Modular Monad Transformers

Mauro Jaskelioff

Functional Programming Laboratory—University of Nottingham

Abstract. During the last two decades, monads have become an indispensable tool for structuring functional programs with computational effects. In this setting, the mathematical notion of a monad is extended with operations that allow programmers to manipulate these effects. When several effects are involved, monad transformers can be used to build up the required monad one effect at a time. Although this seems to be modularity nirvana, there is a catch: in addition to the construction of a monad, the effect-manipulating operations need to be lifted to the resulting monad. The traditional approach for lifting operations is non-modular and ad-hoc. We solve this problem with a principled technique for lifting operations that makes monad transformers truly modular.

1 Introduction

Since monads were introduced by Moggi [13, 14] to model computational effects, they have proven to be extremely useful to structure functional programs [20, 19, 9]. Monads are usually accompanied with operations that manipulate the effects they model. For example, an exception monad may come with operations for throwing an exception and for handling it, and a state monad may come with operations for reading and updating the state. Consequently, the structure one is really working with is a monad and a set of operations associated to it.

In order to combine computational effects, one must combine monads. There are many ways of combining monads: distributive laws [2], coproduct of monads [11], and monad transformers [10, 15, 3]. However, these technologies fall short in combining monads with operations, as they only provide means to combine monads. Liang et al. [10] identified this problem more than a decade ago and proposed a workaround, which is not modular. In fact, they have to *lift* operations associated to a monad through a monad transformer in an ad-hoc manner, and therefore the number of liftings of operation grows like the product of the number of monad transformers and operations involved (see Section 3.)

More recently, Plotkin et al. [17, 7] have proposed to look at monads induced by algebraic theories, and to address the problem of combining monads (and associated operations) as a problem of combining algebraic theories. Their approach works very smoothly, but can only deal with monads induced by algebraic theories (and lifting is limited to algebraic operations).

Of all the techniques for combining monads, monad transformers are the most popular among functional programmers, as they are easy to implement and capture all the desired combinations for standard effects[1]. We show that for

[1] We take as standard the monads and operations described in [10].

monad transformers with a *functorial behaviour* there is a uniform definition of lifting for a class of operations, which includes (after some minor repackaging) all the operations described in [10]. The main contributions of this article are:

- Identifying a class of operations associated to a monad, called *algebraic $\hat{\Sigma}$-operations*, that are easy to lift along any *monad morphism* (Section 4).
- Showing that all $\hat{\Sigma}$-operations for a monad can be lifted (through any *functorial monad transformer*) by interpreting them as algebraic $\hat{\Sigma}$-operations for a related monad (Section 5).
- Comparing our uniform lifting to more ad-hoc liftings found in the literature or in Haskell's libraries. This has revealed a mismatch with one definition in Haskell's monad transformer library (as discussed in Section 4).

Our approach extends both the traditional monad transformer approach [10] with the addition of uniform liftings, and the algebraic approach [7], since algebraic operations are a special case of algebraic $\hat{\Sigma}$-operations.

Remark 1. This article is aimed at researchers and programmers interested in using monads to structure functional programs with computational effects. Formally we work with system $F\omega$. In examples and remarks we may freely use extensions of $F\omega$ or idioms that are customary in functional languages.

Much of the terminology we introduce is borrowed from Category Theory. Usually, there is not an exact correspondence between category-theoretic notions and their formalization in a calculus. For instance, monads expressible in the simple typed lambda calculus correspond to strong monads in a CCC [14]. In what follows, when we say *monad* we mean *expressible monad* in $F\omega$ (and similarly for other category-theoretic notions).

2 Preliminaries

We work with system $F\omega$ and its equational theory induced by $\beta\eta$-equivalence (for details, see [1, 5]). One may replace $F\omega$ with a weaker system, like HML [6] (which distinguishes types from type schemas), or a stronger system, like CC [4]. To fix the notation, we recall the syntax of $F\omega$

$$\begin{array}{rl} \text{kinds} & k ::= * \mid k \to k \\ \text{type constuctors} & U ::= X \mid U \to U \mid \forall X\!:\!k.\ U \mid \Lambda X\!:\!k.\ U \mid U\,U \\ \text{terms} & e ::= x \mid \lambda x\!:\!X.\ e \mid \Lambda X\!:\!k.\ e \mid e\,U \end{array}$$

We write e_U for $e\,U$ (polymorphic instantiation) and we often write definitions $g_X(x\!:\!A) \mathrel{\hat{=}} t$ when we mean $g \mathrel{\hat{=}} \forall X\!:\!*.\ \lambda x\!:\!A.\ t$. We often write term application using a tuple, that is, we write $t\,(z_1, \ldots, z_n)$ for $t\ z_1\ \ldots\ z_n$.

Following [18] we express in the setting of $F\omega$ several category-theoretic notions, such as functors, natural transformations, monads, monad transformers. Familiarity with these notions is not needed to understand the rest of the paper, but interested readers may want to look at [16, 3].

Definition 2 (Functor [18]). *The set* **Functor** *of* functors *consists of pairs* $\hat{F} = (F, \mathsf{map}^F)$*, where* $F : * \to *$ *is a type constructor and*

$$\mathsf{map}^F : \mathsf{Map}(F) \mathrel{\hat{=}} \forall X, Y : *.\ (X \to Y) \to FX \to FY$$

is a term such that for all $f : A \to B$ *and* $g : B \to C$

$$\mathsf{map}^F_{A,A}\ \mathsf{id}_A = \mathsf{id}_{FA} \tag{1}$$

$$\mathsf{map}^F_{A,C}\ (g \cdot f) = \mathsf{map}^F_{B,C}\ g \cdot \mathsf{map}^F_{A,B}\ f \tag{2}$$

where, $\mathsf{id} \mathrel{\hat{=}} \Lambda X : *.\ \lambda x : X.\ x$ *and* $g \cdot f$ *is function composition* $\lambda x : A.g(f\ x)$*.*

The composite functor $\hat{F} \circ \hat{G}$ *is the pair* $(F \cdot G, \mathsf{map})$ *where*

$$\mathsf{map}_{A,B}\ (f : A \to B) \mathrel{\hat{=}} \mathsf{map}^F_{GA,GB}(\mathsf{map}^G_{A,B} f).$$

Definition 3 (Natural transformation). *Given two functors* $\hat{F}$ *and* $\hat{G}$*, the set* $\mathbf{Nat}(\hat{F}, \hat{G})$ *of* natural transformations *from* $\hat{F}$ *to* $\hat{G}$ *consists of terms* $\tau : F \xrightarrow{\bullet} G \mathrel{\hat{=}} \forall X : *.\ FX \to GX$*, such that for all* $f : A \to B$

$$\mathsf{map}^G_{A,B}\ f \cdot \tau_A \ = \ \tau_B \cdot \mathsf{map}^F_{A,B}\ f \tag{3}$$

The term $\iota \mathrel{\hat{=}} \Lambda(M : * \to *)(X : *).\ \lambda m : MX.\ m$ *is the identity natural transformation,* $\sigma \circ \tau \mathrel{\hat{=}} \Lambda X : *.\ \sigma_X \cdot \tau_X$ *is composition of natural transformations, and* $\hat{\Sigma}(\tau : F \xrightarrow{\bullet} G) : \Sigma \cdot F \xrightarrow{\bullet} \Sigma \cdot G \mathrel{\hat{=}} \Lambda X : *.\ \mathsf{map}^{\Sigma}_{FX,GX}(\tau_X)$ *is the application of a functor* $\hat{\Sigma}$ *to a natural transformation* τ *from* $\hat{F}$ *to* $\hat{G}$*.*

Definition 4 (Monad). *The set* **Monad** *of* monads *consists of triples* $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$*, where* $M : * \to *$ *is a type constructor and*

$$\mathsf{ret}^M : \mathsf{Ret}(M) = \forall X : *.\ X \to MX$$

$$\mathsf{bind}^M : \mathsf{Bind}(M) = \forall X, Y : *.\ MX \to (X \to MY) \to MY$$

are terms such that for every $a : A$*,* $f : A \to MB$*,* $m : MA$ *and* $g : B \to MC$*:*

$$\mathsf{bind}^M_{A,B}(\mathsf{ret}^M_A\ a, f) \ = \ f\ a \tag{4}$$

$$\mathsf{bind}^M_{A,A}(m, \mathsf{ret}^M_A) \ = \ m \tag{5}$$

$$\mathsf{bind}^M_{A,C}(m, \lambda a : A.\ \mathsf{bind}^M_{B,C}(f\ a, g)) \ = \ \mathsf{bind}^M_{B,C}(\mathsf{bind}^M_{A,B}(m, f), g) \tag{6}$$

