Initial Algebra Semantics is Enough!

Patricia Johann* and Neil Ghani**

Abstract. Initial algebra semantics is a cornerstone of the theory of modern functional programming languages. For each inductive data type, it provides a `fold` combinator encapsulating structured recursion over data of that type, a Church encoding, a `build` combinator which constructs data of that type, and a `fold`/`build` rule which optimises modular programs by eliminating intermediate data of that type. It has long been thought that initial algebra semantics is not expressive enough to provide a similar foundation for programming with nested types. Specifically, the `fold`s have been considered too weak to capture commonly occurring patterns of recursion, and no Church encodings, `build` combinators, or `fold`/`build` rules have been given for nested types. This paper overturns this conventional wisdom by solving all of these problems.

1 Introduction

Initial algebra semantics is one of the cornerstones of the theory of modern functional programming languages. It provides support for `fold` combinators encapsulating structured recursion over data structures, thereby making it possible to write, reason about, and transform programs in principled ways. Recently, (13) extended the usual initial algebra semantics for inductive types to support not only standard `fold` combinators, but Church encodings and `build` combinators for them as well. In addition to being theoretically useful in ensuring that `build` is seen as a fundamental part of the basic infrastructure for programming with inductive types, this development has practical merit: the `fold` and `build` combinators can be used to define `fold`/`build` rules which optimise modular programs by eliminating intermediate inductive data structures. When applied to lists, this optimisation is known as *short cut fusion*.

Nested data types have become increasingly popular in recent years (1; 3; 5; 6; 7; 14; 15; 16; 17; 20). They have been used to implement a number of advanced data types in languages, such as Haskell, which support higher-kinded types. Among these data types are those with constraints, such as perfect trees (16); types with variable binding, such as untyped λ-terms (2; 5; 8); cyclic data structures (11); and certain dependent types (21). The expressiveness of nested types lies in their generalisation of the traditional treatment of types as free-standing individual entities to entire families of types. To illustrate this point, consider the type of lists of elements of type `a`. This type can be realised in Haskell via the declaration `data List a = Nil | Cons a (List a)`. As this declaration makes clear, the type `List a` can be defined independently of any type `List b` for `b` distinct from `a`. Moreover, since each type `List a` is, in isolation, an inductive type, the type constructor `List` is seen to define a *family of inductive types*. Compare the declaration for `List a` with the declaration

* Rutgers University, Camden, NJ, USA. email: `pjohann@crab.rutgers.edu`
** University of Nottingham, Nottingham, UK. email: `nxg@cs.nott.ac.uk`

```
data Lam a = Var a | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
```

defining the type `Lam a` of untyped λ-terms over variables of type `a` up to α-equivalence. By contrast with `List a`, the type `Lam a` cannot be defined in terms of only those elements of `Lam a` that have already been constructed. Indeed, elements of the type `Lam (Maybe a)` are needed to build elements of `Lam a` so that, in effect, the entire family of types determined by `Lam` has to be constructed simultaneously. Thus, rather than defining a family of inductive types as `List` does, `Lam` defines an *inductive family of types*.

Given the increased expressivity of nested types over inductive types, and the ensuing growth in their use, it is natural to ask whether initial algebra semantics can give a principled foundation for structured programming with nested types. Until now this has not been considered possible. In particular, `fold` combinators derived from initial algebra semantics for nested types have not been considered expressive enough to capture certain commonly occurring patterns of structured recursion over data of those types. This has led to a theory of *generalised* `fold`*s* for nested types (1; 3; 6). Moreover, no Church encodings, `build` combinators, or `fold`/`build` fusion rules have been proposed or defined for nested types.

This paper overturns this conventional wisdom and provides the ideal result, namely that *initial algebra semantics is enough to provide a principled foundation for programming with nested types.* Our major contributions are as follows:

- We define a generalised `fold` combinator `gfold` for *every* nested type and show it to be uniformly interdefinable with the corresponding `hfold` combinator derived from initial algebra semantics. Our `gfold` combinators coincide with the generalised `fold`s in the literature whenever the latter are defined. The `hfold` combinators provided by initial algebra semantics thus capture *exactly the same kinds of recursion* as the generalised `fold`s in the literature.
- We give the first-ever Church encodings for nested types. In addition to being interesting in their own right, these encodings are the key to defining the first-ever `build` combinators for nested types. Coupling each `hbuild` combinator with its corresponding `hfold` combinator in turn gives the first-ever `hfold`/`hbuild` rules for nested types, and thus extends short cut fusion to these types. A similar story holds for the `gfold` and `gbuild` combinators.

