Algebras for Parameterised Monads

Robert Atkey

LFCS, School of Informatics, University of Edinburgh, UK
bob.atkey@ed.ac.uk

Abstract. Parameterised monads have the same relationship to adjunctions with parameters as monads do to adjunctions. In this paper, we investigate algebras for parameterised monads. We identify the Eilenberg-Moore category of algebras for parameterised monads and prove a generalisation of Beck's theorem characterising this category. We demonstrate an application of this theory to the semantics of type and effect systems.

1 Introduction

Monads [7] have a well-known relationship with algebraic theories [10], and have also been used by Moggi [8] to structure denotational semantics for programming languages with effects. Plotkin and Power [9] have used the connection to algebraic theories to generate many of the monads originally identified by Moggi as useful for modelling effectful programming languages. The operations of the algebraic theories are in direct correspondence with the effectful operations of the programming language.

In previous work [1] we have argued that a generalisation of monads, *parameterised monads*, are a useful notion for interpreting programming languages that record information about the effects performed in the typing information [4]. The generalisation is to consider unit and multiplication operations for functors $T : S^{\mathsf{op}} \times S \times C \to C$ for some parameterising category S and some base category C. The two S parameters are used to indicate information about the start and end states of a given computation; thus a morphism $x \to T(s_1, s_2, y)$ represents a computation that, given input of type x, starts in a state described by s_1 and ends in a state described by s_2, with a value of type y.

In this paper, we investigate the connection between parameterised monads and a notion of algebraic theories with parameters. In our previous work [1], we noted that parameterised monads arise from adjunctions with parameters—pairs of functors $F : S \times C \to A$ and $G : S^{\mathsf{op}} \times A \to C$ that, roughly, are adjoint for every $s \in S$. Here, we extend this relationship to show that there are natural notions of Kleisli category and Eilenberg-Moore category for parameterised monads, and that they are respectively initial and final in the category of adjunctions with parameters giving a particular parameterised monad.

We then go on to consider the appropriate notion of algebraic theory for parameterised monads. This turns out to require operations that have an arity, as with normal algebraic theories, and also domain and codomain sorts that are objects of S. We write such operations as $\sigma : s_2 \leftarrow s_1^X$, where X is the arity. These

theories are similar to multi-sorted algebraic theories, except that all argument positions in the operation have the same sort. Multi-sorted algebraic theories can be represented as normal monads on presheaf categories over the set of sorts. Our restriction of multi-sorted algebraic theories allows us to expose the sorts in the parameters of the parameterised monad itself.

Algebras for our notion of algebraic theories with parameters are given by functors $a : S^{\mathsf{op}} \to C$, with morphisms $\sigma_a : a(s_1)^X \to a(s_2)$ for operations $\sigma : s_2 \leftarrow s_1^X$.

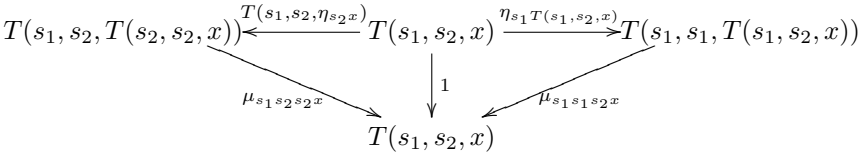
A central result in relating algebraic theories and monads is Beck's theorem [2, 7], which characterises the Eilenberg-Moore category of T-algebras for a given monad T in terms of the preservation and creation of coequalisers. In this paper, we prove a generalisation of Beck's theorem for parameterised monads.

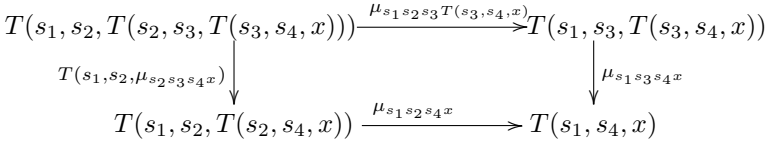
As an application of the theory we have developed, we use algebraic theories with parameters to give a semantics to a toy programming language that records information about effects performed.

Overview In the next section we recall the definition of parameterised monad, and develop the relationship between adjunctions with parameters and parameterised monads, in particular describing the category of algebras for a given monad. In Section 3, we define a notion of algebraic theory with parameters, where the free algebras give rise to parameterised monads. To show that the category of such algebras is isomorphic to the category of algebras for the derived monad, we prove a generalisation of Beck's theorem in Section 4. In Section 5, we consider the case when the parameterising category has structure given by functors and natural transformations. We then apply the results developed to the semantics of type and effect systems in Section 6.

2 Parameterised Monads

Definition Assume a small category S. An S-parameterised monad on another category C is a 3-tuple $\langle T, \eta, \mu \rangle$, where T is a functor $S^{\mathsf{op}} \times S \times C \to C$, the unit η is a family of arrows $\eta_{s,x} : x \to T(s, s, x)$, natural in x and dinatural in s and the multiplication μ is a family of arrows $\mu_{s_1 s_2 s_3 x} : T(s_1, s_2, T(s_2, s_3, x)) \to T(s_1, s_3, x)$, natural in s_1, s_3 and x and dinatural in s_2. These must make the following diagrams commute:





Example 1. Every non-parameterised monad is a parameterised monad for any category of parameters. Set $T(s_1, s_2, x) = Tx$.

Example 2. Our main motivating example for introducing the concept of parameterised monad is for modelling global state in a programming language, where the type of the state may change over time. Take S to be a category of state "types", with a terminal object representing the empty state. Assume that C is cartesian closed, and that there is a functor $\widehat{\cdot} : S \to C$, preserving the terminal object. Take $T(s_1, s_2, x) = (x \times \widehat{s_2})^{\widehat{s_1}}$, with the evident unit and multiplication.

Example 3. It is well-known that (in Set) every monoid M gives a monad $Tx = M \times x$, where the unit of the monad is given using the unit of the monoid, and likewise for multiplication. Analogously, every (small) category S_0 gives an S-parameterised monad, where S is a subcategory of S_0. Set $T(s_1, s_2, x) = S_0(s_1, s_2) \times x$. Monad unit is given using the identities of S_0, and monad multiplication is given using composition.

