then the effect of \(\text{foldr}\) on the list can be achieved simply by function application. This is expressed as follows.

**Theorem 1 (Foldr-build)** If \(g\) has the polymorphic type \(g :: (\forall b . (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow b) \rightarrow [a]\) then for all \(k\) and \(z\) it is the case that \(\text{foldr} \, k \, z \, (\text{build} \, g) = g \, k \, z\).

The proof follows easily from the parametricity theorem implied by the type of \(g\) [Wad89, GLPJ93, J03]. In effect, the polymorphism of \(g\) ensures that \(g\) behaves uniformly for every possible substitution of its arguments, and that there are no exceptional cases lurking in the code of \(g\) that may behave differently in different settings.

To see the power of the foldr-build theorem, consider the following definitions.

---

1 Sad, the presence of pseudo-polymorphic \(\text{seq}\) in Haskell actually causes the theorem to fail, because the resulting parametricity theorem is no longer quite strong enough, even though there is a general form of parametricity that works for \(\text{seq}\) [JuV03]. However, in practice a simple static check was enough to ensure that the bad situation did not arise.
map f xs = build (\c n -> foldr (c . f) n xs)
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)

where down n creates a list from n down to 1. We could use these definitions to rewrite the expression
sum (map sqr (down z)) as follows

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

In just a couple of rewriting steps we have a purely recursive definition of the computation, with no
intermediate lists. Obviously this is a simplistic example, but many examples of this form arise in prac-
tice, particularly when desugaring list comprehensions, or working with array expressions. And while
programmers would not themselves write definitions in this form, all the primitive list-processing func-
tions can be defined this way in the compiler’s prelude, and many other definitions can be automatically
transformed into this form [LS95].

3.1 Inverses

The usual type for foldr is

\[\text{foldr :: (a -> b -> b) -> b -> [a] -> b}\]

If we reorder the arguments to place the list argument first, add explicit quantifiers for the type variables,
and if we push the b quantifier in as much as possible, we get:

\[\text{foldr' :: forall a. [a] -> (forall b. (a -> b -> b) -> b -> b)}\]

Similarly, if we are explicit about the type of build we would write:

\[\text{build :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]}\]

Now the foldr-build theorem reduces to simply stating that (foldr' . build) = id. As it is also
trivially the case that (build . foldr') = id, we see that build and foldr' are inverses.

3.2 Higher-order Folds

The foldr-build technique works even for functions like reverse that are not expressible as first-order
folds. First we define reverse as a higher-order fold,

\[\text{reverse xs = (foldr (\x k ys -> k (x:ys)) id xs) []}\]

in which the fold computation constructs a function which is applied to the extra argument [] supplied
at the end, and then we abstract over the (:) and [] as follows,

\[\text{reverse xs = build (\c n -> (foldr (\x k p -> k (c x p)) id xs) n)}\]

In passing, we also α-renamed ys to p, as the name ys suggests a list element, yet in general the
abstracted list structure is polymorphic and may not be building a list. With this definition, list fusion
proceeds exactly the same as before. For example, doing a variation to the previous derivation starting
this time from the expression \[\text{sum (reverse (map sqr (down z))}\]

As before, this is a contrived example designed to show the form of the code fusion transformation.
sum (reverse (map sqr (down z)))
= foldr (+) 0 (build (\c' n' \rightarrow (foldr (\x k p \rightarrow k (c' x p)) id
  (build (\c n \rightarrow foldr (c . sqr) n (down z)))
  n'))))
= foldr (\x k p \rightarrow k (x+p)) id (build (\c n \rightarrow foldr (c . sqr) n (down z))) 0
= foldr (\x k p \rightarrow k (sqr x + p)) id (down z) 0
= (let loop x = if x==0 then (\p \rightarrow p) else (\p \rightarrow (let loop (x-1) (sqr x + p)) in loop m) 0
  in loop m)
= let loop x p = if x==0 then p else loop (x-1) (sqr x + p)
  in loop m

The result is a tight recursive definition with an accumulating parameter—we needed an η-expansion in
the final step to obtain said parameter.