We make several other important contributions. First, we execute the above program in a generic style by providing *a single* generic `hfold` combinator, *a single* generic `hbuild` operator, and *a single* generic `hfold`/`hbuild` rule, each of which can be specialised to any particular nested type of interest — and similarly for the generalised combinators. Secondly, while the theory of nested types has previously been developed only for limited classes of nested types arising from certain syntactically defined classes of rank-2 functors, our development handles *all* rank-2 functors. Finally, we give a complete implementation of our ideas in Haskell, available at `http://www.cs.nott.ac.uk/~nxg`. This demonstrates the practical applicability of our ideas, makes them more accessible, and provides a partial guarantee of their correctness via the Haskell type-checker. This paper can therefore be read both as abstract mathematics, and as providing the basis for experiments and practical applications. Past work on nested types did not come with full implementations, in part because essential features such as explicit and nested `forall`-types have only recently been added to Haskell.

Our result that initial algebra semantics is expressive enough to provide a foundation for programming with nested types allows us to capitalise on the increased expressiveness of nested types over inductive types without requiring the development of any fundamentally new theory. Moreover, this foundation is simple, clean, and accessible to anyone with an understanding of the basics of initial algebra semantics. This is important, since it guarantees that our results are immediately usable by functional programmers. Further, by closing the gap between initial algebra semantics and Haskell's data types, this paper clearly contributes to the foundations of functional programming. This paper also serves as a compelling demonstration of the practical applicability of left and right Kan extensions — which are the main technical tools used to define our `gfold`s and prove them interdefinable with the `hfold`s — and thus has the potential to render them mainstays of functional programming.

The paper is structured as follows. Section 2 recalls the initial algebra semantics of inductive types. Section 3 recalls the derivation of `fold` combinators from initial algebra semantics for nested types, and derives the first Church encodings, `build` combinators, and `fold`/`build` rules for them. Section 4 defines our `gfold` combinators for nested types and shows that they are interdefinable with their corresponding `hfold` combinators. It also derives our `gbuild` combinators and `gfold`/`gbuild` rules for nested types. Section 5 mentions the coalgebraic duals of our combinators and draws some conclusions.

2 Initial Algebra Semantics for Inductive Types

Inductive data types 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 is well known (12; 13; 23), every inductive type has an associated `fold` and `build` combinator which can be implemented generically in Haskell as

```
newtype M f = Inn {unInn :: f (M f)}

ffold :: Functor f => (f a -> a) -> M f -> a
ffold h (Inn k) = h (fmap (ffold h) k)

fbuild :: Functor f => (forall b. (f b -> b) -> b) -> M f
fbuild g = g Inn
```

These `fbuild` and `ffold` combinators can be used to construct and eliminate inductive data structures of type `M f` from computations. Indeed, if `f` is any functor, `h` is any function of any type `f a -> a`, and `g` is any function of closed type `forall b. (f b -> b) -> b`, we have the `fold`/`build` rule:

```
ffold h (fbuild g) = g h                                    (1)
```

When specialised to lists, this gives the familiar combinators

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

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

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`. On the other hand, `build` 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]`. List transformers can be written in terms of both `foldr` and `build`. For example, the standard `map` function for lists can be implemented as

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

The function `build` is not just of theoretical interest as the producer counterpart to the list consumer `foldr`. In fact, `build` is an important ingredient in *short cut fusion* (9; 10), a widely-used program optimisation which capitalises on the uniform production and consumption of lists to improve the performance of list-manipulating programs. For example, if `sqr x = x * x`, then the specialisation of (1) to lists — i.e., the rule `fold c n (build g) = g c n` — can transform the modular function `sum (map sqr xs) :: [Int] -> Int` which produces an intermediate list into an optimised form which produces no such lists:

```
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
```

If we are to generalise the treatment of inductive types given above to more advanced data types, we must ask ourselves why `fold` and `build` combinators exist for inductive types and why the associated `fold`/`build` rules are correct. One elegant answer is provided by *initial algebra semantics*. Within the paradigm of initial algebra semantics, every data type is the carrier of the initial algebra μF of a functor $F : \mathcal{C} \to \mathcal{C}$. If $\mathcal{C}$ has both an initial object and ω-colimits, and F preserves ω-colimits, then F does indeed have an initial algebra. Lambek's lemma ensures that the structure map *in* of an initial algebra is an isomorphism, and thus that the carrier of the initial algebra of a functor is a fixed point of that functor. The interpretation of a given data type as an initial algebra of a functor F ensures that there is a unique F-algebra homomorphism from this initial F-algebra to any other F-algebra. If (A, h) is an F-algebra, then $\texttt{fold}\, h : \mu F \to A$ is the map underlying this homomorphism and makes the following diagram commute:

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

From this diagram, we see that the type of `fold` is $(FA \to A) \to \mu F \to A$ and that $\texttt{fold}\, h$ satisfies $\texttt{fold}\, h\ (in\ t) = h\ (F\ (\texttt{fold}\, h)\ t)$. This justifies the definition of the `ffold` combinator given above. Also, the uniqueness of the mediating map ensures that, for every algebra h, the map $\texttt{fold}\, h$ is defined uniquely. This provides the basis for the correctness of `fold` fusion for inductive types, which states that if h and h' are F-algebras and ψ is an F-algebra homomorphism from h to h', then $\psi\, .\, \texttt{fold}\, h = \texttt{fold}\, h'$. But note that `fold` fusion (3; 5; 6; 7; 20), is completely different from, and inherently simpler than, the `fold`/`build` fusion which is central in this paper, and which we discuss next.

