Haskell Programming with Nested Types: A Principled Approach[†]

Patricia Johann and Neil Ghani[‡]
({patricia,ng}@cis.strath.ac.uk)
University of Strathclyde, Glasgow, G1 1XH, Scotland

Abstract. Initial algebra semantics is one of the cornerstones of the theory of modern functional programming languages. For each inductive data type, it provides a Church encoding for that type, a `build` combinator which constructs data of that type, a `fold` combinator which encapsulates structured recursion over data of that type, and a `fold`/`build` rule which optimises modular programs by eliminating from them data constructed using the `build` combinator, and immediately consumed using the `fold` combinator, for 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 in Haskell. Specifically, the standard `fold`s derived from initial algebra semantics have been considered too weak to capture commonly occurring patterns of recursion over data of nested types in Haskell, and no `build` combinators or `fold`/`build` rules have until now been defined for nested types. This paper shows that standard `fold`s are, in fact, sufficiently expressive for programming with nested types in Haskell. It also defines `build` combinators and `fold`/`build` fusion rules for nested types. It thus shows how initial algebra semantics provides a principled, expressive, and elegant foundation for programming with nested types in Haskell.

1. Introduction

Initial algebra semantics is one of the cornerstones of the theory of modern functional programming languages. It provides support for `fold` combinators which encapsulate structured recursion over data structures, thereby making it possible to write, reason about, and transform programs in principled ways. Recently, [15] extended the usual initial algebra semantics for inductive types to support not only standard `fold` combinators, but also 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, for each inductive type, a `fold`/`build` rule which optimises modular programs by eliminating from them data of that type constructed using its `build` combinator and immediately consumed using its `fold` combinator.

[†] This is a revised and extended version of the conference paper [23].

[‡] Supported in part by EPSRC grant EP/C511964/2.

Nested data types have become increasingly popular in recent years[4, 6, 7, 8, 16, 17, 18, 20, 28]. They have been used to implement a number of advanced data types in languages, such as the widely-used functional programming language Haskell [32], which support higher-kinded types. Among these data types are those with constraints, such as perfect trees [18]; types with variable binding, such as untyped λ-terms [2, 6, 10]; cyclic data structures [13]; and certain dependent types [29]. In addition, examples of nested types supporting algorithms which are exponentially more efficient than those supported by corresponding inductive structures are given in [31].

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, 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 the type `List a` is, in isolation, an inductive type — i.e., a fixed point of a (certain kind of) data type constructor — the type constructor `List` is seen to define a *family of inductive types*. The kind of recursion captured by inductive types is known as *uniform recursion* [31] or *simple recursion* [39].

Compare the declaration for `List a` with the declaration

```
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. Here, the constructor `Abs` models the bound variable in an abstraction of type `Lam a` by the `Nothing` constructor of type `Maybe a`, and any free variable `x` of type `a` in an abstraction of type `Lam a` by the term `Just x` of type `Maybe a`; example representations of some particular λ-terms are given in Example 3. The key observation about the type `Lam a` is that, by contrast with `List a`, it 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, the type constructor `Lam` defines an *inductive family of types*. The kind of recursion captured by nested types is a special case of *non-uniform recursion* [8, 31] or *polymorphic recursion* [30].

There are many ways to represent inductive families of types [9]. The fact that inductive families can be represented as nested data types in Haskell has generated significant interest in principled techniques for programming with such types. It has also encouraged the development of Haskell libraries of pre-defined functions for manipulating nested types [4, 16]. These library functions can then be combined in a mix-and-match fashion to construct more sophisticated functions which manipulate data of nested types.

Associated with modularly constructed programs are inefficiencies arising from allocating, filling, deconstructing, and deallocating cells for so-called *intermediate data structures*, i.e., data structures which "glue" program components together but otherwise play no role in computations. Even in lazy languages like Haskell this is expensive, both slowing execution time and increasing heap requirements. When the intermediate data structures are lists or other algebraic structures, *short cut fusion* [11, 12, 26] can often be used improve program performance. Short cut fusion is a local transformation based on two combinators for each algebraic type — a `build` combinator, which uniformly produces structures of that type, and a standard `fold` combinator, which uniformly consumes structures of that type — and one corresponding replacement rule for each such type, known as its `fold/build` rule.[1] The `fold/build` rule for an algebraic type replaces a call to the `build` combinator for that type which is immediately followed by a call to the `fold` combinator for that type with an equivalent computation that does not construct the intermediate structure of that type that is introduced by `build` and immediately consumed by `fold`.

The `fold` and `build` combinators for algebraic data types, as well as their associated `fold/build` rules, can be derived from initial algebra semantics. Indeed, these combinators and rules derive uniformly from standard isomorphisms between algebraic data types and their Church encodings. In [14, 15] this theoretical perspective was used to show that even non-algebraic inductive types have associated `fold` and `build` combinators and corresponding `fold/build` rules. This was achieved by extending the well-known initial algebra semantics of inductive types to include a generic Church encoding, generic `build` and `fold` combinators, and a generic `fold/build` rule, each of which can be specialised to any particular inductive type of interest. These specialisations can be used to eliminate intermediate structures of any inductive type that are produced by the instance of `build` for that type and immediately consumed by its instance of `fold`. Details appear in Section 2 below.

[1] The standard `fold` combinator for lists is known as `foldr` in Haskell, and the replacement rule underlying short cut fusion for lists is known as the `foldr/build` rule.

The uniform derivation of [14, 15] is far preferable to defining `fold` and `build` combinators and `fold/build` rules on a case-by-case basis for each inductive type.

Given the increased interest in nested types and the ensuing growth in their use, it is natural to ask whether initial algebra semantics can give a similar principled foundation for structured programming with nested types in Haskell. Until now this has not been considered possible. For example, Bird and Paterson state in the abstract of [7] that the `fold` combinators derived from initial algebra semantics for nested types are not expressive enough to capture certain commonly occurring patterns of structured recursion over data of nested types in Haskell. Indeed, they write:

> *Although the categorical semantics of nested types turns out to be similar to the regular case, the fold functions are more limited because they can only describe natural transformations. Practical considerations therefore dictate the introduction of a generalised fold function in which this limitation can be overcome.*

This echoes similar assertions appearing in Section 6 of [5]. The supposed limitations of the standard `fold`s has led to the development of so-called *generalised* `fold`*s* for nested types in Haskell [4, 7, 16, 17, 18, 20]. But despite their name, these generalised `fold`s have until now not been known to be true `fold`s. As a result, the initial algebra-based general principles that underlie programming with, and reasoning about, standard `fold`s in Haskell have not been available for these generalised `fold`s. Moreover, no corresponding `build` combinators or `fold/build` fusion rules have until now been proposed or defined for nested types.

The major contribution of this paper is to confirm that

> *Initial algebra semantics is enough to provide a principled foundation for structured programming with nested types in Haskell.*

We achieve this by showing that:

- For the class of nested types definable in Haskell, there is no need for the generalised `fold`s from the literature. That is, the `gfold` combinators which implement generalised `fold`s are uniformly interdefinable with the `hfold` combinators which implement the standard `fold`s derived from initial algebra semantics. This means that, contrary to what had previously been thought, the `hfold` combinators capture *exactly the same kinds of recursion*, and so are every bit as expressive, as generalised `fold`s.

Interdefinability also guarantees that the same principles that underlie programming with, and reasoning about, standard `fold`s in Haskell also provide the heretofore missing theoretical foundation for programming with, and reasoning about, generalised `fold`s in that setting.

- We can define `hbuild` combinators for each nested type definable in Haskell. To the best of our knowledge, such combinators have not previously been defined for nested types. Coupling each of these with the corresponding `hfold` combinator gives an `hfold`/`hbuild` rule for its associated nested type, and thus extends short cut fusion from inductive types to nested types in Haskell. Moreover, just as the `gfold` combinators can be defined in terms of the corresponding `hfold` combinators, so a `gbuild` combinator can be defined from each `hbuild` combinator, and a `gfold`/`gbuild` rule can be defined for each nested type in Haskell. Neither `build` combinators nor `fold`/`build` fusion rules developed in this paper have previously been defined for nested types.

We make several other important contributions as well. First, we actually execute the above programme 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 in Haskell. A similar remark applies to the generalised combinators and generalised `fold`/`build` rule. This approach emphasises that our development is highly principled, and allows us to treat the combinators and rules for all nested types in Haskell simultaneously, rather than in an *ad hoc* fashion which depends on the particular type under consideration.

Secondly, our results apply to all nested types definable in Haskell. In particular, this includes nested types defined using type classes, GADTs, monads, and other Haskell features. It is worth noting that interdefinability of standard and generalised `fold`s is considered in [1] for the class of nested types expressible in a variant of F_ω. In fact, [1] goes further and establishes the interdefinability of standard and generalised `fold`s for all higher-ranked types definable in that variant. But because the results of [1] are applicable only to those nested types definable in a particular type theory, because there are manifest differences between their type theory and a Turing complete programming language like Haskell, and because `build` combinators and `fold`/`build` rules for nested types are not given in [1], Haskell programmers may prefer our development.

Thirdly, as in [1], interdefinability of the `gfold` and `hfold` combinators is proved here using right Kan extensions, which can be considered

a form of "generalised continuation" [7]. Bird and Paterson also use right Kan extensions, but only as a *meta-level reasoning device* by which they justify writing object-level programs in terms of generalised `fold`s. By contrast, we use right Kan extensions as an *object-level programming device* which can be used directly to structure Haskell programs. We similarly use left Kan extensions, which can be considered a form of computation involving hidden state, at the object level to establish interdefinability of the `gbuild` and `hbuild` combinators.

Finally, we demonstrate the practical benefit of our development with a variety of examples and a complete implementation of our ideas in Haskell, available at `http://personal.cis.strath.ac.uk/~ng`. The code runs in GHCi version 6.6; the `-fglasgow-exts` option, included in the code file, is needed to handle the nested `forall`-types arising in our nested data type definitions. Our implementation 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.

Our observation that initial algebra semantics is expressive enough to provide a foundation for programming with nested types in Haskell allows us to apply known principles to them, and thus, importantly, to program with them in a principled and effective manner *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. It also serves as a compelling demonstration of the practical applicability of left and right Kan extensions — which, as mentioned above, are the main technical tool used to define the generalised combinators and prove them interdefinable with their counterparts derived from initial algebra semantics — and thus has the potential to render them mainstays of functional programming. More generally, it shows how categorical abstractions can be used to inspire functional programming constructs.

The remainder of this paper is structured as follows. Section 2 recalls the initial algebra semantics of inductive types. It also discusses the relationship between the category theory used in this paper and the functional programming constructs it inspires. Section 3 recalls the derivation of standard `fold` combinators from initial algebra semantics for nested types in Haskell, and defines `build` combinators and

`fold`/`build` rules for these types. Section 4 defines the `gfold` combinators for nested types in Haskell and shows that they are interdefinable with their corresponding `hfold` combinators. It also derives our `gbuild` combinators and `gfold`/`gbuild` rules for nested types in Haskell. Section 5 discusses related work, while Section 6 mentions the coalgebraic duals of our combinators and rules, and draws some conclusions.

2. Initial Algebra Semantics for Inductive Types

In this section we review the standard initial algebra semantics for lists, for algebraic data types, and, finally, for all inductive types. We recall from [14, 15] how the standard initial algebra semantics gives, for each inductive type, a `fold` combinator encapsulating a commonly occuring type-independent pattern of structured recursion over data of that type. We further recall how to extend the standard initial algebra semantics to derive Church encodings, `build` combinators, and `fold`/`build` fusion rules for inductive types. The resulting *extended initial algebra semantics* provides a principled and expressive infrastructure for programming with data having inductive types. This semantics will be generalised in Section 3 to derive a similarly principled and expressive infrastructure for programming with nested types in Haskell.

2.1. Structured programming with Lists

In Haskell, structured programming with lists is accomplished using the built-in list data type constructor `[-]`, the associated data constructors `(:)` and `[]` (for `Cons` and `Nil`, respectively) and the recursion combinator `foldr` defined in Figure 1. The `foldr` combinator is widely used because it captures a commonly occurring type-independent pattern of computation for consuming lists. The development of a substantial collection of techniques for reasoning about programs which uniformly consume lists using `foldr` has further encouraged this form of structured programming over lists. Intuitively, `foldr c n xs` produces a value by replacing all occurrences of `(:)` in `xs` by `c` and the single occurrence of `[]` in `xs` by `n`. For instance, `foldr (+) 0 xs` sums the (numeric) elements of the list `xs`. The combinator `foldr` is included in the Haskell prelude.

Uniform production of lists, on the other hand, is accomplished using the less well-known combinator `build`. This combinator takes as input a function providing a type-independent template for constructing "abstract" lists, and produces a corresponding "concrete" list. For example, `build (\c n -> c 4 (c 7 (c 5 n)))` produces the

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr c n xs = case xs of []   -> n
                          z:zs -> c z (foldr c n zs)

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

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

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

Figure 1. Combinators and functions for lists.

list `[4,7,5]`. The definitions of `build` and the other list-processing functions used in this paper are given in Figure 1.

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 [11, 12], a widely-used program optimisation which capitalises on the uniform production and consumption of lists to improve the performance of list-manipulating programs. Short cut fusion is based on the `foldr`/`build` rule which states that, for every function `g :: forall b. (a -> b -> b) -> b -> b`,