Note that, while foldr and foldl are mutually definable on flat structures such as arrays, they are
not at all dual in the world of partial structures such as lazy lists. In particular, just like reverse, the
foldl function is expressible exactly as a higher-order foldr as follows,

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl g z xs = (foldr (\x k w \rightarrow k (g w x)) id xs) z

The converse does not hold: foldr is not expressible in terms of foldl, as the result is always strict
whereas foldr is not.

3.3 Zip and Folds

Foldr-build works beautifully for a huge range of list processing functions but comes to a crashing stop
with zip. Like with reverse, by using a higher-order instance of foldr we can define zip as a fold on
either one of the branches. The other list is passed as an inherited attribute as follows:

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

Unfortunately, using this technique leads us to define two asymmetric versions of zip. 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 became accepted folklore that zip cannot be defined as a fold on both branches at the same
time.

While this is true if we are restricted to ground terms or first-order functions, it is not true if we move
to the the world of hyperfunctions.

4 Coroutining Folds

In order to handle accumulator functions like reverse or even foldl, we had to write the functions as
higher-order folds. That is, the foldr works in first-order function spaces such as [a]->[a], construct-
ing compositions of functions as it traverses its list argument. Interestingly, this is not as inefficient as it
may sound if the final argument for the function is already present—as we saw in the derivation above,
η-expansion flattens the function construction down to a use of an accumulating parameter.
Simple first-order constructions won’t allow us to tame zip, but with the power of universal domain equations behind us (e.g. $D \cong (D \rightarrow D)_\perp$) we have the flexibility to try much more “interesting” functional structures. In this spirit, we will depart from any obligation to satisfy any particular type system at this stage, and feel free to explore definitions that would be rejected by Haskell (so long as we can fix them up afterwards).

We will give fold an extra argument that behaves as a coroutining continuation. This gives a (non-standard) definition of fold as follows:

\[
\text{fold } \text{[]} \text{ } \text{c } \text{ } \text{n } = \lambda k \rightarrow n \\
\text{fold } (x:xs) \text{ } \text{c } \text{ } \text{n } = \lambda k \rightarrow c x (k (\text{fold } xs \text{ } \text{c } \text{ } \text{n}))
\]

Following the observation in Section 3.1 we provide the list argument first. The intuition behind the definition is that the fold function receives an interleaving continuation $k$, applies the “cons-function” $c$ to $x$ and to the result of applying the continuation $k$ to the recursive call of fold. Note that, in the recursive call, the continuation $k$ is not provided. Instead, $k$ will accept the recursive call of fold as its own interleaving continuation, and may subsequently call it with a new continuation. Thus computations and continuations switch roles back and forth repeatedly.

To see this in practice, consider the case where the interleaving continuation is another instance of fold itself.

\[
\text{fold } [1,2,3] \text{ } \text{c } \text{ } \text{n } (\text{fold } [7,8] \text{ } \text{d } \text{ } \text{m}) \\
= c 1 (\text{fold } [7,8] \text{ } \text{d } \text{ } \text{m} (\text{fold } [2,3] \text{ } \text{c } \text{ } \text{n})) \\
= c 1 (d 7 (c 2 (d 8 (c 3 (\text{fold } [] \text{ } \text{d } \text{ } \text{m} (...)))))) \\
= c 1 (d 7 (c 2 (d 8 (c 3 m))))
\]

The folds over the [1,2,3] and [7,8] elements each invoke the other in turn—like coroutines—and thereby produce the interleaving effect.

The fold example showed exactly two interleaving computations coroutining with one another. To generalize the idea to allow one, two, or more coroutining computations, we introduce the following two operations:

\[
\text{self } k = k \text{ self} \\
(f \# g) k = f (g \# k)
\]

These are both recursive definitions. The self function acts as a trivial continuation, and # acts as a composition operation which composes two coroutining continuations into a single coroutining continuation.

Notice, for example, that 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 to operate on the next element, with itself as the next continuation to be invoked.