Although `fold` combinators for inductive types can be derived entirely from, and understood entirely in terms of, initial algebra semantics, regrettably the standard initial algebra semantics does not provide a similar principled derivation of the `build` combinators or the correctness of the `fold`/`build` rules. This situation was rectified in (13), which considered the initial F-algebra for a functor F to be not only the initial object of the category of F-algebras, but also the limit of the forgetful functor from the category of F-algebras to the underlying category $\mathcal{C}$ as well. When F has an initial algebra, no extra structure is required of either F or $\mathcal{C}$ for this limit to exist. This characterisation of initial algebras as both limits *and* colimits is what we call the *extended initial algebra semantics*. As shown in (13), an initial F-algebra has a different universal property as a limit from the one it inherits as a colimit. This alternate universal property ensures:

- The projection from the limit (the initial F-algebra) to the carrier of each algebra defines the `fold` combinator with type $(Fx \to x) \to \mu F \to x$.
- The mediating morphism maps a cone with arbitrary vertex c to a map from c to μF. Since a cone with vertex c has type $\forall x.(Fx \to x) \to c \to x$, the mediating morphism defines the `build` combinator, which will thus have type $(\forall x.\,(Fx \to x) \to c \to x) \to c \to \mu F$.
- The correctness of the `fold`/`build` fusion rule $\texttt{fold}\, h\ .\ \texttt{build}\, g\ =\ g\, h$ 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 a specific algebra.

The extended initial algebra semantics thus shows that, given a parametric interpretation of the quantifier `forall`, there is an isomorphism between the type `c -> M f` and the "generalised Church encoding" `forall x. (f x -> x) ->`

`c -> x`. The term "generalised" reflects the presence of the parameter `c`, which is absent in other Church encodings (23), but is essential to the derivation of `build` combinators for nested types. Choosing `c` to be the unit type gives the usual isomorphism between an inductive type and its usual Church encoding.

3 Initial Algebra Semantics for Nested Types

Although many types of interest can be expressed as inductive types, these types are not expressive enough to capture all data structures of interest. Such structures can, however, often be expressed in terms of *nested types*.

Example 1 *The type of* perfect trees *over type* `a` *is given by*

```
data PTree a = PLeaf a | PNode (PTree (a,a))
```

The recursive constructor `PNode` stores not pairs of trees, but rather trees with data of pair types. Thus, `PTree a` is a nested type for each `a`. Perfect trees are easily seen to be in one-to-one correspondence with lists whose length is a power of two, and hence illustrate how nested types can be used to capture structural constraints on data types. Another example of nested types is given by

Example 2 *The type of (α-equivalence classes of)* untyped λ-terms *over variables of type* `a` *is given by*

```
data Lam a = Var a | App (Lam a) (Lam a) | Abs (Lam (Maybe a))
```

Elements of type `Lam a` include `Abs (Var Nothing)` and `Abs (Var (Just x))`, which represent $\lambda x.x$ and $\lambda y.x$, respectively. We observed above that each nested type constructor defines an inductive family of types. It is thus natural to model nested types as least fixed points of functors on the category of endofunctors on $\mathcal{C}$, written $[\mathcal{C}, \mathcal{C}]$. In this category, objects are functors and morphisms are natural transformations. We call such functors *higher-order functors*, and denote the fixed point of a higher-order functor `f` by `Mu f`. Our implementation cannot use the constructor `M` introduced above because Haskell lacks polymorphic kinding.

```
class HFunctor f where
  ffmap :: Functor g => (a -> b) -> f g a -> f g b
  hfmap :: Nat g h -> Nat (f g) (f h)

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

A higher-order functor thus maps functors to functors via the `ffmap` operation and natural transformations to natural transformations via the `hfmap` operation. While not explicit in the class definition above, the programmer is expected to verify that if `g` is a functor, then `f g` satisfies the functor laws. The type of natural transformations can be given in Haskell by `type Nat g h = forall a. g a -> h a`, since a parametric interpretation of the `forall` quantifier ensures that the naturality squares commute. Putting this all together, we have

Example 3 *The nested types of perfect trees and untyped λ-terms from Examples 1 and 2 arise as fixed points of the higher-order functors*