```
foldr c n (build g) = g c n                                    (1)
```

When this rule is considered as a replacement rule oriented from left to right and is applied to a program, it yields a new "fused" program that avoids constructing the intermediate list produced by `build g` and immediately consumed by `foldr c n` in the original. For example, if `sum` and `map` are defined as in Figure 1, if `sqr x = x * x`, and if `sumSqs = sum . map sqr`, then

```
sumSqs :: [Int] -> Int
sumSqs xs = 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
```

No intermediate lists are produced by this final version of `sumSqs`.

2.2. Structured programming with Algebraic Types

The above infrastructure for structured programming with lists, based on the constructors `(:)` and `[]` and the combinators `foldr` and `build`, can be generalised to other algebraic data types. An *algebraic data type* is, intuitively, a fixed point of a (covariant) functor which maps type variables to a type constructed using sum, product, arrow, `forall`, and other algebraic data types defined over those type variables — see [33] for a formal definition. Algebraic data types can be parameterised over multiple types, and can be mutually recursive, but not all types definable using Haskell's `data` mechanism are algebraic. For example, neither nested types nor fixed points of mixed variance functors are algebraic.

Every algebraic data type `D` has associated `fold` and `build` combinators. Operationally, the `fold` combinator for an algebraic data type `D` takes as input appropriately typed replacement functions for each of `D`'s constructors, together with a data element `d` of `D`. It replaces all (fully applied) occurrences of `D`'s constructors in `d` by applications of their corresponding replacement functions. The `build` combinator for an algebraic data type `D` takes as input a function `g` providing a type-independent template for constructing "abstract" data structures from values. It instantiates all (fully applied) occurrences of abstract constructors appearing in `g` with corresponding applications of the "concrete" constructors of `D`.

A typical example of a non-list algebraic data type is the type of arithmetic expressions over variables of type `a`. This type can be represented in Haskell as

```
data Expr a = EVar a
            | Lit Int
            | Op Ops (Expr a) (Expr a)

data Ops = Add | Sub | Mul | Div
```

The `buildE` and `foldE` combinators associated with `Expr` appear in Figure 2. Many commonly occurring patters of computation which consume expressions can be written using `foldE` — see, for example, the function `accum` of Figure 2, which maps an expression to the list of variables occurring in it.

Just as compositions of list-consuming and list-producing functions can be fused using the `foldr`/`build` rule, so compositions of expression-consuming and expression-producing functions defined using `foldE` and `buildE` can be fused via the `fold`/`build` rule for expressions. This rule states that, for every function `g :: forall b. (a -> b) -> (Int -> b)`

```
foldE :: (a -> b) -> (Int -> b) ->
            (Ops -> b -> b -> b) -> Expr a - > b
foldE v l o e = case e of
                 EVar x -> v x
                 Lit i -> l i
                 Op op e1 e2 -> o op (foldE v l o e1)
                                     (foldE v l o e2)

buildE :: (forall b. (a -> b) -> (Int -> b) ->
              (Ops -> b -> b -> b) -> b) -> Expr a
buildE g = g EVar Lit Op

accum :: Expr a -> [a]
accum = foldE (\x -> [x]) (\i -> []) (\op -> (++))

mapE :: (a -> b) -> Expr a -> Expr b
mapE env e = buildE (\v l o -> foldE (v . env) l o e)
```

Figure 2. Combinators and functions for expressions.

```
-> (Ops -> b -> b -> b) -> b,
```

$$\texttt{foldE v l o (buildE g) = g v l o} \tag{2}$$

For example, if `env :: a -> b` is a renaming environment and `e` is an expression, then a function `renameAccum` which accumulates variables of renamings of expressions can be defined modularly as

```
renameAccum :: (a -> b) -> Expr a -> [b]
renameAccum env e = accum (mapE env e)
```

Using rule (2) and the definitions in Figure 2 we can derive the following more efficient version of `renameAccum`:

```
  renameAccum env e
= foldE (\x -> [x]) (\i -> []) (\op -> (++))
     (buildE (\v l o -> foldE (v . env) l o e))
= (\v l o -> foldE (v . env) l o e)
     (\x -> [x]) (\i -> []) (\op -> (++))
= foldE ((\x -> [x]) . env) (\i -> []) (\op -> (++)) e
```

Unlike the original version of `renameAccum`, this one does not construct the renamed expression but instead accumulates variables "on the fly" while renaming.

2.3. Structured programming with Inductive Types

As it turns out, a similar story about structured programming can be told for all inductive data types. *Inductive data types* are fixed points of functors, i.e., of type constructors which support `fmap` functions, and *inductive data structures* are data structures of inductive type. 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

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

stating that `fmap` preserves identities and composition. Satisfaction of the functor laws is, however, not enforced by the compiler. Instead, it is the programmer's responsibility to ensure that the `fmap` function for each instance of Haskell's `Functor` class behaves appropriately.

As is well-known [14, 15, 37], 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
```

The `fbuild` and `ffold` combinators for a functor `f` 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 type `forall b. (f b -> b) -> b`, then rules (1) and (2) above generalise to the following `fold/build` rule for `M f`:

```
ffold h (fbuild g) = g h
```
(3)

The following example of short cut fusion is borrowed from [14]. An *interactive input/output computation* [35] is either i) a value of type `a`, ii) an input action, which, for every input token of type `i`, results in a new interactive input/output computation, or iii) an output configuration consisting of an output token of type `o` and a new interactive input/output computation. This is captured in the declaration

```
data IntIO i o a = Val a
                 | Inp (i -> IntIO i o a)
                 | Outp (o, IntIO i o a)
```

The data type `IntIO i o a` is `M (K i o a)` for the functor `K i o a` defined by

```
data K i o a b = Vk a | Ik (i -> b) | Ok (o,b)

instance Functor (K i o a) where
  fmap k (Vk x)     = Vk x
  fmap k (Ik h)     = Ik (k . h)
  fmap k (Ok (y,z)) = Ok (o, k z)
```

We can obtain `ffold` and `fbuild` combinators for this functor by first instantiating the above generic definitions of `ffold` and `fbuild` for `f = K i o a`, and then using standard type isomorphisms to unbundle the type arguments to the functor (and guide the case analysis performed by `ffold`). Unbundling allows us to treat the single argument `h :: K i o a b -> b` to the instantiation of `ffold` as a curried triple of "constructor replacement functions" `v :: a -> b`, `p :: (i -> b) -> b`, and `q :: (o,b) -> b`, and to give these three functions, rather than the isomorphic "bundled" function `h`, as arguments to `ffold`. Unbundling is not in any sense necessary; its sole purpose is to allow the instantiation to take a form more familiar to functional programmers. Similar remarks apply at several places below, and unbundling is performed without comment henceforth. Instantiating `f` to `K i o a`, we have

```
IntIOfold :: (a -> b) -> ((i -> b) -> b) -> ((o,b) -> b)
                                 -> IntIO i o a -> b
intIOfold v p q k = case k of
                      Val x      -> v x
                      Inp h      -> p (intIOfold v p q . h)
                      Outp (y,z) -> q (y, intIOfold v p q z)

intIObuild :: (forall b. (a -> b) -> ((i -> b) -> b) ->
                          ((o,b) -> b) -> b) -> IntIO i o a
intIObuild g = g Val Inp Outp

intIOfold v p q (intIObuild g) = g v p q
```

As discussed in [14, 15], the `ffold` and `fbuild` combinators for inductive types, as well as the corresponding `ffold`/`fbuild` rules, generalise those for algebraic types. Moreover, the `ffold` and `fbuild`

combinators for inductive types are defined uniformly over the functors whose fixed points those types are. We will use this observation to good effect in the next subsection.

