Chapter 1

Generalizing the AUGMENT Combinator

Neil Ghani[1], Tarmo Uustalu[2], and Varmo Vene[3]

Abstract: The usual initial algebra semantics of inductive types provides a clear and uniform explanation for the FOLD combinator. In an APLAS 2004 paper [1], we described an alternative equivalent semantics of inductive types as limits of algebra structure forgetting functors. This gave us an elegant universal property based account of the BUILD and AUGMENT combinators, which form the core of the shortcut deforestation program transformation method by Gill et al. [2, 3]. Here we present further evidence for the flexibility of our approach by showing that a useful AUGMENT-like combinator is definable for a far wider class of parameterized inductive types than free monads, namely for all monads arising from a parameterized monad via an initial algebra construction.

1.1 INTRODUCTION

The standard approach to programming with inductive types is based on the IN/FOLD (constructors/structural recursion) syntax derived directly from their initial algebra semantics. This encourages a modular style of programming where functions are composed of simpler functions using intermediate data structures as a glue. Apart from the obvious advantages for the programmer, the use of intermediate data structures, which have to be constructed, traversed and discarded, can make programs inefficient both in terms of time and space. This calls for methods for transforming a modular program into an equivalent but more efficient program where these intermediate structures have been eliminated.

[1]Dept. of Math. and Comp. Sci., University of Leicester, University Road, Leicester, LE1 7RH, UK; Email: N.Ghani@mcs.le.ac.uk

[2]Institute of Cybernetics, Tallinn Univ. of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia; Email: tarmo@cs.ioc.ee

[3]Dept. of Computer Science, University of Tartu, Liivi 2, EE-50409 Tartu, Estonia; Email: varmo@cs.ut.ee

One of the most successful methods for eliminating intermediate data structures has been shortcut deforestation, originally introduced by Gill et al. [2, 3] for lists and generalized for other inductive types by Launchbury and Sheard and Takano and Meyer [8, 9]. This is based on the interaction of FOLD with a novel constructor combinator BUILD, capturing uniform production of data structures in a fashion matching the consumption pattern of FOLD. Deforestation happens by repeated application of a simple transformation rule of FOLD/BUILD fusion. Gill [3] also introduced an enhanced AUGMENT combinator for lists and Johann [5] generalized it for arbitrary inductive types with distinguished non-recursive constructors (so that list types and their nil constructors became instances).

Despite the success of the shortcut deforestation method, its theoretical explanations (see, e.g., [9]) have not been fully satisfactory, typically relying on informal uses of Wadler's [12] free theorems, without discussing the exact conditions under which these are available. It is only recently that Johann [6, 5] has proved the correctness of FOLD/BUILD and FOLD/AUGMENT fusion via parametricity of contextual equivalence. In an APLAS 2004 paper [1], we approached the issue from a different perspective. We set out to give a language-independent axiomatic specification of BUILD and AUGMENT and derived it from an elementary alternative semantics of inductive types as limits of algebra structure forgetting functors. That semantics has BUILD rather than IN as the basic constructor of an inductive type and explains also the impredicative encoding of inductive types.

This short paper is a companion paper to [1]. We present further evidence for the merits of the transparency of our framework by deriving a strong generalization of the AUGMENT combinator. We observe that AUGMENT is a monadic operation and show that AUGMENT is definable not only for free monads—which is essentially what Johann's version [5] amounts to—, but far more generally for any monad arising from a parameterized monad via the initial algebras based construction of Uustalu [10]. The extension operation of such monads can be seen as a grafting operation. The FOLD/AUGMENT fusion rule makes it possible to deforest programs involving this form of grafting.

The paper is organized as follows. In Sections 2, 3 we give a very condensed motivation for BUILD and AUGMENT combinators and review the account of FOLD/BUILD and AUGMENT from [1]. In Section 4, which is the main section, we extend this account to the generalized version AUGMENT, to conclude in Section 5.

We use Haskell in our examples, because its syntax is familiar to many, but this is purely illustrative. We are by no means asserting that Haskell's polymorphism is fully compliant with the parametricity requirements imposed by our semantics.