The composition operator # plays the dual role. 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. So if we wanted to interleave three instances of fold we could do so as follows:

\[
\text{fold } [25] \text{ } \text{c } \text{ } \text{n } (\text{fold } [1,2,3] \text{ } \text{d } \text{ } \text{m} \# \text{fold } [7,8] \text{ } \text{f } \text{ } \text{p}) \\
= c 25 ((\text{fold } [1,2,3] \text{ } \text{d } \text{ } \text{m} \# \text{fold } [7,8] \text{ } \text{f } \text{ } \text{p}) (\text{fold } [] \text{ } \text{c } \text{ } \text{n})) \\
= c 25 (\text{fold } [1,2,3] \text{ } \text{d } \text{ } \text{m} \# \text{fold } [7,8] \text{ } \text{f } \text{ } \text{p} \# \text{fold } [] \text{ } \text{c } \text{ } \text{n}) \\
= c 25 (d 1 ((\text{fold } [7,8] \text{ } \text{f } \text{ } \text{p} \# \text{fold } [] \text{ } \text{c } \text{ } \text{n}) (\text{fold } [2,3] \text{ } \text{d } \text{ } \text{m}))) \\
= c 25 (d 1 (\text{fold } [7,8] \text{ } \text{f } \text{ } \text{p} (\text{fold } [] \text{ } \text{c } \text{ } \text{n} \# \text{fold } [2,3] \text{ } \text{d } \text{ } \text{m})))
\]
Coroutining Folds with Hyperfunctions

\[
= c 25 \left( d 1 \left( f 7 \left( \text{fold } [] \ c \ n \ # \ \text{fold } [2,3] \ d \ m \right) \ (\text{fold } [7,8] \ f \ p) \right) \right)
\]

Naturally \texttt{self} and \# behave well together, as expressed in the following theorem:

**Theorem 2** \# is associative, and \texttt{self} is a left and right identity for \#.

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

\[
(fold \ [25] \ c \ n \ # \ fold \ [1,2,3] \ d \ m \ # \ fold \ [7,8] \ f \ p) \ \text{self}
\]

The power of this version of \texttt{fold} is apparent when we see that \texttt{zip} is definable in terms of independent \texttt{fold}s on its two branches, thus:

\[
\text{zip } xs \ ys = (\text{fold } xs \ \text{first } [] \ # \ \text{fold } ys \ \text{second } \text{Nothing}) \ \text{self}
\]

where

\[
\begin{align*}
\text{first } x \ \text{Nothing} &= [] \\
\text{first } x \ (\text{Just } (y,xys)) &= (x,y) : xys \\
\text{second } y \ xys &= \text{Just } (y,xys)
\end{align*}
\]

Note that while \texttt{first} is strict—the presence or absence of a \(y\) element is needed to know whether to produce a new output element—the second function is not, thus \texttt{zip} has its usual non-strict behavior.

### 4.1 Universal Domains

What kind of functions are \texttt{self} and \#? The definition of \texttt{self} requires a solution to the function equation \(t_1 = (t_1 \to t_0) \to t_0\), and a similar equation arises from within the definition of \#. Moreover, the definition of \texttt{fold} also requires an infinite type. In this case, the general form of the type equation that needs to be solved is \(H \ a \ b = H \ b \ a \to b\), which when expanded gives the infinite type

\[
H \ a \ b = (((((\ldots \to a) \to b) \to a) \to b)
\]

Clearly no set-theoretic solution exists for these, but they live quite happily within any domain that contains an image of its own function space, such as \(D\) above. It is instructive to consider to CPO-theoretic approximations to \(H\). The first is

\[
\begin{align*}
H0 \ a \ b &= 1 \\
H1 \ a \ b &= (1 \to a) \to b \\
&= a \to b \\
H2 \ a \ b &= (H1 \ a \ b \to a) \to b \\
&= (a \to b) \to a \to b
\end{align*}
\]

Interestingly, all the bs are in positive (covariant) positions and all the as are in negative (contravariant) positions. In effect, the a’s act as arguments, and the b’s act as results. That is, the type \(H \ a \ b\) is a kind of function from \(a\) to \(b\), or even a stack of functions from \(a\) to \(b\). We use the term \textit{hyperfunction} to express this.