2.4. A Theoretical Perspective

Thus far we have purposely taken a determinedly computational point of view so as to make clear the relevance of this paper to programming in general, and to Haskell programming in particular. However, even from a programming perspective the above development is rather incomplete. If we are to generalise the treatment of inductive types given above to more advanced data types, we must ask:

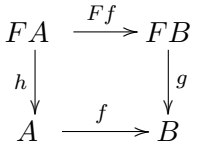
> *Why is the above style of structured programming possible, i.e., why do the* `ffold` *and* `fbuild` *combinators exist for inductive types and why are the associated* `ffold`/`fbuild` *rules correct?*

A principled answer to this question is clearly important if we are to program with, and reason about, the `ffold` and `fbuild` combinators, and also if we are to generalise them to more expressive types.

This section offers precisely such a principled answer. However, readers without the required background in category theory, or whose main focus is not on the categorical foundations of the combinators and fusion rules for advanced data types, can safely omit this section and other categorical discussions in the paper since all of the relevant category-theoretic constructs used in this paper are implemented in Haskell. Readers who choose to do this will miss some of the motivations for the theory of nested types, and some of the connections between the theory of inductive types and the theory of nested types, but will miss no necessary facts. In any case, we do not attempt a complete reconstruction of all of category theory here. Instead, we introduce only those concepts that form the basis of our principled approach to programming with nested types in Haskell.

The key idea underlying our approach is that of *initial algebra semantics*. Within the paradigm of initial algebra semantics, every data type is the carrier μF of the initial algebra of a suitable functor $F : \mathcal{C} \to \mathcal{C}$ for some suitable, fixed category $\mathcal{C}$. In more detail, suppose we have fixed such a category $\mathcal{C}$. An *algebra* for a functor $F : \mathcal{C} \to \mathcal{C}$ (or, simply, an *F-algebra*) is a pair (A, h) where A is an object of $\mathcal{C}$ and $h : FA \to A$ is a morphism of $\mathcal{C}$. Here, A is called the *carrier* of the algebra and h is called its *structure map*. As it turns out, the F-algebras for a given functor F themselves form a category. In the category of F-algebras, a morphism $f : (A, h) \to (B, g)$ is a map $f : A \to B$ in $\mathcal{C}$

such that the following diagram commutes:

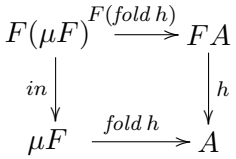


We call such a morphism an *F-algebra homomorphism*.

Now, if the category of F-algebras has an initial object — called an *initial algebra for* F, or, more simply, an *initial* F*-algebra* — then Lambek's Lemma ensures that the structure map of this initial F-algebra is an isomorphism, and thus that its carrier is a fixed point of F. If it exists, the initial F-algebra is unique up to isomorphism. Henceforth we write $(\mu F, in)$ for the initial F-algebra comprising the fixed point μF of F and the isomorphism $in : F(\mu F) \to \mu F$.

2.4.1. *Folds*

The standard interpretation of a type constructor is as a functor F, and the standard interpretation of the data type it defines is as the least fixed point of F. As noted above, this least fixed point is the carrier of the initial F-algebra. Initiality ensures that there is a unique F-algebra homomorphism from the initial F-algebra to any other F-algebra. The map underlying this F-algebra homomorphism is exactly the *fold* operator for the data type μF. Thus if (A, h) is any F-algebra, then $fold\ h : \mu F \to A$ makes the following diagram commute:



From this diagram, we see that the type of *fold* is $(FA \to A) \to \mu F \to A$ and that $fold\ h$ satisfies $fold\ h\ (in\ t) = h\ (F\ (fold\ h)\ t)$. 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 $fold\ h$ 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 $\psi\,.\,fold\ h = fold\ h'$. But note that *fold* fusion [4, 6, 7, 8, 28], is completely different from, and inherently simpler than, the *fold*/*build* fusion which is central in this paper, and which we discuss next; see Section 5 for a full discussion.

It is reasonable to ask at this point when an initial F-algebra is guaranteed to exist for a functor F on a category $\mathcal{C}$. It is possible that

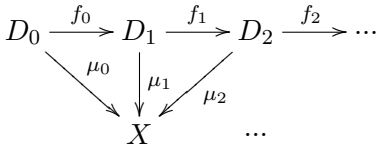
some functors on a given category will have initial algebras and some not, but the following result (see, e.g., [36]) gives sufficient conditions.

THEOREM 1. *If $\mathcal{C}$ has both an initial object and ω-colimits, and if F preserves ω-colimits, then $\mathcal{C}$ will have an initial F-algebra.*

To unpack Theorem 1, we first introduce the idea of an ω-chain. An *ω-chain* (D, f) is a family D of objects D_i together with a family f of morphisms $f_i : D_i \to D_{i+1}$ for all $i \geq 0$. Diagrammatically, we have

$$D_0 \xrightarrow{f_0} D_1 \xrightarrow{f_1} D_2 \xrightarrow{f_2} \cdots$$

A *cocone* $\mu : D \to X$ of an ω-chain (D, f) is an object X and a family of morphisms $\mu_i : D_i \to X$ such that $\mu_i = \mu_{i+1} \circ f_i$ for all $i \geq 0$.



A *colimit* of an ω-chain (D, f) is a cocone $\mu : D \to X$ with the property that if $\nu : D \to Y$ is also a cocone of (D, f), then there is a unique morphism $k : X \to Y$ such that $\nu_i = k \circ \mu_i$. Colimits of ω-chains are also called *ω-colimits*. A functor F *preserves ω-colimits* if, for every ω-colimit $\mu : D \to X$, the cocone $F\mu : FD \to FX$ is also an ω-colimit.

The proof of Theorem 1 considers the particular ω-chain (D, f) given by

$$0 \xrightarrow{!} F0 \xrightarrow{F!} F^2 0 \xrightarrow{F^2!} \cdots$$

where 0 is the initial object in $\mathcal{C}$, F is the particular functor of interest, and ! is the unique morphism from 0 to $F0$. It next observes that, under the hypotheses of the theorem, if $\mu : D \to X$ is an ω-colimit of (D, f) then $F\mu : FD \to FX$ is an ω-colimit of the ω-chain (FD, Ff) given by

$$F0 \xrightarrow{F!} F^2 0 \xrightarrow{F^2!} F^3 0 \xrightarrow{F^3!} \cdots$$

Finally, this observation is used to show directly that if $\nu : FX \to X$ is the unique morphism such that $\mu_{i+1} = \nu \circ F\mu_i$ for all $i \geq 0$, then (X, ν) is an initial F-algebra.

Many categories arising in the semantics of programming languages are well known to have initial objects and ω-colimits. Moreover, in any category, identity and constant functors preserve ω-colimits, as do

products and coproducts of functors which preserve ω-colimits. As a result, all polynomial functors preserve ω-colimits.

Regarding the choice of a suitable category $\mathcal{C}$ in which to work, we note that, unfortunately, a semantics of Haskell does not exist independently of any specific implementation. In particular, there is no known categorical semantics for full Haskell. Nevertheless, programmers and researchers often proceed as though there were, or may someday be, and we follow in this well-established tradition. There is a sense, then, that the treatment of structured programming for nested types put forth in this paper takes place "in the abstract", given that the existence of a suitable category $\mathcal{C}$ is simply assumed, and thus that our programming constructs may best be considered "categorically inspired". But we prefer a slightly different take on the situation, regarding this paper as *prescriptive*, since it indicates properties that any eventual categorical semantics of Haskell should — even intuitively — satisfy.

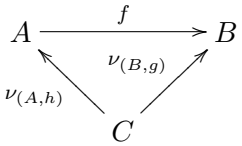
2.4.2. *Church encodings, builds, and fold/build fusion rules*
Although the above discussion shows that *fold* combinators for inductive types can be derived entirely from, and understood entirely in terms of, initial algebra semantics, regrettably the standard initial algebra semantics does not provide a similar principled derivation of the *build* combinators or the correctness of the *fold*/*build* rules. In fact, *build* has been regarded as a kind of optional "add-on" which is not a fundamental part of the basic infrastructure for programming with inductive types. The practical consequence of this has been that the *build* combinators have been largely overlooked and treated as poor relatives of their corresponding *fold* combinators, and thus unworthy of fundamental study.

This situation was rectified in [15], where the standard initial algebra semantics was extended to support not only *fold* combinators for inductive types, but also Church encodings and *build* combinators for them. Indeed, [15] considers 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. We summarise this result and its consequences, which we later apply to derive our `build` combinators for nested types.

If F is a functor on $\mathcal{C}$, then the *forgetful functor* U_F maps F-algebras to objects in $\mathcal{C}$ by forgetting the F-algebra structure. That is, U_F maps an F-algebra (A, h) to its carrier A, and maps an F-algebra homomorphism $f : (A, h) \to (B, g)$ to the underlying $f : A \to B$ in $\mathcal{C}$.

If C is an object in $\mathcal{C}$, then a *U_F-cone for C* comprises, for every F-algebra (A, h), a morphism $\nu_{(A,h)} : C \to A$ in $\mathcal{C}$ such that, for any

F-algebra homomorphism $f : (A, h) \to (B, g)$, we have $\nu_{(B,g)} = f \circ \nu_{(A,h)}$.



We write (C, ν) for this cone, and call C its *vertex* and the morphism $\nu_{(A,h)}$ the *projection* from C to (A, h). A *U_F-cone morphism* $g : (C, \nu) \to (D, \mu)$ between U_F-cones (C, ν) and (D, μ) is a morphism $g : C \to D$ in $\mathcal{C}$ such that for any F-algebra (A, h), we have $\mu_{(A,h)} \circ g = \nu_{(A,h)}$. We call g a *mediating morphism* from C to D. A *U_F-limit* is a U_F-cone to which there is a unique U_F-cone morphism from any other U_F-cone. When they exist, U_F-limits are unique up to isomorphism. Moreover, no extra structure is required of either F or $\mathcal{C}$ for the U_F-limit to exist — it is simply the carrier of the initial F-algebra.