1.2 SEMANTICS OF FOLD/BUILD

Traditionally, the inductive type determined by an endofunctor F on a semantic category $\mathcal{C}$ is modelled by a chosen initial F-algebra. Such a characterization via a universal property immediately suggests a syntax and a theory for programming with the type in question and for reasoning about it. The initial algebra semantics gives us the familiar IN/FOLD syntax and theory, see, e.g., [4]. Indeed, by introducing the notations $(\mu F, \mathsf{in}_F)$ for the initial F-algebra and $\mathsf{fold}_{F,C}\varphi$ for the unique map to an F-algebra (X, φ), we institute a type μF (the inductive type) equipped with a constructor in_F and a destructor fold_F (the corresponding structural recursor) with the typing rules

$$\mathsf{in}_F : F(\mu F) \to \mu F \quad \frac{(X, \varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}_{F,X}\varphi : \mu F \to X}$$

β-conversion rule

$$\frac{(X, \varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}_{F,X}\varphi \circ \mathsf{in}_F = \varphi \circ F\mathsf{fold}_{F,X}\varphi}$$

and η- and permutative conversion rules

$$\mathsf{fold}_{F,\mu F}\mathsf{in}_F = \mathsf{id}_{\mu F} \quad \frac{f : (X, \varphi) \to (Y, \psi) \in F\text{-}\mathbf{alg}}{f \circ \mathsf{fold}_{F,X}\varphi = \mathsf{fold}_{F,Y}\psi}$$

As a well-known example, lists over type A arise as the initial algebra of the functor $1 + A \times -$. The Haskell Prelude introduces the (parameterized) type of lists and equips it with IN/FOLD syntax as follows, the β-rule defining the computational behavior of the FOLD combinator.

```
data [a] = [] | a : [a]

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

In shortcut deforestation, however, one uses a different syntax: producers of lists must be presented as applications of a BUILD combinator

```
build :: (forall x. (a -> x -> x) -> x -> x) -> [a]
build theta = theta (:) []
```

The merit is that applications of FOLD can be cancelled by those of BUILD as described by the so-called FOLD/BUILD fusion law