Every monad $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$ *has an underlying functor* (M, map^M)*, denoted by* $\hat{M}$*, where* $\mathsf{map}^M_{A,B}\,(f : A \to B)\,(m : MA) \mathrel{\hat{=}} \mathsf{bind}^M_{A,B}(m, \mathsf{ret}^M_B \cdot f)$*.*

Example 5 (State Monad). The monad for modelling side-effects on a state of type S is $\hat{\mathsf{S}} = (\mathsf{S}, \mathsf{ret}^{\mathsf{S}}, \mathsf{bind}^{\mathsf{S}})$, where $\mathsf{S}\,(X : *) \mathrel{\hat{=}} S \to X \times S$ and

$$\mathsf{ret}^{\mathsf{S}}_X(x : X) : \mathsf{S}\,X \mathrel{\hat{=}} \lambda s : S.\ (x, s)$$

$$\mathsf{bind}^{\mathsf{S}}_{X,Y}(m : \mathsf{S}\,X, f : X \to \mathsf{S}\,Y) : \mathsf{S}\,Y \mathrel{\hat{=}} \lambda s : S.\ \mathsf{let}\ (a, s') = m\ s\ \mathsf{in}\ f\,a\,s'$$

Intuitively, a computation $\mathsf{S}X$ takes an initial state and produces a value of type X and a final state, $\mathsf{ret}^{\mathsf{S}}$ does not change the state, and $\mathsf{bind}^{\mathsf{S}}$ threads the state. A simple calculation shows that equations 4–6 hold. □

Example 6 (Continuation Monad). The monad for modelling continuations of result type R is $\hat{\mathsf{C}} = (\mathsf{C}, \mathsf{ret}^{\mathsf{C}}, \mathsf{bind}^{\mathsf{C}})$, where $\mathsf{C}\,(X : *) \mathrel{\hat{=}} (X \to R) \to R$ and

$$\mathsf{ret}^{\mathsf{C}}_X\,(x : X) \colon \mathsf{C}\,X \mathrel{\hat{=}} \lambda k : X \to R.\ k\,x$$
$$\mathsf{bind}^{\mathsf{C}}_{X,Y}(m : \mathsf{C}X, f : X \to \mathsf{C}\,Y) \colon \mathsf{C}\,Y \mathrel{\hat{=}} \lambda k : Y \to R.\ m\,(\lambda x : X.\ f\,x\,k)$$

Intuitively, $\mathsf{C}X$ is a computation that given a continuation $X \to R$ returns a result in R, $\mathsf{ret}^{\mathsf{C}}$ simply runs a continuation, and $\mathsf{bind}^{\mathsf{C}}(m, f)$ runs m with a continuation constructed by running f in the current continuation. □

Definition 7 (Monad Morphism). *Given two monads $\hat{M}$ and $\hat{N}$, the set* $\mathbf{MM}(\hat{M}, \hat{N})$ *of* monad morphisms *from $\hat{M}$ to $\hat{N}$ consists of terms $\xi \colon M \xrightarrow{\bullet} N$, such that for every $a : A$, $m : MA$ and $f : A \to MB$*

$$\mathsf{ret}^N_A\,a \;=\; \xi_A(\mathsf{ret}^M_A\,a) \tag{7}$$
$$\xi_B(\mathsf{bind}^M_{A,B}(m, f)) \;=\; \mathsf{bind}^N_{A,B}(\xi_A\,m, \xi_B \cdot f) \tag{8}$$

Remark 8. A simple consequence of equations 7–8 is that a monad morphism is also a natural transformation between the underlying functors.

In order to combine effects, instead of writing a monad from scratch, one can add more effects to a pre-existing monad using monad transformers.

Definition 9 (Monad Transformer). *The set* **MT** *of* monad transformers *consists of tuples $\hat{T} = (T, \mathsf{ret}^T, \mathsf{bind}^T, \mathsf{lift}^T)$, where $T \colon (* \to *) \to (* \to *)$ and*

$$\mathsf{ret}^T : \forall M : * \to *.\ \mathsf{Ret}(M) \to \mathsf{Bind}(M) \to \mathsf{Ret}(TM)$$
$$\mathsf{bind}^T : \forall M : * \to *.\ \mathsf{Ret}(M) \to \mathsf{Bind}(M) \to \mathsf{Bind}(TM)$$
$$\mathsf{lift}^T : \forall M : * \to *.\ \mathsf{Ret}(M) \to \mathsf{Bind}(M) \to \forall X : *.\ MX \to TMX$$

are terms such that for every monad $\hat{M}$, the tuple $\hat{T}\hat{M} \mathrel{\hat{=}} (TM, \mathsf{ret}^T_{\hat{M}}, \mathsf{bind}^T_{\hat{M}})$ is a monad and $\mathsf{lift}^T_{\hat{M}}$ is a monad morphism from $\hat{M}$ to $\hat{T}\hat{M}$, where

$$\mathsf{ret}^T_{\hat{M}} \mathrel{\hat{=}} \mathsf{ret}^T_M(\mathsf{ret}^M, \mathsf{bind}^M),\ \mathsf{bind}^T_{\hat{M}} \mathrel{\hat{=}} \mathsf{bind}^T_M(\mathsf{ret}^M, \mathsf{bind}^M),$$
$$\mathsf{lift}^T_{\hat{M}} \mathrel{\hat{=}} \mathsf{lift}^T_M(\mathsf{ret}^M, \mathsf{bind}^M).$$

From now on we will drop type information of kind $*$ from examples, in order to make them more readable.

Example 10. The state monad transformer $\hat{\mathcal{S}} = (\mathcal{S}, \mathsf{ret}^{\mathcal{S}}, \mathsf{bind}^{\mathcal{S}}, \mathsf{lift}^{\mathcal{S}})$ adds side-effects to an existing monad, where $\mathcal{S}\,(M : * \to *)(X : *) \mathrel{\hat{=}} S \to M(X \times S)$, and

$$\mathsf{ret}^{\mathcal{S}}_{\hat{M}}(x : X) : \mathcal{S}MX \mathrel{\hat{=}} \lambda s.\ \mathsf{ret}^M(x, s)$$
$$\mathsf{bind}^{\mathcal{S}}_{\hat{M}}(t : \mathcal{S}MX, f : X \to \mathcal{S}MY) : \mathcal{S}MY \mathrel{\hat{=}} \lambda s.\ \mathsf{bind}^M(t\,s, \lambda(x, s').\ f\,x\,s')$$
$$\mathsf{lift}^{\mathcal{S}}_{\hat{M}}(m : MX) : \mathcal{S}MX \mathrel{\hat{=}} \lambda s.\ \mathsf{bind}^M(m, \lambda x.\ \mathsf{ret}^M(x, s))$$

A simple calculation shows that equations 4–6 hold for $\hat{\mathcal{S}}\hat{M}$ and equations 7–8 hold for $\mathsf{lift}^{\mathcal{S}}_{\hat{M}}$, whenever equations 4–6 hold for $\hat{M}$. □