The characterisation of initial algebras as both limits *and* colimits is what we call the *extended initial algebra semantics*. As shown in [15], 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:

- For each F-algebra, the projection from the vertex of the U_F-limit (i.e., from μF) to the carrier of that F-algebra defines the *fold* operator with type $(Fx \to x) \to \mu F \to x$.
- A U_F-cone with vertex C can be thought of as having the type $\forall x.(Fx \to x) \to C \to x$. The unique mediating morphism from the vertex C of a U_F-cone to the vertex μF of the U_F-limit defines the *build* operator with type $(\forall x.\,(Fx \to x) \to C \to x) \to C \to \mu F$.
- The correctness of the *fold*/*build* fusion rule $fold\ h\ .\ build\ g\ =\ g\,h$ then follows from the fact that *fold* after *build* is a projection after a mediating morphism from C to μF, and is thus equal to the projection from C to the carrier of h.

The extended initial algebra semantics thus shows that, given a parametric interpretation of universal quantification for types, there is an isomorphism between the type $C \to \mu F$ and the "generalised Church encoding" $\forall x.(Fx \to x) \to C \to x$.

The term "generalised" reflects the presence of the parameter C, which is absent in the usual Church encodings for inductive types [37]. Choosing C to be the unit type gives the usual isomorphism between an inductive type and its usual Church encoding. This isomorphism comprises precisely the *fold* (up to order of arguments) and *build* operators

for that type. Writing $fold'\ m\ h$ for $fold\ h\ m$ we have

$$\begin{aligned} fold' &:: \mu F \to \forall x.(Fx \to x) \to x \\ build &:: (\forall x.(Fx \to x) \to x) \to \mu F \end{aligned}$$

From this we see that correctness of the *fold*/*build* rule for inductive types is one half of the requirement that *build* and $fold'$ are mutually inverse. A generic `build` combinator and a generic Church encoding for inductive types are essentially given in [37], but attention is restricted there to "functors whose operation on functions are continuous." By contrast, our `build` combinator is entirely generic over all instances of Haskell's functor class. Moreover, the `build`s in [37] do not appear to be derived from any universal property.

In the next section we will see that, in Haskell, nested types can be defined as fixed points of higher-order functors. Categorically speaking, higher-order functors are functors, and thus have associated *fold*s, *build*s, and *fold*/*build* rules. This observation can be used to derive `build` combinators and `fold`/`build` rules for nested types in Haskell. Our derivation will make use of the generalised Church encodings for non-unit type parameters C.

3. Initial Algebra Semantics for Nested Types

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

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

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

Here, 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 lengths are powers of two, and hence illustrate how nested types can be used to capture structural constraints on data types.

The data type of bushes [5] provides an additional example of a nested type.

EXAMPLE 2. *The type of* bushes *over type* `a` *is given by*

```
data Bush a = BLeaf | BNode (a, Bush (Bush a))
```

Note the nested recursive call to `Bush` on the right-hand side of `Bush`'s definition.

Finally, we recall the data type of untyped λ-terms from the introduction.

EXAMPLE 3. *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))
```

Specific elements of type `Lam a` include `Abs (Var Nothing)`, which represents $\lambda x.x$, and `Abs (Var (Just x))`, which represents $\lambda y.x$.

We observed in the introduction that each nested type constructor defines an inductive family of types, rather than a family of inductive types. This leads us to consider type constructors which are themselves defined inductively. Just as inductive types arise as least fixed points of functors which map types to types, so inductive type constructors arise as fixed points of functors which map functors to functors. This observation amounts to recalling that a standard way to solve a parameterised equation is to regard it as a higher-order (in our setting, fixed point) equation with its parameters abstracted.

We thus model nested types as least fixed points of functors on the category of endofunctors on the base category $\mathcal{C}$, i.e., as least fixed points of *higher-order functors* on $\mathcal{C}$. In the category of endofunctors on $\mathcal{C}$, which we write $[\mathcal{C},\mathcal{C}]$, objects are functors on $\mathcal{C}$ and morphisms are natural transformations between functors on $\mathcal{C}$. Thus higher-order functors must preserve identity natural transformations and compositions of natural transformations. From Section 2.4, we know that to ensure that a higher-order functor $F : [\mathcal{C},\mathcal{C}] \to [\mathcal{C},\mathcal{C}]$ has an initial F-algebra — and hence a least fixed point — the category $[\mathcal{C},\mathcal{C}]$ must have an initial object and ω-colimits, and that F must preserve ω-colimits. But only that F preserves ω-colimits actually needs to be verified since the initial object of $[\mathcal{C},\mathcal{C}]$ and ω-colimits in $[\mathcal{C},\mathcal{C}]$ are inherited from those in $\mathcal{C}$. Specifically, the initial object of $[\mathcal{C},\mathcal{C}]$ is simply the functor which maps every object in $\mathcal{C}$ to the initial object of $\mathcal{C}$, and which maps every morphism in $\mathcal{C}$ to the identity morphism on the initial object of $\mathcal{C}$. The ω-colimit of an ω-chain (G, f) in $[\mathcal{C},\mathcal{C}]$ is a functor H on $\mathcal{C}$ together with a family of natural transformations $\mu_i : G_i \to H$ such that (i) for any A, the cocone (HA, μ_A) comprising HA and the $(\mu_i)_A$ is the ω-colimit of the ω-chain (GA, f_A) comprising the G_iA and the $(f_i)_A$, and (ii) for any $f : A \to B$, $Hf : HA \to HB$ is the unique morphism such that $(\mu_i)_B \circ G_if = Hf \circ (\mu_i)_A$. The existence and

uniqueness of this morphism follow from the facts that (HB, ν_B), where $(\nu_i)_B = (\mu_i)_B \circ G_i f$, is a cocone of the ω-chain (GA, fA) and (HA, μ_A) is the colimit of this ω-chain.

Higher-order functors can be implemented by the following Haskell type class, which is a higher-order analogue of Haskell's `Functor` class.

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

A higher-order functor thus maps functors to functors and maps between functors to maps between functors, i.e., natural transformations to natural transformations. The first of these observations is captured by the requirement that a higher-order functor supports an `ffmap` operation. While not explicit in the `HFunctor` class definition, the programmer is expected to verify that if `g` is a functor, then `f g` satisfies the functor laws. The second observation is captured by the requirement that a higher-order functor support an `hfmap` operation. To see this, first observe that the type of natural transformations can be given in Haskell by

```
type Nat g h = forall a. g a -> h a
```

Assuming a parametric interpretation of the `forall` quantifier, an element of the type `Nat g h` can be thought of as a uniform family of maps from `g` to `h`. This ensures that the naturality square for `g` and `h` commutes. Although not explicitly required by the Haskell definition of `Nat g h`, both `g` and `h` are expected to be functors. Moreover, like the `fmap` functions for functors, the `hfmap` functions for higher-order functors are expected to preserve identities and composition. Here, identities are identity natural transformations, and compositions are compositions of natural transformations.

We can now implement nested types as fixed points of `HFunctors` in Haskell. These are defined by

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

Note that because Haskell lacks polymorphic kinding, our implementation cannot use the constructor `M` introduced to represent fixed points of first-order functors above. This is the reason for introducing the new type constructor `Mu` to represent fixed points of higher-order functors. We write `Mu f` for the fixed point of the higher-order functor `f`.

EXAMPLE 4. *The nested types of perfect trees, bushes, and untyped λ-terms from Examples 1, 2, and 3 arise as fixed points of the higher-order functors*

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

data HBush f a = HBLeaf | HBNode (a, f (f a))

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

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

We should check that `HPTree`, `HBush`, and `HLam` are instances of the `HFunctor` class, and that all functor laws requiring programmer verification are satisfied. Such checks are usually straightforward. For example, in the case of `HPTree` we have

```
instance HFunctor HPTree where
  ffmap f (HPLeaf a) = HPLeaf (f a)
  ffmap f (HPNode a) = HPNode (fmap (pair f) a)
  hfmap f (HPLeaf a) = HPLeaf a
  hfmap f (HPNode a) = HPNode (f a)
```

Here, `pair f (x,y) = (f x, f y)`. Verifying that if `g` is a functor then `HPTree g` is a functor amounts to showing that `ffmap id = id` and `ffmap (k . l) = ffmap k . ffmap l` for functions `k` and `l`. That `HPTree` maps natural transformations to natural transformations follows from the type-correctness of `hfmap` under the aforementioned assumption of a parametric model.

We now turn our attention to deriving `fold` and `build` combinators and `fold`/`build` rules for nested types in Haskell. In fact, initial algebra semantics makes this very easy. We simply instantiate all of the ideas from Section 2 in our category of endofunctors. We begin by recalling the standard `fold` combinators for nested types from the literature.

3.1. Folds for Nested Types

As we have seen, one of the strengths of the standard initial algebra semantics for inductive data types is the uniform definition of `fold`s for consuming inductive structures. Categorically, a `fold` operator takes as input an algebra for a functor and returns a morphism from the fixed point of the functor to the carrier of the algebra. Since a nested type is nothing more than a fixed point, albeit of a higher-order functor, the same idea can be used to derive `fold`s for nested types in Haskell. Of course, an algebra must now be an algebra for a higher-order functor

whose fixed point the nested type constructor is. The structure map of such an algebra will thus be a natural transformation. Further, the result of the `fold` will be a natural transformation from the nested type to the carrier of the algebra. These definitions can be implemented in Haskell as

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

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

EXAMPLE 5. *The* `hfold` *combinator for perfect trees is*

```
hfoldPTree :: (forall a. a -> f a) ->
                (forall a. f (a,a) -> f a) ->
                  PTree a -> f a

hfoldPTree f g (PLeaf x)  = f x
hfoldPTree f g (PNode xs) = g (hfoldPTree f g xs)
```

EXAMPLE 6. *The* `hfold` *combinator for bushes is*

```
hfoldBush :: (forall a. f a) ->
               (forall a. (a, f (f a)) -> f a)
                 Bush a -> f a

hfoldBush l n BLeaf          = l
hfoldBush l n (BNode (x, b)) =
         n (x, hfmap (hfoldBush l n) (hfoldBush l n b))
```

EXAMPLE 7. *The* `hfold` *combinator for* λ*-terms is*