```
foldr c n (build theta) == theta c n
```

In [1], we motivated this different syntax by showing that the inductive type given by F can alternatively be interpreted as a chosen limit of the functor $U_F : F\text{-}\mathbf{alg} \to C$. By definition, a *$U_F$-cone* is an object C in $\mathcal{C}$ and, for any F-algebra (X, φ), a map $\Theta_X\varphi : C \to X$ in $\mathcal{C}$, such that (*) for any F-algebra map $f : (X, \varphi) \to (Y, \psi)$, we have $f \circ \Theta_X\varphi = \Theta_Y\psi$. A *$U_F$-cone map* $h : (C, \Theta) \to (D, \Xi)$ is a map $h : C \to D$ in $\mathcal{C}$ such that, for any F-algebra (X, φ), we have $\Xi_X\varphi \circ h = \Theta_X\varphi$. A *$U_F$-limit* is a U_F-cone to which there is a unique map from any other U_F-cone. Writing $(\mu^* F, \mathsf{fold}^*_F)$ for the U_F-limit (in case it exists) and $\mathsf{build}^*_{F,C}\Theta$ for the unique map from a U_F-cone (C,Θ), we obtain a type $\mu^* F$ with a destructor fold^*_F and a constructor build^*_F governed by the typing rules

$$\frac{(X,\varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}^*_{F,X}\varphi : \mu^* F \to X} \qquad \frac{f : (X,\varphi) \to (Y,\psi) \in F\text{-}\mathbf{alg}}{f \circ \mathsf{fold}^*_{F,X}\varphi = \mathsf{fold}^*_{F,Y}\psi} \qquad \frac{(C,\Theta) \in U_F\text{-}\mathbf{cone}}{\mathsf{build}^*_{F,C}\Theta : C \to \mu^* F}$$

The corresponding β-conversion rule is

$$\frac{(C,\Theta) \in U_F\text{-}\mathbf{cone} \quad (X,\varphi) \in F\text{-}\mathbf{alg}}{\mathsf{fold}^*_{F,X}\varphi \circ \mathsf{build}^*_{F,C}\Theta = \Theta_X \varphi}$$

and the η- and permutative conversion rules are

$$\mathsf{id}_{\mu^* F} = \mathsf{build}^*_{F,\mu^* F}\mathsf{fold}^*_F \qquad \frac{h : (C,\Theta) \to (D,\Xi) \in U_F\text{-}\mathbf{cone}}{\mathsf{build}^*_{F,D}\Xi \circ h = \mathsf{build}^*_{F,C}\Theta}$$

Inspecting these rules, one sees immediately that fold^*_F and build^*_F type exactly as FOLD and BUILD should, although with the reservation that the argument of build^*_F is required to meet the coherence condition (*). Moreover, the β-conversion rule is axiomatic FOLD/BUILD fusion. This makes the limit of forgetful functor semantics an appealing candidate justification for the FOLD/BUILD syntax (with BUILD rather than IN as the primitive constructor) and for the fusion rule. Remarkably, the condition (*) crisply explicates the kind of parametricity of polymorphism that is required for everything to work.

That the alternative semantics is adequate by being equivalent to the standard one is demonstrated by the following proposition.

Proposition 1.1. *Let $\mathcal{C}$ be a category and $F : \mathcal{C} \to \mathcal{C}$ be a functor. (a) If there is an initial F-algebra $(\mu F, \mathsf{in}_F)$, then μF is the vertex of a U_F-limit. (b) If there is a U_F-limit $(\mu^* F, \mathsf{fold}^*_F)$, then $\mu^* F$ is the carrier of an initial F-algebra.*

Proof (constructions). (a) Set, for any U_F-cone (C,Θ), $\mathsf{build}_{F,C}\Theta =_{\mathrm{df}} \Theta_{\mu F}\mathsf{in}_F$; $(\mu F, \mathsf{fold}_F)$ is a U_F-cone and $\mathsf{build}_{F,C}\Theta$ a unique map from any U_F-cone.

(b) Set, for any F-algebra (X,φ), $\mathsf{infold}^*_{F,X}\varphi =_{\mathrm{df}} \varphi \circ F\mathsf{fold}^*_{F,X}\varphi$; set $\mathsf{in}^*_F =_{\mathrm{df}} \mathsf{build}^*_{F,F(\mu^* F)}\mathsf{infold}^*_F$; $(\mu^* F, \mathsf{in}^*_F)$ is an F-algebra and $\mathsf{fold}^*_{F,X}\varphi$ is a unique map to any F-algebra (X,φ). $\square$

The condition (*) turns out to be equivalent to a strong dinaturality condition.

Definition 1.1. *Let $H, K : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathcal{D}$ be functors. A* strongly dinatural transformation *$\Theta : H \to K$ is a family of maps $\Theta_X : H(X,X) \to K(X,X)$ in $\mathcal{D}$ for all objects X in $\mathcal{C}$ such that, for every map $f : X \to Y$ in $\mathcal{C}$, object W and maps $p_0 : W \to H(X,X)$, $p_1 : W \to H(Y,Y)$ in $\mathcal{D}$, if the square in the following diagram commutes, then so does the hexagon:*

$$\begin{tikzcd}
 & H(X,X) \arrow[rrr, "\Theta_X"] \arrow[dr, "H(X,f)"'] & & & K(X,X) \arrow[dr, "K(X,f)"] & \\
W \arrow[ur, "p_0"] \arrow[dr, "p_1"'] & & H(X,Y) & \Rightarrow & & K(X,Y) \\
 & H(Y,Y) \arrow[rrr, "\Theta_Y"'] \arrow[ur, "H(f,Y)"] & & & K(Y,Y) \arrow[ur, "K(f,Y)"'] &
\end{tikzcd}$$

Proposition 1.2. *Let $\mathcal{C}$ be a locally small category and $F : \mathcal{C} \to \mathcal{C}$ a functor. A U_F-cone structure with vertex C is the same thing as a strong dinatural transformation from* $\mathsf{Hom}(F\ -,-)$ *to* $\mathsf{Hom}(C,-)$.

The relevance of this proposition for programming languages is that the coherence condition (*) is for free whenever strong dinaturality is.

1.3 THE AUGMENT OF FREE MONADS

BUILD presentations of producers do not always suffice for shortcut deforestation. A prototypical example for lists is the append function

```
append as bs = foldr (:) bs as
```

The definition as a BUILD

```
append as bs = build (\ c n -> foldr c (foldr c n bs) as)
```

introduces an unnatural traversal of the list appended and this traversal is not necessarily removed in shortcut deforestation.

The solution proposed by Gill [3] consists in deploying an enhanced version of BUILD, called AUGMENT. For lists, the Haskell version of this combinator is the following:

```
augment :: (forall x. (a -> x -> x) -> x -> x) -> [a] -> [a]
augment theta bs = theta (:) bs
```

An AUGMENT represents nothing else than a BUILD followed by an append:

```
augment theta bs == append (build theta) bs
```

And conversely, a BUILD is just an AUGMENT applied to nil:

```
build theta == augment theta []
```

But the reason to introduce AUGMENT is that it obeys a special law of fusion with FOLD:

```
foldr c n (augment theta bs) == theta c (foldr c n bs)
```

This law is similar by its spirit to the FOLD/BUILD law. But differently from a BUILD, an AUGMENT does not cancel a FOLD completely: the reduct contains a residual FOLD.

The definition of the append function as an AUGMENT is