```
data HPTree f a = HPLeaf a | HPNode (f (a,a))

data HLam f a = HVar a | HApp (f a) (f a) | HAbs (f (Maybe a))
```

respectively. Indeed, the types `PTree a` *and* `Lam a` *are isomorphic to the types* `Mu HPTree a` *and* `Mu HLam a`*.*

Pleasingly, `fold` combinators for nested types can be derived by simply instantiating the ideas from Section 2 in a category of endofunctors. Of course, now the structure map of an algebra is a natural transformation, and the result of a `fold` is a natural transformation from a nested type to the carrier of the algebra. Using the synonym `type Alg f g = Nat (f g) g` for such algebras, we have

```
hfold :: HFunctor f => Alg f g -> Nat (Mu f) g
hfold m (In u) = m (hfmap (hfold m) u)
```

Example 4 *The* `fold` *combinator for perfect trees is*[1]

```
foldPTree :: (forall a. a -> f a) ->
                     (forall a. f (a,a) -> f a) -> PTree a -> f a
foldPTree f g (PLeaf x)  = f x
foldPTree f g (PNode xs) = g (foldPTree f g xs)
```

The uniqueness of `hfold`, guaranteed by its derivation from initial algebra semantics, provides the basis for the correctness of `fold` fusion for nested types (7). As mentioned above, `fold` fusion is not the same as `fold`/`build` fusion. In particular, the latter has not previously been considered for nested types.

Recall from Section 2 that Church encodings and `build` combinators for inductive types can be derived from the characterisation of the initial F-algebra as the limit of the forgetful functor from the category of F-algebras to the underlying category, and that this gives an isomorphism between types of the form `c -> M f` and generalised Church encodings `forall x. (f x -> x) -> c -> x`. Since this isomorphism holds for *all* functors, including higher-order ones, We should be able to instantiate it for higher-order functors to derive Church encodings and `build` combinators for nested types. And indeed we can. This gives the following Haskell code:

```
hbuild :: (HFunctor f, Functor c) =>
              (forall x. Alg f x -> Nat c x) -> Nat c (Mu f)
hbuild g = g In
```

It is worth noticing that each `hbuild` combinator follows the definitional format of the `build` combinators for inductive types: it applies its argument to the structure map `In` of the initial algebra of the higher-order functor `f` with which it is associated. For our running example of perfect trees, we have the following:

[1] Here we have used standard type isomorphisms to "unbundle" the input type `Alg HPTree f` for `foldPTree`. Such unbundling will be done without comment henceforth.

Example 5 *The* `hbuild` *combinator for perfect trees is given concretely by*

```
buildPTree :: (forall x. (forall a. a -> x a) ->
                         (forall a. x (a,a) -> x a) ->
                         (forall a. c a -> x a)) -> Nat c PTree
buildPTree g = g PLeaf PNode
```

The extended initial algebra semantics ensures that `hbuild` and (an argument-permuted version of) `hfold` are mutually inverse, and thus that the following `fold`/`build` rule holds for nested types:

Theorem 1 *If* `f` *is a higher-order functor,* `c` *and* `a` *are functors,* `h` *is the structure map of an algebra* `Alg f a`*, and* `g` *is any function of closed type* `forall x. Alg f x -> Nat c x`*, then*

```
hfold h . hbuild g = g h                                    (2)
```

Note that the *application* of `ffold h` to `fbuild g` in (1) has been generalised by the *composition* of `hfold h` and `hbuild g` in (2). This is because `c` remains uninstantiated in the nested setting, whereas it is specialised to the unit type in the inductive one. For our running example, we have the following:

Example 6 *The instantiation of (2) for perfect trees is*