```
hfoldLam :: (forall a. a -> f a) ->
              (forall a. f a -> f a -> f a) ->
                (forall a. f (Maybe a) -> f a) ->
                  Lam a -> f a

hfoldLam v ap ab (Var x)   = v x
hfoldLam v ap ab (App d e) = ap (hfoldLam v ap ab d)
                                (hfoldLam v ap ab e)
hfoldLam v ap ab (Abs l)   = ab (hfoldLam v ap ab l)
```

The uniqueness of `hfold`, guaranteed by its derivation from initial algebra semantics, provides the basis for the correctness of `fold` fusion for nested types [8]. 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.

3.2. Church Encodings and Builds For Nested Types

The extended initial algebra semantics also gives us `build` combinators for nested types in Haskell. To see this, we begin by recalling the derivation of the generic `build` combinator for inductive types as explained in Section 2.4. The key observation was that, given a parametric interpretation of universal type quantification, there is an isomorphism between $C \to \mu F$ and the generalised Church encoding $\forall x.(Fx \to x) \to C \to x$. But this isomorphism holds for *all* functors, including higher-order ones. We should therefore be able to instantiate this isomorphism for a higher-order functor F to derive a generic Church encoding and a generic `build` combinator for nested types. And indeed we can, provided we interpret the isomorphism in an endofunctor category.

The Church encoding of a nested type which is representable in Haskell as `Mu f` for a higher-order functor `f` can be written as

```
forall x. (Alg f x) -> Nat c x
```

We therefore define the generic `build` combinator for such nested types to be

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

It is worth noticing that the generic `hbuild` combinator follows the definitional format of the generic `build` combinator for inductive types: it applies its argument to the structure map `In` of the initial algebra of the higher-order functor `f` over which it is parameterised. Of course, the structure morphism of the initial algebra `In` is not a function of type `f (M f) -> M f` in this setting, but rather a natural transformation of type `Nat (f (Mu f)) (Mu f)`.

We can instantiate the generic `hbuild` combinator for any particular nested type of interest.

EXAMPLE 8. *The Church encoding and* `hbuild` *combinator for perfect trees are given concretely by*

```
forall x. (forall a. a -> x a) ->
          (forall a. x (a,a) -> x a) ->
          (forall a. c a -> x a)
```

and

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

EXAMPLE 9. *The Church encoding and* `hbuild` *combinator for bushes are given concretely by*

```
forall x. (forall a. x a) ->
          (forall a. (a, x (x a)) -> x a) ->
          (forall a. c a -> x a)
```

and

```
hbuildBush :: (forall x. (forall a. x a) ->
                          (forall a. (a, x (x a)) -> x a) ->
                          (forall a. c a -> x a)) ->
                  Nat c Bush
hbuildBush g = g BLeaf BNode
```

EXAMPLE 10. *The Church encoding and* `hbuild` *combinator for* λ*-terms are given concretely by*

```
forall x. (forall a. a -> x a) ->
          (forall a. x a -> x a -> x a) ->
          (forall a. x (Maybe a) -> x a) ->
          (forall a. c a -> x a)
```

and

```
hbuildLam :: (forall x. (forall a. a -> x a) ->
                        (forall a. x a -> x a -> x a) ->
                        (forall a. x (Maybe a) -> x a) ->
                        (forall a. c a -> x a)) ->
                  Nat c Lam
hbuildLam g = g Var App Abs
```

3.3. Short Cut Fusion for Nested Types

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 in Haskell:

FUSION RULE FOR NESTED TYPES *If* `f` *is a higher-order functor,* `c` *and* `x` *are functors,* `h` *is (the structure map of) an algebra* `Alg f x`*, and* `g` *is any function of type* `forall x. Alg f x -> Nat c x`*, then*

```
hfold h . hbuild g = g h
```
(4)

Note that the *application* of `ffold h` to `fbuild g` in (3) has been generalised by the *composition* of `hfold h` and `hbuild g` in (4). This is because `c` remains uninstantiated in the nested setting, whereas it was specialised to the unit type in the inductive one.

EXAMPLE 11. *The instantiations of (4) for perfect trees, bushes, and λ-terms are*

```
hfoldPTree l n . (hbuildPTree g) = g l n

hfoldBush l n . (hbuildBush g) = g l n

hfoldLam v ap ab . (hbuildLam g) = g v ap ab
```

To give the flavour of short cut fusion in action, we consider the non-trivial application of the bit reversal algorithm of Hinze [18]. The bit reversal algorithm describes an operation on lists of length 2^n which swaps elements whose indices have binary representations that are the reverses of one another. For example, the bit reversal algorithm transforms the list `[a0,a1,a2,...,a7]` into `[a0,a4,a2,a6,a1,a5,a3,a7]`.

We start with the simple and modular bit reversal protocol

```
brp1 :: [a] -> [a]
brp1 = shuffle . unshuffle
```

where `shuffle`, `unshuffle`, and the auxiliary functions in terms of which they are defined are given in Figure 3. The function `shuffle` uniformly and recursively consumes a perfect tree in a "left-right inorder" fashion, and prepends the list of data appearing as first elements of pairs at the `PNode`s of subtrees onto the list of data appearing as second elements of these pairs. The function `unshuffle` uniformly produces a perfect tree by constructing, for each non-empty list of data, the `PNode` obtained by first splitting that list into two sublists of roughly equal

```
shuffle :: PTree a -> [a]
shuffle = hfoldPTree (\x -> [x]) (cat . unzip)

unshuffle :: [a] -> PTree a
unshuffle = hbuildPTree unsh

cat :: ([a],[a]) -> [a]
cat (xs,ys) = xs ++ ys

zip' :: ([a],[b]) -> [(a,b)]
zip' (xs,ys) = zip xs ys

unsh :: (forall a. a -> x a) ->
          (forall a. x (a,a) -> x a) ->
             (forall a. [a] -> x a)
unsh u z [e] = u e
unsh u z es  = z (unsh u z (zip' (uninter es)))

uninter :: [a] -> ([a],[a])
uninter []     = ([],[])
uninter (x:xs) = (x:as,bs)
                   where (bs,as) = uninter xs
```

Figure 3. Functions for a bit reversal protocol

size. This is done by alternating which sublist data elements are added to the front of, and then zipping these sublists together. Unshuffling a list to get a perfect tree and then shuffling the data in that tree thus reverses the bits in the input list.

The function `brp1` is typical of modularly constructed programs, in that it produces and then immediately consumes a data structure, in this case a perfect tree. We can use the instantiation of (4) to perfect trees from Example 11 to transform `brp1` into the following more efficient program `brp2` which doesn't construct the intermediate perfect tree.

```
brp2 :: [a] -> [a]
brp2 [x] = [x]
brp2 xs  = (cat . unzip . brp2 . zip' . uninter) xs
```

This can be accomplished via the calculation

```
  brp1 xs
= hfoldPTree (\x -> [x]) (cat . unzip) (hbuildPTree unsh xs)
```

```
= unsh (\x -> [x]) (cat . unzip) xs
= if length xs == 1 then [head xs]
   else (cat . unzip) (unsh (\x -> [x])
                            (cat . unzip)
                            (zip' (uninter xs)))
= if length xs == 1 then xs
   else (cat . unzip . brp2 . zip' . uninter) xs
```

4. Generalised Folds, Builds, and Short Cut Fusion

In the last section we derived generic `hfold` and `hbuild` combinators, and a generic `hfold/hbuild` fusion rule for nested types definable in Haskell. Our derivations are based on the very same initial algebra semantics that underlies the analogous development for inductive types discussed in Section 2.4. Both the theory underlying our development, and its implementation, are clean, simple, and principled.

4.1. Generalised folds

In this section we recall the generalised `fold` combinators for nested types in Haskell — here called `gfold`s — from the literature [4, 7]. We also introduce a generic generalised `build` combinator `gbuild` and a generic `gfold/gbuild` fusion rule which can be instantiated to give a `gbuild` combinator and a `gfold/gbuild` rule for each nested type. We show that the `gfold` combinator is an instance of the generic `hfold` combinator, that the generic `gbuild` combinator is an instance of the generic `hbuild` combinator, and that the generic `gfold/gbuild` rule is an instance of the generic `hfold/hbuild` rule. These results are important because, until now, it has been unclear which general principles should underpin the definition of the `gfold` combinators for nested types in Haskell, and because `gbuild` combinators and `gfold/gbuild` rules for them have not heretofore existed. Our rendering of the generalised combinators and fusion rule 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 govern the behaviour of* `gfold`*,* `gbuild`*, and* `gfold/gbuild` *fusion.*

Our reduction of `gfold`s to `hfold`s can be seen as a counterpart for nested types definable in Haskell to the results for rank-2 functors in

the type-theoretic setting of [1]. On the other hand, [1] doesn't mention `build` combinators or `fold/build` rules for nested types at all.

Generalised folds arise when we want to consume a structure of type `Mu f a` for a *single* type `a`. The canonical example from the literature involves the function `psum :: PTree Int -> Int` which sums the (integer) data in a perfect tree [18]. At first glance it seems `psum` cannot be expressed in terms of `hfold` since `hfold` consumes data of polymorphic type, and `PTree Int` is not such a type. At the same time, any naive attempt to define `psum` will fail because the recursive call to `psum` must consume a structure of type `PTree (Int,Int)` rather than `PTree Int`. More generally, we often want to consume expressions involving one specific instance of a nested type, rather than a polymorphic family of elements of a nested type. This is precisely what generalised `fold`s for nested types are designed to do [4, 7].

Like the `hfold` combinator for a nested type in Haskell, its 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)
```

Note, however, that `Mu f ‘Comp‘ g` is not necessarily an inductive type constructor. As a result, there is no clear theory upon which the definition of `gfold`s can be based, and it is unclear what general principles should underpin them. One practical consequence of this lack of principled foundations for generalised `fold`s is that it has not been clear how to reason about these combinators for those higher-order functors which have heretofore been known to support them.

A major contribution of this paper is to show that the `hfold` combinators defined in Section 3.1 for nested types in Haskell are expressive enough to implement their generalised `fold`s. Indeed, we derive the generic `gfold` combinator from the corresponding generic `hfold` combinator (and, trivially, vice-versa). In this way we extend the class of higher-order functors for which generalised `fold` combinators can be defined to include higher-order functors definable using all the features of Haskell. In addition to providing structured recursion combinators for such higher-order functors, our derivation of generalised `fold`s as particular standard `fold`s for higher-order functors in Haskell makes it

possible to lift the reasoning principles supported by the `hfold` combinators to reasoning principles for the generalised `fold` combinators. It also allows us to define associated generalised `build` combinators and generalised `fold`/`build` fusion rules for all nested types in Haskell.