### 5 Typing

To code hyperfunctions in Haskell we introduce \(H\) as a newtype, and define appropriate access functions.
newtype H a b = Fn {invoke :: H b a -> b}

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

self :: H a a
self = lift id

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

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

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

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

The use of the explicit constructor Fn and deconstructor invoke obscure some of the definitions a little, though they provides us with informative types in return. In particular, 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. 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.

We defined lift using a new operator (<<), which acts rather like a cons operator by taking a function element f and adding to the “stack” of functions q (and invoking an intervening coroutining continuation in-between, of course). Without types we could define (<<) by (f<<q) k = f (k q).

These definitions make it easy to define a hyperfunction form of the fold operation:

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 (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)

This ability to define either in terms of the other shows that fold and foldr are equivalent to one another.

6 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 (:|) [])

Our conjecture is that under appropriate circumstances, the fusion law holds:

fold . build = id

Initially we hoped that the “appropriate circumstances” would simply be the parametric nature of g’s type. Now it appears we’ll have to be slightly more clever. For example, restricting g to be constructed by repeated applications of << and base turns out to be sufficient, and this could be achieved by making the H type abstract. In this paper, the fusion law is left as a conjecture requiring further characterization, and we will focus instead on better characterizing the hyperfuntions themselves. Just before we do, let us see the fusion law in practice—now with zip. Consider the example of fusing

\[ \text{sum} \ (\text{zipW} \ (+) \ (\text{map} \ \text{sqr} \ \text{xs}) \ (\text{map} \ \text{inc} \ \text{ys})) \]

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

\[ \text{zipW} \ f \ \text{xs} \ \text{ys} = \text{build} \ (\text{zipW’} \ f \ \text{xs} \ \text{ys}) \]
\[ \text{zipW’} \ f \ \text{xs} \ \text{ys} \ c \ n = \text{fold xs} \ \text{first} \ n \ # \ \text{fold ys} \ \text{second} \ \text{Nothing} \]

where

\[
\begin{align*}
\text{first} \ x \ \text{Nothing} & = n \\
\text{first} \ x \ (\text{Just} \ (y,xys)) & = c \ (f \ x \ y) \ xys \\
\text{second} \ y \ xys & = \text{Just} \ (y,xys)
\end{align*}
\]

In this case, the zipper-function f is applied within the right-hand side of first whenever a pair of entries from the two lists are present. The other list processing functions are defined as follows:

\[ \text{map} \ f \ \text{xs} =\text{build} \ (\\lambda \ c \ n \rightarrow \text{fold xs} \ (c \ . \ f) \ n) \]
\[ \text{sum} \ \text{xs} = \text{run} \ (\text{fold} \ \text{xs} \ (+) \ 0) \]

The fusion proceeds through beta reduction, symbolic composition, and application of fold-build.

\[ \text{sum} \ (\text{zipW} \ (+) \ (\text{map} \ \text{sqr} \ \text{xs}) \ (\text{map} \ \text{inc} \ \text{ys})) \]
\[ = \text{run} \ (\text{fold} \ (\text{zipW} \ (+) \ (\text{map} \ \text{sqr} \ \text{xs}) \ (\text{map} \ \text{inc} \ \text{ys})) \ (+) \ 0) \]
\[ = \text{run} \ (\text{fold} \ (\text{map} \ \text{sqr} \ \text{xs}) \ \text{first} \ 0 \ # \ \text{fold} \ (\text{map} \ \text{inc} \ \text{ys}) \ \text{second} \ \text{Nothing}) \]

where

\[
\begin{align*}
\text{first} \ x \ \text{Nothing} & = 0 \\
\text{first} \ x \ (\text{Just} \ (y,xys)) & = (x * y) + xys \\
\text{second} \ y \ xys & = \text{Just} \ (y,xys)
\end{align*}
\]

\[ = \text{run} \ (\text{fold} \ \text{xs} \ (\text{first’} \ . \ \text{sqr}) \ 0 \ # \ \text{fold} \ \text{ys} \ (\text{second’} \ . \ \text{inc}) \ \text{Nothing}) \]

where

\[
\begin{align*}
\text{first’} \ x \ \text{Nothing} & = 0 \\
\text{first’} \ x \ (\text{Just} \ (y,xys)) & = (\text{sqr} \ x \ * \ y) + xys \\
\text{second’} \ y \ xys & = \text{Just} \ (\text{inc} \ y, xys)
\end{align*}
\]