Example 11. The exception monad transformer $\hat{\mathcal{X}} = (\mathcal{X}, \mathsf{ret}^{\mathcal{X}}, \mathsf{bind}^{\mathcal{X}}, \mathsf{lift}^{\mathcal{X}})$ adds exceptions to an existing monad, where $\mathcal{X}(M : * \to *)(X : *) \mathrel{\hat{=}} M(Z + X)$ (here Z is the type of exceptions), and

$$\mathsf{ret}^{\mathcal{X}}_{\hat{M}}(x : X) : \mathcal{X}MX \mathrel{\hat{=}} \mathsf{ret}^{M}(\mathsf{inr}\, x)$$

$$\mathsf{bind}^{\mathcal{X}}_{\hat{M}}(t : \mathcal{X}MX, f : X \to \mathcal{X}MY) : \mathcal{X}MY \mathrel{\hat{=}} \mathsf{bind}^{M}(t, \lambda c.\, \mathsf{case}\ c\ \mathsf{of}\ \mid \mathsf{inl}\, z \Rightarrow \mathsf{ret}^{M}(\mathsf{inl}\, z) \mid \mathsf{inr}\, x \Rightarrow f\, x)$$

$$\mathsf{lift}^{\mathcal{X}}_{\hat{M}}(m : MX) : \mathcal{X}MX \mathrel{\hat{=}} \mathsf{bind}^{M}(m, \mathsf{ret}^{\mathcal{X}}_{\hat{M}}\, x)$$

A simple calculation shows that equations 4–6 hold for $\hat{\mathcal{X}}\hat{M}$ and equations 7–8 hold for $\mathsf{lift}^{\mathcal{X}}_{\hat{M}}$ whenever equations 4–6 hold for $\hat{M}$. □

3 Operations and lifting

We seek a general technique for lifting operations associated to a monad $\hat{M}$ to another monad $\hat{N}$. In this section we make precise what kind of operations our technique will be able to handle, and what lifting means.

Definition 12 ($\hat{\Sigma}$-operation). *If $\hat{\Sigma}$ is a functor and $\hat{M}$ is a monad, then a $\hat{\Sigma}$-operation for $\hat{M}$ is a natural transformation* op *in* $\mathbf{Nat}(\hat{\Sigma} \circ \hat{M}, \hat{M})$.

Example 13. The standard operations for the state monad are

$$\mathsf{get}\,(k : S \to \mathsf{S}X) : \mathsf{S}X \mathrel{\hat{=}} \lambda s.\, k\, s\, s$$
$$\mathsf{set}\,(s : S, m : \mathsf{S}X) : \mathsf{S}X \mathrel{\hat{=}} \lambda\_.\, m\, s.$$

The operation get applies the current state to its argument, and set sets runs a stateful computation in the provided state. They are $\hat{\Sigma}$-operations for the following functors

$$\Sigma^{\mathsf{get}}\, X \mathrel{\hat{=}} S \to X \qquad \mathsf{map}^{\Sigma^{\mathsf{get}}}(f : X \to Y, t : \Sigma^{\mathsf{get}} X) : \Sigma^{\mathsf{get}} Y \mathrel{\hat{=}} \lambda s.\, f\,(t\, s)$$
$$\Sigma^{\mathsf{set}}\, X \mathrel{\hat{=}} S \times X \qquad \mathsf{map}^{\Sigma^{\mathsf{set}}}(f : X \to Y, (s, x) : \Sigma^{\mathsf{set}} X) : \Sigma^{\mathsf{set}} Y \mathrel{\hat{=}} (s, f\, x).$$

In Fig. 1, we show some $\hat{\Sigma}$-operations (all the monads and $\hat{\Sigma}$-operations are presented along the paper, except for the list monad and its operations for which the reader may consult [19]). Interestingly, all the operations considered in [10] for these monads are *definable* in terms of $\hat{\Sigma}$-operations. For example, we can use the $\hat{\Sigma}$-operations in Example 13 to define the more usual operations

$$\underline{\mathsf{get}} : \mathsf{S}\, S \mathrel{\hat{=}} \mathsf{get}_S(\mathsf{ret}^{\mathsf{S}}_S) = \lambda s.\, (s, s)$$
$$\underline{\mathsf{set}} : S \to \mathsf{S}\, \mathbf{1} \mathrel{\hat{=}} \lambda s.\, \mathsf{set}_{\mathbf{1}}(s, \mathsf{ret}^{\mathsf{S}}_{\mathbf{1}}(\bullet)) = \lambda s\ s'.\, (\bullet, s)$$

where $\bullet$ is the sole inhabitant of the unit type $\mathbf{1}$. In the same manner, we can define

$$\underline{\mathsf{ask}} : \mathsf{R}\, E \mathrel{\hat{=}} \mathsf{ask}_E(\mathsf{ret}^{\mathsf{R}}_E) = \lambda e.\, e$$

Monad	Signature	$\hat{\Sigma}$-operations
List $\mathsf{L}X \mathrel{\hat{=}} [X]$	$\Sigma^{\mathsf{empty}} X \mathrel{\hat{=}} \mathbf{1}$ $\Sigma^{\mathsf{append}} X \mathrel{\hat{=}} X \times X$	$\mathsf{empty}_X : \mathbf{1} \to \mathsf{L}X$ $\mathsf{append}_X : \mathsf{L}X \times \mathsf{L}X \to \mathsf{L}X$
Output $\mathsf{O}X \mathrel{\hat{=}} X \times [A]$	$\Sigma^{\mathsf{output}} X \mathrel{\hat{=}} [A] \times X$ $\Sigma^{\mathsf{flush}} X \mathrel{\hat{=}} X$	$\mathsf{output}_X : [A] \times \mathsf{O}X \to \mathsf{O}X$ $\mathsf{flush}_X : \mathsf{O}X \to \mathsf{O}X$
State $\mathsf{S}X \mathrel{\hat{=}} S \to X \times S$	$\Sigma^{\mathsf{get}} X \mathrel{\hat{=}} S \to X$ $\Sigma^{\mathsf{set}} X \mathrel{\hat{=}} S \times X$	$\mathsf{get}_X : (S \to \mathsf{S}X) \to \mathsf{S}X$ $\mathsf{set}_X : S \times \mathsf{S}X \to \mathsf{S}X$
Environment $\mathsf{R}X \mathrel{\hat{=}} E \to X$	$\Sigma^{\mathsf{ask}} X \mathrel{\hat{=}} E \to X$ $\Sigma^{\mathsf{local}} X \mathrel{\hat{=}} (E \to E) \times X$	$\mathsf{ask}_X : (E \to \mathsf{R}X) \to \mathsf{R}X$ $\mathsf{local}_X : (E \to E) \times \mathsf{R}X \to \mathsf{R}X$
Exception $\mathsf{X}X \mathrel{\hat{=}} Z + X$	$\Sigma^{\mathsf{throw}} X \mathrel{\hat{=}} \mathbf{1}$ $\Sigma^{\mathsf{handle}} X \mathrel{\hat{=}} X \times (Z \to X)$	$\mathsf{throw}_X : \mathbf{1} \to \mathsf{X}X$ $\mathsf{handle}_X : \mathsf{X}X \times (Z \to \mathsf{X}X) \to \mathsf{X}X$
Continuation $\mathsf{C}X \mathrel{\hat{=}} (X \to R) \to R$	$\Sigma^{\mathsf{abort}} X \mathrel{\hat{=}} R$ $\Sigma^{\mathsf{callcc}} X \mathrel{\hat{=}} (X \to R) \to X$	$\mathsf{abort}_X : R \to \mathsf{C}X$ $\mathsf{callcc}_X : ((\mathsf{C}X \to R) \to \mathsf{C}X) \to \mathsf{C}X$

Fig. 1. $\hat{\Sigma}$-operations for the standard monads.

for the environment monad, and

$$\underline{\mathsf{output}} : [A] \to \mathsf{O}\mathbf{1} \mathrel{\hat{=}} \lambda w.\, \mathsf{output}_{\mathbf{1}}(w, \mathsf{ret}^{\mathsf{O}}_{\mathbf{1}}(\bullet)) = \lambda w.\, (\bullet, w)$$

for the output monad. The usual call-with-current-continuation $\underline{\mathsf{callcc}}$ and the $\hat{\Sigma}$-operation callcc are defined as:

$$\underline{\mathsf{callcc}}\,(f : (X \to \mathsf{C}Y) \to \mathsf{C}X) : \mathsf{C}X \mathrel{\hat{=}} \lambda k.\, f\,(\lambda x\ \_.\, k\, x)\, k$$
$$\mathsf{callcc}\,(f : (\mathsf{C}X \to R) \to \mathsf{C}X) : \mathsf{C}X \mathrel{\hat{=}} \lambda k.\, f\,(\lambda m.\, m\, k)\, k$$