```
foldPTree l n . buildPTree g = g l n
```

From Section 2, to ensure that a higher-order functor F on $\mathcal{C}$ has an initial algebra we need that the category $[\mathcal{C},\mathcal{C}]$ has an initial object and ω-colimits, and that F preserves ω-colimits. But only the latter actually needs to be verified since the initial object and ω-colimits in $[\mathcal{C},\mathcal{C}]$ are inherited from those in $\mathcal{C}$.

4 Generalised Folds, Builds, and Short Cut Fusion

In this section we recall the generalised `fold` combinators — here called `gfold`s — from the literature (1; 3; 6). We also introduce a corresponding generalised `build` combinator `gbuild` and a `gfold`/`gbuild` fusion rule for each nested type. We show that, just as the `gfold` combinators are instances of the `hfold` combinators, so the `gbuild` combinators are instances of the `hbuild` combinators, and the `gfold`/`gbuild` rules can be derived from the `hfold`/`hbuild` rules. These results are important because, until now, it has been unclear which general principles should underpin the definition of `gfold` combinators, and because `gbuild` combinators and `gfold`/`gbuild` rules have not existed. Our rendering of the generalised combinators and fusion rules as instances of their counterparts from Section 3 shows that *the same principles of initial algebra semantics that govern the behaviour of* `hfold`*,* `hbuild`*, and* `hfold`/`hbuild` *fusion also govern the behaviour of* `gfold`*,* `gbuild`*, and* `gfold`/`gbuild` *fusion.* In particular, whereas `gfold`s have previously been defined only for certain syntactically defined classes of higher-order functors, initial algebra semantics allows us to define `gfold`s for *all* higher-order functors, and to do so in such a way that our `gfold`s coincide with the `gfold`s in the literature whenever the latter are defined. Our reduction of `gfold`s to `hfold`s can thus be seen as an extension of the results of (1).

Generalised folds arise when we want to consume a structure of type `Mu f a` for a *single* type `a`. The canonical example is the function `psum :: PTree Int -> Int` which sums the (integer) data in a perfect tree (16). It seems `psum` cannot be expressed in terms of `hfold` since `hfold` consumes expressions of polymorphic type, and `PTree Int` is not such a type. Naive attempts to define `psum` will fail because the recursive call to `psum` must consume a structure of type `PTree (Int,Int)` rather than `PTree Int`. These considerations have led to the development of *generalised* `fold` *combinators* for nested types (1; 3; 6). Like the `hfold` combinator for a nested type, the generalised `fold` takes as input an algebra of type `Alg f g` for a higher-order functor `f` whose fixed point the nested type constructor is. But while the `hfold` returns a result of type `Nat (Mu f) g`, the corresponding generalised `fold` returns a result of the more general type `Nat (Mu f ‘Comp‘ g) h`, where `Comp` represents the composition of functors:

```
newtype Comp g h a = Comp {icomp :: g (h a)}

instance (Functor g, Functor h) => Functor (g ‘Comp‘ h) where
    fmap k (Comp t) = Comp (fmap (fmap k) t)
```

However, `Mu f ‘Comp‘ g` is not necessarily an inductive type constructor, so there is no clear theory upon which the definition of `gfold`s can be based. Alternatively, `psum` can be defined using an accumulating parameter as follows:

```
psum :: PTree Int -> Int
psum xs = psumAux xs id

psumAux :: PTree a -> (a -> Int) -> Int
psumAux (PLeaf x) e  = e x
psumAux (PNode xs) e = psumAux xs (\(x,y) -> e x + e y)
```

Here, `psumAux` generalises `psum` to take as input an environment of type `a -> Int` which is updated to reflect the extra structure in the recursive calls. Thus, `psumAux` is a polymorphic function which returns a continuation of type `(a -> Int) -> Int`. To construct our generalised `fold`s, we will actually use a generalised form of continuation whose environment stores values parameterised by a functor `g`, and whose results are parameterised by a functor `h`. We have

```
newtype Ran g h a = Ran {iran :: forall b. (a -> g b) -> h b}
```

Categorically, these continuations are just right Kan extensions, which are defined as follows. Given a functor $G : \mathcal{A} \to \mathcal{B}$ and a category $\mathcal{C}$, precomposition with G defines a functor $\_ \circ G : [\mathcal{B}, \mathcal{C}] \to [\mathcal{A}, \mathcal{C}]$. A *right Kan extension* is a right adjoint to $\_ \circ G$. More concretely, given a functor $H : \mathcal{A} \to \mathcal{C}$, the right Kan extension of H along G, written $\mathsf{Ran}_G H$, is defined via the natural isomorphism $[\mathcal{A}, \mathcal{C}](F \circ G, H) \cong [\mathcal{B}, \mathcal{C}](F, \mathsf{Ran}_G H)$. The classic end formula (see (19) for details) underlies the implementation of a right Kan extension in Haskell as a universally quantified type, with relational parametricity guaranteeing that we do get a proper end as opposed to simply a universally quantified formula.

We stress that no categorical knowledge of Kan extensions is needed to understand the remainder of this paper; indeed, the few concepts we use which involve them will be implemented in Haskell. However, we retain the terminology to highlight the mathematical underpinnings of generalised continuations, and to bring to a wider audience the computational usefulness of Kan extensions.

The bijection characterising right Kan extensions can be implemented as