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.
A recap on the context of this approach is probably useful at this point. Simple folds at ground
types (that is, that build non-function structures) have been well studied. These uses of fold construct
synthesized attributes only. Once we allow folds to build functions as their results, we gain significant
extra power. The function arguments allow us to model inherited attributes that allow reverse and
even foldl to be seen as instances of foldr. Once we allow folds to build hyperfunctions, we enable
coroutining between distinct fold computations. This appears to go beyond the usual language of attribute
grammars with inherited and synthesized attributes as it now permits attributes to flow between the nodes
of different trees that are at the same level, and a (nearly) symmetric definition of zip becomes possible.

7 Hyperfunctions Axiomatically

The operators we defined on hyperfunctions encourage us to move away from the explicit model-theoretic
view of H, and consider instead an axiomatic approach. This will allow us to consider other models which
may be more efficient implementations in certain cases.

We continue to use the notation H a b, but now for an abstract type of hyperfunctions. We regard
H a b as describing the set (or more likely, the CPO) of arrows between objects a and b in an new hyper-
function category which shares the same objects as the original. We require the following operations:

(#) :: H b c -> H a b -> H a c
lift :: (a->b) -> H a b
run :: H a a -> a
(<<) :: (a->b) -> H a b -> H a b

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
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))

where self :: H a a is defined by self = lift id, and fix and composition have their usual
meanings. These axioms make hyperfunctions into a category. The lift function is a functor from
the base category into the hyperfunction category, and (lift f) lets us see an underlying function f as
a hyperfunction. The # operation extends the composition, and run extends the fix point operator. The
following definition of mapH

mapH :: (a' -> a) -> (b -> b') -> H a b -> H a' b'
mapH r s f = lift s # f # lift r

demonstrates that H is contravariantly functorial in its first argument, and covariantly in its second.

Using these definitions, we can now define some of the other hyperfunction operations:

invoke :: H a b -> H b a -> b
invoke f g = run (f # g)

base :: b -> H a b
base k = lift (const k)
where \( \text{const} \ k = \lambda x \rightarrow k \) is the constant function. It now follows that \( \text{run} \ f = \text{invoke} \ f \ \text{self} \), giving a definition of run whenever invoke is more naturally defined as a primitive (as in the earlier definition).

Without \( \ll \) and its axioms, the system has a trivial model in which \( H \ a \ b = a \rightarrow b \), with \# being ordinary function composition, \( \text{lift} \ f = f \), and so on. With \( \ll \), the trivial model is no longer possible.

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

**Theorem 3** The functor \( \text{lift} \) is faithful (i.e. if \( \text{lift} \ f = \text{lift} \ g \) then \( f = g \)).

The theorem follows by an easy calculational proof. We define:

\[
\text{project} :: H \ a \ b \rightarrow (a \rightarrow b)
\]

\[
\text{project} \ q \ x = \text{invoke} \ q \ (\text{base} \ x)
\]

It suffices to show that \( \text{project} \) is a left-inverse of \( \text{lift} \), i.e. that \( \text{project} \ (\text{lift} \ f) = f \). Indeed:

\[
\text{project} \ (\text{lift} \ f) \ x
\]
\[
= \text{invoke} \ (\text{lift} \ f) \ (\text{base} \ x)
\]
\[
= \text{run} \ ((\text{lift} \ f) \# \text{base} \ x)
\]
\[
= \text{run} \ ((f \ll \text{lift} \ f) \# \text{base} \ x)
\]
\[
= f \ (\text{run} \ (\text{base} \ x \# \text{lift} \ f))
\]
\[
= f \ (\text{run} \ (\text{lift} \ (\text{const} \ x) \# \text{lift} \ f))
\]
\[
= f \ (\text{run} \ ((\text{const} \ x \ll \text{base} \ x) \# \text{lift} \ f))
\]
\[
= f \ (\text{const} \ x \ (\text{run} \ (\text{lift} \ f \# \text{base} \ x)))
\]
\[
= f \ x
\]

as required.