Adjunctions with Parameters For categories C and A, an S-parameterised adjunction $\langle F, G, \eta, \epsilon \rangle : C \rightharpoonup A$ consists of functors $F : S \times C \to A$ and $G : S^{\mathsf{op}} \times A \to C$, the unit $\eta_{s,x} : x \to G(s, F(s, x))$, natural in x and dinatural in s, and the counit $\epsilon_{s,a} : F(s, G(s, a)) \to a$, natural in a and dinatural in s, satisfying the following triangular laws:

$$\begin{array}{ccc} G(s,a) & \xrightarrow{\eta_{s,G(s,a)}} & G(s, F(s, G(s, a))) \\ & \searrow^{1} & \downarrow{\scriptstyle G(s,\epsilon_{s,a})} \\ & & G(s,a) \end{array} \qquad \begin{array}{ccc} F(s,x) & \xrightarrow{F(s,\eta_{s,x})} & F(s, G(s, F(s, x))) \\ & \searrow^{1} & \downarrow{\scriptstyle \epsilon_{s,F(s,x)}} \\ & & F(s,x) \end{array}$$

By Theorem §IV.7.3 in Mac Lane [7], if we have a functor $F : S \times C \to A$ such that for every object s, $F(s, -)$ has a right adjoint $G_s : A \to C$, then there is a unique way to make a bifunctor $G : S^{\mathsf{op}} \times A \to C$ such that $G(s, -) = G_s$ and the pair form a parameterised adjunction.

Parameterised adjunctions have the same relationship to parameterised monads as adjunctions have to monads. First, we go from parameterised adjunctions to monads:

Proposition 4. *Given an S-parameterised adjunction $\langle F, G, \eta, \epsilon \rangle : C \rightharpoonup A$, there is an S-parameterised monad on C defined with functor $G(s_1, F(s_2, x))$, unit $\eta_{s,x}$ and multiplication $\mu_{s_1 s_2 s_3 x} = G(s_1, \epsilon_{s_2, F(s_3, x)})$.*

In the opposite direction, from monads to adjunctions, we have the same situation as for non-parameterised monads. There are two canonical adjunctions arising from a parameterised monad; the initial and terminal objects in the category of adjunctions that define the monad.

First, we define the category of adjunctions that we are interested in. Given an S-parameterised monad $\langle T, \eta, \mu \rangle$ on C, the category $\mathrm{PAdj}(T)$ has as objects S-parameterised adjunctions that give the monad T by the construction above;

and arrows $f : (\langle F, G, \eta, \epsilon\rangle : C \rightharpoonup A) \to (\langle F', G', \eta, \epsilon'\rangle : C \rightharpoonup A')$ are functors $f : A \to A'$ such that $G = Id \times f; G'$ and $F' = F; f$ and $\epsilon'_{s,fa} = f(\epsilon_{s,a})$. Note that, by the condition that all the adjunctions form the same parameterised monad, all objects of this category have the same unit. The definition of arrow is derived from the standard definition of a transformation of adjoints, extended to parameterised adjunctions and restricted to those that define the same monad.

Kleisli Category For an S-parameterised monad $\langle T, \eta, \mu\rangle$ on C, the Kleisli category C_T has pairs of objects of S and C as objects and arrows $f : (s_1, x) \to (s_2, y)$ are arrows $x \to T(s_1, s_2, y)$ in C. Identities are given by the unit of the monad, and composition is defined using the multiplication.

Proposition 5. *The functors*

$$F_T : S \times C \to C_T : (s, x) \mapsto (s, x) : (g, f) \mapsto \eta_{s_1,x}; T(s_1, g, f)$$
$$G_T : S^{\mathsf{op}} \times C_T \to C : (s_1, (s, x)) \mapsto T(s_1, s, x) : (g, f) \mapsto T(g, s_2, f); \mu_{s_1 s_2 s'_2 y}$$

form part of an S-parameterised adjunction between C and C_T. This adjunction is initial in PAdj(T).

Eilenberg-Moore Category of Algebras The second canonical parameterised adjunction that arises from a parameterised monad is the parameterised version of the Eilenberg-Moore category of algebras for the monad. For an S-parameterised monad $\langle T, \eta, \mu\rangle$ on C, a T-algebra is a pair $\langle x, h\rangle$ of a functor $x : S^{\mathsf{op}} \to C$ and an family $h_{s_1 s_2} : T(s_1, s_2, x(s_2)) \to x(s_1)$, natural in s_1 and dinatural in s_2. These must satisfy the diagrams:

$$\begin{array}{ccc} T(s_1, s_2, T(s_2, s_3, x(s_3))) & \xrightarrow{\mu_{s_1 s_2 s_3 x(s_3)}} & T(s_1, s_3, x(s_3)) \\ \downarrow{\scriptstyle T(s_1, s_2, h_{s_2 s_3})} & & \downarrow{\scriptstyle h_{s_1 s_3}} \\ T(s_1, s_2, x(s_2)) & \xrightarrow{h_{s_1 s_2}} & x(s_1) \end{array} \qquad \begin{array}{ccc} x(s) & \xrightarrow{\eta_{s,x(s)}} & T(s, s, x(s)) \\ & \searrow^{1} & \downarrow{\scriptstyle h_{s,s}} \\ & & x(s) \end{array}$$

A T-algebra map $f : \langle x, h\rangle \to \langle y, k\rangle$ is a natural transformation $f_s : x(s) \to y(s)$ such that

$$\begin{array}{ccc} T(s_1, s_2, x(s_2)) & \xrightarrow{T(s_1, s_2, f_{s_2})} & T(s_1, s_2, y(s_2)) \\ \downarrow{\scriptstyle h_{s_1 s_2}} & & \downarrow{\scriptstyle k_{s_1 s_2}} \\ x(s_1) & \xrightarrow{f_{s_1}} & y(s_1) \end{array}$$

commutes.

Clearly, T-algebras for a monad and their maps form a category, which we call C^T.