```
append as bs = augment (\ c n -> foldr c n as) bs
```

A similar AUGMENT combinator can be easily defined for arbitrary inductive types with non-recursive constructors to obtain a version of shortcut deforestation tailored specifically for producer functions that perform grafting, see Johann [5]. In [1], we justified a slightly more general version in the style our FOLD/BUILD account and, in line with our general methodology, derived it from a unique existence situation. In that version, AUGMENT combinators are associated to free monads, which are families of inductive types rather than single inductive types.

Recall that a *monad* (in *extension form*) on a category $\mathcal{C}$ is an object mapping $T : |\mathcal{C}| \to |\mathcal{C}|$ together with, for any object A, a map $\eta_A : A \to TA$ (*unit*), and for any map $f : A \to TB$, a map $f^\star : TA \to TB$ (*extension operation*) such that (i) for any map $f : A \to TB$, $f^\star \circ \eta_A = f$, (ii) for any object A, ${\eta_A}^\star = \mathsf{id}_{TA}$, and (iii) for any maps $f : A \to TB$, $g : B \to TC$, $(g^\star \circ f)^\star = g^\star \circ f^\star$.

Recall also that monads in Haskell are implemented via a type constructor class:

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

The Haskell jargon for the extension operation is ‘bind’.

Consider an endofunctor H on a category $\mathcal{C}$ and let $T' : \mathcal{C} \to [\mathcal{C}, \mathcal{C}]$ be the functor given by $T'AX =_{\mathrm{df}} A + HX$. Then, if an initial $T'A$-algebra (= a free H-algebra over A) exists for every object A of $\mathcal{C}$, we can get a functor $T : \mathcal{C} \to \mathcal{C}$ by defining $TA =_{\mathrm{df}} \mu(T'A)$. This models an inductive type parameterized in non-recursive constructors. Decompose each map $\mathsf{in}_{T'A}$ into two maps $\eta_A : A \to TA$ and $\tau_A : H(TA) \to TA$ by setting

$$\begin{aligned}
\eta_A &=_{\mathrm{df}} \mathsf{in}_{T'A} \circ \mathsf{inl}_{A,H(TA)} \\
\tau_A &=_{\mathrm{df}} \mathsf{in}_{T'A} \circ \mathsf{inr}_{A,H(TA)}
\end{aligned}$$

Define, finally, for any map $f : A \to TB$, a map $f^\star : TA \to TB$ by

$$f^\star =_{\mathrm{df}} \mathsf{fold}_{T'A,TB}[f, \tau_B]$$

Conceptually, η packages the non-recursive constructors of the parameterized type, τ packages the recursive constructors, and $()^\star$ is substitution for non-recursive constructors.

It is standard knowledge the data $(T, \eta, ()^\star)$ so constructed constitute a monad which, more specifically, is the free monad generated by the functor H.

The following proposition supplies the parameterized inductive type T with an AUGMENT combinator.

Proposition 1.3. *Let $\mathcal{C}$ be a category, $H : \mathcal{C} \to \mathcal{C}$ a functor such that initial algebras of all functors $T'A =_{\mathrm{df}} A + H-$ exist. Let T, η, τ, $()^\star$ be defined as above. Then, for any map $f : A \to TB$ and $U_{T'A}$-cone (C, Θ) there exists a unique map $h : C \to TB$ such that, for any $T'B$-algebra $(X, [\varphi_0, \varphi_1])$, it holds that*

$$\mathsf{fold}_{T'B,X}[\varphi_0, \varphi_1] \circ h = \Theta_X([\mathsf{fold}_{T'B,X}[\varphi_0, \varphi_1] \circ f, \varphi_1])$$

We denote the unique map by $\mathsf{augment}_{T',C}(\Theta, f)$.

On the level of syntax, the proposition justifies the introduction of a constructor combinator $\mathsf{augment}_{T'}$ with a typing rule

$$\frac{(C,\Theta) \in U_{T'A}\textbf{-cone} \quad f : A \to TB}{\mathsf{augment}_{T',C}(\Theta, f) : C \to TB}$$

and β-conversion rule

$$\frac{(C,\Theta) \in U_{T'A}\textbf{-cone} \quad f : A \to TB \quad (X, [\varphi_0, \varphi_1]) \in T'B\textbf{-alg}}{\mathsf{fold}_{T'B,X}[\varphi_0, \varphi_1] \circ \mathsf{augment}_{T',C}(\Theta, f) = \Theta_X[\mathsf{fold}_{T'B,X}[\varphi_0, \varphi_1] \circ f, \varphi_1]}$$

which is fusion of FOLD and AUGMENT. One also gets a number of conversion rules pertaining to the monad structure on T, some of which we will list in the next section.

The combinator of Johann [5] differs from our version by having the type of non-recursive constructors fixed (so that $B = A$), thus confining the action of the combinator to a single inductive type rather than a family one would obtain by letting the type of non-recursive constructors to vary. (The reason must be that, in the prototypical case of lists, the type of non-recursive constructors is constantly 1; normally, one does not consider the possibility of supporting multiple nil's drawn from a parameter type.) Unfortunately, this restriction hides the important role that the monad structure on T plays in the construction. This role is central for our generalization in the next section.

A typical example of a free monad is the parameterized type of binary leaf labelled trees. Binary leaf labelled trees with leaf label type A are the initial algebra of the functor $A + - \times -$. They are Haskell-implementable as follows:

```
data BLTree a = Leaf a | Bin (BLTree a) (BLTree a)

foldB :: (a -> x) -> (x -> x -> x) -> BLTree a -> x
foldB l b (Leaf a) = l a
foldB l b (Bin as0 as1) = b (foldB l b as0) (foldB l b as1)

instance Monad BLTree where
  return a = Leaf a
  as >>= f = foldB f Bin as
```

The BUILD and AUGMENT combinators are implementable as follows:

```
buildB :: (forall x. (a -> x) -> (x -> x -> x) -> x)
                                                -> BLTree a
buildB theta = theta Leaf Bin

augmentB :: (forall x. (a -> x) -> (x -> x -> x) -> x)
                            -> (a -> BLTree b) -> BLTree b
augmentB theta f = theta f Bin
```

The shortcut deforestation laws say that

```
foldB l b (buildB theta) == theta l b
foldB l b (augmentB theta f) == theta (foldB l b . f) b
```

It is also possible to give a generic implementation of the AUGMENT combinator of free monads that caters for all possible shapes of leaf-labelled trees at one go. The code is nearly identical to what we have just seen.

We observe that AUGMENT combinators are about uniform production of inductive data in combination with grafting. Grafting presupposes monadic structure. We have seen that an AUGMENT combinator can be defined for any free monad. But there exist parameterized inductive types carrying a non-free monad structure! Could it be that an AUGMENT-like combinator subject to a fusion law is definable for some of them too? We will answer this question affirmatively in the next section.

1.4 A GENERALIZED AUGMENT COMBINATOR

The AUGMENT combinator for the monad of the free algebras of a functor (or, in other words, the free monad) is a meaningful combinator obeying a useful FOLD/AUGMENT fusion rule, but it is not as general as possible. As we will now show, a similar combinator is possible for any monad obtained from a parameterized monad via initial algebras as described by Uustalu [10]. This generalization is, to our knowledge, entirely new.

We must start by reviewing the construction in [10].

A *parameterized monad* is an object mapping $T' : |\mathcal{C}| \to |[\mathcal{C}, \mathcal{C}]|$ together with, for any objects A, X, a map $\eta'_{A,X} : A \to T'AX$ and, for any map $f : A \to T'BX$, a map $f^\circ : T'AX \to T'BX$ such that (i') for any map $f : A \to T'BX$, $f^\circ \circ \eta'_{A,X} = f$, (ii') for any objects A, X, $(\eta'_{A,X})^\circ = \mathsf{id}_{T'AX}$, (iii') for any maps $f : A \to T'BX$, $g : B \to T'CX$, $(g^\circ \circ f)^\circ = g^\circ \circ f^\circ$, (iv') for any object A and map $\xi : X \to Y$, $T'A\xi \circ \eta'_{A,X} = \eta'_{A,Y}$, and (v') for any maps $f : A \to T'BX$, $\xi : X \to Y$, $T'B\xi \circ f^\circ = (T'B\xi \circ f)^\circ \circ T'A\xi$. It is easy to verify that a parameterized monad on $\mathcal{C}$ is essentially just a functor from $\mathcal{C}$ to the category $\mathbf{Monad}(\mathcal{C})$ of monads on $\mathcal{C}$, but we adopt the above presentation as this will be more convenient.

Now if $(T', \eta', ()^\circ)$ is a parameterized monad on $\mathcal{C}$, we can consider the initial algebras $(TA, \alpha_A) =_{\mathrm{df}} (\mu(T'A), \mathsf{in}_{T'A})$, provided all functors $T'A$ have an initial algebra. Define, for any object A, a map $\eta_A : A \to TA$ by

$$\eta_A =_{\mathrm{df}} \alpha_A \circ \eta'_{A,TA}$$

and, for any map $f : A \to TB$, a map $f^\star : TA \to TB$ by

$$f^\star =_{\mathrm{df}} \mathsf{fold}_{T'A,TB}\,(\alpha_B \circ (\alpha_B^{-1} \circ f)^\circ)$$

By a proposition in [10], $(T, \eta, ()^\star)$ is a monad. Its extension operation $()^\star$ is, in a meaningful but very liberal sense, a grafting operation.

Below are some examples of monads arising from parameterized monads in the manner just described:

- The monad of the free algebras of a given functor $H : \mathcal{C} \to \mathcal{C}$ (the free monad generated by H). This is the example we already saw: the inducing parameterized monad T' is defined by $T'AX =_{\mathrm{df}} A + HX$ (so that, for any type X, $T'{-}X$ is the exceptions monad with exceptions type HX).
- The monad of finitely branching node labelled trees (rose trees). The inducing parameterized monad T' is defined by $T'AX =_{\mathrm{df}} A \times \mathsf{List}X$. Notice that in this type of trees values are stored at inner nodes, not at leaves, so substitution must be a bit unusual. The substitution function $f^\star : TA \to TB$ that corresponds to a given substitution rule $f : A \to TB$ is specified by the equation

  $$f^\star(x[t_1,\ldots t_n]) =_{\mathrm{df}} y[s_1,\ldots,s_m, f^\star(t_1),\ldots,f^\star(t_n)] \text{ where } f(x) =_{\mathrm{df}} y[s_1,\ldots,s_m]$$