The operation $\underline{\mathsf{callcc}}$ can be defined from callcc as:

$$\underline{\mathsf{callcc}}\, f \mathrel{\hat{=}} \mathsf{callcc}\,(\lambda k.\, f\,(\lambda x\ \_.\, k\,(\mathsf{ret}^{\mathsf{C}} x))) \tag{9}$$

Definition 14 (Lifting). *Let* op *be a* $\hat{\Sigma}$*-operation for* $\hat{M}$ *and* ξ *be a monad morphism from* $\hat{M}$ *to* $\hat{N}$*. A* lifting *of* op *to* $\hat{N}$ *along* ξ *is a* $\hat{\Sigma}$*-operation* op^N *for* $\hat{N}$ *such that for all* $X : *$*,*

$$\xi_X \cdot \mathsf{op}_X \;=\; \mathsf{op}^N_X \cdot (\mathsf{map}^{\Sigma}_{MX,NX}\, \xi_X) \tag{10}$$

or equivalently, such that the following diagram commutes:

$$\begin{array}{ccc} \Sigma(NX) & \xrightarrow{\mathsf{op}^N_X} & NX \\ {\scriptstyle \mathsf{map}^{\Sigma}\xi_X}\uparrow & & \uparrow{\scriptstyle \xi_X} \\ \Sigma(MX) & \xrightarrow[\mathsf{op}_X]{} & MX \end{array}$$

This definition can be specialised to the case of a monad transformer $\hat{T}$ by taking $\hat{N} \mathrel{\hat{=}} \hat{T}\hat{M}$ and $\xi \mathrel{\hat{=}} \mathsf{lift}^T_{\hat{M}}$. In this case we call op^N a lifting of op through $\hat{T}$.

In the absence of a general technique, the only way to lift an operation is to do it in an ad-hoc manner, for each monad transformer [10]. Although this works, the approach has significant shortcomings:

– The number of liftings grows like the product of the number of operations and monad transformers. This is clearly non-modular: adding a new monad transformer with some operations involves showing how to lift every existing operation through the new monad transformer, and showing how to lift the new operations through every existing monad transformer.
– Without a uniform definition of lifting, one could have different ad-hoc liftings of the same operation through a monad transformer, and no clear criteria to choose among them.
– There is no division of concerns: defining a lifting involves understanding the intended semantics of both the transformer and the operation.

We show that for well-behaved $\hat{\Sigma}$-operations, called algebraic, there is a unique way to lift them among a monad morphism. Moreover, for all $\hat{\Sigma}$-operations (not necessarily algebraic) there is a uniform way to lift them through a wide class of monad transformers, called functorial monad transformers.

4 Unique Lifting of Algebraic Operations

We characterize operations that interact well with bind.

Definition 15 (Algebraic $\hat{\Sigma}$-operation). *A $\hat{\Sigma}$-operation* op *for* $\hat{M}$ *is* algebraic *provided that for every* $f : A \to MB$ *and* $t : \Sigma(MA)$

$$\mathsf{bind}^M_{A,B}(\mathsf{op}_A\, t, f) \;=\; \mathsf{op}_B(\mathsf{map}^\Sigma_{MA,MB}(\lambda m : MA.\ \mathsf{bind}^M_{A,B}(m, f))\ t) \quad (11)$$

or equivalently, that the following diagram commutes:

$$\begin{array}{ccc} \Sigma(MA) & \xrightarrow{\mathsf{op}_A} & MA \\ {\scriptstyle \mathsf{map}^\Sigma \mathsf{bind}^M(-,f)}\downarrow & & \downarrow{\scriptstyle \mathsf{bind}^M(-,f)} \\ \Sigma(MB) & \xrightarrow[\mathsf{op}_B]{} & MB \end{array}$$

Remark 16. The notion of algebraic operation given in [17] corresponds to algebraic $\hat{\Sigma}$-operations for functors $\hat{\Sigma}$ of the form $\Sigma X = A \times (B \to X)$.

As examples of algebraic $\hat{\Sigma}$-operations we have all the operations in Fig.1, except for flush, local and handle, for which equation 11 does not hold. Remarkably, callcc is an algebraic $\hat{\Sigma}$-operation despite not being algebraic in the sense of [17] and hence, not tractable in that approach. With our generalization, callcc is not only tractable, but also well-behaved.

The following proposition presents a bijection between algebraic operations and natural transformations of a particular type. It provides an alternative way of verifying that an operation is algebraic and it will play a crucial role in showing how to lift algebraic operations.

Proposition 17. *There is a bijection between algebraic $\hat{\Sigma}$-operations for $\hat{M}$ and natural transformations from $\hat{\Sigma}$ to $\hat{M}$ given by:*

$$\phi(\mathsf{op}\colon \Sigma \cdot M \xrightarrow{\bullet} M)\colon (\Sigma \xrightarrow{\bullet} M) \mathrel{\hat{=}} \Lambda X\colon *.\ \mathsf{op}_X \cdot (\mathsf{map}^{\Sigma}_{X,MX} \mathsf{ret}^M_X)$$
$$\psi(\mathsf{op}'\colon \Sigma \xrightarrow{\bullet} M)\colon \Sigma \cdot M \xrightarrow{\bullet} M \mathrel{\hat{=}} \Lambda X\colon *.\ \mathsf{join}^M_X \cdot \mathsf{op}'_{MX}$$

where $\mathsf{join}^M_X \mathrel{\hat{=}} \lambda m\colon M(MX).\ \mathsf{bind}^M_{MX,X}(m, \mathsf{id}_{MX})\colon M(MX) \to MX$ *is the multiplication of* $\hat{M}$*. We call* op' *the natural transformation corresponding to the algebraic* $\hat{\Sigma}$*-operation* op*.*

Remark 18. When $\Sigma X = A \times (B \to X)$ there is a further bijection between algebraic $\hat{\Sigma}$-operations op for $\hat{M}$ and maps $\mathsf{op}''\colon A \to MB$, namely

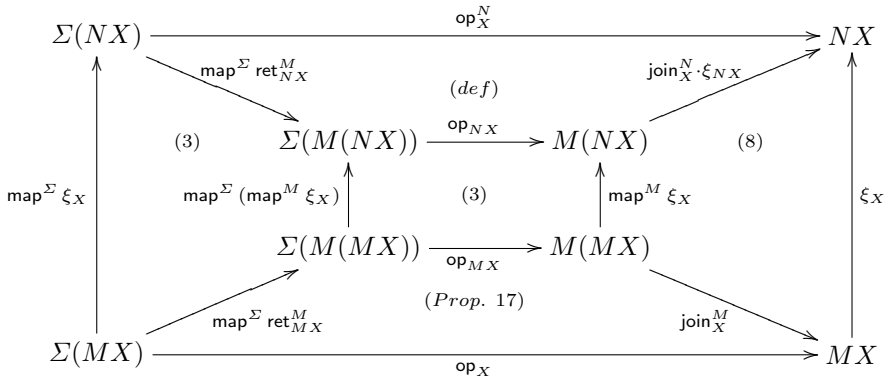
$$\mathsf{op}''(a\colon A) \mathrel{\hat{=}} \mathsf{op}_B(a, \mathsf{ret}^M_B).$$

Theorem 19 (Algebraic Lifting). *Given an algebraic $\hat{\Sigma}$-operation op for $\hat{M}$ and a monad morphism ξ from $\hat{M}$ to $\hat{N}$, define the term $\mathsf{op}^N\colon \Sigma \cdot N \xrightarrow{\bullet} N$ as*

$$\mathsf{op}^N_X \mathrel{\hat{=}} \mathsf{join}^N_X \cdot \xi_{NX} \cdot \mathsf{op}_{NX} \cdot (\mathsf{map}^{\Sigma}_{NX,M(NX)}\, \mathsf{ret}^M_{NX})$$

op^N is an algebraic $\hat{\Sigma}$-operation for $\hat{N}$ and a lifting of op along ξ. Moreover, op^N is the unique lifting of op which is algebraic.