8 A Stream Model for \( H \)

Now that we view \( H \) as an abstract type, we are free to investigate alternative models for it. We have two other models which provide useful insights into the core functionality required for \text{zip} fusion.

The elements of \( H \) we have been using behave like a stream of functions: some initial portion of work is performed, and then the remaining work is delayed and given to the continuation to be invoked at some point in the future (if at all). 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 this family of hyperfunctions explicitly as an infinite stream. We use the name \( L \) for this model.

\[
data \ L \ a \ b = (a \rightarrow b) :\ll: \ L \ a \ b
\]

\[
\text{invoke} :: \ L \ a \ b \rightarrow \ L \ b \ a \rightarrow b
\]

\[
\text{invoke} \ fs \ gs = \text{run} \ (fs \# gs)
\]

\[
(\#) :: \ L \ b \ c \rightarrow \ L \ a \ b \rightarrow \ L \ a \ c
\]

\[
(f :\ll: \ gs) \# (g :\ll: \ gs) = (f \ . \ g) :\ll: (fs \# gs)
\]
\[ \text{self} :: L \ a \ a \\
\text{self} = \text{lift \ id} \]

\[ \text{lift} :: (a\rightarrow b) \rightarrow L \ a \ b \\
\text{lift \ } f = f ::<<: \text{lift \ } f \]

\[ \text{base} :: a \rightarrow L \ b \ a \\
\text{base \ } x = \text{lift \ } (\text{const \ } x) \]

\[ (\ll) :: (a\rightarrow b) \rightarrow L \ a \ b \rightarrow L \ a \ b \\
(\ll) = (::<<:) \]

\[ \text{run} :: L \ a \ a \rightarrow a \\
\text{run \ } (f ::<<: \text{fs}) = f \ (\text{run \ } \text{fs}) \]

One interesting aspect of this model is that \text{run} is more naturally primitive than \text{invoke}, whereas in the original function-space model \( H \) the opposite was the case. On the other hand, the identity and associativity laws between \# and \text{self} become very easy to prove just by fixed point induction and properties of composition. In contrast, the corresponding theorems about the \( H \) model turned out to be rather challenging, to say the least [KLP01].

The stream of functions acts like a fixpoint waiting to happen. Two things could occur: either the functions are interspersed with another stream of functions, or all the functions are composed together by \text{run}. In this way, \text{run} ties the recursive knot, and removes opportunities for further coroutining.

The \text{fold} function is defined exactly as before in terms of \ll and base. Its behavior is given as follows:

\[ \text{fold \ } [x_1,x_2,x_3] \ c \ n \\
= c \ x_1 ::<<: c \ x_2 ::<<: c \ x_3 ::<<: \text{const \ } n ::<<: \ldots \]

where the \ldots indicates an infinite stream of const \( n \). Thus \text{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 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 \text{zip} turns out to be defined in terms of \#, which is defined just like \text{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 the original type \( H \) of hyperfunctions, the \( L \) model can be also used for fold-build fusion, and the stream structures are optimized away. Any that exist after the fusion phase may (in principle) be removable by inlining the definition of \text{run}. In other words, if the compiler is able to clean up sufficiently, the stream structure simply will not exist at run-time—it’s purpose is compile-time only.

Though we won’t prove it here, the \( L \) model is the simplest possible model of hyperfunctions. Formally, in the category of hyper function models, the \( L \) model is an initial object. The \( L \) model seems to capture something essential about using hyperfunctions to define \text{zip}. It expresses the “linear” behavior of \text{fold} as it traverses its input lists.