- The monad of inductive hyperfunctions [7] with a fixed domain. The inducing parameterized monad T' is defined by $T'AX =_{\mathrm{df}} (X \Rightarrow E) \Rightarrow A$ where E is a fixed object of $\mathcal{C}$.
- A further interesting monad is obtained from the parameterized monad T' defined by $T'AX =_{\mathrm{df}} \mathsf{List}(A + X)$. The bind operation of this monad is defined in terms of the bind operation of the lists monad.

Now, as promised, we can introduce an AUGMENT combinator can for all monads that arise from a parameterized monad in the fashion described above. This results again from a uniquely satisfied property.

Proposition 1.4. *Let $\mathcal{C}$ be a category and $(T', \eta', ()^\circ)$ a parameterized monad on it such that initial algebras of all functors $T'A$ exist. Let T η, τ, $()^\star$ be defined as above. Then, for any map $f : A \to TB$ and $U_{T'A}$-cone (C,Θ) there exists a unique map $h : C \to TB$ such that, for any $T'B$-algebra (X,φ), it holds that*

$$\mathsf{fold}_{T'B,X}\,\varphi \circ h = \Theta_X\,(\varphi \circ (T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ \alpha_B^{-1} \circ f)^\circ)$$

As earlier, we denote the unique map by $\mathsf{augment}_{T',C}(\Theta, f)$.

Proof. We prove that

$$\mathsf{augment}_{T',C}(\Theta, f) = \Theta_{TB}\,(\alpha_B \circ (\alpha_B^{-1} \circ f)^\circ)$$

For any $T'B$-algebra (X,φ), making use of the axiom (v') of parameterized monads, we have

$$\begin{array}{cl} & \mathsf{fold}_{T'B,X}\,\varphi \circ \alpha_B \circ (\alpha_B^{-1} \circ f)^\circ \\ = & \varphi \circ T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ (\alpha_B^{-1} \circ f)^\circ \\ = & \varphi \circ (T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ \alpha_B^{-1} \circ f)^\circ \circ T'A\,(\mathsf{fold}_{T'B,X}\,\varphi) \end{array}$$

Hence, $\mathsf{fold}_{T'B,X}\,\varphi$ is a $T'A$-algebra map from $(TB, \alpha_B \circ (\alpha_B^{-1} \circ f)^\circ)$ to $(X, \varphi \circ (T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ \alpha_B^{-1} \circ f)^\circ)$. Hence by (C,Θ) being a $U_{T'A}$-cone

$$\mathsf{fold}_{T'B,X}\,\varphi \circ \Theta_{TB}\,(\alpha_B \circ (\alpha_B^{-1} \circ f)^\circ) = \Theta_X\,(\varphi \circ (T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ \alpha_B^{-1} \circ f)^\circ)$$

as needed.

Assume now we have a map $h : C \to TB$ such that, for any $T'B$-algebra (X, φ),

$$\mathsf{fold}_{T'B,X}\,\varphi \circ h = \Theta_X\,(\varphi \circ (T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ \alpha_B^{-1} \circ f)^\circ)$$

Then

$$\begin{array}{rcl} h & = & \mathsf{fold}_{T'B,TB}\,\alpha_B \circ h \\ & = & \Theta_{TB}\,(\alpha_B \circ (T'B\,(\mathsf{fold}_{T'B,TB}\,\alpha_B) \circ \alpha_B^{-1} \circ f)^\circ) \\ & = & \Theta_{TB}\,(\alpha_B \circ (\alpha_B^{-1} \circ f)^\circ) \end{array}$$

which completes the proof of uniqueness. $\square$

This unique existence motivates a combinator $\mathsf{augment}_{T'}$ obeying the following rules:

- typing rule:
$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB}{\mathsf{augment}_{T',C}(\Theta, f) : C \to TB}$$

- β-conversion rule (= FOLD/AUGMENT fusion law):
$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB \quad (X, \varphi) \in T'B\text{-}\mathbf{alg}}{\mathsf{fold}_{T'B,X}\varphi \circ \mathsf{augment}_{T',C}(\Theta, f) = \Theta_X\,(\varphi \circ (T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ \alpha_B^{-1} \circ f)^\circ)}$$

- conversion rules relating to the monad structure:
$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB}{\mathsf{augment}_{T',C}(\Theta, f) = f^\star \circ \mathsf{build}_{T'A,C}\Theta} \qquad \frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone}}{\mathsf{build}_{T'A,C}\Theta = \mathsf{augment}_{T',C}(\Theta, \eta_A)}$$
$$\frac{(C, \Theta) \in U_{T'A}\text{-}\mathbf{cone} \quad f : A \to TB \quad g : B \to TC}{g^\star \circ \mathsf{augment}_{T',C}(\Theta, f) = \mathsf{augment}_{T',C}(\Theta, g^\star \circ f)}$$

Notice that the first two rules in the last group state that the AUGMENT and BUILD combinators are interdefinable using the unit and bind of the monad.

Instantiating the first of them with $(C, \Theta) = (TA, \mathsf{fold}_{T'A})$, we obtain that bind is AUGMENT applied to FOLD:

$$\frac{f : A \to TB}{\mathsf{augment}_{T',TA}(\mathsf{fold}_{T'A}, f) = f^\star}$$

Combining this with the FOLD/AUGMENT fusion rule, we get the following rule for fusing FOLD with bind:

$$\frac{f : A \to TB \quad (X, \varphi) \in T'B\text{-}\mathbf{alg}}{\mathsf{fold}_{T'B,X}\varphi \circ f^\star = \mathsf{fold}_{T'A,X}(\varphi \circ (T'B\,(\mathsf{fold}_{T'B,X}\,\varphi) \circ \alpha_B^{-1} \circ f)^\circ)}$$

Here is an implementation of finitely branching node labelled trees (rose trees) in Haskell, including the monad structure.