Proof. The operation op^N is a lifting since the following diagram commutes:



By Prop. 17, op^N is algebraic and, by the same proposition, it must be the unique lifting of op which is algebraic. □

For example, when $\hat{N} = \hat{\mathcal{X}}\hat{\mathsf{S}}$ and $\xi = \mathsf{lift}^{\mathcal{X}}_{\hat{\mathsf{S}}}$, thus $NX = S \to ((Z + X) \times S)$, then the algebraic lifting of the algebraic $\hat{\Sigma}$-operation get yields the operation

$$\mathsf{get}^{\mathcal{X}}_X(k\colon S \to \mathcal{X}\mathsf{S}X)\colon \mathcal{X}\mathsf{S}X \mathrel{\hat{=}} \lambda s.\, k\, s\, s.$$

Since callcc is an algebraic $\hat{\Sigma}$-operation, we can apply the algebraic lifting and obtain for every monad morphism ξ from C to $\hat{N}$ a lifted algebraic operation

$\mathsf{callcc}^N : \forall X : *.\ ((NX \to R) \to NX) \to NX$. For example, for $\hat{N} = \hat{\mathcal{S}}\hat{\mathsf{C}}$ and $\xi = \mathsf{lift}^{\mathcal{S}}_{\mathsf{C}}$, thus $NX = S \to ((X \times S) \to R) \to R$, then the operation simplifies to:

$$\mathsf{callcc}^{\mathcal{S}}\,(f : (\mathcal{S}\mathsf{C}X \to R) \to \mathcal{S}\mathsf{C}X) : \mathcal{S}\mathsf{C}X = \lambda s\ k.\ f\,(\lambda m.\ m\,(s,k))\,s\,k.$$

We can define a lifted version of $\underline{\mathsf{callcc}}$ in terms of $\mathsf{callcc}^{\mathcal{S}}$ in the same manner as equation 9 and obtain:

$$\underline{\mathsf{callcc}}^{\mathcal{S}}\,(f : (X \to \mathcal{S}\mathsf{C}Y) \to \mathcal{S}\mathsf{C}X) : \mathcal{S}\mathsf{C}X = \lambda s\ k.\ f\,(\lambda x\ s'\ \_.\ k\,(x,s))\,s\,k.$$

The author has used the uniform lifting of callcc to verify the ad-hoc liftings of callcc in Haskell's monad transformer library (mtl). This verification revealed that the uniform lifting above coincided with all of the library's liftings, except for one: the library's lifting of callcc through the state monad transformer is not consistent with the rest of the liftings.[2] The ad-hoc lifting of callcc in mtl is:

$$\underline{\mathsf{callcc}\text{-}\mathsf{mtl}}^{\mathcal{S}}\,(f : (X \to \mathcal{S}\mathsf{C}Y) \to \mathcal{S}\mathsf{C}X) : \mathcal{S}\mathsf{C}X = \lambda s\ k.\ f\,(\lambda x\ s'\ \_.\ k\,(x,s'))\,s\,k.$$

The difference is that the ad-hoc lifted operation preserves changes in the state produced during the construction of the new continuation even when the current continuation is used. However, all the other liftings of $\underline{\mathsf{callcc}}$ in the library do not preserve produced effects when using the current continuation.

5 Lifting of Operations

We now show how to lift $\hat{\Sigma}$-operations. To achieve this, we need to refine the definition of monad transformer. All the standard monad transformers fit into this refined definition, except the monad transformer for continuations.

Definition 20 (Functorial Monad Transformer). *The set* **FMT** *of* functorial monad transformers *consists of tuples* $\hat{T} = (T, \mathsf{ret}^T, \mathsf{bind}^T, \mathsf{lift}^T, \mathsf{hmap}^T)$, *where the first four components give a monad transformer (see Def. 9), and*

$$\mathsf{hmap}^T : \forall M, N : * \to *.\ \mathsf{Map}(M) \to \mathsf{Map}(N) \to (M \xrightarrow{\bullet} N) \to (TM \xrightarrow{\bullet} TN)$$

is a term such that for all monads $\hat{M}$, $\hat{N}$ *and* $\hat{P}$,

- hmap^T *preserves natural transformations and monad morphisms, i.e.*
  - $\tau : \mathbf{Nat}(\hat{M}, \hat{N})$ *implies* $\mathsf{hmap}^T_{\hat{M},\hat{N}}\,\tau : \mathbf{Nat}(\hat{T}\hat{M}, \hat{T}\hat{N})$
  - $\xi : \mathbf{MM}(\hat{M}, \hat{N})$ *implies* $\mathsf{hmap}^T_{\hat{M},\hat{N}}\,\xi : \mathbf{MM}(\hat{T}\hat{M}, \hat{T}\hat{N})$
- hmap^T *respects identities and composition of natural transformations, i.e.*
  - $\mathsf{hmap}^T_{\hat{M},\hat{M}}\,\iota_M = \iota_{TM}$

[2] In another monad transformer library by Iavor S. Diatchki, called `MonadLib`, all the liftings correspond to the uniform lifting obtained above.

- $\tau : \mathbf{Nat}(\hat{M}, \hat{N})$ *and* $\sigma : \mathbf{Nat}(\hat{N}, \hat{P})$ *imply*

$$(\mathsf{hmap}^T_{\hat{N},\hat{P}}\sigma) \circ (\mathsf{hmap}^T_{\hat{M},\hat{N}}\tau) = \mathsf{hmap}^T_{\hat{M},\hat{P}}(\sigma \circ \tau)$$

– lift^T *is natural, i.e.*

$$\tau : \mathbf{Nat}(\hat{M}, \hat{N}) \text{ implies } (\mathsf{hmap}^T_{\hat{M},\hat{N}}\tau)_X \cdot \mathsf{lift}^T_{\hat{M},X} = \mathsf{lift}^T_{\hat{N},X} \cdot \tau_X \tag{12}$$

where $\mathsf{hmap}^T_{\hat{M},\hat{N}} \mathrel{\hat{=}} \mathsf{hmap}^T_{M,N}(\mathsf{map}^M, \mathsf{map}^N)$.

Example 21. The monad transformer $\hat{\mathcal{S}}$ becomes functorial with $\mathsf{hmap}^{\mathcal{S}}$ given by

$$\mathsf{hmap}^{\mathcal{S}}_{\hat{F},\hat{G}}(\tau : F \xrightarrow{\bullet} G)(X : *)(t : \mathcal{S}FX) : \mathcal{S}GX \mathrel{\hat{=}} \lambda s : S.\ \tau(t\, s)$$

Some tedious calculations show that it satisfies all the required properties. □

Example 22. The monad transformer $\hat{\mathcal{X}}$ becomes functorial with $\mathsf{hmap}^{\mathcal{X}}$ given by

$$\mathsf{hmap}^{\mathcal{X}}_{\hat{F},\hat{G}}(\tau : F \xrightarrow{\bullet} G)(X : *)(t : \mathcal{X}FX) : \mathcal{X}GX \mathrel{\hat{=}} \tau(t)$$

□

In order to lift $\hat{\Sigma}$-operations we will exploit impredicative polymorphism of system $F\omega$ to define a monad transformer $\mathcal{K}$ (which is not functorial) such that every $\hat{\Sigma}$-operation op for $\hat{M}$ induces an algebraic $\hat{\Sigma}$-operation $\mathsf{op}^{\mathcal{K}}$ for $\hat{\mathcal{K}}\hat{M}$, and op can be recovered from $\mathsf{op}^{\mathcal{K}}$ by pre- and post-composition of $\mathsf{op}^{\mathcal{K}}$ with two natural transformations. The unique algebraic lifting allows to lift $\mathsf{op}^{\mathcal{K}}$ through any monad transformer $\hat{T}$, and obtain an algebraic $\hat{\Sigma}$-operation $\mathsf{op}^{\mathcal{K},T}$ for $\hat{T}(\hat{\mathcal{K}}\hat{M})$. Finally, when $\hat{T}$ is functorial, one recovers from $\mathsf{op}^{\mathcal{K},T}$ a lifting of op through $\hat{T}$, in the same way as one recovers op from $\mathsf{op}^{\mathcal{K}}$.