Proposition 6. *Given an S-parameterised monad $\langle T, \eta, \mu\rangle$, the functors*

$$F^T : S \times C \to C^T : (s, x) \mapsto \langle T(-, s, x), \mu_{s_1 s_2 x}\rangle : (g, f) \mapsto T(-, g, f)$$
$$G^T : S^{\mathsf{op}} \times C^T \to C : (s, \langle x, h\rangle) \mapsto x(s) : (g, f) \mapsto x(g); f_{s_1}$$

form a parameterised adjunction, with unit η and counit $\epsilon^T_{s,\langle x,h\rangle} = h_{-,s}$. This adjunction is terminal in PAdj(T).

3 Algebraic Theories with Parameters

Signatures, Terms and Theories An S-parameterised signature is a collection of operations $\sigma \in \Sigma$, where each operation has an associated arity $ar(\sigma)$, which is a countable set, a domain $dom(\sigma) \in S$ and a co-domain $cod(\sigma) \in S$. As a shorthand, we write an operation σ with $ar(\sigma) = X$, $dom(\sigma) = s_1$ and $cod(\sigma) = s_2$ as $\sigma : s_2 \leftarrow s_1^X$.

Given a signature Σ and a countable set V of variables, we can define the sets $T_\Sigma(s_1, s_2, V)$ of terms as the smallest sets closed under the following two rules:

$$\frac{v \in V \qquad f \in S(s_1, s_2)}{\mathsf{e}(f, v) \in T(s_1, s_2, V)}$$

$$\frac{\sigma : s_2 \leftarrow s_1^X \in \Sigma \qquad \{t_x \in T(s_2, s_2', V)\}_{x \in X} \qquad f \in S(s_1', s_1)}{\mathsf{op}(f, \sigma, \{t_x\}) \in T(s_1', s_2', V)}$$

By this definition, terms consist of trees of operations from the signature, punctuated by morphisms from the parameterising category S and terminated by $\mathsf{e}(f, v)$ terms. The morphisms from S are used to bridge differences between the domains and codomains of each operation.

An S-parameterised algebraic theory is a pair $(\Sigma, \mathcal{E})$ of a signature and a set of equations $t = t' \in \mathcal{E}$ between terms, where for each pair, both t and t' are in $T_\Sigma(s_1, s_2, V)$ for some s_1 and s_2.

Example 7. An example signature is given by the global state parameterised monad from Example 2. Given a category of state types S, with a terminal object 1, we take the underlying category to be Set and the assume that the functor $\widehat{\cdot} : S \to \text{Set}$ maps every object of S to a countable set. Our signature has two families of operations: $\mathsf{read}_s : s \leftarrow s^s$ and $\mathsf{write}_s(x \in \widehat{s}) : 1 \leftarrow s^1$. The required equations are:

$$\begin{aligned}
\mathsf{op}(id, \mathsf{read}_s, \lambda x.\mathsf{e}(id, v)) &= v \\
\mathsf{op}(id, \mathsf{read}_s, \lambda x.\mathsf{op}(!, \mathsf{write}_s(x), \mathsf{e}(id, v_x))) &= \mathsf{op}(id, \mathsf{read}_s, \lambda x.v_x) \\
\mathsf{op}(id, \mathsf{read}_s, \lambda x.\mathsf{op}(id, \mathsf{read}_s, \lambda y.\mathsf{e}(id, v_{xy}))) &= \mathsf{op}(id, \lambda x.\mathsf{e}(id, v_{xx})) \\
\mathsf{op}(id, \mathsf{write}_s(y), \mathsf{op}(id, \mathsf{read}_s(\lambda x.\mathsf{e}(!, v_x)))) &= v_y
\end{aligned}$$

where we use ! for the unique terminal morphism in S. These equations state that: (1) reading but not using the result is the same as not reading; (2) reading and then writing back the same value is the same as just reading; (3) reading twice in a row is the same as reading once; and (4) writing, reading and clearing is the same as doing nothing.

Algebras For a signature Σ and category C with countable products, a Σ-C-algebra consists of a functor $a : S^{\mathsf{op}} \to C$ and for each operation $\sigma : s_2 \leftarrow s_1^X \in \Sigma$ an arrow $\sigma_a : a(s_1)^X \to a(s_2)$. Given two such algebras a homomorphism

$f : (a, \{\sigma_a\}) \to (b, \{\sigma_b\})$ is a natural transformation $f : a \to b$ such that for all $\sigma : s_2 \leftarrow s_1^X \in \Sigma$, the diagram

$$\begin{array}{ccc} a(s_1)^X & \xrightarrow{f_{s_1}^X} & b(s_1)^X \\ {\scriptstyle \sigma_a}\downarrow & & \downarrow{\scriptstyle \sigma_b} \\ a(s_2) & \xrightarrow{f_{s_2}} & b(s_2) \end{array}$$

commutes.

Example 8. An algebra for the theory in Example 7 is given by $a(s) = (\widehat{s' \times x})^s$ for some set x and $s' \in S$. It is easy to see how to give the appropriate implementations of read and write. Elements of $a(s)$ for some s' and x represent reading and writing computations that start with the global state of type s and end with global state of type s' and a result value in x.

Given a Σ-C-algebra a, terms $t \in T(s_1, s_2, V)$ give rise to derived operations $[t] : a(s_2)^V \to a(s_1)$ by induction on the term t:

$$[\mathsf{e}(f, v)] = \pi_v; a(f)$$
$$[\mathsf{op}(f, \sigma, \{t_i\})] = \langle [t_x] \rangle; \sigma_a; a(f)$$

where π_v is the projection of the countable products of C, and $\langle \cdot \rangle$ is pairing.

A $(\Sigma, \mathcal{E})$-C-algebra consists of a Σ-C-algebra that satisfies every equation in $\mathcal{E}$: for all $t = t' \in \mathcal{E}(s_1, s_2)$ and valuations $f : 1 \to a(s_2)^V$, then $f; [t] = f; [t']$. The collection of $(\Sigma, \mathcal{E})$-C-algebras and homomorphisms forms a category $(\Sigma, \mathcal{E})$-$\mathrm{Alg}(C)$.

Free Algebras A $(\Sigma, \mathcal{E})$-C-algebra $\langle a, \{\sigma_a\} \rangle$ is free for an S-object s and C-object x if there is an arrow $\eta : x \to a(s)$ such that for any other algebra $\langle b, \{\sigma_b\} \rangle$, there is a unique homomorphism $h : a \to b$ such that $\eta; h_a = f$.