```
data Tree a = Node a [Tree a]

foldT :: (a -> [x] -> x) -> Tree a -> x
foldT phi (Node a tas) = phi a (map (foldT phi) tas)

instance Monad Tree where
  return a = Node a []
  ta >>= f = foldT (\ a tbs' -> let Node b tbs = f a
                                 in Node b (tbs ++ tbs')) ta
```

The BUILD and AUGMENT combinators are implementable as follows:

```
buildT :: (forall x . (a -> [x] -> x) -> x) -> Tree a
buildT theta = theta Node

augmentT :: (forall x . (a -> [x] -> x) -> x)
                                 -> (a -> Tree b) -> Tree b
augmentT theta f = theta (\ a tbs' -> let Node b tbs = f a
                                       in Node b (tbs ++ tbs'))
```

FOLD/BUILD fusion and FOLD/AUGMENT fusion say that

```
foldT phi (buildT theta) == theta phi
foldT phi (augmentT theta f) == theta (\ a xs ->
                        let Node b tbs = f a
                        in phi b (map (foldT phi) tbs ++ xs))
```

For inductive hyperfunction spaces, we get the BUILD and AUGMENT combinators as follows:

```
data Hyper e a = H { unH :: (Hyper e a -> e) -> a }

foldH :: (((x -> e) -> a) -> x) -> Hyper e a -> x
foldH phi (H h) = phi (\ g -> h (g . foldH phi))

instance Monad (Hyper e) where
  return a = H (const a)
  ha >>= f = foldH (\ g -> H (\ k -> unH (f (g k)) k)) ha

buildH :: (forall x . (((x -> e) -> a) -> x) -> x)
                                                -> Hyper e a
buildH theta = theta H

augmentH :: (forall x . (((x -> e) -> a) -> x) -> x)
                          -> (a -> Hyper e b) -> Hyper e b
augmentH theta f = theta (\ g -> H (\ k -> unH (f (g k)) k))
```

The corresponding fusion laws are