Definition 23 (Codensity). $\hat{\mathcal{K}}$ *is the monad transformer* $(\mathcal{K}, \mathsf{ret}^{\mathcal{K}}, \mathsf{bind}^{\mathcal{K}}, \mathsf{lift}^{\mathcal{K}})$ *such that for every monad* $\hat{M}$

$$\mathcal{K}MX \mathrel{\hat{=}} \forall Y : *.\ (X \to MY) \to MY$$

$$\mathsf{ret}^{\mathcal{K}}_{\hat{M},X}\,(x : X) : \mathcal{K}MX \mathrel{\hat{=}} \Lambda Y : *.\lambda k : X \to MY.\ k\, x$$

$$\mathsf{bind}^{\mathcal{K}}_{\hat{M},X,Y}(c : \mathcal{K}MX, f : X \to \mathcal{K}MY) : \mathcal{K}MY \mathrel{\hat{=}}$$

$$\Lambda Z : *.\ \lambda k : Y \to MZ.\ c_Z\,(\lambda x : X.\ (f\, x)_Z\, k)$$

$$\mathsf{lift}^{\mathcal{K}}_{\hat{M},X}(m : MX) : \mathcal{K}MX \mathrel{\hat{=}} \Lambda Y : *.\lambda k : X \to MY.\mathsf{bind}^{M}_{X,Y}(m, k)$$

Remark 24. The monad transformer $\hat{\mathcal{K}}$ is related to the construction of the codensity monad for an endofunctor (see [12]). In what follows, we use only some properties of $\hat{\mathcal{K}}$, which are provable by simple calculations in system $F\omega$. Thus, we do not exploit in full the universal property of the codensity monad.

Definition 25. *Let $\hat{M}$ be a monad. Then, we define the terms*

$$\kappa(\tau : \Sigma \cdot M \xrightarrow{\bullet} M) : \Sigma \xrightarrow{\bullet} \mathcal{K}M \mathrel{\hat{=}} \Lambda X : *.\ \lambda s : \Sigma X.\ \Lambda Y : *.\ \lambda k : X \to MY.\ \tau_Y(\mathsf{map}^{\Sigma}\, k\, s)$$

$$\mathsf{from}_{\hat{M}} : \mathcal{K}M \xrightarrow{\bullet} M \mathrel{\hat{=}} \Lambda X : *.\ \lambda c : \mathcal{K}MX.\ c_X\,(\mathsf{ret}^M_X)$$

and for every $\hat{\Sigma}$-operation op *for $\hat{M}$ we define*

$$\mathsf{op}^{\mathcal{K}} : \Sigma \cdot \mathcal{K}M \xrightarrow{\bullet} \mathcal{K}M \mathrel{\hat{=}} \psi(\kappa\ \mathsf{op})$$

where ψ is defined in Prop. 17.

Proposition 26. *Given a monad $\hat{M}$ and a $\hat{\Sigma}$-operation* op *for $\hat{M}$, then*

a) $\mathsf{from}_{\hat{M}}$ *is a natural transformation from $\hat{\mathcal{K}}\hat{M}$ to $\hat{M}$ such that*

$$\iota_M = \mathsf{from}_{\hat{M}} \circ \mathsf{lift}^{\mathcal{K}}_{\hat{M}}$$

b) $\mathsf{op}^{\mathcal{K}}$ *is an algebraic $\hat{\Sigma}$-operation for $\hat{\mathcal{K}}\hat{M}$ such that*

$$\mathsf{op} = \mathsf{from}_{\hat{M}} \circ \mathsf{op}^{\mathcal{K}} \circ (\hat{\Sigma}\ \mathsf{lift}^{\mathcal{K}}_{\hat{M}}) \tag{13}$$

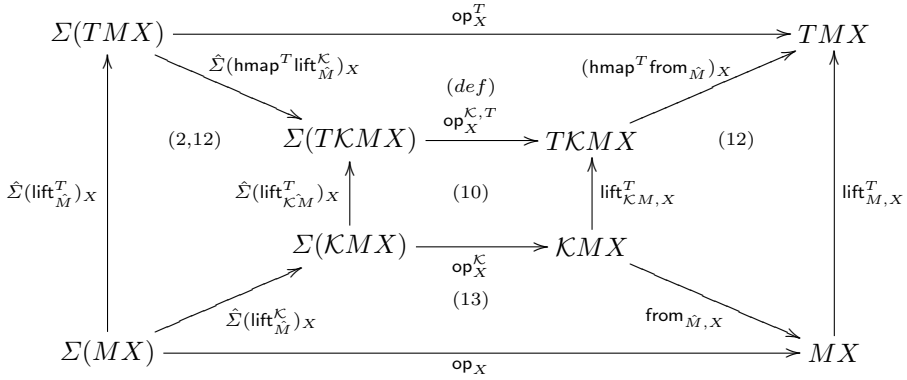
where ι and $\circ$ are the identity and composition of natural transformations, and $\hat{\Sigma}$ is the application of a functor to a natural transformation (see Definition 3).

Theorem 27 (Lifting). *Given a $\hat{\Sigma}$-operation* op *for a monad $\hat{M}$ and a functorial monad transformer $\hat{T}$, let $\mathsf{op}^T : \Sigma \cdot (TM) \xrightarrow{\bullet} TM$ be the term*

$$\mathsf{op}^T = (\mathsf{hmap}^T_{\hat{\mathcal{K}}\hat{M},\hat{M}}\mathsf{from}_{\hat{M}}) \circ \mathsf{op}^{\mathcal{K},T} \circ (\hat{\Sigma}(\mathsf{hmap}^T_{\hat{M},\hat{\mathcal{K}}\hat{M}}\mathsf{lift}^{\mathcal{K}}_{\hat{M}})) \tag{14}$$

where $\mathsf{op}^{\mathcal{K},T}$ is the algebraic lifting of $\mathsf{op}^{\mathcal{K}}$ through $\hat{T}$, then op^T is a lifting of op *through $\hat{T}$.*

Proof. The following diagram commutes:



□

When op is an algebraic $\hat{\Sigma}$-operation for $\hat{M}$, there is a simpler way to lift op through $\hat{T}$. The following result says that when both liftings are defined, they yield the same result.

Proposition 28. *If* op *is an algebraic* $\hat{\Sigma}$*-operation for* $\hat{M}$ *and* $\hat{T}$ *a functorial monad transformer, then the algebraic lifting of* op *along* $\mathsf{lift}^T_{\hat{M}}$ *given by Theorem 19 coincides with the lifting of* op^T *given by Theorem 27.*

Example 29. We specialize the lifting in Theorem 27 to several concrete functorial monad transformers and an arbitrary $\hat{\Sigma}$-operation op for a monad $\hat{M}$.

- When $\hat{T} = \hat{\mathcal{S}}$, thus $\mathcal{S}MX = S \to M(X \times S)$, the lifting simplifies to:

$$\mathsf{op}^{\mathcal{S}}_X\,(t\colon \Sigma(\mathcal{S}MX)) : \mathcal{S}MX = \lambda s.\, \mathsf{op}_{X\times S}(\mathsf{map}^{\Sigma}\,\tau^s\,t)$$

  where $\tau^s(f\colon S \to M(X\times S)) = f\,s$.
- When $\hat{T} = \hat{\mathcal{X}}$, thus $\mathcal{X}MX = M(Z + X)$, the lifting simplifies to:

$$\mathsf{op}^{\mathcal{X}}_X(t\colon \Sigma(\mathcal{X}MX)) : \mathcal{X}MX = \mathsf{op}_{Z+X}\,t.$$

- When $\hat{T}$ is $\hat{\mathcal{R}}$, the functorial monad transformer for environments of type E [10], thus $\mathcal{R}MX = E \to MX$, the lifting simplifies to:

$$\mathsf{op}^{\mathcal{R}}_X\,(t\colon \Sigma(\mathcal{R}MX)) : \mathcal{R}MX = \lambda e.\, \mathsf{op}_X(\mathsf{map}^{\Sigma}\,\tau^e\,t)$$

  where $\tau^e(f\colon E \to MX) = f\,e$.
- When $\hat{T}$ is $\hat{\mathcal{O}}$, the functorial monad transformer for output of type $[A]$ [10], thus $\mathcal{O}MX = M(X \times [A])$, and the lifting simplifies to:

$$\mathsf{op}^{\mathcal{O}}_X(t\colon \Sigma(\mathcal{O}MX)) : \mathcal{O}MX = \mathsf{op}_{X\times[A]}\,t.$$

□

The example above shows that Theorem 27 subsumes the incremental approach in [15, 3]. In the following, we apply the lifting theorem to the remaining non-algebraic operations local, handle, and flush. Because of Proposition 28, for algebraic operations it makes more sense to use the simpler algebraic lifting.