Proposition 9. *Given a functor $F : S \times C \to (\Sigma, \mathcal{E})$-$Alg(C)$ such that for all $s \in S$ and $x \in C$, $F(s, x)$ is free for s and x, then $T(s_1, s_2, x) = F(s_2, x)(s_1)$ can be given the structure of an S-parameterised monad on C.*

Free Algebras in Set We now construct the free $(\Sigma, \mathcal{E})$-algebra over a set A. We do this in two stages by first constructing the free Σ-algebra and then by quotienting by the equations in $\mathcal{E}$. Fix Σ, $\mathcal{E}$ and A.

Lemma 10. *The sets $T_\Sigma(s_1, s_2, A)$ can be made into a functor $T_\Sigma : S^{\mathsf{op}} \times S \times \mathrm{Set} \to \mathrm{Set}$.*

Lemma 11. *For all s, the functor $T_\Sigma(-, s, A)$ is the carrier of a Σ-algebra.*

Proof. For each operation $\sigma : s_2 \leftarrow s_1^X$, define $\sigma_{T,s,A} : T_\Sigma(s_1, s, A)^X \to T_\Sigma(s_2, s, A)$ as $\sigma_{T_\Sigma}(\langle t_x \rangle) = \mathsf{op}(id, \sigma, \langle t_x \rangle)$.

Proposition 12. *For each s, A, the algebra $\langle T_\Sigma(-, s, A), \{\sigma_{T_\Sigma}\}\rangle$ is the free algebra for s and A.*

Proof. (Sketch). Given $f : A \to B(s)$ for some Σ-algebra $\langle B, \{\sigma_B\}\rangle$, define $h_{s'} : T(s', s, A) \to B(s')$ by recursion on terms:

$$h_{s'}(\mathsf{e}(g, a)) = B(g)(f(a))$$
$$h_{s'}(\mathsf{op}(g, \sigma, \langle t_i\rangle)) = B(g)(\sigma_B(\langle h_{dom(\sigma)}(t_i)\rangle))$$

The rest of the proof is straightforward, with a little wrinkle in showing that this homomorphism h is unique, which requires a Yoneda-style appeal to the naturality of homomorphisms.

In preparation for quotienting the above free algebra by a set of equations, we define congruences for parameterised algebras and show that quotienting by a congruence gives a parameterised algebra. Given a Σ-algebra $\langle a, \{\sigma_a\}\rangle$ in Set, a family of equivalence relations $\equiv_s$ on a is a congruence if, for all $\sigma : s_1^X \to s_2$, $\forall x \in X. y_x \equiv_{s_1} z_x$ implies $\sigma_A(\langle y_x\rangle) \equiv_{s_2} \sigma_A(\langle z_x\rangle)$ and for all $g : s \to s'$, $x \equiv_{s'} y$ implies $a(g)x \equiv_s a(g)y$.

Lemma 13. *If $\langle A, \{\sigma_A\}\rangle$ is a Σ-algebra and $\equiv$ is a congruence, then $\langle A/\equiv, \{\sigma_A\}\rangle$ is also a Σ-algebra.*

Using our set of equations $\mathcal{E}$, for each $s \in S$, define relations $t \equiv_{s'} t'$ on $T_\Sigma(s', s, A)$ as the smallest congruences satisfying the rule:

$$\frac{(t, t') \in \mathcal{E} \qquad \langle t_x, t'_x \in T_\Sigma(s_2, s, A)\rangle \qquad \forall x. t_x \equiv_{s_2} t'_x}{[t]t_x \equiv_{s_1} [t']t'_x}$$

Set $T_{(\Sigma,\mathcal{E})}(-, s, A) = T_\Sigma(-, s, A)/\equiv$.

Proposition 14. *$T_{\Sigma,\mathcal{E}}(-, s, A)$ is the free $(\Sigma, \mathcal{E})$-algebra over s and A.*

4 Beck's Theorem

In this section we prove Beck's theorem [2] characterising the Eilenberg-Moore category of algebras for a parameterised monad. We base our proof on the proof of Beck's theorem for non-parameterised monads given by Mac Lane [7]. Recall that Beck's theorem for non-parameterised monads characterises the category C^T by the use of coequalisers that are preserved and created by the right adjoint. Due to the additional parameterisation, we cannot use coequalisers for parameterised monads, so we use families of pairs of morphisms with common domain that we call S-spans, and consider universal and absolute versions of these.

We will be dealing with functors that have pairs of contra- and co-variant arguments such as $x : S^{\mathsf{op}} \times S \to A$, where each pair is applied to the same object. We abbreviate such applications like so: $x^s = x(s, s)$, and similarly for multiple pairs of contra-/co-variant arguments.

An S-span in a category A is a pair of functors $x : S^{\mathsf{op}} \times S \times S^{\mathsf{op}} \times S \to A$ and $y : S^{\mathsf{op}} \times S \to A$ and a pair of dinatural transformations $d^{0s_1s_2} : x^{s_1s_2} \to y^{s_1}$ and $d^{1s_1s_2} : x^{s_1s_2} \to y^{s_2}$. A *fill-in* for an S-span is an object z and dinatural transformations $e^s : y^s \to z$ such that, for all s_1, s_2, $d^{0s_1s_2}; e^{s_1} = d^{1s_1s_2}; e^{s_2}$. A *universal* fill-in (e, z) for an S-span d^0, d^1, such that for any other fill-in (f, c) there is a unique arrow $h : z \to c$ such that, for all s, $e^s; h = f^s$. An *absolute* fill-in is a fill-in such that for any functor $T : A \to A'$, the resulting fill-in is universal.

We are interested in S-spans because they arise from T-algebras. Given an S-parameterised monad (T, η, μ) on C, for each T-algebra $\langle x, h\rangle$ we have a fill-in square in $C^{S^{\mathsf{op}}}$:

$$\begin{array}{ccc} T(-, s_1, T(s_1, s_2, x(s_2))) & \xrightarrow{\mu_{-s_1s_2x(s_2)}} & T(-, s_2, x(s_2)) \\ \downarrow{\scriptstyle T(-,s_1,h_{s_1s_2})} & & \downarrow{\scriptstyle h_{-s_2}} \\ T(-, s_1, x(s_1)) & \xrightarrow{h_{-s_1}} & x(-) \end{array}$$

which is just the associativity law for the T-algebra.