9 \hspace{1em} \text{A State-Machine Model for Hyperfunctions}

We need to go a little further than \( L \) to find a good model for doing \text{zip}-fusion in practice. 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. In one sense it was easy. By restricting to fusion along a single branch of zip-like functions, we always ended up with a single ultimate recursive origin for the computation. All the foldr-build rules had to do was place in the subsequent processing of list elements into the appropriate places within this (arbitrarily recursive) structure, and we were done.

In contrast, multi-branch fusion may have many sources each acting as a partial origin of the computation, so we may need to combine multiple recursive generators. This is very hard in general, so to make the problem tractable we focus on recursive generators that are state machines, also known as tail calls or anamorphisms. This leads us to yet another model for hyperfunctions where we represent the state of a function as an anamorphism. The type of this state element can be hidden by using a rank-2 universally quantified type.

```haskell
data A a b where
  Hide :: (u -> Either b (a -> b,u)) -> u -> A a b

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

(#) :: 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')

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)

(<<) :: (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
```

The form of the type declaration uses the GADT syntax. The declaration gives `Hide` a type in which `u` is universally quantified. By the usual interchange laws between universal and existential quantifiers, this is equivalent to

```haskell
Hide :: (exists u . (u -> Either b (a -> b,u)), u) -> A a b
```
In other words, the objects of the A model are state machines whose inner state is completely hidden from the outside world, except inasmuch as they produce the next portion of a coroutining function on demand.

Going back to zip-fusion, we can put these definitions to work. We define a couple of typical generators.

\[
down z = \text{build} \ (\down' \ z)
\]

\[
down' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{b} \rightarrow \text{c}) \rightarrow \text{c} \rightarrow A \ \text{b} \ \text{c}
\]

\[
down' \ w \ c \ n = \text{Hide} \ (\lambda z \rightarrow \text{if } z\leq \text{0} \ \text{then } \text{Left } n \\
\text{else } \text{Right } (c \ z, z-1)) \ \\
w
\]

\[
upto \ i \ j = \text{build} \ (\upto' \ i \ j)
\]

\[
upto' :: \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int} \rightarrow \text{b} \rightarrow \text{c}) \rightarrow \text{c} \rightarrow A \ \text{b} \ \text{c}
\]

\[
upto' \ a \ b \ c \ n = \text{Hide} \ (\lambda (i,j) \rightarrow \text{if } i>j \ \text{then } \text{Left } n \\
\text{else } \text{Right } (c \ i, (i+1,j))) \\
(a, b)
\]

As an example, we fuse the expression \(\text{sum} \ (\text{zipW} \ (*) \ (\text{upto} \ 2 \ 10) \ (\text{down} \ 6))\). The derivation proceeds as follows.

\[
\text{sum} \ (\text{zipW} \ (*) \ (\text{upto} \ 2 \ 10) \ (\text{down} \ 6))
= \text{run} \ (\text{fold} \ (\text{zipW} \ (*) \ (\text{upto} \ 2 \ 10) \ (\text{down} \ 6)) \ (+) \ 0)
= \text{run} \ (\text{zipW'} \ (*) \ (\text{upto} \ 2 \ 10) \ (\text{down} \ 6) \ (+) \ 0)
= \text{run} \ (\text{fold} \ (\text{upto} \ 2 \ 10) \ c \ 0 \ # \ \text{fold} \ (\text{down} \ 6) \ d \ \text{Nothing})
\]

where

\[
c x \ \text{Nothing} = 0 \\
c x \ (\text{Just } (y,xys)) = (x * y) + xys \\
d y xys = \text{Just } (y,xys)
\]

\[
= \text{run} \ (\text{upto'} \ 2 \ 10 \ c \ 0 \ # \ \text{down'} \ 6 \ d \ \text{Nothing})
\]

where

\[
c x \ \text{Nothing} = 0 \\
c x \ (\text{Just } (y,xys)) = (x * y) + xys \\
d y xys = \text{Just } (y,xys)
\]

\[
= \text{run} \ (\text{Hide} \ (\lambda (i,j) \rightarrow \text{if } i>j \ \text{then } \text{stop } 0 \ \text{else } (c \ i, (i+1,j))) \\
(2,10)
#
\]