Example 30. The monad for environments of type E and its operations for reading the environment and performing a computation in a modified environment are shown below.

$$\begin{aligned}
\mathsf{R}(X\colon *) &\mathrel{\hat{=}} E \to X\\
\mathsf{ret}^{\mathsf{R}}\,(x\colon X)\colon \mathsf{R}\,X &\mathrel{\hat{=}} \lambda_.\, x\\
\mathsf{bind}^{\mathsf{R}}(m\colon \mathsf{R}\,X, f\colon X \to \mathsf{R}\,Y)\colon \mathsf{R}\,Y &\mathrel{\hat{=}} \lambda e.\, f\,(m\,e)\,e\\
\mathsf{ask}\,(f\colon E \to \mathsf{R}X)\colon \mathsf{R}X &\mathrel{\hat{=}} \lambda e.\, f\ e\ e\\
\mathsf{local}\,(f\colon E \to E, m\colon \mathsf{R}X)\colon \mathsf{R}X &\mathrel{\hat{=}} \lambda e.\, m\,(f\,e)
\end{aligned}$$

Applying Theorem 27 to the non-algebraic, $\hat{\Sigma}$-operation local we obtain the following lifted operation for any functorial monad transformer $\hat{T}$:

$$\mathsf{local}^{T}\,(f : E \to E, t : T\mathsf{R}X) : T\mathsf{R}X \mathrel{\hat{=}} \mathsf{hmap}^{T}_{\hat{\mathcal{K}}\hat{\mathsf{R}},\hat{\mathsf{R}}}\ \mathsf{from}_{\hat{\mathsf{R}}}\ (\mathsf{local}^{\mathcal{K},T}(f, t'))$$
$$\text{where} \qquad t' : T\mathcal{K}\mathsf{R}X \mathrel{\hat{=}} \mathsf{hmap}^{T}_{\hat{\mathsf{R}},\hat{\mathcal{K}}\hat{\mathsf{R}}}\ \mathsf{lift}^{\mathcal{K}}_{\hat{\mathsf{R}}}\ t$$
$$\mathsf{local}^{\mathcal{K},T}\,(f : E \to E, t : T\mathcal{K}\mathsf{R}X) : T\mathcal{K}\mathsf{R}X \mathrel{\hat{=}} \mathsf{join}^{T}_{\hat{\mathcal{K}}\hat{\mathsf{R}}}\,(\mathsf{lift}^{T}_{\hat{\mathcal{K}}\hat{\mathsf{R}}}\,(\Lambda Y.\,\lambda k.\,\mathsf{local}\,(f, k\,t)))$$

– When $\hat{T} = \hat{\mathcal{S}}$, thus $\mathcal{S}\mathsf{R}X = S \to E \to (X \times S)$, the lifting simplifies to:

$$\mathsf{local}^{\mathcal{S}}\,(f : E \to E, t : \mathcal{S}\mathsf{R}X) : \mathcal{S}\mathsf{R}X = \lambda s\ e.\,t\ s\,(f\ e).$$

– When $\hat{T} = \hat{\mathcal{X}}$, thus $\mathcal{X}\mathsf{R}X = E \to (Z + X)$, the lifting simplifies to:

$$\mathsf{local}^{\mathcal{X}}\,(f : E \to E, t : \mathcal{X}\mathsf{R}X) : \mathcal{X}\mathsf{R}X = \lambda e.\,t\,(f\ e).$$

– When $\hat{T} = \hat{\mathcal{R}}$, thus $\mathcal{R}\mathsf{R}X = E \to E \to X$, the lifting simplifies to:

$$\mathsf{local}^{\mathcal{R}}\,(f : E \to E, t : \mathcal{R}\mathsf{R}X) : \mathcal{R}\mathsf{R}X = \lambda e\ e'.\,t\ e\ (f\ e')$$

– When $\hat{T} = \hat{\mathcal{O}}$, thus $\mathcal{O}MX = E \to (X \times [A])$, the lifting simplifies to:

$$\mathsf{local}^{\mathcal{O}}\,(f : E \to E, t : \mathcal{O}\mathsf{R}X) : \mathcal{O}\mathsf{R}X = \lambda e.\,t\,(f\ e)$$

□

Note that we can arrive at the concrete liftings above—where both $\hat{T}$ and op are fixed—by either Example 29 (where we first fix $\hat{T}$) or the definition of local^{T} above (where we first fix op), but only by fixing the monad transformer we get a significant simplification of the lifting.

Example 31. The monad for exceptions of type Z and its operations for throwing and handling exceptions are shown below.

$$\mathsf{X}(X : *) \mathrel{\hat{=}} Z + X$$
$$\mathsf{ret}^{\mathsf{X}}(x : X) : \mathsf{X}\,X \mathrel{\hat{=}} \mathsf{inr}\,x$$
$$\mathsf{bind}^{\mathsf{X}}(m : \mathsf{X}\,X, f : X \to \mathsf{X}\,Y) : \mathsf{X}\,Y \mathrel{\hat{=}} \mathsf{case}\ m\ \mathsf{of}\ |\ \mathsf{inl}\ z \Rightarrow \mathsf{inl}\ z\ |\ \mathsf{inr}\ x \Rightarrow f\ x$$
$$\mathsf{throw}\,(z : Z) : \mathsf{X}\,X \mathrel{\hat{=}} \mathsf{inl}\ z$$
$$\mathsf{handle}\,(m : \mathsf{X}\,X, h : Z \to \mathsf{X}\,X) : \mathsf{X}\,X \mathrel{\hat{=}} \mathsf{case}\ m\ \mathsf{of}\ |\ \mathsf{inl}\ z \Rightarrow h\ z\ |\ \mathsf{inr}\ x \Rightarrow \mathsf{inr}\ x$$

We obtain the following liftings for the non-algebraic $\hat{\Sigma}$-operation handle.

– When $\hat{T} = \hat{\mathcal{S}}$, thus $\mathcal{S}\mathsf{X}X = S \to Z + (X \times S)$, the lifting is:

$$\mathsf{handle}^{\mathcal{S}}\,(t : \mathcal{S}\mathsf{X}X, h : Z \to \mathcal{S}\mathsf{X}X) : \mathcal{S}\mathsf{X}X = \lambda s.\,\mathsf{case}\ t\ s\ \mathsf{of}\ |\ \mathsf{inl}\,z \Rightarrow h\,z\,s$$
$$|\ \mathsf{inr}\,x \Rightarrow \mathsf{inr}\,x$$

– When $\hat{T} = \hat{\mathcal{X}}$, thus $\mathcal{X}\mathsf{X}X = Z + (Z + X)$, the lifting is:

$$\mathsf{handle}^{\mathcal{X}}(t\colon \mathcal{X}\mathsf{X}X, h\colon Z \to \mathcal{X}\mathsf{X}X)\colon \mathcal{X}\mathsf{X}X = \mathsf{case}\ t\ \mathsf{of}\ \mid \mathsf{inl}\, z \Rightarrow h\, z \mid \mathsf{inr}\, x \Rightarrow \mathsf{inr}\, x.$$

– When $\hat{T} = \hat{\mathcal{R}}$, thus $\mathcal{R}\mathsf{X}X = E \to (Z + X)$, the lifting is:

$$\mathsf{handle}^{\mathcal{R}}\,(t\colon \mathcal{R}\mathsf{X}X, h\colon Z \to \mathcal{R}\mathsf{X}X)\colon \mathcal{R}\mathsf{X}X = \lambda e.\, \mathsf{case}\ t\, e\ \mathsf{of}\ \mid \mathsf{inl}\, z \Rightarrow h\, z\, e \mid \mathsf{inr}\, x \Rightarrow \mathsf{inr}\, x$$