Lemma 15. *This fill-in square is universal and absolute.*

The next lemma will also be useful.

Lemma 16. *Universal fill-ins have the following cancellation property. For two arrows $f, g : z \to c$, if, for all s, $e^s; f = e^s; g$, then $f = g$.*

Theorem 17 (Beck). *Let $\langle F, G, \eta, \epsilon\rangle : C \rightharpoonup A$ be an S-parameterised adjunction and $\langle T, \eta, \mu\rangle$ the derived S-parameterised monad. Then A is isomorphic to C^T iff for each S-span d^0, d^1 in A such that the S-span $G(-, d^0), G(-, d^1)$ has an absolute fill-in $(e^-_-, z(-))$, there is a unique universal fill-in (e^-, z) of d^0, d^1 such that $G(s, z) = z(s)$ and $G(s, e^{s'}) = e^{s'}_s$.*

Proof. First, we show the forward direction. We know that A and C^T are isomorphic, so it suffices to show that given S-spans

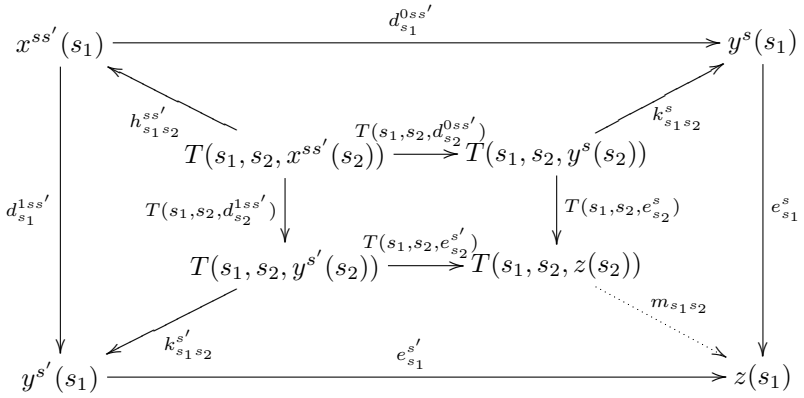
$$\begin{array}{ccc} \langle x^{s_1s_2}, h^{s_1s_2}\rangle & \xrightarrow{d^{0s_1s_2}} & \langle y^{s_1}, k^{s_1}\rangle \\ \downarrow{\scriptstyle d^{1s_1s_2}} & & \\ \langle y^{s_2}, k^{s_2}\rangle & & \end{array}$$

of T-algebras for which the corresponding S-spans in $C^{S^{\mathsf{op}}}$ via G^T have absolute universal fill-ins $(e^s_-, z(-))$

$$\begin{array}{ccc} x^{s_1s_2}(-) & \xrightarrow{d^{0s_1s_2}} & y^{s_1}(-) \\ \downarrow{\scriptstyle d^{1s_1s_2}} & & \downarrow{\scriptstyle e^{s_1}_-} \\ y^{s_2}(-) & \xrightarrow{e^{s_2}_-} & z(-) \end{array}$$

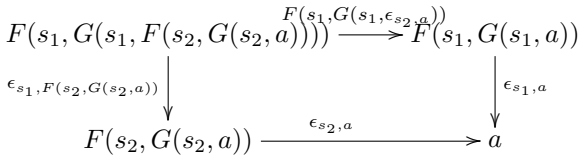
then there is a unique universal fill-in $(e, \langle z, m\rangle)$ in C^T such that $G^T(s, e^{s'}) = e^{s'}_s$ and $G^T(s, \langle z, m\rangle) = z(s)$.

We already know that $z(-)$ is a functor $S^{\mathsf{op}} \to C$ and that the e^s are natural, so we use z as the functor part of our T-algebra. We must find a T-algebra structure $m_{s_1,s_2} : T(s_1, s_2, z(s_2)) \to z(s_1)$. This is induced from the universal fill-in:



The inner and outer squares are the fill-in properties for $T(s_1, s_2, e^-_{s_2})$ and $e^-_{s_1}$ respectively. The top and left regions commute because d^0 and d^1 are homomorphisms of T-algebras. Thus, $k^-_{s_1s_2}; e^-_{s_1}$ is a fill-in for the inner square. Since e^s is an absolute universal fill-in there is a unique arrow $m_{s_1s_2} : T(s_1, s_2, z(s_2)) \to z(s_1)$ such that $T(s_1, s_2, e^s_{s_2}); m_{s_1s_2} = k^s_{s_1s_2}; e^s_{s_1}$. To complete this direction of the proof we must show that the family $m_{s_1s_2}$ is natural in s_1 and dinatural in s_2, that it is the structure map for a T-algebra, and that e^s is a universal fill-in of d^0, d^1. These are easily checked by construction of the appropriate diagrams, and use of Lemma 16.

In the reverse direction, we use the fact that for each object $a \in A$, the adjunction $\langle F, G, \eta, \epsilon\rangle : C \rightharpoonup A$ provides a fill-in



in A. This is the "canonical presentation" of a. This definition, and Lemma 15, are used to prove that there is a unique morphism in PAdj(T) from any other A' to A in a manner similar to Mac Lane's proof. This then implies the result we desire.

Corollary 18. *For any S-theory $(\Sigma, \mathcal{E})$, $(\Sigma, \mathcal{E})$-Alg is parameterised-monadic over* Set.

5 Structured Parameterisation

In this section we develop a small amount of theory to deal with the case when we have structure on the parameterising category S, expressed using functors, that we wish to have on the parameterised monad itself.