```
toRan :: Functor k => Nat (k `Comp` g) h -> Nat k (Ran g h)
toRan s t = Ran (\env -> s (Comp (fmap env t)))

fromRan :: Nat k (Ran g h) -> Nat (k `Comp` g) h
fromRan s (Comp t) = iran (s t) id
```

The polymorphic function `psumAux` is a natural transformation from `PTree` to `Ran (Con Int) (Con Int)`, where `Con k` is the constantly `k`-valued functor defined by `newtype Con k a = Con {icon :: k}`.[2] This suggests that an alternative to inventing a generalised `fold` combinator to define `psumAux` is to first endow the functor `Ran (Con Int) (Con Int)` with an appropriate algebra structure and then define `psumAux` as the application of `hfold` to that algebra.

Giving a direct definition of an algebra structure for `Ran g h` turns out to be rather cumbersome. Instead, we circumvent this difficulty by drawing on the intuition inherent in the continuations metaphor for `Ran g h`. If `y` is a functor, then an *interpreter* for `y` with a polymorphic environment which stores values parameterised by `g` and whose results are parameterised by `h` is a function of type `type Interp y g h = Nat y (Ran g h)`. Such an interpreter takes as input a value of type `y a` and an environment of type `a -> g b`, and returns a result of type `h b`. Associated with the type synonym `Interp` is the function

```
runInterp :: Interp y g h -> y a -> (a -> g b) -> h b
runInterp k y e = iran (k y) e
```

An *interpreter transformer* can now be defined as a function which takes as input a higher-order functor `f` and functors `g` and `h`, and returns a map which takes as input an interpreter for any functor `y` and produces an interpreter for the functor `f y`. We can define a type of interpreter transformers in Haskell by

```
type InterpT f g h = forall y. Functor y =>
                            Interp y g h -> Interp (f y) g h
```

We argue informally that interpreter transformers are relevant to the study of nested types. Recall that the `hfold` combinator for a higher-order functor `f` must compute a value for each value of type `Mu f a`, and the functor `Mu f` can be

[2] The use of constructors such as `Con` and `Comp` is required by Haskell. Although the price of lengthier code and constructor pollution is unfortunate, we believe it is outweighed by the benefits of having an implementation.

considered the colimit of the sequence of approximations `f^n 0`, where `0` is the functor whose value is constantly the empty type. We can define an interpreter for `0` since there is nothing to interpret. An interpreter transformer allows us to produce an interpreter for `f 0`, then for `f^2 0`, and so on, and thus contains all the information necessary to produce an interpreter for `Mu f`. This intuition can be formalised by showing that interpreter transformers are algebras. We have:

```
toAlg :: InterpT f g h -> Alg f (Ran g h)
toAlg interpT = interpT idNat

fromAlg :: HFunctor f => Alg f (Ran g h) -> InterpT f g h
fromAlg h interp = h . hfmap interp
```

where `idNat :: Nat f f` is the identity natural transformation defined by `idNat = id`. Parametricity and naturality guarantee that `toAlg` and `fromAlg` are mutually inverse. Thus, interpreter transformers are simply algebras over right Kan extensions presented in a more computationally intuitive manner. We now define

```
gfold :: HFunctor f => InterpT f g h -> Nat (Mu f) (Ran g h)
gfold interpT = hfold (toAlg interpT)
```

The function

```
rungfold :: HFunctor f =>
                 InterpT f g h -> Mu f a -> (a -> g b) -> h b
rungfold interpT   = iran . gfold interpT
```

removes the `Ran` constructor from the output of `gfold` to expose the underlying function. An alternative definition of `gfold` would have ``Nat (Mu f `Comp` g) h`` as its return type and use `toRan` to compute functions whose natural return types are of the form `Nat (Mu f) (Ran g h)`. But, contrary to expectation, `gfold` combinators defined in this way are not expressive enough to represent all uniform consumptions with return types of this form. For example, the function `fmap :: (a -> b) -> Mu f a -> Mu f b` in the `Functor` instance declaration for `Mu f` given at the end of this section is written using the `gfold` combinator defined above. However, defining `fmap` as the composition of `toRan` and a call to a `gfold` combinator with return type of the form ``Nat (Mu f `Comp` g) h`` is not possible. This is because the use of `toRan` assumes the functoriality of `Mu f` — which is precisely what defining `fmap` establishes.

We have thus defined the first-ever generalised `fold` combinators for *all* higher-order functors and done so *uniformly* in terms of their corresponding `hfold` combinators. Our definition is different from, but, as noted above, provably equal to, the definition given in (1) for the class of functors treated there. It also differs from all definitions of generalised `fold`s appearing in the literature, since none of these establishes that the `gfold` combinator for any nested type can be defined in terms of its corresponding `hfold` combinator.

We come full circle by using the specialisation of the `gfold` combinator to the higher-order functor `HPTree` to define a function `sumPTree` which is equivalent to `psum`. We first define an auxiliary function `sumAuxPTree`, in terms of which `sumPTree` itself will be defined. To define `sumAuxPTree` we must define an interpreter transformer; we do this by giving its two unbundled components:

```
type PLeafT g h = forall y. forall a.
                       Nat y (Ran g h) -> a -> Ran g h a
type PNodeT g h = forall y. forall a.
                       Nat y (Ran g h) -> y (a,a) -> Ran g h a

gfoldPTree :: PLeafT g h -> PNodeT g h -> PTree a -> Ran g h a
gfoldPTree l n = foldPTree (l idNat) (n idNat)

psumL :: PLeafT (Con Int) (Con Int)
psumL pinterp x = Ran (\e -> e x)

psumN :: PNodeT (Con Int) (Con Int)
psumN pinterp x = Ran (\e -> runInterp pinterp x (update e))

update e (x,y) = e x `cplus` e y
                 where cplus (Con a) (Con b) = Con (a+b)

sumAuxPTree :: PTree a -> Ran (Con Int) (Cont Int) a
sumAuxPTree = gfoldPTree psumL psumN

sumPTree :: PTree Int -> Int
sumPTree = icon . fromRan sumAuxPTree . Comp . fmap Con
```

Thus, `sumPTree` is essentially `fromRan sumAuxPTree` — ignoring the constructor pollution introduced by Haskell, that is.

Our next example uses generalised `fold`s to show that untyped λ-terms are an instance of the monad class. Here, `gfold` is used to define the `bind` operation `>>=`, which captures substitution.

```
subAlg :: InterpT HLam (Mu HLam) (Mu HLam)
subAlg k (HVar x)   = Ran (\e -> e x)
subAlg k (HApp t u) = Ran (\e -> In (HApp (runInterp k t e)
                                          (runInterp k u e)))
subAlg k (HAbs t)   = Ran (\e -> In (HAbs (runInterp k t (lift e))))

lift e (Just x) = fmap Just (e x)
lift e Nothing  = In (HVar Nothing)

instance Monad (Mu HLam) where
   return  = In . HVar
   t >>= f = rungfold subAlg t f
```

Finally, note that we can also put the generic form of generalised `fold`s to good use. We illustrate this by using `gfold` to establish that all nested types are functors as follows. Let `Id a = Id unid :: a`. Then

```
mapAlg :: HFunctor f => InterpT f Id (Mu f)
mapAlg k t = let k1 t = runInterp k t Id
             in Ran (\e -> In (hfmap k1 (ffmap (unid . e) t)))

instance HFunctor f => Functor (Mu f) where
      fmap k t = rungfold mapAlg t (Id . k)
```

It is natural to ask whether or not there exist generalised `build` combinators corresponding to our generalised `fold`s. Since the `gfold` combinators return results of type `Nat (Mu f) (Ran g h)`, their corresponding generalised `build`s should produce results with types of the form `Nat c (Mu f)`. But the fact that generalised `fold`s are representable as certain `hfold`s suggests that we should be able to define such generalised `build`s in terms of our `hbuild` combinators, rather then defining entirely new `build` combinators. Taking `c` to be the left Kan extension `Lan g h` dual to `Ran g h` (see (19) for details) and implemented in Haskell as

```
data Lan g h a = forall b. Lan (g b -> a, h b)
```

we have

```
gbuild :: HFunctor f => (forall x. Alg f x -> Nat (Lan g h) x)
                        -> Nat (Lan g h) (Mu f)
gbuild = hbuild
```

The Haskell functions

```
toLan :: Functor f => Nat h (f `Comp` g) -> Nat (Lan g h) f
toLan s (Lan (val, v)) =  fmap val (icomp (s v))

fromLan :: Nat (Lan g h) f -> Nat h (f `Comp` g)
fromLan s t = Comp (s (Lan (id, t)))
```

code the bijection between types of the form `Nat h (f `Comp` g)` and `Nat (Lan g h) f` characterising left Kan extensions. The simplicity of the definition of `gbuild` highlights the importance of choosing an appropriate formalism, here Kan extensions, to reflect inherent structure. While it appears that defining the `gbuild` combinators requires no effort at all once we have the `hbuild` combinators, the key insight lies in introducing the abstraction `Lan` and using the bijection between `Nat h (f `Comp` g)` and `Nat (Lan g h) f`.

As an immediate consequence of Theorem 1 we have

Theorem 2 *If* `f` *is a higher-order functor,* `g`*,* `h` *and* `h'` *are functors,* `k` *is an algebra presented as an interpreter transformer of type* `InterpT f g h'`*, and* `l` *is a function of closed type* `forall x. Alg f x -> Nat (Lan g h) x`*, then*