– When $\hat{T} = \hat{\mathcal{O}}$, thus $\mathcal{O}\mathsf{X}X = Z + (X \times [A])$, the lifting is:

$$\mathsf{handle}^{\mathcal{O}}(t\colon \mathcal{O}\mathsf{X}X, h\colon Z \to \mathcal{O}\mathsf{X}X)\colon \mathcal{O}\mathsf{X}X = \mathsf{case}\ t\ \mathsf{of}\ \mid \mathsf{inl}\, z \Rightarrow h\, z \mid \mathsf{inr}\, x \Rightarrow \mathsf{inr}\, x.$$

□

Example 32. The monad for output of a type $[A]$ and its operations for outputting a list, and flushing the output are shown below.

$$\begin{aligned}
\mathsf{O}\,(X\colon *) &\mathrel{\hat{=}} X \times [A] \\
\mathsf{ret}^{\mathsf{O}}\,(x\colon X)\colon \mathsf{O}X &\mathrel{\hat{=}} (x, \mathsf{empty}(\bullet)) \\
\mathsf{bind}^{\mathsf{O}}(m\colon \mathsf{O}X, f\colon X \to \mathsf{O}Y)\colon \mathsf{O}X &\mathrel{\hat{=}} \mathsf{let}\ (x, w) = m\ \mathsf{in} \\
&\qquad \mathsf{let}\ (x', w') = f\, x\ \mathsf{in}\ (x', \mathsf{append}(w, w')) \\
\mathsf{output}\,(\,(w, m)\colon W \times \mathsf{O}X)\colon \mathsf{O}X &\mathrel{\hat{=}} \mathsf{let}\ (x, w') = m\ \mathsf{in}\ (x, \mathsf{append}(w', w)) \\
\mathsf{flush}\,(m\colon \mathsf{O}X)\colon \mathsf{O}X &\mathrel{\hat{=}} \mathsf{let}\ (x, _) = m\ \mathsf{in}\ (x, \mathsf{empty}(\bullet))
\end{aligned}$$

where $\mathsf{empty}(\bullet)$ is the empty list, and append appends two lists. We obtain the following liftings for the non-algebraic $\hat{\Sigma}$-operation flush.

– When $\hat{T} = \hat{\mathcal{S}}$, thus $\mathcal{S}\mathsf{O}X = S \to ((X \times S) \times [A])$, the lifting is:

$$\mathsf{flush}^{\mathcal{S}}\,(t\colon \mathcal{S}\mathsf{O}X)\colon \mathcal{S}\mathsf{O}X = \lambda s.\, \mathsf{let}\ (x, \_) = t\, s\ \mathsf{in}\ (x, \mathsf{empty}(\bullet))$$

– When $\hat{T} = \hat{\mathcal{X}}$, thus $\mathcal{X}\mathsf{O}X = (Z + X) \times [A]$, the lifting is:

$$\mathsf{flush}^{\mathcal{X}}\,((c, w)\colon \mathcal{X}\mathsf{O}X, h\colon Z \to \mathcal{X}\mathsf{O}X)\colon \mathcal{X}\mathsf{O}X = (c, \mathsf{empty}(\bullet))$$

– When $\hat{T} = \hat{\mathcal{R}}$, thus $\mathcal{R}\mathsf{O}X = E \to (X \times [A])$, the lifting is:

$$\mathsf{flush}^{\mathcal{R}}\,(t\colon \mathcal{R}\mathsf{O}X)\colon \mathcal{R}\mathsf{O}X = \lambda e.\, \mathsf{let}\ (x, \_) = t\, e\ \mathsf{in}\ (x, \mathsf{empty}(\bullet))$$

– When $\hat{T} = \hat{\mathcal{O}}$, thus $\mathcal{O}\mathsf{O}X = (X \times [A]) \times [A]$, the lifting is:

$$\mathsf{flush}^{\mathcal{O}}\,((p, w)\colon \mathcal{O}\mathsf{O}X, h\colon Z \to \mathcal{O}\mathsf{O}X)\colon \mathcal{O}\mathsf{O}X = (p, \mathsf{empty}(\bullet))$$

□

6 Conclusion

Monad transformers allow programmers to modularly construct a monad, but for their potential to be fully realized, the lifting of operations should also be modular. We have defined a uniform lifting through any monad transformer with a *functorial behaviour*. This lifting is applicable to a wide class of operations which includes all operations considered in [10] and all the operations in Haskell's mtl, except for listen. Through several examples, we have given evidence that our uniform lifting subsumes the more or less ad-hoc definitions of lifting that could be found in the literature.

Our initial focus on algebraic operations is inspired by Plotkin et al. [7], where a monad is constructed from an algebraic theory presented by algebraic operations and equations, and combined monads are obtained by combination of theories. This approach is appealing, but it can cope only with monads corresponding to algebraic theories and with algebraic operations.

The current design of monad transformer libraries is based on the traditional approach to operation lifting which has other problems besides non-modularity. The experimental library Monatron [8] implements a new design which not only lifts operations uniformly, but also avoids many of these problems.

There are several possible directions for further research:

- The lifting of $\hat{\Sigma}$-operations assumes functorial monad transformers. In order to accomodate the continuation monad transformer, we plan to extend the results in the article to mixed-variant functorial monad transformers.
- Instead of assuming an operation $\Sigma \cdot M \xrightarrow{\bullet} M$, we can consider operations $HM \xrightarrow{\bullet} M$, where H is a functor in an endofunctor category. This allows us to model the mtl operation listen and obtain a lifting for it. However, in general, obtaining a lifting seems to depend on the operation inducing an algebraic $\hat{\Sigma}$-operation for another monad. General techniques for finding such a lifting need to be investigated.
- Given a $\hat{\Sigma}$-operation for $\hat{M}$, we can obtain its lifting through any functorial monad transformer. However, its general formulation is rather involved, and we would like to obtain a simpler lifting (perhaps under certain extra assumptions, as in Proposition 28).

Since the traditional non-modular solution for lifting operations through monad transformers was introduced, there has been little progress in this area. We hope that the new approach developed in this article leads to new and exciting ways of designing structured effectful functional programs.

Acknowledgments. I would like to thank Nils Anders Danielsson, Neil Ghani, Graham Hutton, Peter Morris, Wouter Swierstra, and the anonoymous referees for their detailed and insightful comments. Finally, I would like to specially thank Eugenio Moggi for his generous assistance in significantly improving this article.

References

1. Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.
2. M. Barr and C. Wells. *Toposes, Triples and Theories*, volume 278 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, New York, 1985.
3. N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *International Summer School On Applied Semantics APPSEM2000*, pages 42–122. Springer-Verlag, 2000.
4. Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
5. Neil Ghani. Eta-expansions in F-omega. In *Proceedings of CSL'96*, number 1258 in Lecture Notes in Computer Science, pages 182–197. Springer-Verlag, 1996.
6. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL*, pages 341–354, 1990.
7. Martin Hyland, Gordon D. Plotkin, and John Power. Combining effects: Sum and tensor. *Theor. Comput. Sci.*, 357(1-3):70–99, 2006.
8. Mauro Jaskelioff. Monatron: an extensible monad transformer library. Available at `http://www.cs.nott.ac.uk/~mjj/pubs/monatron.pdf`. Submitted for publication., 2008.
9. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL*, pages 71–84, 1993.
10. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL*, pages 333–343, 1995.
11. Cristoph Lüth and Neil Ghani. Composing monads using coproducts. In *ICFP*, pages 133–144, 2002.
12. Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971. Second edition, 1998.
13. Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Computer Society, 1989.
14. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
15. Eugenio Moggi. Metalanguages and applications. In *Semantics and Logics of Computation*, Publications of the Newton Institute. CUP, 1997.
16. Benjamin C. Pierce. *Basic Category Theory for Computer Scientists (Foundations of Computing)*. The MIT Press, August 1991.
17. Gordon D. Plotkin and John Power. Semantics for algebraic operations. *ENTCS*, 45, 2001.
18. John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Inf. Comput.*, 105(1):1–29, 1993.
19. Philip Wadler. Comprehending monads. *MSCS*, 2(4):461–493, 1992.
20. Philip Wadler. The essence of functional programming. In *POPL*, pages 1–14, 1992.