Motivation Consider the use of parameterised monads to model effectful computation, where the kinds of effects we may perform are regulated by the parameters of the monad. In the type system of the next section, we have typing judgements of the form $\Gamma; a \vdash M : A; b$, where a and b represent objects of our parameterising category, dictating the start and end states of the computation. This is interpreted as a morphism $[\![\Gamma]\!] \to T(a, b, [\![A]\!])$ for some parameterised monad. The system has a "let" construct for sequencing two programs. Given two typed programs $\Gamma; a \vdash M : A; b$ and $\Gamma, x : A; b \vdash N : B; c$ then an obvious way to type the sequencing of these programs is $\Gamma; a \vdash \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N : B; c$, matching the bs, and to use the multiplication of the monad for the semantics. However, it is likely that the program N requires some knowledge of its start state beyond that given to it by M. If c denotes the extra knowledge required by N, we write the combination as $b \bullet c$ and the full sequencing rule is given by

$$\frac{\Gamma; a \vdash M : A; b \qquad \Gamma, x : A; b \bullet c \vdash N : B; d}{\Gamma; a \bullet c \vdash \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N : B; d.}$$

The operation $- \bullet c$ can be interpreted as a functor on the parameterising category. To give a semantics to the "let" construct, we require some morphism $(- \bullet s)^\dagger : T(s_1, s_2, x) \to T(s_1 \bullet s, s_2 \bullet s, x)$. We call this a *lifting* of the functor $- \bullet s$.

Definition Given an strong S-parameterised monad (T, η, μ, τ), and a functor $F : S \to S$, a *lifting* of F to T is a natural transformation $F^\dagger_{s_1 s_2 x} : T(s_1, s_2, x) \to T(Fs_1, Fs_2, x)$ that commutes with the unit, multiplication and strength of the monad:

$$F^\dagger; T(Fs_1, Fs_2, F^\dagger); \mu = \mu; F^\dagger \qquad \eta; F^\dagger = \eta_F \qquad \tau; F^\dagger = x \times F^\dagger; \tau$$

This definition is from [1].

Example 19. For the state category, we take the free symmetric monoidal category on the category S of state types used in Example 2, which we label S'. We extend the functor $\widehat{\cdot}$ to be $S' \to C$ by mapping objects $s_1 \otimes s_2$ to $\widehat{s_1} \times \widehat{s_2}$, hence $\widehat{\cdot}$ becomes a strict symmetric monoidal functor. A lifting of $s \otimes -$ can be defined as: $(s \otimes -)^\dagger = c \mapsto \lambda(s, s_1).\text{let } (s_2, a) = c(s_1) \text{ in } ((s, s_2), a)$ and similarly for $(- \otimes s)^\dagger$.

Liftings for Algebraic Theories For the S-parameterised monads generated from algebraic theories as in Section 3, we can define liftings for endo-functors F on S, provided there is enough structure on the operations of the algebras. Given

an S-theory $(\Sigma, \mathcal{E})$, if, for every operation $\sigma : s_1 \leftarrow s_2^X \in \Sigma$ there is an operation $F\sigma : Fs_1 \leftarrow (Fs_2)^X$, and all the equations generated by $\mathcal{E}$ are preserved by lifting all operations, then we can define the following operations on the sets $T_\Sigma(s_1, s_2, x)$:

$$F^\dagger(\mathsf{e}(g, x)) = \mathsf{e}(Fg, x) \qquad F^\dagger(\mathsf{op}(g, \sigma, \{t_x\})) = \mathsf{op}(Fg, F\sigma, \{F^\dagger(t_x)\})$$

Proposition 20. *This definition defines a lifting of F to $T_{(\Sigma,\mathcal{E})}$.*

6 Semantics for Type and Effect Systems

We now give an application of the theory developed above by using it to give a semantics to a toy language with explicitly typed effectful operations.

Effect Systems For our language we use a partially-ordered set $\mathcal{A}$ of *permissions* that has an order-preserving binary associative operation $\bullet : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$. Elements of $\mathcal{A}$ are used to represent permissions that permit a program to carry out certain effects. We write the ordering of $\mathcal{A}$ as $a \Rightarrow b$, intended to be reminiscent of logical implication. We use $a, b, c, ...$ to range over elements of $\mathcal{A}$.

The types of our toy language are given by:

$$A, B ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{unit} \mid A \times B \mid (A; a) \to (B; b)$$

The types are standard except for the function type $(A; a) \to (B; b)$. This is the type of functions that take As and return Bs, but may also perform effects allowed by permissions a and bestow permissions b. We refer to types that do not contain function types as *ground*.

An *effect system* consists of a permission algebra $\mathcal{A}$ and a set Ω of operations $\mathsf{op} : (A; a) \to (B; b)$, where A and B are ground.

Example 21. The traditional effect system recording when a program reads or writes global variables can be expressed in our system. Given a set of locations *Loc*, we take $\mathcal{A}$ to be the power set of $\{\mathsf{r}_l, \mathsf{w}_l \mid l \in \mathit{Loc}\}$, ordered by reverse inclusion. The binary operation is defined as $\varepsilon \bullet \varepsilon' = \varepsilon \cup \varepsilon'$. We read $\{\mathsf{r}_{l_1}, \mathsf{r}_{l_2}, \mathsf{w}_{l_2}\}$ as the permission to read the locations l_1 and l_2, the permission to write to location l_2. We have two families of operations:

$$\mathtt{read}_l : (\mathtt{unit}, \{\mathsf{r}_l\}) \to (\mathtt{int}, \{\mathsf{r}_l\})$$
$$\mathtt{write}_l : (\mathtt{int}, \{\mathsf{w}_l\}) \to (\mathtt{unit}, \{\mathsf{w}_l\})$$

The read operation requires that we have the permission to perform a read on the required location and it bestows the permission to still read that location afterwards; likewise for writing.