```
gfold k . (gbuild l) = l (toAlg k)          (3)
```

Examples of generalised short cut fusion in action will be given in a journal version of this paper.

5 Conclusion and Future work

We have extended the standard initial algebra semantics for nested types to augment the standard `hfold` combinators for such types with the first-ever Church encodings, `hbuild` combinators, and `hfold`/`hbuild` rules for them. In fact, we have capitalised on the uniformity of the isomorphism between nested types and their Church encodings to derive a single generic standard `hfold` combinator, a single generic standard `hbuild` operator, and a single generic standard `hfold`/`hbuild` rule, each of which can be specialised to any particular nested type of interest. We have also defined a generic generalised `fold` combinator, a generic generalised `build` combinator, and a generic generalised `fold`/`build` rule, each of which is uniformly interdefinable with the corresponding standard construct for nested types. The uniformity of both the standard and generalised constructs derives from a technical approach based on initial algebras of functors. Our generalised `fold` combinators coincide with the generalised `fold`s in the literature when the latter are defined. Moreover, our approach is the first to apply to *all* nested types, and thus provides a principled and elegant foundation for programming with them. We also give the first (Haskell) implementation of these combinators, and illustrate their use in several examples. We believe this paper contributes to a settled foundation for programming with nested types.

In fact, our approach also straightforwardly dualises to the coinductive setting. Shortage of space prevents us from giving the corresponding constructs and results in detail in this paper, so we simply present their implementation:

```
type CoAlg f g  = Nat g (f g)

hunfold :: HFunctor f => CoAlg f g -> Nat g (Mu f)
hunfold k x = In (hfmap (hunfold k) (k x))

hdestroy :: (HFunctor f, Functor c) =>
                  (forall g. CoAlg f g -> Nat g c) -> Nat (Mu f) c
hdestroy g = g out

out :: Nat (Mu f) (f (Mu f))
out (In t) = t

-- fusion rule: hdestroy g . hunfold k = g k
```

The categorical semantics of (13) reduces correctness of `fold`/`build` rules to the problem of constructing a parametric model which respects that semantics. An alternative approach is taken in (18), where the operational semantics-based parametric model of (22) is used to validate the fusion rules for algebraic data types introduced in that paper. Extending these techniques to tie the correctness of `fold`/`build` rules into an operational semantics of the underlying functional language is one direction for future work. Finally, the techniques of this paper may provide insights into theories of `fold`s, `build`s, and fusion rules for advanced data types, such as mixed variance data types, GADTs, and dependent types.

Bibliography

[1] A. Abel and R. Matthes and T. Uustalu. Iteration schemes for higher-order and nested datatypes. *Theoretical Computer Science* 333(1-2) (2005), pp. 3–66.

[2] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. Proc., Computer Science Logic, pp. 453–468, 1999.

[3] I. Bayley. Generic Operations on Nested Datatypes. Ph.D. Dissertation, University of Oxford, 2001.

[4] R. Bird and L. Meertens. Nested datatypes. Proc., Mathematics of Program Construction, pp. 52–67, 1998.

[5] R. Bird and R. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming* 9(1) (1998), pp. 77–91.

[6] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing* 11(2) (1999), pp. 200–222.

[7] P. Blampied. Structured Recursion for Non-uniform Data-types. Ph.D. Dissertation, University of Nottingham, 2000.

[8] M. Fiore and G. D. Plotkin and D. Turi. Abstract syntax and variable binding. Proc., Logic in Computer Science, pp. 193–202, 1999.

[9] A. Gill. Cheap Deforestation for Non-strict Functional Languages. Ph.D. Dissertation, Glasgow University, 1996.

[10] A. Gill and J. Launchbury and S. L. Peyton Jones. A short cut to deforestation. Proc., Functional Programming Languages and Computer Architecture, pp. 223–232, 1993.

[11] N. Ghani and M. Haman and T. Uustalu and V. Vene. Representing cyclic structures as nested types. Presented at Trends in Functional Programming, 2006.

[12] N. Ghani and P. Johann and T. Uustalu and V. Vene. Monadic augment and generalised short cut fusion. Proc., International Conference on Functional Programming, pp. 294–305, 2005.

[13] N. Ghani and T. Uustalu and V. Vene. Build, augment and destroy. Universally. Proc., Asian Symposium on Programming Languages, pp. 327–347, 2003.

[14] R. Hinze. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science* 3(4) (1999), pp. 193–214.

[15] R. Hinze. Efficient generalized folds. Proc., Workshop on Generic Programming, 2000.

[16] R. Hinze. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming* 10(3) (2000), pp. 305-317.

[17] R. Hinze. Manufacturing datatypes. *Journal of Functional Programming* 11(5) (2001), pp. 493–524.

[18] P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-order and Symbolic Computation* 15 (2002), pp. 273–300.

[19] S. MacLane. Categories for the Working Mathematician. Springer-Verlag, 1971.

[20] C. Martin and J. Gibbons and I. Bayley. Disciplined efficient generalised folds for nested datatypes. *Formal Aspects of Computing* 16(1) (2004), pp. 19–35.

[21] C. McBride and J. McKinna. View from the left. *Journal of Functional Programming* 14(1) (2004), pp. 69–111.

[22] A. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science* 10 (2000), pp. 1–39.

[23] A. Takano and E. Meijer. Shortcut deforestation in calculational form. Proc., Functional Programming Languages and Computer Architecture, pp. 306–313, 1995.