We use the example of `psum` to illustrate our derivation. First note that `psum` can be defined by specialising an auxiliary function whose type generalises that of `psum` 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 see that the generalised `fold` for `PTree` is an instance of its `hfold` counterpart, 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}
```

This `Ran`-type was first introduced by Bird and Paterson [7], who used it for meta-level reasoning about nested types. In addition, they introduced the continuations metaphor for it. We show below that `Ran`-types can also be used as object-level devices for structuring programs.

The categorical constructs represented by `Ran`-types are just right Kan extensions, which can be defined as follows. Given endofunctors[2] G and H on $\mathcal{C}$, the *right Kan extension of H along G*, written $\mathsf{Ran}_G H$, is a pair (R, ϵ) comprising an endofunctor R on $\mathcal{C}$ and a natural transformation ϵ from $R \circ G$ to H with the property that if (K, α) is another such pair, then there exists a unique natural transformation σ from K to R such that $\alpha = \epsilon \circ \sigma_G$. If it exists, $\mathsf{Ran}_G H$ is unique up to natural isomorphism. Moreover, the assignment $\sigma \mapsto \epsilon \circ \sigma_G$ constitutes a natural isomorphism between natural transformations from K to R and those from $K \circ G$ to H which is natural in K. This isomorphism actually determines $\mathsf{Ran}_G H$ from G and H, and so can be seen as characterising $\mathsf{Ran}_G H$.

[2] Although when viewed categorically G and H need not have the same domain and codomain, they do in this research and so we take advantage of this to simplify the presentation of Kan extensions.

The classic representation of the right Kan extension $\mathsf{Ran}_G H = (R, \epsilon)$ defines R to be the functor which maps an object C to the end $\int_B (C \to GB) \to HB$ (and a morphism $f : C \to C'$ to the unique morphism from $\int_B (C \to GB) \to HB$ to $\int_B (C' \to GB) \to HB$, which exists by the universal property of the end $\int_B (C' \to GB) \to HB$) justifies our implementation of $\mathsf{Ran}_G H$ in Haskell as a parameterised family of universally quantified types (see [27] for details). We first use the fact that $(C \to GB) \to HB$ is essentially the type of a polymorphic function to implement the right Kan extension $\mathsf{Ran}_G H$ as the `a`-parameterised family `Ran g h a` of types `forall b. (a -> g b) -> h b`. The proof that this implementation really does satisfy the isomorphism characterising $\mathsf{Ran}_G H$ then involves constructing a natural isomorphism between `Nat k (Ran g h)` and `Nat (k ‘Comp‘ g) h`. The essence of this construction is captured in the definitions of `fromRan` and `toRan` below.

The `psum` example above illustrates how right Kan extensions can be used to ensure that the `hfold` combinators derived from initial algebra semantics are expressive enough to capture forms of recursion traditionally thought to require `gfold`s. One question worth asking is when a category $\mathcal{C}$ has enough structure for right Kan extensions to exist. As we have already observed, right Kan extensions can be expressed as ends. Since ends can be expressed as limits, it suffices to require that $\mathcal{C}$ have all limits. While this is not an overly strong condition, it does exclude, for example, realisability models such as PER which do not have *all* limits. This is a pity because PER is often thought of as one of the canonical models of universal type quantification. Fortunately the situation for PER is recoverable by observing that, as shown in [3], `forall`-types are interpreted as ends in PER, and PER is thus guaranteed to have at least those limits which are Kan extensions.

Although we have taken care to motivate our results categorically, 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 Kan extensions 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. We will in fact do so throughout the paper to emphasise that our results are inspired by category theory, and to illustrate how categorical ideas can be transcribed into Haskell code.

With these definitions in place, the polymorphic function `psumAux` can be represented as a natural transformation from the functor `PTree`

to the functor `Ran (Con Int) (Con Int)`, where `Con k` is the constantly `k`-valued functor defined by[3]

```
newtype Con k a = Con {icon :: k}
```

This suggests that an alternative to inventing a generalised `fold` combinator to define `psumAux` is to endow the functor `Ran (Con Int) (Con Int)` with an `HPTree`-algebra structure and then define `psumAux` to be the application of `hfold` to that algebra. The functions

```
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
```

constitute a Haskell implementation of the isomorphism between `Nat (k `Comp` g) h` and `Nat k (Ran g h)` which characterises right Kan extensions. So if we can endow `Ran g h` with an `f`-algebra structure — i.e., if we can construct a term of type `Alg f (Ran g h)` for the higher-order functor `f` — then we can use `fromRan` to write the generalised `fold` for `f` with return type `Nat (Mu f `Comp` g) h` in terms of the instance of `hfold` for `f` with return type `Nat (Mu f) (Ran g h)`. The observation that this is indeed possible is the starting point for our derivation of generalised `fold`s as `hfold`s over `Ran`-types. It is worth observing that the definition of `toRan` relies on the functoriality of `k`, whereas that of `fromRan` does not. This asymmetry crucially informs our choice of type for the generic `fold` combinator later in this section.

Giving a direct definition of an algebra structure for the generalised continuation `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` is a function of type

```
type Interp y g h = Nat y (Ran g h)
```

Such an interpreter uses a polymorphic environment which stores values parameterised by `g` and whose results are parameterised by `h`. It therefore takes as input a term 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

[3] 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.

```
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 the 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
```

Types equivalent to `Interp` and `InterpT` appear in [1].

We can 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 term of type `Mu f a`, and that the functor `Mu f` can be thought of as the colimit of the sequence of approximations `f^n 0` of `n`-fold compositions of `f` applied to the functor `0` whose value is constantly the empty type (which is the initial object in this setting). Clearly, we can define an interpreter for the functor `0` since there is nothing to interpret. Moreover, an interpreter transformer will allow us to next produce an interpreter for `f 0`, then for `f^2 0`, and so on. Thus an interpreter transformer contains all of the information necessary to produce an interpreter for `Mu f`. This intuition that interpreter transformers contain all of the information required to define generalised `fold`s can now be formalised by showing that interpreter transformers are algebras. In Haskell, 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

idNat :: Nat f f
idNat = id
```

The definition of `toAlg` requires that `Ran g h` is a member of the `Functor` class. This is established via the instance declaration

```
instance Functor (Ran g h) where
  fmap f (Ran c) = Ran (\d -> c (d . f))
```

Parametricity and naturality guarantee that `toAlg` and `fromAlg` are mutually inverse. Indeed, we have

```
  toAlg (fromAlg k)
= toAlg (\ interp -> k . hfmap interp)
= (\ interp -> k . hfmap interp) idNat
= k . hfmap idNat
= k . idNat
= k
```

and

```
  fromAlg (toAlg interpT)
= fromAlg (interpT idNat)
= \ interp -> (interpT idNat) . (hfmap interp)
= \ interp -> interpT interp
= interpT
```

The fourth equality in the first derivation uses the fact that `hfmap` preserves identity natural transformations. The third equality in the second derivation uses the naturality in `y` of the type

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

of `interpT`, and the fourth equality there holds by extensionality. Categorically, the fact that `toAlg` and `fromAlg` are mutual inverses is just a specific instantiation of the Yoneda Lemma. Thus we see that interpreter transformers are simply more computationally intuitive presentations of algebras whose carriers are right Kan extensions. Of course, such transformers may also be interesting in their own right, with applications other than the one mentioned here.

We now use these observations to give our generic `gfold` combinator for nested types in Haskell. This `gfold` combinator will take as input an algebra for `Ran g h` presented as an interpreter transformer, and return a polymorphic function which consumes a nested type to produce a generalised continuation. Concretely, we define our generic `gfold` combinator by

```
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 continuation, which is more useful in practice.

An alternative definition of `gfold` would have return type `Nat (Mu f ‘Comp‘ g) h` and use `toRan` to compute functions whose natural return types are of the form `Nat (Mu f) (Ran g h)`. But, contrary to expectation, a `gfold` combinator given by such an alternative definition is not expressive enough to represent all uniform consumptions with return types of the form `Nat (Mu f) (Ran g h)`. 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 easily written in terms of the `gfold` combinator defined above. On the other hand, 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. Note that this is not a semantic issue, but is a consequence of Haskell's typechecking.

In summary, we have made good on our promise to show that the generic `gfold` combinator for nested types in Haskell is interdefinable with the generic `hfold` combinator for such types. Our definition differs from all characterisations of generalised `fold`s appearing in the functional programming literature, since none of these establishes interdefinability.

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`. For this, 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, which we do by giving its two unbundled components. We have

```
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 = hfoldPTree (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) (Con 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.

Rather than using the `PTree` data type declaration from Section 3, we could instead have defined `PTree` to be `Mu HPTree`. In this case, functoriality of `PTree` (which is required in the definition of `sumPTree` above) would be obtained from Example 13 rather than directly. Similar comments apply at several places below.

Because the `gfold` combinators are just particular instances of the `hfold` combinators, and because we concretely gave the `hfold` combinators for the nested types `Bush` and `Lam` in Section 3, we do not give concrete presentations of the corresponding `gfold` combinators here. Instead, we give two additional applications of generalised `fold`s.

EXAMPLE 12. *Generalised* `fold`*s can be used to show that untyped* λ*-terms are an instance of the monad class. The generalised* `fold` *can be used to define the* `bind` *operation* `>>=`*, which captures substitution, as follows:*

```
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
```

That the `return` *and* `>>=` *operations for* `Mu HLam` *satisfy the monad laws can established by direct calculation using the uniqueness of the* `hfold` *operator.*

EXAMPLE 13. *Generalised* `fold`*s can also be used to show that nested types in Haskell are instances of the* `Functor` *class. We have*

```
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)
```

Here, `Id` *is the identity functor, given in Haskell by*

```
newtype Id a = Id {unid :: a}

instance Functor Id where
   fmap f (Id x) = Id (f x)
```

That the `fmap` *operation defined above for* `Mu f` *satisfies the functor laws can be established by direct calculation.*

4.2. Generalised Builds

In the previous section we noted that, at first glance, the standard `hfold` combinators for nested types in Haskell appear to be too polymorphic to express a consumer of a structure of type `Mu f a` for a specific type `a`. The standard resolution of this problem from the functional programming literature has been to define generalised `fold` combinators for such types which, given appropriate inputs, return consumers with types of the form `Nat (Mu f ‘Comp‘ g) h`. But, as we have just shown, there is in fact no need to invent wholly new generalised `fold` combinators. Instead, we can simply convert each generalised `fold` combinator with return type `Nat (Mu f ‘Comp‘ g) h` to the corresponding `hfold` combinator with return type `Nat (Mu f) (Ran g h)` as needed.