Example 22. This example enforces an ordering on effects performed, in a similar manner to history effects [11]. The permission algebra is given by:

$$H ::= \alpha \mid H_1 + H_2 \mid H_1.H_2 \mid \epsilon$$

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \frac{}{\Gamma \vdash b : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}}$$

$$\frac{\Gamma \vdash V_1 : \mathtt{int} \qquad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 < V_2 : \mathtt{bool}} \qquad \frac{\Gamma \vdash V_1 : \mathtt{int} \qquad \Gamma \vdash V_2 : \mathtt{int}}{\Gamma \vdash V_1 + V_2 : \mathtt{int}}$$

$$\frac{\Gamma \vdash V_1 : A \qquad \Gamma \vdash V_2 : B}{\Gamma \vdash (V_1, V_2) : A \times B} \qquad \frac{\Gamma \vdash V : A_1 \times A_2}{\Gamma \vdash \pi_i V : A_i}$$

$$\frac{\Gamma, x : A; a \vdash M : B; b}{\Gamma \vdash \lambda(x : A; a).M : (A; a) \to (B; b)} \qquad \frac{\Gamma \vdash V : A \qquad A \sqsubseteq B}{\Gamma \vdash V : B}$$

$$\frac{\Gamma \vdash V : A}{\Gamma; a \vdash \mathtt{val}_a\ V : A; a} \qquad \frac{\Gamma; a \vdash M : A; b \qquad \Gamma, x : A; b \bullet c \vdash N : B; d}{\Gamma; a \bullet c \vdash \mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N : B; d}$$

$$\frac{\Gamma \vdash V_1 : (A; a) \to (B; b) \qquad \Gamma \vdash V_2 : A}{\Gamma; a \vdash V_1\ V_2 : B; b}$$

$$\frac{\Gamma \vdash V : \mathtt{bool} \qquad \Gamma; a \vdash M : A; b \qquad \Gamma; a \vdash N : A; b}{\Gamma; a \vdash \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N : A; b}$$

$$\frac{\Gamma \vdash V : A \qquad \mathtt{op} : (A; a) \to (B; b)}{\Gamma; a \vdash \mathtt{op}\ V : B; b} \qquad \frac{a \Rightarrow a' \qquad \Gamma; a' \vdash M : A; b' \qquad b' \Rightarrow b}{\Gamma; a \vdash M : A; b}$$

$$\frac{\Gamma; a \vdash M : A; b \qquad A \sqsubseteq B}{\Gamma; a \vdash M : B; b}$$

Fig. 1. Typing Rules

over some set of basic permissions α. The ordering treats $H_1.H_2$ as an associative binary operator with unit ϵ, and $H_1 + H_2$ as a meet. Define $H_1 \bullet H_2$ as $H_1.H_2$ Operations can now be defined that require some basic permission $\alpha \in \{\mathsf{in}, \mathsf{out}\}$ to complete:

$$\mathtt{input} : (\mathtt{unit}, \mathsf{in}) \to (\mathtt{int}, \epsilon) \qquad \mathtt{output} : (\mathtt{int}, \mathsf{out}) \to (\mathtt{unit}, \epsilon)$$

The lack of commutativity in $H_1.H_2$ ensures that operations must be carried out in the predefined order prescribed by the starting set of permissions. The lifting operation ensures that a sub-program need not have complete knowledge of the rest of the program's effects.

Typing Rules The typing rules for our language are presented in Figure 1. We split the language into effect-free values V and possibly side-effecting programs M, based on the fine-grain call-by-value calculus of Levy *et al* [6].

$$V ::= x \mid n \mid b \mid () \mid V_1 < V_2 \mid V_1 + V_2 \mid (V_1, V_2) \mid \pi_i V \mid \lambda(x : A; a).M$$

$$M ::= \mathtt{val}_a\ V \mid \mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N \mid V_1\ V_2 \mid \mathtt{if}\ V\ \mathtt{then}\ M\ \mathtt{else}\ N \mid \mathtt{op}\ V$$

Values are typed using a judgement of the form $\Gamma \vdash V : A$, where Γ is a list of variable : type pairs with no duplicated variable names and A is a type. The value typing rules are standard for the given constructs apart from the λ-abstraction rule. This rule's premise is a typing judgement on a program M. Such judgements have the form $\Gamma; a \vdash M : A; b$, where a and b are effect assertions. This judgement states that the program M, when its free variables are typed as in Γ, has effects afforded by a and returns a value of type A, allowing further effects afforded by b.

The language has the usual value and sequencing constructs for a monadic language, extended with permissions. Values are incorporated into programs by the $\mathtt{val}_a\ V$ construct. This is the `return` of normal monadic programming extended with the fact that the program starts in a state described by a and does nothing to that state. Two programs are sequenced using the $\mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N$ construct we introduced in the previous section. There are two subtyping rules for values and computations. The subtyping relation is defined as:

$$\frac{}{\mathtt{int} \sqsubseteq \mathtt{int}} \qquad \frac{}{\mathtt{bool} \sqsubseteq \mathtt{bool}} \qquad \frac{A \sqsubseteq A' \qquad B \sqsubseteq B'}{A \times A' \sqsubseteq B \times B'}$$

$$\frac{A' \sqsubseteq A \qquad a' \Rightarrow a \qquad B \sqsubseteq B' \qquad b \Rightarrow b'}{(A; a) \to (B; b) \sqsubseteq (A'; a') \to (B'; b')}$$

Semantics Using a Parameterised Monad For an effect system $(\mathcal{A}, \Omega)$, the semantics of the language is defined using an $\mathcal{A}$-parameterised monad (T, η, μ) on Set with a lifting for $\bullet$. This gives the following interpretation of the types:

$$[\![\mathtt{int}]\!] = \mathbb{Z} \qquad [\![\mathtt{bool}]\!] = \mathbb{B} \qquad [\![\mathtt{unit}]\!] = \{\star\} \qquad [\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![(A; a) \to (B; b)]\!] = [\![A]\!] \to T(a, b, [\![B]\!]).$$

The interpretation of types extends to typing contexts in the standard way. We interpret value judgements $\Gamma \vdash V : A$ as functions $[\![\Gamma]\!] \to [\![A]\!]$, and computation judgements $\Gamma; a \vdash M : B; b$ as functions $[\![\Gamma]\!] \to T(a, b, [\![B]\!])$. For each operation $\mathtt{op} : (A; a) \to (B; b)$ in Ω we require a function $[\![\mathtt{op}]\!] : [\![A]\!] \to T(a, b, [\![B]\!])$.

The interpretation of terms is now relatively straightforward. Due to the consequence and subtyping rules, we must give the interpretation over typing derivations and not the structure of the terms. To resolve this coherence issue we relate this semantics to a semantics in which these rules are no-ops below.