```
foldH phi (buildH theta) == theta phi
foldH phi (augmentH theta f) ==
    theta (\ g -> phi (\ k -> unH (f (g k)) (k . foldH phi)))
```

Finally, the general construction of monads from parameterized monads via initial algebras may also be implemented generically:

```
class Monad' m where
  return' :: a -> m a x
  (>>|) :: m a x -> (a -> m b x) -> m b x
  fmap' :: (x -> y) -> (m a x -> m a y)

data Mu m a = In {unIn :: m a (Mu m a) }

fold :: Monad' m  => (m a x -> x) -> Mu m a -> x
fold phi (In tas) = phi (fmap' (fold phi) tas)

instance Monad' m => Monad (Mu m) where
  return a = In (return' a)
  ta >>= f = fold (In . (>>| unIn . f)) ta

build :: (forall x. (m a x -> x) -> x) -> Mu m a
build theta = theta In

augment :: Monad' m => (forall x. (m a x -> x) -> x)
                                  -> (a -> Mu m b) -> Mu m b
augment theta f = theta (In . (>>| unIn . f))
```

The corresponding FOLD/BUILD and FOLD/AUGMENT fusion rules are as follows:

```
fold phi (build theta) == theta phi
fold phi (augment theta f) ==
    theta (phi . (>>| fmap' (fold phi) . unIn . f))
```

1.5 CONCLUSION AND FUTURE WORK

We have demonstrated that the semantics of inductive types in terms of limits of algebra structure forgetting functors does not only explain the typing (including, notably, the coherence condition on the argument) of the BUILD combinator and the FOLD/BUILD fusion rule, but exhibits a flexibility thanks to which it is not hard to generalize Gill's originally rather specific AUGMENT combinator to a considerably wider class of parameterized inductive types. This reveals, in particular, that the original FOLD/AUGMENT fusion for lists, which at first sight looks like an ad hoc rule, is really a special case of a very systematic rule, a tip of a small iceberg.

As future work we plan to achieve a similar account for the vanish combinators of Voigtländer [11]. We also plan to study the relation of strong dinaturality and parametricity.

Acknowledgments The authors were partially supported by the Royal Society ESEP programme within joint research project No. 15642. The second and third author were also partially supported by the Estonian Science Foundation under grant No. 5567.

REFERENCES

[1] N. Ghani, T. Uustalu, and V. Vene. Build, augment and destroy, universally. In W.-N. Chin, *Proc. of 2nd Asian Symp. on Programming Languages and Systems, APLAS'04*, v. 3302 of *Lect. Notes in Comput. Sci.*, pp. 327–347. Springer-Verlag, Berlin, 2004.

[2] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Conf. Record of 6th ACM SIGPLAN-SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93*, pp. 223–232. ACM Press, 1993.

[3] A. J. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Univ. of Glasgow, 1996.

[4] T. Hagino. A typed lambda calculus with categorical type constructors. In D. H. Pitt, A. Poigné, and D. E. Rydeheard, eds., *Proc. of 2nd Int. Conf. on Category Theory and Computer Science, CTCS'87*, v. 283 of *Lecture Notes in Computer Science*, pp. 140–157. Springer-Verlag, 1987.

[5] P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation*, 15(4):273–300, 2002.

[6] P. Johann. Short-cut fusion is correct. *J. of Functional Programming*, 13(4):797–814, 2003.

[7] S. Krstić, J. Launchbury, and D. Pavlović. Categories of processes enriched in final coalgebras. In F. Honsell and M. Miculan, eds., *Proc. of 4th Int. Conf. on Found. of Software Science and Computation Structures, FoSSaCS'01*, v. 2030 of *Lect. Notes in Comput. Sci.*, pp. 303–317. Springer-Verlag, 2001.

[8] J. Launchbury and T. Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN-SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pp. 314–323. ACM Press, 1995.

[9] A. Takano and E. Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th of ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pp. 306–313. ACM Press, 1995.

[10] T. Uustalu. Generalizing substitution. *Theoretical Informatics and Applications*, 37(4):315-336, 2003.

[11] J. Voigtländer. Concatenate, reverse and map vanish for free. In *Proc. of 7th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'02*, v. 37(9) of *SIGPLAN Notices*, pp. 14–25. ACM Press, 2002.

[12] P. Wadler. Theorems for free! In *Proc. of 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89*, pp. 347–359. ACM Press, 1989.