\[
\text{Hide} \ (\lambda z \rightarrow \text{if } z\leq \text{0} \ \text{then } \text{stop Nothing else } (d \ z, z-1)) \ \\
6
\]

where

\[
c x \ \text{Nothing} = 0 \\
c x \ (\text{Just } (y,xys)) = (x * y) + xys \\
d y xys = \text{Just } (y,xys)
\]

\[
= \text{run} \ (\text{Hide} \ (\lambda ((i,j),z) \rightarrow \text{case if } i>j \ \text{then } \text{Left } 0 \ \text{else } \text{Right } (c \ i, (i+1,j)) \ \text{of} \\
\text{Left } n \rightarrow \text{Left } n \\
\text{Right } (f,y) \rightarrow
\]

\[
\text{Right } (f,y) \rightarrow
\]

\[
\text{Right } (f,y) \rightarrow
\]

\[
\text{Right } (f,y) \rightarrow
\]

\[
\text{Right } (f,y) \rightarrow
\]

\[
\text{Right } (f,y) \rightarrow
\]
The first few steps are just the fold-build from before, and the remainders normal program simplification. The result is quite impressive. All intermediate lists have been removed, and the multiple generators merged into a single generator.

10 Conclusion

The original motivation for hyperfunctions was to broaden the power of the fold-build fusion technique to be able to handle multiple input lists. In this it succeed, and we have demonstrated that many occurrences of \texttt{zip} can be eliminated using the fold-build technique, leading to the fusion of multiple list generation routines. The implementation is simple and it works well, but as yet it is not clear how well it would work in large examples. Of course, as noted in the introduction, the whole approach of foldr-build has been eclipsed by the stream fusion techniques [CLS07]. In stream fusion, lists are represented as (non-recursive) state-machine stream processors, and it turns out to be quite feasible to fuse these state-machines together, including for the case of \texttt{zip}.

However, coroutining folds may turn out to have merit in their own right. In particular, the real insights of this papers are twofold:

First, we were forced to realize that the \texttt{fold} function is even more powerful than we had previously thought. In particular, it came as a palpable shock that \texttt{fold} was able to express interleaving compu-
tations. 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 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 [Hug00]. Again, whether they will turn out to be useful in other applications remains to be seen.

11 Acknowledgements

An early version of these ideas was presented to the IFIP working group on functional programming (WG2.8, 1999) where we received helpful feedback from many people, but especially Simon Peyton Jones, Richard Bird, Ralph Hinze, Erik Meijer.

References

[GLPJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A Shortcut to Deforestation. In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 223 – 232, 1993.

[Hug00] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming* (Elsevier) 37 (1-3), May 2000. doi:10.1016/S0167-6423(99)00023-4

[J03] P. Johann. Short cut fusion is correct. *Journal of Functional Programming*, 13, pp 797-814. C.U.P. 2003. doi:10.1017/S0956796802004409

[JV03] P. Johann, J. Voigtlaender. Free theorems in the presence of seq. *Proceedings, Conference on Principles of Programming Languages*. POPL 2004. doi:10.1145/964001.964010

[KLP01] S. Krstic, J. Launchbury, and D. Pavlovic. Hyperfunctions. In *Proceeding of Fixed Points in Computer Science*, FICS 2001

[LKPS00] J. Launchbury, S. Krstic, and T. E. Sauerwein. Zip Fusion with Hyperfunctions. *Unpublished*, 2000.

[LS95] J. Launchbury and T. Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *Proceedings, Conference on Functional Languages and Computer Architecture*, pp. 314 – 323, 1995. doi:10.1145/224164.224223

[Sch86] D.A. Schmidt. Denotational Semantics, Wm. C. Brown, 1986.

[Wad89] P. Wadler. Theorems for Free! In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 347 – 359, 1989. doi:10.1145/99370.99404

[Wad90] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science* 73 (1990), pp. 231 – 248. doi:10.1016/0304-3975(90)90147-A

[CLS07] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion. From Lists to Streams to Nothing at All. In *Proceedings, International Conference on Functional Programming*, 2007. doi:10.1145/1291151.1291199