To define the semantics we use the *bind* operator of type $T(a, b, x) \times (x \to T(b, c, y)) \to T(a, c, y)$, derived from the multiplication of the monad. We give the cases for the parameterised monad structure (where the first three are understood to actually apply to the typing rule with the given term constructor):

$$[\![\mathtt{val}_a\ V]\!]\rho = \eta_a([\![V]\!]\rho)$$
$$[\![\mathtt{let}\ x \Leftarrow M\ \mathtt{in}\ N]\!]\rho = bind_{(a \bullet r)(b \bullet r)d}([\![M]\!]\rho)\ \bullet^\dagger\ r, \lambda x.[\![N]\!](\rho, x)$$

$$[\![\mathtt{op}\ V]\!]\rho = [\![\mathtt{op}]\!]([\![V]\!]\rho)$$

$$\left[\!\!\left[\frac{a \Rightarrow a' \quad \Gamma; a' \vdash M : A; b' \quad b' \Rightarrow b}{\Gamma; a \vdash M : A; b}\right]\!\!\right]\rho = T(a \Rightarrow a', b' \Rightarrow b, [\![M]\!]\rho)$$

In the interpretation of `let`, note the use of the lifting operator $(\bullet^\dagger r)$ to lift the interpretation of M so that it can be sequenced with N.

The subtyping relation $A \sqsubseteq B$ is interpreted as an function $[\![A]\!] \to [\![B]\!]$ over the derivation, using the functor action of T on $\mathcal{A}$. Subtyping rules are interpreted by composition with this interpretation.

Generating Parameterised Monads for Effect Systems We now construct an $\mathcal{A}$-parameterised monad suitable for the previous section by using the free algebras in Sections 3 and 5. Given an effect system $(\mathcal{A}, \Omega)$ we define the corresponding $\mathcal{A}$-signature Σ_Ω as the union of sets of operations deriving from each $\mathtt{op} : (A; a) \to (B; b) \in \Omega$:

$$\{\mathtt{op}_x : a \leftarrow b^X \mid x \in [\![A]\!]\} \cup \{\mathtt{op}_x \bullet c : (a \bullet c) \leftarrow (b \bullet c)^{[\![B]\!]} \mid x \in [\![A]\!], c \in \mathcal{A}\}$$

So for each operation `op` we have a family of operations for each possible input value in the interpretation of A and each possible permission context in which the operation may be used. The duplication of operations over permission contexts will be used to define the lifting operations. The restriction to ground types in operations means that we are not using the definition of the monad that we are currently constructing. By Propositions 9 and 14 we obtain an $\mathcal{A}$-parameterised monad suitable for interpreting our toy language.

Relating Parameterised and Unparameterised Semantics Finally in this section, we relate our semantics with typed algebraic operations to a standard semantics for a given effect system. The basic idea is to assume that, for an effect system $(\mathcal{A}, \Omega)$ and some set of equations $\mathcal{E}$, there is a non-parameterised monad M that gives an adequate semantics for the chosen operational semantics of this language. If the free algebra for this monad supports the theory $(\Sigma_\Omega, \mathcal{E})$ that we have used to prove any equivalences, then we can define an erasure function $\mathsf{erase} : T(a, b, A) \to MA$ that replays each effect in the typed interpretation in the untyped one. By a logical relations argument using Katsumata's notion of $\top\top$-lifting [5], it is possible to show that, for closed programs M of ground type, this means that $\mathsf{erase}([\![M]\!]^t) = [\![M]\!]^u$, where $[\![-]\!]^t$ and $[\![-]\!]^u$ are the semantics in T and M respectively.

Proposition 23. *If the untyped semantics in M is adequate, meaning that if $[\![M]\!]^u = [\![N]\!]^u$ then M and N are contextually equivalent, then $[\![M]\!]^t = [\![N]\!]^t$ implies that M and N are equivalent for all type derivation contexts $-; a \vdash C[-] : \mathtt{bool}; b$.*

Using this proposition it is possible to prove the equivalences given by Benton *et al* [3], using Plotkin and Power's equations for global state [9] and induction on the terms of the free algebra.

7 Conclusions

We have extended the relationship between parameterised monads and adjunctions to include the Eilenberg-Moore category of algebras. We have also given a description of algebraic theories with parameters, and used a generalisation of Beck's theorem to show that the Eilenberg-Moore category and the category of algebras for a given theory coincide. We then used this theory to give a semantics for a toy programming language with effect information recorded in the types.

In future work we wish to develop the theory of algebraic theories with parameters and parameterised monads further:

- We wish to understand more about how structure on the parameterising category S may be carried over to the parameterised monad itself. For every functor $F : S \to S$, with a lifting $F^\dagger$ it is possible to define functors $F^T : C^T \to C^T$ and $F_T : C_T \to C_T$. However, an attempt to define a category of adjunctions with such functors fails because the operation $F^\dagger$ factors across the adjunctions in two different ways for C^T and C_T.
- We wish to construct free algebras with parameters in other categories, especially in some category of domains suitable for interpreting recursion.
- We have not characterised the parameterised monads that arise from algebraic theories. We expect that this will simply be parameterised monads that preserve the right kind of filtered colimits, as for normal monads [10].

References

1. Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19, 2009.
2. Jonathan Mock Beck. Triples, algebras and cohomology. *Reprints in Theory and Applications of Categories*, 2:1–59, 2003.
3. N. Benton, A. Kennedy, M. Hofmann, and L. Beringer. Reading, writing and relations. In *APLAS*, volume 4279 of *LNCS*, pages 114–130. Springer, 2006.
4. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, 1986.
5. Shin-ya Katsumata. A Semantic Formulation of $\top\top$-Lifting and Logical Predicates for Computational Metalanguage. In C.-H. Luke Ong, editor, *CSL*, volume 3634 of *LNCS*, pages 87–102. Springer, 2005.
6. Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. and Comp.*, 185:182–210, 2003.
7. Saunders Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 2nd edition, 1998.
8. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
9. Gordon D. Plotkin and John Power. Notions of computation determine monads. In *FOSSACS 2002*, number 2303 in LNCS, April 2002.
10. Edmund Robinson. Variations on algebra: Monadicity and generalisations of equational theories. *Formal Asp. Comput.*, 13(3–5):308–326, 2002.
11. Christian Skalka and Scott Smith. History Effects and Verification. In W.-N. Chin, editor, *APLAS*, volume 3302 of *LNCS*, pages 107–128. Springer, 2004.