It is natural to ask if there are generalised `build` combinators for nested types in Haskell that correspond to their generalised `fold`s. Intuitively, such a generalised `build` should produce expressions of type `Mu f (g a)` for some `f`, `g`, and `a`, so that if a generalised `build` is followed by a generalised `fold` which consumes a structure of type `Mu f (g a)`, then an appropriate generalised `fold`/`build` rule can be used to eliminate the intermediate structure of this type. 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 generalised `build` combinators. We show in this section that we can indeed derive generalised `build`s for nested types in Haskell in this way, and that we can do so in a manner generic over the data type under consideration.

In the next section we will show that a generic generalised `fold/build` rule can also be deduced from the generic `hfold/hbuild` rule for nested types in Haskell that we have already considered in Section 3.3.

Since the generic `gfold` combinator returns results of type `Nat (Mu f) (Ran g h)`, its corresponding generic generalised `build` should produce results with types of the form `Nat c (Mu f)`. Taking `c` to be the existential type[4]

```
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
```

Computationally, the `Lan`-type `forall b. Lan (g b -> a, h b)` can be regarded as comprising a hidden state `b`, an observation function of type `g b -> a`, and an element of the type `h b` obtained by applying the functor `h` to the hidden state `b`. Categorically, `Lan`-types are left Kan extensions. Left Kan extensions are duals of right Kan extensions, and can be defined as follows. Given endofunctors G and H on $\mathcal{C}$, the *left Kan extension of H along G*, written $\mathsf{Lan}_G H$, is a pair (L, ϵ) comprising an endofunctor L on $\mathcal{C}$ and a natural transformation ϵ from H to $L \circ G$ with the property that if (S, α) is another such pair, then there exists a unique natural transformation σ from L to S such that $\alpha = \sigma_G \circ \epsilon$. If it exists, $\mathsf{Lan}_G H$ is unique up to natural isomorphism. Moreover, the assignment $\sigma \mapsto \sigma_G \circ \epsilon$ constitutes a natural isomorphism between natural transformations from L to K and those from H to $K \circ G$ which is natural in K. This isomorphism actually determines $\mathsf{Lan}_G H$ from G and H, and so can be seen as characterising $\mathsf{Lan}_G H$.

The classic representation of the left Kan extension $\mathsf{Lan}_G H = (L, \epsilon)$ as the functor which maps an object C to the coend $\int^B (GB \to C) \times HB$ (and a morphism $f : C \to C'$ to the unique morphism from $\int^B (GB \to C) \times HB$ to $\int^B (GB \to C') \times HB$, which exists by the universal property of the coend $\int^B (GB \to C) \times HB$) justifies our implementation of $\mathsf{Lan}_G H$ in Haskell as a parameterised family of existentially quantified types (see [27] for details). The proof that this implementation really does satisfy the isomorphism characterising $\mathsf{Lan}_G H$ then involves constructing a natural isomorphism between `Nat (Lan g h) k` and `Nat h (k ‘Comp‘ g)`. The essence of this construction is captured in the following definitions of `fromLan` and `toLan`:

[4] In a data type declaration in Haskell, a universal quantifier outside a constructor is read as existentially quantifying the variable it binds.

```
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)))
```

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 generic `gbuild` combinator requires no effort at all once we have the generic `hbuild` combinator, 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`.

We conclude this section by using the specialisation of the `gbuild` combinator to the higher-order functor `HPTree` to define a function `mkTree :: Int -> PTree Int` which takes an integer `n` as input and returns the perfect tree of depth `n` storing `n` in all of its leaves. We have

```
gbuildPTree :: (forall x. (forall a. a -> x a) ->
                          (forall a. x (a,a) -> x a) ->
                          (forall a. Lan g h a -> x a)) ->
               Lan g h a -> PTree a
gbuildPTree g = g PLeaf PNode

tree :: (forall a. a -> x a) ->
          (forall a. x (a,a) -> x a) ->
            Nat (Lan (Con Int) (Con Int)) x
tree l n (Lan (z, Con 0)) = l (z (Con 0))
tree l n x                = n (tree l n (count x))

count :: Lan (Con Int) (Con Int) a ->
            Lan (Con Int) (Con Int) (a,a)
count (Lan (z, Con n)) =
            Lan (\i -> (z (s i), z (s i)), Con (n-1))

s :: Con Int a -> Con Int a
s (Con n) = Con (n+1)

mkTree :: Int -> PTree Int
mkTree n = gbuildPTree tree (Lan (icon, Con n))
```

4.3. Generalised Short Cut Fusion

As an immediate consequence of the fusion rule for nested types we have

GENERALISED FUSION RULE FOR NESTED TYPES *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 term of type* `forall x. Alg f x -> Nat (Lan g h) x`*, then*

```
gfold k . (gbuild l) = l (toAlg k)                    (5)
```

We can apply the specialisation of this rule to fuse the modular function `smTree = sumAuxPTree . mkTree`. This gives

```
  smTree (Lan (z, Con n))
= (gfoldPTree psumL psumN . gbuildPTree tree) (Lan (z, Con n))
= tree (psumL idNat) (psumN idNat) (Lan (z, Con n))
= if n == 0 then Ran (\e -> e (z (Con 0)))
   else psumN idNat (tree (psumL idNat)
                          (psumN idNat)
                          (count (Lan (z, Con n))))
= if n == 0 then Ran (\e -> e (z (Con 0)))
   else Ran (\e -> iran (tree (psumL idNat)
                         (psumN idNat)
                         (count (Lan (z, Con n))))
                (update e))
= if n == 0 then Ran (\e -> e (z (Con 0)))
   else Ran (\e -> iran (smTree (count (Lan (z, Con n))))
                         (update e))
```

Thus, we have

```
  runInterp smTree (Lan (z, Con n)) e
= if n == 0 then e (z (Con 0))
   else runInterp smTree (count (Lan (z, Con n))) (update e)
```

Although this fused version of the `smTree` looks more complicated, it avoids construction of the intermediate perfect tree.

5. Related Work

The first attempt to give initial algebra semantics for nested types goes back to Bird and Meertens [5], who sought to marry the conceptual importance of nested types in functional programming with the standard initial algebra semantics for algebraic types. Their overall aim was to expand the class of data types over which polytypic programming could be performed. Although they base their semantics on higher-order functors and use these to define `fold`s for nested types, they conclude from consideration of specific programs over nested types that initial algebra semantics is limited and loses expressive power. Specifically, they assert that standard `fold`s can only express computations whose results are natural transformations (see Section 6); this assertion is echoed in [7]. To overcome the perceived lack of expressiveness of standard `fold`s, Bird and Meertens introduce several alternative approaches to defining `fold`s for specific nested types. Unfortunately, it is not clear, even to them, whether or not these approaches are generalisable to other such types.

In another attempt to overcome the perceived limitations of standard `fold`s for nested types, Bird and Paterson [7] introduced generalised `fold`s into the literature. They show how to construct a generalised `fold` combinator for each member of a syntactically restricted class of nested types, as well as how these combinators can be used to express computations whose results are not natural transformations. The importance of right Kan extensions to the study of generalised `fold`s and standard `fold`s first appears in this paper — but they are used only at the meta-level to prove uniqueness of generalised `fold`s and correctness of `map` fusion laws and `fold` fusion laws for generalised `fold`s, rather than as object-level programming structuring devices. Kan extensions thus serve as a meta-tool in [7] to support the use of generalised `fold`s. By contrast, our work uses Kan extensions as an object-level tool to eliminate the need for generalised `fold`s in favour of standard `fold`s, and thus to return initial algebras to center stage as a powerful and expressive foundation for structured programming with nested types in Haskell.

We stress that the fusion laws proved for generalised `fold`s by Bird and Paterson — and by others for other notions of `fold`s over nested types — do not include `fold/build` fusion. Of course, this would be impossible given that `build`s have not heretofore been defined for nested types. Specifically, `fold` fusion is concerned with when, for any notion of a `fold` under consideration, a `fold` followed by another function can be written as a `fold`, i.e., when the equivalence `f . fold g = fold h` holds. This is quite different from, and only tangentially related to, the

issue of when `fold h . build g = g h`. Conditions under which `fold` fusion can be applied typically derive from the uniqueness of the map `fold h` under consideration.

Combinators other than `fold`s have been defined for certain syntactic classes of nested types as well. For example, [16] defines polytypic `map`, equality, and reduction functions for a class of functors indexed over so-called rational trees. The question of whether initial algebra semantics of higher-order functors can serve as a viable basis for polytypism on nested types is raised in [16] but, interestingly, no attempt is made there to answer the question. The thesis [4] trades a functional approach to nested types for a relational approach, and uses the latter to give three types of generic `fold`s, as well as to define `map`, `zip`, membership, and reduction functions, and `map` and `fold` fusion rules, for nested types in Haskell.

A different approach to polytypic programming, involving modeling types by terms of the simply typed lambda calculus augmented with a family of recursion operators, and defining polytypic functions by induction on the syntax of type constructors, is presented in [19]. This approach succeeds in handling all nested types expressible in Haskell, but places considerable demands on the type system. In [8], `fold`s are defined for nested types by folding over infinite structures, called algebra families, that contain all possible values for data constructors. An initial algebra semantics is given for algebra families, and algebra family `fold`s and `map`s are defined. In [4] it is shown that nested types represent constraints on regular data types, and thus that nested data types can always be removed from programs by embedding them into regular data types. Our work shows that we can have the expressivity of nested types and the benefits of initial algebra semantics, all without resorting to such embeddings. In fact, all approaches to polytypic programming with nested types in Haskell are based on the premise that the standard `fold`s derived from initial algebra semantics are insufficiently expressive for solving practical problems — a premise we show in this paper to have been erroneous from the outset.

A related line of inquiry concerns the efficiency of generalised `fold`s for nested types. Hinze [17] gives a variation, defined by induction on the structure of data type definitions, of the generalised `fold`s of Bird and Paterson for a syntactically defined class of nested types. More efficient versions of these alternative generalised `fold`s are then derived by observing that Bird and Paterson's generalised `fold`s require extra `map` parameters, and that these parameters are the source of inefficiencies. By giving more general types to some of these extra `map` parameters, and the fusing generalised `fold`s with these `map`s to get new `fold`s, these inefficiencies can be eliminated. Unfortunately, not all alternative

generalised `fold`s have corresponding efficient versions, and [17] leaves open the question of whether the generalised `fold`s of [7] are amenable to the same efficiency improvements as Hinze's variations are.

An affirmative answer to this question is provided in [28], which uses initial algebra semantics to give unique characterisations of efficient counterparts to the generalised `fold`s of [7]. Like the others, this paper considers only a restricted class of functors, but it gives efficient generalised `fold`s inductively for this class, and then proves their uniqueness. Also given are `map` and `fold` fusion rules, and a "`fold` equivalence" result which relates the efficient generalised `fold`s to the ordinary generalised `fold`s for the class of nested data types for which both are defined. Similarities between the initial algebra definitions of efficient generalised `fold`s and Mendler-style inductive types are noted in the conclusion, which suggests that it might be possible to extend the framework of [38] to include them.

Picking up on this thread, [1] provides an in-depth study of iteration for higher-ranked types definable in extensions of F^ω with recursion combinators and associated β-rewrite rules. It uses a type-theoretic presentation of right Kan extensions to provide reduction-preserving translations of these extensions into F^ω, from which termination of their reduction rules follows from strong normalisation of F^ω. It also establishes interdefinability of standard and generalised `fold`s for all nested types definable in the type theory considered there. By contrast, we consider all nested types definable — using type classes, GADTs, monads, and other features — in the Turing complete programming language Haskell, and show how right Kan extensions can be used to establish interdefinability of the generalised and standard `fold` combinators for such types. We also show how left Kan extensions can be used at the object level to establish interdefinability of the `gbuild` and `hbuild` combinators for nested types in Haskell. Analogues of our `build` combinators and `fold`/`build` fusion rules are not discussed in [1], even for the nested types expressible in the type theory considered there.

Applications involving nested types abound. The use of a nested type to implement a bit reversal protocol goes back to [18], which considers in detail the problem of programming with perfect trees, but does not attempt to develop a theory of nested types. Data structure implementations based on nested types appear in [31], as well as in [20] and [21], which contain applications involving nested types and higher-order nested types which capitalise on the use of nested types to record constraints. Other applications appear in, for example, [6] and [22].

6. Conclusion and Future work

We have extended the standard initial algebra semantics for nested types in Haskell to augment the standard `hfold` combinators for such types with `hbuild` combinators and `hfold`/`hbuild` rules for them. In fact, we have capitalised on the uniformity of the isomorphism between such types and their Church encodings to give a single generic `hbuild` combinator and a single generic `hfold`/`hbuild` rule, each of which can be specialised to any such type of interest. We have also shown that the generalised `fold` combinators from the literature are uniformly interdefinable with appropriate instances of the generic `hfold` combinator for nested types in Haskell, and we have defined generalised `build` combinators and generalised `fold`/`build` rules for these types. Like the definitions of the `hbuild` combinator and the `hfold`/`hbuild` rule, those for `gbuild` and the `gfold`/`gbuild` rule can also be defined generically.

The uniformity of both the standard and generalised constructs derives from a technical approach based on initial algebras of functors. Our approach applies to all nested types definable in Haskell, and thus provides a principled and elegant foundation for programming with them. We give a completely generic 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 Haskell.

This work reported in this paper can be seen as providing a means of incorporating nested types and accompanying combinators into systems based on initial algebra semantics of data types — such as the Haskell extension PolyP [25] — in such a way that fundamental changes to the basic framework of those systems are not required.

It is worth noting that our development also applies in the coinductive setting. In the case of coinductive types, it is now well known that a generic `unfold` combinator can be defined:

```
unfold :: Functor f => (x -> f x) -> x -> M f
unfold k x = Inn (fmap (unfold k) (k x))
```

The uniformity of this definition entails that it can be specialised to any functor definable in Haskell. The coinductive dual of `build` is known as `destroy`, and in [15] a generic `destroy` combinator was given for coinductive types:

```
destroy :: Functor f =>
            (forall x . (x -> f x) -> x -> c) -> M f -> c
destroy g = g outt

outt (Inn t) = t
```

This combinator can be used to define generic `unfold` and `destroy` combinators for programming with nested types in Haskell. Like the `hfold` and `hbuild` combinators, these are defined uniformly over instances of the `HFunctor` class. We have

```
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 =>
             (forall g. Functor 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
```

Finally, we have the following `unfold`/`destroy` rule for nested types in Haskell:

```
hdestroy g . hunfold k = g k
```

It is straightforward to derive generalised versions of these combinators using the techniques of this paper.

The categorical semantics of [15] reduces correctness of `fold`/`build` rules to the problem of constructing parametric models which respect that semantics. The category-theoretic ideas underlying the results of this paper entail a similar reduction, which we see as prescribing properties that any eventual categorical semantics of the underlying functional language should satisfy. An alternative approach to correctness is taken in [26], where the operational semantics-based parametric model of [34] is used to validate the fusion rules for algebraic 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. Benchmarking the rules and developing a preprocessor for automatically locating instances where `fold`/`build` rules for nested types can be applied are additional topics of interest. Finally, the techniques of this paper may provide insights into theories of `fold`s, `build`s, and fusion rules for more advanced data types, such as mixed variance data types, GADTs, and dependent types [24].

References

1. Abel, A., Matthes, R., and Uustalu, T. Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science* 333(1-2) (2005), pp. 3–66.
2. Altenkirch, T., and Reus, B. Monadic presentations of lambda terms using generalized inductive types. Proc., Computer Science Logic, pp. 453–468, 1999.
3. Bainbridge, E. S., Freyd, P. J., Scedrov, A., and Scott, P. J. Functorial polymorphism. *Theoretical Computer Science* 70(1) (1990), pp. 35–64. Corrigendum in 71(3) (1990) 431.
4. Bayley, I. Generic Operations on Nested Datatypes. Ph.D. Dissertation, University of Oxford, 2001.
5. Bird, R. and Meertens, L. Nested datatypes. Proc., Mathematics of Program Construction, pp. 52–67, 1998.
6. Bird, R. and Paterson, R. de Bruijn notation as a nested datatype. *Journal of Functional Programming* 9(1) (1999), pp. 77–91.
7. Bird, R. and Paterson, R. Generalised folds for nested datatypes. *Formal Aspects of Computing* 11(2) (1999), pp. 200–222.
8. Blampied, P. Structured Recursion for Non-uniform Data-types. Ph.D. Dissertation, University of Nottingham, 2000.
9. Dybjer, P. Inductive Families. *Formal Aspects of Computing* 6(4) (1994), pp. 440–465.
10. Fiore, M., Plotkin, G. D., and Turi, D. Abstract syntax and variable binding. Proc., Logic in Computer Science, pp. 193–202, 1999.
11. Gill. A. Cheap Deforestation for Non-strict Functional Languages. Ph.D. Dissertation, Glasgow University, 1996.
12. Gill, A., Launchbury, J. and Peyton Jones, S. L. A short cut to deforestation. Proc., Functional Programming Languages and Computer Architecture, pp. 223–232, 1993.
13. Ghani, N., Haman, M., Uustalu, T., and Vene, V. Representing cyclic structures as nested types. Presented at Trends in Functional Programming, 2006.
14. Ghani, N., Johann, P., Uustalu, T., and Vene, V. Monadic augment and generalised short cut fusion. Proc., International Conference on Functional Programming, pp. 294–305, 2005.
15. Ghani, N., Uustalu, T., and Vene, V. Build, augment and destroy. Universally. Proc., Asian Symposium on Programming Languages, pp. 327–347, 2003.
16. Hinze, R. Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science* 3(4) (1999), pp. 193–214.
17. Hinze, R. Efficient generalized folds. Proc., Workshop on Generic Programming, pp. 1–16, 2000.
18. Hinze, R. Functional Pearl: Perfect trees and bit-reversal permutations. *Journal of Functional Programming* 10(3) (2000), pp. 305–317.

19. Hinze, R. A New Approach to Generic Functional Programming. Proc., Principles of Programming Languages, pp. 119–132, 2000.
20. Hinze, R. Manufacturing datatypes. *Journal of Functional Programming* 11(5) (2001), pp. 493–524.
21. Hinze, R. and Juering, J. Generic Haskell: Applications. In *Generic Programming: Advanced Lectures*, pp. 57–97, 2003.
22. Hughes, R. J. M. and Swierstra, S. D. Polish parsers, step by step. Proc., International Conference on Functional Programming, pp. 239–248, 2003.
23. Johann, P. and Ghani, N. Initial algebra semantics is enough! Proc., Typed Lambda Calculi and Applications, pp. 207–222, 2007.
24. Johann, P. and Ghani, N. Foundations for Structured Programming with GADTs. Proc., Principles of Programming Languages, pp. 297–308, 2008.
25. Jansson, P. and Juering, J. PolyP - a polytypic programming language extension. Proc., Principles of Programming Languages, pp. 470–482, 1997.
26. Johann, P. A generalization of short-cut fusion and its correctness proof. *Higher-order and Symbolic Computation* 15 (2002), pp. 273–300.
27. MacLane, S. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
28. Martin, C, Gibbons, J., and Bayley, I. Disciplined efficient generalised folds for nested datatypes. *Formal Aspects of Computing* 16(1) (2004), pp. 19–35.
29. McBride, C. and McKinna, J. View from the left. *Journal of Functional Programming* 14(1) (2004), pp. 69–111.
30. Mycroft, A. Polymorphic type schemes and recursive definitions. Proc., International Symposium on Programming, pp. 217–228, 1984.
31. Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, 1998.
32. Peyton Jones, S. L. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
33. Pitts, A. Parametric polymorphism, recursive types, and operational equivalence. Unpublished Manuscript.
34. Pitts, A. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science* 10 (2000), pp. 1–39.
35. Plotkin, G. and Power, J. Notions of computation determine monads. Proc., Foundations of Software Science and Computation Structure, pp. 342–356, 2002.
36. Smyth, M. B. and Plotkin, G. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing* 11(4) (1982), pp. 761–783.
37. Takano, A. and Meijer, E. Shortcut deforestation in calculational form. Proc., Functional Programming Languages and Computer Architecture, pp. 306–313, 1995.
38. Uustalu, T. and Vene, V. Mendler-style inductive types. *Nordic Journal of Computing* 6(3) (1999), pp. 343–361.

39. Wehr, M. Non-uniform recursion: The solution (minimal sorting for fold). Available at `www.citeseer.ist.psu.edu/wehr00nonuniform.html`.